The Design and Implementation of the ULIX Operating System

Hans-Georg Eßer
Felix C. Freiling
The Design and Implementation of the ULIX Operating System

Hans-Georg Eßer, Felix C. Freiling
Foreword

Welcome to the ULIX book. If you want to discover how precisely a modern operating system performs its multitasking magic, you’ve probably found the right textbook. On the next 600 pages you can read about the implementation of ULIX, a Unix-like system, and while you read the chapters which are organized along similar topics as those of most operating system textbooks, you will see the full implementation. We left out nothing, the whole code is there (actually: the book is the code, but more about that later), and you can read it step by step with each implementation part shown where it makes sense—which is very different from just providing a collection of source code files.

If you’re already familiar with other operating system texts, you will notice that this book tends to be more practical. We often discuss only one solution to a problem (where other books mention a variety and sometimes give a qualified comparison of several techniques). We do not completely ignore alternative approaches, but since we give helpful explanations of all the code parts, we tend to write more about our solutions than about the ones which we did not choose for the ULIX kernel.

Writing this book was a fun experience, because for us, the authors, it meant learning a lot as we went along with the implementation and documentation task. While previous exposure to theoretical books on operating system principles was certainly helpful for deciding what to implement, it did not help a lot with the question of how to do it. Many technical references and some online tutorials for OS development beginners were useful (and much needed) in order to overcome the numerous challenges which the ULIX development encompassed. We hope that you will find reading our book as pleasant and instructive as we found writing it, and we encourage you to try your own experiments with kernel development: The book provides several exercises in which you can modify or add to the ULIX code.

Our overall hope is that you will find this book helpful and that it takes you on a journey that intensifies your knowledge of OS internals and the necessary hardware-related details. After reading, let us know if we’ve met your expectations (→ feedback@ulixos.org).

Hans-Georg Eßer
Felix C. Freiling

Erlangen, September 2015
This page intentionally left blank
# Table of Contents

**Foreword**  
1

**List of Figures**  
13

**List of Tables**  
16

1 **Introduction**  
1.1 Literate Programming .................................................. 18 
1.2 The ULIX Operating System .............................................. 20 
  1.2.1 The Name of the Game ........................................... 21 
  1.2.2 Design Principles of ULIX ....................................... 21 
  1.2.3 ULIX Features .................................................... 21 
1.3 Tools ........................................................................... 22 
1.4 Copying ........................................................................ 24 
1.5 Notation ........................................................................ 24 
  1.5.1 Numbers and Units ................................................ 24 
  1.5.2 Identifiers ............................................................. 26 
  1.5.3 Margin Notes ......................................................... 26 
  1.5.4 Code Chunks .......................................................... 26 
  1.5.5 Coding Standard .................................................... 28 
  1.5.6 “Going where?” ....................................................... 29 
1.6 Creating an Operating System From Scratch ...................... 30 
  1.6.1 Selection of Target Hardware ................................... 30 
  1.6.2 Language of Choice ................................................. 30 
  1.6.3 Applications .......................................................... 31 
1.7 What’s in the Book? ......................................................... 31 
1.8 Helpful Previous Knowledge ............................................ 36 
1.9 Online Resources .......................................................... 36 
1.10 Further Reading ............................................................ 38 
1.11 About the Authors ........................................................ 39
2 Layout of the Kernel Code

2.1 The main() Function
2.2 Type Definitions
2.3 Assembler Code

2.3.1 Turning Interrupts On and Off
2.4 The User Mode Library

2.4.1 Functions of the Library
2.5 Next Steps

3 Managing Memory

3.1 Contiguous Allocation

3.1.1 Fixed Equal Size Partitioning
3.1.2 Fixed Variable Size Partitioning
3.1.3 Dynamic Partitioning
3.1.4 A Few Words on Disk Partitions
3.1.5 Segmentation
3.1.6 Fragmentation

3.2 Non-Contiguous Allocation

3.2.1 Virtual Memory
3.2.2 Address Translation
3.2.3 Virtual Memory Requirements
3.2.4 Page-based Virtual Memory
3.2.5 Page-based Virtual Memory in Ulix

4 Boot Process and Memory Management in Ulix

4.1 GRUB Loads the Ulix Kernel
4.2 The Ulix Memory Layout

4.3 From Real Mode to Protected Mode

4.3.1 Segmentation in Real Mode
4.3.2 Privilege Levels in Protected Mode
4.3.3 Segmentation in Protected Mode
4.3.4 Preparations for Paging

4.4 Virtual Memory for the Kernel

4.4.1 Page Descriptors and Page Table Descriptors in Ulix
4.4.2 Identity Mapping the Kernel Memory
4.4.3 Installing the “Flat” GDT
4.4.4 Accessing the Video RAM

4.5 Physical Memory: Page Frames in Ulix

4.5.1 Bitwise Manipulation
4.5.2 Direct Access to the Physical RAM
4.5.3 PEEK and POKE Functions
4.5.4 Allocating and Releasing Frames

4.6 Managing Pages in Ulix

4.6.1 Allocating Pages
4.6.2 Releasing Pages ............................................. 122
4.7 Next Steps .................................................. 124
4.8 Exercises ..................................................... 124

5 Interrupts and Faults ............................................. 129
5.1 Examples for Interrupt Usage ................................. 130
5.2 Interrupt Handling on the Intel Architecture ............. 130
5.2.1 Using Ports for I/O Requests .............................. 132
5.2.2 Initializing the PIC ......................................... 134
5.2.3 Interrupt Descriptor Table ................................. 136
5.2.4 Writing the Interrupt Handler .............................. 140
5.3 Faults .......................................................... 147
5.4 Exercises ..................................................... 153

6 Implementation of Processes .................................... 157
6.1 Address Spaces for Processes ................................. 158
6.1.1 Memory Layout of a Process .............................. 159
6.1.2 Creating a New Address Space ............................ 160
6.1.3 Destroying an Address Space .............................. 166
6.1.4 Switching between Address Spaces ....................... 169
6.1.5 Enlarging an Address Space ............................... 172
6.2 Thread Control Blocks and Thread Queues ................. 174
6.2.1 Thread State ............................................... 176
6.2.2 Implementing Lists of Threads ............................ 180
6.2.3 Allocating and Initializing a New TCB ................. 187
6.3 Starting the First Process ..................................... 189
6.3.1 Preparations ............................................... 189
6.3.2 Loading the Program ...................................... 190
6.3.3 Creating the Kernel Stack ................................. 191
6.3.4 Activating the New Process ............................... 192
6.3.5 Configuring the TSS Structure and Entering User Mode 193
6.4 System Calls .................................................. 199
6.4.1 System Calls in ULIx ..................................... 200
6.4.2 Making System Calls ...................................... 203
6.4.3 Handling Errors with errno ............................... 205
6.5 Forking a Process ............................................. 207
6.5.1 Reserving Memory and a Fresh TCB ..................... 210
6.5.2 Filling the Child TCB ..................................... 210
6.5.3 The Child’s Kernel Stack ................................. 211
6.5.4 Copying the Process’ User Mode Memory .............. 211
6.5.5 A Child Is Born ............................................ 212
6.5.6 The fork System Call ...................................... 213
6.5.7 Testing fork ................................................ 214
6.6 Exiting from a Process ....................................... 215
6.6.1 The exit System Call ........................................... 216
6.6.2 The waitpid System Call ........................................ 218
6.6.3 Giving Up the CPU: The resign System Call ............. 220

6.7 Information about Processes ...................................... 221
  6.7.1 The gettid, getpid and getppid System Calls ............ 222
  6.7.2 The getpsinfo and setpsname System Calls ............... 223

6.8 ELF Loader .......................................................... 225
  6.8.1 ELF File Format .................................................. 225
  6.8.2 Implementation of the ELF Loader ......................... 228
  6.8.3 User Mode Binaries .............................................. 235

6.9 Exercises ............................................................. 237

7 Implementation of Threads ........................................... 245
  7.1 Threads, Teams of Threads and Virtual Processors .......... 246
    7.1.1 Teams of Threads ............................................ 246
    7.1.2 Natural Concurrency ......................................... 247
    7.1.3 Advantages of Concurrent Programming .................. 248
    7.1.4 Virtual vs. Physical Processors ........................... 249
  7.2 Thread Requirements and Thread Types ....................... 250
    7.2.1 Thread Requirements ......................................... 250
    7.2.2 Utility of Threads ............................................ 250
    7.2.3 Types of Threads ............................................. 252
  7.3 Implementation of Threads in ULIx ............................. 254
    7.3.1 Creating Threads Instead of Processes .................. 254
    7.3.2 System Call for Thread Creation .......................... 258
    7.3.3 Terminating Threads .......................... 259
    7.3.4 Thread Synchronization ............................... 261

8 Scheduling ............................................................. 263
  8.1 Monoprocessor Scheduling ...................................... 263
    8.1.1 Quality Metrics ............................................. 264
    8.1.2 Preemptive vs. Non-preemptive Scheduling .............. 264
    8.1.3 First-Come First-Served .................................... 265
    8.1.4 Shortest Job First .......................................... 265
    8.1.5 Round Robin .................................................. 266
    8.1.6 Virtual Round Robin ......................................... 268
    8.1.7 Priority-Based Scheduling .................................. 269
    8.1.8 Multi-Level Scheduling ..................................... 270
  8.2 Multiprocessor Scheduling ....................................... 271
  8.3 Implementation of the ULIx Scheduler ......................... 272
    8.3.1 Stack Usage .................................................. 273
    8.3.2 The Implementation .......................................... 275
    8.3.3 Letting the init Process Idle .............................. 281
  8.4 Exercises .......................................................... 283
9 Handling Page Faults
  9.1 The Ulix Page Fault Handler ................................................. 288
    9.1.1 Enlarging the Stack .................................................. 291
  9.2 The Swap File ................................................................. 292
    9.2.1 Paging Out and In ..................................................... 295
    9.2.2 Letting the Page Fault Handler Page In a Page .................. 298
    9.2.3 Testing ................................................................. 298
  9.3 Page Replacement Strategy .................................................. 300
    9.3.1 Page Locking ............................................................ 301
    9.3.2 Page Replacement Strategies ........................................ 302
  9.4 Page Replacement Implementation in Ulix .................................. 304
    9.4.1 The Swapper Process .................................................. 309
10 Talking to the Hardware ......................................................... 313
  10.1 Keyboard ................................................................. 313
    10.1.1 Scan Code Tables ...................................................... 313
    10.1.2 Virtual Consoles ...................................................... 317
    10.1.3 Keyboard Interrupt Handler ........................................ 319
    10.1.4 The Keyboard Queue .................................................. 323
  10.2 Terminals ................................................................. 325
    10.2.1 Terminal Output ....................................................... 335
    10.2.2 Status Line Management .............................................. 337
    10.2.3 Initializing the Screen ............................................... 337
  10.3 System Timer ............................................................... 338
    10.3.1 Setting the Frequency ............................................... 338
    10.3.2 Reading the Date and Time ......................................... 339
    10.3.3 Implementation of the Timer Handler ............................. 342
    10.3.4 Tasks for the Timer .................................................. 342
  10.4 Serial Ports ............................................................... 343
11 Synchronization ............................................................... 347
  11.1 Critical Sections .......................................................... 348
    11.1.1 The Case of the Lost List Element .................................. 348
    11.1.2 Defining Critical Sections ......................................... 349
  11.2 Hardware-based Synchronization .......................................... 352
    11.2.1 Disabling Interrupts ................................................ 352
    11.2.2 Using Special Hardware Instructions ............................ 353
    11.2.3 Monoprocessor vs. Multiprocessor Synchronization .......... 355
    11.2.4 Nested Critical Sections .......................................... 356
  11.3 Semaphores ............................................................... 358
    11.3.1 Semantics of Semaphores ......................................... 359
    11.3.2 Single Mutual Exclusion ......................................... 359
    11.3.3 Initialization of Semaphores .................................... 360
    11.3.4 Implementing P and V ................................................. 361
## Contents

### 11.3.5 User-Level Semaphores

### 11.3.6 Semaphores in Unix

### 11.4 Unix Locks

#### 11.4.1 Locking, Unlocking and Just Wishing

### 11.5 Pthread Mutexes for Threads

#### 11.5.1 Creating a New Mutex

#### 11.5.2 Locking and Unlocking a Mutex

#### 11.5.3 Destroying a Mutex

#### 11.5.4 The Library Functions

#### 11.5.5 Testing

### 11.6 Kernel Synchronization

#### 11.6.1 Overview

#### 11.6.2 Minimizing the Size of Critical Sections: Interruptible Kernels

#### 11.6.3 Unix as a Non-Interruptible Kernel

#### 11.6.4 Illustrating Problems of Interruptible Kernels

### 11.7 Further Reading

### 11.8 Exercises

### 12 Filesystems

#### 12.1 Introduction to Filesystems

##### 12.1.1 Simple Filesystems

##### 12.1.2 Advanced Filesystems

#### 12.2 Mounting: the Unix Way to Access Many Volumes

#### 12.3 The Unix Virtual Filesystem

##### 12.3.1 Finding the Device and Local Path

##### 12.3.2 Constants for Filesystems

##### 12.3.3 Global File Descriptors

##### 12.3.4 Opening a File

##### 12.3.5 Reading, Writing and Other Operations

##### 12.3.6 Detect Terminals

##### 12.3.7 Status

##### 12.3.8 Directories

#### 12.4 System Calls for File Access

##### 12.4.1 Library Functions

##### 12.4.2 Reading from Standard Input

##### 12.4.3 Working Directory, Relative Paths

#### 12.5 The Minix Filesystem

##### 12.5.1 The Minix Superblock

##### 12.5.2 Zone and Inode Bitmaps

##### 12.5.3 Reading and Writing Inodes

##### 12.5.4 Directory Entries

##### 12.5.5 The \texttt{i_mode} Entry of the Inode

##### 12.5.6 Opening and Closing Files
Contents

12.6.7 Reading and Writing ........................................... 469
12.6.8 Linking and Unlinking ......................................... 479
12.6.9 Truncating Files .................................................. 484
12.6.10 Making and Removing Directories ............................... 486
12.6.11 Listing a Directory .............................................. 489
12.6.12 Filesystem Information: df .................................... 491
12.7 The /dev Filesystem .................................................. 494
12.8 Default Contents of the Filesystem ................................. 500
12.9 Further Reading ...................................................... 501
12.10 Exercises .............................................................. 501

13 Disk I/O ........................................................................... 503
13.1 Block and Character Devices .......................................... 503
13.2 Device Selection .......................................................... 504
13.3 A Simple Buffer Cache ................................................ 508
13.4 Serial Hard Disk .......................................................... 514
13.4.1 Kernel Code for the Serial Disk ................................. 515
13.4.2 The External Controller Process ................................. 522
13.5 The Hard Disk Controller .............................................. 525
13.5.1 Sending Commands to the Controller .......................... 525
13.5.2 The Blocked Queue ................................................ 529
13.5.3 Reading and Writing ............................................... 529
13.5.4 Interrupt Handler .................................................. 532
13.5.5 Hard Disk Initialization .......................................... 533
13.6 The Floppy Controller .................................................. 535
13.6.1 Talking to the Controller .......................................... 535
13.6.2 Setting Up the DMA Transfer ................................... 538
13.6.3 Starting and Stopping the Motor ............................... 544
13.6.4 Handling Floppy Interrupts ...................................... 544
13.6.5 Reading and Writing ............................................... 548
13.6.6 Resetting and Recalibrating ...................................... 550
13.6.7 Floppy Driver Initialization ...................................... 552

14 Signals ........................................................................... 555
14.1 Use Cases for Signals .................................................. 556
14.2 Signals in Classical Unix Systems .................................... 557
14.3 Implementation of Signals in ULIx ................................. 559
14.3.1 Library Functions .................................................. 568
14.3.2 Example Program .................................................. 568

15 Users and Groups .......................................................... 571
15.1 Users and Groups in ULIx ............................................. 573
15.1.1 Checking Permissions ............................................. 574
15.1.2 Changing User and Group IDs ................................. 580
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1.3 The su Program</td>
<td>586</td>
</tr>
<tr>
<td>15.1.4 The getuid() and getgid() Functions</td>
<td>586</td>
</tr>
<tr>
<td>15.1.5 Changing Owner, Group and Permissions</td>
<td>587</td>
</tr>
<tr>
<td>15.2 Exercises</td>
<td>591</td>
</tr>
<tr>
<td>16 Small Standard Library</td>
<td>593</td>
</tr>
<tr>
<td>16.1 Strings</td>
<td>593</td>
</tr>
<tr>
<td>16.2 Memory</td>
<td>596</td>
</tr>
<tr>
<td>16.3 Formatted Output</td>
<td>597</td>
</tr>
<tr>
<td>16.3.1 printf in the Kernel</td>
<td>597</td>
</tr>
<tr>
<td>16.3.2 printf in the User Mode Library</td>
<td>598</td>
</tr>
<tr>
<td>16.3.3 The Generic Implementation</td>
<td>599</td>
</tr>
<tr>
<td>17 Debugging Help</td>
<td>603</td>
</tr>
<tr>
<td>17.1 The Kernel Mode Shell</td>
<td>603</td>
</tr>
<tr>
<td>17.2 A System Call That Displays an Inode</td>
<td>610</td>
</tr>
<tr>
<td>17.3 Printing the Page Directory</td>
<td>611</td>
</tr>
<tr>
<td>17.4 Helper Functions for Printing</td>
<td>611</td>
</tr>
<tr>
<td>17.5 Printing the Frame Table and Page Table</td>
<td>612</td>
</tr>
<tr>
<td>18 The ULIX Build Process</td>
<td>615</td>
</tr>
<tr>
<td>18.1 Required Software</td>
<td>615</td>
</tr>
<tr>
<td>18.1.1 Toolchain on Mac OS X</td>
<td>616</td>
</tr>
<tr>
<td>18.1.2 Other Useful Tools</td>
<td>617</td>
</tr>
<tr>
<td>18.2 Downloading the Development Environment</td>
<td>618</td>
</tr>
<tr>
<td>18.3 Bootstrapping: How to Start</td>
<td>618</td>
</tr>
<tr>
<td>18.3.1 Makefiles</td>
<td>620</td>
</tr>
<tr>
<td>18.3.2 Linker Configuration Files</td>
<td>622</td>
</tr>
<tr>
<td>18.3.3 The Assembler Pre-Parser</td>
<td>624</td>
</tr>
<tr>
<td>18.3.4 Creating Modules with module.nw</td>
<td>626</td>
</tr>
<tr>
<td>18.3.5 Pretty Printing for the Book</td>
<td>626</td>
</tr>
<tr>
<td>18.4 Directory Hierarchy and Makefiles</td>
<td>628</td>
</tr>
<tr>
<td>18.5 Making and Booting</td>
<td>628</td>
</tr>
<tr>
<td>18.5.1 User Mode Applications</td>
<td>628</td>
</tr>
<tr>
<td>18.5.2 The Disk Images</td>
<td>629</td>
</tr>
<tr>
<td>18.5.3 Alternative Boot Options</td>
<td>629</td>
</tr>
<tr>
<td>18.5.4 Informative Files</td>
<td>629</td>
</tr>
<tr>
<td>18.5.5 Manually Inspecting the C Files</td>
<td>629</td>
</tr>
<tr>
<td>18.5.6 Other Emulators</td>
<td>630</td>
</tr>
<tr>
<td>18.6 Online Resources: the ulixos.org Website</td>
<td>630</td>
</tr>
<tr>
<td>18.7 Tools</td>
<td>630</td>
</tr>
<tr>
<td>18.7.1 bindump</td>
<td>630</td>
</tr>
<tr>
<td>19 Where to Go Now?</td>
<td>633</td>
</tr>
</tbody>
</table>
# Contents

## A Introduction to C

- A.1 No Classes, no Objects ........................................ 635
- A.2 Data Types, Arrays and Pointers .......................... 636
- A.3 Strings? There is no String ............................... 638
- A.4 String Operations ............................................. 640
- A.5 Pointer Arithmetic ............................................. 642
- A.6 C Pre-Processor ............................................... 643
- A.7 Further Reading .................................................. 645

## B Introduction to Intel x86 Assembler

- B.1 CPU Registers .................................................. 647
- B.2 A Few Standard Commands ................................... 649
  - B.2.1 Moving Data Around ..................................... 649
  - B.2.2 Different Integer Sizes .................................. 650
  - B.2.3 Arithmetic Operations .................................... 650
  - B.2.4 Jumps, Calls, Comparisons and Conditional Jumps ....... 651
  - B.2.5 Pushing and Popping With the Stack ..................... 652
- B.3 Special Commands .............................................. 652
  - B.3.1 Data Storage (db, dw, dd) ............................... 652
  - B.3.2 Constants (equ) ........................................... 653
  - B.3.3 Macros (%macro) ........................................... 653
- B.4 gcc Inline Assembler ........................................... 654
- B.5 Further Reading ................................................. 656

## C Other Educational Operating Systems

### Appendices

- Chunk Index ......................................................... 661
- Identifier Index .................................................... 667
- Index .............................................................. 679
- Bibliography ......................................................... 693
- Image Credits ........................................................ 700
- GNU General Public License ....................................... 703
## List of Figures

1.1 Noweb tools extract source code and documentation. .................. 23  
1.2 Code chunks contain pointers that allow quick navigation to other chunks. 27  
1.3 Mind map for process functionality. ................................. 33  
1.4 The ulixos.org website has a download area and general information. .. 37  
1.5 The development environment is a virtual Debian Linux. ............. 38  

3.1 Partitioning: Fixed equal size. .................................... 51  
3.2 Partitioning: Fixed variable size. ................................. 53  
3.3 Partitioning: Buddy System. .................................... 56  
3.4 Partitioning: Buddy System (tree representation). .................. 56  
3.5 Partitioning: Buddy System (impossible join). ..................... 57  
3.6 Segmentation: Calculation of the physical address. .................. 58  
3.7 Segmentation: With segment limits. ................................ 59  
3.8 Internal vs. external fragmentation. ................................ 60  
3.9 Compaction of externally fragmented memory. ........................ 61  
3.10 Address translation via an address decoder logic. .................. 64  
3.11 Address translation for virtual memory. ............................ 65  
3.12 Organization of the virtual address space. ........................ 66  
3.13 Organization of the virtual address space with multiple stacks. ........ 66  
3.14 Schematic view of a refined translation table for virtual memory. .... 68  
3.15 Pages and page frames. ............................................. 70  
3.16 Structure of virtual address. ..................................... 71  
3.17 Virtual addresses with a 10-bit offset. ............................. 71  
3.18 Address translation using page descriptors. ........................ 71  
3.19 Successful page translation. ...................................... 73  
3.20 Multi-level page table example, page size = 1 byte. ............... 74  
3.21 Paging on the Intel i386 architecture. ............................. 75  
3.22 Tree structure of descriptors. .................................... 76  
3.23 Example structure of a multi-level virtual address. ................ 77  
3.24 Address translation using a two-level page descriptor tree. .......... 78  
3.25 Address translation using a three-level page descriptor tree. ......... 78  
3.26 Saving main memory using null descriptors in hierarchic page tables. .. 80  
3.27 Multiple indirection in Unix filesystems. .......................... 82
Figures

4.1 Simplified view of the Ulx memory layout. ........................................... 88
4.2 Intel i386 segment descriptor with base and limit. ................................. 90
4.3 Segment descriptors for code and data segments in the GDT. ....................... 92
4.4 Two-layered page table tree for the Intel x86 architecture. ......................... 98
4.5 Virtual address for Intel x86: directory, table and offset. ......................... 98
4.6 Page descriptors hold 20 address bits and twelve configuration bits. .......... 99
4.7 Page table descriptors hold 20 address bits and twelve configuration bits. ... 102
4.8 Identity mapping, 1st attempt: 1st MByte virtual memory = 1st MByte RAM. .............................................................. 105
4.9 Second mapping attempt: Kernel starts at 3 GB + 1 MB (virtual). ............ 107
4.10 Final mapping: Kernel starts at 3 GB + 1 MB and 1 MB (virtual). ............ 107

5.1 Two PICs are cascaded, which allows for 15 distinct interrupts. ............... 131
5.2 Accessing parts of EAX as AX, AH and AL. ........................................ 133
5.3 Structure of an interrupt descriptor. ...................................................... 136
5.4 Stack layout when entering an interrupt handler. ................................... 141
5.5 Stack after interrupt handler initialization by the assembler part. ............. 143

6.1 Memory layout of a process. .................................................................... 160
6.2 States and state transitions in the simple state model for threads. ............ 177
6.3 Thread states and transitions for a system with swapping. ....................... 179
6.4 Process states and state transitions as implemented by Ulx. ..................... 181
6.5 Implementation of ready queue and blocked queues. .............................. 182
6.6 Layout of a segment selector ................................................................. 193
6.7 The TSS (Task State Segment) structure. .............................................. 195
6.8 Segment descriptor vs. TSS descriptor. .................................................. 196
6.9 Forking a process creates an almost identical copy. ............................... 207
6.10 Calling fork four times creates this tree structure. ................................ 215

7.1 Assignment of virtual to physical processors. ......................................... 249
7.2 Distribution of the CPU burst length. ..................................................... 251
7.3 Schematic view of the processor hierarchy. ............................................ 252
7.4 Multiple kernel stacks and protective buffers. ........................................ 256

8.1 Service times for the FCFS scheduler. ................................................... 265
8.2 Time slices for a Round Robin scheduler. ............................................. 266
8.3 Determining time slices that serve interactive processes. ......................... 268
8.4 Two types of priority schedulers. ........................................................... 269
8.5 Kernel stack after timer interrupt occurs. .............................................. 273
8.6 Kernel stack before calling irq_handler. ............................................... 274

9.1 The “Clock” page replacement algorithm. ............................................. 303

10.1 Layout of a US-American keyboard. .................................................... 314
10.2 Layout of a German keyboard. ............................................................. 314
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.3</td>
<td>Scancodes for the US-American keyboard.</td>
<td>314</td>
</tr>
<tr>
<td>11.1</td>
<td>Example of adding an element to the front of a linked list.</td>
<td>348</td>
</tr>
<tr>
<td>11.2</td>
<td>Concurrent threads losing a linked list element.</td>
<td>350</td>
</tr>
<tr>
<td>11.3</td>
<td>Mutual exclusion between two threads.</td>
<td>351</td>
</tr>
<tr>
<td>11.4</td>
<td>Avoiding busy waiting by running a different thread.</td>
<td>358</td>
</tr>
<tr>
<td>11.5</td>
<td>Synchronization in user and kernel mode.</td>
<td>375</td>
</tr>
<tr>
<td>11.6</td>
<td>Synchronization functions for user and kernel mode.</td>
<td>378</td>
</tr>
<tr>
<td>11.7</td>
<td>One-sided vs. two-sided synchronization.</td>
<td>382</td>
</tr>
<tr>
<td>11.8</td>
<td>Two ways to enter the <code>debloc</code> functions with interrupts off or on.</td>
<td>384</td>
</tr>
<tr>
<td>12.1</td>
<td>Punch cards were used to store program code and data.</td>
<td>394</td>
</tr>
<tr>
<td>12.2</td>
<td>Organization of disk blocks in a contiguous filesystem.</td>
<td>396</td>
</tr>
<tr>
<td>12.3</td>
<td>Organization of disk blocks in a non-contiguous filesystem.</td>
<td>397</td>
</tr>
<tr>
<td>12.4</td>
<td>Unix inodes store direct and indirect block addresses.</td>
<td>401</td>
</tr>
<tr>
<td>12.5</td>
<td>Windows uses separate drive letters for each volume.</td>
<td>402</td>
</tr>
<tr>
<td>12.6</td>
<td>Unix systems integrate all volumes in one tree.</td>
<td>402</td>
</tr>
<tr>
<td>12.7</td>
<td>Layered design of the Virtual Filesystem.</td>
<td>404</td>
</tr>
<tr>
<td>12.8</td>
<td>Opening a file via the user mode library function <code>open()</code></td>
<td>407</td>
</tr>
<tr>
<td>12.9</td>
<td>Relationship of global, local and process file descriptors.</td>
<td>423</td>
</tr>
<tr>
<td>12.10</td>
<td>Layout of a Minix-formatted volume.</td>
<td>441</td>
</tr>
<tr>
<td>12.11</td>
<td>Minix subsystem functions called <code>mx_open</code>.</td>
<td>463</td>
</tr>
<tr>
<td>12.12</td>
<td>Single and double indirection in Minix inodes.</td>
<td>472</td>
</tr>
<tr>
<td>12.13</td>
<td>Displaying a symbolic link (in an old Ulix version).</td>
<td>483</td>
</tr>
<tr>
<td>12.14</td>
<td>Default contents on a fresh Ulix root disk.</td>
<td>501</td>
</tr>
<tr>
<td>A.1</td>
<td>Intermediate code files generated by the <code>gcc</code> compiler.</td>
<td>646</td>
</tr>
<tr>
<td>B.1</td>
<td>Accessing parts of <code>EAX</code> as <code>AX</code>, <code>AH</code> and <code>AL</code> (identical to Figure 5.2).</td>
<td>648</td>
</tr>
</tbody>
</table>
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Definition of “kilobyte”, “megabyte” and “gigabyte”.</td>
<td>25</td>
</tr>
<tr>
<td>3.1</td>
<td>Example of a segment table with three segments.</td>
<td>58</td>
</tr>
<tr>
<td>4.1</td>
<td>Contents of the Multiboot header</td>
<td>86</td>
</tr>
<tr>
<td>6.1</td>
<td>Analysis of the initial stack of a process after calling exec().</td>
<td>231</td>
</tr>
<tr>
<td>7.1</td>
<td>Possible order of execution for the threads shown in Figure 7.3.</td>
<td>253</td>
</tr>
<tr>
<td>12.1</td>
<td>Directory of a contiguous filesystem.</td>
<td>396</td>
</tr>
<tr>
<td>12.2</td>
<td>Directory of a non-contiguous filesystem.</td>
<td>397</td>
</tr>
<tr>
<td>12.3</td>
<td>Characteristics of the Minix filesystem versions.</td>
<td>440</td>
</tr>
<tr>
<td>12.4</td>
<td>Layout of a Minix-formatted 1.44 MB floppy disk.</td>
<td>443</td>
</tr>
<tr>
<td>13.1</td>
<td>List of block devices supported by Unix.</td>
<td>505</td>
</tr>
<tr>
<td>13.2</td>
<td>The status register of the IDE controller.</td>
<td>526</td>
</tr>
<tr>
<td>14.1</td>
<td>Standard signals on five Unix systems.</td>
<td>560</td>
</tr>
</tbody>
</table>
Operating systems are an important part of computing. They mediate between the complex intricacies of modern hardware and the abstract needs of users and applications. Also, operating systems are one of the oldest research areas in computer science. Moreover, the topic is one of the core subjects in academic computer science curricula. So there are many books and other teaching resources available.

Because of the wealth of material and the practical appeal of the topic, operating systems are a fun subject to teach which can also make it an enjoyable course for students. As regular instructors of two of these courses we have learned that one of the key aspects of a good operating systems course is to discuss and analyze real operating systems, i.e., operating systems that work in practice and can actually be used by people. To this end, open source systems like FreeBSD and Linux have established themselves as good objects of study because they have commercial value and their source code can be accessed and scrutinized by lecturers and students in course.

Unfortunately, it is hard to use the source code of such “real” systems directly and extensively in class. One main reason is the complexity of the code. Operating systems naturally have to deal with details of the underlying hardware. If a system is designed to be portable, the source code must cater for multiple computer architectures, which makes it even more complex. For example, at the time of writing this book, the current Linux version 3.14.5\(^1\) had 29 subfolders in the `linux-3.14.5/arch/` directory which contained the specific code files for those architectures.

```
$ cd ~/Downloads/linux-3.14.5/arch/
$ find . -type d -maxdepth 1 -mindepth 1 | sed -e 's/\./\//;/' | column -c160
alpha blackfin ia64 mips s390 um
```

While not all platforms are supported equally well, in same cases there is a tremendous amount of platform-specific code as the following examples demonstrate which show the lines of code of source files in the x86, ia64, alpha and powerpc subdirectories:

```
$ for arch in x86 ia64 alpha powerpc; do printf "%8s" $arch:; \
> find $arch/ -type f | xargs cat | wc -l; done

  x86:  318881
  ia64:  115179
  alpha:  62383
powerpc:  439961
```

Furthermore, modern operating systems usually offer an increasing number of features which must all be expressed by code. Another reason for the complexity of operating systems code is that practical operating systems must be extremely efficient. Every instruction cycle and memory cell used by the operating system cannot be used by the applications and their users. That is why operating systems code is often highly optimized. There are many other reasons that prevent instructors from showing real source code in class.

In our view the most important reason for not using real code directly in class is that operating systems have not been written with a human reader in mind, especially non-expert readers like students of operating systems classes who want to learn the basics of the area. Even when this is not the case and an operating system was developed for teaching purposes, such as the Minix operating system, it is often a tough task to browse through the complete source code that is split into several separate files and is ordered according to constraints of the programming language.

That is precisely where our approach differs: You can read this book from front to back and discover the theory and the implementation of the shown principles in the source code of the ULIX operating system that serves as an example.

## 1.1 Literate Programming

When we write software, do we really think about a human reader? The harder we try, the more we feel constrained by the programming tools available. When writing a complex piece of code, wouldn’t you sometimes like to include a figure into the code to explain the complex interactions of variables? Or when you implement an algorithm from a textbook, wouldn’t you like to give an automatic reference to this book in the source code? Or when one part of the code is similar to another part because you used the same idea, wouldn’t you like to use automatic cross-referencing between these two parts? And aren’t you bored of writing all these comment signs (like // in C or Java) all the time (or worse,
1.1 Literate Programming

aligning the stars when using /* and */? If you have ever felt such a desire, you are ready for literate programming.

Literate programming is a programming technique originally developed by Donald E. Knuth to write the \TeX \text{\textregistered} typesetting system. The source code of \TeX \text{\textregistered} appeared 1986 as a book called “\TeX \text{\textregistered} – The Program” [Knu86]. Reading that book is an entirely different experience from reading “normal” source code. It contains the entire source code, not just important excerpts, and it is real code that was compiled into the original version of the typesetting program.

With literate programming there is a conceptual change in the programming approach: Here, documentation comes first. A literate program is basically a text describing the program, and the actual code is inserted in the documentation, thus reversing the normal ordering. So instead of classically documenting code as in the following example:

```c
// This function takes an argument, squares it, and adds the constant 42.
int square_and_add (x) {
    const int add = 42;  // declare the constant
    x = x*x;            // square x
    return x + add;     // calculate sum, return result
}
```

a literate programming version of the same code might look like this:

```
We will write a function square_and_add which takes an argument x, squares it and adds some constant. Squaring a variable is just a multiplication of the value with itself:
\langle square x \rangle \equiv
\begin{align*}
x &= x*x;
\end{align*}

With this knowledge we can implement the function:
\langle function implementation \rangle \equiv
\begin{align*}
\text{int square_and_add (x) \{} \\
\quad \langle declare constant add \rangle \\
\quad \langle square x \rangle \\
\quad \text{return x + add;}
\text{\{} \\
\end{align*}

Now, what’s left is to decide on the constant add—we pick 42:
\langle declare constant add \rangle \equiv
\begin{align*}
\text{const int add = 42;}
\end{align*}

This is an implementation of the function \( f: x \mapsto x^2 + 42 \).
```

Note how the function definition uses two code chunks: one that was presented before the function and another one which came after the function. Literate programming makes the developer independent of any specific ordering of code fragments which the language may
Introduction

Following the idea of literate programming, this book is a book for students. Its source code can be used to generate executable code (for the Ulix operating system) and a \LaTeX{} file that can be typeset to an introductory text on operating systems (this book).

When you look at the source file \texttt{ulix-book.nw} from which this book was created, you will see how code chunks are declared: The chunk name is always put between << and >>=, then follows the code which is terminated with a @ character on a single line. When referencing a code chunk the same syntax (without the equals sign) is used. So for example, in order to declare the \textit{(function implementation)} chunk from above, the following lines are needed:

\begin{verbatim}
<<function implementation>>=
int square_and_add (x) {
    <<declare constant add>>
    <<square x>>
    return x + add;
}
@
\end{verbatim}

Chunks may also be continued, i.e., some pages (or even chapters) after the initial definition of a code chunk it is “defined again.” That does not replace the original definition, but add to it. In the source file this is done by simply writing

\begin{verbatim}
<<function implementation>>=
...
@
\end{verbatim}

again (with additional code lines inside), but in the PDF version the representation of a continued code chunk looks slightly different: Instead of

\begin{verbatim}
(function implementation) =
\end{verbatim}

it will look like this:

\begin{verbatim}
(function implementation) + =
\end{verbatim}

with a \texttt{+ =} sign that indicates the \textit{continuation}—similarly to C’s += operator that also has an “add to” meaning (e.g. \texttt{x+=5;} means: add 5 to the value of \texttt{x}).

1.2 The Ulix Operating System

Throughout this book we present the whole source code of the Ulix operating system. That encompasses the kernel but also the user mode library which application developers must use in order to interface with kernel functions. The book does not show source code for (all) the applications.
1.2 The ULIX Operating System

1.2.1 The Name of the Game

The name ULIX is intentionally similar to the name Unix. This is meant to imply that the
code is influenced a lot by things we have seen in Unix/Linux style operating systems.
The focus on Unix is solely based in the past experiences of the authors, it is in no way
intended to imply that Unix is better or worse than other operating systems.

The choice of the letter “L” in ULIX is supposed to concisely express that the system
is meant for learning and teaching operating systems. Besides, the name ULIX is one of
the few four-letter abbreviations that do not seem to have been chosen for other software
systems yet. Furthermore, ULIX is probably the first operating system that is written as a
literate program, and so ULIX can also stand for “literate Unix”.

1.2.2 Design Principles of ULIX

The following principles have determined the design of ULIX:

- ULIX is for learning and teaching principles of operating systems in a course. ULIX
  should never be a practical system in the sense that it can be used to run real applica-
tions.

- Nevertheless, ULIX should be a real operating system, i.e., the code should be exe-
cutable on some well-defined computer architecture. If necessary, it should be possi-
ble to port ULIX to other platforms, but portability is not a core requirement of ULIX.

- The design and implementation of ULIX should be governed by the principle of sim-
plicity, avoiding optimizations, focusing on understandable and correct code.

- It should be possible to use the source code directly in class. The source code should
  be written with the human reader in mind.

Given the above design principles, one point should be clear—but is important enough
to mention it anyway: While trying to be real, ULIX is not practical, i.e., we disclaim
any fitness for practical use. On the one hand, the code is not guaranteed to be free of
programming errors. On the other hand, the performance of ULIX is such that it will not
achieve any required quality of service in practice. ULIX is purely for learning. The path
chosen is the one of simplicity. So when you have read this book, you will have a fairly
good idea of how ULIX works, but only a faint idea of how operating systems work in
general. Therefore, this book is not (and never will be) a replacement for the available
excellent general textbooks on operating systems.

1.2.3 ULIX Features

Some of the words in the following feature list may not make sense to you right now,
but if you already have some knowledge of operating system principles, this gives you an
idea of what kind of content to expect in the book. If you are familiar with some Unix
system as a user or administrator, at least most of the application programs mentioned in
the following paragraph should be well-known.
ULIX is a classical Unix-like operating system which features processes with separate address spaces, threads, paging, a virtual filesystem (currently supporting floppies and hard disks, with Minix as the primary filesystem) and synchronization via kernel mutexes and semaphores. It has a preemptive scheduler (implementing a simple Round Robin strategy) and allows for the integration of new interrupt and system call handlers at runtime. ULIX supports up to ten text mode terminals. It works on 32-bit Intel-compatible CPUs and provides system calls (via the classical int 0x80) and corresponding user mode library functions which are compatible to other Unix systems, e.g. fork, execv, exit, waitpid, signal, and kill for process control, some of the pthread_* and pthread_mutex_* functions for thread control [IEE95], open, read, write, lseek and close for file access, brk for dynamic memory (heap) management etc. There are also a few user mode programs (cat, chgrp, chown, chmod, clear, cp, df, diff, free, grep, hexdump, kill, ls, man, mkdir, ps, readelf, rm, rmdir, stat, sync, touch, vi and wc), and some more commands are implemented as shell built-ins of the ULIX shell sh, e.g. cd and pwd.

A login mechanism asks for user name and password and checks these against entries in /etc/passwd (where the passwords are stored in plaintext and world-readable since hashing and encryption are not available in ULIX), and authenticated users can only access files for which they have the required access permissions. Also, signaling other processes via the kill function or program requires the user to own the targeted process (or have administrator privileges).

Redirection of standard input, standard output and standard error are supported via closing one of the file descriptors 0, 1 or 2 and opening a new file (which will reuse that descriptor). The shell understands the <, > and 2> syntax for starting programs with redirections, and it can also use the feature to execute shell script files (via sh <script.sh).

ULIX uses a buffer cache for disk read and write operations so that repeated access to the same data on disk is faster than first access. A swapper process runs in the background and checks whether physical memory fills up too much: if so, it will pick parts of memory and write them to disk in order to increase the available memory. If such a “paged-out” memory area is accessed later, it will be “paged in” again before the program that wants to use it can continue.

The system can be run inside a virtual machine using qemu and it writes a kernel log to a (virtual) serial port; you can save this output in a file for later analysis. The simple shell can launch user mode programs (which need to be compiled outside ULIX). Such programs are stored using the ELF executable format [TIS95]. ULIX expects to have 64 MByte of RAM and maps that physical RAM to a fixed address space region for easy access to the physical memory. POSIX threads are somewhat limited in that each thread has a fixed-size stack (whereas the primary or only thread has a stack which automatically increases its size as needed).

1.3 Tools

The original goal was to write a book like “\TeX – The Program” [Knu86] with pretty-printed sourcecode and extensive automatic cross-references and indexes (especially the
famous mini indexes on right-hand pages). This can be achieved using the Knuth/Levy CW\-EB documentation tool [KL03, KL01] that also has hypertext extensions. Mini indexes can be generated by an extension called CTWILL [Knu93] that is the program used to generate the source of “\TeX – The Program”. However, the CW\-EB family of tools is restricted to \TeX as typesetting language which we wanted to avoid in favour of \LaTeX. While there exists an experimental adaption of CW\-EB for \LaTeX by Joachim Schrod [Sch13], CW\-EB is also restricted to the C programming language, so more than one language (e.g. C code, assembler code and configuration files) cannot be handled.

For this book we decided to use Norman Ramsey’s noweb tool [Ram94] which combines a simple syntax with language independency: it uses \LaTeX (or alternatively HTML) as input and output format for the documentation. There are also several extensions which add pretty-printing and indexing.

The main source for the book is one huge \LaTeX file that serves as input to the noweb tool chain. The program noweave converts it into a classical \LaTeX file that can then be processed as usual; for this book we chose the Xe\LaTeX variant [XeL14] of \LaTeX because it handles UTF-8-encoded input files and provides better options for font selection. Together with standard \LaTeX tools (bibtex and makeindex) it produces the PDF file that you’re currently reading in the PDF viewer or as a printed copy.

The other part of the noweb tool chain generates the Ul\-ix source code files: notangle extracts code chunks and saves them in source files which will then be compiled or assembled with the GNU C compiler gcc and the nasm assembler.

Figure 1.1 shows a simplified view of the build process for both the book as well as the kernel, the user mode library and a sample application. We will give a detailed description in Chapter 18.

![Diagram](image.png)

Figure 1.1: Using the Noweb tool chain it is possible to generate this book and also the Ul\-ix kernel binary and other system files.
1.4 Copying

ULiX is Free Software, following the definition of this term that was coined by the Free Software Foundation (FSF). You can copy and browse the code and compile it into any form you like. If you find bugs in the source code, please drop us a message so that we can fix the bug in the next iteration of the software. Similar to \( \text{\LaTeX} \), ULiX is meant to eventually become a stable platform that does not evolve anymore, i.e., we will eventually stop issuing new releases. That’s why we retain the entire copyright that must be mentioned in all files associated with the system. If you think that ULiX should be fundamentally changed, become more efficient etc., you can take the code but should call the resulting system something other than “ULiX”.

---

1.5 Notation

In this section we describe the various conventions used throughout the book.

1.5.1 Numbers and Units

This book makes heavy use of hexadecimal numbers and (less often) octal and binary numbers. We expect readers to be familiar with these positional numeral systems with bases 16, 8 and 2.

- Hexadecimal numbers such as 0xAB12CD34 will appear with a monospaced font and a 0x prefix (since that is the notation also used in C). Sometimes we will insert a dot in the middle (0xAB12.CD34) to improve readability.

- Octal numbers such as 56701_8 also use the monospaced font but have a small _8 index at the end to indicate that the octal system is used. C uses a different notation: every
number which starts with a zero is considered to be an octal number which sometimes causes problems for readers who are not familiar with this convention.

- We write binary numbers like 100101, in a similar fashion, but with a \textit{b} index at the end. Standard C has no way to express binary numbers, but the GNU C compiler allows the non-standard syntax \texttt{0b100101} (with a \texttt{0b} prefix, similar to the \texttt{0x} prefix for hexadecimal numbers).

We often discuss kilobytes, megabytes (or rarely gigabytes) when talking about memory or disk areas and file sizes. When we do so, we use the historical meanings as displayed in Table 1.1. Note that most newer books use the reformed interpretation according to which kilo-, mega- and gigabytes refer to powers of 1000 of bytes, which is more consistent with other uses of “kilo” and “mega” like, for example, kilometers and megatonnes. The new terms for the old units are “kibibyte”, “mebibyte”, “gibibyte” etc. [Inter00].

However, having a special unit name for 1000 bytes or a million bytes is quite useless since these sizes have no meaning: 1000 is not a power of 2 (but 1000 = 2^3 \times 5^3), and everything in the hardware is organized by powers of 2. That is why we have decided to stick with the classical meanings even though they are considered wrong by today’s standards. If you’re already used to the new notation, please replace any occurrence of “kilobyte” with “kibibyte”, “megabyte” with “mebibyte” etc.

Note also that the often-used term “1.44 megabyte floppy disk” makes no sense at all, because such a floppy stores 2880 sectors each of which is 512 bytes large. Thus, the disk size is 1440 KByte which is 1.40625 MByte. When using the new terms the size could be expressed as 1474.56 kB or 1.47456 MB, and none of those numbers should lead to “1.44”—you can only arrive there by mixing the old and the new terminology and claiming that a megabyte was 1000 \times 1024 bytes.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Abbreviation</th>
<th>Size in bytes</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kilobyte (“Kibibyte”)</td>
<td>KByte</td>
<td>(2^{10} = 1024^1 = 1024)</td>
<td>0x00000400</td>
</tr>
<tr>
<td>Megabyte (“Mebibyte”)</td>
<td>MByte</td>
<td>(2^{20} = 1024^2 = 1048576)</td>
<td>0x00100000</td>
</tr>
<tr>
<td>Gigabyte (“Gibibyte”)</td>
<td>GByte</td>
<td>(2^{30} = 1024^3 = 1073741824)</td>
<td>0x40000000</td>
</tr>
<tr>
<td>“New Kilobyte”</td>
<td>kB</td>
<td>(1000^1 = 1000)</td>
<td>0x000003E8</td>
</tr>
<tr>
<td>“New Megabyte”</td>
<td>MB</td>
<td>(1000^2 = 1000000)</td>
<td>0x000F4240</td>
</tr>
<tr>
<td>“New Gigabyte”</td>
<td>GB</td>
<td>(1000^3 = 1000,000,000)</td>
<td>0x3B9ACA00</td>
</tr>
</tbody>
</table>

Table 1.1: In this book we use the classical meanings of “kilobyte”, “megabyte” and “gigabyte”.

For comparison, we have added the new interpretations to the table; especially when looking at the hexadecimal representations you can see that these units do not occur in practical settings.
1.5.2 Identifiers

Names of variables, functions, constants and other code elements that appear in a regular paragraph also use the monospaced font, and we sometimes add round brackets to a function name when we want to emphasize that it is the name of a function. So you may find references to “the read_file function” or simply to “read_file()”. When a variable or function is declared or defined in a code chunk, there will often be a suffix which indicates the page number where the definition can be found, e.g. for the readblock_hd function.

Some functions appear to exist twice: once inside the kernel, and once in the user mode library which applications must link to access those kernel functions. An example is the open function which opens a file. Surely, the kernel needs a way to open files, and applications also have that need. While both functions will never appear in the same context, they do appear in this same book which contains cross references to the place where a function was defined. Thus, using the same name would cause some confusion, so we use slightly different names in the kernel by adding a u_prefix to the name. The kernel uses the u_open function, and the user mode library provides an open function.

Constants (defined via the pre-processor’s #define macro) use all-upper-case names and underscores as word separators, for example MEM_SIZE.

1.5.3 Margin Notes

You will have noticed already that the page margin sometimes contains a few words, for example, in the above paragraphs, there’s a “u_prefix” margin note. These notes are helpful in two ways: when you thumb through the pages of the book, you can quickly identify key words, and when you have looked up a word in the index and gone to the right page, a margin note leads you to the right line.

The margin also contains code chunk numbers in [ ] brackets (see the following section).

1.5.4 Code Chunks

The most important bits of information within each code chunk are the chunk name and the actual code, but there is more, and knowing what the additional elements of a code chunk mean will help you navigate the code more easily, for example when you find a function call to a function that you have forgotten (or that may be defined only later in the text).

Figure 1.2 shows two example code chunks that demonstrate all the elements you will find in the chunks.

- First of all, each code chunk has a name. When a chunk is defined for the first time and you’re on page 123 of the book, the chunk will begin with a line like

  \( \langle \text{chunk name 123} \rangle \equiv \)

  which confirms that this chunk indeed starts on page 123. If more than one chunk starts on this page, a lower case letter is appended to the page number so that you can distinguish between them, e.g. 123a, 123b and 123c for three code chunks on
page 123. If the chunk definition started earlier (and this is a continuation) then the original chunk number will appear next to the chunk name.

- Regardless of whether the current chunk is an initial definition or a continuation, it always has an individual chunk number. This number is displayed in the margin inside [ ] brackets, e.g. [122c] in the figure.
- The top line of the chunk contains up to three further chunk numbers on the right-hand side. If you find a chunk number in round brackets (such as (62a) in the figure), then it is a reference to the first place where this chunk is used. The next two chunk numbers are prefixed or suffixed with a left or right facing triangle. If a number with a left triangle exists (in the example: < 122b), this points to the previous continuation, and a number with a right triangle (in the figure: 124c >) leads you to the next continuation. Using these forward and backward pointers you can find all locations where this chunk is defined (and thus read the whole source code that the chunk is made of).

When one or more of these three numbers do not exist, that has the obvious meaning: A missing (...) reference means that the chunk is used nowhere. That should only happen for “root chunks”: those are the chunks which `notangle` extracts to create the

![Figure 1.2: Code chunks come with links to other parts of the source code and let you navigate through the sources: you can quickly find out where a function or variable was defined, and below the defining chunk you see all the places where it is used.](image-url)
compiler/assembler source files. A missing forward or backward pointer simply indicates that you have reached the last or first part of the chunk definition, respectively.

- Under some chunks you find a line that starts with "Used:" and lists all (known) identifiers and the chunk numbers where they were defined. Both examples in Figure 1.2 contain such a line.

- Finally, a code chunk may define one or more identifiers. In that case, you will see a block starting with "Defines:" and a separate line for each defined identifier, giving its name and all the locations in the book where it is used. In the example figure only the second code chunk has such a block. The "Used:" and "Defines:" blocks let you track the usage of functions, variables and constants throughout the whole code which is often more helpful than searching for an identifier in the code files.

- There is one case that we treat in a special way: The C language demands that functions are declared before their first use. Since we do not want to consider the ordering of function implementations it is often necessary to insert a function prototype which will appear early in the C file, whereas the implementation occurs at a later position. So there will be prototype code of the form

```
int function_name (arguments);  // prototype
```

and implementation code that looks like this:

```
int function_name (arguments) {  // implementation
...
    return ret_val;
}
```

Technically (from a literate programming point of view), only one of those code blocks defines the function, and the other one uses it. In the book we will often, but not always, present both parts on the same page though they will appear in different places in the generated C file. We have decided to make the implementation chunk the defining chunk (since that is what you will want to look up when you see usage of a function). We have also removed the "Used:" line for the prototype chunk since it will refer to the chunk which immediately follows.

Even if you have a printed copy of this book, it is helpful to use the PDF version as well, since all chunk numbers are links to those chunks, and most identifier names are clickable, too, leading to the place where a variable, constant or function is defined.

### 1.5.5 Coding Standard

The code uses two spaces for indentation, and the curly brackets { and } which declare the beginning and the end of a block are always written in this form:

```c
int sum_of_first_ten () {
    int j, sum;
    for (j = 1; j < 11; j++) {
```

```c
sum += j;
}
return sum;
}

We often use the // one-line comments which originally were not part of the C standard, but are supported by modern C compilers. So instead of classical C comments such as

```
int tmp = array[i].mem; /* the memory address */
```

you will more often see this form:

```
int tmp = array[i].mem; // the memory address
```

In the book, the comparison operators <= and >= are displayed as ≤ and ≥, but this is the only kind of pretty-printing that we have applied to the source code chunks, except for a little color highlighting of comments, brackets and the exclamation mark and slanted printing of the #define and #include pre-processor commands:

```c
#include "ulixlib.h" // use the library
#define TRUE 1 // define a symbolic constant
void example_function (int param) {
    int vector[10];
    for (i = 0; i ≤ 10; /* inline comment */ i++) {
        if ( !param ) { vector[i] = vector[i] * 2; }
    }
}
```

In functions which have no return value (typed as void) we omit the return statement because the function automatically returns when it reaches the end of the function’s body; the C standard allows this practice, and it saves a line in each such function.

### 1.5.6 “Going where?”

At the beginning of some sections you will find a short text in two-column layout using a different font, such as the following:

<table>
<thead>
<tr>
<th>Going where?</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is an example for a “Going where?” paragraph. Right now it has no useful contents, so let’s just note that you will soon see the first lines of real code.</td>
</tr>
</tbody>
</table>

Sometimes it is easy to get lost and wonder: why have we been dealing with this or that function? In such cases, the “Going where” paragraph gives you some orientation. How do things fit together? Where are we in the larger picture?
1.6 Creating an Operating System From Scratch

Writing an operating system is very different from developing an application. That is because, when you start out, there is literally nothing. For example, there are no libraries that would provide standard functions such as `printf`. Also there are no rules dictating how the operating system is going to handle things—it is up to the developers to decide what those rules should be.

First of all, some elementary questions need to be answered:

- What kind of hardware will the OS work on?
- What programming language(s) should be used for development?
- What kind of applications will the OS be able to host?

If you expect to create an OS using Java that will be compatible with Windows, Linux and OS X and will also sport a high performance 3D engine so that the latest console games run on it, then this textbook will be pretty disappointing.

1.6.1 Selection of Target Hardware

The computing world is diverse, allowing for all sorts of hardware architectures. CPUs can have very different features—if you have attended a course on computer architectures, you will have noted things like RISC and CISC CPUs with very small and simple or huge and complicated instruction sets. In this book we will focus on the 32-bit Intel architecture, for the simple reason that most people have quick access to an Intel-compatible machine or can at least run an Intel-based operating system in an emulator. With their 32-bit architecture, the CPUs can access 4 GByte of physical memory which is enough for most purposes. During the last years 32-bit CPUs have become a bit outdated, as the latest processors have a 64 bit wide address bus and provide internal registers with the same size.

Intel hardware has some legacy problems because even the latest (64-bit) Intel chips are compatible with old systems from last century’s 80s. We will see this when we discuss the management of memory.

1.6.2 Language of Choice

Operating systems are very close to the actual hardware. In fact you won’t see any other class of “programs” which get any closer to the hardware (except for a computer’s firmware), because there’s always the OS as a natural barrier between hard- and software. We’re in the area of systems programming, and this is where “old school” languages still dominate. So with most operating systems you’ll see lots of C code. For those who have never heard of C (without a `++` or `#` postfix): C is a procedural language that was created in 1969–1973 by Dennis Ritchie⁴ and it’s a predecessor to C++, Java and C#. It does not know objects.³ U⁴⁴⁰x was (mostly) written in C.

---

² Ritchie reviewed the early history of C in an article [Rit93].
³ For those readers who are unfamiliar with C, we have included a short introduction to C which requires C++ or Java knowledge, see appendix A on page 635.
The principle of simplicity demands that we use “clear C”, i.e., we discipline ourselves to non-optimized and clear code that can also be understood by people familiar with Java. We also restrict inclusion of library header files (we want to be self contained).

Even closer to the hardware is assembler code, and for that reason all the early operating systems were programmed in assembler. Assembler code is what a C compiler will generate when you provide it with some C source code. Today it is no longer necessary to write complete operating systems in assembler (some people still do this, e.g. the BareMetal [Sey13] developers), but you’ll still need some assembler code from time to time, because some parts of the OS need to access CPU registers or execute special machine instructions which are not available in the C language.⁴

For the Intel processor platform, two “dialects” exist, the Intel and the AT&T one. The GNU C compiler supports both but defaults to the AT&T variant. We have decided to use the Intel syntax, because it is closer to C’s syntax: For example, you can load the EAX register with the value 0 via the command \texttt{mov} \texttt{eax, 0} (in Intel syntax). So the target of the \texttt{mov} command comes first which resembles the C command \texttt{eax = 0}. In AT&T syntax, the operands are reversed, with the target coming last and extra syntactical elements being needed (\texttt{mov $0, %eax}).

### 1.6.3 Applications

An operating system X will run applications which have been developed for X (let’s call them “X applications”), and it will be either impossible or very hard to run Y applications for any Y which is not X. Every OS creates its own software universe, and if you want to run a program from a parallel universe, you’ll need some sort of emulation—which is not a topic of this book.

Most applications require libraries which are typically considered part of the operating system. Even for something as simple as printing “Hello world”, you need a library that contains the code which is necessary to make the OS print something (in a text console, a window or perhaps on a printer).

In principle it would be possible to port many of the existing Unix (text-mode) applications to \texttt{Ulx}, however most programs make excessive use of the available libraries, so those would have to be ported first. This version of \texttt{Ulx} comes with a limited set of tools, including a very limited version of the \\texttt{vi} editor. So do not expect to replace your current Linux installation with a \texttt{Ulx} system.

### 1.7 What’s in the Book?

Reading this book, you will see introductory descriptions of several theoretical concepts, and at the same time you’ll see the complete source code necessary to implement these concepts.

What’s in the book is in principle what’s required for a typical operating system—but note that for all theoretical problems there are lots of possible solutions, and \texttt{Ulx} pro-

---

⁴ Appendix B on page 647 provides an introduction to the x86 Assembly language.
vides one solution to most problems. In many chapters we will begin with a theory section which gives broader information about the topic. We might present several different approaches but show only the implementation of one of them: the one we picked for Ulx. This will typically be the most simple solution, because we did not want to make the code too bloated.

Let us first look at some of the tasks an operating system needs to perform. For every modern system, the process (or more precisely: several processes) is a central idea. Many processes may coexist on a machine, and the CPU shall execute them in parallel. We need ways to spawn new processes and terminate processes which have completed their tasks.

The OS kernel will need internal **fork**, exec and **exit** functions for process creation and termination, but those will not be available to running programs (which are not running inside the kernel)—this problem is typically solved via system calls. So we need a system call interface and system calls **fork**, exec and **exit** which allow access to the kernel functions.

Starting a new process often requires loading the program code from a disk, and once the program is running, it will perform I/O operations: typically with the disk, the keyboard and the video display. Keyboard input arrives via keyboard interrupts, so our operating system must deal with interrupts. These are also needed for talking to the disk, and since we do not want to perform raw sector read/write operations to access data, we need a logical filesystem (which will be the Minix filesystem, version 2). Both the OS itself and processes will **open**, **read** and **write** files, and in order to let processes perform these actions, we need more system calls that—again—allow access to these functions.

Since each process has its separate memory, we need kernel routines for memory management. Ulx uses paging and creates address spaces for all processes (which are basically page tables with some additional administrative data). For each newly created process the system must reserve some memory; at least it will have to create a new address space. Later (at process termination) this memory must be set free. We must define a memory layout that describes what parts of (virtual) memory belong to the kernel and what parts belong to processes. A process may also ask for more memory while it is running, this requires another system call (**brk**) which can be implemented by allocating a new page for the process.

Unix systems use a signaling system that allows both processes to send signals to other processes as well as the kernel itself to signal a process. Processes may register signal handlers in order to avoid the default actions for receiving a signal (typically: ignore or abort). Again we need more system calls for registering handlers (**signal**) and sending signals (**kill**). The scheduler must check whether a process has pending signals when it activates it; in that case it must call the signal handlers (or perform the default action).

If we combine the ideas about processes which we have presented so far and add some further process properties, we arrive at the mind map shown in Figure 1.3: it is rather complex, and we have only looked at the system from the process perspective.

Thus, we will cover the following topics:

- **Memory Management Theory** (Chapter 3) gives an introduction to the possible ways in which the system’s main memory can be shared between several processes. We start with some simple models and quickly turn to paging, today’s standard method.
Figure 1.3: Mind map for process functionality.
• **Boot Process and Memory Management in Ulx** (Chapter 4): After the theoretical introduction you’re prepared to look at the first steps of the system initialization. We need to load the kernel, but where should we place it in RAM? The code in this chapter enables the paging mechanism and builds the foundation for process-related code which will allow to switch between several address spaces.

• **Interrupts and Faults** (Chapter 5) trigger the execution of handler functions which we must provide—they should do something useful about the event that started them, for example read data from a device that signaled the completion of some activity.

• **Processes and Threads** (Chapters 6 and 7) are all about running programs which most people consider the primary task of an operating system. We present code for classical Unix processes (handled by the fork, exec, exit functions) and an approximation of POSIX threads.

• **Scheduling** (Chapter 8) provides Ulx with its multitasking feature. You will learn about some approaches towards scheduling and then see the implementation of a round-robin scheduler. This is both about deciding when to switch to a different process (and which one) and about the switching itself: What do we have to do to temporarily halt one process so that another one can run?

• **Handling Page Faults** (Chapter 9) is a continuation of both the chapters on memory management and fault handling: a page fault is a fault that we can often recover from without terminating the causing process. In this chapter we also present our routines for paging out and back in: When memory becomes scarce, we write parts of a process’ memory to a disk file in order to free space—when we need it again, we bring it back.

• **Talking to the Hardware** (Chapter 10) deals with support for some standard devices, including the screen, the keyboard and the on-board clock’s timer.

• **Synchronization** (Chapter 11) is necessary because we allow multitasking. We show the implementation of kernel semaphores and mutexes which are standard primitives that help protect data against data-corrupting simultaneous access. We also explain how interrupt handlers and system call handlers (called by processes) which access shared data can be synchronized.

• **Filesystems** (Chapter 12): Here we deal with the logical aspects of accessing files on media. We present the Ulx Virtual Filesystem and our implementation of the Minix filesystem. Actually talking to the disk drive controllers is left to the next chapter.

• **Accessing Hard Disk and Floppy Disk Drives** (Chapter 13) requires some understanding of the protocols that these controllers “speak”. This chapter is very technical, but you do not have to deal with all the details in order to see how disk access works and data can be transferred between a disk and memory.

• **Signals** (Chapter 14) are a classical Unix mechanism which allows a simple kind of messaging: processes can send signals to other processes which makes them either terminate or call a registered signal handler. One use case is killing a process.
• **Users and Groups** (Chapter 15) let several users share one system but keep each user’s data private. Every file belongs to one specific user (its owner), and each process is associated with one user (its creator), as well. We show how Ulix implements the standard user/group mechanisms of Unix systems.

• The **Small Standard Library** (Chapter 16) provides often-used but less interesting functions such as `printf` and `memcpy`. This is not really what an operating system is concerned with, but without output and string management functions we could not do a lot.

• **Debugging Help** (Chapter 17) contains the code of the kernel shell and its internal commands. That shell is only available for debugging purposes.

• **Build Process** (Chapter 18): In this chapter we discuss how you can extract the source code from this book (assuming you have its Noweb source file `ulix-book.nw` and then build the system. Read it if you want to modify the system. If you only want to run Ulix, there are easier ways to get started.

• Finally, for those new to C and/or Assembler, there are introductions to C (Appendix A) and to the Intel x86 Assembly Language (Appendix B).

We will give a more detailed description of some of these topics in Chapter 2.

If you copy all the code from the book into appropriate files (or download the version we provide on the website) you can compile it into an operating system that will actually boot on the qemu PC emulator.

We’ll set the OS name and version now:

```c
(macro definitions 35a)≡
#define UNAME "Ulix-i386 0.13"
#define BUILDDATE "Mon Nov 2 17:33:51 CET 2015"
```

Defines:
- `BUILDDATE`, used in chunks 337c and 605a.
- `UNAME`, used in chunks 337c, 605a, 609, and 610a.

```c
(version information 35b)≡
/*
v0.01 2011/06 first version: boots, enables interrupts, keyboard handler, protected mode; most code taken from kernel tutorials
v0.02 2011/07/31 paging for the kernel (not yet for user space)
v0.03 2011/08/12 paging with Higher Half Kernel / GDT trick (preparation for user space)
v0.04 2011/08/17 dynamic memory allocation: request frame, request new page (with update of page table; creation of new page table if last used one is full)
v0.05 2012/10/02 serial hard disk and external storage server (for use with qemu & co.)
v0.07 2013/04/05 Scheduling and fork / exec / exit / waitpid are working.
v0.08 2013/07/13 Minix Filesystem support (replaces "simplefs"). Can read, write, create files. Kernel uses floppy (FDC controller) instead of serial disk
Terminal support (up to ten terminals with shells)
*/
```

We'll set the OS name and version now:
Welcome to Ulix 0.13!

1.8 Helpful Previous Knowledge

This book is targeted towards both undergraduate and post-graduate students and instructors who consider using a Unix-like system in a course on the design and implementation of operating systems. At the minimum, readers should be familiar with the following topics:

- Software development with a C-like language, e.g. C++, C# or Java
- Data structures and algorithms
- Basic understanding of some assembler language (the Ulix sources contain a few lines of 32-bit Intel assembler code)

Experience with the following topics is not required but considered helpful:

- Standard Unix library functions such as open, read, write, fork, exec (as they are sometimes taught in a system programming course)
- Unix command line (shell), such as bash or ksh
- \LaTeX{} document preparation system (required for most of the exercises in the book)
- Software development with C
- 32-bit Intel Assembler language
- Build process in a Unix environment, using makefiles and command line tools for compiling, assembling and linking

1.9 Online Resources

The Ulix website http://www.ulixos.org/ has a download area with files which are needed for working on the exercises in this book (Figure 1.4).

This is the

First Edition (09/2015)
1.9 Online Resources

Figure 1.4: Download resources and information about the project are available on the http://www.ulixos.org website.

of the book, so at the time of writing the website contained only the resources for this edition, but later editions may use different versions of the download files; so you should make sure that you access the right files.

The two main files you find in the download area are the following:

- Development system: We provide a VirtualBox appliance which you can import in the VirtualBox virtualization software. It contains a Debian Linux 6.0.1 installation with a simple desktop (Xfce) and all the development tools that you need for compiling the current Ulix version (see Figure 1.5). The sources are included as well, and also a few feature-reduced versions of the Ulix source code which you can extend when you work on the exercises. This virtual machine is the recommended development environment.

- The Noweb source file ulix-book.nw from which you can generate this book and the Ulix kernel. We recommend this download if all you want to do is regenerate the PDF file of the book. For compiling the Ulix kernel you should use the development system described above, because the source code depends on the right compiler version being installed.

For a detailed description of how to get started, see Section 18.3 which describes how you can set up a development environment. Note that you need no such installation if you simply want to read the book. However, if you want to work on the exercises which you can find in some chapters, you need the tools and sources.
1.10 Further Reading

There are several educational operating systems which have been used in courses or as the basis for textbooks with a similar purpose as this one. They are all united by the idea that students should take a look at real code in order to fully understand what tasks an operating system has to fulfill and how this can be done in practice. In addition, for some “real” operating systems documentation of design principles and implementation details is available that is also helpful in an educational setting. In Appendix C we give several examples for both categories.

In our view, the two most positive examples of operating system exposition for students are the well-known Minix operating system and the less well-known Xinu operating system:

- Minix was originally written by Andrew Tanenbaum [Tan87] to serve as a minimal working example for teaching his operating systems course at VU University Amsterdam (Vrije Universiteit Amsterdam). In its most advanced form (Minix 3), the system is still well-structured, simple and very well documented [TW06]. However, it has also evolved into a commercially relevant system with all advantages and disadvantages. Although the Minix book [TW06] contains the entire source code of the operating system, it lives a separate life being relegated to an appendix that essentially fills the second half of the book.
UlIX uses version 2 of the Minix filesystem as its native filesystem because Linux supports this filesystem very well. However, no code was borrowed from Minix except for some structure declarations.

- The Xinu system was written by Douglas Comer [Com84] for the DEC LSI 11/2 microcomputer (a successor of the famous PDP 11 minicomputer for which UNIX was initially written). This system was later ported to the IBM PC’s Intel 8088 processor [CF88], the Apple Macintosh’s Motorola 68000 CPU [CM89] and, quite recently, to the Linksys E2100L Wireless Router [Com11] which uses a MIPS processor. There are even further ports of the Xinu code (e.g. to 32-bit Intel machines), but those have not led to new editions of the Xinu book. While not written in literate programming style, the documentation and presentation of code portions somewhat resemble literate programming.

1.11 About the Authors

Felix C. Freiling is a professor of computer science at Friedrich Alexander University Erlangen-Nürnberg and heads the Chair for IT Security Infrastructures of the Computer Science department (http://www.cs.fau.de/). Before coming to Erlangen, he was a professor at RWTH Aachen and University of Mannheim. He started the UlIX project during his time in Mannheim where he gave lectures on operating system principles.

Hans-Georg Eßer is a PhD student at the same chair; he studied computer science and mathematics at RWTH Aachen. He is the editor-in-chief of EasyLinux magazine (http://www.easylinux.de/) and has been teaching operating systems at several universities of applied sciences since 2006. The completion of this book (and the UlIX implementation) was the major part of his PhD thesis.
You will soon see the first lines of code of the UlIX operating system. The presentation of code roughly follows the order in which it was developed and in which you could also work if you wanted to write your own kernel.

When the system runs through its initialization steps, some tasks need to be handled before others, for example we need to get the memory access right before everything else, and we need to establish methods for handling interrupts and faults before we can start talking to hardware components. Here’s an overview:

**Memory.** This is the first thing the OS will face: The boot loader will load the kernel into RAM where the kernel will start executing. But where precisely is that? And how do we have to compile the kernel so that it will work properly in the RAM areas we load it to? (Think of function calls and references to data addresses: We must know at compile time where the kernel’s code and data will be located.)

There are different modes of memory usage, and we need our OS to use Virtual Memory so that we can protect the kernel from the processes and the processes from each other.

We need to check in which mode the CPU runs when the kernel is loaded, and then we have to create a transition to the mode we want to use: The latter one is called Paging, and it provides all the protection mechanisms we need while using the memory in a flexible way.

So, the first implementation chapter deals with loading the kernel and setting up the memory in such a way that the next initialization steps of the OS can work properly. Then we switch to paging which is a needed preparation for introducing processes.
Interrupts and Faults. Before we can start using the hardware, we need to deal with interrupts: several devices use interrupts to tell us that they have completed some activity, and there are components like the clock chip which regularly generates an interrupt.

Closely related to interrupts are faults: they occur when the operating system (or a process) tries to do something that’s impossible, e.g. access a memory address that does not exist or for which access is forbidden. We need to treat these, too, because if we don’t, then any fault will just make the whole system halt.

The main difference between interrupts and faults is that faults occur as a direct consequence of some specific instruction that our code executes. In that sense they are synchronous. Interrupts on the other hand occur without any connection to the currently executing instruction, since they are not triggered (immediately) by our code but by some device. That is why they are called asynchronous.

Handling an interrupt and handling a fault are very similar tasks, so we deal with both in the same chapter. We present a framework which lets us supply handlers as we need them, so for example the concrete interrupt handler for the IDE disk controller will be shown later, but it will use the code that’s presented here.

Basic Hardware Support. Once we’ve established the general interrupt handling mechanism, we can start setting up the hardware. We begin with the clock chip (which generates timer interrupts) and the keyboard (which generates an interrupt each time you press a key). We’ll look at further hardware components in later chapters, for example the code for floppy and hard disks will follow after the filesystem chapter.

Processes. Here we introduce one of the most important data structures of the operating system, the process control block (PCB). We also have to revisit the virtual memory code and introduce a mechanism to switch between different page tables—every process will use its own one, since every process has its own virtual memory.

The following chapter discusses the necessary changes for supporting threads which share an address space if they belong to the same process.

In the chapters that deal with processes, threads and the scheduler you will also see how the system can make a context switch which happens when one process is interrupted and another one continues executing.

System Calls. The operating system does not only guarantee that the memory areas of the kernel and all the processes are protected against one another, it also disables direct access to the hardware. So if a process wants to open and read a file, it needs a mechanism for calling a kernel function which does just that. The standard mechanism for calling OS functions is a system call. The system call interface somewhat resembles the interrupt handling interface which we’ve seen earlier. We only explain how the general mechanism works—concrete system calls will be implemented where we need them, for example in the filesystem chapter.
**Filesystem.** The UliX filesystem code provides us with a virtual filesystem. With it, we can use several kinds of drives (we'll show the code for floppy disks, hard disks and a virtual device we've named the “serial hard disk”), and we can support several logical filesystem formats: UliX has a driver for the Minix filesystem which was introduced for the Minix operating system. Linux can use Minix media as well which makes it easier to prepare the disk images we use with UliX. Code for other logical filesystems (e.g. FAT or NTFS) could easily be added.

We will now present the overall layout of the UliX kernel code. Basically every kernel has to do some essential setup (such as initializing RAM and other components of the machine), then activate interrupts and the process system, create a first (init) process and transfer control to that process (which will start further processes). UliX is no different, but where it differs from other operating systems is that we're going to store (almost) everything in one single C source file, ulix.c. If you take a look at other systems you will find a huge collection of source (*.c) and header (*.h) files and several makefiles which turn the source files into object files (*.o) independently before linking them all together into a single binary. We'll also link the kernel binary because we have some assembler code in the file start.asm that is translated with nasm, but all the rest goes into ulix.c.

The literate programming approach allows us to combine everything without losing the overview: the pages you look at right now structure the code well enough.

Since C requires functions to be defined before they are called and user-defined types to be declared before they are used in function prototype definitions, we need to make sure that all symbols are known at the right time.

- We start with two code chunks ⟨constants 112a⟩ and ⟨macro definitions 35a⟩ where we put all pre-compiler (#define) statements.
- Then follow ⟨public elementary type definitions 45e⟩ (mostly stuff like typedef unsigned int uint32_t) and ⟨type definitions 91⟩ (for structures). The distinction is necessary since structure definitions use elementary types.
- Next come the ⟨function prototypes 45a⟩ and the ⟨global variables 92b⟩: the latter ones often receive their initial values at the point of declaration.
- Finally, the real code starts: with all kernel functions in the chunk named ⟨function implementations 100b⟩, and at the end of the file you will find the ⟨kernel main 44b⟩ function (main44b(1)).

Most of these code chunks “exist” twice, for example ⟨constants 112a⟩ and ⟨public constants 46a⟩. The code chunks with a public prefix will later be included in the user mode library’s header file (ulixlib.h) or the implementation file (ulixlib.c), and this trick allows us to automatically keep data structures and constants synchronized between kernel and user land, and it saves us from having duplicate code chunks for standard functions like memcpy which are used both in kernel and user level land.

So this is the basic structure, with most of the kernel code collected in just one C file, ulix.c:
2.1 The main() Function

The main() function of the Ulx kernel brings the system up and starts the first (user mode) process, an init program which launches several copies of a login program that in turn start simple shells. This happens on a number of virtual terminals (text consoles), allowing tests with several logged-in users.

Before all other hardware we initialize the serial port since we use it for debugging purposes; the function debug_printf sends information to the first serial port, and when we run Ulx in the qemu emulator, we can grab that output and display it elsewhere—even before it is possible to write to the (virtual) screen. Note that you will not find code lines with debug_printf statements in the PDF version of this book, but if you download the sources, you can see them directly in the Noweb file (and also in the generated source code). In order to enable this debugging, the DEBUG macro must be defined via #define DEBUG (see Chapter 16.3.1).

Setting up the memory is a complex task which we describe in detail in Chapter 4.4. Next we can start accessing the memory buffer of the graphics card to display messages...
on the screen, we will define the \langle setup video 337c \rangle chunk in Chapter 10.2.

Since ULIx can be (and has been) used for Bachelor’s or Master’s thesis projects, we provide a mechanism to integrate the OS code with a module that is implemented by the student; during kernel initialization we will call the

\langle function prototypes 45a \rangle =

\begin{verbatim}
extern void initialize_module();
\end{verbatim}

Defines:
initialize_module, used in chunk 44b.

function which may perform further initialization tasks.

\langle initialize system 45b \rangle =

\begin{verbatim}
\langle install the interrupt descriptor table 146d \rangle
\langle install the fault handlers 148b \rangle
\langle install the interrupt handlers 139b \rangle
\langle install the timer 339a \rangle
\langle enable interrupts 47b \rangle
\end{verbatim}

For initializing the filesystem, we first detect the hardware and then print the (currently static) mount table:

\langle initialize filesystem 45c \rangle =

\begin{verbatim}
fdc_init(); ata_init(); // register floppy and hard disks
print_mount_table();
\end{verbatim}

Uses ata_init 534b, fdc_init 552c, and print_mount_table 406.

When everything is prepared we can finally enter user mode: We enable the interrupts, load the init program and start it as the first process.

\langle start init process 45d \rangle =

\begin{verbatim}
printf("Starting five shells on tty0..tty4. Press [Ctrl-L] for de/en keyboard.\n");
start_program_from_disk("/init"); // load flat binary of init
// never reach this line!
\end{verbatim}

Uses printf 601a and start_program_from_disk 189.

With the first process being active, initialization of the system is finally complete.

\section{2.2 Type Definitions}

Before we introduce the first data structures, we define some elementary data types which make our code more readable. For example, the C language has no specific "byte" and "boolean" data types. Instead, an unsigned char is used whenever bytes or booleans are needed. We also define a "word" type.

\langle public elementary type definitions 45e \rangle =

\begin{verbatim}
typedef unsigned char byte;
typedef unsigned char boolean;
typedef unsigned short word;
\end{verbatim}

\[45e\]
Along with the boolean datatype we also provide constants for the two standard values 1 and 0 (note that C regards any non-zero integer value as true). NULL is a null pointer.

```c
#define true 1
#define false 0
#define NULL (void*) 0
```

Defines:
NULL, used in chunks 120, 121, 146a, 164a, 258b, 367b, 369c, 463a, 467–70, 475a, 484e, and 607b.

Sometimes we create data structures with differently-sized components, and in those cases we want to show clearly how “big” each element is. So we also define uint*_t types which are standard in many systems, e.g. in the Linux kernel:

```c
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;
typedef unsigned long long uint64_t;
```

```c
typedef int size_t;  // short names for "unsigned int",
typedef unsigned int uint;  // "unsigned long" and
typedef unsigned long ulong;  // "unsigned long long" (64 bit)
typedef unsigned long long ulonglong;
```

Defines:
size_t, used in chunks 420c, 429b, 594, and 596.
ulong, used in chunks 109, 340e, and 341.
ulonglong, used in chunk 534.

Memory addresses in our code are always 32 bits wide since Ulix is a 32-bit operating system. We introduce an address type:

```c
typedef unsigned int memaddress;
```

Defines:
memaddress, used in chunks 100, 103, 105b, 108, 111b, 113b, 115d, 151c, 161, 166a, 169, 170, 172a, 173a, 175, 192b, 197, 211–13, 228b, 231, 232, 234b, 255a, 257–59, 279c, 289–91, 515a, 567c, 568b, and 604a.

### 2.3 Assembler Code

As we will occasionally have to use assembler statements and the standard command in the GNU C compiler gcc is __asm__, we define a shorthand:

```c
#define asm __asm__
```

We have created an assembler pre-processor which replaces code that has the form of the left side with code that looks like the right side:
2.4 The User Mode Library

Besides the kernel we also need a user mode library which provides some standard features for user mode applications.

The library code consists of two files: `ulixlib.c` contains the implementations of the library functions, whereas `ulixlib.h` provides declarations which have to be included both in `ulixlib.c` and any program that wants to use the library functions. In this section we introduce the code chunks that the library files are made of.

The header file shares some constants and type declarations as well as some generic function prototypes with the kernel:

```c
asm {
    starta: mov eax, 0x1001 // comment
    mov ebx, 'A' // more comment
    int 0x80
}
```

This allows usage of the Intel assembler syntax (without changing the normal compilation process which uses AT&T syntax), it also enables us to add comments in the code, and the new syntax is closer to C.

The pre-processor also understands `asm volatile`. What it cannot cope with is variable / register usage; thus, occasionally there will be appearances of the less readable standard assembler syntax.

Note that it does not change the number or position of code lines. The source code for the assembler parser is shown on page 624 ff.

### 2.3.1 Turning Interrupts On and Off

We will often have to disable and re-enable interrupts. The assembler instructions are `cli` (clear interrupt flag; disables the interrupts) and `sti` (set interrupt flag; enables them). Instead of writing `asm("cli")` or `asm("sti")` (which would force you to remember which of the commands turns the interrupts on or off) we provide code chunks for them:

```c
 ⟨disable interrupts 47a⟩≡ (282c 352 353 357b 383a 390 533b) [47a]
 asm ("cli"); // clear interrupt flag

 ⟨enable interrupts 47b⟩≡ (45 151c 282c 290b 324b 352 353 357 383a 384b 390 533b 610a) [47b]
 asm ("sti"); // set interrupt flag
```
2.4.1 Functions of the Library

Some functions are needed both in the kernel and in the user mode library. We put their implementations into the ⟨public function implementations 455a⟩ chunk (which is included both by the kernel and the library C files) so that we need not present the code for `memcpy`, `strncpy` and other standard functions twice.

So, whenever you see a code chunk that starts with “⟨public”, you know that it contains declarations or code which will appear both in the kernel and in the user mode library.

2.5 Next Steps

In the following chapter we introduce the theory of memory management—that’s a requirement for doing any further steps, since the next Ulix code chunks show you how we load the kernel from the boot manager. But in order to do that, we need to first think about how we want to use the physical memory. Thus, Chapter 3 gives you all the needed theory, and then the following Chapter 4 explains the boot process and the Ulix approach towards memory usage.
Managing Memory

Processing power (that is, the computing cycles of the CPU) and memory are the two most important resources for any machine. In the next chapter we will describe how to boot the Unix operating system, and that procedure will include copying the kernel into the computer’s main memory. We need to know where to put it and how to continue using memory.

Doing that properly requires some understanding of the general concepts of memory management and also of the concrete mechanisms provided by the target CPU which, in our case, is the Intel i386. In this chapter we start with an overview of the memory management theory, and the following chapter will present the implementation details of the management solution which we have chosen.

Memory management is all about the question: “How can we make the best use of the available physical memory?” When a computer only needs to execute one single program, there is not much to do. But the invention of multi-tasking led to the task of providing several processes with sufficient memory to store their code and data. The extra requirement of memory protection (processes shall not access the memory areas of other processes or of the operating system) further complicates the task.

There are also many similarities between memory management and filesystem management (which we discuss in detail in Chapter 12): In both cases, a fixed-size resource (the physical memory or the collection of sectors on a disk) needs to be shared.

Note that from a process’ point of view, memory is a direct resource and is needed for running the process: If the process’ program code is not available in memory (at least partially), it cannot be executed, because the CPU can only execute instructions that are located in RAM. On the other hand, disk space is not something that a process will (directly) need: while access to certain files may be necessary for running a program, it is not needed permanently. But if we look at disk space from a file’s point of view, we could
say that a file (in order to exist and allow access to it) needs disk space in a similar way as a process needs memory to run. From that point of view, we can compare a process which has no memory with a disk file that was moved to tertiary storage (e.g., a magnetic tape that is part of a tape collection).

These are some of the concepts that appear in both memory management and file systems:

**Partitioning of resources:** On a system that will handle several processes in parallel, memory must somehow be partitioned so that each process can use a fraction of the RAM. Individual cells of memory are exclusive: They can hold precisely one byte of information, and it has to belong to a specific process (or the operating system itself) at any given time. It may not be necessary that a process has memory throughout its whole lifetime (for we will see that concepts such as swapping and paging allow data to be stored on the disk for a while), but at least in those moments when a process is actually executed by the CPU, it will have to be given some memory. It may be useful to limit the maximum RAM that a process can access at any given time.

Similarly, if a system allows several files (and possibly directories) to be created, accessed and modified on the disk, disk space has to be partitioned in a way that the disk can hold all these files and present simple means to look up files on the disk and access them. The smallest unit of storage would in theory also be a byte, and such a byte (now meaning the fixed location on disk) can only belong to one file (or to the filesystem metadata) at any given time. As in the memory situation, it may be useful to define a maximum filesize so that no file uses too much of this resource, though most filesystems that implement such limits do this on a per-user basis and not on per-file basis—limiting disk usage per user (or per user group) is called a *quota system*.

**Access control:** Access to memory locations should always be exclusive to one process (or the operating system), otherwise a process could read or even modify the process memory of a different process which is not advisable, because it would be a source of instability or security problems.

Access to files is also often handled in a way that makes it exclusive to a file owner, typically the file creator (or perhaps some other users, depending on the access concepts a specific filesystem may have). And from the view of processes it may be necessary to restrict file access to only one process (even in a situation when several processes belong to the same user who is also the owner of the file), so that no errors can result from parallel access to a file.

**Free space management:** Memory and disk usage must be handled dynamically, because processes newly appear and are removed from the system all the time, their needs for memory may change during the process runtime, and also files can be created and deleted as well as grown while the system is active.
In many memory management schemes there will be a list of free memory locations. We will see that it is not useful to grant memory access byte-wise, memory will often be partitioned into equal-sized smallest chunks of memory that can be assigned to a process or removed from it. If we call these smallest chunks (say, of size 1 KByte) memory frames, then there will need of a “free frame list” that knows which frames are currently unused.

In the same way disks aren’t typically accessed byte-, but block-wise, a block being a fixed size segment of the disk space. Note also that read and write operations on the raw disk device always transfer a whole block of data and not a single byte (which is why they are called block devices as opposed to character devices. We will need a “free block list” in order to know which blocks are still available for file storage and which are not.

Methods for administering such free frame lists and free block lists will be similar.

### 3.1 Contiguous Allocation

In this section we present the most simple methods to distribute memory among processes and disk space among files. Contiguity means that a process gets to use a contiguous (connected) area of memory, there are no “holes” in it which would be memory areas assigned to a different process or not assigned at all. If memory did have such holes, a process would have to keep track of which memory regions it can use and which not.

#### 3.1.1 Fixed Equal Size Partitioning

Consider a computer that has 1 GByte of RAM. If we divide this memory into 1-MByte-sized partitions, then we get 1024 such partitions, some of which will have to be reserved to the operating system itself (see Figure 3.1). Assuming that 1000 “unused” partitions will remain, such a system would allow for up to 1000 processes to be started and held in memory in parallel. Each of the processes will then have its own 1 MByte memory partition, meaning it can use up to this 1 MByte for storing its own program code, stack and data.

![Figure 3.1: Fixed equal size partitioning is the simplest but also the least flexible approach to partitioning memory or disk space.](image)
Obviously this method is not very flexible and it limits the possible usage of the system in two ways:

- No more than about 1000 processes can be run in parallel. If there was need for, say, running 2000 or more processes at the same time, then the whole system would have to be reconfigured (with smaller, but more memory partitions) and completely rebooted.

- No more than 1 MByte of RAM can be given to a single process. If a program required more than that, say 2 MByte or more memory, again the system would have to be reconfigured.

- It would be completely impossible to change the system parameters in either direction, i.e., allow for more than 1000 processes and some of them using more than 1 MByte RAM.

Now, in the same way consider a harddisk of size 1 GByte and a similar partitioning scheme that would allow 1024 (minus a few) files of up to 1 MByte size to be written to this disk. The same problems as in the memory example would occur: There would be a filesize limit as well as a limit on the number of files, and changing the filesystem structure in order to either allow more or larger files would require the disk to be newly formatted, and a change would only increase one of these numbers while reducing the other.

This simple approach is called “fixed equal size partitioning” in both the memory and the harddisk case, and besides the limitations already discussed it leads to a problem called *internal fragmentation*: While the RAM is fully split into partitions, i.e., there remains no unpartitioned and possibly unusable memory (that would be external fragmentation), a lot of memory will go unused, e.g., when a process runs that needs only a few kilobytes of RAM but still gets the whole 1 MByte. There is no way for other processes to claim some of this unused memory because the fixed partitioning forbids this (see Section 3.1.6).

It is an example of contiguous allocation methods: Contiguity means that all the parts of a process’ memory (or of a file on disk) are stored in consecutive frames/blocks, and also in order. So no jumps to other memory locations or disk blocks are necessary when reading the whole file (or the process’ whole memory) from the first to the last byte. The opposite of this in non-contiguous allocation, and we will get to that approach in Section 3.2.

Note that we use the term “disk partition” in a non-standard way; we do not mean the logical partitions into which a disk is separated on today’s standard computers in order to create several logical volumes (in Windows language: *drives*) each of which is formatted with its own filesystem. For simplicity we assume that a harddisk contains exactly one filesystem and that this filesystem uses all of the disk, as it is the case on floppy disks and (some) USB sticks. See section 3.1.4 for a few words about the classical understanding of “disk partition”.

---

*internal fragmentation*
3.1.2 Fixed Variable Size Partitioning

The partitioning scheme that gives all partitions equal size causes the two limitations in file size and file number. A little more flexibility is introduced when we dispense with the equality condition: That leads us to a new method of creating fixed partitions, but of varying sizes (see Figure 3.2).

![Figure 3.2: Fixed variable size partitioning gets rid of the file size and file number limitations, but still the partitioning parameters cannot change once the system uses the partitions.](image)

It is just a small alteration, but it already improves the situation a lot: In the memory case, if a process is associated with a memory partition and it wants to extend its memory usage beyond the current partition’s limits, it can be relocated to a different (larger) partition, and on the other hand many more processes can use the system if there is a good mix of processes with small and large memory demands. Note that this partitioning scheme is still fixed: At system boot-time, the memory partitions are created and cannot be modified until the next booting (and a modification is likely to require a recompilation of the operating system kernel).

In the same way a filesystem with fixed partitions will profit from this modification by allowing both more and (some) larger files. The strictness of the partitioning applies here as well: Once the disk has been formatted, the partition (i.e.: maximum file) sizes can only be changed by reformatting the whole disk.

3.1.3 Dynamic Partitioning

A lot more freedom in memory or disk space allocation is possible if the partitioning becomes fully dynamic: This means that no partitioning occurs at system initialization or while formatting the disk, but instead partitions are created as need for them occurs.

This has the effect that the operating system must carry out a lot more administrative work. For example, keeping an overview of free areas of memory becomes more complicated, because whatever data structures are used for the memory or disk allocation, they are now dynamic.

3.1.3.1 Free Frame Lists

The simplest approach is to keep a list of free locations. This list will be called a free frame list in memory management or a free block list in filesystems. Typically there is a smallest possible fragment that can be allocated, called a frame or block, and free space
managements only deals with these frames/blocks. The smaller the frames or blocks are, the more of them exist and have to be handled by the free frame list.

One approach is to have a linked list that contains descriptions of free areas, e.g. a start address and a length for each one. In the list each entry points to the next entry when working with pointers. In order to find a free area of a given size an algorithm will walk through this list and stop when it finds an area of sufficient size. For this purpose it may be necessary to scan the whole list if (in the worst case) the only fitting area is at the end of this list. When a number of previously free frames is allocated to a process (or blocks to a file), the list has to be modified,

- either by removing the entry if the whole lot of contiguous blocks are allocated,
- or by modifying the entry if only a few of the blocks are allocated, and they are located at the beginning or end of the area described by this entry,
- or by splitting the entry in two parts, if (for whatever reason) a section taken from the middle is allocated, leaving free areas in front of and behind them.

If the used space is later released, it must be added to the list again, possibly creating a list entry that has to be merged with entries describing directly neighboring areas.

Another possibility is to work with bitmaps: For each frame/block a bit in this bitmap defines whether it is free (0) or in use (1). Here no complex list administration (with the mentioned splitting and mergers of list entries) is required, however allocation and release of blocks lead to modification of several bits in the bitmap, and looking up free space of a given size means finding a number of consecutive 0-bits in the bitmap.

Note that it does not matter at all whether we think of memory frames or disk blocks, the concepts are identical. Differences will however appear when thinking of storage of these lists or bitmaps: In the memory case it is obvious that the list must also lie in memory for quick access. In the filesystem case it might make sense to store the list in memory (and not on disk as well) in order to speed up the lookup of free areas—but depending on the size of the free block list, it may be too large to keep all of it in memory.

3.1.3.2 Allocation

When working with dynamic allocation of free areas, there will typically be a choice among several free areas which are of sufficient size, and the procedure for choosing one of them will have consequences both on performance and on external fragmentation (an increasing number of small unallocated areas): If the decision algorithm is very complex, allocation will always take a lot of time; if it is simple, there will be many small unpartitioned (not allocated) areas which are too small to be useful anymore, so this external fragmentation will lead to memory or the disk filling up more quickly than necessary.

On the following pages we will present five simple approaches to allocation called first-fit, next-fit, best-fit, worst-fit and quick-fit; and after that a more advanced concept called the Buddy System will be introduced.

First-fit This strategy picks the first free area of sufficient size. It has the advantage of being fast, because once an acceptable area has been found, the system looks no fur-
ther. On the negative side, first-fit leads to a lot of fragmentation and continuously reduces big areas, so that processes which start later and need a big area cannot run.

**Next-fit** This is a variant of first-fit. The difference is that after every allocation the system keeps in mind the position of this allocated area. The search algorithm then continues immediately behind this area. That way all of memory is being used, whereas first-fit might use only areas in the low-address range if demands can be fulfilled there.

**Best-fit** The idea behind best-fit is to allocate space in the smallest possible area where the process fits. Assuming that what is left after the allocation is likely to become unusable (because it is too small), this approach tries to minimize the waste of space.

**Worst-fit** Exactly the opposite of best-fit, worst-fit searches for the biggest free area and allocates space within it. From the perspective of the remaining space, this maximizes the size of the new free area that is left after allocation, hoping that it will still be large enough to allow for further allocations.

**Quick-fit** is a combination of a first-fit approach with fixed partitioning. A part of the available memory is partitioned into areas of some varying sizes which are often requested. (What is often requested must be known from experiences with memory allocations.) So there will be lists of free partitions of some standard sizes, say, 1 KByte, 2 KByte, ..., 16 KByte. If a memory request of one of those sizes occurs, the system will first try to satisfy it with one of the partitions in the corresponding list. Only if that fails, memory will be allocated from the unpartitioned space using first-fit.

### 3.1.3.3 Buddy System

The *Buddy System* [Kno65] assumes that we start with a free memory area whose size in bytes is a power of 2. The system reacts to memory requests of arbitrary sizes by repeatedly dividing a free chunk of memory in two halves until a chunk becomes available which is just large enough to satisfy the request. An example illustrates this system better than the description: Let’s assume that 1 MByte of memory (1024 KByte) is available at the start. This memory chunk is not partitioned. If a request for 90 KByte arrives, the Buddy System takes the following steps:

- Since the 1024 KByte chunk is too large, it is split into two 512 KByte chunks.
- 512 KByte are still too large, so the first of these chunks is split into two 256 KByte chunks.
- Again, 256 KByte are too large, so another split takes place, turning the first of the 256 KByte chunks into two chunks of size 128 KByte.
- The first of the 128 KByte chunks is chosen since it can satisfy the request. It is marked as used.
Figure 3.3: The Buddy System repeatedly splits available space in halves. The chunks with bold face descriptors are chosen for splitting; the green and bold chunk satisfies the request of 90 KByte.

Figure 3.3 shows how the Buddy System partitions the memory step by step when trying to satisfy this request. Afterwards, there are three free chunks left, their sizes are 128 KByte, 256 KByte and 512 KByte. If another request for 90 KByte arrives, the free 128 KByte chunk is chosen; in case of a 40 KByte request, the 128 KByte chunk would be split again. The way in which memory is partitioned can also be represented by a tree; Figure 3.4 shows the tree which corresponds to the situation described above.

Figure 3.4: The tree representation of memory that was partitioned by the Buddy System shows available memory chunks in the tree’s leaves.

When a used chunk of memory is returned and its equal-sized direct neighbor (in the tree) is a free leaf, these two leaves are joined to form a new leaf of twice the size; this process continues, possibly all the way up to the root of the tree. The tree view makes it easier to see which chunks can be joined and which cannot. Figure 3.5 shows an example in which joining is impossible: the second and third 128 KByte blocks are not direct neighbors, they would have to be joined with their left and right neighbors first.
3.1 Contiguous Allocation

<table>
<thead>
<tr>
<th>128 KB</th>
<th>128 KB</th>
<th>128 KB</th>
<th>128 KB</th>
<th>512 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 KB</td>
<td>256 KB</td>
<td>128 KB</td>
<td>512 KB</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.5: This is an impossible join operation; a tree representation would show that the two 256 KByte blocks marked bold are not direct neighbors.

3.1.4 A Few Words on Disk Partitions

As mentioned above, we have not been talking about hard disk partitions in the sense of creating several logical volumes on a disk for use by various operating systems (e.g. a Windows and a Linux partition) or for structuring the disk so that different data can be stored on different partitions (e.g. “drives” C: and D: for Windows or partitions /, /home and /usr for Linux)—now we do, because this kind of partitioning is another example for contiguous allocation with flexible size. Most disks have a partition table as created by Windows, Linux, DOS and other operating systems when initializing a hard disk. (The BSD operating systems use a different method to partition disks, calling the partitions slices and the partition table disklabel.)

A classical partition table puts no limits on the sizes of individual partitions, but allows only up to four (primary) partitions for whose administrative data it reserves space in the first blocks of the disk. There, you basically find the start address and the length of each partition. If more than four partitions are needed, one of the four must be set up as an extended partition that holds an additional partition table and the logical partitions that reside inside the extended one.

If we ignore logical partitions, we see that this is a simple implementation of dynamical contiguous allocation with flexible size; partitions can be created and deleted, each partition has to be contiguous, and in principle the partitioning also suffers from external fragmentation: If you start with a 40 GByte disk that is partitioned into four 10 GByte partitions and you resize each of them to 9 GByte, you end up with four unused 1 GByte areas that cannot be used. If there was no “four partitions” limitation, the four free areas could be made into four separate 1 GByte partitions, but never into one 4 GByte one, since these four areas are not contiguous.

In order to change the size of a formatted partition (i.e., one with a valid filesystem on it), always two steps are necessary, with their order depending on whether the partition is to be extended or shrunk: The logical filesystem must be resized and the partition itself must be resized.

- When extending a partition, the operation on the partition (and partition table) comes first. Only when this is completed, can the filesystem size be increased as well so that it grows into the newly available space.

- When shrinking a partition, the filesystem has to be modified first, because for example files residing in the parts of the partition that is to be removed must be relocated to a different area on the partition first. Only then can the partition itself be resized (making the removed parts unaccessible to the filesystem).
Note that modifying a filesystem size requires more than (possibly moving files from an area that is to be removed and) changing the information about the partition size in the partition’s metadata, for example on a Unix filesystem the free block bitmap has to be grown or shrunk as well in order to correspond to the changed number of blocks.

### 3.1.5 Segmentation

Segmentation is a memory management technique that requires support by the processor. In general, an address consists of a segment number and an offset (an address that is relative to the physical start of that segment). Segments may (but need not) have a size—if they have, the CPU can check whether access to an address exceeds the boundary of that segment.

The CPU must know the start addresses (often called *base addresses*) and sizes of the segments. That can be achieved by filling a segment table or by loading special CPU registers. Table 3.1 shows an example with three segments on a 64 MByte machine. Figure 3.6 shows the CPU-internal implementation of address calculation for a machine which is not aware of segment limits.

<table>
<thead>
<tr>
<th>No.</th>
<th>start address</th>
<th>size</th>
<th>absolute range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$0x00000000$</td>
<td>$0x100000$ (1 MByte)</td>
<td>$0x00000000$ – $0x000FFFFF$</td>
</tr>
<tr>
<td>2</td>
<td>$0x00800000$</td>
<td>$0x400000$ (4 MByte)</td>
<td>$0x00800000$ – $0x00BFFFFF$</td>
</tr>
<tr>
<td>3</td>
<td>$0x03F00000$</td>
<td>$0x100000$ (1 MByte)</td>
<td>$0x03F00000$ – $0x03FFFFFF$</td>
</tr>
</tbody>
</table>

Table 3.1: Example of a segment table with three segments.

In the first and the third segment, relative addresses in the $0x000000 – 0xFFFFF$ range are valid (since those segments are 1 MByte large), and in the second segment, relative addresses from the $0x000000 – 0x3FFFFF$ range can be used (4 MByte).

![Diagram](image)

Figure 3.6: Calculating the physical address from a segment number and a logical address is simple: The CPU just adds the segment base address and the logical address.
A complete address is a \((\text{segment}, \text{address})\) tuple, for example, \((2, 0x\text{ABCD})\) would refer to the relative address \(0x\text{ABCD}\) in segment 2, and its physical address can be calculated by adding \(0x\text{ABCD}\) to the segment’s start address: \(0x\text{ABCD} + 0x800000 = 0x80\text{ABCD}\).

If limits are not checked, then there is the special case of overflow (when the sum of relative address and base address exceed the maximum addressable value). Consider for example the case of a CPU with a 32-bit address bus (which allows addresses ranging from \(0x0\) to \(0xFFFFFFFF\). If the base address is set to \(0xE0000000\) and the relative address \(0x40000000\) is used, the processor will calculate the sum \(0x12000000\)—which is not a 32-bit value. Overflow occurs, and the 33rd bit is dropped, the resulting absolute 32-bit address is \(0x20000000\). You will see an application of this behavior in the next chapter.

Figure 3.7 shows the additional integration of a limit check.

![Logical Address Diagram](attachment:image.png)

Figure 3.7: An additional limit value guarantees that segment borders are never overstepped.

We will discuss two concrete segmentation mechanisms in the next chapter: The Intel 8086 processor used a segmentation technique that is still available in today’s Intel-compatible processors for compatibility reasons (starting with the 80286, it has been called **Real Mode**; see Section 4.3.1), and the Intel 80286 and 80386 processors introduced two (similar) improved segmentation models that were and are used in **Protected Mode** (see Section 4.3.3).
### 3.1.6 Fragmentation

When memory is allocated dynamically, that can lead to a waste effect called *fragmentation* that we already mentioned. It comes in two varieties:

**Internal Fragmentation** means that a memory area was allocated that is larger than the actually needed area. For example, in the Buddy System example, the request for 90 KByte was served with a 128 KByte chunk of memory. If we assume that exactly 90 KByte are needed, then an additional 38 KByte were allocated which will not be used by the requester, and they also cannot be used for anything else since they are marked as used (from the allocator’s point of view).

**External Fragmentation** is the effect that can be best seen in the general dynamic allocation systems (e.g., Best Fit or Quick Fit): After a longer sequence of allocating memory chunks and returning them there will be small memory areas which are marked as free, but are too small to be useful. In that case it is possible that the total free memory size is quite large, but even a modest memory request will fail because there is no sufficiently large contiguous chunk of memory.

To summarize the two types, internal fragmentation refers to unused memory that is part of allocated chunks, whereas external fragmentation represents free, non-allocated memory that is too small to satisfy a typical request (see Figure 3.8).

![internal fragmentation: \(\square\) = unused space inside partition, \(\blacksquare\) = allocated](image1.png)

![external fragmentation: \(\square\) = unallocated space outside partition, \(\blacksquare\) = allocated](image2.png)

**Figure 3.8: Internal vs. external fragmentation.**

External fragmentation can be cured via a process that is called *compaction* (also: *defragmentation*) where the memory management system detects the fragments and reorders the allocated memory chunks so that those fragments disappear; after compaction all (or most) allocated chunks will be located right behind one another, and all free areas will have moved to the end of the physical memory (Figure 3.9). But this approach is costly (because it requires intensive memory copy operations) and it is only applicable if the processes which use the memory can cope with relocation of their memory.
before compaction (external fragmentation)

A B C D E F G

after compaction

A B C D E F G

Figure 3.9: Compaction cures external fragmentation but is costly and needs to be repeated whenever the fragmentation level is high again.

3.2 Non-Contiguous Allocation

So far we have seen several examples for contiguously assigning memory or disk space to processes or files. That allows for a very simple handling of accesses, because only an absolute start address and the length of a (memory or disk) partition must be known.

However, since this leads to strong limitations in usability, all modern operating systems use a more flexible approach both for process memory and files, assigning memory frames and harddisk blocks non-contiguously.

Non-contiguous allocation makes things more complicated, and for process memory it is worse than for files:

- If a file consists of several, non-contiguous blocks which are spread all over the disk, there has to be a list of blocks that tells the operating system where to find the data. When trying to read a specific byte from the file, the address within the file has to be translated into a disk block and a relative address inside that block. It also means that reading a file from beginning to end can no longer be achieved by reading several disk blocks in their natural order, but the operating system has to jump from one location to another all the time, so several disk head movements are involved which slows the access.

- With memory things become even more complicated: Here also some kind of table is needed that will be used for address translation, but memory access is different from file access. A program performs a lot of memory access operations: every (absolute) jump to another instruction in the program code, every direct data access (where the content of some memory address is loaded into a CPU register) and similarly each indirect memory access (e.g. `mov [edx], eax`) works with an absolute address.

A process could be told the absolute address ranges of the memory locations it was given access to, imagine it has a list like this:

1. region 1: 0x10000–0x10FFF (4 KByte)
2. region 2: 0x14000–0x15FFF (8 KByte)
Assume further that the program code is 6 KByte long and the rest of the memory will be used for data (we ignore the stack in this simple example). Then the program code will have to be split between the first and second region, and the data between the second and third one. (For this example we also ignore that it might be a better approach to store all of the program code in the second region and use the first and third one for data, especially since the program might not know the precise number and sizes of partitions before it actually gets them.)

If there is a jump instruction in the program code that leads from the front part of the code to the rear part, it will cross region borders. Also when the programs needs to access its data it has to be aware whether the currently needed data reside in the second or in the third region.

All these problems can be solved with a method called address relocation. When using this system, at compile time a list of address references will be generated. It lists all references to data or instruction addresses that are used within the program code. At load time the system must know the maximum memory demand of the program, assign memory regions and then use the relocation list to adjust the memory references to the concrete region locations.

While this means some overhead during compile and load time, it works quite well, but only for static addresses. If the program dynamically “acquires” memory using some function such as malloc(), then this function will also have to be informed about the memory regions and return proper addresses.

Another problem with this approach occurs when the operating system allows a process to be swapped out to disk (i.e., all of its memory is written to the disk) and swapped in again later: When swapping it back in, the process may be given different regions than before, and then all address references have to be relocated again, this time not only considering the addresses that were stored in the relocation table, but also the dynamically assigned addresses.

The relocation approach makes it hard to protect one process’ memory against accesses by another process, because the address calculation in the relocation step only guarantees that static address references are fixed at program start; the program would however be free to access any part of the memory unless the operating system somehow checked each memory access against the region list for this process. The simple start and length registers from the contiguous case would no longer be sufficient, because there are possibly a lot of regions if a process uses much memory.

Note that for files a similar scheme can be adopted, and it causes much fewer problems: typical operations on a file are seek, read and write operations, and they will require translation of linear file positions (thinking of a file’s bytes as numbered from 0 to \( n - 1 \) for a file size of \( n \)) to absolute disk addresses inside the disk partitions. This translation occurs with every single access to the file. It could be avoided by also using some kind of address relocation as in the memory case, but the gained performance would not be worth the extra effort, because address translation is fast in comparison to disk access.
3.2 Non-Contiguous Allocation

3.2.1 Virtual Memory

A virtual memory is an abstraction of physical memory. Roughly speaking, a virtual memory is an array of memory cells. Usually the size of the virtual memory corresponds to the maximum addressable space allowed by the hardware. A computer may handle multiple such address spaces (and therefore virtual memories) at the same time. They are used to encapsulate effects of programs on memory. Briefly spoken, every program has its own virtual memory and no program can (easily) access the virtual memory of another program.

Physical memory is physical, i.e., it consists of hardware circuits that must be produced, bought and installed on the mainboard of the computer. Virtual memory can be created and destroyed on demand. This is its main distinguishing feature from physical memory. Virtual memory is virtual, i.e., it is a construct which exists only in software. A computer can have much more virtual memory than physical memory. In such cases, a mechanism “multiplexes” the available physical memory resources to possibly multiple virtual memories.

In this section we will have a look at how virtual memory can be implemented. We will look at the idea of address translation in Section 3.2.2 and sketch the requirements for virtual memory from a user’s point of view in Section 3.2.3. We will go through the three historic stages of virtual memory development. In essence, these stages reflect the increasing hardware support for virtual memory in computer architecture. You have already seen the first technique (segmentation, in Section 3.1.5) which provides modified addresses by adding a (segment-dependent) base address to logical addresses. Early approaches basically organize physical memory in a slightly more convenient way, but the transparency of this mechanism is naturally limited. Here, we focus on virtual memory implemented with the help of an external memory management unit (MMU). This approach is the most common one and is also the one chosen in the implementation of UNIX which is described in Section 3.2.5.

3.2.2 Address Translation

The term address space refers to a space of addressable units in a computer. Every computer based on the Von-Neumann architecture (named after John von Neumann) has at least one (physical) address space. Its size depends on the size of the address bus. If the computer has 32 bits on the address bus, the hardware can address $2^{32}$ distinct units of memory. If one such unit is a byte, the architecture supports an address space of 4 GByte.

Having 32 bits on the address bus, addresses are 32-bit values between $0x0000.0000$ and $0xFFFF.FFFF$. In physical memory, not all addresses may be backed by real memory circuits on the mainboard. (Access to such an address usually causes a specific type of interrupt on the CPU or returns an undefined value when it is accessed.) If we view the physical address space of a computer it therefore may have “holes” (see left side of Figure 3.10).
But how does the machine “know” where memory circuits are and where not? The hardware internally stores a mapping of parts of the address space to memory circuits in the form of an address decoder logic. This logic is a simple boolean circuit that translates an address on the address bus into one-out-of-\( n \) bits. This bit is used to select the particular memory circuit on the mainboard via its chip select pin. Only a circuit with an enabled chip select pin will load or store data which travels over the data bus. For example, if the address \( 0x0000.0000 \) is put on the address bus, the logic enables (only) the memory chip that is responsible for serving that address (see right side of Figure 3.10).

Not only memory chips can be activated through such a logic. Also external devices can be mapped into the physical address space. Through this mechanism, they can provide their programming registers just like normal memory cells which can be read and written by the CPU using normal load and store commands. This is the basis for memory-mapped I/O (which we do not discuss in this book, except for the video adapter’s screen buffer).

The effect of such an address decoder logic is that the mapping of memory chips to physical addresses is literally hardwired into the system. This mapping cannot easily be changed. Therefore, programming physical memory directly makes it necessary to know the precise whereabouts of the structure of physical memory. This is only advisable where the physical address space is rather small and well-structured. In the old days with less memory, operating systems like MS-DOS could afford to work directly on physical memory: Their programming manuals contained detailed accounts of where RAM and ROM were placed in the physical address space. With today’s 32 or 64 Bit desktop systems this is not advisable anymore. It is far better to use a homogeneous virtual address space which is independent of the precise placing of memory chips in the physical address space. This can be achieved through an address translation step performed before the address decoder logic kicks in.

Address translation needs some form of hardware support. The idea is depicted in Figure 3.11: The virtual address put on the address bus is taken and translated by the hardware using some translation table to a physical address which is fed into the decoder logic to se-
3.2 Non-Contiguous Allocation

Figure 3.11: Address translation for virtual memory.

lect the right memory chip. This translation requires additional hardware/software effort. But from general experience this pays off quickly given the simplicity and homogeneity of the virtual address space. Program execution can now be performed entirely in the virtual space. An additional advantage is that the address translation is flexible: It can be redefined in software.

3.2.3 Virtual Memory Requirements

A virtual memory is a homogeneous sequence of memory cells together with their content. It can be regarded as a “well-behaved” address space with its content. The homogeneity is what makes the address space nice: All memory cells are considered to return well-defined values. So in contrast to physical memory there are no “undefined” regions of storage in a virtual memory.

3.2.3.1 Types of Data

A virtual memory completely defines the memory context of a running application. This means that it has to provide all necessary data for executing the program. Three types of data are commonly distinguished:

1. Program code (also called text). This refers to all instructions to be executed by the CPU.
2. Data. This refers to the contents of all variables used by the program.
3. Stack. This refers to data used to manage subroutine calls.

Usually these different types of data are collected and stored in different regions of the virtual memory.

The data region is further separated into two areas. The first area is for static data. Static data are variables and data structures which are known at compile time of a program and exist throughout the execution of the program. Examples of static data are global variables. The second area is for dynamic data which is usually called the heap. The heap holds...
variables which are dynamically allocated at runtime by the program (e.g., using `malloc` in C or `new` in C++ or Java). Data on the heap usually depends on program parameters which are only known at runtime.

For completeness we note that a certain form of dynamic data is also stored on the stack. Compilers often generate code that stores local variables of subroutines on the stack. This is especially noteworthy for recursive functions. Also, parameters are often passed to subroutines via the stack.

### 3.2.3.2 Address Space Organization

From a user’s point of view we would like to organize virtual memory in a clear and tidy way. In practice, text, stack and data areas are located in virtual memory in a fixed order (see Figure 3.12). Starting at address 0 we find the text area with all program code followed by static data. These areas are both fixed in size throughout the lifetime of the program. The remaining part of virtual memory is divided up between the more dynamic parts: heap and stack. The heap is usually placed right behind static data at the “lower” end of the free space. The stack is located on the opposite side. Note that while the heap grows in an intuitive way towards rising addresses, the stack grows rather unintuitively into the direction of falling addresses. In this way, the free space between heap and stack is utilized in the most effective way since any memory cell can either be used by the stack or by the heap. Imagine the alternative where the stack would have been placed “half way” up the virtual memory just to allow it to grow in the direction of rising addresses. In such a case, the free memory cells could only be used by either stack or heap.

As we see later in Chapter 7, it may be necessary to provide multiple stacks in virtual memory (for the same program). In such a case we try to utilize the virtual memory as efficiently as possible by dividing up the remaining space into equal parts for each stack. This maximizes the distance between each stack.
Looking at Figure 3.12 and especially Figure 3.13 immediately shows a problem which arises with this memory layout: Dynamic data areas can grow to such an extent that they collide with others. In normal circumstances (i.e., one heap and one stack) this is not a problem because the free space between them is very large. As an example, consider the classic 4 GByte of virtual memory, a 20 MByte program (text and data) and initially empty stack and heap. The gap which opens up between them has a size of 4076 MByte. This is quite some memory to allocate in heap and stack. Of course, the probability that heap and stack collide multiplies with the number of stacks. If a collision is not avoided it usually causes strange and hard to track down runtime errors. As we will see later, it is possible to effectively protect from such collisions with hardware support.

### 3.2.3.3 Amount of Useable Virtual Memory

Without any additional help, the amount of effectively useable virtual memory cannot be larger than the amount of physical memory installed in the computer. Fortunately, most programs do not really use a lot of the available virtual memory so that you don’t always have to equip your system with a full (e.g., 4 GByte) main memory. However, using some tricks it is possible to “simulate” more physical memory using secondary storage. The details of this mechanism will become clear later when we discuss page-based virtual memory. The main idea however is to add special information to the translation table and use main memory as a cache for secondary storage. If a part of virtual memory is not in the cache, program execution is interrupted, the missing data is brought into the cache, and the program resumes operation thereafter. Note that if something is brought into main memory in this process, other information may have to be written out of the cache, i.e., from main memory to secondary storage. This performance overhead is the price you have to pay. The advantage of this scheme is that secondary storage hardware is much cheaper than main memory chips. In well-designed systems it is possible to simulate a substantial amount of main memory using secondary storage without much performance overhead.

### 3.2.3.4 Protection of Code, Data and Stack

We often want to make sure that certain memory areas are only used in the specific way they were intended to. For example, the program code of a process should not be modified, but only executed (which requires only read access). On the other hand, data areas must be changeable, but we don’t want a process to treat data as code and start executing it. Some malware works by storing binary code in the stack of a process and then jumping to that code; if the system does not accept a jump to a stack address, this type of attack is impossible. We would also like to differentiate between what we will call user mode (a process executing its own code) and kernel mode (the process executing a service function inside the kernel) and grant access to specific addresses only when the system is in kernel mode.

Thus, it makes sense to have access attributes which allow reading, writing and executing and which may also depend on the current (user/kernel) mode. When we set up the memory for a new process we should be able to tell the system which memory areas can be accessed in which ways.
3.2.3.5 Summary of Requirements

To summarize, here are the main requirements we have for virtual memory from a user’s perspective:

- Virtual memory should provide a homogeneous address space.
- The size of virtual memory should be independent of the size of physical memory in the system.
- Virtual memory should be able to protect different types of data from certain forms of access (e.g., text from being written).
- Collisions of heap and stack should be detected and avoided whenever possible.

If the system provides multiple virtual memories (one for each program), then we have the additional requirements:

- Virtual memories should be protected from one another, i.e., a program running in one address space should not be able to access the other address space and vice versa.
- The physical resources of the system should be distributed in a fair manner between the existing virtual memories.
- Physical memory should be used efficiently. Especially any type of fragmentation should be avoided.

![Schematic view of a refined translation table for virtual memory.](image-url)
As a glimpse on to the implementation of the above requirements we return to the idea of a translation table which was previously discussed in Section 3.2.2, this time with some more details. Figure 3.14 depicts the idea of a translation table with the additional information necessary to implement all above requirements. The table not only holds information about the physical address which belongs to the virtual address. It also contains flags that indicate access restrictions (like read, write, execute) as well as pointers to secondary storage should this memory location be stored there. Note that the figure only gives a schematic view which is very simplistic, even impossible to realize. After all, the translation table must somehow fit into (physical) main memory to be useable. If we need one (physical) memory cell (at least) to store the translation information for every (virtual) memory cell, we could never simulate more virtual memory than we have physical memory available. The main challenge therefore lies in implementing this concept in a way such that the usage of memory as well as the translation time is minimized.

### 3.2.4 Page-based Virtual Memory

All modern operating systems use a virtual memory management mechanism called paging. The idea behind paging is to give each process a virtual memory space that is addressed contiguously and linearly (starting with an address 0 and ending with an address size – 1) and that is partitioned into a set of memory pages. All pages have the same size, say, 1 KByte, and they are mapped to page frames which are equally sized chunks of the real memory. With the help of a page table each access to a virtual address is translated to a real address by first calculating to which page the address belongs, then looking up the corresponding page frame via the page table and finally locating the relative position within that page frame (see Figure 3.15)—the technical details of this approach are what we discuss in the rest of this chapter.

This approach makes compiling an application very easy: All references to addresses, be they jump instructions or data accesses, can be stored with absolute addresses inside the program, and no relocation takes place when loading the program. Instead each memory address will be translated using the page table.

When, for whatever reasons, locations of page frames have to be changed, it only takes a correction of the page table to make sure that the program continues to be runnable. This scheme also allows for individual pages to be removed from memory altogether and stored on the hard disk for later retrieval—this is called paging as well, and it is not to be confused with swapping a process’ memory (meaning: writing all of it to the disk). A process that has some of its pages paged to disk can still be run, whereas a process that was swapped to disk must first be swapped back in before it can resume action. However, the disk space reserved for paging is often called swap space for historical reasons. E.g. the Linux operating system calls paging partitions or files swap partitions and swap files, but it does not implement swapping; it pages.

Although page frames and pages have the same size, they are totally different concepts. A page is a logical unit of virtual memory. Any virtual address resides in some page. A frame is a concrete area of physical memory waiting to hold some page. A large part of
3.2.4.1 Hardware Support

In contrast to segment-based virtual memory, page-based virtual memory needs no hardware support on the CPU, i.e., no special registers. This means that this type of virtual memory can (at least in principle) be implemented with any CPU on the market. In a sense, the hardware support is “outsourced” to a dedicated device called the memory management unit (MMU). The MMU can be thought of as a hardware address translator that sits on the address bus and divides it into two parts. One part between the CPU and the MMU is considered the “virtual address” part of the address bus, the other (between MMU and main memory) is the “physical address” part. When the CPU issues a virtual address onto the address bus, the MMU transparently translates it into a physical address on the other side, i.e., it changes the value of the bits as the address passes through the MMU from one to the other side.

To tell the truth, the MMU doesn’t change all the address bus bits, but only the higher order bits. The $k$ lower order bits remain unchanged. The value $d$ represented by the $k$ lower order bits is called the offset of the address (see Figure 3.16). The idea of this separation is the following: The higher order bits of the virtual address implicitly refer to the page that the virtual address is located in. The $k$ lower order bits then are interpreted as the offset of the address within the page, i.e., the distance from the beginning of the page to the address.

The interpretation of the address in page number and offset has several consequences. The main one is that the size of a page must be a power of 2. If the $k$ lower order bits
represent the offset within a page, the number of addresses in a page is exactly $2^k$. For a value of $k = 10$, a page would contain exactly $2^{10} = 1024$ Bytes. The value of $k = 10$ is a typical value in practice where there are 32 bits on the address bus. This case is depicted in Figure 3.17. It shows that the $k$ lower order bits (those with index 0 to 9) define the offset $d$ and the remaining $32 - 10 = 22$ bits (with indexes 10 to 31) define the page number $p$.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
page number & offset \\
\hline
\end{tabular}
\caption{Structure of virtual address.}
\end{figure}

\begin{figure}[h]
\centering
\begin{tabular}{cccc}
31 & 10 & 9 & 0 \\
\hline
page number (22 bits), $p$ & offset ($k = 10$ bits), $d$ \\
\hline
\end{tabular}
\caption{An address consists of a page number and an offset. This example uses $k = 10$.}
\end{figure}

### 3.2.4.2 Page Descriptors and Address Translation

The central data structure to manage pages in virtual memory is the page descriptor. There is exactly one page descriptor per page in virtual memory. All page descriptors are held within the operating system in a big internal table called the page table. The page descriptor contains all information necessary to locate the contents of the (virtual) page in physical memory, therefore knowledge of the starting address of the page table is the key to performing address translation. So to enable address translation, the page table register $\text{PTR}$ of the MMU is pointed to that starting address (see Figure 3.18). Assuming that there is just one big table, the MMU can now directly locate the page descriptor of page $p$ by doing a small address calculation.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{page_descriptors}
\caption{Address translation using page descriptors.}
\end{figure}
Given the size of a page descriptor to be $x$ bytes, then the page descriptor of page $p$ has the address:

$$\text{PTR} + p \cdot x$$

As mentioned above, a page descriptor is a data structure that holds all necessary information to manage the associated virtual page. Here is an overview over the types of information that can be stored in a page descriptor of page $p$:

- Since the page descriptor is used to perform address translation, it must contain a pointer to the physical page frame of page $p$. The MMU adds the offset of the virtual address to this pointer to yield the actual physical address of the virtual address in question.

- Since page frames act as a cache for the contents of pages, certain management bits must be present to handle cache contents. Recall that main memory is regarded as a cache for pages stored on secondary storage (see Section 3.2.3.3). The first such bit is the presence bit (P bit). The P bit indicates whether or not the page contents are present in main memory.

- The next management bit is the reference bit (R bit). Roughly speaking, it indicates whether or not the page descriptor was referenced within some period of time. The R bit is set by the MMU with every access to the page descriptor in main memory. Technically speaking, the reference bit is not actually an essential management bit of the cache, but rather a bit which is used to optimize the cache performance. This will be discussed later in Section 9.3.

- A vital cache management bit is the dirty bit (D bit), sometimes called written bit. It is set by the MMU whenever the contents of page $p$ are written. The D bit is important since it solves the cache coherence problem: Contents of the cache (i.e., main memory) and secondary storage can diverge if main memory is written and secondary storage not (due to performance reasons). The D bit indicates exactly when this divergence exists. Pages which have diverged in main memory eventually have to be made coherent with secondary storage again.

- The page descriptor also contains protection bits used to manage the type of access allowed to this page.

- The page descriptor usually also contains several multi-purpose bits which can be used by the operating system for different means.

Ignoring protection and multi-purpose bits, this is what a page descriptor could look like in code:

```c
typedef struct page_desc_struct {
    void *frame_addr;       // address of page frame for this page
    unsigned int present : 1; // presence bit
    unsigned int referenced : 1; // reference bit
    unsigned int written : 1; // dirty bit
} page_desc;
```

[72]  

(tentative declaration of page descriptor 72)
3.2 Non-Contiguous Allocation

To summarize, we now recall how a successful page translation finally happens (see Figure 3.19):

1. The CPU accesses a virtual address \( v \) on the address bus. The virtual address consists of a page number \( p \) and an offset \( d \) in the page.
2. The MMU uses the page number \( p \) and the base address of the page table (stored in \( \text{PTR} \)) to locate the page descriptor of the page.
3. Using the page frame number \( k \) stored in the page descriptor, the MMU adds the offset \( d \) to form the final physical address in main memory.

What can potentially go wrong during page translation? The simplest error condition is that the virtual memory location doesn’t (yet) exist in physical memory. This means that the page table doesn’t know where it should point to. This information is encoded in a special null page descriptor. If the MMU tries to translate an address and finds such a null page descriptor in the page table it signals an interrupt to the CPU which must be handled immediately.

The next error condition concerns the protection bits. The MMU checks whether the current access context is allowed by the protection bits. For example, if the CPU wants to write something to a virtual address and the protection bits don’t allow write access, then again the MMU raises an interrupt with the CPU.

The final error condition which we discuss here refers to the fact that main memory is just a cache for secondary storage: a cache miss may happen. What is a cache miss in this context? It means that the page accessed by the current instruction exists but it currently isn’t in main memory (the cache). This is indicated by the presence bit (P bit) in
the page descriptor. If the P bit is not set, an interrupt is raised by the CPU. In effect the interrupt handler must try and load the page contents back in to main memory so that the application that wished to access the page contents can continue to operate. More details on how this works will follow later.

### 3.2.4.3 Page Descriptor Trees

In contrast to the naive address translation described at the end of Section 3.2.3 where we had one entry in the translation table per virtual address, the idea of pages reduces the size of the page table dramatically. The larger the page size, the smaller the page table because translation information and protection bits etc. are stored per page. However, page tables still have considerable size. The problem is partly a result of the memory layout sketched in Section 3.2.3.2 because code, data and heap reside on one end of virtual memory and the stack on the other end. This means the page table must always cover all pages in virtual memory. As an example, imagine you need eight bytes for each page descriptor (which is not much), a 12-bit offset for pages (giving a page size of 4 KByte, rather large) and a

![Diagram](image-url)

**Figure 3.20:** Example of a multi-level page table that maps individual virtual addresses to physical addresses (page size: 1 byte).
32-bit address bus. In total you have $2^{20}$ entries in the page table, each uses eight bytes, yielding a page table size of 8 MByte. Small computer systems with only 16 or 64 MByte of physical memory could only hold one or two page tables (at most) since additionally the contents of pages also must be stored in main memory.

Given the fact that most programs have a large void space in virtual memory between heap and stack, there is much potential to save storage space here. Recall the example we discussed in Section 3.2.3.2 where a program of 20 MByte had a gap of almost 4 GByte in virtual memory. If a page had the size of 1 KByte, then this program would need “only” 2000 entries (out of $2^{20}$), meaning that more than 99.5% of the page table is not used.

The common solution employed in operating systems is to have hierarchic page tables or page descriptor trees. A single page table is regarded as a special case of a hierarchic page table. Each entry in the page table is either a page descriptor or a pointer to another page table.

The idea is to start with a small page table (i.e., a page table with a small number of entries). Each such entry is responsible for covering a relatively large part of the virtual address space. Each entry can be refined by another page table in a similar way. For example in Figure 3.20, virtual memory has 16 addresses (numbered 0000 to 1111). The first level page table has four entries. Each entry is responsible for handling a quarter of the entire virtual address space, i.e., four addresses. An entry at the first level then points to a second level page table with again four entries but which deals with the details of address translation. In this example you can already see that a full (single level) page table would have needed 16 page descriptors. In the hierarchic version we need only three small tables with four entries each, i.e., twelve page descriptors altogether.

A real-world example from the Intel i386 architecture shows the effect even more dramatically: The Intel processor uses 4-KByte-sized pages, thus a virtual address is split into a 20-bit page number and a 12-page offset. Using a hierarchical page table, it is possible to split the 20 page number bits in two halves.

Following Intel terminology, the first ten bits are used as index into the page directory, and the second ten bits reference a page table.

The first ten bits in this example can be called upper page number, the last ten bits lower page number (Figure 3.21). The effect on the page table size is a reduction to roughly the square root, i.e., from $2^{20}$ to $2^{10}$ entries. (If the size of one entry were one byte, the reduced table would have exactly square root size.)

![Figure 3.21: On the Intel i386 architecture, paging uses a (per-process) page directory whose entries point to page tables. This constitutes a hierarchical page table with 20 bits for the page number and twelve bits for the offset.](image-url)

Notice however that while going from a million entries to 1000, this introduces 1000 secondary page tables. If a process actually used this much RAM, all the secondary page tables would be filled, and no space would be saved. (Actually, in that case you get an
increased amount of space used for tables, because the primary table counts extra.) But normal processes will not have such enormous memory demands, and this saves a lot of space because secondary page tables can be created on demand—as long as a process uses only a few kilo- or megabytes of RAM, only the first few secondary page tables need exist.

### 3.2.4.4 Page Descriptors and Page Table Descriptors

In general, a hierarchic page table is a tree of descriptors. Descriptors can be of two forms:

- **Page descriptors** (PD) are the “leaves” of the tree. They are page descriptors in their original sense, including information about the location of the page frame and protection bits.

- **Page table descriptors** (PTD) are the “inner nodes” of the tree. They basically are pointers to descriptors (either page descriptors or page table descriptors).

The resulting tree-like structure is visualized in Figure 3.22.

Since we know what is part of a page descriptor already (see Section 3.2.4.2), what is part of a page table descriptor? Generally we can find these entries:

- A flag indicating the *type* of the descriptor, i.e., is it a page descriptor or a page table descriptor. In fact, also every page descriptor needs this field.

- The address of the page table which this page table descriptor points to.

![Figure 3.22: Tree structure of descriptors.](image-url)
• In case it is not clear from the hardware architecture, a page table descriptor may also store the size of the page table. This is analogous to the size of a segment in segment-based virtual memory.

• A presence bit (P bit) which indicates whether the page table pointed to by the descriptor is in main memory or not. This indicates that also page tables can themselves be paged out into secondary storage. We will get back to the problems this may cause later in Section 3.2.5.

• Multi purpose bits, just like in a page descriptor.

### 3.2.4.5 Structure of a Virtual Address

In a hierarchic page table, the virtual address is used in a special way to traverse the tree of descriptors. If there are \(L\) levels in the page descriptor tree, the bits of the page address \(p\) within a virtual address are subdivided into \(L\) parts \(p_1, p_2, \ldots, p_L\) such that

\[
p = p_1 \oplus p_2 \oplus \ldots \oplus p_L
\]

(where \(\oplus\) denotes the join operation of strings). Mathematically, this corresponds to

\[
p = p_1 \times 2^{e_1} + p_2 \times 2^{e_2} + \ldots + p_L \times 2^{e_L}
\]

for some decreasing sequence of exponents \(\{e_1, e_2, \ldots, e_L\}\) with \(e_L = 0\). The address translation starts with the leftmost (highest order) bits. Briefly spoken, the first bits (in \(p_1\)) are an index into the first level page table, the next bits (in \(p_2\)) an index into the second level page table and so on. Therefore the number of bits per level determines the size of the page table at that level.

As an example, consider the division of \(p\) into parts in Figure 3.23. The first seven bits (for \(p_1\)) allow a first level table size of \(2^7 = 128\) entries. In a 32 bit system, each entry covers an area of 32 MByte. The second level’s seven bits for \(p_2\) handle again 128 entries, each of them now covering 256 KByte. Finally, the third level’s seven bits (for \(p_3\)) are an index into a page table with 128 entries, each entry finally covering a full page of 2 KByte (eleven bits remain for the offset). As in this example, it is common to have two or three distinct levels in the page descriptor tree of a virtual address space.

![Figure 3.23: Example structure of a multi-level virtual address; \(p = p_1 \oplus p_2 \oplus p_3 = p_1 \times 2^{14} + p_2 \times 2^7 + p_3\).](image)

### 3.2.4.6 Multi-Level Address Translation

We now discuss several examples of how address translation works using hierarchic page tables. The first example is depicted in Figure 3.24. It shows a two-level descriptor tree. How does the MMU perform address translation here? It starts with the “root” page table,
virtual address

page table register

MMU

physical address

Figure 3.24: Address translation using a two-level page descriptor tree.

virtual address

page table register

MMU

physical address

Figure 3.25: Address translation using a three-level page descriptor tree.
pointed to by the MMU register \( \text{PTR} \). This is the first level page table. The MMU takes the first part \( p_1 \) of the virtual address as an index into this table where it finds a page table descriptor. This points to the relevant second level page table. Within this page table, the second part \( p_2 \) of the virtual address is used as an index. Since we are at the highest level of the tree, the descriptor at index \( p_2 \) in the second level page table is a real page descriptor allowing to perform the address translation into a physical address.

The example above can easily be extended to three-level page descriptor trees. As an example consider Figure 3.25. In contrast to the first example, the level two page table does not point to the page frame but to a level three page table. The virtual address has a third part \( p_3 \) which is used as an index into this table where we find the page descriptor finally pointing to the page frame.

We can save a little storage space in the descriptor if its type is clear from the context. For example, the Intel i386 CPU assumes a two-level descriptor tree. Any descriptor found at level 1 is automatically a page table descriptor. All other descriptors (i.e., those at level 2) must be page descriptors.

3.2.4.7 Discussion

The advantage of hierarchic page tables is their potential to save significantly on main memory. Unused parts of virtual address spaces can be \"removed\" from the page table using null page descriptors. A null descriptor is a special descriptor indicating that the virtual memory at this location is void or unused. Usually it is encoded by a special flag or value in a field of the descriptor (either page descriptor or page table descriptor). By placing a null descriptor into the descriptor tree, all pages below this descriptor are effectively removed from virtual memory. The lower the level of the null descriptor, the larger the part of virtual memory which is mapped out.

To see how effectively null descriptors can be used, we reconsider the example we introduced in Section 3.2.3.2 and revisited in Section 3.2.4.3: the classic organization of virtual memory with a 20 MByte program, an empty heap and an empty stack. Recall that a single page table would need roughly 8 MByte of main memory. Using hierarchic page tables and null descriptors we can reduce the amount of necessary storage to 5 KByte, as is shown in Figure 3.26. Here the example is simplified to the situation where the program has no code and data pages to show the effect more clearly. Remember that on each level of the descriptor tree we had 128 entries per table, each entry requiring eight bytes. This means each table has a size of 1 KByte. By placing null descriptors in all places which point to empty virtual memory, we end up with page tables only for those parts of the system which really exist. Since the assumed hardware of this example places page descriptors only on level 3, we need to extend the descriptor tree up to level 3 for the two page descriptors necessary to point to heap and stack. When you count the number of page tables, you end with 5. Hence we need only 5 KByte instead of 8 MByte.

The disadvantage of a multi-level page tables is that address translation takes slightly longer. This is because the MMU has to traverse the tree (i.e., follow the pointers) when doing the address translation. A multi-level page table needs one main memory lookup per level, which takes longer than a single memory lookup if there were only a single
Figure 3.26: Saving main memory using null descriptors in hierarchic page tables.

page table. The memory savings however usually outweigh the performance drawback. Furthermore, performance can still be in the range of a single memory lookup (or less) by using caching, as we explain in the following section.

3.2.4.8 Translation Look-aside Buffers and the Locality Principle

Each memory access requires address translation which needs yet (at least) one other memory access for reading the page table, so it makes sense to use some kind of caching mechanism for the page to frame translation because most programs will not randomly access memory but instead access addresses which are close to one another. Think of loops reading all the elements of an array: they will be stored consecutively. So after one access to a memory frame it is likely that further accesses to the same frame will occur soon after the first one. This is called the locality principle. Lookups of the same frame would mean translating the page number to a page frame number again and again—in order to speed up this process many memory management units contain a translation look-aside buffer (TLB). That is a special type of memory called associative memory which can store page/page frame pairs and allows lookup in constant time: In order to find the page frame for a given page (assuming it is stored in the buffer) there is no need to loop over the entries in the buffer, but the buffer will return the frame number immediately if it contains the page number. If it does not, the result is an error, and the normal lookup process will start. But
finding a frame via the TLB is orders of magnitude faster than going through the regular tables, and this holds even more if split page tables are used.

The size of the TLB is typically very small, because those kinds of chips are limited in their size but the locality principle will guarantee that for “well-behaving programs” (i.e., those that respect this principle) it will be sufficient to dramatically speed up the address translation.

Since the TLB is part of the memory management unit, it will be used automatically by the CPU; no specific programming is necessary to activate or use it.

Note that the (page → page frame) mapping exists for every single process in the system: Since each process has its own virtual memory space, it makes no sense to combine their page tables in some kind of system-wide table. This has consequences for the TLB as well: If it, as described so far, only stores page and frame numbers, then every context switch to another process will invalidate all its entries. So if the scheduler switches processes very often, this will limit the use of the TLB. Alternatively the TLB could be constructed in a way that maps (process ID, page number) pairs to frames: That would keep all entries valid across context switches, but with different processes always accessing different page frames it would only work well in a setup with either very few processes or with a sufficiently increased TLB size.

### 3.2.4.9 Digression: Indirection in Filesystems

Somewhat similar to the way in which a page table holds information about the page frames currently used by a process, filesystems keep records of disk areas used by a file. A thing that is shared by both methods is the use of equal-sized partitions of the medium—in the case of hard disks they are called blocks or clusters and typically have the size of a few kilobytes, e.g., 1, 2, 4 or 8 KByte.

For each file the operating system has to keep a list of blocks that the file’s data occupy. With very large files this list also becomes very large, because a file of size 1 MByte uses 1024 blocks, if the block size is 1 KByte.

Storing the block list in the overall data structure that the operating systems keeps for administering the filesystem is not very efficient, because in order to allow for huge files, each such entry would have to reserve space for a possibly very long list—even for those files that only use a few blocks. Thus many filesystems store the block list in special data blocks. This approach is called single indirection: From wherever the information about a certain file is stored, entries do not point directly to a data block, but to an indirection block that contains further pointers to several other data blocks. These entries can be block numbers, since by multiplying the block number with the block size the absolute disk address can be calculated. If it takes two bytes to store a block number and a data block has a size of 4 KByte, then 2048 block addresses can be stored in one indirection block. When the first indirection block is fully used, a second one can be introduced in order to allow for even bigger files.

Typically the administration data will not only contain pointers to indirection blocks but also a few direct pointers (to data blocks) so that in the case of small files it is possible to find all data blocks without going through indirection blocks. Only when the number
Managing Memory

Inode

Figure 3.27: Multiple indirection in Unix filesystems: A Minix inode stores seven direct block numbers and three block numbers for single, double and triple indirection blocks. Triple indirection is not implemented in ULIx.
of data blocks exceeds the number of directly stored block addresses, a first indirection block will be used.

With single indirection the maximum size of files grows a lot; however it is still limited: If there are 20 pointers to indirection blocks and such a block stores 2048 block numbers (as above), then this allows for 40 K data blocks or file sizes of up to 40 K × 4 KByte = 160 MByte. By adding more and more indirect pointers in order to allow for yet bigger files, the administrative data for a single file grows equally; so a second level of indirection is introduced to keep the file entries small. With double indirection there are pointers that point to indirection blocks which link to further indirection blocks. Those then finally point to address blocks. What we said about the number of block addresses remains valid in the case of double indirection, but now one double indirection pointer allows to address $2048 \times 2048 = 2048^2$ (or roughly four million) data blocks.

If this is still not good enough, triple indirection or even higher levels of indirection can be introduced: With each additional indirection step the maximum file size grows by the same factor (2048 in the example). But notice that it makes no point to use, say, ten or eleven layers of indirection just to be prepared for any possible future demands on file sizes: Indirection leads to extra accesses; in order to read a specific block from the disk whose block number is only available through a long indirection, several blocks have to be read from the disk. If the block is on a triple indirection path, it actually takes at least five read operations to retrieve the data: The first one is for looking up the address of the first level indirection block in the file’s administrative data. The second to fourth are for reading the indirection blocks, and the fifth one is the data block itself.

Whatever level of indirection is used, there are typically also indirection entries of all lower levels: In the same way that it makes sense to keep a few direct block number entries to speed up access to very small files, it is useful to have one (or a few) single indirections for those medium size files that do not require double indirection, and so on.

Figure 3.27 shows an example for multiple indirection in a Unix type filesystem. What is called inode in the image is a special administrative entry for a file that holds most of this file’s attributes including direct and indirect block numbers as well as things such as owner, owning group, access permissions, but not a filename. We will come to this later when we discuss examples of real filesystems in Chapter 12.

### 3.2.4.10 Further Reading

A comparison of three paging architectures (x86, PowerPC and MIPS) by Bruce Jacob and Trevor Mudge is available online [JM98].

### 3.2.5 Page-based Virtual Memory in Ulx

Virtual memory in Ulx is designed along the following principles:

- Every process will have its own virtual address space, i.e., an own page table (tree). Address spaces of processes are protected from one another, i.e., it is not possible to access address space a from a process that uses address space b and vice versa.
• Pages are stored in a set of page frames. Pages can be locked in physical memory. Locked pages cannot be paged out.

• Page replacement is done on a global basis, i.e., page replacement algorithms treat all frames in the same way irrespective of what pages reside in the frames (unless, of course, they are locked).

• The kernel has its own virtual address space. The virtual address space of every process is accessible from the virtual address space of the kernel.

In the following chapter you will see the data structures that ULIx uses to implement paging on the Intel i386 architecture, but we will only set up an initial page table that we need for completing the boot process. It will get more interesting in Chapter 6.1 when we introduce address spaces for processes.
Our goal is to have an operating system which will be able to boot from some media. Obviously we cannot just compile a standard binary executable file for some platform (e.g. the one we use for development), but instead will need a file format that a boot loader can load and execute on a real machine (or inside some PC emulator or virtualization software).

Writing a boot loader is not a hard task as long as we create the boot disk in such a way that the kernel binary is stored contiguously on the disk and we know its first sector number on the disk: the BIOS provides functions for reading sectors from disk. The older Linux boot loader LILO [AC00] worked this way: after rebuilding the Linux kernel and writing it to the boot disk, the boot loader had to be reinstalled because the sectors containing the kernel were hard-coded into the boot loader. LILO’s successor GRUB [Fre05] uses a more complex approach: it contains drivers for several filesystems and can find its configuration data and the kernel without knowing the sector numbers; it just looks up the relevant data in the disk’s directories.

We have decided against implementing our own boot loader because this tool is not part of the kernel. As capable tools such as GRUB already exist, it makes no sense to reinvent the wheel.

### 4.1 GRUB Loads the Ullix Kernel

We will use a FAT-formatted GRUB boot floppy disk (ulix-fd0.img) onto which we copy the Ullix kernel binary ulix.bin, and we configure the boot loader by placing the following file MENU.LST in the BOOT/GRUB subdirectory of the same disk:
When we power on the emulated (or real) PC, the BIOS will search for bootable media, and it will find an acceptable boot sector on the floppy disk. It loads the GRUB boot loader into memory and executes it. GRUB understands several filesystems, including Minix and FAT, and it will recognize the FAT disk and locate the above configuration file. Its only entry tells it to load the kernel binary.

But to what memory location does GRUB copy the kernel, and where will it start executing it? The kernel binary will be an ELF file (Executable and Linking Format, see Section 6.8) that contains a description of what data to copy from the ELF file to which memory locations.

GRUB supports kernel images which have a Multiboot header. The Multiboot specification [OFBI09] states:

“An OS image must contain an additional header called Multiboot header, besides the headers of the format used by the OS image. The Multiboot header must be contained completely within the first 8192 bytes of the OS image and must be longword (32-bit) aligned. In general, it should come as early as possible and may be embedded in the beginning of the text segment after the real executable header.”

In order to put a proper Multiboot header into our kernel binary, we will need to write some assembler code. The nasm assembler offers commands such as db (data byte), dw (data word) and dd (data double word) for writing eight bits, 16 bits or 32 bits of data into the file (see Appendix B.3), and the equ statement lets us define symbolic constants, similar to C’s #define pre-processor statement. The full header is only twelve bytes long, its contents are shown in Table 4.1.

<table>
<thead>
<tr>
<th>Bytes</th>
<th>Content</th>
<th>Fixed Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00–03</td>
<td>magic string</td>
<td>0x1badb002</td>
</tr>
<tr>
<td>04–07</td>
<td>flags</td>
<td></td>
</tr>
<tr>
<td>08–11</td>
<td>checksum</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Contents of the Multiboot header.

What are the proper values for the flags and checksum fields? According to the Multiboot specification, we need to set the bits 0 and 1 of the flags entry:

0: this is the “page alignment” flag, it guarantees that the kernel will be loaded to a physical address which is a multiple of 4096, the default page size on the Intel architecture.
4.2 The Ulix Memory Layout

1: the “memory information” flag provides the loaded operating system with data about the available memory. This data will be accessible via the “Multiboot information structure”, and the boot manager must place the address of this structure in the EBX register. Ulix does not currently use these data.

The following code shows a slightly modified version of the Multiboot header definition which we took from Bran’s Kernel Development Tutorial [Fri05].

```
(start.asm 87)≡

(section .setup)
[bits 32]
align 4
mboot:
  MB_HEADER_MAGIC equ 0x1BADB002
 ; Header flags: page align (bit 0), memory info (bit 1)
  MB_HEADER_FLAGS equ 11b ; Bits: 1, 0
  MB_CHECKSUM equ -(MB_HEADER_MAGIC + MB_HEADER_FLAGS)

 ; GRUB Multiboot header, boot signature
  dd MB_HEADER_MAGIC ; 00..03: magic string
  dd MB_HEADER_FLAGS ; 04..07: flags
  dd MB_CHECKSUM ; 08..11: checksum

Defines:
  mboot, used in chunk 94.

(For an explanation of the assembler commands equ (which is similar to C’s #define) and dd (which writes data right into the assembled object file) see Section B.3 of the appendix on x86 assembly language.)

When we look at the compiled kernel, we will find the Multiboot header. Note that 32-bit numbers are stored in little-endian: for example, the magic string 0x1badb002 shows up as 02 b0 ad 1b.

$ hexdump -C ulix.bin
[...]
00001000 02 b0 ad 1b 03 00 00 00 fb 4f 52 e4 17 00 12 00 |...........OR.....|
[...]

4.2 The Ulix Memory Layout

While it’s too early to discuss the memory management implementation in detail, we need to decide now what the general memory layout is going to be: when we load the kernel it will end up somewhere in memory, so we must say where that is going to be.

Ulix will implement virtual memory (with paging), and every process on the system will have its own address space which is basically a mapping of virtual addresses to physical memory.

We split the available 4 GByte of virtual memory which are available on a 32-bit machine into 3 GByte for user space (addresses 0x00000000 – 0xBFFFFFFF) and 1 GByte for kernel space.
space (addresses \(0xC0000000 - 0xFFFFFFFF\)). Every process will see the same upper 1 GByte of kernel space (see Figure 4.1).

<table>
<thead>
<tr>
<th>0xFFFFFFFF</th>
<th>Kernel space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xC0000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0xBFFFFFFF</th>
<th>User space</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: This is a simplified view of the Ulxix memory layout.

This memory layout is similar to the one used by 32-bit-Linux which also puts the kernel in the upper quarter of the virtual memory.

With this in mind we will compile the Ulxix kernel such that it uses addresses above \(0xc0000000\).

### 4.3 From Real Mode to Protected Mode

**Real Mode**

When the computer powers up, it runs in **Real Mode**, a legacy mode of operation which is compatible to earlier Intel CPU generations—all the way back to the Intel 8086 processor from 1978.

When using a simple boot loader the operating system has to do the switch from Real Mode to **Protected Mode** manually, however GRUB activates Protected Mode for us, so all we have to do in the early initialization is to set up segmentation properly.

### 4.3.1 Segmentation in Real Mode

In Real Mode, the CPU uses 16 bit wide registers to address the memory, and via a built-in method called segmentation a 20 bit wide address space can be used. That is achieved via an odd technique: several segment registers can be used to point to a different memory range. Segments are always \(2^{16}\) bytes = 64 KByte large.

Let’s see an example for this: Assume we have a machine with 512 KByte of RAM, it has **physical addresses** \(0x00000\) to \(0x7FFF\). In principle, the 16-bit registers of the 8086 CPU could only access the first 64 KByte (with addresses \(0x0000\) to \(0xFFFF\)) of that memory—the first segment. In order to access the second segment (addresses \(0x10000\) to \(0x1FFFF\)), we use a segment register, e.g. DS (data segment). The segment registers are also 16 bits wide, but we intend to store a 20-bit wide address inside them. Since 20 bits cannot fit inside a 16-bit register, we discard the four lowest bits, assuming they are always 0.

Here are the binary representations of \(0x10000\) and \(0x1FFF\):

\[
0x10000 = 10000000000000000_b \\
0x1FFF = 1111111111111111_b
\]
The 20-bit wide representation of $0x10000$ is $00010000000000000000_b$, and when we remove the lowest four zero bits, we get $00010000000000000000_b = 0x10000 >> 4 = 0x1000$.

We can now access addresses $0x10000$ to $0x1FFFF$ by setting $DS$ to $0x1000$ and actually addressing $0x0$ to $0xFFFF$, because the CPU will automatically left-shift $DS$ by four bits and add the resulting value to the addresses we supply.

Segmented addresses are always written in the form $seg:addr$ and called *logical addresses*, for the example above, $0x10000 = 1000:0000$ and $0x1FFFF = 1000:FFFF$.

We could partition the 512 KByte $= 8 \times 64$ KByte RAM of our example machine into eight segments with addresses $0000:x, 1000:x, \ldots, 7000:x$. The maximum amount of memory that the 8086 CPU can use is 1 MByte, and for a machine with that much RAM we would add the segments with addresses $8000:x, 9000:x, \ldots, F000:x$.

However, it is not required that a segment starts at a multiple of 64 KByte; instead a segment register may hold any 16-bit value (which will be left-shifted into a 20-bit address with four trailing zeroes), thus the start address of a segment is some multiple of $2^8 = 16$.

Now, for our operating system we do not want to use Real Mode, since it offers no memory protection and 1 MByte of memory is not much. The alternative is Protected Mode, and it also uses segmentation, but in a more complex way. The 80386 processor’s segmentation mode is similar to segmentation on the 80286 CPU with the difference that the 80386 supports a 32-bit address space instead of a 24-bit address space, as well as paging.

### 4.3.2 Privilege Levels in Protected Mode

When we run the PC in Protected Mode there are four different “privilege levels” 0–3 in which the CPU can operate. Protected Mode segments allow us to declare the rights needed to access a segment, for example, if we are currently running in privilege level 3 and try to read a memory address in a segment which only allows access from level 0, our attempt will fail.

Later, when we talk about paging, we will give a more detailed description of the privilege levels; for now it is sufficient to note that *Unix* will use two of these levels: level 0 for the kernel and level 3 for user mode applications (processes). We will use the following phrases as synonyms:

- “The system runs in privilege level 0”,
- “the system is in kernel mode”, and
- “the system runs in ring 0”.

Similarly for level 3, we treat

- “the system runs in privilege level 3”,
- “the system is in user mode”, and
- “the system runs in ring 3”

as equivalent statements.


4.3.3 Segmentation in Protected Mode

The more modern implementation of segmentation does not store addresses in the segment registers, instead it works with a segment table and lets the segment registers point to entries in that table. A table entry does not only specify where a segment starts but also what length it has. If the length is not set to the maximal value (0xFFFFFFFF), it is possible to generate a forbidden access (to an address outside the segment) which the CPU will block, generating a fault.

When we start the OS initialization we must provide such a segment table, since it is not possible to use Protected Mode without one: Volume 3 of the Intel (R) 64 and IA-32 Architectures Software Developer’s Manual [Int11, p. 3-1] states:

“When operating in protected mode, some form of segmentation must be used. There is no mode bit to disable segmentation.”

Sample assembler code for setting up segmentation can be found in the same document on p. 419 (p. 9-23).

After GRUB turns over control to our kernel, a segment table is in use (after all, we’re already running in Protected Mode), however we will discard that table and provide our own one.

The data structure we have to create is called the “global descriptor table” (GDT) and consists of several segment descriptors.

Each segment descriptor is eight bytes long, and besides other values it contains a 32-bit base address and a 20-bit limit which is left-shifted by 12 bits to form a 32-bit limit. During the shift, 1-bits are inserted on the right, thus for example 0xFFFF (20 1-bits) becomes 0xFFFFFFFF (32 1-bits) during the shift.

Both values are spread in a weird pattern across the descriptor:

- base (bits 0..23): in bytes 2, 3, 4 of the descriptor
- base (bits 24..31): in byte 7 of the descriptor
- limit (bits 0..15): in bytes 0, 1 of the descriptor
- limit (bits 16..19): lower four bits of byte 6 of the descriptor

![Figure 4.2: In a segment descriptor the base and limit values are spread across the eight bytes in a weird pattern.](image)

A descriptor contains more than just the address range (see Figure 4.2): The upper four bits of byte 6 contain two flags (granularity and size) and two 0 bits;
4.3 From Real Mode to Protected Mode

- we must set the granularity bit to 1: this causes the left-shift for the limit value; if the bit was 0, we could declare the limit in bytes instead of multiples of 4 KByte.
- the size bit must also be set to 1, declaring this descriptor to be a 32-bit protected mode descriptor; otherwise it would be a 16-bit descriptor which is something we don’t need (it exists for backwards compatibility with the 80286).

Thus, the upper four bits of byte 6 are always $1100_2$.

We have now described everything except byte 5 of the descriptor which defines its type. Its bits have the following functions:

7: present bit, must be set to 1

6/5: descriptor privilege level (DPL), must be set to 00 for ring 0 (kernel mode) or 11 (=3) DPL for ring 3 (user mode)

4: reserved, must contain 1

3: executable bit, we will set this to 1 in our code segment descriptor and to 0 in our data segment descriptor

2: direction bit / conforming bit:

- for the data segment, 0 means that the segment grows upwards;
- for the code segment, 0 means that the code in this segment can only be executed if the CPU operates in the ring that is declared in bits 6/5 (privilege level).

1: readable bit / writable bit: we always set these to 1; for a code segment it means that we can also read from this segment, and for a data segment it means we can also write to it.

0: accessed bit: we set this to 0; the CPU flips it to 1 when this segment is accessed.

The corresponding C datatype for a GDT entry is the following:

```c
⟨type definitions 91⟩

struct gdt_entry {
    unsigned int limit_low : 16;
    unsigned int base_low : 16;
    unsigned int base_middle : 8;
    unsigned int access : 8;
    unsigned int limit_high : 4;
    unsigned int flags : 4;
    unsigned int base_high : 8;
};

Defines:
    gdt_entry, used in chunk 110a.
```
When we tell the processor to use our table we cannot directly point it to the beginning of the GDT (e.g. via `gdt[0]`), instead we need an extra data structure which just contains the size and the start address of the table:

```
struct gdt_ptr {
    unsigned int limit : 16;
    unsigned int base : 32;
} __attribute__((packed));
```

Defines: `gdt_ptr`, used in chunk 92b.

With `__attribute__((packed))` we force the compiler to store the data precisely in this way, otherwise optimizations could change the order.

```
struct gdt_entry gdt[6];
struct gdt_ptr gp;
```

Defines: `gdt`, used in chunks 109c, 110a, and 196a.

Note that `gdt[0]` allows us to store six segment descriptors; this is the number of descriptors we will use in Ulx—in general, many more descriptors (up to 8192) can be used.

Since we want to create segment descriptors for kernel mode (ring 0) and we need one code and one data selector, the type bytes will be

- `10011010b` for the code segment
  (present; ring 0; fixed-1; executable; exact privilege level; allow reading; not accessed)
- `10010010b` for the data segment
  (present; ring 0; fixed-1; not executable; grow upwards; allow writing; not accessed).

Figure 4.3: Our descriptors for the code segment (top) and the data segment (bottom) only differ in the Type fields.
4.3 Preparations for Paging

Without delving into the details of paging, we need to prepare our kernel for its usage right now: Later, when we turn on virtual memory, we want the kernel to use virtual addresses 0xc0000000 – 0xffffffff and user mode programs to use virtual addresses 0x00000000 – 0xbfffffff. This means that we have to compile and link the kernel in such a way that all absolute addresses (of functions and data) lie in the range above 0xc0000000. But that’s a large number, and we do not expect to have physical memory with such high addresses. By setting the base address in our descriptors to 0x40000000, we can load the kernel to low addresses.

Imagine for example that the kernel calls a function which has the entry address 0xc0001234. With the way we will set up the segments the base address 0x40000000 will be added, resulting in

\[
\begin{align*}
0xc0001234 \\
+ 0x40000000 \\
= 0x100001234
\end{align*}
\]

which is no longer a 32 bit address; the leading 1 will be lost in this addition, resulting in the address 0x0001234—where we’ll physically put the code of this function.

This is called the “higher half trick” [Son07, Rob01] (even though we’re reserving the upper quarter and not the upper half of the virtual memory for the kernel).

Now, in order to actually do the memory initialization, we need to load the global descriptor table and activate it. Again, we use code from Bran’s Kernel Development Tutorial [Fri05]:

```assembly
section .setup

trickgdt: dw gdt_end - gdt_data - 1 ; GDT size
dd gdt_data ; linear address of GDT
gdt_data: dd 0, 0 ; selector 0x00: empty entry

; code selector 0x08:
; base 0x40000000, limit 0xFFFF, type 10011010, flags 1100
db 0x0f, 0xFF, 0, 0, 0, 10011010b, 11001111b, 0x40

; data selector 0x10:
; base 0x40000000, limit 0xFFFF, type 10010010, flags 1100
db 0x0f, 0xFF, 0, 0, 0, 10010010b, 11001111b, 0x40

gdt_end:
```

Defines:

- trickgdt, used in chunk 94.
global start

[section .setup]

start: ; BEGIN higher half trick
    lgdt [trickgdt]
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:higherhalf ; far jump to the higher half kernel

[section .text]

higherhalf: ; END higher half trick
    mov esp, _sys_stack ; set new stack
    push esp ; save ESP
    push ebx ; address of mboot structure (from GRUB)

    extern main ; C function main() in ulix.c
    call main
    jmp $ ; infinite loop

Defines:
    start, used in chunks 95b and 620b.
Uses _sys_stack 95a, main 44b, mboot 87, and trickgdt 93.

This code does the following things:

- It loads our descriptor table via the \lgdt instruction,
- It sets the segmentation registers DS, ES, FS, GS and SS to 0x10 (thus making them point to our data segment descriptor which has index 0x10),
- Since it cannot directly write a value to the CS register, it makes a far jump. The instruction jmp 0x08:higherhalf\sub{94} jumps to the address higherhalf\sub{94} in the segment specified by the segment descriptor with index 0x08—and this automatically sets CS properly.
- Then it defines the stack we will use during system initialization (by loading ESP) and finally calls the \texttt{main\sub{44b}()} function from our C file ulix.c.

Note that most parts of the code “live” in the .setup section of the code, whereas the last lines (starting with the higherhalf\sub{94} label) live in the .text section.

We will also need an additional stack, and we reserve its place in the same assembler file. The \texttt{resb} instruction does just that: it reserves a certain number of bytes, in our case $32 \times 1024$ for 32 KByte of stack memory. Since a stack grows from higher to lower addresses (downwards), the label _sys_stack\sub{95a} follows after the reserved bytes:
4.3 From Real Mode to Protected Mode

\[\textit{start.asm} 87\] +

\begin{verbatim}
global stack_first_address

global stack_last_address

[section .bss]
stack_first_address:
    resb 32*1024 ; reserve 32 KByte for the stack

stack_last_address:
_sys_stack:

Defines:
_sys_stack, used in chunk 94.
stack_first_address, used in chunk 604.
stack_last_address, used in chunk 604.
\end{verbatim}

We use separate sections because we will tell the linker \texttt{ld} to use different addresses for those sections. This can be achieved with the following linker configuration file \texttt{ulix.ld}. (This linker file is a modified version of the one provided in Bran’s Kernel Development Tutorial [Fri05]; we changed the output format to \texttt{elf32-i386} and the start address to 0 and introduced an offset of 0xc0000000 for the main parts of the kernel.)

\[\textit{ulix.ld} 95b\]

\begin{verbatim}
OUTPUT_FORMAT("elf32-i386")
ENTRY(start)
phys = 0x00100000;
virt = 0xC0000000;
SECTIONS {
    . = phys;

    .setup : AT(phys) { *(.setup) }

    . += virt;

    .text : AT(code - virt) { code = .;
        *(.text)
        *(.rodata*)
        . = ALIGN(4096); }

    .data : AT(data - virt) { data = .;
        *(.data)
        . = ALIGN(4096); }

    .bss : AT(bss - virt) { bss = .;
        *(COMMON*)
        *(.bss*)
        . = ALIGN(4096); }

    end = .; }

Uses start 94.
\end{verbatim}

The file mainly accomplishes two things:
• it tells the linker `ld` to use addresses in the address range starting with 0x100000 (= 1 MByte mark) for everything that we have declared as part of the `.setup` section, and it will also cause the boot manager to load the whole kernel to 0x100000.

The `.setup` region contains the code that runs before paging is enabled. It prepares the segment table and switches it on.

• the line `. += virt;` lets the linker use modified addresses for all the other sections: wherever an absolute address occurs in a CPU instruction, it will get 0xC0000000 added to its original value. (Technically, it adds 0xC0000000 to the current output location counter “.”: if the last linked instruction ended on position 0x105555, linking would continue with address 0xc0105556.)

If this line was not followed by

```
.text : AT(code - virt) { code = .;
```

it would have the effect to generate code and data which would be loaded at addresses beyond 0xC0000000, but the `AT` statement says that it shall be placed in a different location: `code` is set to `.`, which is the current location, and `AT` calculates `code - virt` which is just the next address behind the `.setup` section. The consequence is that the text section will be loaded within the second MByte of physical RAM, and—without enabling paging—it will not be executable in that place because all addresses in the code will have an additional offset of 0xC0000000. This must later be corrected before jumping into that section. Our segment table does just that.

• `.text` is the code section, it will contain everything that gets executed (except for the parts in `.setup`).

• `.data` and `.bss` contain program data structures which can be read and written, but not executed. The difference between the two is that variables in `.data` have an explicit non-zero initialization in the code, whereas the ones in `.bss` do not—the linker initially sets them up with zeroes.

This means: for code in the `.setup` section, physical addresses are identical with the addresses used in that part of the binary. That does not hold for the rest of the kernel, where all addresses are increased by 0xC0000000—which would normally render that part of the code useless, but our segmentation trick with a base address of 0x40000000 makes it just right.

The `.ALIGN` statements force the linker to align each section to the start of a page (or page frame) of size 4096. You can find more information about the linker in the `ld` manual [CT04].

When we inspect the linked kernel binary `ulix.bin` with the `objdump` tool, we see what happens:

```
$ objdump -h ulix.bin
ulix.bin:   file format elf32-i386
Sections:
```
The columns VMA and LMA display the first virtual memory address and the load memory address of each section. The first one shows what memory location the code was prepared for, and the second one shows the absolute load address, i.e., where the boot loader will store the section in RAM.

As we’ve already described, our code starts executing in the .setup section, where it sets up the GDT and enables it; then it continues execution in the .text section.

### 4.4 Virtual Memory for the Kernel

Setting up the memory consists of creating an initial page table for the kernel: This is a two-step procedure, we start with an *identity mapping*: that is a page table which maps virtual addresses 1:1 to physical addresses. The identity mapping lets us smoothly enable paging; when the CPU fetches the instruction which follows immediately after the enabling instruction, that memory access uses the memory management unit (MMU) and the translation information in the page table, whereas all earlier instructions accessed the memory (almost) directly.

\[
\begin{align*}
\langle \text{setup memory} \rangle & \equiv \\
\langle \text{setup identity mapping for kernel} \rangle & \equiv \\
\langle \text{enable paging for the kernel} \rangle & \equiv \\
\langle \text{install flat gdt} \rangle & \equiv
\end{align*}
\]

We will discuss the details in this section.

Since we implement ULIx-i386 for Intel chips, we need to have a look at the Intel architecture which uses a two-layer design (see Figure 4.4).

When paging is active, the CPU register CR3 (control register 3) points to a page directory which is a collection of 1024 page directory entries. Each of those entries is four bytes large, so the whole page directory has a size of 4 KByte (one page).

Each page directory entry points to a page table (its address is given via bits 31..12).

Page tables have the same size as page directories (4 KByte), and they also hold 1024 entries, the page table entries. Such an entry points to a frame (again using bits 31..12).

We will look at these data structures in detail in the following sections. While—as an OS designer—you are free to implement many things in any way you can conceive, the Intel processor expects page directories and page tables to have a well-defined form that cannot be changed.
4.4.1 Page Descriptors and Page Table Descriptors in ULiX

We now define the structures `page_desc` and `page_table_desc` for page (table) descriptors. They use the layout that is required by the Intel CPU. Recall that it expects that the page table tree has exactly two levels. Descriptors at level 2 are either null descriptors or page descriptors. Descriptors at level 1 are either null descriptors or page table descriptors.

Intel uses a different vocabulary: In the Intel terminology,

- a page descriptor is a *page table entry* (PTE; within a page table), and
- a page table descriptor is a *page directory entry* (PDE; in a page directory).

Each entry (for both page tables and page directories) is four bytes long; a page table or page directory contains 1024 such entries, filling exactly one page (of size 4 KByte).

A virtual address is always split in three parts: the page table number (bits 31–22), the page number (21–12) and the offset (11–0), see Figure 4.5.

```
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
|   page table   |   page   |   offset   |
```

Figure 4.5: Virtual addresses consist of three parts: a 10 bit index into the page directory, 10 further bits as entry into a page table and 12 bits as offset.

Page table number and page number are Intel’s specific terms for the general concept of split page table numbers where each portion corresponds to some level of page tables.
The system must associate a page directory with each process, the start address of the page directory must be stored in the process descriptor base register (PDBR), the upper 20 bits of control register 3 (CR3).

20 bits are sufficient to store an address because pages are page-size-aligned, i.e., they all start at addresses with zeroes in the lower 12 bits. (A page has size 4 KByte = $2^{12}$ bytes, which is why the offset length is 12.)

### 4.4.1.1 Page Table Entries

The upper 20 bits of a page descriptor contain the upper bits of a frame address (in RAM); the remaining bits of that address are zeroes for the same alignment reason as already described above: In RAM, no frame starts at an “odd” address which is not a multiple of the page size. Thus the frame address can easily be extracted from the page descriptor by setting the lower 12 bits to zero, if we treat the whole page descriptor as a 32-bit integer (which can be achieved with a cast operation in C, see also Exercise 13 on page 126):

\[
\text{frame_address} = \text{page_descriptor} \& 0xFFFFF000;
\]

// F (hex) = 1111 (bin); 0 (hex) = 0000 (bin)

The remaining bits in the page descriptor are either unused and can be used by the operating system for its own purposes (bits 9–11) or store attributes of this page descriptor (see Figure 4.6):

- Bits 8 and 7 must always be 0.
- Bit 6 holds the Dirty (D) flag.
- Bit 5 holds the Accessed (A) flag. It is automatically set by the MMU when this page is accessed.
- Bit 4 is called Page Cache Disabled (PCD) – if set, data from this page must not be cached.
- Bit 3 is called Page Write Transparent (PWT), we will ignore this one and always set it to 0.
- Bit 2 holds the User Accessible (U) flag.
- Bit 1 holds the Writeable (W) flag.
- Bit 0 holds the Present (P) flag.

![Figure 4.6: A page descriptor stores the upper 20 bits of the frame address and administrative data.](image)
We can now describe the page descriptor in C:

```
typedef struct {
    unsigned int present : 1; // 0
    unsigned int writeable : 1; // 1
    unsigned int user_accessible : 1; // 2
    unsigned int pwt : 1; // 3
    unsigned int pcd : 1; // 4
    unsigned int accessed : 1; // 5
    unsigned int dirty : 1; // 6
    unsigned int zeroes : 2; // 8.. 7
    unsigned int unused_bits : 3; // 11.. 9
    unsigned int frame_addr : 20; // 31..12
} page_desc;
```

Defines:
page_desc, used in chunks 72, 100, 101b, and 295b.

We repeat the code for calculating the physical address from this page descriptor, but now in a proper function that we can use later:

```
memaddress page_desc_2_frame_address (page_desc pd) {
    // pointer magic/cast: a page descriptor is not really an unsigned int, but we want to treat it as one
    memaddress address = *(memaddress*)(&pd);
    return address & 0xFFFFF000; // set lowest 12 bits to zero
}
```

Uses memaddress 46c and page_desc 100a.

(This uses the casting trick we mentioned above: First we take the address of the page descriptor pd with the & operator, then we cast this pointer to a page descriptor to a pointer to a 32-bit integer with (memaddress46c*), and last we access the value with the * operator. Note that it is not possible to directly cast the page descriptor to an integer with a command like address = (memaddress)pd;—that is forbidden in the C language.)

The following function fills a page descriptor with values; its address must be provided (the space must be reserved by the caller). We provide the descriptor’s address as the first argument via a pointer; the other arguments are the present, writeable, user accessible and dirty bits and—most important—the physical frame address:

```
page_desc *fill_page_desc (page_desc *pd, unsigned int present,
    unsigned int writeable, unsigned int user_accessible,
    unsigned int dirty, memaddress frame_addr) {

    memset (pd, 0, sizeof (page_desc)); // first fill the four bytes with zeros

    pd->present = present; // then enter the argument values in
    pd->writeable = writeable; // the proper struct members
    pd->user_accessible = user_accessible;
```
The upper 20 bits contain—again—the upper bits of a frame address (in RAM), the same alignment argument allows us to leave out the lowest 12 bits of the address.

The remaining bits in the page descriptor are either unused and can be used by the operating system for its own purposes (bits 9–11) or store attributes of this page descriptor:

- Bits 8 and 7 must always be 0. (Actually if bit 7 is set, this declares that the page described by this entry is a 4 MByte, not 4 KByte, page. We will not discuss 4 MByte pages.)
Bit 6 is undocumented, we will always set it to 0.

The remaining fields are identical to those of a page table entry:

- Bit 5 holds the Accessed (A) flag.
- Bit 4 is called Page Cache Disabled (PCD) – if set, data from all pages belonging to this page table must not be cached.
- Bit 3 is called Page Write Transparent (PWT), we will ignore this one and always set it to 0.
- Bits 2, 1 and 0 hold the User Accessible (U), Writeable (W) and Present (P) flags.

The PCD, U, W and P flags enforce these properties for all pages that belong to this page table. Figure 4.7 shows the layout of the descriptor which is almost identical to that of the page descriptor, except for the missing Dirty flag.

![Figure 4.7: A page table descriptor stores the upper 20 bits of the physical address of the page table it points to and administrative data.](image)

The C structure which describes page table descriptors only differs from the page descriptor structure page_desc by replacing the dirty field with an undocumented field:

```c
typedef struct {
    unsigned int present : 1; // 0
    unsigned int writeable : 1; // 1
    unsigned int user_accessible : 1; // 2
    unsigned int pwt : 1; // 3
    unsigned int pcd : 1; // 4
    unsigned int accessed : 1; // 5
    unsigned int undocumented : 1; // 6
    unsigned int zeroes : 2; // 8.. 7
    unsigned int unused_bits : 3; // 11.. 9
    unsigned int frame_addr : 20; // 31..12
} page_table_desc;
```

Defines:

- `page_table_desc`, used in chunk 103.

For extracting the frame address from a page table descriptor we rewrite the function `page_desc_2_frame_address` by simply using the new `page_table_desc` structure instead of `page_desc`.
Uses memaddress 46c and page_table_desc 102.

and we also duplicate fill_page_desc 100c() as fill_page_table_desc 103b(). Note that the function has one argument less since the dirty attribute does not exist in page table descriptors:

```
(function implementations 100b) +≡  
memaddress page_table_desc 2-frame_address (page_table_desc ptd) { 
    memaddress address = *(memaddress*)(&ptd);
    return address & 0xFFFFF000;
}
```

```
Uses memaddress 46c and page_table_desc 102.
```

Just as we did for the page tables, we provide macros KMAPD 103c and UMAPD 103c which let us call fill_page_table_desc 103b with standard values:

```
#define UMAPD(ptd, frame) fill_page_table_desc (ptd, true, true, true, frame)
#define KMAPD(ptd, frame) fill_page_table_desc (ptd, true, true, false, frame)
```

```
Defines:
    fill_page_table_desc, used in chunks 103c, 105b, 108, and 116b.
Uses memaddress 46c, memset 596c, and page_table_desc 102.
```

A page directory contains 1024 page table descriptors:

```
typedef struct {
    page_table_desc ptds[1024];
} page_directory;
```

```
Defines:
    page_directory, used in chunks 105a, 122c, 164a, 165b, 167c, 169a, 171c, 211, 296, 297, 307a, and 308c.
Uses page_table_desc 102.
```
4.4.2 Identity Mapping the Kernel Memory

We will now use a trick that allows a smooth transition from non-paging mode to paging mode: identity mapping creates a page directory for the kernel that maps the first virtual addresses to identical hardware addresses. When we later switch on paging, nothing changes for the kernel, because the MMU will be set up to use a page table that translates virtual addresses to the same addresses we’ve used before. This will also demonstrate why we’ve set up the segment tables with 0x40000000 offsets earlier.

4.4.2.1 First Attempt at a Kernel Layout

We now present a first and intuitive approach to placing the kernel in memory—both real memory and virtual memory; we will soon see that this approach is not the best possible choice and use a different layout.

Here are some general considerations that will lead us in the following steps:

- When the machine starts it must load the kernel into RAM. At that time paging is not yet enabled, so when the computer begins executing our kernel it uses physical memory addresses.
- Since we cannot know how much physical memory will be installed in a machine, it makes sense to place the kernel in some area with low memory addresses, such as the first megabyte of RAM.
- At some point in time during initialization of the operating system we will enable paging. However, code execution must logically continue at the next instruction and must not become confused by the fact that addresses are now translated by the MMU. The easiest thing to do is compiling and linking the kernel with addresses starting at 0x0.

If we assume that the kernel (and its stack) fit in 1 MByte of memory, we can reserve this physical memory area (0x0000.0000–0x000F.FFFF). The instruction that is going to enable paging will be sitting somewhere in this area, so we have to make sure that after paging is turned on, the instruction pointer will point to the instruction that follows immediately.

We need to identify the virtual addresses 0x0000.0000 to 0x000F.FFFF with the same physical addresses. This amounts to 256 page table entries; they all fit in one page table (kernel_pt105a), and the page directory (kernel_pd105a) will have exactly one non-null entry pointing to that one page table.

This way, when the kernel has enabled paging, the first megabyte of virtual addresses will be in use (and reserved for the kernel), whereas the rest will have null pointers in the page directory and the page tables. So when we later talk about processes and threads, we can create processes which use virtual memory addresses starting at (virtual) address 0x00100000 (just after the first MByte), and those processes will have the first $2^{20}$ addresses unmapped. That way, when the process makes a syscall, we can modify the page tables so that the kernel’s address space is added—then all addresses (0x00000000 – 0x000FFFFF: kernel; 0x00100000 and above: process) will be available so that data can be copied between process and kernel memory (see Figure 4.8).
Let’s start by declaring the necessary global variables:

```c
page_directory kernel_pd __attribute__((aligned (4096)));
page_table kernel_pt __attribute__((aligned (4096)));
```

// prefer to work with pointers
page_directory *current_pd = &kernel_pd;
page_table *current_pt = &kernel_pt;

Defines:
- current_pd, used in chunks 105b, 108, 109a, 111b, 115, 116, 121–23, 170c, 279c, 603, and 611b.
- current_pt, used in chunks 106a, 108, and 603.
- kernel_pd, used in chunks 105b, 106c, 108, 162e, 164b, and 604b.
- kernel_pt, used in chunks 105b, 108, and 604b.

Uses page_directory 103d and page_table 101b.

We need to declare these with __attribute__((aligned (4096))) so that the C compiler aligns them properly in pages.

```c
for (int i = 0; i < 1024; i++) {
    fill_page_table_desc (&current_pd->ptds[i], false, false, false, 0);
}
```

// make page table kernel_pt first entry of page directory kernel_pd
KMAPD (&(kernel_pd.ptds[0]), (memaddress)(&kernel_pt));

Figure 4.8: Identity mapping, first attempt: We identify the first MByte of virtual memory with the first MByte of physical RAM.
for (int i = 0; i < 1024; i++) {
    // map 1024 pages (4 MB)
    (identity map page i in kernel_pt 106a)
};

Uses current_pd 105a, fill_page_table_desc 103b, kernel_pd 105a, kernel_pt 105a, KMAPD 103c, and memaddress 46c.

In order to identity-map page i in kernel_pt we need to fill the i
th entry. Page frame i starts at physical address i × PAGE_SIZE:

[106a] (identity map page i in kernel_pt 106a)≡ KMAP ( & (current_pt->pds[1]), i*4096 );

Uses current_pt 105a and KMAP 101a.

Finally, we have to enable paging in the CPU. That can be achieved by making some changes to the control registers CR0 and CR3 as follows:

- Control register 3 (CR3) must contain the address of the page directory (kernel_pd 105a),
- in control register 0 (CR0) we must set the PG (paging) bit which is bit 31. Setting this single bit is done by calculating cr0 = cr0 | (1<<31).

[106b] (global variables 92b)≡
char *kernel_pd_address; // address of kernel page directory

Defines:
kernel_pd_address, used in chunks 106c and 109a.

[106c] (enable paging for the kernel 1st attempt 106c)≡
kernel_pd_address = (char*) (&kernel_pd);
asm volatile ("mov %0, %cr3" : : "r"(kernel_pd_address)); // write CR3
cr0; asm volatile ("mov %cr0, %0" : "=r"(cr0) : ); // read CR0

cr0 |= (1<<31); // Enable paging by setting PG bit 31 of CR0
asm volatile ("mov %0, %cr0" : "r"(cr0) ); // write CR0

Uses kernel_pd 105a and kernel_pd_address 106b.

We call this code block (enable paging for the kernel 1st attempt 106c) because we’re not done yet; with the higher half trick, &kernel_pd 105a will not be a physical address and won’t fit the base values in our segment descriptors.

4.4.2.2 Second Attempt at a Kernel Layout

For several reasons (which we will not dig into), legacy properties of Intel machines suggest to keep the first megabyte of RAM unused. So we will physically store the kernel in the second MByte. However, once paging is turned on, the kernel’s addresses shall be found in the last of the four gigabytes (starting at 0xc0000000), as we’ve shown at this chapter’s beginning (see Figure 4.1 on page 88). We want processes to use the first three gigabytes and the kernel to reside in the last of the four available gigabytes of virtual memory. Figure 4.9 shows the necessary mapping of virtual addresses to physical addresses.

This is not an identity mapping, so when we start the system we run into a problem: We could link the kernel twice and also physically load it twice, into the physical ranges [1 MB, 2 MB] and [3 GB + 1 MB, 3 GB + 2 MB]—but that would require our physical memory to be big enough.
Virtual addresses

0
1 MB
2 MB
3 GB
3 GB + 1 MB
3 GB + 2 MB
4 GB

Physical memory

0
1 MB
2 MB

Figure 4.9: Second mapping attempt: Kernel starts at 3 GB + 1 MB (virtual) or 1 MB (physical).

Virtual addresses

0
1 MB
2 MB
3 GB
3 GB + 1 MB
3 GB + 2 MB
4 GB

Physical memory

0
1 MB
2 MB

Figure 4.10: Final mapping: Kernel starts at 3 GB + 1 MB and 1 MB (virtual) or 1 MB (physical).
The solution is to work with a double mapping, as can be seen in Figure 4.10 (during initialization).

So we do the following:

1. Load the kernel to physical addresses [1 MB, 2 MB] (0x100000 – 0x1FFFFFF).
2. Do the Higher Half Trick (see page 93): Enable the trick GDT with base address 0x40000000 and jump to the higher half.
3. Setup an identity mapping from [1 MB, 2 MB] (virtual) to [1 MB, 2 MB] (physical). Actually, we’ll map the whole first 4 megabytes, since one page table describes 4 MByte of virtual memory.
4. Additionally set up a mapping from [3 GB + 1 MB, 3 GB + 2 MB] (0xC0100000 – 0xC01FFFFFF; virtual) to [1 MB, 2 MB] (physical) – the corresponding page table has the same contents as the first one, it just gets pointed to from a different page directory entry. Again, we’ll map a whole 4 MByte block, [3 GB, 3 GB + 4 MB] to [0, 4 MB].
5. Activate paging.
6. Install a new “flat” GDT with base address 0x0.
7. Get rid of the initial mapping for [0, 4 MB] (virtual).

After that the kernel sees virtual addresses starting at 3 GByte (0xC0000000) only, and things work perfectly with the linker configuration we’ve discussed on page 95.

Now we have to modify the setup of the identity mapping:

```plaintext
(setup identity mapping for kernel)

// file page directory with null entries
for (int i = 0; i < 1024; i++) {
    fill_page_table_desc (&(current_pd->ptds[i]), false, false, false, 0);
}

// make page table kernel_pt the first entry of page directory kernel_pd
// maps: 0x00000000..0x003FFFFF -> 0x00000000..0x003FFFFF (4 MB)
KMAPD (&(current_pd->ptds[0]), (memaddress)(current_pt)-0xC0000000);

// make page table kernel_pt also the 768th entry of page directory kernel_pd
// maps: 0xC0000000..0xC03FFFFF -> 0x00000000..0x003FFFFF (4 MB)
KMAPD (&(current_pd->ptds[768]), (memaddress)(current_pt)-0xC0000000);

// map 1023 pages (4 MB minus 1 page)
for (int i = 0; i < 1023; i++) {
    (identity map page i in kernel_pt)
}

kputs("Kernel page directory set up.
\n");
```

Uses current_pd 105a, current_pt 105a, fill_page_table_desc 103b, kernel_pd 105a, kernel_pt 105a, KMAPD 103c, kputs 335b, and memaddress 46c.

Note that we’re leaving one entry free (mapping only 1023 pages)—we’ll later need this one to create the next page table.
Then we can load the process descriptor base register (PDBR, part of CR3) and modify CR0 so that the CPU switches to paging mode—the following code is almost the same as the one in ⟨enable paging for the kernel 1st attempt⟩, but it calculates the physical address of the page directory by subtracting 0xC0000000:

\[
\text{kernel\_pd\_address} = (\text{char*})\text{(current\_pd)} - 0xC0000000; \\
\text{asm volatile ("mov %0, \%cr3" : : ":r"(kernel\_pd\_address)) ; // write CR3} \\
\text{uint cr0; asm volatile ("mov \%cr0, \%0" : ":r"(cr0) : ); // read CR0} \\
cr0 |= (1<<31); // Enable paging by setting PG bit 31 of CR0 \\
\text{asm volatile ("mov \%0, \%cr0" : ":r"(cr0) ); // write CR0}
\]

Uses current\_pd 105a and kernel\_pd\_address 106b.

After paging is enabled we first update the GDT (and make it flat), then we can get rid of the identity mapping which works with low addresses.

### 4.4.3 Installing the “Flat” GDT

One of the last steps is installing the “flat” GDT: it looks like our trick GDT, but uses a base address of 0x0 instead of 0x40000000. We’re already executing C code, so we’ll provide a C function for loading the GDT:\(^1\)

\[
\text{void fill\_gdt\_entry (int num, ulong base, ulong limit, byte access, byte gran) } \\
\text{extern void gdt\_flush () ;}
\]

\[
\text{void fill\_gdt\_entry (int num, ulong base, ulong limit, byte access, byte gran) }
\]

\[
\text{// base address; split in three parts} \\
\text{gdt[num].base\_low = (base & 0xFFFF); // 16 bits} \\
\text{gdt[num].base\_middle = (base >> 16) & 0xFF; // 8 bits} \\
\text{gdt[num].base\_high = (base >> 24) & 0xFF; // 8 bits}
\]

\[
\text{// limit address; split in two parts} \\
\text{gdt[num].limit\_low = (limit & 0xFFFF); // 16 bits} \\
\text{gdt[num].limit\_high = (limit >> 16) & 0x0F; // 4 bits}
\]

\[
\text{// granularity and access flags} \\
\text{gdt[num].flags = gran \& 0xF;} \\
\text{gdt[num].access = access; }
\]

Defines:
- fill\_gdt\_entry, used in chunks 110a, 194a, and 197a.
Uses gdt 92b and ulong 46b.

The following code shows how the first three GDT entries are created, however, we will later introduce three further entries which we need for user mode processes—you can ignore that for now, but that is the reason why we reserve space for six GDT entries

\(^1\) The functions in this section are—again—based on code from Bran’s Kernel Development Tutorial [Fri05].
(instead of three) and we include the code chunk \textit{(install GDTs for User Mode 194a)} and function calls to \texttt{gdt\_flush_{110b}} and \texttt{tss\_flush_{197c}}, the latter of which will be explained when we discuss processes.

\begin{verbatim}
\langle install flat gdt 110a\rangle ≡
// We'll have six GDT entries; only three are defined now
gp.limit = (sizeof (struct gdt_entry) * 6) - 1; // must be -1
gp.base = (int) &gdt;
fill_gdt_entry (0, 0, 0, 0, 0); // null descriptor

// code segment: base = 0, limit = 0xFFFFF
fill_gdt_entry (1, 0, 0xFFFFF, 0b10011010, 0b1100);

// data segment: base = 0, limit = 0xFFFFF
fill_gdt_entry (2, 0, 0xFFFFF, 0b10010010, 0b1100);

\langle install GDTs for User Mode 194a\rangle // explained later
gdt_flush (); // Notify the CPU of changes
tss_flush (); // explained later
\end{verbatim}

Uses \texttt{fill\_gdt\_entry_{109c}}, \texttt{gdt_{92b}}, \texttt{gdt\_entry_{91}}, \texttt{gdt\_flush_{110b}}, \texttt{gp_{92b}}, and \texttt{tss\_flush_{197c}}.

\begin{verbatim}
(We've declared \texttt{gp_{92b}} and \texttt{gdt_{92b}} on page 92 after defining the C data structures for the GDT.) The function \texttt{gdt\_flush_{110b}} resides in the assembler file; it uses the \texttt{lgdt} instruction to load the new segment descriptors, sets all segment registers except \texttt{CS} to 0x10 and then makes a far jump for setting \texttt{CS} as well (to 0x08).

\begin{verbatim}
\langle start.asm 87\rangle +≡ ≫ 95a 144ί
[section .text]
    extern gp ; defined in the C file
global gdt_flush

\texttt{gdt\_flush: lgdt [gp]}
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax
    jmp 0x08:flush2

\texttt{flush2: ret}
\end{verbatim}

Defines:
\texttt{gdt\_flush}, used in chunks 110a, 111b, and 116b.
Uses \texttt{gp_{92b}}.

If you compare this code with the code for the higher half trick (see page 93), you will see that it basically does the same and just uses a different address in the \texttt{lgdt} instruction.

Effectively, the flat GDT sort of disables segmentation: the segmentation unit maps logical addresses to identical linear addresses (which are then translated into physical addresses by the paging unit).
4.4.4 Accessing the Video RAM

We need to access the video adapter’s text mode framebuffer which is mapped into the physical address space starting at 0xB8000 (and takes up 4 KByte of memory). So we provide a mapping for this memory as well:

```c
page_table video_pt __attribute__((aligned (4096))); // must be aligned!
```

Defines:
- `video_pt`, used in chunk 111b.
- `page_table` 104b.

We create a new page table (initialized with null entries) and from there we only create a mapping of 4 KByte (starting at 0xB8000).

```c
for (int i = 0; i < 1024; i++) {
    // null entries:
    fill_page_desc (&(video_pt.pds[i]), false,false,false,false,0);
}
```

KMAP (&(video_pt.pds[0xB8]), 0xB8*4096); // one page of video RAM

// enter new table in page directory
KMAPD (&(current_pd->ptds[0]), (memaddress (&video_pt) - 0xC0000000 );

gdt_flush ();
```

Uses `current_pd` 105a, `fill_page_desc` 100c, `gdt_flush` 110b, `KMAP` 101a, `KMAPD` 103c, `memaddress` 46c, and `video_pt` 111a.

4.5 Physical Memory: Page Frames in Ulix

The physical memory consists of page frames, some of which are already in use. When we dynamically assign frames to pages (i.e., change some page table), we need to know which frames are free and which are in use. For that purpose we use a bitmap (that we will call the frame table) which holds the current usage state of every frame. Since we have just set up the initial memory usage, we know exactly what our memory looks like at this point in time, so now is a good time to create and initialize that bitmap.

We assume that our system has 64 MByte of physical RAM. The size of the frame table depends on the size of the available physical memory which we define to contain `MEM_SIZE` many addresses. Dividing this by the `PAGE_SIZE` gives us the number of page frames. This of course assumes that `MEM_SIZE` is larger than `PAGE_SIZE` and both values are powers of two.

```c
#define MEM_SIZE 1024*1024*64 // 64 MByte
```

Defines:
- `MEM_SIZE`, used in chunks 112a and 499a.
\[ \text{constants 112a}\equiv (44a) 132\]  
\#define MAX_ADDRESS MEM_SIZE-1  // last valid physical address  
\#define PAGE_SIZE 4096  // Intel: 4K pages  
\#define NUMBER_OF_FRAMES (MEM_SIZE/PAGE_SIZE)

Defines:  
NUMBER_OF_FRAMES, used in chunks 112, 115b, 118c, and 613c.  
PAGE_SIZE, used in chunks 113b, 115c, 121b, 122a, 163–65, 167, 169b, 172a, 173a, 209b, 211b, 257, 261, 289c, 291, 293d, 294, and 298a.

Uses MEM_SIZE 111c.

The usage of main memory is directly reflected in the amount of frames which are not free. We will try to keep track of the number of free frames throughout the lifetime of the system in a global variable free_frames\[112b\].

\[ \text{global variables 92b}\equiv (44a) 111a 112c\]  
unsigned int free_frames = NUMBER_OF_FRAMES;

Defines:  
free_frames, used in chunks 112e, 119, 123c, 310a, 311b, 342b, 513e, 604b, and 613c.

Uses NUMBER_OF_FRAMES 112a.

So NUMBER_OF_FRAMES\[112a\] is the number of bits we need to store in the frame table. Since a byte holds eight bits, we need a structure that is NUMBER_OF_FRAMES\[112a\]/8 bytes large:

\[ \text{global variables 92b}\equiv (44a) 112b 115a\]  
char place_for_ftable[NUMBER_OF_FRAMES/8];  
unsigned int *ftable = (unsigned int*) &place_for_ftable;

Defines:  
ftable, used in chunks 112–14 and 603.  
place_for_ftable, used in chunk 603.

Uses NUMBER_OF_FRAMES 112a.

\[ \text{initialize system 45b}\equiv (44b) 111b 112e\]  
memset(ftable, 0xff, 128);  // all frames are free

Uses ftable 112c, memset 596c, and NUMBER_OF_FRAMES 112a.

Now we need to tell the frame table that some of our frames are already in use: We have two mappings for the first 4 MByte of physical RAM (even though we don’t use the first MByte at all). So we declare the first 4 MByte as used. 4 MByte contain 1024 pages, thus the first 1024 frames must be marked used. 1024/8 = 128; we set the first 128 bytes to 0xff = 11111111b. We also subtract the frames in these 4 MByte from free_frames\[112b\]:

\[ \text{initialize system 45b}\equiv (44b) 112d 115b\]  
memset(ftable, 0xff, 128);  
free_frames -= 1024;

Uses free_frames 112b, ftable 112c, and memset 596c.

4.5.1 Bitwise Manipulation

We want to be able to set and clear single entries in our frame table, so we have to access single bits: read them, write them and test them.

We can think of a frame number as consisting of
4.5 Physical Memory: Page Frames in Ulix

- an upper part that is an index into the frame table (which is built from 32-bit unsigned ints). Every such unsigned int stores 32 bits.
- and a lower part that is an offset whose value can lie between 0 and 31, giving a precise position within one such indexed unsigned int.

So we get \( \text{frameno} = 32 \times \text{index} + \text{offset} \), like this:

\[
\text{frameno} = \ldots i_4 i_3 i_2 i_1 i_0 o_4 o_3 o_2 o_1 o_0
\]

When we divide a frame number by 32, we find the unsigned int which stores the bit we’re searching for. The modulo function gives us the offset:

\[
\begin{align*}
\text{INDEX_FROM_BIT} &\equiv \frac{b}{32} & \text{// 32 bits in an unsigned int} \\
\text{OFFSET_FROM_BIT} &\equiv b \% 32
\end{align*}
\]

Defines:
\text{INDEX}\_FROM\_BIT, used in chunks 113b and 114a. 
\text{OFFSET}\_FROM\_BIT, used in chunks 113b and 114a.

The following two functions allow us to set or clear individual bits in the frame table:

\[
\begin{align*}
\text{set_frame} &\colon (\text{memaddress}, \text{frame_addr}) \\
\text{clear_frame} &\colon (\text{memaddress}, \text{frame_addr})
\end{align*}
\]

Note how individual bits are set or cleared:

- \(|=\) and \&=\) work in a similar way as \( +=\) for addition, however they perform “bitwise or” and “bitwise and”, respectively. So \( x|=y \) is short for \( x=x|y \) and \( x&=y \) is short for \( x=x\&y \).
- In the \text{set_frame} function, \( 1 <<= \) offset uses left shift to create a value whose offset’s bit is set (and all others are not), e.g. \( 1 <<= 3 \) is \( 00000000000000000000000000001000 \) (in binary notation).

Footnote: The macros \text{INDEX}\_FROM\_BIT and \text{OFFSET}\_FROM\_BIT, the functions \text{set_frame}, and \text{clear_frame} have been taken from http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html, they were slightly modified and adapted to Ulix.
• Next the corresponding unsigned int is “bitwise-or”ed with this value. That means: all bits which were already 1, remain 1; and the offset’s bit is being set (whatever its value was before).
• In a similar way the clear_frame function can clear a bit. It also starts with a shift operation, but the result goes through bitwise negation (~) which flips all bits. For example, ~(1 << 3) is 11111111111111111111111111110111.b. So there is exactly one 0 bit in there with all other bits being 1.
• Then the corresponding unsigned int is “bitwise-and”ed with this value. That means: all bits which were already 0, remain 0; and the offset’s bit is being cleared (whatever its value was before).

What remains is a function that can test a bit. It returns true (1) or false (0):

```
static boolean test_frame(unsigned int frame) {
    // returns true if frame is in use (false if frame is free)
    unsigned int index = INDEX_FROM_BIT(frame);
    unsigned int offset = OFFSET_FROM_BIT(frame);
    return (ftable[index] & (1 << offset)) >> offset;
}
```

Defines:
• test_frame, used in chunks 114b, 118c, 119b, and 613c.
Uses ftable, INDEX_FROM_BIT, and OFFSET_FROM_BIT.

A result of 0 means that a frame is available, whereas 1 means that the frame is already in use—which corresponds to the way we have already initialized a part of the frame table.

The function uses left and right shifts in order to always return either 0 or 1. If you never do any comparisons with 1, but only call the function in if statements such as

```
if (test_frame(frameno)) {
    // result non-0 (true); frame is not available
} else {
    // result 0 (false); frame is available
}
```

then you can skip the right shift at the end of the line and make the calculation a bit faster.

### 4.5.2 Direct Access to the Physical RAM

So far we haven’t encountered any conceptual problems, but consider this: The information stored in the page directories, page tables and in the frame table refers to physical memory. But the kernel has activated paging, and even though it runs with the most privileges any code on the machine can get, it cannot directly access physical memory. Yet it has to modify or create new page tables and it has to update the frame table. So the kernel needs to have an understanding of what is going on in the physical memory, without accessing it.
If physical memory is very small in comparison to the virtual address space, it is possible to permanently map all of the RAM into some area of the kernel’s virtual address space. For our code we assume that the machine has only 64 MByte of RAM—compared to the 4 GByte address space that is not much. We can spare 64 MByte of the virtual kernel memory and sponsor a mapping to this physical RAM. We will put this in the area 0x00000000 ...0x03FFFFFF, so that any physical address \(x\) can be accessed via the virtual address \(x + 0x00000000\). However, there’s a cost: the corresponding page table entries will require some room: 64 MByte = 16384 pages, so we will need 16384 page table entries each of which uses 4 bytes. Thus, it requires 16 pages (64 KByte) to store the extra page tables. Where can we put these tables? To simplify things we’ll declare yet more static variables which hold our 16 tables:

```c
[49x544]⟨global variables 92b⟩+
page_table kernel_pt_ram[16] __attribute__((aligned (4096)));
[49x544]⟨initialize system 45b⟩+
for (uint fid = 0; fid < NUMBER_OF_FRAMES; fid++) {
    ⟨map page starting at 0x00000000 + PAGE_SIZE*fid to frame fid 115c⟩
}
```

The code for this mapping is not too complicated, either:

```c
KMAP( &(kernel_pt_ram[fid/1024].pds[fid%1024]), fid*PAGE_SIZE ) ;
```

(Note that instead of \&(kernel_pt_ram[fid/1024].pds[fid%1024]) we could have used \&(kernel_pt_ram[0].pds[fid]) which would access out of bound indices of kernel_pt_ram[0].pds, but since these arrays are arranged one after the other without other data in between, it would work as well.)

To finalize this, we have to enter the 16 new page tables in 16 page directory entries. Note that we need the physical addresses of the page tables, not the virtual ones. While \&(kernel_pt_ram[115a][i]) delivers the virtual address just fine, it does not help to write it into the page directory. Subtracting 0xC0000000 does the job: we know that we loaded the kernel at 0x100000 with addresses starting at 0xC0100000, so we just need to subtract that artificial offset, and we’re good.

```c
KMAPD( &(current_pd->ptds[832+i]), physaddr );
```

```c
kputs(“RAM: 64 MByte, mapped to 0x00000000-0x0D3FFFFFF\n”);
```
Since we will often have to access a physical address, we’ll define a macro `PHYSICAL` that will translate an address from the first 64 MByte to the 0x0000.0000 ...0x03FF.FFFF range:

```c
#define PHYSICAL(x) ((x)+0xd0000000)
```

Defines: `PHYSICAL`, used in chunks 116c, 117, 121–23, 165b, 166a, 171c, 209b, 211c, 293d, 294, 307a, 308c, 327b, 329b, 335b, and 609.

Now that we can access all of the physical addresses (including video memory) we can get rid of the video mapping for 0xb8000 ...0xb9000, we’ll actually remove the first entry of the page directory which so far mapped part of the first (virtual) 4 MByte.

```c
VIDEORAM = 0xD00B8000;
// remove first page table (including the old video mapping)
fill_page_table_desc (&current_pd->ptds[0], 0, 0, 0, 0);
gdt_flush ();
```

Uses `current_pd` 105a, `fill_page_table_desc` 103b, `gdt_flush` 110b, and `VIDEORAM` 327b.

We define a macro which casts `VIDEORAM` into a word pointer which will later be helpful for accessing individual characters on the screen (they are encoded as words, not bytes):

```c
#define textmemptr ((word*)VIDEORAM)
```

Defines: `textmemptr`, used in chunks 329b, 335b, and 609. Uses `VIDEORAM` 327b.

### 4.5.2.1 MMU Emulation

Sometimes we want to find out what frame is used by a page. We present a function

```c
unsigned intpageno_to_frameno(unsigned int pageno);
```

for this purpose which basically works like the MMU when it translates addresses: It uses the fact that for a page number `pageno` we first look at entry `pageno/1024` of the page directory, locate the referenced page table and then look at entry `pageno%1024` of that page table. When the page is not mapped to a frame, the function returns -1:

```c
unsigned intpageno_to_frameno(unsigned int pageno) {
    unsigned int pdindex = pageno/1024;
    unsigned int ptindex = pageno%1024;
    if ( !current_pd->ptds[pdindex].present ) {
        return -1; // we don't have that page table
    } else {
        // get the page table
        page_table *pt = (page_table*)
            (PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12 ));
```
if ( pt->pds[ptindex].present ) {
    return pt->pds[ptindex].frame_addr;
} else {
    return -1;  // we don't have that page
}

#define PEEK(addr)    (*((byte *)addr))
#define POKE(addr, b) (*((byte *)(addr) = (b))
#define PEEKPH(addr)  (*((byte *)(PHYSICAL(addr)))
#define POKEPH(addr, b) (*((byte *)(PHYSICAL(addr)) = (b))

#define PEEK_UINT(addr) (*((uint *)(addr))
#define POKE_UINT(addr, b) (*((uint *)(addr) = (b))
#define PEEKPH_UINT(addr) (*((uint *)(PHYSICAL(addr)))
#define POKEPH_UINT(addr, b) (*((uint *)(PHYSICAL(addr)) = (b))

4.5.3 PEEK and POKE Functions

If you remember home computers like the Commodore C64 or the Schneider/Amstrad CPC, their built-in Basic interpreters often had a way to directly access memory contents. The classical command names were PEEK\textsubscript{117} (for reading) and POKE\textsubscript{117} (for writing). Here they are again: They convert an address into a pointer to a byte and then read or write.

(Credits to Dan Henry who supplied the first two lines of this code on http://www.keil.com/forum/8275/.)

Note that frame_addr holds (the upper 20 bits of) a physical address. Luckily we have a mapping of the physical address space to \texttt{0xd000.0000} and above that we can access via the \texttt{PHYSICAL} macro—otherwise we would have no way of accessing the page table.

PEEK\textsubscript{117} and POKE\textsubscript{117} read and write bytes using virtual addresses, PEEKPH\textsubscript{117} and POKEPH\textsubscript{117} do the same with physical addresses (using our \texttt{0x00000000} trick with the \texttt{PHYSICAL} macro), and finally PEEK\texttt{\_UINT}\textsubscript{117}, POKE\texttt{\_UINT}\textsubscript{117}, PEEKPH\texttt{\_UINT}\textsubscript{117} and POKEPH\texttt{\_UINT}\textsubscript{117} do the same as the first four functions but work with unsigned 32-bit integers instead of bytes.

With the little-endian ordering of larger integers, the following code
4 Boot Process and Memory Management in Ulx

(peek and poke example 118a) ≡

unsigned int testvar;
unsigned int address = (unsigned int)&testvar;
POKE (address, 0x12);
POKE (address+1, 0x34);
POKE (address+2, 0x56);
POKE (address+3, 0x78);
printf ("32-bit value: 0x%\n", PEEK_UINT (address));

prints
32-bit value: 0x78563412
(and not 0x12345678).

4.5.4 Allocating and Releasing Frames

So far we have not used any dynamically generated data structures in the kernel, so there
was no need for some kind of allocation function for the kernel.

When we start creating processes, we will need to reserve (virtual) memory for those
processes, and there may also be areas in the kernel which need memory.

So we will start simple: with a function that requests a new frame of physical memory.
It has the following definition:

(int request_new_frame () {

( find a free frame und reserve it 118c)

);

Defines:
request_new_frame, used in chunks 121a, 164–66, 173a, 192a, 211a, 257c, 291, 297, and 608b.

This alone will not be all too useful—only in combination with entering it in some paging
table that memory will be accessible (unless code uses the mapping of the physical RAM
to 0x0000.0000 and above).

Finding a free frame is simple: We look at the frame table and return the first available
frame:

(unsigned int frameid;
boolean found;
start_find_frame:
found = false;
for (frameid = 0; frameid < NUMBER_OF_FRAMES; frameid++) {
  if ( !test_frame (frameid) ) {
    found=true;
    break;   // frame found
  }
}

Uses frameid, NUMBER_OF_FRAMES 112a, and test_frame 114a.
Then we use `set_frame` to mark the frame used and return the frame ID:

\[
\begin{align*}
\text{if (found)} & \text{ } \{ \\
\quad \text{set_frame (frameid*4096); } \\
\quad \text{free_frames--;} \\
\quad \text{return frameid; }
\}
\text{else } \{ \\
\quad \langle \text{page replacement: free one frame} \rangle \quad \text{// will be explained later} \\
\quad \text{goto start_find_frame; } \\
\quad \text{// return -1; // never fail}
\}
\end{align*}
\]

Uses `frameid`, `free_frames`, and `set_frame`.

Note that the function clears a frame if no free one is available—we will explain the code chunk `⟨page replacement: free one frame⟩` in Chapter 9.4.

We'll add code for releasing a frame: basically we just call `clear_frame`, but we also need to modify `free_frames`:

\[
\text{void release_frame (int frame)} \{ \\
\quad \text{if (test_frame (frame)) } \{
\quad \quad \text{// only do work if frame is marked as used} \\
\quad \quad \text{clear_frame (frame << 12); } \\
\quad \quad \text{free_frames++; }
\quad \}\;
\}
\]

Defines:
- `release_frame`, used in chunks 123c, 167c, 169a, 261, and 296.
- `clear_frame, free_frames`, and `test_frame`.

We may call `release_frame` for unused frames which will have no effect.

### 4.6 Managing Pages in Ulix

Since we have now established a mechanism for reserving frames, we can proceed with page requests. You have already seen all the required data structures when we initialized paging in Chapter 4.4 (pp. 97 f.).

#### 4.6.1 Allocating Pages

Now we need to implement functions for dynamically requesting new pages and releasing them after they are no longer needed. Both are only possible via requesting and releasing frames, and we need to update existing page directories and page tables as well as occasionally create new page tables.

\[
\text{void *request_new_page ();} \\
\text{void *request_new_pages (int number_of_pages);}
\]

---

4.6 Managing Pages in Ulix

Since we have now established a mechanism for reserving frames, we can proceed with page requests. You have already seen all the required data structures when we initialized paging in Chapter 4.4 (pp. 97 ff.).

#### 4.6.1 Allocating Pages

Now we need to implement functions for dynamically requesting new pages and releasing them after they are no longer needed. Both are only possible via requesting and releasing frames, and we need to update existing page directories and page tables as well as occasionally create new page tables.

\[
\text{void *request_new_page ();} \\
\text{void *request_new_pages (int number_of_pages);} 
\]
Getting one page is just a special case of getting several ones:

\[\text{function implementations} + \cdots\] (44a) <119b 120d>

\begin{verbatim}
void *request_new_page () { return request_new_pages (1); }
\end{verbatim}

Defines:
request_new_page, used in chunks 164a, 211a, and 608b.
Uses request_new_pages 120b.

The real work must be done here:

\[\text{function implementations} + \cdots\] (44a) <120a 122d>

\begin{verbatim}
void *request_new_pages (int number_of_pages) {
  \langle find contiguous virtual memory range 120c \rangle
  \langle enter frames in page table 121a \rangle
};
\end{verbatim}

Defines:
request_new_pages, used in chunks 119c, 120a, and 608b.

That function is more useful, but there’s a lot to do in the two code chunks:

- In \langle find contiguous virtual memory range 120c \rangle we need to find a contiguous block of virtual memory addresses which are unused so far. That’s important if, say, we want to reserve memory for a large array of data which will be spread across several pages: We need the virtual address range to have no “holes” so that accessing array entries by index and pointer arithmetic work properly. With 1 GByte of virtual (kernel) memory available we expect that this will always be possible. In order to find unmapped pages we use the function \text{mmu_p} (), an improved version of \text{pageno_to_frameno}.  

- For the next step \langle enter frames in page table 121a \rangle we must reserve a frame for each page we want to map and enter its address in the page table.

\[\text{find contiguous virtual memory range} 120c\] (120b)

\begin{verbatim}
unsigned int first_page = 0xc0000; // first page
unsigned int count = 0; // number of contiguous pages
while (count < number_of_pages && first_page+count ≤ 0xfffff) {
  if ( mmu_p (current_as, first_page + count) == -1 ) {
    count++;
  } else {
    // the block we just looked at is too small
    first_page += (count+1); // restart search
    count = 0;
  }
}
if (count != number_of_pages) return NULL; // could not find a sufficiently large area
\end{verbatim}

Uses current_as 170b, first_page, mmu_p 171c, and NULL 46a.

(The function mmu_p does the same as pageno_to_frameno, see page 116, but it works with address spaces which we have not yet defined—we’ll introduce them when we implement the process system. current_as 170b, refers to the current address space, which you can ignore right now.)

There is one condition under which simply entering the data in the page table will fail: if we fill the last entry of the page table, it will afterwards be full, and the next attempt
to create a new page will find no place to store it. Then it will be too late to create a new page table (because that new page table must also have a virtual address).

So we check now whether we're attempting to fill the last entry.

\[
\text{for } (\text{int } \text{pageno} = \text{first_page}; \ \text{pageno} < \text{first_page}+\text{count}; \ \text{pageno}++) \{ \\
\text{int } \text{newframeid} = \text{request_new_frame }(); \ // \text{get a fresh frame for this page} \\
\text{if } (\text{newframeid} == -1) \{ \ // \text{exit if no frame was found} \\
\text{// this can only happen if the swap file is full} \\
\text{return NULL;} \\
\}
\]

\text{unsigned int } \text{pdindex} = \text{pageno}/1024; \\
\text{unsigned int } \text{ptindex} = \text{pageno}\%1024; \\
\text{page_table }*\text{pt}; \\
\text{if } (\text{ptindex} == 0 \&\& !\text{current_pd->ptds[pdindex].present}) \{
\text{// new page table!} \\
\text{\langle create new page table \rangle 122a} \\
\text{newframeid} = \text{request_new_frame }(); \ // \text{get yet another frame} \\
\text{if } (\text{newframeid} == -1) \{ \\
\text{return NULL;} \ // \text{exit if no frame was found} \\
\text{// again, this can only happen if the swap file is full} \\
\}
\}
\]

Now we need to access the page directory and the right page table again:

\[
\text{if } (\!\text{current_pd->ptds[pdindex].present}) \{ \\
\text{// we don't have that page table -- this should not happen!} \\
\text{kputs("FAIL! No page table entry\n");} \\
\text{return NULL; } \\
\}\text{else} \{ \\
\text{// get the page table} \\
\text{pt = (page_table*)\text{PHYSICAL}(current_pd->ptds[pdindex].frame_addr << 12 }) ;} \\
\text{// enter the frame address} \\
\text{KMAP ( &(pt->pds[ptindex]), newframeid \* PAGE_SIZE );} \\
\text{// invalidate cache entry} \\
\text{asm volatile("invlpg %0": : "m"(*(char*)(pageno<<12)) );} \\
\}\]

Finally, we clear the new page and return a pointer to the new page:

\[
\text{\langle enter frames in page table \rangle 121a} \quad \equiv \quad (120b) \text{ \langle 121a 121c \rangle} [121b] \\
\text{memset }((\text{void}*) \text{(pageno\*4096)}, 0, 4096); \\
\text{return }((\text{void}*) \text{(first_page\*4096)});
\]

(\text{The last line executes the invlpg instruction which invalidates the cache entry for the modified page if one exists.})
If a page table does not yet exist, it has to be created and referenced from the page directory. On-the-fly creation of a new page table works like this: We have a frame at newframeid which we can use, its physical address is newframeid << 12. It is not mapped (so it has no virtual address); we’ll deal with this fact later. We can use our PHYSICAL function to “talk” to it directly: The address is PHYSICAL(newframeid<<12). So we create the page table and fill it with zeroes:

```
(create new page table 122a)≡
  pt = (page_table*) PHYSICAL(newframeid<<12);
  memset (pt, 0, PAGE_SIZE);
```

We need to tell the page directory that this page table is responsible for the next chunk of memory. When we calculated pdindex and ptdindex above, we found that ptdindex is 0, and pdindex points to the page directory entry that is currently empty. So it is just pdindex which we have to use:

```
// KMAPD(&(*(current_pd->ptds[pdindex])), newframeid << 12);
```

The line above is commented out; it used to work before we introduced processes (and address spaces). The following code is a variation of the above line which updates all page directories (each process has its own one). This will become clear when you reach the process chapter.

```
for (addr_space_id asid=0; asid<1024; asid++) {
  if (address_spaces[asid].status == AS_USED) { // is this address space in use?
    page_directory *tmp_pd = address_spaces[asid].pd;
    KMAPD (&(tmp_pd->ptds[pdindex]), newframeid << 12);
  }
}
```

Note: these new page tables only exist physically. Their frames are marked as used, but no virtual addresses point to them. Is that a problem? We can always get their physical addresses through the page directory. So we should be fine.

### 4.6.2 Releasing Pages

Now we are able to request new pages, but occasionally we will also want to release them. That is much simpler: In order to release a page, we simply have to

```
void release_page (unsigned int pageno) {
  (remove page to frame mapping from page table 123a)
  (release corresponding frame 123c)
};
```

Defines:
release_page, used in chunks 123d, 166, 167, and 169a.
First we have to get rid of the page mapping, i.e., we need to find the entry in the correct page table and replace it with a null entry. The lookup code is similar to the code for creating the new entry (in ⟨enter frames in page table 121a⟩). However, we first test whether a page mapping exists, because if not, we can return immediately:

\[
\begin{align*}
\text{remove page to frame mapping from page table 123a} & \equiv \quad (122d) \quad 123b \quad [123a] \\
& \quad \text{// int frameno = pageno_to_frameno (pageno);} \\
& \quad \text{int frameno = mmu_p (current_as, pageno);} \quad \text{// we will need this later} \\
& \quad \text{if (frameno == -1) \{ return; \}} \quad \text{// exit if no such page}
\end{align*}
\]

Uses current_as 170b, mmu_p 171c, and pageno_to_frameno 116e.
(As you can see, this code originally used pageno_to_frameno 116e. However, with the introduction of address spaces, this does not work any longer since pageno_to_frameno 116e is only aware of the first address space (that belongs to the kernel). As already mentioned, we will provide an mmu_p 171c function which is very similar to pageno_to_frameno 116e, but takes an extra argument which lets us specify the address space. The variable current_as 170b always stores the ID of the currently active address space.)

Next we look up the right entry and set it to zero:

\[
\begin{align*}
\text{remove page to frame mapping from page table 123a} & + \equiv \quad (122d) \quad <123a \quad [123b] \\
& \quad \text{unsigned int pdindex = pageno/1024;} \\
& \quad \text{unsigned int ptindex = pageno%1024;} \\
& \quad \text{page_table *pt;} \\
& \quad \text{pt = (page_table*) (PHYSICAL(current_pd->ptds[pdindex].frame_addr << 12));}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{// write null page descriptor} \\
& \quad \text{fill_page_desc ( & (pt->pds[ptindex]), false, false, false, false, 0 );}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{// invalidate cache entry} \\
& \quad \text{asm volatile ("invlpg %0" : : "m"(*(char*) (pageno<<12)) ;}
\end{align*}
\]

Uses current_pd 105a, fill_page_desc 100c, page_table 101b, PHYSICAL 116a, and write 429b.

We need to invalidate the cache entry for this page so that any further access to addresses inside the page lead to page faults.

Lastly, we free the frame—we have the release_frame 119b function for that:

\[
\begin{align*}
\text{release corresponding frame 123c} & \equiv \quad (122d) \quad [123c] \\
& \quad \text{release_frame (frameno);} \quad \text{// note: this increases free_frames}
\end{align*}
\]

Uses free_frames 112b and release_frame 119b.

That’s all there is to it.
Sometimes we will want to release a whole consecutive range of pages, so we’ll add an extra function for this purpose:

\[
\begin{align*}
\text{function implementations 100b} & + \equiv \quad (44a) \quad 122d \quad 133b \quad [123d] \\
& \quad \text{void release_page_range (unsigned int start_pageno, unsigned int end_pageno) \{} \\
& \quad \quad \text{for (int i = start_pageno; i < end_pageno+1; i++) release_page (i);} \\
& \quad \};
\end{align*}
\]

Defines: release_page_range, used in chunk 608b.
Uses release_page 122d.
4.7 Next Steps

The system is now initialized and uses paging to manage the physical memory. However, whenever something goes wrong, the system will halt or reboot, and it cannot access any hardware, except the video card: Since the video memory can be accessed like normal RAM, we can display status or error messages, but that is all. Especially the code allows no interaction, because we cannot read the keyboard’s status.

In the upcoming chapter we describe the mechanism that is required to handle faults (something went wrong) and interrupts (some device wants to inform the system about an event). With interrupts we can also have timers, and that brings us one step closer to working with processes which need working timers so that we can switch back and forth between several programs that run simultaneously.

4.8 Exercises

This is the first set of exercises. You will need the development environment that you can download from the Ulix website (http://www.ulixos.org/). Install the virtual machine in VirtualBox and login as ulix (with password ulix).

Tutorial 1

1. On the Ulix development system, locate the directory /home/ulix/tutorial/01/. This directory contains parts of the code that we’ve presented so far; we’ve added the printf function which lets the C program write to the screen.

   Read the source code files ulix.c and start.asm. Compile the sources with make and run the kernel with make run. You should see the following output:

   ```
   Booting 'ULIX-1386 (c) 2008-2011 Felix Freiling & Hans-Georg Esser'
   root (fd0)
   Filesystem type is fat, using whole disk
   kernel /ulix.bin
   [Mullihout-elf, <0x100000:0x46:0x0>, <0x100000:0x1b0:0x0>, <0x101000:0x0:0x9
   000>, shtab=0x10a190, entry=0x1002a1]

   Hello World! This is not Ulix yet :)

   address of main() [ulix.c]: c01005b0
   address of start [start.asm]: 0010002a
   stack: c0101000 - c0109000
   ```

2. Obviously, the kernel executes the C function main. How does the system jump from the early assembler code (beginning with the start label) into the C function?

3. The file ulix.dump contains a listing of the generated assembler code. Here you can find all labels from the assembler file start.asm and also the function names from ulix.c. Search the file for the labels start, higherhalf and main and check for which memory addresses the corresponding code has been generated. (You find the memory addresses on the very left in hexadecimal format without leading “0x”.)
Do you recognize the GDT trick in the assembler code? It is initiated via a “long jump” (jmp) with a logical address segment:address as jump target.

Do also search for the labels stack_first_address and stack_last_address and compare the shown addresses with those from the VM’s output (in the last line)—they should match.

4. The kernel uses the printf() function (you can find its code in the separate printf.c file) for text output, but its main task to is to format the output as requested by the format strings (such as %s for strings or %d for integers) and send it to the terminal character-wise via the kputch function whose implementation resides in ulix.c. How does kputch write characters to the screen? This will also be explained later when we discuss the corresponding code section of the full Ulix sources. It may help to search the web or this book for “0xb8000” and “video”. Consider how pointers can be used for accessing memory: The commands

char *mem; mem = (char*) 0x1234; *mem = 'a';

...can write the byte ‘a’ (ASCII value: 0x61) to the memory address 0x1234.

5. Why does the following line from kputch()

```
screen = (char*) 0xc0000000 + 0xb8000 + posy*160 + posx*2;
```

use multiplicators 160 and 2, and why does it add 0xc0000000?

6. Use the command objdump -h ulix.bin to check which memory areas are used by the three sections .setup, .text and .bss. (You can ignore the additional sections .comment, .stab and .stabstr.) Compare the values with the information that the boot loader GRUB outputs in the “Multiboot-elf,...” line when it loads the kernel.

7. The folder tutorial/02/ in the ulix/ directory contains an improved version of the Ulix kernel which implements paging.

Read the source code files ulix.c and start.asm and localize the code chunks which you have seen in this chapter. (There is also additional code that we have not discussed yet.)

8. Compile/assemble and link the code with make and start the system with make run.

9. In ulix.c the kputch() function has seen some slight changes, there is now the following code block:

```
if (paging_ready)
    screen = (char*) 0xb8000 + posy*160 + posx*2;
else
    screen = (char*) 0xc0000000 + 0xb8000 + posy*160 + posx*2;
```

It evaluates the paging_ready variable which is initially set to false and changed to true after paging has been initialized. In that case, adding 0xc0000000 is no longer necessary for calculating the address (cf. exercise 5). Why does that work?
10. (Literate Programming) Convert the files ulix.c and start.asm (from tutorial/02/) into a literate program tutorial02.nw. You can restrict the documentation that you add to some keywords and roughly follow the ordering of descriptions in this chapter.

11. Test that you can reconvert the literate program into the original code files (or at least sufficiently similar versions which also compile and generate a working kernel). Also create a LATEX file and from that a PDF version.

劝译 3.12. In the folder tutorial/03/ you find yet another version of the ULIX kernel which contains an improved version of the paging code. It is a literate program (ulix.nw). cd into the folder and use make to extract the source code files ulix.c und start.asm from ulix.nw. They will automatically be compiled or assembled. Then launch the kernel with make run. The system performs some tests of memory management and then halts.

Also take a look at the PDF file ulix.pdf (this file is written in German language, so it is likely that you want to skip this step) that you can recreate with make pdf if you apply changes to ulix.nw. In the document you can find a description of the code for frame and page management; it is also a sample solution for exercise 10. Compare how the code is broken down into code chunks in your own solution and in the sample solution.

13. We have defined the page table descriptors (page_table_desc102) and the page descriptors (page_desc100a) as structures. Since they are both exactly 32 bits large, there is an alternative interpretation as unsigned int. The goal of this exercise is to modify the literate program so that it works with these simple integer types instead of the structures.

First, create a copy of the folder (so that you can keep the original files). If you cd into the tutorial folder, you can do that with the

```
cp -r 03 03-copy
```

command and make all your changes in 03-copy/.

a) Start with changing the type declarations for page_table_desc und page_desc to

```c
typedef unsigned int page_table_desc;
typedef unsigned int page_desc;
```

b) The types page_directory and page_table are not needed any longer; instead you can declare individual directories or tables like this:

```c
page_table_desc pd[1024] __attribute__((aligned(4096)));
page_desc pt[1024] __attribute__((aligned(4096)));
```

That way, you can access entry n of pt via pt[n] instead of pt.pds[n]. The expression pt (without an index) serves as a pointer (of type unsigned int*) to the start of the table. If you need to hand over a single descriptor to functions like fill_page_desc() or fill_page_table_desc(), you can pass them a pointer to pt[n], i.e., &pt[n].
c) After the changes all functions which work with these types are broken, you need to modify them. For example, in order to fill a page descriptor in `fill_page_desc()`, you can take an address `frame_addr` and set its lowest twelve bits to 0:

```c
  tmpvalue = frame_addr & 0b11111111111111111111000000000000;
```

or

```c
  tmpvalue = frame_addr & 0xFFFFF000;
```

(The hexadecimal number combines four bits to one hex digit.) Then you add the flags in the lower twelve bits. You could define flag constants which are based on the bit positions, e.g.,

```c
#define FLAG_PRESENT 1<<0 // Bit 0: present
#define FLAG_ACCESSSED 1<<5 // Bit 5: accessed
```

etc. Then use a bitwise “or” operation (`|`), e.g.

```c
  tmpvalue = tmpvalue | FLAG_ACCESSSED;
```

to set a specific bit. When everything is done, you can write the value with `*desc = tmpvalue;`. (You must also modify the function prototype and pass a pointer to `unsigned int` when you call the function.)

d) In order to extract the address from a descriptor, you simply set the lowest twelve bits to 0 (like above). On the other hand, you can extract single flags by performing a bitwise “and” operation (`&`) with the appropriate `FLAG_*` constant and test whether the result is 0 or not:

```c
  if ((descriptor & FLAG_ACCESSSED) == 0) { /* flag is not set */ }
```

14. Verify that the old and new program versions work identically. In both versions you can use the `hexdump` function which displays a memory region as hex dump. Use the starting and ending addresses of the page table or page directory as arguments, e.g.

```
  hexdump ( (unsigned int)current_pd, (unsigned int)current_pd + 4096 );
```

(which will output the whole page-sized table). The output is written to the file `output.txt` (in the current directory of the development VM), you can later compare them:

```
  cd 03/; make; make run > output.txt
  cd 03-copy/; make; make run > output.txt
  cd ..; diff ~/03*/output.txt
```

If `diff` creates no output, the two files are identical. (The `hexdump` calls must occur after the test changes to the tables.)

15. Check with `make pdf` that your literate program can still be converted to a PDF file—if that does not work, identify and remove the errors.
5

Interrupts and Faults

All modern CPUs and even many of the older ones such as the Zilog Z80 8-bit processor can be interrupted: the CPU has an input line which can be triggered by an external device connected to this line. When such an interrupt occurs, the current activity is suspended, and the CPU continues operation at a specified address: it executes an interrupt handler.

In principle a device could be directly connected to the CPU, but modern machines contain many devices which want to interrupt the processor, e.g. the disk controllers, the keyboard controller, the serial ports, or the on-board clock. Thus an extra device, called the interrupt controller, intermediates between the other devices and the CPU. One of the advantages of such an interrupt controller is that it is programmable: it is possible to enable or disable specific interrupts whereas the CPU itself can only completely enable or disable all interrupts, using the sti (set interrupt flag) and cli (clear interrupt flag) instructions. (These machine instructions exist on Intel-x86-compatible CPUs; other chips have similar instructions.) Being programmable also means that interrupt numbers can be remapped (we will see later why this is helpful). Interrupt controllers with these features are called programmable interrupt controllers (PICs), and we’ll use that abbreviation throughout the rest of this chapter.

After the implementation of interrupts we will also take a look at fault handling since the involved mechanisms are very similar to those which we need for handling interrupts. As we mentioned in the introduction, the main difference between interrupts and faults is that faults occur as a direct consequence of some specific instruction that our code executes. In that sense they are synchronous. Interrupts on the other hand occur without any connection to the currently executing instruction, since they are not triggered (immediately) by our code but by some device. That is why they are called asynchronous.
5.1 Examples for Interrupt Usage

Interrupt handling is a core functionality which is used in lots of places: without interrupts we would not be able to build a useful operating system.

Let’s look at some example features of Ulix which depend heavily on interrupts:

**Multi-tasking** Ulix can execute several processes in parallel and switch between them using a simple round-robin scheduling mechanism. That is only possible because the clock chip on the motherboard regularly generates timer interrupts, and Ulix installs a timer interrupt handler which—when activated—calls the scheduler to check whether it is time to switch to a different process. If there were no interrupts, we could only implement *non-preemptive scheduling* which relies on the processes to give up the CPU voluntarily.

**Keyboard input** Whenever you press or release a key on a PC, either event generates an interrupt. Ulix picks up these interrupts and the keyboard interrupt handler reads a key press or key release code from the controller.

A keyboard driver does not need interrupts, but the alternative is to constantly poll (query) the keyboard controller in order to find out whether a new event has occurred. That’s possible but wastes a lot of CPU time. Polling does not work well in a multi-tasking environment. (However for a single-tasking operating system it may be good enough.)

**Media** Reading and writing hard disks and floppy disks also depends on interrupts: In the Ulix implementation of filesystems (and disk access) a process which wants to read or write makes a system call which sends a request to the drive controller. Then Ulix puts the calling process to sleep. Once the request has been served, the drive controller generates an interrupt, and the interrupt handler for the hard disk controller or the floppy disk controller (these are two separate handlers) deals with the data and wakes up the sleeping process.

Again, this could be done without interrupts. But the process would have to remain active and continuously poll the controller to find out whether the data transfer has been completed.

**Serial ports** Finally, the serial ports are similar to the keyboard, since all of them are *character devices*: they transfer single bytes (instead of blocks of bytes).

5.2 Interrupt Handling on the Intel Architecture

**Intel 8259 PIC** The classical IBM PC used the Intel 8259 Programmable Interrupt Controller, compatible descendents of which are still used in modern computers. The 8259 has eight input lines (through which up to eight separate devices may connect) and one output line which forwards received interrupt signals to the CPU. It is possible to use more than one 8259 PIC
since these controllers can be cascaded which means that a second controller’s output pin is connected with one of the first controller’s input pins (typically the one for device 2, see Figure 5.1). With that cascade, devices connected to the first controller keep their normal numbers (0, 1, 3–7 with 2 reserved for the second controller), and devices connected to the second controller use device numbers between 8 and 15, allowing for a total of 15 (= 16 − 1) separate device numbers. The first or primary controller is called Master PIC, the second one is the Slave PIC (as it is not directly connected to the CPU but relies on the master PIC to have its interrupts signals forwarded). The numbers 0–15 are called Interrupt Request Numbers (IRQs).

As you can see from the figure, there is a fixed mapping of some devices to specific IRQs. We will use the following IRQs in the Ulix implementation:

- **0: Timer Chip.** On a PC’s mainboard you can find a (programmable) timer chip which regularly generates interrupts. We will use timer interrupts to call the scheduler (besides other tasks).
- **1: Keyboard.** This is the interrupt generated by PS/2 keyboards. A USB keyboard would be handled differently, but we do not support USB devices.
- **2: Slave PIC.** As already mentioned, IRQ 2 is reserved for connecting the secondary (slave) PIC.
- **3: Serial Port 2.** The second serial port will be used for our implementation of what we’ve called the serial hard disk—you can find it in Chapter 13.4.
- **4: Serial Port 1.** We only use the first serial port for output (when running Ulix in a PC emulator), thus we will not install an interrupt handler for this IRQ.
- **6: Floppy.** This is the IRQ for the floppy controller. It can handle up to two floppy drives.
- **14: Primary IDE Controller.** And finally, 14 is the IRQ of the primary IDE controller. Many PC mainboards contain two controllers, with each of them allowing two drives to connect. The secondary IDE controller would generate the interrupt number 15,
but we’re going to support only one controller.

We can define names for the IRQ numbers right now:

```c
#define IRQ_TIMER 0
#define IRQ_KBD 1
#define IRQ_SLAVE 2 // Here the slave PIC connects to master
#define IRQ_COM2 3
#define IRQ_COM1 4
#define IRQ_FDC 6
#define IRQ_IDE 14 // primary IDE controller; secondary has IRQ 15
```

Defines:
- IRQ_COM1, used in chunk 344c.
- IRQ_COM2, used in chunks 344c and 520a.
- IRQ_FDC, used in chunk 552c.
- IRQ_IDE, used in chunk 534b.
- IRQ_KBD, used in chunk 323b.
- IRQ_SLAVE, used in chunk 139b.
- IRQ_TIMER, used in chunk 339a.

5.2.1 Using Ports for I/O Requests

**Going where?** We want to initialize the PICs, which means directly talking to these controllers. Like with most other devices we can use the machine instructions `in` and `out` to find out the PIC’s current status and tell it what to do. Here we provide the code which lets us access the controllers.

**I/O Ports** Access to many hardware components (including the PICs) is possible via **I/O ports**. Using `in` and `out` machine instructions it is possible to transfer bytes, words or doublewords between a CPU register and a memory location or register on some device (such as a hard disk controller).

The Intel 80386 Programmer’s Reference Manual [Int86, pp. 146–147] explains:

“The I/O instructions `IN` and `OUT` are provided to move data between I/O ports and the `EAX` (32-bit I/O), the `AX` (16-bit I/O) or `AL` (8-bit I/O) general registers. `IN` and `OUT` instructions address I/O ports either directly, with the address of one of up to 256 port addresses coded in the instruction, or indirectly via the `DX` register to one of up to 64K port addresses.

`IN` (Input from Port) transfers a byte, word or doubleword from an input port to `AL`, `AX` or `EAX`. If a program specifies `AL` with the `IN` instruction, the processor transfers 8 bits from the selected port to `AL`. If a program specifies `AX` with the `IN` instruction, the processor transfers 16 bits from the port to `AX`. If a program specifies `EAX` with the `IN` instruction, the processor transfers 32 bits from the port to `EAX`.

`OUT` (Output to Port) transfers a byte, word or doubleword to an output port from `AL`, `AX` or `EAX`. The program can specify the number of the port using the same methods as the `IN` instruction.”
For accessing 8-bit, 16-bit and 32-bit ports, the Intel assembler language provides separate commands `inb / outb` (byte), `inw / outw` (word) and `inl / outl` (long: doubleword) which make it explicit what kind of transfer is wanted. We’ll use them in the functions:

```c
#define byte inportb (word port) { byte retval; asm volatile("inb %%dx, %%al" : =a"{retval} : =d"{port}); return retval;
void outportw (word port, byte data) { asm volatile("outb %%al, %%dx" : =d"{port}, =a"{data});
void outportw (word port, word data) { asm volatile("outw %%ax, %%dx" : =d"{port}, =a"{data});
```

There are several possible C implementations with inline assembler code, the following code is most readable:

```c
#define byte inportb (word port) { byte retval; asm volatile("inb %%dx, %%al" : =a"{retval} : =d"{port}); return retval;
```

We could provide `inportl` and `outportl` (for 32-bit values) in a similar fashion, using `inl`, `outl` and the 32-bit register `EAX` (instead of the 16-bit and 8-bit versions `AX` and `AL`), but we do not need them. (Remember that `EAX`, `AX` and `AL` are (parts of) the same register, see Figure 5.2. On a 64-bit machine, `RAX` is the 64-bit extended version of `EAX`.)

Figure 5.2: The lower half of `EAX` is `AX` which in turn is split into `AH` (high) and `AL` (low).
5.2.2 Initializing the PIC

Now that we have functions for talking to devices we can set up the two PICs. We will configure one as master and the other as slave, and we also remap the interrupt numbers from 0–15 to 32–47 because the first 32 numbers are reserved for faults (see Section 5.3).

The PICs can be accessed via the following four ports:

\[
\text{\textit{\{constants\}} 112a} + \equiv \\
/ I/O Addresses of the two programmable interrupt controllers
\]

# define IO_PIC_MASTER_CMD 0x20 // Master (IRQs 0–7), command register
# define IO_PIC_MASTER_DATA 0x21 // Master, control register

# define IO_PIC_SLAVE_CMD 0xA0 // Slave (IRQs 8–15), command register
# define IO_PIC_SLAVE_DATA 0xA1 // Slave, control register

Defines:

- IO_PIC_MASTER_CMD, used in chunks 135a and 146a.
- IO_PIC_MASTER_DATA, used in chunks 135, 139e, and 140a.
- IO_PIC_SLAVE_CMD, used in chunks 135a and 146a.
- IO_PIC_SLAVE_DATA, used in chunks 135, 139e, and 140a.

They need to be initialized by sending them four “Initialization Command Words” (ICW) called ICW1, ICW2, ICW3 and ICW4 in a specific order, using specific ports. Each of the PICs has a command register and a data register. During normal operation we can write to the data register (using the ports IO_PIC_MASTER_DATA\textsubscript{134} and IO_PIC_SLAVE_DATA\textsubscript{134} for PIC1 or PIC2, respectively) to set the interrupt mask: That’s a byte where each bit tells the controller whether it shall respond to a specific interrupt (1 means: mask, i.e., ignore the interrupt; 0 means: forward it to the CPU). We will start with an interrupt mask of 0xFF for each controller (all bits are 1), thus all hardware interrupts will be ignored.

The following code was taken from Bran’s Kernel Development Tutorial [Fri05] (e.g. from the source file irq.c) and modified.

For programming the controller, we can send configuration data to the data port, but we have to initialize the programming by writing to the command port. The complete sequence is as follows:

- First we send ICW1 to both PICs. ICW1 is a byte whose bits have the following meaning [Int88, p. 11]:
  - 0 \(D_0\): ICW4 needed? We set this to 1 since we want to program the controller.
  - 1 \(D_1\): Single (1) / Cascade (0) mode: We set this to 0 since there’s a slave.
  - 2 \(D_2\): Call Address Interval (ignored), the default value is 0.
  - 3 \(D_3\): Level (1) / Edge (0) Triggered Mode: we set this to 0.
  - 4 \(D_4\): Initialization Bit: We set it to 1 because we want to initialize the controller.
  - 5, 6, 7 \(D_5, D_6, D_7\): not used on x86 hardware, set to 0.

This results in the byte \(00010001\) (0x11). The value is the same for both PICs. As mentioned before, ICW1 must be sent to the PICs’ command registers.
5.2 Interrupt Handling on the Intel Architecture

remap the interrupts to 32..47

\[\text{outportb (IO\_PIC\_MASTER\_CMD, 0x11); // ICW1: initialize; begin programming}\]
\[\text{outportb (IO\_PIC\_SLAVE\_CMD, 0x11); // ICW1: dito, for PIC2}\]

Uses IO\_PIC\_MASTER\_CMD 134, IO\_PIC\_SLAVE\_CMD 134, and outportb 133b.

- In the next step we send ICW2 to the PICs’ data registers. The lowest three bits specify the offset for remapping the interrupts. Since the first 32 interrupts must be reserved for processor exception handlers (e.g. “Division by Zero” and “Page Fault” handlers), we map the interrupts 0–15 to the range 32–47 (0x20 – 0x2f).

  Each PIC would normally generate interrupts in the range 0–7, thus the offset is not the same for both PICs: For PIC1 it is 0x20 (32; mapping 0–7 to 32-39), and for PIC2 it is 0x28 (40; mapping 0–7 to 40–47).

\[\text{outportb (IO\_PIC\_MASTER\_DATA, 0x20); // ICW2 for PIC1: offset 0x20}\]
\[\text{(remaps 0x00..0x07 -> 0x20..0x27)}\]
\[\text{outportb (IO\_PIC\_SLAVE\_DATA, 0x28); // ICW2 for PIC2: offset 0x28}\]
\[\text{(remaps 0x08..0x0f -> 0x28..0x2f)}\]

Uses IO\_PIC\_MASTER\_DATA 134, IO\_PIC\_SLAVE\_DATA 134, and outportb 133b.

- The next command word is ICW3. Its functionality depends on whether we send it to the master (PIC1) or the slave (PIC2): The PICs already know that they are master and slave (because we sent that information as part of ICW1) [Int88, p. 12].

  The master expects a command word byte in which each set bit specifies a slave connected to it. We have only one slave and want to make it signal new interrupts on interrupt line 2 of the master. Thus, only the third bit (from the right) must be set: 00000100b = 0x04.

  The slave needs a slave ID. We give it the ID 2 = 0x02.

\[\text{outportb (IO\_PIC\_MASTER\_DATA, 0x04); // ICW3 for PIC1: there’s a slave on IRQ 2}\]
\[\text{0b000000100 = 0x04}\]
\[\text{outportb (IO\_PIC\_SLAVE\_DATA, 0x02); // ICW3 for PIC2: your slave ID is 2}\]

Uses IO\_PIC\_MASTER\_DATA 134, IO\_PIC\_SLAVE\_DATA 134, and outportb 133b.

- To end the sequence, we send ICW4 which is just 0x01 for x86 processors [Int88, p. 12].

\[\text{outportb (IO\_PIC\_MASTER\_DATA, 0x01); // ICW4 for PIC1 and PIC2: 8086 mode}\]
\[\text{outportb (IO\_PIC\_SLAVE\_DATA, 0x01)}\]

Uses IO\_PIC\_MASTER\_DATA 134, IO\_PIC\_SLAVE\_DATA 134, and outportb 133b.

With the remapping in place we can now create entries for the interrupt handler table—we need some new data structures for them.
5.2.3 **Interrupt Descriptor Table**

The PICs are initialized and will do the right thing when an interrupt occurs, but we haven’t told the CPU yet what to do when it receives one. This calls for a new data structure, the *Interrupt Descriptor Table*, which we must define according to the Intel standards and fill with proper values.

While the first Intel-8086/8088-based personal computers used a fixed address in RAM to store the interrupt handler addresses, modern machines let us place the table anywhere in memory. After preparing the table we must use the machine instruction `lidt` (load interrupt descriptor table register) to tell the CPU where to search.

The procedure we need to follow is similar to the one for activating segmentation via a GDT (see pages 90–92):

1. We first store interrupt descriptors (each of which is eight bytes large) in a table consisting of `struct idt_entry` entries,
2. then we create some kind of pointer structure `struct idt_ptr` which contains the length and the start address of the table,
3. and finally we execute `lidt` (compare this to `lgdt` for the GDT).

Figure 5.3 shows the layout of an IDT entry. The *Flags* halfbyte (second line, left in the figure) consists of

- the present flag (bit 3) which must always be set to 1,
- two bits (2 and 1) for the Descriptor Privilege Level (DPL). We will always set this to $11_b = 3$ since we want all interrupts to be available all the time (when we’re in kernel or user mode) and
- a so-called “storage segment” flag (bit 0; which must be set to 0 for an “interrupt gate”, see next entry).

The *Type* halfbyte declares what kind of descriptor this is: we will always set it to $1110_b$, making this descriptor an

- **80386 32-bit interrupt gate** descriptor (which is what we want).

<table>
<thead>
<tr>
<th>Address: 31–16</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>GDT Selector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 15–0</td>
</tr>
</tbody>
</table>

Figure 5.3: An Interrupt Descriptor contains the address of an interrupt handler and some configuration information.
Besides this type, there are alternatives:

- $0101_b$ for an 80386 32-bit task gate,
- $0110_b$ for an 80286 16-bit interrupt gate,
- $0111_b$ for an 80286 16-bit trap gate and
- $1111_b$ for an 80386 32-bit trap gate,

but we will not go into the details about these. Instead of interrupt gates we could also use trap gates, the difference between those being that “for interrupt gates, interrupts are automatically disabled upon entry and reenabled upon IRET which restores the saved EFLAGS” [OSD13]. We will use a trap gate for the system call handler (see Chapter 6.4).

An interrupt descriptor table entry is described by the following datatype definitions:

```c
struct idt_entry
{
    unsigned int addr_low : 16;  // lower 16 bits of address
    unsigned int gdtsel : 16;   // use which GDT entry?
    unsigned int zeroes : 8;    // must be set to 0
    unsigned int type : 4;      // type of descriptor
    unsigned int flags : 4;
    unsigned int addr_high : 16; // higher 16 bits of address
} __attribute__((packed));
```

Defines:

- `idt_entry`, used in chunks 138a and 146d.

The selector must be the number of a code segment descriptor (in the GDT); we will always set this to $0x08$ since our kernel (ring 0) code segment uses that number (see code chunk ⟨install flat gdt 110a⟩ on page 110).

The IDT pointer has the same structure as the GDT pointer: it informs about the length and the location of the IDT:

```c
struct idt_ptr
{
    unsigned int limit : 16;
    unsigned int base : 32;
} __attribute__((packed));
```

Defines:

- `idt_ptr`, used in chunk 138a.

In theory, an interrupt number can be any byte, i.e., a value between 0 and 255. We will use a full IDT with 256 entries even though most of the entries will be null descriptors—if somehow an interrupt is generated which has a null descriptor, the CPU will generate an “unhandled interrupt” exception. We will talk about exceptions right after we’ve finished the interrupt handling code.
Interrupts and Faults

[138a] (global variables 92b) +≡ 
  struct idt_entry idt[256] = { { 0 } };
  struct idt_ptr idtp;

Defines:
  idt, used in chunks 138c and 146d.
  idtp, used in chunks 146d and 147a.
Uses idt_entry 137a and idt_ptr 137b.

The variables idt_{138a} and idtp_{138a} will now be used in a way that is similar to how we used gdt_{92b} (a struct gdt_entry\_91[] array) and gp_{92b} (a struct gdt_ptr\_92a structure) when we wrote the GDT code.

We start with a function

[138b] (function prototypes 45a) +≡ 
  void fill_idt_entry (byte num, unsigned long address,
    word gdtsel, byte flags, byte type);

which writes an entry of the IDT:

[138c] (function implementations 100b) +≡ 
  void fill_idt_entry (byte num, unsigned long address,
    word gdtsel, byte flags, byte type) {
    if (num ≥ 0 && num < 256) {
      idt[num].addr_low = address & 0xFFFF; // address is the handler address
      idt[num].addr_high = (address >> 16) & 0xFFFF;
      idt[num].gdtsel = gdtsel; // GDT sel.: user mode or kernel mode?
      idt[num]. zeroes = 0;
      idt[num].flags = flags;
      idt[num].type = type;
    }
  }

Defines:
  fill_idt_entry, used in chunks 138b, 139b, 148b, and 202e.
Uses idt 138a.

Parts of all of our interrupt handlers will be assembler code (which we store in start_{94}.asm); we’ll explain soon why that has to be. For the moment, let’s declare 16 external function symbols irq0_{144}, irq1_{144}, ..., irq15_{144} whose addresses we’re about to enter into the IDT with fill_idt_entry_{138c}:

[138d] (function prototypes 45a) +≡ 
  extern void irq0(), irq1(), irq2(), irq3(), irq4(), irq5(), irq6(), irq7();
  extern void irq8(), irq9(), irq10(), irq11(), irq12(), irq13(), irq14(), irq15();

We will store the function addresses in an array which simplifies accessing them:
5.2 Interrupt Handling on the Intel Architecture

> ⟨global variables⟩ \[92b\] +≡ \[44a\] (44a) <138a 145b>  [139a]

void (*irqs[16])( ) = {
    irq0, irq1, irq2, irq3, irq4, irq5, irq6, irq7,  // store them in
    irq8, irq9, irq10, irq11, irq12, irq13, irq14, irq15  // an array
};

Defines: irqs, used in chunk 139b.

Uses irq0 144, irq1 144, irq10 144, irq11 144, irq12 144, irq13 144, irq14 144, irq15 144, irq2 144, irq3 144, irq4 144, irq5 144, irq6 144, irq7 144, irq8 144, and irq9 144.

The following code chunk enters their address in the IDT:

> ⟨install the interrupt handlers⟩ \[139b\] ≡ \[45b\] (45b) 323c ▷ [139b]

> ⟨install the interrupt handlers⟩ \[139b\] ≡ \[45b\] (45b) 323c ▷ [139b]

set_irqmask (0xFFFF);  // initialize IRQ mask
enable_interrupt (IRQ_SLAVE);  // IRQ slave

for (int i = 0; i < 16; i++) {
    fill_idt_entry (32 + i,
    (unsigned int)irqs[i],
    0x08,
    0b1110,  // flags: 1 (present), 11 (DPL 3), 0
    0b1110);  // type: 1110 (32 bit interrupt gate)
}

Uses enable_interrupt 140b, fill_idt_entry 138c, IRQ_SLAVE 132, irqs 139a, and set_irqmask 139e.

This code chunk sets the IRQ mask to 0xFFFF = 1111111111111111b, via

> ⟨function prototypes⟩ \[45a\] +≡ \[44a\] (44a) <138d 139d>  [139c]

static void set_irqmask (word mask);

which disables all interrupts, and then it enables the interrupt for the slave PIC with

> ⟨function prototypes⟩ \[45a\] +≡ \[44a\] (44a) <139e 140f>  [139d]

static void enable_interrupt (int number);

—both functions have not been mentioned so far. The IRQ mask is a 16-bit value in which each bit says whether some interrupt is enabled (value 0) or not (value 1). We must talk to both PICs to set the mask, the master PIC gets the lower eight bits (for the interrupts 0–7), the slave PIC gets the upper eight bits (for the interrupts 8–15):

> ⟨function implementations⟩ \[45b\] +≡ \[44a\] (44a) <138c 140a>  [139e]

static void set_irqmask (word mask) {
    outportb (IO_PIC_MASTER_DATA, (char)(mask % 256));
    outportb (IO_PIC_SLAVE_DATA, (char)(mask >> 8));
}

Defines: set_irqmask, used in chunks 139 and 140b.

Uses IO_PIC_MASTER_DATA 134, IO_PIC_SLAVE_DATA 134, and outportb 133b.

We can also read the mask from the two PICs with a similar function we call

> ⟨function prototypes⟩ \[45a\] +≡ \[44a\] (44a) <139d 146b>  [139f]

word get_irqmask ();
in which we read the two data registers instead of writing them:

```c
(word get_irqmask () {
    return inportb (IO_PIC_MASTER_DATA)
    + (inportb (IO_PIC_SLAVE_DATA) << 8);
}
```

Defines:
- `get_irqmask`, used in chunks 139f and 140b.
- Uses `inportb`, `IO_PIC_MASTER_DATA`, and `IO_PIC_SLAVE_DATA`.

In the following chapters we will often enable a specific interrupt for some device after we’ve prepared its usage, e.g. for the floppy controller. For that purpose, we will always use `enable_interrupt()` like we did above. It simply reads the current IRQ mask, clears a bit, and writes the new value back:

```c
(static void enable_interrupt (int number) {
    set_irqmask (get_irqmask () & ~(1 << number) // 16 one-bits, but bit "number" cleared
   );
}
```

Defines:
- `enable_interrupt`, used in chunks 139, 323b, 339a, 344c, 520a, 534b, and 552c.
- Uses `get_irqmask` and `set_irqmask`.

### 5.2.4 Writing the Interrupt Handler

Everything is prepared for interrupt handlers — now we need to define them, i.e., implement the `irq0()`, ..., `irq15()` functions. This step requires some assembler code and some C code.

We have installed handlers for all 16 interrupts, but what do they do? We will define part of their code in the assembler file, but we start with a description of what we expect to happen in general.

When an interrupt occurs, the CPU suspends the currently running code, saves some information on the stack, and then jumps to the address that it finds in the IDT. (It also uses a different stack and switches to kernel mode if it was in user mode when the interrupt occurred.) Then the interrupt handler runs, and once it has finished its job, it returns with the `iret` instruction. `iret` is different from the regular `ret` instruction which normal functions use for returning to the calling function: it is the special “return from interrupt” instruction which restores the original state (user or kernel mode, stack, EFLAGS register) so that the regular code can continue as if the interrupt had never happened.

Switching to the interrupt handler can mean a change of the privilege level that the CPU executes in: So far we’ve only let Linux work in ring 0 (kernel mode), but later when we introduce processes it can happen that an interrupt occurs while the CPU runs in ring 3 (user mode). If that is the case, the privilege level changes (from 3 to 0). When such a
transition occurs, the information (return address etc.) is not written to the process’ user mode stack, but on the process’ kernel stack which is located elsewhere and normally used during the execution of system calls—we’ll describe that in more detail later. For now, the relevant piece of information is that different information gets stored on the “target stack”: In case of a privilege change the CPU first writes the contents of the SS and ESP registers on the (new) stack—this does not happen if the CPU was already operating in ring 0. Next, EFLAGS, CS and EIP are written to the stack: that is all we need for returning to the interrupted code. Figure 5.4 shows the different stack contents when the interrupt handler starts executing [Int86, p. 159].

We cannot directly use a C function as an interrupt handler because once it would finish its work, it would do a regular RET which does not do what we want. (Of course we could use inline assembler code inside the C function to make it work anyway, but it makes more sense to directly implement parts of the handlers in assembler.)

### 5.2.4.1 The Context Data Structure

We want to be able to define handler functions in C which get called from the assembler code. Those functions will all have the following prototype:

```c
void handler_function (context_t *r);
```

where `context_t` is a central data structure that can hold all the registers we use on the Intel machine. It will also be used in fault handlers, system call handlers and several other functions which need information about the current state.

We define the `context_t` structure so that it matches the way in which we set up the stack in the assembler part of the handler:
5.2.4.2 Assembler Part of the Handler

In order to have a handler function see useful values in the structure that \( r \) points to, we need to push the register contents in the reverse order onto the stack:

```asm
pusha
push ds
push es
push fs
push gs
push esp
```

; pointer to the context_t

The first instruction `pusha` (push all general registers) pushes a lot of registers onto the stack: \( EAX, ECX, EDX, EBX \), the old value of \( ESP \) (before the pusha execution began), \( EBP, ESI \), and \( EDI \)—in that order. We add the segment registers \( DS, ES, FS \) and \( GS \), and you can see that we've successfully handled the first two lines of the `context_t` type definition. When the interrupt occurred, the registers \( EFLAGS, CS \) and \( EIP \) (and possibly also \( SS \) and the user mode's \( ESP \)) were also pushed on the stack which gives us the values in the fourth line of the `context_t` definition.

What's missing are the values on the third line: We want to tell the handler which interrupt occurred so that we can use the same interrupt handler for several interrupts—for example, if we supported both IDE controllers (with interrupts 14 and 15) we could use that trick to run the same IDE handler when either of those interrupts occurred; thus, between the automatically happening push operations and the ones we perform in \( \langle \text{push registers onto the stack} \rangle \) we also push the interrupt number and another value \( err\_code \) which can hold an error code. Interrupts don’t have an error code, but we will recycle the same code later when we deal with faults, and some of those do provide an error code.

The final `push esp` statement in \( \langle \text{push registers onto the stack} \rangle \) is necessary because we cannot just place the structure contents on the stack: the handler function expects a pointer \( \langle \text{context_t} \rangle *r \), and \( ESP \) contains just that pointer: the start address of the structure. Figure 5.5 shows the layout of the stack after the assembler part has finished the preparations.

Later, when the handler’s task is completed, we will need to pop the registers from the stack—in the reverse order:
Figure 5.5: Stack after interrupt handler initialization by the assembler part.

\[
\langle \text{pop registers from the stack} \rangle \equiv \quad (143b \ 144 \ 150b \ 202c) \quad [143a]
\]

\[
\langle \text{irq15 example} \rangle \equiv \quad [143b]
\]

Now here's an example of how we could implement the interrupt handler for IRQ 15:

\[
\langle \text{irq15 example} \rangle \equiv \quad [143b]
\]

Uses \texttt{irq_handler} 146a.

This contains all we need:
1. The two push commands add the error code and the interrupt number (which is 15 in this example).

2. With *(push registers onto the stack)* we complete the context_t data structure and also push a pointer to it.

3. Now the stack is prepared properly to call the C function irq_handler.

4. After returning, we first have to undo the push operations with *(pop registers from the stack)*.

5. Then we modify the stack address: we add 8, thus undoing the two push operations for the error code and the interrupt number.

6. Finally we return from the handler with iret.

We need almost the same code 16 times (for IRQs 0 to 15)—the only difference between the 16 versions is the interrupt number that we push in the second instruction. We simplify our code by having our individual handlers just push the two values (0 and the interrupt number) and then jump to an address which provides the common commands. The 0 value is a placeholder for an error code which cannot occur in interrupt handlers, but (as mentioned before) we will also implement fault handlers which shall use the same stack layout, and some of them will write a fault-specific error code into that location.

![start.asm](start.asm)

```assembly
%macro irq_macro 1
  push byte 0       ; error code (none)
  push byte %1      ; interrupt number
  jmp irq_common_stub ; rest is identical for all handlers
%endmacro

irq0: irq_macro 32
irq1: irq_macro 33
irq2: irq_macro 34
irq3: irq_macro 35
irq4: irq_macro 36
irq5: irq_macro 37
irq6: irq_macro 38
irq7: irq_macro 39
irq8: irq_macro 40
irq9: irq_macro 41
irq10: irq_macro 42
irq11: irq_macro 43
irq12: irq_macro 44
irq13: irq_macro 45
irq14: irq_macro 46
irq15: irq_macro 47
```
5.2 Interrupt Handling on the Intel Architecture

extern irq_handler  ; defined in the C source file

irq_common_stub:     ; this is the identical part
    ⟨push registers onto the stack 142b⟩
call irq_handler     ; call C function
    ⟨pop registers from the stack 143a⟩
add esp, 8
iret

Defines:
    irq0, used in chunk 139a.
    irq1, used in chunk 139a.
    irq10, used in chunk 139a.
    irq11, used in chunk 139a.
    irq12, used in chunk 139a.
    irq13, used in chunk 139a.
    irq14, used in chunk 139a.
    irq15, used in chunk 139a.
    irq2, used in chunk 139a.
    irq3, used in chunk 139a.
    irq4, used in chunk 139a.
    irq5, used in chunk 139a.
    irq6, used in chunk 139a.
    irq7, used in chunk 139a.
    irq8, used in chunk 139a.
    irq9, used in chunks 138d and 139a.

Uses irq_handler 146a.

Our interrupt handling code is a slightly improved version of the code which Bran’s Kernel Tutorial [Fri05] uses; the original code contains some extra instructions that we don’t need for the Ulix kernel.

5.2.4.3 C Part of the Handler

Finally, we show what happens when the assembler code calls the external handler function irq_handler(int) that we implement in the C file.

The first thing our handler needs to do is acknowledge the interrupt. For that purpose it sends the command

⟨constants 112a⟩:+≡

#define END_OF_INTERRUPT 0x20

Defines:
    END_OF_INTERRUPT, used in chunk 146a.

to all PICs which are involved: In case of an interrupt number between 0 and 7 that is only the primary PIC; in case the number is 8 or higher, both controllers need to be informed. Omitting this step would stop the controller from raising further interrupts which would basically disable interrupts completely.

Next we check whether a specific handler for the current interrupt has been installed in the

⟨global variables 92b⟩:+≡

void *interrupt_handlers[16] = { 0 };

Defines:
    interrupt_handlers, used in chunk 146.
array of interrupt handlers.

```c
void irq_handler (context_t *r) {
    int number = r->int_no - 32;  // interrupt number
    void (*handler)(context_t *r);  // type of handler functions

    if (number ≥ 8) {
        outportb (IO_PIC_SLAVE_CMD, END_OF_INTERRUPT);  // notify slave PIC
    }
    outportb (IO_PIC_MASTER_CMD, END_OF_INTERRUPT);  // notify master PIC (always)

    handler = interrupt_handlers[number];
    if (handler != NULL) {
        handler (r);
    }
}
```

Defines:
- `irq_handler`, used in chunks 143b and 144.
- Uses `context_t`, `END_OF_INTERRUPT`, `interrupt_handlers`, `IO_PIC_MASTER_CMD`, `IO_PIC_SLAVE_CMD`, `NULL`, and `outportb`.

As a last step we provide a function

```c
void installInterrupt_handler (int irq, void (*handler)(context_t *r));
```

which lets us enter (pointers to) handler functions in this array; it is pretty simple:

```c
void install_interrupt_handler (int irq, void (*handler)(context_t *r)) {
    if (irq ≥ 0 && irq < 16)
        interrupt_handlers[irq] = handler;
}
```

Defines:
- `install_interrupt_handler`, used in chunks 146b, 323b, 339a, 520a, 534b, and 552c.
- Uses `context_t` and `install_interrupt_handler`.

Early in the ⟨initialize system⟩ step of the kernel’s main() function we need to load the Interrupt Descriptor Table Register (IDTR) so that the CPU can find the table:

```c
idtp.limit = (sizeof (struct idt_entry) * 256) - 1;  // must do -1
idtp.base = (int) &idt;
idt_load ();
```

Uses `idt`, `idt_entry`, `idt_load`, and `idtp`.

It uses the assembler function

```c
extern void idt_load ();
```
which is related to gdt_flush, just writing the address of idtp to the IDTR register via the lidt instruction instead of writing the address of gp to GDTR via lgdt:

\[
\text{lidt}
\]

\[
\text{lidt~}[\text{idtp}]
\]

\[
\text{ret}
\]

Defines:

idt_load, used in chunk 146.

Uses idtp 138a.

In the following chapters we will often use this function in commands similar to

\[
\text{install\_interrupt\_handler~(IRQ\_SOMEDEV, somedev\_handler);} \]

For comparison, once more gdt_flush and idt_load:

\[
\text{extern~gp~;~defined~in~the~C~file}
\]

\[
\text{global~gdt\_flush}
\]

\[
\text{gdt\_flush:~lgdt~}[\text{gp}]
\]

\[
\text{mov~ax,~0x10}
\]

\[
\text{mov~ds,~ax}
\]

\[
\text{mov~es,~ax}
\]

\[
\text{mov~fs,~ax}
\]

\[
\text{mov~gs,~ax}
\]

\[
\text{mov~ss,~ax}
\]

\[
\text{jmp~0x08:flush2}
\]

\[
\text{flush2:~ret}
\]

(The gdt_flush function does more than idt_load since it also updates all segment registers.)

### 5.3 Faults

As we’ve mentioned in the introduction to this chapter, handling a fault is very similar to handling an interrupt. Since you’ve just seen the interrupt code, you will recognize many concepts at once while we present the fault handling code.

Like we defined the interrupt handlers irq0_144() to irq15_144() in the assembler file start.asm, we do the same with 32 fault handler functions fault0_149b() to fault31_149b().

\[
\text{extern~void}
\]

\[
\text{fault0()},~\text{fault1()},~\text{fault2()},~\text{fault3()},~\text{fault4()},~\text{fault5()},~\text{fault6()},~\text{fault7()},~\text{fault8()},~\text{fault9()},~\text{fault10()},~\text{fault11()},~\text{fault12()},~\text{fault13()},~\text{fault14()},~\text{fault15()},~\text{fault16()},~\text{fault17()},~\text{fault18()},~\text{fault19()},~\text{fault20()},~\text{fault21()},~\text{fault22()},~\text{fault23()},~\text{fault24()},~\text{fault25()},~\text{fault26()},~\text{fault27()},~\text{fault28()},~\text{fault29()},~\text{fault30()},~\text{fault31}();
\]
Interrupts and Faults

and we enter these in the IDT just like we did with the irq*() functions.

\[
\langle \text{global variables 92b} \rangle + \equiv \\
\text{void (*faults[32])()} = \\
\quad \text{fault0, fault1, fault2, fault3, fault4, fault5, fault6, fault7, fault8, fault9, fault10, fault11, fault12, fault13, fault14, fault15, fault16, fault17, fault18, fault19, fault20, fault21, fault22, fault23, fault24, fault25, fault26, fault27, fault28, fault29, fault30, fault31}
\]

Defines:
- faults, used in chunks 148b and 607c.

We install those handlers in the same way that we registered the interrupt handlers earlier (see page 139):

\[
\langle \text{install the fault handlers 148b} \rangle + \equiv \\
\text{for (int i = 0; i < 32; i++)}
\]

\[
\quad \text{fill_idt_entry (i,}
\]

\[
\quad \quad \quad \text{(unsigned int)faults[i]},
\quad \quad \quad \text{0x08,}
\quad \quad \quad \text{0b1110, // flags: 1 (present), 11 (DPL 3), 0}
\quad \quad \quad \text{0b1110); // type: 1110 (32 bit interrupt gate)}
\]

Uses faults 148a and fill_idt_entry 138c.

In the assembler file we use the same trick for the fault*() functions that you’ve just seen for irq*():

\[
\langle \text{start.asm 87} \rangle + \equiv \\
\text{global fault0, fault1, fault2, fault3, fault4, fault5, fault6, fault7, fault8, fault9, fault10, fault11, fault12, fault13, fault14, fault15, fault16, fault17, fault18, fault19, fault20, fault21, fault22, fault23, fault24, fault25, fault26, fault27, fault28, fault29, fault30, fault31}
\]

The handlers all look similar: We push one or two bytes on the stack and then jump to fault_common_stub150b. The choice of one or two arguments depends on the kind of interrupt that occurred: for some faults the CPU pushes a one-byte error code on the stack, and for some others it does not. In order to have the same stack setup (regardless of the fault) we push an extra null byte in those cases where no error code is pushed.

The code always looks like one of the following two cases:
Since we do not want to type this repeatedly, we use nasm’s macro feature which lets us write simple macros for both cases. fault_macro_0 handles the cases where we need to push an extra null byte (as in fault5 above), and fault_macro_no0 handles the other cases (as in fault8 above):

\[
\begin{align*}
\text{start.asm} &+\equiv \\
\%macro & \text{fault \_ macro \_ 0} \text{ 1} \\
& \quad \text{push byte 0 ; error code} \\
& \quad \text{push byte %1} \\
& \quad \text{jmp \ fault \_ common \_ stub} \\
\%endmacro \\
\%macro & \text{fault \_ macro \_ no0} \text{ 1} \\
& \quad \text{; don't push error code} \\
& \quad \text{push byte %1} \\
& \quad \text{jmp \ fault \_ common \_ stub} \\
\%endmacro
\end{align*}
\]

Defines:
- fault\_macro\_0, used in chunk 149b.
- fault\_macro\_no0, used in chunk 149b.

Uses fault\_common\_stub.

With these macros the rest is straight-forward:

\[
\begin{align*}
\text{start.asm} &+\equiv \\
fault0: & \text{fault\_macro\_0 0 ; Divide by Zero} \\
fault1: & \text{fault\_macro\_0 1 ; Debug} \\
fault2: & \text{fault\_macro\_0 2 ; Non Maskable Interrupt} \\
fault3: & \text{fault\_macro\_0 3 ; INT 3} \\
fault4: & \text{fault\_macro\_0 4 ; INTO} \\
fault5: & \text{fault\_macro\_0 5 ; Out of Bounds} \\
fault6: & \text{fault\_macro\_0 6 ; Invalid Opcode} \\
fault7: & \text{fault\_macro\_0 7 ; Coprocessor not available} \\
fault8: & \text{fault\_macro\_no0 8 ; Double Fault} \\
fault9: & \text{fault\_macro\_0 9 ; Coprocessor Segment Overrun} \\
fault10: & \text{fault\_macro\_no0 10 ; Bad TSS} \\
fault11: & \text{fault\_macro\_no0 11 ; Segment Not Present} \\
fault12: & \text{fault\_macro\_no0 12 ; Stack Fault} \\
fault13: & \text{fault\_macro\_no0 13 ; General Protection Fault} \\
fault14: & \text{fault\_macro\_no0 14 ; Page Fault} \\
fault15: & \text{fault\_macro\_0 15 ; (reserved)} \\
fault16: & \text{fault\_macro\_0 16 ; Floating Point} \\
fault17: & \text{fault\_macro\_0 17 ; Alignment Check} \\
fault18: & \text{fault\_macro\_0 18 ; Machine Check} \\
fault19: & \text{fault\_macro\_0 19 ; (reserved)} \\
fault20: & \text{fault\_macro\_0 20 ; (reserved)} \\
fault21: & \text{fault\_macro\_0 21 ; (reserved)} \\
fault22: & \text{fault\_macro\_0 22 ; (reserved)} \\
fault23: & \text{fault\_macro\_0 23 ; (reserved)} \\
fault24: & \text{fault\_macro\_0 24 ; (reserved)} \\
fault25: & \text{fault\_macro\_0 25 ; (reserved)} \\
fault26: & \text{fault\_macro\_0 26 ; (reserved)}
\end{align*}
\]
defines:
fault0, used in chunks 147b and 148a.
fault1, used in chunks 147b and 148a.
fault10, used in chunks 147b and 148a.
fault11, used in chunks 147b and 148a.
fault12, used in chunks 147b and 148a.
fault13, used in chunks 147b and 148a.
fault14, used in chunks 147b and 148a.
fault15, used in chunks 147b and 148a.
fault16, used in chunks 147b and 148a.
fault17, used in chunks 147b and 148a.
fault18, used in chunks 147b and 148a.
fault19, used in chunks 147b and 148a.
fault2, used in chunks 147b and 148a.
fault20, used in chunks 147b and 148a.
fault21, used in chunks 147b and 148a.
fault22, used in chunks 147b and 148a.
fault23, used in chunks 147b and 148a.
fault24, used in chunks 147b and 148a.
fault25, used in chunks 147b and 148a.
fault26, used in chunks 147b and 148a.
fault27, used in chunks 147b and 148a.
fault28, used in chunks 147b and 148a.
fault29, used in chunks 147b and 148a.
fault3, used in chunks 147b and 148a.
fault30, used in chunks 147b and 148a.
fault31, used in chunks 147b and 148a.
fault4, used in chunks 147b and 148a.
fault5, used in chunks 147b and 148a.
fault6, used in chunks 147b and 148a.
fault7, used in chunks 147b and 148a.
fault8, used in chunks 147b and 148a.
fault9, used in chunks 147b and 148a.

uses fault_macro_0 149a and fault_macro_no0 149a.

fault_common_stub is—almost—a rewrite of irq_common_stub, the only difference is that we call a different C function fault_handler in the middle.

[150a] \langle start.asm 87\rangle\+≡

extern fault_handler

uses fault_handler 151c.

The stub saves the processor state, calls the handler function and restores the stack frame:

[150b] \langle start.asm 87\rangle\+≡
fault_common_stub:

\langle push registers onto the stack\rangle
    call fault_handler ; call C function
\langle pop registers from the stack\rangle
    add esp, 8 ; for errcode, irq no.
    iret

defines:
fault_common_stub, used in chunk 149a.

uses fault_handler 151c.
Initially our fault handlers will just output a message stating the cause of the fault and then halt the system; later we will provide fault handlers for some types of faults which try to solve the problem and let the operation go on. Here are the error messages:

```c
char *exception_messages[] = {
    "Division By Zero", "Debug",       // 0, 1
    "Non Maskable Interrupt", "Breakpoint", // 2, 3
    "Into Detected Overflow", "Out of Bounds", // 4, 5
    "Invalid Opcode", "No Coprocessor",  // 6, 7
    "Double Fault", "Coprocessor Segment Overrun", // 8, 9
    "Bad TSS", "Segment Not Present",  // 10, 11
    "Stack Fault", "General Protection Fault", // 12, 13
    "Page Fault", "Unknown Interrupt",   // 14, 15
    "Coprocessor Fault", "Alignment Check", // 16, 17
    "Machine Check",                      // 18
};
```

Defines:

- `exception_messages`, used in chunk 152a.

We get the correct message by accessing the proper entry of the array, e.g., for a page fault (with fault number 14) it is stored in `exception_messages[14].`

### Our C fault handler

```c
void fault_handler (context_t *r) {
    if (r->int_no == 14) { // fault 14 is a page fault
        page_fault_handler (r); return;
    }

    memaddress fault_address = (memaddress)(r->eip);

    if (r->int_no < 32) {
        return;
    } // fault handler: display status information

    if (fault_address < 0xc0000000) { // user mode
        return;
    } // fault handler: terminate process
}
```
Interrupts and Faults

```c
interrupts (276b)

// error inside the kernel
printf ("\n");
asm ("jmp kernel_shell");
```

Defines:
- `fault_handler`, used in chunks 150 and 151b.
- `context_t` 142a
- `kernel_shell` 610a
- `memaddress` 46c
- `page_fault_handler` 289a
- `printf` 601a

For displaying the status information we look at the register contents which are provided by `r`. Especially interesting are the task number, the address space number, the address of the faulting instruction, the `EFLAGS` register and the error code which the CPU has provided upon entry into the fault handler.

```c
(fault handler: display status information 152a)
printf ("'s' Exception at 0x%08x (task=%d, as=%d).\n", r->int_no, r->eip, current_task, current_as);
printf ("eflags: 0x%08x errcode: 0x%08x\n", r->eflags, r->err_code);
printf ("eax: %08x ebx: %08x ecx: %08x edx: %08x \n", r->eax, r->ebx, r->ecx, r->edx);
printf ("eip: %08x esp: %08x int: %8d err: %8d \n", r->eip, r->esp, r->int_no, r->err_code);
printf ("cs: 0x%02x ds: 0x%02x ss: 0x%02x fs: 0x%02x ss: 0x%02x \n", r->cs, r->ds, r->es, r->fs, r->ss);
printf ("User mode stack: 0x%08x-0x%08x\n", address_spaces[current_as].stacksize, TOP_OF_USER MODE_STACK);
```

If a process was running, the fault handler terminates it:

```c
(fault handler: terminate process 152b)
thread_table[current_task].state = TSTATE_ZOMBIE;
remove_from_ready_queue (current_task);
r->ebx = -1; // exit code for this process
syscall_exit (r);
```

Since we have not talked about processes yet, you need not worry about the reference to the thread table via `thread_table[current_task]` or `remove_from_ready_queue (current_task)`.

A page fault need not be a problem: it often occurs because the code attempted to access an invalid address (which is bad), but yet more often the address will be valid, but the page won’t be in the physical RAM. That situation can be helped. In Chapter 9 we will implement the `page fault handler`. It requires a working hard disk since we will `page out` pages to the disk and later `page them in` again.
5.4 Exercises

16. Keyboard driver: Polling

In the folder tutorial/04/ you find a version of the Ulix kernel which contains the new interrupt and fault handling code. It is a literate program (ulix.nw). You will now develop a simple keyboard driver, extending the provided code, and you should try to retain the literate programming style, i.e., integrate code and documentation in the file.

cd into the folder and open the file ulix.nw. At the end you will find a section “keyboard driver” where you can place your new code; at least most of it. The rest of the file corresponds to the literate program from the last exercise, but some new mechanisms have been added.

As a first step you can test querying the keyboard controller via polling:

a) The keyboard controller can be accessed via two ports (0x60 and 0x64) which you can read from via inportb_133b(). Append the port numbers to the (constants 112a) code chunk:

```
#define KBD_DATA_PORT 0x60
#define KBD_STATUS_PORT 0x64
```

The data port delivers information about pressed and released keys, and the status port lets you check whether a key was pressed (or released) at all.

b) Try to continuously read the data port in a loop and print the results (as numbers). You can query with the following code:

```c
byte scancode;
scancode = inportb (KBD_DATA_PORT);
```

(We have prepared an empty code chunk (kernel main: user-defined tests) at the end of the Noweb file which will be called after initialization.) Print the scan codes with printf_601a(). This will quickly fill the screen (even if you add no newlines to your printf_601a call), so you should clear the screen when you reach the bottom line:

```
if (posy == 25) clrscr ()
```

You will notice that this approach writes a continuous (and quick) stream of data onto the screen. While the system is running, press a few keys; that will modify the output. (You may have to keep the keys pressed to recognize the changes.) The values are keyboard scan codes, each of them represents an action of pressing or releasing a key. You will see the same value again and again until a new press/release event occurs.

c) Improve the code by also checking the status register (via the status port). That works in the same way that you’ve accessed the data port, but uses the port number KBD_STATUS_PORT. If the lowest bit of the return value is set (which you
can check with if ((status & 1) == 1)), then there is a fresh scan code, and only
then should you query the data port. The modified code will only generate an
output when you press or release a key.

d) The scan codes for pressing and releasing a key only differ in the eighth bit (it it
set for release events), so for example scan codes for the “A” key are 30 and 158
(= 30 + 128), for its left neighbor key “S” they are 31 and 159 (= 31 + 128). Create
a mapping table which stores the upper case letters which correspond to a few
scan codes. You need not look up an ASCII table but can simply enter characters,
such as 'A' or 'B', in the table. Initialize the table with zero values:

```c
char scancode_table[128] = { 0 };
```

Then you can start with adding entries for the scan codes you already know, e.g.
30 and 31 for the “A” and “S” keys. (We ignore release codes in this table.)

```c
scancode_table[30] = 'A';
scancode_table[31] = 'S';
```

Identify the return key’s scan code. The corresponding character is '\n'.

Modify your existing code so that it does not only print the scan code but also
the character (if it is known, i.e., the corresponding table entry is not 0). Test
your program. (The printf format code for characters is %c. When you print
release scan codes you get negative numbers—you can get rid of them by casting
the scan code to the int type.)

This leaves you with a simple, polling keyboard driver.

17. Keyboard driver: Interrupts

In this exercise you switch to an interrupt-based keyboard driver.

a) Add the following lines to an appropriate place in the system initialization, e.g.
in the ⟨kernel main: initialize system ⟩ chunk:

```c
install_interrupt_handler (IRQ_KBD, keyboard_handler);
enable_interrupt (IRQ_KBD);
asm ("sti"); // enable interrupts
```

IRQ_KBD is already #defined as 1: It is the interrupt number used by the key-
board controller. Now you have to implement the keyboard handler. It has the
signature

```c
void keyboard_handler (struct regs *r);
```

(struct regs is the same as context_t in the rest of the book.) It will automatic-
ically be called whenever you press or release a key.

b) Make sure that your handler gets called by letting it print a single character (e.g.
'*') and leaving the handler with return;
c) In the next step you evaluate and print the values that you can read from the keyboard controller. It is not necessary to query the controller’s data port (as you did in the above exercise), because the handler is only called when a new event has occurred. After the output (like above, using the scan code table) you can leave the handler with \texttt{return}. Note that again, due to the simplified \texttt{printf} implementation, you will need to insert \texttt{clrscr} calls when you reach the screen’s bottom.

The advantage of using interrupts is that the main program (in \texttt{main\_44b}) need not concern itself with the keyboard.

d) Next, implement a function

\begin{verbatim}
void kreadline (char *s, int len);
\end{verbatim}

that you call from \texttt{main\_44b}, e.g. with

\begin{verbatim}
char input[41]; // 40 characters plus \0 terminator
kreadline ((char*)input, 40);
\end{verbatim}

The goal is that \texttt{kreadline()} fills the provided string (a char pointer) with the characters you enter (as far as they are known in the scan code table) until you either complete the input by pressing [Enter] or until the maximum number of characters (\texttt{len}) is reached. Only then shall the function return. The main program can then print the read string and start over, using an infinite loop.

The important aspect of this exercise is that the \texttt{kreadline} needs to cooperate with the interrupt handler. You will need two new global variables for an input buffer and for the next character position in the buffer:

\begin{verbatim}
char buffer[256]; // buffer for input
short int pos = 0; // current position in the buffer
\end{verbatim}

The interrupt handler should work as follows:

- If the scan code is larger than 127 (release event), the handler returns immediately (it simply ignores release events).
- When an unknown scan code shows up, the handler also returns at once.
- Otherwise it will print the character and also write it into the buffer.
- Then it increases \texttt{pos} and returns.

\texttt{kreadline()} performs an infinite loop and checks whether (\texttt{pos>0} \&\& \texttt{buffer[pos-1] =='\n')} is true—if so, then the function copies the entered string (from position 0 to \texttt{pos-2}) to \texttt{s}, sets \texttt{pos=0} and returns. Note that the string must be '\\0'-terminated so that \texttt{printf()} can later use it. You can simply replace the '\n' character with '\0' if the input is terminated by pressing [Enter]; if you reach the maximum allowed number of characters, you write '\0' into the last byte of the string.

For copying the string you can use \texttt{strncpy()} . That function works like the corresponding Linux function (see \texttt{man strncpy}), i.e., it expects target, source and
maximum length of the target string as arguments.
Your scan code table will need an entry for the [Enter] key in order to make this work.

18. Backspace Support

Modify your code for the keyboard handler and the kreadline function so that it treats the backspace key appropriately: With that key you shall be able to delete the last character that was entered. It shall both be removed from the screen and from the string which kreadline returns.

19. Faults

The current version of the mini kernel contains fault handlers. Verify that they work correctly by generating some typical faults:

- **Division by zero**: Have your main program perform a division by zero, for example via `int z = 1 / 0;`—you can ignore any compiler warnings that this code will cause.
- **Try to access non-available memory**, e.g., with
  ```
  char *address = (char*)0xE0000000; char tmp = *address;
  ```
- **Set the segment register DS to an invalid segment number**, e.g.:
  ```
  asm("mov $32, %ax; mov %ax, %ds");
  ```
- **You can explicitly cause each fault** (for fault numbers between 0 and 31) by using the assembler instruction int $number. For example, in order to generate an “Out of Bounds Fault” (number 5), you can use the line `asm("int $5");`.

The reward should be a Division by Zero Fault, a Page Fault and a General Protection Fault (and an Out of Bounds Fault in the last step).
We have now written most of the code that we need to introduce the most important concept: the process. In this chapter we take a first look at the data structures and kernel functions which will let us create and schedule processes.

- In Chapter 6.1 we present the desired memory layout of a Ulix process and describe our implementation of address spaces.
- Chapter 6.2 introduces the central data structure for processes and threads, the process control block (which we will refer to as a thread control block, TCB), as well as queues for handling ready and blocked threads.
- Chapter 6.3 shows what we need to do in order to start the very first process; all further processes will be created via the `fork` mechanism.
- Since forking will require the existence of a system call interface, it is time to introduce it: we present our implementation in Chapter 6.4.
- With system calls available we can explain the implementation of the fork mechanism (and the `fork` system call) in Chapter 6.5.
- While it is an important step to bring new processes into existence, we also need to handle their termination. In Chapter 6.6 we show how to `exit` from a process.

The remaining sections of this chapter are less interesting but still required: We provide a method for requesting process information in Chapter 6.7 (which will let us write a `ps` program), describe the ELF binary format and an ELF program loader in Chapter 6.8 (so that a process can start a different application via `exec`) and finally discuss an idle process that will be activated when there is no other process that could do something useful (Chapter 8.3.3).
Everything you will see in this chapter deals with single-threaded processes. In Chapter 7 we will extend our execution model so that it also supports multiple threads inside one process. You might want to remember this whenever you wonder why we keep talking about thread control blocks (instead of process control blocks) throughout this chapter.

There’s also a need to discuss how the system can switch between concurrent processes and threads: once we have more than one active task, a scheduler must take care of this. We delay this until Chapter 8.

### 6.1 Address Spaces for Processes

We will store information about memory usage in a data structure that we call *address space descriptor*. The idea is that every process uses its own address space while several threads (of the same process) share a common address space.

Address space descriptors are stored in one large *address space table*. This table must be finite, i.e., there must exist a maximum number of address spaces for the system. This must correspond to the maximum number of threads $\text{MAX\_THREADS}_{176a}$ that we’ll soon define: While threads may share an address space, it is impossible for one thread to use more than one address space. Thus $\text{MAX\_ADDR\_SPACES}_{158a}$ has to be $\leq \text{MAX\_THREADS}_{176a}$—we give both constants the same value:

$$
\text{MAX\_ADDR\_SPACES} = 1024
$$

As we will see later, every thread may have its own virtual address space and needs to own a reference to an address space descriptor. Even the kernel will have to do that. Since there can be so many address spaces, we need a shorthand to identify virtual address spaces. We introduce the type *addr_space_id* to do this. It is declared as *unsigned int*. Basically, an *addr_space_id* can be thought of as an index into the address space table. So rather than storing a complete address space descriptor per thread, we will rather store an address space identifier.

We already note that the thread control block (which will be the central data structure for processes and threads) needs an *addr_space_id* element. That data structure is called TCB$_{175}$, and we will define it later in this chapter, but you will often see the code chunk *⟨more TCB entries 158c⟩* that lets us append entries to this structure whenever the need occurs:
6.1 Address Spaces for Processes

6.1.1 Memory Layout of a Process

Every process needs three (virtual) memory areas.

- **Code and Data**: We will later compile user mode binaries which expect to be loaded to the virtual addresses 0x0 and above. This area is used for the code (the machine instructions in the binary) as well as variables defined statically in the program. The heap will exist just behind this memory area, processes can later dynamically expand this memory area using the `sbrk` function.

\[
\begin{align*}
&\text{\textit{constants}} \ 112a \ + \equiv \ (44a) \ \langle 158a \ 159b \rangle \ [159a] \\
&\text{\texttt{#define}} \ \text{BINARY_LOAD_ADDRESS} \ 0x0 \\
&\text{Defines:} \\
&\quad \text{BINARY_LOAD_ADDRESS, used in chunks 163c, 190c, and 192d.}
\end{align*}
\]

- **User Mode Stack**: Every process needs its own stack: That is where the CPU will store return addresses and arguments whenever the process makes a function call. We’ll use the virtual addresses below 0xb0000000 which will leave a lot of space between the code and data and the stack: We want the stack to grow automatically, so we’ll start with just one single page of memory for the stack and increase it as needed: When you think of recursive functions where the end of the recursion depends on some calculation inside the program, it is clear that we cannot have a maximum size for the stack. Expanding the stack is a task for the page fault handler which we’ve already mentioned. You will see its implementation on page 287 ff.

\[
\begin{align*}
&\text{\textit{constants}} \ 112a \ + \equiv \ (44a) \ \langle 159a \ 159c \rangle \ [159b] \\
&\text{\texttt{#define}} \ \text{TOP_OF_USER_MODE_STACK} \ 0xb0000000 \\
&\text{Defines:} \\
&\quad \text{TOP_OF_USER_MODE_STACK, used in chunks 152a, 165a, 167b, 192d, 231, 289c, and 291.}
\end{align*}
\]

- **Kernel Stack**: For several reasons we need a second stack when a process switches to kernel mode (using a system call, see Section 6.4). While it would be possible to share one kernel stack between all the processes, that would also limit us: With a single kernel stack we would run into problems when two or more processes need to enter kernel mode at the same time.

There’s also a security aspect: The kernel stack may contain kernel data that the process should not have access to.

We’ll put the kernel stack just under the kernel space of memory, at addresses below 0xc0000000.

\[
\begin{align*}
&\text{\textit{constants}} \ 112a \ + \equiv \ (44a) \ \langle 159b \ 162a \rangle \ [159c] \\
&\text{\texttt{#define}} \ \text{TOP_OF KERNEL_MODE_STACK} \ 0xc0000000 \\
&\text{Defines:} \\
&\quad \text{TOP_OF KERNEL_MODE_STACK, used in chunks 192b, 196a, 211b, 257b, 261, and 280a.}
\end{align*}
\]

Thus, the memory layout of a process is as shown in Figure 6.1. The double line below 0xc0000000 marks the barrier between process-specific and generic memory ranges: everything above 0xc0000000 is globally visible and identical in every address space, whereas the
lower addresses differ for each process, and they do not exist at all before the first process has been created.

Address ranges marked ‘K’ in the last column need kernel privileges to be accessed. Heap areas will only become available after they are manually requested. The ‘(U)’ annotation of the combined stack/heap area refers to the fact that it is not allocated when a process starts but rather can grow both from the top and from the bottom, depending on the process’ actions.

<table>
<thead>
<tr>
<th>Address Range</th>
<th>Usage</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xD4000000 – 0xFFFFFFFF</td>
<td>unused</td>
<td>–</td>
</tr>
<tr>
<td>0x00000000 – 0xD3FFFFFF</td>
<td>64 MByte Physical RAM (mapped)</td>
<td>K</td>
</tr>
<tr>
<td>0xC0000000 – 0xCFFFFFFF</td>
<td>Kernel Code and Data</td>
<td>K</td>
</tr>
<tr>
<td>0xBFFFF000 – 0xBFFFFFFF</td>
<td>Kernel Stack (4 KByte = one page)</td>
<td>K</td>
</tr>
<tr>
<td>0xB0000000 – 0xBFFFFEFF</td>
<td>unused</td>
<td>–</td>
</tr>
<tr>
<td>... – 0xAFFFFFFF</td>
<td>User Mode Stack</td>
<td>U</td>
</tr>
<tr>
<td></td>
<td>User Mode Stack (grows automatically)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Heap (can be grown with sbrk)</td>
<td></td>
</tr>
<tr>
<td>0x00000000 – ...</td>
<td>Process Code and Data</td>
<td>U</td>
</tr>
</tbody>
</table>

Figure 6.1: This is the memory layout of a process.

We will later provide a modified version of this memory model, when we introduce several threads (of the same process), but for now this description is sufficient to understand the process implementation.

### 6.1.2 Creating a New Address Space

Essentially, an address space is just a fresh page directory. Its kernel memory part (addresses 0xC0000000 and higher) will be identical to the kernel’s page directory which we’ve already set up earlier.

It is helpful to reconsider how the CPU (or the MMU) accesses the paging information: A register holds the address of the page directory which has 1024 entries, each of which is either null or points to a page table.

The upper $1024 - 768 = 256$ entries are responsible for the kernel memory ($0xC0000000 – 0xFFFFFFFF$), and the lower 768 entries are available for process memory ($0x00000000 – 0xBFFFFFFF$) with the upper part of each process’ private memory ($0xBFFFF000 – 0xBFFFFFFF$) being the kernel stack which is only available in kernel mode.

We want to allow for three different situations, as far as access to process memory and kernel memory is concerned:
pure kernel mode: The kernel is actively dealing with specific kernel tasks, such as memory management or interrupt service. The kernel’s view on memory in this state is as it was after we enabled paging: It sees its own memory (0xC0000000 – 0xCFFFFFFF) and the physical memory which is mapped to addresses starting at 0xD0000000, but no process memory. We will not use this mode once the first process was created.

process in user mode: A process is active and running in user mode. It only sees its own memory (0x00000000 – 0xAFFFF.FFFF: code, data, heap and user mode stack). The paging information will map the kernel stack and the kernel’s memory as well, but since it will be marked non-user-mode-accessible, that will be the same as not having it at all when running in user mode. In this mode any attempt to access kernel memory (either its data or its code) will generate a page fault—even if the address is valid.

process in kernel mode: A process has entered kernel mode via a system call or an interrupt has occurred. In this situation the page tables must give access to both the current process’ memory and the kernel memory. All 4 GByte of virtual memory are visible. The paging information can be the same as for the process in user mode: the current level of execution (kernel mode instead of user mode) grants the access to all of the virtual memory. (For handling an interrupt it is not necessary to see the current process’ user mode memory, so we could switch to pure kernel mode in order to prevent interrupt handlers from looking at process memory. But since we intend to trust our interrupt handlers, we will not do that.)

We reserve memory for an address space list. This list does not hold the page directory or the page tables it points to, but just the address of the page directory and some status information:

```c
typedef struct {
    void *pd; // pointer to the page directory
    int pid; // process ID (if used by a process; -1 if not)
    short status; // are we using this address space?
    memaddress memstart, memend; // first/last address below 0xc000.0000
    unsigned int stacksize; // size of user mode stack
    memaddress kstack_pt; // stack page table (for kernel stack)
    unsigned int refcount; // how many threads use this address space?
} address_space;
```

Defines: address_space, used in chunks 162b, 173a, and 304a.

`pd` holds the (virtual) address of the page directory. `memstart` and `memend` contain the first and last user mode address (for code, data and heap), and `stacksize` tells the size of the user mode stack. We also want to keep the address of the kernel mode stack’s page table handy, thus we will store it in `kstack_pt`. `refcount` lets us count how often the address space is used—for non-threaded processes this value will always be 1.
We define three possible values for the status field of an address space:

\[
\begin{align*}
\text{define AS_FREE 0} \\
\text{define AS_USED 1} \\
\text{define AS_DELETE 2}
\end{align*}
\]

Defines:
- AS_DELETE, used in chunk 166c.
- AS_FREE, used in chunks 162d, 169a, 171a, and 307a.
- AS_USED, used in chunks 122c, 162e, 163c, and 308c.

The meaning of \text{AS_FREE} and \text{AS_USED} is obvious, but why we need a third state \text{AS_DELETE} will only become clear when you reach the section that deals with process termination. Briefly, we cannot immediately destroy the address space of a process which has actively requested its termination, so that has to happen a bit later, and in the meantime the address space will be marked as “to be deleted”.

The address space table is an array of address space descriptors, and we will need a function that lets us search for a free entry in the table:

\[
\text{int get_free_address_space ()}
\]

It returns an integer value that serves as an index into the table.

We use the first entry of the array \text{address_spaces} for the kernel and let it point to the kernel page directory. We add the code for initializing this entry just after enabling paging:

\[
\text{address_spaces[0].status} = \text{AS_USED}; \\
\text{address_spaces[0].pd} = \&\text{kernel_pd}; \\
\text{address_spaces[0].pid} = -1; \quad \text{// not a process}
\]

Setting \text{pid} to \text{-1} marks this entry as an address space which belongs to no process.
Here is what we need to do in order to create a fresh address space: We first retrieve a new address space ID and mark its entry in the table as used. Then we reserve memory for a new page directory and copy the system’s one into it. Finally we set up some user space memory and add it to the page directory:

```c
int create_new_address_space (int initial_ram, int initial_stack);
```

The argument `initial_ram` defines the amount of process-private memory that should be allocated at once, similarly `initial_stack` is the initial size of the user mode stack which will always be just 4 KByte (though it can grow later). We expect the `initial_ram` and `initial_stack` values to be multiples of the page size (4 KByte)—if not, we will make them so, using this macro:

```c
#define MAKE_MULTIPLE_OF_PAGESIZE (x) \( x = \left( \frac{x + \text{PAGE\_SIZE} - 1}{\text{PAGE\_SIZE}} \right) \times \text{PAGE\_SIZE} \)
```

If the function call is successful, `create_new_address_space` returns the ID of the newly created address space, otherwise `-1`:

```c
int create_new_address_space (int initial_ram, int initial_stack) {
    MAKE_MULTIPLE_OF_PAGESIZE (initial_ram);
    MAKE_MULTIPLE_OF_PAGESIZE (initial_stack);
    // reserve address space table entry
    addr_space_id id;
    if ( (id = get_free_address_space ()) == -1 ) return -1; // fail
    address_spaces[id].status = AS_USED;
    address_spaces[id].memstart = BINARY_LOAD_ADDRESS;
    address_spaces[id].memend = BINARY_LOAD_ADDRESS + initial_ram;
    address_spaces[id].stacksize = initial_stack;
    address_spaces[id].refcount = 1; // default: used by one process
    // reserve memory for new page directory
    int frameno, pageno; // used in the following two code chunks
    if ( initial_ram > 0 ) {  // create initial user mode memory
        \( \text{create initial user mode memory} \)
    }
    if ( initial_stack > 0 ) {  // create initial user mode stack
        \( \text{create initial user mode stack} \)
    }
    return id;
}
```

As usual we use `request_new_page` to get a fresh page of virtual memory which will store the new page directory: that function will also update the page directories of all
already existing address spaces if it has to create a new page table (for addresses in the kernel memory).

\[164a\] \textit{reserve memory for new page directory 164a} \equiv \\
\text{page_directory } \ast \text{new}_pd = \text{request_new_page}(); \\
\text{if } (\text{new}_pd == \text{NULL}) \{ \text{ // Error} \\
\text{printf} ("\text{nERROR: no free page, aborting create_new_address_space\n"}); \\
\text{return -1;} \}; \\
\text{memset} (\text{new}_pd, 0, \text{sizeof} (\text{page_directory}));

Uses create_new_address_space 163c, memset 596c, NULL 46a, page_directory 103d, printf 601a, and request_new_page 120a.

For copying the kernel page directory to the new directory, we simply use an assignment; this copies all references to page tables which exist in the original (kernel) page directory.

\[164b\] \textit{copy master page directory to new directory 164b} \equiv \\
\ast \text{new}_pd = \text{kernel}_pd; \\
\text{memset} ((\text{char}*) \text{new}_pd, 0, 4); \text{ // clear first entry (kernel pd contains old reference to some page table)}

Uses kernel_pd 105a and memset 596c.

Note that once we have more than one address space, we must make sure that all changes to the kernel part (the addresses starting at 0xC0000000) will be made in each copy. The page tables of that area are all shared, but when we create a new page table, we have to write the new mapping into every page directory—you have already seen the code on page 122.

We modify the new page directory so that it contains information about the user mode memory, stack and kernel stack. For that purpose we will use a function

\[164c\] \textit{function prototypes 45a} \equiv \\
\text{int as_map_page_to_frame (int as, unsigned int pageno, unsigned int frameno)};

which can create mappings of page numbers to frame numbers in a specific address space. We will define it just afterwards; it finds out what page table entry to modify and, if needed, also creates a new page table and updates the page directory to point to it.

\[164d\] \textit{create initial user mode memory 164d} \equiv \\
pageno = 0; \\
\text{while } (\text{initial}_\text{ram} > 0) \{ \\
\text{if } ((\text{frameno} = \text{request_new_frame}()) < 0) \{ \\
\text{printf} ("\text{nERROR: no free frame, aborting create_new_address_space\n"}); \\
\text{return -1;} \}; \\
\text{as_map_page_to_frame} (\text{id}, \text{pageno}, \text{frameno}); \\
pageno++; \\
\text{initial}_\text{ram} -= \text{PAGE}\_\text{SIZE}; \\
\};

Uses as_map_page_to_frame 165b, create_new_address_space 163c, PAGE_SIZE 112a, printf 601a, and request_new_frame 118b.
Reserving memory for the user mode stack looks almost the same, we just let the stack grow downwards whereas above the memory addresses moved upwards: so we need to modify the page number with pageno-- instead of pageno++:

\[
\text{create initial user mode stack } 165a \equiv
\]

\[
\begin{align*}
\text{pageno} &= \text{TOP_OF_USER_MODE_STACK} / \text{PAGE_SIZE}; \\
\text{while} (\text{initial_stack} > 0) \{ \\
& \quad \text{if } ((\text{frameno} = \text{request_new_frame}()) < 0) \{ \\
& \quad \quad \text{printf ("nERROR: no free frame, aborting create_new_address_space\n");} \\
& \quad \quad \text{return } -1; \\
& \quad \}\; \text{pageno}--; \\
& \quad \text{as_map_page_to_frame (id, pageno, frameno);} \\
& \quad \text{initial_stack }\leftarrow \text{PAGE_SIZE}; \\
\}
\end{align*}
\]

Uses \text{as_map_page_to_frame} 165b, \text{create_new_address_space} 163c, \text{PAGE_SIZE} 112a, printf 601a, \text{request_new_frame} 118b, and \text{TOP_OF_USER_MODE_STACK} 159b.

We will now describe how to enter the page-to-frame mapping in the new address space’s page tables. Getting new physical memory is not a problem since we already have defined the function \text{request_new_frame} 118b() which reserves a new frame.

The function \text{as_map_page_to_frame} 165b creates such a mapping in a given address space. It will basically be a rewrite of parts of \text{enter frames in page table} 121a.

\[
\text{(function implementations) 100b)} \equiv
\]

\[
\begin{align*}
\text{int as_map_page_to_frame (int as, unsigned int pageno, unsigned int frameno) \{} \\
& \quad \text{// for address space as, map page \#pageno to frame \#frameno} \\
& \quad \text{page_table }*\text{pt}; \\
& \quad \text{page_directory }*\text{pd}; \\
& \quad \text{pd }= \text{address_spaces[as].pd}; \quad \text{// use the right address space} \\
& \quad \text{unsigned int pdindex }= \text{pageno}\/%1024; \quad \text{// calculate pd entry} \\
& \quad \text{unsigned int ptindex }= \text{pageno}\/%1024; \quad \text{// ... and pt entry} \\
& \quad \text{if } (\! \! \text{pd->ptds[pdindex].present }\!\! ) \{} \\
& \quad \quad \text{// page table is not present} \\
& \quad \quad \text{\langle create new page table for this address space 166a \rangle } \quad \text{// sets pt} \\
& \quad \text{\} else } \{} \\
& \quad \quad \text{// get the page table} \\
& \quad \quad \text{pt }= \text{(page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr }\ll 12); \\
& \text{\}); \\
& \quad \text{if } (\text{pdindex }< \text{704} ) \quad \text{// address below 0xb0000000 }\rightarrow \text{ user access} \\
& \quad \text{UMAP ( & (pt->pds[ptindex]), frameno }\ll 12 \); \\
& \quad \text{else } \\
& \quad \quad \text{KMAP ( & (pt->pds[ptindex]), frameno }\ll 12 \); \\
& \quad \text{return } 0; \\
& \text{\}); \\
\end{align*}
\]

Defines:
\text{as_map_page_to_frame}, used in chunks 164, 165a, 173a, 211a, 257c, and 291.

Uses \text{address_spaces} 162b, \text{KMAP} 101a, \text{page_directory} 103d, \text{page_table} 101b, \text{PHYSICAL} 116a, and \text{UMAP} 101a.
In the last lines of this function we differentiate between user mode and kernel mode memory and use the appropriate macro (UMAP or KMAP) to create an entry which allows or forbids access for processes in user mode: The address range \(0x00000000 - 0xffffffff\) grants the process access in user mode, whereas \(0xb0000000 - 0xffffffff\) shall only be accessible in kernel mode.

Remember that every page directory entry lets us address one page table which holds the addresses of up to 1024 pages, or \(\text{1024} \times \text{4096} = 4194304\) bytes (4 MByte) of memory.

\[
\frac{0xb0000000}{1024 \times 4096} = 704
\]

so we must use UMAP if pdindex < 704.

Now we need to explain how to create a new page table: We start with fetching a free frame and point to it from the page directory.

\[
\text{void destroy_address_space (addr_space_id id)};
\]

Its main task is to undo all the memory allocations that were performed when we created the address space so that no memory leaks occur: A sequence like

\[
\text{id = create_new_address_space (...);}\text{;}
\text{destroy_address_space (id);}\text{;}
\]

should return the global memory status to the same situation that it had before the creation:

\[
\text{void destroy_address_space (addr_space_id id) \{}\]
\[
\quad \text{// called only from syscall_exit(), with interrupts off}\]
\[
\quad \text{if (--)address_spaces[id].refcount > 0) return;}\]
\[
\quad \text{addr_space_id as = current_as;}\text{ // remember current address space}\]
\[
\quad \text{current_as = id;}\text{ // set current_as: we call release_page()}\]
\[
\}\text{ // all pages used by the process}
\]
6.1 Address Spaces for Processes

⟨destroy AS: release user mode stack 167b⟩ // all its user mode stack pages
⟨destroy AS: release page tables 167c⟩ // the page tables (0..703)

current_as = as; // restore current_as
address_spaces[id].status = AS_DELETE; // change AS status

// remove kernel stack (cannot do this here, this stack is in use right now)
add_to_kstack_delete_list (id);
}

Defines:
destroy_address_space, used in chunks 166b and 216b.
Uses add_to_kstack_delete_list 168d, addr_space_id 158b, address_spaces 162b, AS_DELETE 162a,
current_as 170b, release_page 122d, and syscall_exit 216b.

The comment in the above chunk’s first line refers to protection of the thread table data.
We will later discuss synchronization issues (in Chapter 11), and the address space table is
one of the critical data structures that must be treated carefully. So, one would expect to
see code for protecting it in this function, but this protection occurs elsewhere. In short,
we will only modify the address space table when we hold a lock for the thread table, and
that lock is already held when the kernel enters this function. (The same holds for the
create_new_address_space 163c function.)

Releasing the user mode memory is done in two simple steps:

⟨destroy AS: release user mode pages 167a⟩≡    (166c) [167a]
for (int i = address_spaces[id].memstart / PAGE_SIZE;
     i < address_spaces[id].memend / PAGE_SIZE;
     i++) {
    release_page (i);
};

Uses address_spaces 162b, PAGE_SIZE 112a, and release_page 122d.

⟨destroy AS: release user mode stack 167b⟩≡    (166c) [167b]
for (int i = TOP_OF_USER_MODE_STACK / PAGE_SIZE - 1;
     i > (TOP_OF_USER_MODE_STACK-address_spaces[id].stacksize) / PAGE_SIZE - 1;
     i--) {
    release_page (i);
};

Uses address_spaces 162b, PAGE_SIZE 112a, release_page 122d, and TOP_OF_USER_MODE_STACK 159b.

After releasing all the individual pages, we can also get rid of the page tables which refer to user mode memory:

⟨destroy AS: release page tables 167c⟩≡    (166c) [167c]
pager_directory *tmp_pd = address_spaces[id].pd;
for (int i = 0; i < 704; i++) {
    if (tmp_pd->ptds[i].present)
        release_frame (tmp_pd->ptds[i].frame_addr);
}

Uses address_spaces 162b, pager_directory 103d, and release_frame 119b.
In the last code chunk the loop goes from 0 to 703 since that is the last page directory entry which points to a page table that is used in user mode (cf. the discussion of UMAP vs. KMAP usage in the implementation of as_map_page_to_frame on page 166).

We will remove the kernel stack later when we’re not using it any more—doing this right now would crash the system because that memory is still in use. For that purpose we use a global variable which contains either 0 or the ID of an address space whose kernel stack needs removal. That is why we called the function

\[168a\] \begin{align*}
\text{add_to_kstack_delete_list} &: (addr\_space\_id \ id) \\
\end{align*}

in the above code. We allow up to 1024 entries in the kernel stack delete list:

\[168b\] \begin{align*}
\text{KSTACK\_DELETE\_LIST\_SIZE} &= 1024 \\
\end{align*}

The kernel stack delete list is just an array of address space IDs that we initialize with null values.

\[168c\] \begin{align*}
\text{kstack\_delete\_list}[\text{KSTACK\_DELETE\_LIST\_SIZE}] &= \{ 0 \} \\
\end{align*}

Entering an address space ID in the delete list is simple:

\[168d\] \begin{align*}
\text{add_to_kstack_delete_list} &: (addr\_space\_id \ id) \\
\text{begin critical section in kernel} &: \text{add_to_kstack_delete_list} \\
\text{end critical section in kernel} &: \text{add_to_kstack_delete_list} \\
\end{align*}

We have not shown the code for the scheduler yet, it is responsible for switching between the processes and is called regularly by the timer interrupt handler. Whenever the system activates the scheduler it will execute the following code chunk (scheduler: free old kernel stacks) which frees those old kernel stacks that we’ve put into the list:
6.1 Address Spaces for Processes

⟨scheduler: free old kernel stacks 169a⟩≡
⟨begin critical section in kernel 380a⟩
for (int entry = 0; entry < KSTACK_DELETE_LIST_SIZE; entry++) {
    if (kstack_delete_list[entry] != 0 && kstack_delete_list[entry] != current_as) {
        // remove it
        addr_space_id id = kstack_delete_list[entry];
        page_directory *tmp_pd = address_spaces[id].pd;
        page_table *tmp_pt = (page_table *) address_spaces[id].kstack_pt;
        // this is the page table which maps the last 4 MB below 0xC0000000
        for (int i = 0; i < KERNEL_STACK_PAGES; i++) {
            int frameno = tmp_pt->pds[1023-i].frame_addr;
            release_frame (frameno);
        }
    }
}
⟨end critical section in kernel 380b⟩
Uses addr_space_id 158b, address_spaces 162b, AS_FREE 162a, current_as 170b, KERNEL_STACK_PAGES 169b, kstack,
KSTACK_DELETE_LIST_SIZE 168b, memaddress 46c, page_directory 103d, page_table 101b, release_frame 119b,
and release_page 122d.

We haven’t defined the constant KERNEL_STACK_PAGES 169b yet: it tells the system how many pages it shall reserve for the kernel stack.

⟨constants 112a⟩≡
⟨define KERNEL_STACK_PAGES 4
define KERNEL_STACK_SIZE PAGE_SIZE * KERNEL_STACK_PAGES⟩
Defines:
KERNEL_STACK_PAGES, used in chunks 169a, 211, 257b, and 261.
KERNEL_STACK_SIZE, used in chunks 192b and 211b.
Uses PAGE_SIZE 112a.

We may sometimes also need to know the size of the kernel stack (KERNEL_STACK_SIZE 169b).

6.1.4 Switching between Address Spaces

In order to switch between two address spaces it is sufficient to load the new address space’s page directory address into the CR3 register.

Note that using the function activate_address_space 170c (which we show in this section) should be avoided because it has the side effect of switching the kernel stack. Even while it is implemented as inline function, it is still not safe to call it: parameter passing creates local variables (on the kernel stack) which are lost after the context switch. We will only use it when we start the very first process.
In earlier versions of the code, \langle scheduler: context switch \rangle was used to make a function call to activate_address_space() and it caused many problems (the operating system crashed). After moving the CR3 loading code directly into the context switch, the problems disappeared.

\[170a\] (function prototypes) \begin{align*}
\text{inline void activate_address_space (addr_space_id id) } & \text{ __attribute__((always_inline))}; \\
\text{addr_space_id current_as = 0}; & \text{ // global variable: current address space}
\end{align*}

Defines: current_as, used in chunks 120c, 123a, 152a, 156a, 166c, 169a, 170c, 173a, 210a, 216b, 232c, 233c, 255, 257, 260a, 279c, 289–91, 298a, 299a, 342b, 605c, and 614a.

Uses addr_space_id 158b.

\[170b\] (global variables) \begin{align*}
\text{memaddress virt} & \text{ = (memaddress)} \text{address_spaces[id].pd}; \text{ // get PD address} \\
\text{memaddress phys} & \text{ = mmu (0, virt);} \text{ // and find its physical address} \\
\text{asm volatile } & \text{("mov %0, %%cr3" : : "r"(phys)); \text{ // write CR3 register}} \\
\text{current_as = id}; & \text{ // set current address space} \\
\text{current_pd = address_spaces[id].pd}; & \text{ // set current page directory}
\end{align*}

Defines: activate_address_space, used in chunks 190a and 605c.

Uses addr_space_id 158b, address_spaces 162b, current_as 170b, current_pd 105a, memaddress 46c, mmu 172a, and write 429b.

Here we use another function called \text{mmu\_a} which emulates the behavior of the memory management unit (MMU) and calculates the physical address belonging to a virtual address with respect to an address space. We will implement it soon.

We provide a helper function

\[170d\] (function prototypes) \begin{align*}
\text{void list_address_spaces ()};
\end{align*}

which shows the list of used address spaces; it is only needed for debugging.

\[170e\] (function implementations) \begin{align*}
\text{void list_address_space (addr_space_id id) } & \text{ \{} \\
\text{int mem } & \text{ = (memaddress)} \text{address_spaces[id].pd}; \\
\text{int phys } & \text{ = mmu (id, (memaddress)} \text{address_spaces[id].pd}; \text{ // emulate MMU} \\
\text{int memstart } & \text{ = address_spaces[id].memstart;} \\
\text{int memend } & \text{ = address_spaces[id].memend;} \\
\text{int stack } & \text{ = address_spaces[id].stacksize;} \\
\text{printf } & \text{("ID: %d, MEM: %08x, PHYS: %08x - USER: %08x, USTACK: %08x\n",} \\
\text{id, mem, phys, memend-memstart, stack);} \\
\end{align*}

Defines: list_address_space, used in chunk 171a.

Uses addr_space_id 158b, address_spaces 162b, memaddress 46c, mmu 172a, and printf 601a.
6.1 Address Spaces for Processes

void list_address_spaces () {
    addr_space_id id;
    for (id = 0; id < MAX_ADDR_SPACES; id++) {
        if (address_spaces[id].status != AS_FREE) {
            list_address_space (id);
        }
    }
}

Defines:
- list_address_spaces, used in chunks 170d and 608b.
- Uses addr_space_id 158b, address_spaces 162b, AS_FREE 162a, list_address_space 170e, and MAX_ADDR_SPACES 158a.

list_address_space 170e also uses the mmu 172a function—it is time to provide its implementation. We start with a function mmu_p 172c which, given an address space ID and a page number, finds out whether the page is mapped in that address space and returns the frame number of the mapped frame.

unsigned int mmu_p (addr_space_id id, unsigned int pageno) {
    unsigned int pdindex = pageno/1024;
    unsigned int ptindex = pageno%1024;
    page_directory *pd = address_spaces[id].pd;
    if ( !pd->ptds[pdindex].present ) {
        return -1;
    } else {
        page_table *pt =物理(pd->ptds[pdindex].frame_addr << 12);
        if ( pt->pds[ptindex].present ) {
            return pt->pds[ptindex].frame_addr;
        } else {
            return -1;
        }
    }
}

Defines:
- mmu_p, used in chunks 120c, 123a, 171b, 172a, 261, 293d, 294, and 614a.
- Uses addr_space_id 158b, address_spaces 162b, page_directory 103d, page_table 101b, and PHYSICAL 116a.

and with mmu_p 172c, we can easily implement mmu 172a because we just have to split a virtual address into page number and offset, then call mmu_p 172c to find the frame number and reassemble that and the offset to form a physical address:
6 Implementation of Processes

[172a]  \begin{function implementations} \text{100b} \end{function implementations} \equiv
\begin{align*}
\text{memaddress \ mmu } (\text{addr\_space\_id \ id, memaddress vaddress}) & \{ \\
& \quad \text{unsigned \ int \ tmp} = \text{mmu\_p} \text{ (id, (vaddress \gg 12))}; \\
& \quad \text{if} \ (\text{tmp} = -1) \\
& \quad \quad \text{return} -1; \quad \text{// fail} \\
& \quad \text{else} \\
& \quad \quad \text{return} \ (\text{tmp} \ll 12) + (\text{vaddress \% PAGE\_SIZE});
\}
\end{align*}
\end{function implementations}

Defines:
\begin{itemize}
\item \text{mmu}, \text{used in chunks 170, 211, and 279c}.
\item \text{addr\_space\_id 158b, memaddress 46c, mmu\_p 171c, and PAGE\_SIZE 112a.}
\end{itemize}

Note that both functions return \(-1\) if the page or virtual address does not exist, but only when calling \text{mmu\_p} we can be sure that a return value of \(-1\) indicates a non-existing page—after all, \text{some} virtual address might be mapped to physical address 0xFFFFFFFF (which is the same as \(-1\)).

6.1.5  \textbf{Enlarging an Address Space}

We want to allow processes to increase their standard memory usage (which is 64 KByte). Unix systems provide an implementation of malloc as part of their standard library.

The \text{brk} system call (and corresponding library function) is still available on modern Unix systems, but its use is advised against. \text{brk} adds one or more pages to the calling process’ data “segment”. The function \text{sbrk} does the same but is more user-friendly: It takes an increment as argument, so if the process needs 16 KByte of extra memory, it can call \text{sbrk}(16*1024). \text{sbrk} returns the lowest address of the new memory: After executing \text{void *mem = sbrk(incr)}, the address range [\text{mem}; \text{mem} + \text{incr} - 1] is available to the process.

How can we do this in UTSR? Remember that each process uses an \text{address\_space} which has elements named \text{memstart} and \text{memend} (the last of which is the first address that is \text{not} available) and a pointer to the address space’s page directory (\text{pd}). Thus, \text{sbrk} just needs to

\begin{itemize}
\item acquire the needed number of frames,
\item modify the page directory so that the new frames are mapped just after the last old pages and
\item update the \text{memend} element.
\end{itemize}

It then returns the first (virtual) address of the first new page.

We start with the kernel-internal function \text{u_sbrk}; we expect that its argument is always a multiple of \text{PAGE\_SIZE}:

\begin{align*}
\begin{function prototypes} \text{45a} \end{function prototypes} \equiv
\begin{align*}
\text{void *} \text{u_sbrk} \ (\text{int incr});
\end{align*}
\end{function prototypes}
\end{align*}
### 6.1 Address Spaces for Processes

*Function implementations* 100b) + ≡ 173a

```c
void *u_sbrk (int incr) {
    int pages = incr / PAGE_SIZE;
    address_space *aspace = &address_spaces[current_as];
    memaddress oldbrk = aspace->memend;

    for (int i = 0; i < pages; i++) {
        int frame = request_new_frame();
        if (frame == -1) { return (void*)(-1); } // error!
        as_map_page_to_frame (current_as, aspace->memend/PAGE_SIZE, frame);
        aspace->memend += PAGE_SIZE;
    }
    return (void*) oldbrk;
}
```

Defines:
- u_sbrk, used in chunks 172–74, 233c, and 257c.
- Uses address_space 161, address_spaces 162b, as_map_page_to_frame 165b, current_as 170b, memaddress 46c, PAGE_SIZE 112a, and request_new_frame 118b.

Next we need to provide a system call for the u_sbrk function so that a process can call this function. So far, you have not seen how U11x implements system calls (we will show this in Chapter 6.4), so you might want to skip the following description and turn back to it when you reach the system call chapter. We've also put a reminder into that section.

As a brief summary, system calls are functions whose start addresses we enter in a syscall table. A process can make a system call by loading a syscall number (which serves as an index into that table) into the EAX register, storing arguments for the syscall in further registers (EBX, ECX, ...) and then executing the int 0x80 assembler instruction. Filling a syscall table entry is handled by the function install_syscall_handler.

The system call number __NR_brk is defined as 45. There is no sbrk system call since normally the sbrk function is implemented by calling a similar brk function. But we only implement u_sbrk and reuse the brk system call number.

### syscall prototypes 173b) ≡ (202a) 206c> 173b

```c
void syscall_sbrk (context_t *r);
```

### code for syscall_sbrk 173c) ≡ (173c)

```c
void syscall_sbrk (context_t *r) {
    // ebx: increment
    r->eax = (memaddress)u_sbrk (r->ebx);
    return;
}
```

### initialize syscalls 173d) ≡ (44b) 206f> 173d

```c
install_syscall_handler (__NR_brk, syscall_sbrk);
```

Uses __NR_brk 204c, install_syscall_handler 201b, and syscall_sbrk 174b.
This is the first of many appearances of a code pattern: Most system call handlers set
\( r->eax \) in order to provide a return value and then leave the function with \( r->eax \). To
simplify our code we will provide a macro \( \text{eax\_return} \) which combines these two activities
and also performs the type cast to an unsigned integer:

\[
\begin{align*}
\texttt{#define eax\_return(retval)} & \texttt{ \{ \texttt{r->eax} = (unsigned int)((retval)); \texttt{return; \}} \\
\end{align*}
\]

Defines:
\( \text{eax\_return} \), used in chunks 174b, 206d, 213d, 219c, 220a, 222b, 223e, 234b, 299a, 310a, 370d, 426b, 433b, 566d, 583a, 587d, and 590b.

With this macro we can rewrite \( \text{syscall\_sbrk} \) like this:

\[
\begin{align*}
\texttt{void syscall\_sbrk(clist\_t *r)} & \texttt{\{} \\
\texttt{\quad \text{\#ebx: increment}} \\
\texttt{\quad \text{eax\_return( u\_sbrk(r->ebx));}} \\
\texttt{\}}
\end{align*}
\]

Defines:
\( \text{syscall\_sbrk} \), used in chunk 173.
Uses \( \text{clist\_t} \) 142a, \( \text{eax\_return} \) 174a, and \( \text{u\_sbrk} \) 173a.

We also provide a user mode library function \( \text{sbrk} \) so that you can simply call \( \text{sbrk} \) in an application program (instead of manually inserting the necessary code for the system
call). Again, this will become clear once you reach the description of our system call
interface—which is only a few pages away. We only display the necessary code without
further explanation:

\[
\begin{align*}
\texttt{void *sbrk(int incr)} & \texttt{\}}
\end{align*}
\]

Defines:
\( \text{sbrk} \), used in chunk 174d.
Uses \( \text{sbrk} \) 174d.

\[
\begin{align*}
\texttt{void *sbrk(int incr) \{} & \texttt{return (void*)syscall2 (__NR_brk, incr); \}}
\end{align*}
\]

Defines:
\( \text{sbrk} \), used in chunk 174c.
Uses \( __NR\_brk \) 204c and syscall2 203c.

6.2 Thread Control Blocks and Thread Queues

TCB: The \textit{thread control block} (TCB) is the central place in the kernel where information about a
thread is held. In the times when people used to speak about processes instead of threads,
the TCB was called the PCB which stands for \textit{process control block}.

One main purpose of the TCB is to store the \textit{processor state} of a thread (sometimes also
called \textit{context}) during the times when it is not assigned to a physical processor. Note that
the processor state is not the same as the thread state. As you will see soon, the state of a
thread can be \textit{running}, \textit{blocked} etc. The processor state is all information that is necessary
to pretend that the processor has never executed any other thread as the one to which the
TCB belongs.
The TCB contains (among other data) the following information:

- A unique identifier of the thread. This is the so-called thread identifier (TID). In previous times, the TID was often called PID for process identifier. We also keep a PID entry in the TCB, it will be needed when we introduce threads as part of a process.
- Storage space to save the processor context, i.e., the registers, the stack pointer(s), etc.
- Depending on the thread state, the TCB contains an indication on what event the thread is waiting for if it is in state blocked.
- Information about the memory that this thread is using—we have already defined address spaces, and we will store an address space ID in the TCB.
- Any other information which may be useful to keep the system running efficiently. For example, statistical information could be stored here on how often the thread has been running in the past. This could help the scheduler make efficient scheduling decisions.

Note that the information about the address space must be handled differently in PCBs and in TCBS. In a system where multiple threads can run within one address space, there is an \( n : 1 \) mapping between threads and address spaces. In a classical Unix system with processes (one address space with exactly one thread), the mapping is \( 1 : 1 \) and each thread can store the full address space information in the PCB itself. With an \( n : 1 \) mapping, an extra data structure is necessary to avoid having redundant information in the TCBs.

So here is the declaration of the TCB structure. We have entries for the thread ID tid, the process ID pid, a parent process ID ppid (so we can build a process tree), the processor context (of type context_t) consisting of the general purpose registers and the special purpose registers and a reference to the address space in which the thread runs (which we already added to the data structure when we discussed address spaces). We also reserve place for three memory addresses which hold the instruction pointer, stack pointer and base pointer contents). More entries will follow later.

\[
\text{typedef struct }
\begin{array}{ll}
\text{thread_id } & \text{pid; } \quad // \text{ process id} \\
\text{thread_id } & \text{tid; } \quad // \text{ thread id} \\
\text{thread_id } & \text{ppid; } \quad // \text{ parent process} \\
\text{int } & \text{state; } \quad // \text{ state of the process} \\
\text{context_t } & \text{regs; } \quad // \text{ context} \\
\text{memaddress } & \text{esp0; } \quad // \text{ kernel stack pointer} \\
\text{memaddress } & \text{eip; } \quad // \text{ program counter} \\
\text{memaddress } & \text{ebp; } \quad // \text{ base pointer}
\end{array}
\]

\} \text{ TCB;}

Defines:
TCB, used in chunks 176b, 188, 190a, 210, 223e, 224f, 255c, 260a, 276c, 280a, 291, 562b, and 580–82.

Uses context_t 142a, memaddress 46c, and thread_id 178a.
The TCBs of all threads are collected in a large table within the kernel. This table is called the thread table. Again, in ancient times when the table was a collection of PCBs instead of TCBs, it was called process table.

The thread table is simply an array of TCBs. The size of the table must be finite, so there exists a maximum number of threads which can coexist at any point in time.

\[
\text{define MAX_THREADS 1024}
\]

Defines:
MAX_THREADS, used in chunks 176b, 188a, 217b, 219c, 223e, 281, 322b, and 605d.

We define the maximum number of threads here. It should somehow correspond to the maximum number of address spaces MAX_ADDR_SPACES\textsuperscript{158a} defined earlier in Section 6.1. For example, it doesn’t make sense to allow more address spaces than threads (since every thread can have at most one address space). We have set both values to 1024 which would let Uix run 1023 processes, each of which has its individual address space. The number 0 is reserved—both in the address space table (where it refers to the kernel address space) and in the thread table (because we use that entry for a different purpose related to the scheduler).

6.2.1 Thread State

For each thread, the TCB contains a state field. We will define the possible values which this field can hold later (p. 180), but here we already give an overview of the what can theoretically happen to a thread.

6.2.1.1 Simple State Model for Threads

Threads have a lifecycle. They are born, live and finally die. During their life they undergo many changes. For example, they sometimes are executed by a physical processor and sometimes not. This is what is called the thread state or simply state. The number and type of states together with the transitions which a thread can experience during its lifetime is called a state model.

The simplest state model which can be found in almost every textbook on operating systems consists of three states: running, ready and blocked. Here’s what these states mean:

- A thread in state running is actually executing on a physical processor. If there is more than one processor in the system, more than one thread can be in this state.
- A thread in state ready is not currently assigned to a physical processor, but it could be assigned. In other words, the thread is ready to run in case a physical processor...
becomes free. Many threads can be in this state at the same time; they are kept within a list called the ready queue.

- A thread in state blocked is waiting for a certain event. Only after this event has happened, it can become running or ready again. There can be many different events for which a thread can wait. For example, a thread could be waiting for a page fault to be serviced, i.e., waiting for an I/O operating to terminate. Another example is that a thread is waiting for a synchronization operation to be executed by another thread (see Chapter 11).

Usually an indication of the event for which the thread is waiting is part of the blocked state. This can be interpreted as many different blocked states. For simplicity, most textbooks therefore reduce these states to just one. Many threads can be in a blocked state at the same time. They are kept internally within one (or more) blocked queues.

We will use this state model in ULIx.

The possible transitions between thread states are depicted in Figure 6.2. We enumerate and explain them here now:

- **add**: a new thread is dynamically created and enters the set of threads in the state ready.
- **assign**: a new thread from state ready is assigned to the processor and becomes running.
- **block**: a running thread invokes a blocking system operation (e.g., I/O), runs into a page fault or must wait for some other event to continue operation. Now a new thread can become running. (Note the difference between the thread state blocked and the state transition block.)
- **deblock**: the event for which a blocked thread is waiting has happened. Consequently, the blocked thread is transferred to the state ready. (Sometimes this transition is also called ready, but since this can be confused with the thread state ready we prefer to call it deblock.)

![Figure 6.2: States and state transitions in the simple state model for threads.](image)
• **resign**: the thread which is currently running has finished executing parts of its program and leaves the physical processor. It transits from state running back into state ready. Now a new thread can become running.

• **retire**: a currently running thread has finished executing its program code and terminates its lifetime.

Not all possible transitions from one state to the other exist in the state model because a reduced state model decreases the complexity of the implementation. For example it is rather uncommon to transit from blocked directly to running. Similarly, a newly created thread must be ready first before it may become running.

The transitions of a thread from one state to the other are initiated by the operating system and happen “instantaneously”. Since a state change needs many machine instructions, real instantaneousness cannot be achieved, so the operating system simulates atomic transitions using synchronization operations in the kernel (see Chapter 11). In essence, the atomic transitions are implemented in such “atomic” kernel functions which carry the same name as the state transitions (e.g., assign, resign, etc.). The place in the kernel where all these functions are collected is called the dispatcher.

Here are the forward declarations of the dispatcher functions. They will be implemented on the following pages. Note that the dispatcher operation block takes an indication to the event on which the thread is blocking. This indication is encoded in the particular blocked queue to which the thread should be added. The data type of blocked_queue will be explained below.

```c
void add (thread_id t);
void block (blocked_queue *q, int new_state);
void deblock (thread_id t, blocked_queue *q);
void retire (thread_id t);
```

(Instead of a resign function we will later provide a code chunk ⟨resign⟩. Assigning happens only inside the scheduler, so we do not implement a specific function for this activity.)

### 6.2.1.2 Thread States with Swapping

In case of shortage of memory it may make sense to completely “swap out” a process with all its threads and virtual memory to external storage. In this case it is necessary to define an additional thread state swapped, which also leads to a more complex set of state
transitions since both a ready or blocked thread might be swapped out; depending on the implementation even an active process may ask for being swapped out. The system must remember the last state and restore it when it swaps the process back in (see Figure 6.3). ULinux does not implement swapping since it uses the more advanced paging model.

### 6.2.1.3 Thread Queues

The operating system has to perform bookkeeping of the state of threads. This can be done in several ways. One approach would be to store an entry state of an enumeration type in the TCB that can have the values blocked, ready or running. This is viable but not actually necessary. In modern operating systems the thread state is stored implicitly through the collection of linked lists. These lists contain threads and function as queues. The *ready queue* for example is a list of threads which all are in state ready.

As global data structures we therefore need a couple of global variables:

- For every CPU in the system we need a reference to the thread that is currently assigned to the processor. For monoprocessor systems (like those that ULinux supports) it is sufficient to provide a global variable `current_task` of type `thread_id`. For multiprocessor systems we would have to provide such a variable for every CPU in the system.
- A *ready queue* enumerating all threads that are in state *ready*.
- For every separate class of events which can cause a thread to go into state *blocked*, we need a *blocked queue* enumerating all threads that wait for such an event.
- In case we have a system with swapping, another list is necessary holding all swapped out threads. This is called the *swapped out queue*.

Figure 6.3: Thread states and state transitions for a system that swaps processes out and back in.
6.2.1.4 Thread States in the ULIx Implementation

ULIx does not support swapping but writes out individual pages to disk, so we will not need a swapped state. However, we will use several separate states to indicate a specific blocked state. (As we described above, we could use queue membership to indicate the specific blocked state, but using several states lets us access the state more quickly.) The following code chunk lists all the possible states that a process (or thread) can be in:

```c
// Thread states
#define TSTATE_READY 1 // process is ready
#define TSTATE_FORK 3 // fork() has not completed
#define TSTATE_EXIT 4 // process has called exit()
#define TSTATE_WAITFOR 5 // process has called waitpid()
#define TSTATE_ZOMBIE 6 // wait for parent to retrieve exit value
#define TSTATE_WAITKEY 7 // wait for key press event
#define TSTATE_WAITFLP 8 // wait for floppy
#define TSTATE_LOCKED 9 // wait for lock
#define TSTATE_STOPPED 10 // stopped by SIGSTOP signal
#define TSTATE_WAITHD 11 // wait for hard disk
```

Defines:
- TSTATE_EXIT, used in chunks 216b, 217a, 260a, 281, and 564a.
- TSTATE_FORK, used in chunks 210b and 255c.
- TSTATE_LOCKED, used in chunks 361c, 366a, 391, and 392.
- TSTATE_READY, used in chunks 184b, 186b, 278, 563b, and 564c.
- TSTATE_STOPPED, used in chunk 563.
- TSTATE_WAITFLP, used in chunks 545b and 564c.
- TSTATE_WAITFOR, used in chunks 217a, 219c, 281, and 564c.
- TSTATE_WAITKEY, used in chunks 416b and 564c.
- TSTATE_ZOMBIE, used in chunks 152b, 217a, and 281.

We also define a list of state names which can be used when displaying the process list:

```c
char *state_names[12] = {
    "---", "READY", "---", "FORK", "EXIT", "WAIT4", "ZOMBY", "W_KEY", // 0..7
    "W_FLP", "W_LCK", "STOPD", "W_IDE" // 8..11
};
```

Defines:
- state_names, used in chunk 605d.

Figure 6.4 shows the state transitions in ULIx. The various TSTATE_* states are shown as a single state in order to simplify the picture.

6.2.2 Implementing Lists of Threads

Queues are standard data structures offered by almost all modern programming languages. As an example, Java provides the generic class ArrayList<E> in which objects of any type E can be stored and manipulated with standard operations like add(), size() and get(). Unfortunately, “plain” C does not offer this convenience so we have to implement queues by ourselves.
6.2 Thread Control Blocks and Thread Queues

As explained in Section 6.2.1.3, we have to maintain a couple of thread queues within the kernel. In ULIX we maintain only two: the ready queue and the blocked queue. In fact, the blocked queue is not a single queue but there can be multiple blocked queues, one for every event upon which a thread can wait.

When implementing such queues, we could think about using the standard implementation of a (double) linked list found in any introductory textbook on programming. However these implementations usually are examples of programming with dynamic memory allocation, e.g., in C using the malloc library call to allocate fresh memory on the heap. This would be a problem in ULIX since we neither have a heap nor a library to call into.

So how can we program linked lists without allocating memory? The first option is to do it like Knuth did it in \TeX \[Knu86]\ and provide both a large memory area plus functions for memory allocation and deallocation by ourselves. Since this would be overkill, we choose the second option, which is also the option taken in many operating systems: We use the thread table to implement lists. The idea is to declare two additional entries in the thread control block: one entry called next and one called prev. Both point to other entries in the thread table. So consider a thread control block TCB\(_t\). The entry \(t\).next points to the “next” thread in the queue that \(t\) belongs to. Similarly, \(t\).prev points to the “previous” thread in \(t\)’s queue.

We define the range of these two pointers to be \(\text{thread}_id\)\(_{178a}\).

\[
\langle \text{more TCB entries}\rangle_{158c} \equiv \begin{align*}
\text{thread}_id\ \text{next}; & \quad \text{// id of the \"next\" thread} \tag{175} \\
\text{thread}_id\ \text{prev}; & \quad \text{// id of the \"previous\" thread} \\
\end{align*}
\]

Uses thread_id 178a.

An example of the semantics of the prev and next entries in the thread table is shown in Figure 6.5. It shows that the thread identifier 0 is used as an “end marker” for the lists. It

![Diagram of thread states and state transitions](image.png)
Figure 6.5: Implementation of ready queue and blocked queues. The beginning of the ready queue is implicitly defined by entry 0 in the thread table. The beginning of a blocked queue is a pair of thread identifiers pointing into the thread table from “outside”.

also shows that the prev entry of the first entry in the queue points to the last element in the queue. In this way, it is easily possible to navigate through the queues in any way which is convenient.

Figure 6.5 also shows a small implementation trick. The thread identifier of the thread itself is always equal to the index of the thread in the thread table. Given a TID of \( t \), then \( \text{thread_table}_{176b}[t] \) is the thread control block of that thread. This also means that the entry \( \text{tid} \) in the thread control block is more or less superfluous.

Now since we are using the value 0 to mark the end of a list, the entry 0 in the thread table has become more or less useless to store thread information. We use it instead as the “anchor” of the ready queue. So to access the first element in the ready queue, we just need to look into:

\[
\text{thread_table}_{176b}[0].\text{next}
\]

The last entry in the ready queue can similarly be accessed using the following expression:

\[
\text{thread_table}_{176b}[0].\text{prev}
\]

The ready queue in the figure contains threads 1, 4 and 7 (in this order).

Recalling the simple state model of threads in Section 6.2.1.1, every thread is in exactly one state at any time. This means that a thread is either running, ready or blocked. It also means that a thread can be in at most one queue at a time. In case the thread is blocked instead of ready, we can re-use the \( \text{prev} \) and \( \text{next} \) entries in the thread table to implement
the blocked list. We only need to have an anchor for this blocked list. This anchor will be a structure similar to entry 0 in the thread table, but without all the extra fields.

\[
\text{typedef struct \{}
\begin{align*}
\text{thread_id & next; & // id of the `next'' thread} \\
\text{thread_id & prev; & // id of the `previous'' thread}
\end{align*}
\} \text{ blocked_queue};
\]

Defines:

\text{blocked_queue}, used in chunks 183c, 185–87, 218b, 323d, 360a, 362, 365a, 366c, 368, 391a, 522a, 529a, 544d, and 606.
Uses \text{thread_id} 178a.

So assume \(b\) is a variable of type \text{blocked_queue} 183a, representing a blocked list. If both entries in \(b\) are 0, then the list is empty. If not, then using the thread table we can now find the first, second etc. element by following the next pointers. This way, we can traverse the entire list until we reach an entry in which \text{next}==0. That’s the end of the list. Looking again at Figure 6.5, the blocked queue contains threads 2, 5 and 6 (in this order).

Finally, here’s a useful function to initialize a blocked queue:

\[
\text{void initialize_blocked_queue (blocked_queue *q)};
\]

This is just to encapsulate the semantics of “emptiness”.

6.2.2.1 Dispatcher Operations as Critical Sections

Before we actually implement queue operations we need to make a slight forward reference to Chapter 11 on synchronization and introduce the notion of a critical section, at least intuitively. Briefly spoken, a \textit{critical section} is a sequence of code that accesses shared resources. Such a section is critical, because (as is explained in Chapter 11) concurrent accesses to a shared resource can wreak havoc with these resources, potentially making them unusable. The shared resources in question here are the shared kernel queues. What we need to ensure is that no two critical sections are executed concurrently. We refrain here from explaining how this can be achieved. What we however do at this point is to mark the beginning and end of the critical sections using the code chunk \(\langle \text{begin critical section in kernel} 380a \rangle\) and \(\langle \text{end critical section in kernel} 380b \rangle\) and leave the implementation to Chapter 11.
6.2.2.2 Implementing the Ready Queue

We now provide some convenient functions to add and remove threads from the queues. We start with the ready queue. The function `add_to_ready_queue(t)` adds the thread with identifier `t` to the `end` of the ready queue. It assumes that the TCB of thread `t` has been set up and initialized already.

The function `remove_from_ready_queue(t)` removes the thread with identifier `t` from the ready queue. It assumes that `t` is contained in the ready queue.

Adding to the end of the ready queue is as easy since we have a double linked list.

Removing is similarly easy.

We initialize the ready queue to be empty.

Note that we cannot use our `initialize_blocked_queue` function. Even though we access the same elements (prev and next), the structures do not match.
6.2.2.3 Implementing a Blocked Queue

Blocked queues are implemented similar to the ready queue, except that the functions are parameterized with a blocked list anchor defined above. We provide again two functions to add and remove a thread from a blocked list. Adding to the queue happens at the “end” of the queue. An additional function allows to inspect the “front” queue element.

\[
\begin{align*}
\text{add_to_blocked_queue} & (\text{thread_id } t, \text{blocked_queue } \ast bq); \\
\text{remove_from_blocked_queue} & (\text{thread_id } t, \text{blocked_queue } \ast bq); \\
\text{front_of_blocked_queue} & (\text{blocked_queue } bq); \\
\end{align*}
\]

We implement the easy inspector function to retrieve the front of a blocked queue first. It is so easy that we could have avoided writing this function altogether, but we spell it out for students who have learnt the concept of information hiding.

\[
\begin{align*}
\text{front_of_blocked_queue} & (\text{blocked_queue } bq) \\
& \{ \\
& \quad \text{return } bq.\text{next}; \\
& \} \\
\end{align*}
\]

We now implement `add_to_blocked_queue`. Adding happens at the end of the queue. The following code is an adaption of the code for the ready queue. The conditional statement at the end is necessary since `thread_table[bq][0]` is not the anchor of a blocked queue.

\[
\begin{align*}
\text{add_to_blocked_queue} & (\text{thread_id } t, \text{blocked_queue } \ast bq) \{ \\
& \begin{aligned}
& \text{begin critical section in kernel } 380a \\
& \text{thread_id last } = bq->\text{prev}; \\
& bq->\text{prev } = t; \\
& \text{thread_table}[t].\text{next } = 0; \quad / \! / \ t \text{ is } `\text{last}' \text{ thread} \\
& \text{thread_table}[t].\text{prev } = \text{last}; \\
& \text{if } (\text{last } = 0) \{ \\
& \quad \text{bq->next } = t; \\
& \} \text{ else } \{ \\
& \quad \text{thread_table}[\text{last}].\text{next } = t; \\
& \} \\
& \end{aligned} \\
& \begin{aligned}
& \text{end critical section in kernel } 380b \\
& \} \\
\end{aligned}
\end{align*}
\]

Removal is similar to the function of the ready queue, except for again the special cases at the end.
6.2.2.4 Simple Dispatcher Operations

We now look at the implementations of the three simplest dispatcher operations. These are `add`, `retire`, and `deblock`. They are simple because they basically only move threads from one queue to the other.

The functions `add` and `retire` take as parameter the identifier of the thread which is newly born or about to die. The function `deblock` needs another argument: The blocked queue from which the thread is to be removed. Note that `add` and `retire` do not need to be declared as critical sections, because the queue operation already is. `deblock` however must be executed without interruption so that kernel data structures remain consistent (see Chapter 11).

```c
void add (thread_id t) {
    add_to_ready_queue (t);
}

void retire (thread_id t) {
    remove_from_ready_queue (t);
}

void deblock (thread_id t, blocked_queue *q) {
    remove_from_blocked_queue (t, q);
    add_to_ready_queue (t);
    thread_table[t].state = TSTATE_READY;
}
```
6.2 Thread Control Blocks and Thread Queues

Defines:
- `deblock`, used in chunks 217a, 281, 322a, 362, 366c, 368, 391a, 522c, 532d, and 546a.

Uses `add_to_ready_queue` 184b, `blocked_queue` 183a, `remove_from_blocked_queue` 186a, `remove_from_ready_queue` 184c, `thread_id` 178a, `thread_table` 176b, and `TSTATE_READY` 180a.

For blocking the current thread we provide a function `block` which takes two arguments:
a blocked queue that the thread shall be moved to and the new state (e.g. `TSTATE_WAIT_HD`): 

\[
\text{function implementations } 100b\] \(\equiv\) (44a) <186b 188d> \[187a\]
\[
\begin{align*}
\text{void block (blocked_queue *q, int new_state) } & \\
\text{if (current_task == 0) return;} & \\
\text{thread_table[current_task].state = new_state;} & \\
\text{remove_from_ready_queue (current_task);} & \\
\text{add_to_blocked_queue (current_task, q);} & \\
\text{end critical section in kernel} 380b
\end{align*}
\]

Uses `add_to_blocked_queue` 185c, `blocked_queue` 183a, `current_task` 192c, `remove_from_ready_queue` 184c, and `thread_table` 176b.

Note that with the above functions we can easily write code that deblocks the “front” element from a blocked queue (if it exists) as follows:

\[
deblock 186b (\text{front_of_blocked_queue} 188b (bq), &bq);
\]

6.2.3 Allocating and Initializing a New TCB

Whenever we create a new thread or process, we will need a fresh TCB entry and initialize it. We add a `used` entry to the thread control block structure `TCB` 175

\[
\langle \text{more TCB entries} 158c \rangle \equiv (175) \langle 181 205b \rangle \[187b\]
\]

\[
\text{boolean used;}
\]

which lets us declare an entry as used. (Since we initialize the TCB structures with null bytes, we use used and not `free`: remember that `false=0`.)

This will allow us to quickly find a free TCB when we create a new thread. Instead of adding such a field, we could have used a bitmap, but since we restrict ourselves to 1024 TCBs, not much space is wasted this way, and searching for a free TCB will be quick.

We will remember in a global variable

\[
\langle \text{global variables} 92b \rangle \equiv (44a) \langle 180b 192c \rangle \[187c\]
\]

\[
\text{thread_id next_pid = 1;}
\]

Defines:
- `next_pid`, used in chunk 188.

Uses `thread_id` 178a.

at which thread number we will start our search (instead of always searching from 1): this will later lead to a continuous sequence of process/thread numbers: even if we terminate a thread, its TCB will not be recycled immediately.
Implementation of Processes

(find free TCB entry 188a) ≡

```
boolean tcbfound = false;
thread_id tcbid;
for (tcbid = next_pid; ((tcbid<MAX_THREADS) && (!tcbfound)); tcbid++) {
    if (thread_table[tcbid].used == false) {
        tcbfound = true;
        break; // leave for loop
    }
}
```

Uses MAX_THREADS 176a, next_pid 187c, thread_id 178a, and thread_table 176b.

However, once we’ve reached the maximum number (1023), the search for a free TCB will start over, and from that point on thread numbers will no longer indicate the order of creation of the threads.

(find free TCB entry 188a) +≡

```
if (!tcbfound) {
    // continue searching at 1
    for (tcbid = 1; ((tcbid<next_pid) && (!tcbfound)); tcbid++) {
        if (thread_table[tcbid].used == false) {
            tcbfound = true;
            break; // leave for loop
        }
    }
}

if (tcbfound) next_pid = tcbid+1; // update next_pid:
// either tcbfound == false or tcbid == index of first free TCB
```

Uses next_pid 187c, TCB 175, and thread_table 176b.

Once we have a free address space (or reuse one) and also have a free TCB, we can connect them:

(function prototypes 45a) +≡

```
int register_new_tcb (addr_space_id as_id);
```

Uses addr_space_id 158b and register_new_tcb 188d.

This function searches for a free TCB, marks it as used and enters the address space ID which we provide as an argument. Thus, whenever we create a new thread, we always call create_new_address_space 163c first and register_new_tcb 188d afterwards:

(function implementations 100b) +≡

```
int register_new_tcb (addr_space_id as_id) {
    // called by u_fork()
    (find free TCB entry 188a)
    if (!tcbfound) return -1; // no free TCB!
    thread_table[tcbid].used = true; // mark as used
    thread_table[tcbid].addr_space = as_id; // enter address space ID
    return tcbid;
}
```

Defines:
register_new_tcb, used in chunks 188c, 190a, 210a, and 255b.

Uses addr_space_id 158b, TCB 175, thread_table 176b, and u_fork 209c.
6.3 Starting the First Process

Starting the first (init) process is different from how all further processes are created: We have to manually set up the memory regions and data structures and load a first program from the disk. Of course, this requires filesystem support which we will implement later—for now assume that the kernel can use filesystem functions similar to the standard Unix functions open, read and close.

Once the init process is running, all further processes will be created using fork (see Section 6.5). This is what we need to do:

- Setup the TCB (thread control block) list and mark the first TCB as used.
- Create a new address space and reserve memory for user mode (low addresses, with user access).
- Load the process binary from disk into the new address space.
- Reserve memory for the process’ kernel stack (low addresses, without user access).
- Enter all the information in the new TCB.
- Update a data structure called TSS (Task State Segment, see Section 6.3.5).
- Switch from kernel mode (ring 0) to user mode (ring 3) and start executing the process’ code.

```c
void start_program_from_disk (char *progname) {
    (start program from disk: prepare address space and TCB entry)
    (start program from disk: load binary)
    (start program from disk: create kernel stack)
    (start program from disk: set uid, gid, euid, egid)
    (start program from disk: activate the new process)
};
```

As you can see, the start routine is rather complex. We discuss the necessary tasks step by step in the following sections, with one exception: The code chunk (start program from disk: set uid, gid, euid, egid) will be explained much later when we discuss user and group management.

6.3.1 Preparations

We start with registering a new thread control block and fresh address space and entering useful data. The following code chunk contains a few instructions that will not make much sense to you right now since they deal with kernel components you have not seen
yet. For example, it sets the TCB element cwd (current working directory) to "/" and the file descriptors 0, 1 and 2 to standard input, standard output and standard error output—all of that will be discussed in Chapter 12 where we introduce the Ulix filesystem.

It is important that we activate the new process’ address space at this step, because right after that we will load the program binary of the init program into the lower memory of the process, and that would be unavailable in the kernel’s address space.

\[
\text{(start program from disk: prepare address space and TCB entry)} \equiv \\
\text{// create new address space (64 KB + 4 KB stack) and register TCB entry} \\
\text{addr_space_id as = create_new_address_space (64*1024, 4096);} \\
\text{thread_id tid = register_new_tcb (as); // get a fresh TCB} \\
\text{TCB *tcb = &thread_table[tid];} \\
\text{// fill TCB structure} \\
tcb->tid = tcb->pid = tid; \quad \text{// identical thread/process ID} \\
tcb->ppid = 0; \quad \text{// parent: 0 (none)} \\
tcb->terminal = 0; \quad \text{// default terminal: 0} \\
\text{memcpy (tcb->cwd, "/", 2);} \quad \text{// set current directory} \\
\text{memcpy (tcb->cmdline, "new", 4);} \quad \text{// set temporary command line} \\
\text{thread_table[tid].files[0] = DEV_STDIN;} \quad \text{// initialize standard I/O} \\
\text{thread_table[tid].files[1] = DEV_STDOUT;} \quad \text{// file descriptors} \\
\text{thread_table[tid].files[2] = DEV_STDERR;} \\
\text{for (int i = 3; i < MAX_PFD; i++) tcb->files[i] = -1;} \\
\text{activate_address_space (as); // activate the new address space} \\
\]

Uses activate_address_space 170c, addr_space_id 158b, create_new_address_space 163c, cwd, DEV_STDERR 415c, DEV_STDIN 415c, DEV_STDOUT 415c, MAX_PFD 424b, memcpy 596c, register_new_tcb 188d, thread_id 178a, and thread_table 176b.

6.3.2 Loading the Program

Loading the init program is simple because we do not use a special file format for the file, but instead link it into a “flat binary”, similar to the historical .COM files that MSDOS and CP/M used [Vil96, p. 171–175, 182–189]. The loader assumes that the filesize is less than 32 KByte and simply reads the whole file (or its first 32 KByte) into the virtual memory location that starts at BINARY_LOAD_ADDRESS,159a. We have set that constant to 0x0 in Section 6.1.1.

\[
\text{(start program from disk: load binary)} \equiv \\
\text{int fd = u_open (progname, O_RDONLY, 0);} \quad \text{// open the file} \\
\text{u_read (fd, (char*)BINARY_LOAD_ADDRESS, PROGSIZE);} \quad \text{// load to virtual address 0} \\
\text{u_close (fd);} \quad \text{// close the file} \\
\]

Uses BINARY_LOAD_ADDRESS 159a, 0_RDONLY 460b, PROGSIZE 190b, u_close 418a, u_open 412c, and u_read 414b.
The function `start_program_from_disk` is called with the argument "/init", so we need an `init` binary in the root directory of the root disk. That program does not do much, but only starts the `login` program via the `execv` function.

```c
#include "ulixlib.h"

void umain() {
    char *args[] = {"/bin/login", 0};
    execv (args[0], args);
    printf("exec failed\n"); for (;;);
}
```

For compiling this flat binary, we need a special linker configuration file that lets the GNU linker `ld` create such a format:

```ld
OUTPUT_FORMAT("binary")
phys = 0x00000000;
virt = 0x00000000;
SECTIONS {
    . = phys;

    .setup : AT(phys) {
    *(.setup)
}

    .text : AT(code - virt) {
    code = .;
    *(.text)
    *(.rodata*)
    . = ALIGN(4096);
}

    .data : AT(data - virt) {
    data = .;
    *(.data)
    *(.data*)
    . = ALIGN(4096);
}

    .bss : AT(bss - virt) {
    bss = .;
    *(COMMON*)
    *(.bss*)
    . = ALIGN(4096);
    end = .;
}
```

In the makefile for the user mode files (lib-tools/Makefile) the `init` program will later be compiled and linked with

```bash
$(CC) $(CCOPTIONS) -g -c ulixlib.c
$(CC) $(CCOPTIONS) -c init.c
# link it with linker script "process.ld"
$(LD) $(LDOPTIONS) -T process.ld -o init init.o ulixlib.o
```

### 6.3.3 Creating the Kernel Stack

Next we need to provide a kernel stack for the process. So far, ULIX has used the initial kernel stack defined as `_sys_stack95a` in the assembler file `start.asm`, but as we explained earlier, we need a separate kernel stack for every process (or thread).
We have already defined the number of kernel stack pages, KERNEL_STACK_PAGES, so now we simply register as many frames and write a mapping into the page table via as_map_page_to_frame.

```
unsigned int framenos[KERNEL_STACK_PAGES]; // frame numbers of kernel stack
for (int i = 0; i < KERNEL_STACK_PAGES; i++) {
  framenos[i] = request_new_frame();
  as_map_page_to_frame(current_as, 0xbffff - i, framenos[i]);
}
```

After that we need to store the information about the process kernel stack into two TCB fields esp0 and ebp.

```
char *kstack = (char*) (TOP_OF_KERNEL_MODE_STACK - KERNEL_STACK_SIZE);
memaddress adr = (memaddress)kstack; // one page for kernel stack
tcb->esp0 = (uint)kstack + KERNEL_STACK_SIZE; // initialize top-of-stack and
tcb->ebp = (uint)kstack + KERNEL_STACK_SIZE; // ebp (base pointer) values
```

6.3.4 Activating the New Process

Finally we can activate the process. We’ve completed all the required steps, and the program sits in the memory, waiting to be started. Let’s declare the variable current_task (that always holds the ID of the currently executing process or thread)

```
thread_id current_task;
```

and initialize it. We also add the init process to the ready queue and enable the scheduler. Then the last step is switching to user mode.

```
current_task = tid; // make this the current task
add_to_ready_queue (tid); // add process to ready queue
```

The cpu_usermode function will be written in Assembler, we discuss it in detail in the following section.
6.3 Starting the First Process

6.3.5 Configuring the TSS Structure and Entering User Mode

The Intel processor provides no command that would let us switch to user mode explicitly, but there is a way for returning to user mode which requires that the stack is set up properly when executing an \texttt{iret} instruction. That is what normally happens when, for example, a process already runs in user mode and an interrupt forces a jump to the interrupt handler—the CPU modifies the stack so that when the handler executes \texttt{iret}, execution will continue in the process. To make the system switch back to ring 3, the stack contains (besides other values) values which will be loaded into the CS and SS segment registers (which tell the CPU what segments to use for code and stack).

The segment registers always contain a value which is a multiple of 8, making them an offset for the GDT whose entries are eight bytes long. Thus, the three lowest bits of a segment register value are always 0. What we have not mentioned yet is that the CPU modifies the lowest two bits when it pushes the register value on the stack (on interrupt entry), and it reads them when it pops the registers back from the stack (during \texttt{iret}). These two bits are then interpreted as the privilege level to which the CPU shall switch (see Figure 6.6).

```
val = seg | 3;
```

However, we cannot use the segment descriptors which we have created during the system initialization: both the code and the data segment descriptors have the entry flags (which contains a two-bit value descriptor privilege level (DPL), describing the necessary level for accessing this segment) set to 0—the system would halt, because it would switch to user mode but would not be allowed to use the memory. So we need two new segment descriptors which are designed specifically for user mode. They are identical to the old descriptors except for the flags entry where they have the DPL value set to 3 instead of 0.

So here’s how we fill the descriptors:
install GDTs for User Mode

```c
fill_gdt_entry (3, 0, 0xFFFFFFFF, 0b11111010, 0b1100);
fill_gdt_entry (4, 0, 0xFFFFFFFF, 0b11110010, 0b1100);
```

Uses `fill_gdt_entry`.

The numbering continues with 3 since we’ve already filled the null descriptor (0) and the kernel mode code (1) and data (2) segment descriptors (see page 110).

For a better overview, we repeat the explanation of the old (kernel mode) GDT entries and add the two new entries:

- `10011010b` for the kernel code segment
  (present; ring 0; fixed-1; executable; exact privilege level; allow reading; not accessed)
- `10010010b` for the kernel data segment
  (present; ring 0; fixed-1; not executable; grow upwards; allow writing; not accessed)
- `1111010b` for the user mode code segment
  (present; ring 3; fixed-1; executable; exact privilege level; allow reading; not accessed)
- `1110100b` for the user mode data segment
  (present; ring 3; fixed-1; not executable; grow upwards; allow writing; not accessed)

In order to enter user mode we also have to create a structure called **TSS** (task state segment) which is another (and final) entry in the GDT; we must load its GDT offset in a special **task register** (`TR`) using the `ltr` instruction.

The TSS is a 104 bytes long data structure [Int11, p. 303], shown in Figure 6.7. The CPU designers had intended that operating system developers would supply such a structure for each task (process or thread), and it is possible to simplify the task switch by following this suggestion. However, we decided to ignore this possibility and do the task switch without the help of the CPU because that is more instructional.

In our TSS type definition we only mention the elements which we may need and combine less interesting areas of the structure in `long long` elements (`u1`, `u2`, `u3`):

```c
typedef struct {
    unsigned int prev_tss : 32; // unused: previous TSS
    unsigned int esp0, ss0 : 32; // ESP and SS to load when we switch to ring 0
    long long u1, u2 : 64; // unused: esp1, ss1, esp2, ss2 for rings 1 and 2
    unsigned int cr3 : 32; // unused: page directory
    unsigned int eip, eflags : 32;
    unsigned int eax, ecx, edx, ebx, esp, ebp, esi, edi, es, cs, ss, ds, fs, gs : 32;
    // unused (dynamic, filled by CPU)
    long long u3 : 64; // unused: ldt, trap, iomap
} __attribute__((packed)) tss_entry_struct;
```

Defines:
- `tss_entry_struct`, used in chunk 195.

Most of the fields are only useful when using the TSS to perform task switching: they store parts of the task context so that it is not necessary to keep track of them in the thread control block. If you are interested in this approach, you can read more about it in the Intel manual [Int11].
The esp0 field must hold the address of the top of the kernel stack, and ss0 must contain the segment number for kernel mode (0x10). The CPU will automatically set the stack pointer to that value when it switches from user mode to kernel mode (ring 0).

We use the thread control block entry to store and retrieve the process context. That is why we need only one TSS. (We cannot omit the TSS completely because the CPU demands that one exists.)

```
(global variables 92b) +≡
tss_entry_struct tss_entry;
```

Defines:
tss_entry, used in chunk 197a.
Uses tss_entry_struct 194b.

<table>
<thead>
<tr>
<th>I/O Map Base Address</th>
<th>(reserved)</th>
<th>LDT Segment Selector</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>(reserved)</td>
<td></td>
<td>GS</td>
<td>96</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>FS</td>
<td>92</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>DS</td>
<td>88</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>SS</td>
<td>84</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>CS</td>
<td>80</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>ES</td>
<td>76</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>ESI</td>
<td>72</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>EBP</td>
<td>68</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>ESP</td>
<td>64</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>EBX</td>
<td>60</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>EDX</td>
<td>56</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>ECX</td>
<td>52</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>EAX</td>
<td>48</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>EFLAGS</td>
<td>44</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>EIP</td>
<td>40</td>
</tr>
<tr>
<td>CR3 (PDBR)</td>
<td></td>
<td>SS2</td>
<td>36</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>ESP2</td>
<td>32</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>SS1</td>
<td>28</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>ESP1</td>
<td>24</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>SS0</td>
<td>20</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>ESP0</td>
<td>16</td>
</tr>
<tr>
<td>(reserved)</td>
<td></td>
<td>Previous Task Link</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 6.7: The TSS (Task State Segment) structure.
We add the data from `tss_entry` to the GDT definition (see pages 110 and 193), this is the last GDT entry, and it uses number 5 since we’ve already used the entries 0–4.

\[
\text{write_tss}(5, 0x10, \text{TOP_OF_KERNEL_MODE_STACK}); \quad // \text{gdt no., ss0, esp0}
\]

Uses gdt 92b, TOP_OF_KERNEL_MODE_STACK 159c, and write_tss 197a.

Here’s the prototype of `write_tss` which calls `fill_gdt_entry` to make the GDT entry point to `tss_entry`:

\[
\text{static void write_tss(int num, word ss0, void *esp0)};
\]

Regular GDT entries store a base address and a limit to perform the address transformation from a logical to a linear address (which is then further translated by the paging mechanism). The TSS has a different purpose, but still gets stored in the same table. Here the base address is recycled so that it holds the address of our `tss_entry` structure, and the limit field stores the size of the structure, minus 1. The GDT entry type is 0, and the required access value is \(0xe9 = 11101001\) b.

As a reminder, Figure 6.8 shows the format of a regular segment descriptor at the top (this is the same as Figure 4.2); below, you see the slightly modified format of the TSS descriptor [Int11, p. 7–7]. B (bit 9 of the third word) can be 0 or 1, and we set it to 0, the value is only relevant when using several TSS structures.

\[
\begin{array}{ccccccccc}
\hline
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \\
\hline
\text{Base: 31–24} & \text{Gr Sz 0 0} & \text{Limit: 19–16} \\
\text{P DPL 1 Type A} & \text{Base: 23–16} \\
& \text{Base: 15–0} \\
& \text{Limit: 15–0} \\
\hline
\text{Base: 31–24} & \text{Gr 0 0 0} & \text{Limit: 19–16} \\
\text{P DPL 0 1 0 B 1} & \text{Base: 23–16} \\
& \text{Base: 15–0} \\
& \text{Limit: 15–0} \\
\hline
\end{array}
\]

Figure 6.8: Segment descriptor (top) vs. TSS descriptor (bottom).

When we call `fill_gdt_entry`, we have to set the bits 7–4 of the fourth word to \(0000\) b (in the last argument `gran` of the function call) and the bits 15–8 of the third word to \(11101001\) b (in the second last argument, `access`). This is the interpretation of the access bitmap:

- \(11101001\) b for the TSS descriptor (present; ring 3; fixed-0; TSS type (1001)).
With this information, we can implement `write_tss197a`:

```c
static void write_tss (int num, word ss0, void *esp0) {
    fill_gdt_entry (num, (uint) &tss_entry, sizeof (tss_entry) - 1,
                   0b11101001, 0b000000); // write TSS entry to GDT
    memset (&tss_entry, 0, sizeof (tss_entry)); // fill TSS with zeros
    tss_entry.ss0 = ss0; // kernel stack segment
    tss_entry.esp0 = (memaddress)esp0; // kernel stack pointer
}
```

Defines:
- `write_tss`, used in chunks 196, 197b, and 280a.
- `fill_gdt_entry`, `memaddress`, `memset`, and `tss_entry`.

Thus, all five calls of `fill_gdt_entry109c` together look like this:

```c
fill_gdt_entry (0, 0, 0, 0, 0); // null descriptor
fill_gdt_entry (1, 0, 0xFFFFFFFF, 0b10011010, 0b1100); // kernel, code
fill_gdt_entry (2, 0, 0xFFFFFFFF, 0b10010010, 0b1100); // kernel, data
fill_gdt_entry (3, 0, 0xFFFFFFFF, 0b11111010, 0b1100); // user, code
fill_gdt_entry (4, 0, 0xFFFFFFFF, 0b11110010, 0b1100); // user, data
```

Finally, we add the code for loading the task register `TR` to the assembler file: the index of the TSS in the GDT is 5, so the proper value to load is 5 × 8 = 40 = 0x28. We have to write the requested privilege level (RPL) into the two lowest bits:

```
0x28 | 0x03 = 0x0b
```

```asm
section .text

global tss_flush

tss_flush: mov ax, 0x28 | 0x03
    ltr ax // load the task register
    ret
```

Defines:
- `tss_flush`, used in chunks 110a, 197d, and 280a.

As always we need to tell the C compiler that the assembler function `tss_flush197c` exists elsewhere:

```c
extern void tss_flush ()
```

Lastly, we present the `cpu_usermode198` routine which performs the switch from kernel mode (ring 0) to user mode (ring 3):

```asm
extern void cpu_usermode (memaddress address, memaddress stack); // assembler
```
It prepares a stack that will create the first user mode process when executing `iret`. Since `iret` performs a change of privilege level (from ring 0 to ring 3), it will pop the following:

- instruction pointer EIP
- code segment selector CS
- EFLAGS register
- stack pointer ESP
- stack segment selector SS

The other segment selectors (DS, ES, FS and GS) can be set via `mov` instructions. When we enter the assembler function, there are three values on the stack:

- the return address in `[esp]` (in Assembler syntax, but note: we will never return to that address),
- the first argument in `[esp + 4]`,
- the second argument in `[esp + 8]`.

But each time we push data on the stack, the offsets will change. To make the following code more readable we will start with saving the current `ESP` value in `EBP`—that register will then point to the same address even while we push and pop data.

```
global cpu_usermode

cpu_usermode: cli ; disable interrupts
    mov ebp, esp ; remember current stack address
    mov ax, 0x20 | 0x03 ; code selector 0x20 | RPL3: 0x03
                   ; RPL = requested protection level
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov eax, esp
    push 0x20 | 0x03 ; stack address is 2nd argument
    push eax
    pushf ; EFLAGS
    pop eax ; trick: reenable interrupts when doing iret
    or eax, 0x200
    push eax
    push 0x18 | 0x03 ; code selector 0x18 | RPL3: 0x03
    mov eax, [ebp + 4] ; return address (1st argument) for iret
    push eax
    iret
```

Defines:
- `cpu_usermode`, used in chunk 192d.

(There are many ways to implement this switch to user mode; the version that you see here originates from an osdev.org forum post by Jens Nyberg [Nyb11]. We added
comments and made some changes which make it easier to understand the code. The important point is to set up the stack properly for the final iret instruction.)

We’re using a trick to have interrupts automatically enabled when we execute iret: One of the values on the stack is the EFLAGS register which contains the interrupt enable flag (IF) in bit 9. We cannot directly set that bit in EFLAGS, but we can modify the stack. The sequence pop eax; or eax, 0x200; push eax pops the EFLAGS (which was just pushed in the previous pushf instruction) from the stack, sets bit 9 (2^9 = 512 = 0x200) and pushes the modified value onto the stack.

6.4 System Calls

When the operating system is in kernel mode, it has access to all its internal data and code: it may call any kernel function and, for example, read sectors from a disk or change hardware settings. Processes on the other hand cannot do the same: even though Unix maps the kernel memory in all address spaces, processes cannot access it because the protection bits in the page tables define that this memory area may only be used when the system runs in ring 0—and processes run in ring 3 (user mode).

Even if a process was allowed to call kernel functions (by setting up the page tables differently) that would not help much since privileged machine instructions such as in and out (for talking to hardware devices) cannot be executed in ring 3.

All operating systems provide system calls as a way to access these needed kernel functions: on Unix they allow a controlled switch from user mode to kernel mode via the int instruction which switches to ring 0 and executes a pre-defined interrupt handler. That handler finds out which system call the process wants to execute (by looking at the system call number that must be stored in the EAX register) and then proceeds by calling a system call handler function.

While we implement system calls, we will also create functions for the standard library that user mode programs must link in order to conveniently talk to the operating system via functions such as fork, open, read, etc.

There are several ways to implement system calls. Let’s first look at the way system calls can be called from user space. On 32-bit Intel CPUs, Linux does it via software interrupt 0x80 with arguments in registers:

```assembly
_start: ; tell linker entry point
    mov edx, len ; message length
    mov ecx, msg ; message to write
    mov ebx, 1 ; file descriptor (stdout)
    mov eax, 4 ; system call number (sys_write)
    int 0x80 ; software interrupt 0x80

    mov eax, 1 ; system call number (sys_exit)
    int 0x80 ; software interrupt 0x80

section .data
msg    db 'Hello, world!',0xa ; the string to be printed
len    equ $ - msg ; length of the string
```

(example for system calls in linux 199)
(This example was taken from http://asm.sourceforge.net/intro/hello.html; the comments were modified.)

On a Linux machine you could assemble, link and run this file with

$ nasm -f elf test.asm
$ ld test.o -o test
$ ./test
Hello, world!

In this program EAX always holds the system call number, the other registers (in this example EBX, ECX and EDX are used for arguments. System call 4 is the sys_write syscall.

Other operating systems put arguments on the stack or into specific memory areas. We will stick with the Linux way because it is simple to use registers.

Since adding assembler code to C programs for every system call would be laborious, standard libraries make things simpler for the application developer; this can be done in two steps:

- Supplying a generic syscall function (that takes an arbitrary number of arguments)
  reduces the above code to executing

  ```c
  char *msg = "Hello, world!\n";
  syscall (4, 1, msg, strlen (msg));
  ```

- But that is still unreadable, and also it is not portable because system call numbers are not identical across different Unix versions. Thus, for all standard system calls, some library provides the better known functions (such as write) which allow the above code to be written as

  ```c
  char *msg = "Hello, world!\n";
  write (STDOUT_FILENO, msg, strlen (msg));
  ```

  (with the constant STDOUT_FILENO set to 1).

### 6.4.1 System Calls in Ulix

Ulix provides functions for adding (or modifying) system calls to the system and a generic system call handler. For this purpose, we create a system call table syscall_table that contains pointers to functions, so for example, syscall_table[4] should contain the address of Ulix’s sys_write function. If a system call is not defined, the table entry is a null pointer, so we can initialize the whole table with null bytes:

```c
#define MAX_SYSCALLS 1024 // max syscall number: 1023

void *syscall_table[MAX_SYSCALLS];
```

Defines:
- MAX_SYSCALLS, used in chunks 200 and 201.
Telling Ux what function to execute when a specific system call is made is as simple as writing the address into the proper array entry. Nevertheless, we provide a function

\[
\text{void install_syscall_handler (int syscallno, void *syscall_handler);} \quad 201a
\]

which enters the handler address:

\[
\text{void install_syscall_handler (int syscallno, void *syscall_handler) 
}\quad 201b
\]

\[
\text{if (syscallno ≥ 0 && syscallno < MAX_SYSCALLS) 
syscall_table[syscallno] = syscall_handler;} \quad 201c
\]

Defines:
install_syscall_handler, used in chunks 173d, 201c, 206f, 213e, 217c, 220–22, 224, 235c, 259a, 260c, 282d, and 612a.

Uses MAX_SYSCALLS 200a, syscall_handler 201d, and syscall_table 200b.

So if we have already defined a function sys_write and declared the system call number __NR_write 204c, we could activate the write system call by calling

\[
\text{install_syscall_handler (__NR_write, sys_write);} \quad 201c
\]

The actual system call handler simply checks if there is a handler for the given system call number and (if so) calls it:

\[
\text{void syscall_handler (context_t *r) 
}\quad 201d
\]

\[
\text{if (handler != 0) handler (r);} \quad 201e
\]

Defines:
syscall_handler, used in chunks 201 and 202c.

Uses __NR_get_errno 206e, context_t 142a, MAX_SYSCALLS 200a, printf 601a, set_errno 206b, and syscall_table 200b.

The set_errno 206b function sets the error field of the current TCB and can be used by system call handlers to return an error code (see Section 6.4.3). We will later add system call handlers to a special code chunk named ⟨syscall functions 174b⟩ and put their prototypes in ⟨syscall prototypes 173b⟩.
We add a handler for interrupt 0x80 which looks just like our regular interrupt handlers for hardware-generated interrupts (and also like the fault handlers). The difference is that in this case we call neither irq_handler nor fault_handler, but our new C function syscall_handler. Apart from that we perform the same preparation as in the assembler code which you’ve already seen: We store the context in the proper order on the stack so that syscall_handler which takes a context_t *r as argument can evaluate and possibly change them.

In order to have the system jump to the syscall_handler function, we need to register its address in the interrupt descriptor table (just like we did with the interrupt and fault handlers):

```assembly
extern void syscallh();
```

Uses syscall_handler.

(In case you have forgotten it: \texttt{(push registers onto the stack \ref {chunk 202})} pushes the general purpose registers as well as DS, ES, FS, GS and ESP onto the stack while \texttt{(pop registers from the stack \ref {chunk 143})} pops them back in reverse order. We used this code when we introduced the interrupt handlers.)

In order to have the system jump to the syscallh function, we need to register its address in the interrupt descriptor table (just like we did with the interrupt and fault handlers):

```assembly
fill_idt_entry (128,
    (unsigned int)syscallh,
    0x08,
    0b1110, // flags: 1 (present), 11 (DPL 3), 0
    0b1110); // type: 1110 (32 bit interrupt gate)
```

Uses fill_idt_entry and syscallh.
Note that we create an interrupt gate like in \langle install the interrupt handlers 139b \rangle (p. 139) and \langle install the fault handlers 148b \rangle (p. 148) and not a trap gate, so interrupts will be off when we enter a system call handler. For an interruptible kernel version of Ulix (see the discussion in Chapter 11.6) we would use a trap gate so that interrupts remain enabled.

### 6.4.2 Making System Calls

Actually making a system call works just like in the Linux example we’ve shown earlier:

- load the system call number in EAX,
- load arguments for the syscall in the next registers (EBX, ECX, ...) and
- execute int 0x80.

The return value of the system call can then be read from EAX. The following functions

\langle ulixlib function prototypes 174c \rangle \equiv

\begin{align*}
\text{inline int syscall4} & (\text{int eax}, \text{int ebx}, \text{int ecx}, \text{int edx}) \\
\text{inline int syscall3} & (\text{int eax}, \text{int ebx}, \text{int ecx}) \\
\text{inline int syscall2} & (\text{int eax}, \text{int ebx}) \\
\text{inline int syscall1} & (\text{int eax})
\end{align*}

standardize this process. We do not need them in the kernel, but the user mode library uses them to provide standard functions such as open, read, write, or fork:

\langle ulixlib function implementations 174d \rangle \equiv

\begin{align*}
\text{inline int syscall4} & (\text{int eax}, \text{int ebx}, \text{int ecx}, \text{int edx}) \\
\text{inline int syscall3} & (\text{int eax}, \text{int ebx}, \text{int ecx}) \\
\text{inline int syscall2} & (\text{int eax}, \text{int ebx})
\end{align*}

Defines:
syscall4, used in chunks 203a, 220d, 429b, and 591b.

The \texttt{asm} statement loads the EAX ("a"), EBX ("b"), ECX ("c") and EDX ("d") registers with the supplied values (eax, ebx, ecx, edx), then executes the instruction (int $0x80) and finally writes back the contents of EAX ("a"), which may have been modified, to result. For more information about this syntax see Appendix B.

The other functions work identically, just with less parameters which are stored in less registers:

\langle ulixlib function implementations 174d \rangle \equiv

\begin{align*}
\text{inline int syscall3} & (\text{int eax}, \text{int ebx}, \text{int ecx}) \\
\text{inline int syscall2} & (\text{int eax}, \text{int ebx})
\end{align*}

\begin{align*}
\text{asm} ( "\text{int } 0x80" : "\text{a}" \text{ (result)} : "\text{a}" \text{ (eax)}, "\text{b}" \text{ (ebx)}, "\text{c}" \text{ (ecx)}); \\
\text{return result;}
\end{align*}
inline int syscall1 (int eax) {
    int result;
    asm ("int $0x80" : "=a" (result) : "a" (eax));
    return result;
}

Defines:
syscall1, used in chunks 207b, 227g, 221f, 223b, 260e, 310e, and 513d.
syscall2, used in chunks 174d, 208a, 218a, 259c, 281a, 328g, 333c, 373e, 429b, 434c, 493f, 584c, and 587b.
syscall3, used in chunks 224f, 235e, 331d, 373e, 429b, 434c, 568b, 584c, and 591b.

System calls differ in the number of arguments. Since C provides no internal commands for accessing CPU registers and issuing int calls, we need inline assembler code.

As an example look at the write function which has the prototype

\[
\begin{align*}
\text{int write (int fd, const void *buf, int nbyte);}
\end{align*}
\]

It takes three arguments, thus an implementation in a user mode library would look like this:

\[
\begin{align*}
\text{int write (int fd, const void *buf, int nbyte) }
\quad & \text{return syscall4 (\_NR_write, fd, (int)buf, nbyte);} \\
\end{align*}
\]

For increased Linux compatibility we will use the same system call numbers as Linux does—at least for those calls that Unix does also provide.

The following definitions were taken from the 32-bit Linux¹ file /usr/include/i386-linux-gnu/asm/unistd_32.h:

\[
\begin{align*}
\text{\#define __NR_exit} & \quad 1 \\
\text{\#define __NR_fork} & \quad 2 \\
\text{\#define __NR_read} & \quad 3 \\
\text{\#define __NR_write} & \quad 4 \\
\text{\#define __NR_open} & \quad 5 \\
\text{\#define __NR_close} & \quad 6 \\
\text{\#define __NR_waitpid} & \quad 7 \\
\text{\#define __NR_link} & \quad 9 \\
\text{\#define __NR_unlink} & \quad 10 \\
\text{\#define __NR_execve} & \quad 11 \\
\text{\#define __NR_chdir} & \quad 12 \\
\text{\#define __NR_chmod} & \quad 15 \\
\text{\#define __NR_lseek} & \quad 19 \\
\text{\#define __NR_getpid} & \quad 20 \\
\text{\#define __NR_brk} & \quad 45 \\
\text{\#define __NR_signal} & \quad 48 \\
\text{\#define __NR_dup2} & \quad 63 \\
\text{\#define __NR_getppid} & \quad 64 \\
\text{\#define __NR_symlink} & \quad 83 \\
\text{\#define __NR_readlink} & \quad 85 \\
\text{\#define __NR_readdir} & \quad 89 \\
\text{\#define __NR_truncate} & \quad 92 \\
\text{\#define __NR_ftruncate} & \quad 93 \\
\text{\#define __NR_stat} & \quad 106 \\
\text{\#define __NR_chown} & \quad 182 \\
\text{\#define __NR_getcwd} & \quad 183 \\
\text{\#define __NR_setreuid32} & \quad 203 \\
\text{\#define __NR_setregid32} & \quad 204 \\
\text{\#define __NR_setuid32} & \quad 213 \\
\text{\#define __NR_setgid32} & \quad 214 \\
\end{align*}
\]

Defines:

\begin{itemize}
\item \_NR_brk, used in chunks 173d and 174d.
\item \_NR_chdir, used in chunk 434.
\item \_NR_chmod, used in chunks 590c and 591b.
\item \_NR_chown, used in chunks 590c and 591b.
\item \_NR_close, used in chunks 428a and 429b.
\end{itemize}

As we already mentioned, system calls return arguments by storing the value in EAX.

Now that you have seen how system calls are implemented, you might want to turn back to Chapter 6.1.5 (specifically: to the implementation of the sbrk system call and library function on page 173) because we have already used the system call interface in that code and promised you a reminder once you'd get here.

### 6.4.3 Handling Errors with errno

Most system calls can fail: in that case they need to notify the calling process about the cause of the error. Unix systems traditionally use a special global variable named errno for this purpose; the standard behavior is to make the system call return −1 and put a specific (positive) value into errno.

For Ulix we will provide the error code via a system call (and a corresponding user mode library function) called get_errno(). For entering an error code into the process’ TCB structure, we add the system call and function set_errno(). Every user mode application must include the Ulix standard headers which will contain a macro that defines errno as the result of a system call which executes get_errno(). All attempts to read errno will generate that system call which reads the error field of the TCB. We haven’t defined it yet, so here it is:

```c
int error;
```

(In the TCB we use the name error instead of errno so that we can avoid confusion about which is which.)

We also declare a variable startup_errno which will be used in the early phase of the kernel initialization before the first process is started:

```c
int startup_errno = 0;
```

Defines:

- startup_errno, used in chunk 206b.
Inside the kernel the two functions are easy to implement:

\[ \text{function prototypes 45a}\] +≡
\[
\begin{align*}
\text{int get_errno ()}; \\
\text{void set_errno (int err)};
\end{align*}
\]

\[ \text{function implementations 100b}\] +≡
\[
\begin{align*}
\text{int get_errno ()} \\
\{ \\
\quad \text{if (scheduler_is_active) return thread_table[current_task].error; } \\
\quad \text{else return startup_errno;}
\}
\end{align*}
\]
\[
\begin{align*}
\text{void set_errno (int err)} \\
\{ \\
\quad \text{if (scheduler_is_active) thread_table[current_task].error = err; } \\
\quad \text{else startup_errno = err;}
\}
\end{align*}
\]

Defines:
get_errno, used in chunk 206d.
set_errno, used in chunks 201d, 206, 207c, 562b, 565c, 576, 577, and 579c.
Uses current_task 192c, scheduler_is_active 276e, startup_errno 205c, and thread_table 176b.

Now we need to turn these two functions into system calls. The system call handlers simply call the above functions; an argument (for set_errno 206b()) can be found in the EBX register, and we store a return value (for get_errno 206b()) in the EAX register. Both are available via the context_t 142a structure which is provided as an argument to the system call handlers:

\[ \text{syscall prototypes 173b}\] +≡
\[
\begin{align*}
\text{void syscall_get_errno (context_t *r)}; \\
\text{void syscall_set_errno (context_t *r)};
\end{align*}
\]

\[ \text{syscall functions 174b}\] +≡
\[
\begin{align*}
\text{void syscall_get_errno (context_t *r) } \\
\{ \\
\quad \text{eax_return (get_errno ())}; }
\end{align*}
\]
\[
\begin{align*}
\text{void syscall_set_errno (context_t *r) } \\
\{ \\
\quad \text{set_errno ((int)r->ebx);} ;
\}
\end{align*}
\]

Defines:
syscall_get_errno, used in chunk 206f.
syscall_set_errno, used in chunk 206.
Uses context_t 142a, eax_return 174a, get_errno 206b, and set_errno 206b.

Finally we need to register the system calls:

\[ \text{unix system calls 206e}\] ≡
\[
\begin{align*}
\#define __NR_get_errno 501 \\
\#define __NR_set_errno 502
\end{align*}
\]

Defines:
__NR_get_errno, used in chunks 201d, 206f, and 207b.
__NR_set_errno, used in chunk 206f.

\[ \text{initialize syscalls 173d}\] +≡
\[
\begin{align*}
\text{install_syscall_handler (__NR_get_errno, syscall_get_errno)}; \\
\text{install_syscall_handler (__NR_set_errno, syscall_set_errno)};
\end{align*}
\]

Uses __NR_get_errno 206e, __NR_set_errno 206e, install_syscall_handler 201b, syscall_get_errno 206d, and syscall_set_errno 206d.
6.5 Forking a Process

We’ll collect error codes (such as EACCES, which is the code for “permission denied”) in a new \texttt{error constants} chunk:

\[\langle public constants 46a\rangle + \equiv \langle error constants 370a\rangle\]

and we will fill this collection with entries as we go along and opportunities for generating errors arise.

User mode programs can access the error code via the \texttt{errno} macro which just retrieves the value:

\[\langle ulixlib constants 207b\rangle \equiv \]

\[
\text{\#define } \text{errno } (\text{syscall1(\_NR_get_errno)})
\]

Uses \_NR\_get\_errno and syscall1 203c.

Note that most system calls do not set an error value because we wanted to keep the code compact. But it would be easy to change this: After all, the system call handlers do check for errors and simply return \(-1\) when one occurs. By writing a macro

\[\langle possible macro for readable error returns 207c\rangle \equiv \]

\[
\text{\#define err\_return}(\text{retval}, \text{errno}) \text{\{set\_errno}(\text{errno}); \text{return } \text{retval};\}
\]

you could replace the \texttt{return(-1);} lines in the current code with \texttt{err\_return (-1, ECODE);} lines.

6.5 Forking a Process

We’re getting closer to having a multitasking operating system. We only use the function \texttt{start\_program\_from\_disk} for loading the first (initial) process—for everything else we want to implement the standard Unix way of creating new processes: the \texttt{fork}.

Figure 6.9 shows how a process and its fork proceed over time; the depiction resembles a (two-pronged) fork.

![Figure 6.9: When the system forks a process, it creates an almost identical copy.](image)

Here’s an excerpt from the \texttt{fork} manpage on a Debian GNU/Linux 7.1 machine [Lin12a]:
NAME
fork - create a child process

DESCRIPTION
fork() creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following points:

* The child has its own unique process ID, and this PID does not match the ID of any existing process group (setpgid(2)).

* The child's parent process ID is the same as the parent's process ID.

RETURN VALUE
On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

Basicall, when a Unix process calls fork() (and thus enters the fork system call), the operating system creates a duplicate of the currently running process. After a successful fork operation we have two processes which are almost identical. That means:

- Both processes execute the same program (i.e., the same binary is loaded in their lower memory areas).
- Variables and dynamic memory have identical contents, but the memory is duplicated since both processes may make different changes to that memory once they continue running after the fork.
- They also have their own copies of the user mode and kernel mode stack.
- Most process metadata (the contents of the thread control block) are identical as well, with two important exceptions: the new process has its own process ID (and thread ID), and the new process stores the old process’ thread ID in its parent process ID field (ppid).
- After the fork, both processes return from the fork system call and continue execution in the instruction immediately following the system call—so they need a way to find out whether they are the original process (called parent) or the newly forked process (called child). The user mode fork function will return 0 in the child process and the newly created process’ ID in the parent process.

Note that other Unix implementations do not copy the whole process memory—instead they use a technique called copy-on-write that only creates a copy of the page tables and marks them read-only (in both the parent and child process). This means that initially both processes use the same physical memory, but the read-only mode guarantees that no
problems can occur. When a process tries to modify its memory, that will cause a page fault (due to the missing write permissions), and the fault handler will then create a copy of that page so that both processes have their personal copy of the faulting page. This copy (and the original) will have read and write permissions, and the faulting process can repeat its write operation. U/LX copies all of the memory which is less efficient, but allows a simpler implementation.

While we create a new process we will set its state to TSTATE_FORK to show that its creation is still in progress.

Our goal for this section is to implement the function

\[\text{int } \text{u_fork (context_t } \ast r);\]

which will later be called from the fork system call handler (see page 213).

Since we will need a lot of memory copying operations, we declare two macros which let us copy physical memory areas (`phys_memcpy`) and copy page frames (`copy_frame`):

\[\text{#define phys_memcpy(target, source, size) } \ast \text{memcpy(PHYSICAL(target), PHYSICAL(source), size)}\]

\[\text{#define copy_frame(out, in) phys_memcpy(out } << 12, \text{ in } << 12, \text{ PAGE_SIZE)}\]

Defines:
- `phys_memcpy`, used in chunk 211c.
- `copy_frame`, used in chunk 209b.

Uses `memcpy` 596c, `PAGE_SIZE` 112a, and `PHYSICAL` 116a.

So, `phys_memcpy` does the same as `memcpy` but expects its first two arguments to be physical addresses (instead of virtual ones), and `copy_frame` provides a shortcut for copying physical frames since for that task the number of bytes to copy is always `PAGE_SIZE`).

Next comes the definition of `u_fork`. This function will be called when the fork system call is executed. Again, we declare everything that is done in this function as a critical section. If you ask, why there is no `end critical section in kernel` in this chunk, see `u_fork: branch parent and child`.

\[\text{int } \text{u_fork (context_t } \ast r) \{\}

\[\text{\begin{align*}
\text{begin critical section in kernel} & \text{ 380a)} \\
\text{thread_id old_tid = current_task; } \\
\text{thread_id ppid = old_tid; } \\
\text{u_fork: create new address space and TCB 210a} \\
\text{u_fork: fill new TCB 210b} \\
\text{u_fork: create new kernel stack and copy the old one 211a} \\
\text{u_fork: copy user mode memory 211c} \\
\text{u_fork: branch parent and child 212}\}
\]

\[\}

Defines:
- `u_fork`, used in chunks 188d, 209a, and 213d.
Uses `context_t` 142a, `current_task` 192c, and `thread_id` 178a.

Now, here’s the actual implementation. We will present it in several steps and discuss what’s happening.
6.5.1 Reserving Memory and a Fresh TCB

We start by creating a new address space and cloning the current TCB into a free TCB which we first have to search for.

This step is similar to the first step in start_program_from_disk\[^{189}\], except that memory and stack size are not free parameters, but are copied from the parent process:

\[
\begin{aligned}
\text{addr_space_id}_{\text{old}} = \text{current}\_\text{as}; \\
\text{addr_space_id}_{\text{new}} = \text{create_new_address_space}\left(\text{address_spaces}[\text{old}].\text{memend} - \text{address_spaces}[\text{old}].\text{memstart}, \\
\text{address_spaces}[\text{old}].\text{stacksize}\right); \\
\text{if} \left(\text{new}_{\text{as}} = -1\right) \text{return} -1; \quad \text{// error: cannot create address space}
\end{aligned}
\]

\[
\begin{aligned}
\text{thread_id}_{\text{new}} = \text{register_new_tcb}\left(\text{new}_{\text{as}}\right); \\
\text{if} \left(\text{new}_{\text{tid}} = -1\right) \text{return} -1; \quad \text{// error: cannot create TCB entry}
\end{aligned}
\]

Uses addr_space_id 158b, address_spaces 162b, create_new_address_space 163c, current_as 170b, register_new_tcb 188d, TCB 175, and thread_id 178a.

6.5.2 Filling the Child TCB

Basically the child is an almost identical copy of the parent, so we start with copying the parent TCB to the child TCB. However, we need to modify some values, for example the process, thread and parent process ID as well as the link to the address space. We also copy the open file descriptors, but this needs more work than just copying the information in the TCB; we will explain that in the filesystem chapter where we provide the code chunk ⟨u_fork: copy the file descriptors 425a⟩. The new process is set to state TSTATE_FORK\[^{180}\]a; it will only change to TSTATE_READY\[^{180}\]a when the fork operation is complete.

\[
\begin{aligned}
\text{TCB}_{\text{old}} = \&\text{thread_table}[\text{old}_{\text{tid}}]; \quad \text{// prefer to use pointers} \\
\text{TCB}_{\text{new}} = \&\text{thread_table}[\text{new}_{\text{tid}}]; \\
\text{t}_{\text{new}} = \text{t}_{\text{old}}; \quad \text{// copy the complete TCB} \\
\text{t}_{\text{new}}->\text{state} = \text{TSTATE}_{\text{FORK}}; \\
\text{t}_{\text{new}}->\text{tid} = \text{new}_{\text{tid}}; \\
\text{t}_{\text{new}}->\text{addr}_{\text{space}} = \text{new}_{\text{as}}; \\
\text{t}_{\text{new}}->\text{pid} = \text{t}_{\text{new}}->\text{tid}; \quad \text{// new process; pid = tid} \\
\text{t}_{\text{new}}->\text{ppid} = \text{old}_{\text{tid}}; \quad \text{// set parent process ID}
\end{aligned}
\]

\[
\begin{aligned}
\text{// copy current registers to new thread, except EBX (= return value)} \\
\text{t}_{\text{new}}->\text{regs} = *r; \\
\text{// copy current ESP, EBP} \\
\text{asm volatile} \left("\text{mov} \%\text{esp}, \%0" : "=r"\left(\text{t}_{\text{new}}->\text{esp}\right)\right); \quad \text{// get current ESP} \\
\text{asm volatile} \left("\text{mov} \%\text{ebp}, \%0" : "=r"\left(\text{t}_{\text{new}}->\text{ebp}\right)\right); \quad \text{// get current EBP}
\end{aligned}
\]

⟨u_fork: copy the file descriptors 425a⟩ \quad \text{// see filesystem chapter}

Uses t_new 276c, t_old 276c, TCB 175, thread_table 176b, and TSTATE_FORK 180a.
6.5.3 The Child’s Kernel Stack

The child needs a fresh kernel stack, and that also requires a new page table in which we can enter the mappings of the kernel stack pages to physical page frames.

\[
\langle \text{u_fork: create new kernel stack and copy the old one} \rangle \equiv \quad (209c) \quad 211b \quad [211a]
\]

\[
\text{page_table} \ast \text{stackpgtable} = \text{request_new_page}();
\]

\[
\text{address_spaces[\text{new_as}].kstack}_\pi = (\text{memaddress})\text{stackpgtable};
\]

\[
\text{memset}(\text{stackpgtable}, 0, \text{sizeof}(\text{page_table}));
\]

\[
\text{page_directory} \ast \text{tmp_pd} = \text{address_spaces[\text{new_as}].pd};
\]

\[
\text{KMAPD}(\text{&tmp_pd->ptds}[767], \text{mmu}(0, (\text{uint})\text{stackpgtable}));
\]

\[
\text{int i, j; // counters}
\]

\[
\text{for}(i = 0; \ i < \text{KERNEL_STACK_PAGES}; \ i++)
\]

\[
\text{as_map_page_to_frame(\text{new_as}, 0xbffff - i, \text{request_new_frame}());}
\]

We use the \text{phys_memcpy} \quad \text{macro for copying the frames of the parent’s kernel stack to the child’s kernel stack, we get those physical addresses from the \text{mmu_172a} function, using \text{new_as} for the new page table and \text{old_as} for the old table. It is not possible to simply start with an empty stack (like we did when we created the first process) because the child process, once it is fully created, will be in the middle of executing the fork system call, so the stack must be there and have the same contents as in the parent.

\[
\langle \text{u_fork: create new kernel stack and copy the old one} \rangle \equiv \quad (209c) \quad 211a \quad [211b]
\]

\[
// \text{copy the physical frames}
\]

\[
\text{memaddress } \text{base} = \text{TOP_OF_KERNEL_MODE_STACK} - \text{KERNEL_STACK_SIZE};
\]

\[
\text{for}(i = 0; \ i < \text{KERNEL_STACK_PAGES}; \ i++)
\]

\[
\text{phys_memcpy}(\text{mmu(\text{new_as}, base + i*PAGE_SIZE)}, \text{mmu(\text{old_as}, base + i*PAGE_SIZE), PAGE_SIZE });
\]

Note that the frames that we request here (both via \text{request_new_page} \quad \text{for the page table and request_new_frame} \quad \text{for the kernel stack pages) will be released when the process exits.

6.5.4 Copying the Process’ User Mode Memory

Copying the user mode memory means copying the first 3 GByte except the kernel stack which we’ve done already. This requires a nested loop since for each present page directory entry we look at each present page table entry and then make a copy. We have to look at the first 767 page tables (the 768th table holds the entries for the kernel stack).

\[
\langle \text{u_fork: copy user mode memory} \rangle \equiv \quad (209c) \quad 211c \quad [211c]
\]

\[
// \text{clone first 3 GB (minus last directory entry) of address space}
\]

\[
\text{page_directory} \ast \text{old_pd} = \text{address_spaces[\text{old_as}].pd};
\]

\[
\text{page_directory} \ast \text{new_pd} = \text{address_spaces[\text{new_as}].pd};
\]
6 Implementation of Processes

```c
page_table  *old_pt, *new_pt;
for (i = 0; i < 767; i++) {    // only 0..766, not 767 (= kstack)
    if (old_pd->ptds[i].present) {    // page table present?
        // walk through the entries of the page table
        old_pt = (page_table*)PHYSICAL (old_pd->ptds[i].frame_addr << 12);
        new_pt = (page_table*)PHYSICAL (new_pd->ptds[i].frame_addr << 12);
        for (j = 0; j < 1024; j++)
            if (old_pt->pds[j].present)    // page present?
                copy_frame ( new_pt->pds[j].frame_addr, old_pt->pds[j].frame_addr );
    }
};
```

Uses address_spaces 162b, copy_frame 209b, kstack, page_directory 103d, page_table 101b, and PHYSICAL 116a.

6.5.5 A Child Is Born

All the code you have seen so far is only executed in the original (parent) process. But at some point in time there will be both the parent and the child, and the question is where the child shall start execution. We make the branch right here, as the last step in u_fork.

We start with querying the current instruction pointer (EIP) via the get_eip function. This function returns the address of the instruction after the get_eip call (because it retrieves the return address from the stack, and that address is not the address of the call, but of the instruction where the u_fork function continues after returning from get_eip). That next line of code is the first line that we want to be executed by both processes, thus we store the value in the eip field of the new process’ TCB. That’s the whole trick behind getting the new process to start running at the correct instruction.

The rest is administrative work: In the parent process we add the new process to the ready queue, re-enable the interrupts and return the new process’ thread ID. In the child process we simply return 0. We can check whether we’re in the parent or child by comparing current_task with the ppid variable: The latter is identical in both processes, but the comparison only evaluates to true in the parent process.

[U_fork: branch parent and child] [212]

memaddress eip = get_eip();         // get current EIP
// new process begins to live right here!
if (current_task == ppid) {
    // parent tasks
    t_new->eip = eip;
    add_to_ready_queue (new_tid);
    (end critical section in kernel) // must be done in parent
    return new_tid;                 // in parent, fork returns child's PID
} else {
    // child tasks
    return 0;                       // in child, fork returns 0
}

Uses add_to_ready_queue 184b, current_task 192c, get_eip 213b, memaddress 46c, and t_new 276c.
Since
\[ \text{function prototypes} \quad 45a \] +\[ \text{extern memaddress get_eip} \];
\[ \text{performs its trick by looking at the stack, it must be implemented in the assembler file.} \]
We simply pop the return address from the stack (storing it in \( EAX \)) and push it back so that the stack is as before. The contents of \( EAX \) are always used as functions’ return values, so we’re done:
\[ \text{start.asm} \quad 87 \] +\[ \text{get_eip}: \quad \text{pop eax} \quad ; \text{top of stack contains return address} \]
\[ \text{push eax} \quad ; \text{write it back} \]
\[ \text{ret} \]
Defines:
\[ \text{get_eip, used in chunk 212.} \]

6.5.6 The fork System Call

We can now add the fork system call: As usual, syscall_fork\[ 213d \] calls u_fork\[ 209c \] and stores the return value in \( EAX \) using eax_return\[ 209a \]:
\[ \text{syscall prototypes} \quad 173b \] +\[ \text{void syscall_fork (context_t *r);} \]
\[ \text{syscall functions} \quad 174b \] +\[ \text{void syscall_fork (context_t *r)} \{ eax_return ((unsigned int) u_fork (r)); \} \]
Defines:
\[ \text{syscall_fork, used in chunk 213.} \]
Uses context_t\[ 142a \], eax_return\[ 174a \], and u_fork\[ 209c \].

We add the system call handler to the list:
\[ \text{initialize syscalls} \quad 173d \] +\[ \text{install syscall_handler (__NR_fork, syscall_fork);} \]
Uses __NR_fork\[ 204c \], install syscall_handler\[ 201b \], and syscall_fork\[ 213d \].

And here is the user mode library function:
\[ \text{ulixlib function prototypes} \quad 174c \] +\[ \text{int fork ()}; \]
\[ \text{ulixlib function implementations} \quad 174d \] +\[ \text{int fork ()} \{ \text{return syscall1 (__NR_fork);} \} \]
Defines:
\[ \text{fork, used in chunks 213f and 214.} \]
Uses __NR_fork\[ 204c \] and syscall1\[ 203c \].
6.5.7 Testing fork

The following test program creates a process tree by calling fork four times:

```
#include "../ulixlib.h"

int main ()
{
    printf ("Press Return to end.
");
    int f1 = fork (); int f2 = fork (); int f3 = fork (); int f4 = fork ();
    int pid = getpid (); int ppid = getppid (); int tid = gettid ();
    printf ("[%2d]: pid = %2d, tid = %2d, ppid = %2d, forkrets = [%2d %2d %2d %2d]\n",
            pid, pid, tid, ppid, f1, f2, f3, f4);

    long long int j; for (j = 0; j < 9999999ul; j++) ; // wait
    if (f1!=0 && f2!=0 && f3!=0 && f4!=0) {
        char s[80]; ureadline ((char*)s, 79, false);
    }
    exit (0);
}
```

Uses exit, fork, getpid, getppid, gettid, main, printf, and ureadline.

When running it, we get the following output. Figure 6.10 shows the process tree that is created by the program.

```
esser@ulix[8]:/home/esser$ fork2
Press Return to end.
[11]: pid = 11, tid = 11, ppid = 10, forkrets = [ 0 13 14 15]
[13]: pid = 13, tid = 13, ppid = 11, forkrets = [ 0 0 17 18]
[14]: pid = 14, tid = 14, ppid = 11, forkrets = [ 0 13 0 19]
[15]: pid = 15, tid = 15, ppid = 11, forkrets = [ 0 13 14 0]
[16]: pid = 16, tid = 16, ppid = 12, forkrets = [11 0 0 20]
[17]: pid = 17, tid = 17, ppid = 13, forkrets = [ 0 0 0 21]
[18]: pid = 18, tid = 18, ppid = 13, forkrets = [ 0 0 17 0]
[19]: pid = 19, tid = 19, ppid = 14, forkrets = [ 0 13 0 0]
[20]: pid = 20, tid = 20, ppid = 16, forkrets = [11 0 0 0]
[21]: pid = 21, tid = 21, ppid = 17, forkrets = [ 0 0 0 0]
[10]: pid = 10, tid = 10, ppid = 8, forkrets = [11 12 22 23]
[12]: pid = 12, tid = 12, ppid = 10, forkrets = [11 0 16 24]
[22]: pid = 22, tid = 22, ppid = 10, forkrets = [11 12 0 25]
[23]: pid = 23, tid = 23, ppid = 10, forkrets = [11 12 22 0]
[24]: pid = 24, tid = 24, ppid = 12, forkrets = [11 0 16 0]
[25]: pid = 25, tid = 25, ppid = 22, forkrets = [11 12 0 0]
esser@ulix[4]:/home/esser$
```

You will see similar code when you reach Chapter 7 where we discuss the creation of threads. Some operating systems use one kernel function that can create both new processes and threads, for example the Linux kernel has a clone function that handles both types. We decided against that approach because it makes the function more complex as it often has to check what type of task it is creating.
6.6 Exiting from a Process

In standard Unix implementations there are five ways to end the life of a process:

- The process calls `exit()` explicitly which makes it terminate immediately.
- The process executes `return` in the `main()` function or it reaches the end of that function. That will lead to an implicit call of `exit()` with the same result.
- The process receives a signal (from another process or from the kernel, see Chapter 14). If it has not installed a handler for this signal (or the signal cannot be intercepted), this causes the termination of the process (it aborts). In that case it cannot provide an exit value, instead there’s an error code.
- Some kind of error occurs that causes a signal to be sent to the process (by the kernel). That is a special case of the one above.
- The process calls `abort()` which makes it send a SIGABRT signal to itself. The result is the same as when that signal is sent by a different process or by the kernel.

In all of these cases the parent process can read the `exit status` and find out whether the process terminated normally or was aborted. The argument to `exit()` or `return` can also be used to tell the parent process whether the process finished successfully; traditionally an exit code of 0 means success, and any other value represents a problem that caused the process to (autonomously) terminate. It is not standard practice to use the exit code as some kind of return value; mainly because the exit code is typically restricted to the integer range of 0–255.

Figure 6.10: Calling `fork` four times creates this tree structure.
6.6.1 The exit System Call

UNIX provides an exit system call which terminates the process and stores the exit code (which is the single argument and available via EBX) in the TCB of the process.

```c
void syscall_exit (context_t *r);
```

It starts with disabling the interrupts and closing all open files of the process (this will only make sense after you’ve read the chapter about filesystems). Then it modifies the thread table: It removes the process from the ready queue and sets the process state to TSTATE_EXIT. We cannot get rid of the TCB entry right now because the parent process must get a chance to read the exit code that we store in the exitcode field of the leaving process’ TCB.

Finally, it wakes a waiting parent process, asks for destroyal of the address space (not all of that can happen at once, as we’ve already seen in Section 6.1.3), and updates the TCBs of any children it might have:

```c
void syscall_exit (context_t *r) {
    // exit code is in ebx register:
    // access the thread table
    // close open files
    thread_id pid = thread_table[current_task].pid;
    int gfd;
    for (int pfd = 0; pfd < MAX_PFD; pfd++) {
        if ((gfd = thread_table[pid].files[pfd]) != -1) u_close (gfd);
    }

    // modify thread table
    thread_table[current_task].exitcode = r->ebx; // store exit code
    thread_table[current_task].state = TSTATE_EXIT; // mark process as finished
    remove_from_ready_queue (current_task); // remove it from ready queue
    wake_waiting_parent_process (current_task); // wake parent
    destroy_address_space (current_as); // return the memory
    // remove childrens link to parent

    // finally: call scheduler to pick a different task
    scheduler (r, SCHED_SRC_RESIGN);
};
```

Defines:
- syscall_exit, used in chunks 152b, 166c, 216a, 217c, and 260a.
Uses context_t 142a, current_as 170b, current_task 192c, destroy_address_space 166c, MAX_PFD 424b, remove_from_ready_queue 184c, SCHED_SRC_RESIGN 343a, scheduler 276d, thread_id 178a, thread_table 176b, TSTATE_EXIT 180a, u_close 418a, and wake_waiting_parent_process 217a.

We implement a function

```c
void wake_waiting_parent_process (int pid);
```
that checks whether the parent process is waiting for the current process to finish; we do not provide the code as a code chunk because it will also be used by the \texttt{u\_kill}\textsubscript{562b} function which can terminate arbitrary processes.

If the parent is waiting, then it will be on the \texttt{waitpid\_queue}\textsubscript{218b} queue. We can then transfer it to the ready queue by calling \texttt{deblock}\textsubscript{186b}. If it is not waiting, we turn this process into a \texttt{zombie}: That means that the process will remain in the thread table. That way we give the parent process a chance to read the exit code, since once the TCB is gone, so is the exit code.

\begin{verbatim}
void wake_waiting_parent_process (int pid) {
    // check if we need to wake up parent process
    int ppid = thread_table[pid].ppid;
    if ((thread_table[ppid].state == TSTATE_WAITFOR) &&
        (thread_table[ppid].waitfor == pid)) {
        // wake up parent process
        deblock (ppid, &waitpid_queue);
        thread_table[pid].state = TSTATE_EXIT;
    } else {
        // parent is not waiting, make this process a zombie
        thread_table[pid].state = TSTATE_ZOMBIE;
    }
}
\end{verbatim}

Defines:
\begin{itemize}
\item \texttt{wake_waiting_parent_process}, used in chunks 216 and 564a.
\end{itemize}

Uses \texttt{deblock}\textsubscript{186b}, \texttt{thread\_table}\textsubscript{176b}, \texttt{TSTATE\_EXIT}\textsubscript{180a}, \texttt{TSTATE\_WAITFOR}\textsubscript{180a}, \texttt{TSTATE\_ZOMBIE}\textsubscript{180a}, and \texttt{waitpid\_queue}\textsubscript{218b}.

We will remove zombie processes in the scheduler: It checks whether a zombie’s parent has disappeared and (if so) deletes the zombie’s TCB. You can see the code in the chunk \texttt{scheduler: check for zombies 281}.

We also need to inform children processes that their parent is gone. In that case we set their parent process ID \texttt{PPID} to 1 (the ID of the \texttt{init} process which becomes the \texttt{idle} process).

\begin{verbatim}
for (int pid = 0; pid < MAX_THREADS; pid++)
    if (thread_table[pid].ppid == current_task)
        thread_table[pid].ppid = 1;  // set parent to idle process
\end{verbatim}

Uses \texttt{current\_task}\textsubscript{192c}, \texttt{MAX\_THREADES}\textsubscript{176a}, and \texttt{thread\_table}\textsubscript{176b}.

As usual, we need to enter the system call handler in the table and provide a user mode \texttt{exit}\textsubscript{218a} function that makes the right system call:

\begin{verbatim}
install_syscall_handler (__NR_exit, syscall_exit);
\end{verbatim}

Uses \texttt{__NR\_exit}\textsubscript{204c}, \texttt{install\_syscall\_handler}\textsubscript{201b}, and \texttt{syscall\_exit}\textsubscript{216b}.

\begin{verbatim}
void exit (int exitcode);
\end{verbatim}
6 Implementation of Processes

6.6.2 The waitpid System Call

Often a process wants to wait for the completion of a child process, a typical example is a shell which starts an external program by `fork`ing, executing the program inside the child process and waiting in the parent process.

Here we implement the `waitpid` system call which waits for completion of a given child, the standard definition, taken from the Linux man pages, is the following:

```c
pid_t waitpid (pid_t pid, int *status, int options);
```

In that prototype

- `pid` is the process ID of a child process (`waitpid`) cannot be used to wait for termination of arbitrary, non-child processes,
- `*status` is the address of a status value which will be used to store the exit code of the child process (or an error value if the child was aborted),
- `options` can be used to modify `waitpid`'s behavior; our implementation will ignore any given options.

We need a blocked queue for processes that called `waitpid` since they must not be picked by the scheduler.

```
[218b] ⟨global variables 92b⟩+≡
    blocked_queue waitpid_queue;
```

```
[218c] ⟨initialize system 45b⟩+≡
    initialize_blocked_queue (&waitpid_queue);
```

Several things must be implemented for `waitpid` to work properly:

- We need the system call handler which moves the current (calling) process from the ready queue to the new `waitpid_queue` and calls `resign` (so that the scheduler picks a new process—the `resign` code will be shown right after `waitpid`).
- When a process exits, it must store the exit argument in the thread control block—this TCB must remain intact until the parent process has had a chance to look up the value. (We’ve already shown you that part.)
- If the parent of an exiting process is in the `waitpid_queue`, we move it back to the ready queue. (That is handled by `wake_waiting_parent_process`, see above.)
Once the parent process is picked by the scheduler, it will continue its execution of waitpid and has to read the child’s exit code. After that waitpid, it can delete the TCB entry.

As long as the parent process could not be reactivated, the child’s TCB will remain intact. Note that it is not necessary for the parent process to actually look at the exitcode.

First we add exitcode and waitfor entries to the TCB structure:

```c
int exitcode;
int waitfor;  // pid of the child that this process waits for
```

The system call handler

```c
void syscall_waitpid (context_t *r);
```

works as follows:

```c
void syscall_waitpid (context_t *r) {
    // ebx: pid of child to wait for
    // ecx: pointer to status
    // edx: options (ignored)
    (begin critical section in kernel 380a)
    int chpid = r->ebx;  // child we shall wait for
    // check errors
    if (chpid < 1 || chpid ≥ MAX_THREADS || thread_table[chpid].state == 0) {
        (end critical section in kernel 380b)
        eax_return (-1);  // error
    }
    if (!thread_table[chpid].used) {
        (end critical section in kernel 380b)
        eax_return (-1);  // no such process
    }
    if (thread_table[chpid].ppid != current_task) {
        (end critical section in kernel 380b)
        eax_return (-1);  // not a child of mine
    }

    int *status = (int*)r->ecx;  // address for the status
    thread_table[current_task].waitfor = chpid;
    block (&waitpid_queue, TSTATE_WAITFOR);
    (end critical section in kernel 380b)
    syscall_resign (r);  // here we resign
```

Defines:
syscall_waitpid, used in chunks 219b and 220b.
Uses context_t 142a, current_task 192c, eax_return 174a, MAX_THREADS 176a, syscall_resign 221a, thread_table 176b, TSTATE_WAITFOR 180a, and waitpid_queue 218b.
Calling block only moves the process to a different queue, but it does not stop its execution; for that purpose we must also call syscall_resign.

When we return from syscall_resign, the child must have finished. Unblocking this process happens in syscall_exit(), here we expect to be woken up automatically.

The return value of waitpid is the process ID of the terminated child (chpid) or −1 in case of an error. Since syscall_exit() has updated the exitcode field of the child’s TCB, we can just read it.

```c
#define __NR_waitpid 204

install_syscall_handler (__NR_waitpid, syscall_waitpid);
```

As usual, we register the new system call:

```c
#define __NR_waitpid 204

install_syscall_handler (__NR_waitpid, syscall_waitpid);
```

Here is the user mode function:

```c
int waitpid (int pid, int *status, int options)
{
    return syscall (__NR_waitpid, pid, (uint)status, options);
}
```

6.6.3 Giving Up the CPU: The resign System Call

The resign system call allows a process to give up the CPU, so that the scheduler picks another process immediately.

We call the scheduler with a special argument SCHED_SRC_RESIGN which tells it that it was called from syscall_resign because we want to be able to detect how it was called. This will be explained in more detail in Chapter 8.
6.7 Information about Processes

In this section we implement a few library functions which enable processes and threads to query their process and thread IDs, the parent process ID and information about the overall list of tasks (so that we can write a user mode `ps` program).
6.7.1 The gettid, getpid and getppid System Calls

Each TCB contains two IDs which describe a task: a thread ID tid (which is what the global variable current_task uses to point to the currently executing thread and which is identical to the index into the thread table) and also a process ID pid. Until now, thread and process IDs have always been identical, but when we introduce threads (as parts of a process) in the next chapter, we will arrive at a situation where these IDs differ. So we will provide three functions that retrieve the thread and process IDs (and also the parent process ID):

\[
\begin{align*}
void syscall_gettid (context_t *r) &; // get thread ID \\
void syscall_getpid (context_t *r) &; // get process ID \\
void syscall_getppid (context_t *r) &; // get parent process ID \\
\end{align*}
\]

Getting the thread ID is simple, because the executing thread always has the thread ID stored in current_task. For the process ID and the the parent process ID we need to access the TCB and fetch its pid or ppid entries, respectively.

\[
\begin{align*}
void syscall_gettid (context_t *r) &; \\
void syscall_getpid (context_t *r) &; \\
void syscall_getppid (context_t *r) &; \\
\end{align*}
\]

Defines:
- syscall_getpid, used in chunk 222b.
- syscall_getppid, used in chunk 222b.

Uses context_t, current_pid, current_ppid, current_task, eax_return, and syscall_gettid.

They use these two macros:

\[
\begin{align*}
#define current_pid (thread_table[current_task].pid) \\
#define current_ppid (thread_table[current_task].ppid) \\
\end{align*}
\]

Defines:
- current_pid, used in chunk 222b.
- current_ppid, used in chunk 222b.

Uses current_task and thread_table.

The system call numbers __NR_getpid and __NR_getppid have been defined earlier, they are standard numbers that you can also find on Linux systems. For gettid we need to define a number since that is no standard system call.

\[
\begin{align*}
#define __NR_gettid 21 \\
\end{align*}
\]

Uses __NR_gettid.

\[
\begin{align*}
install_syscall_handler (__NR_gettid, syscall_gettid); \\
install_syscall_handler (__NR_getpid, syscall_getpid); \\
install_syscall_handler (__NR_getppid, syscall_getppid); \\
\end{align*}
\]

Uses __NR_getpid, __NR_getppid, __NR_gettid, install_syscall_handler, syscall_gettid, syscall_getpid, syscall_getppid, and syscall_gettid.
The user mode getpid, getppid and gettid functions

\[
\text{\textit{ulixlib function prototypes 174c}} \quad \text{\texttt{\textbackslash begin{verbatim} int gettid (); \texttt{\textbackslash end{verbatim} int getpid (); \texttt{\textbackslash end{verbatim} int getppid ();\texttt{\textbackslash end{verbatim}}}}}
\]

simply make the appropriate system calls:

\[
\text{\textit{ulixlib function implementations 174d}} \quad \text{\texttt{\textbackslash begin{verbatim} int gettid () \{ return syscall1 (__NR_gettid); \} \texttt{\textbackslash end{verbatim} int getpid () \{ return syscall1 (__NR_getpid); \} \texttt{\textbackslash end{verbatim} int getppid () \{ return syscall1 (__NR_getppid); \} \texttt{\textbackslash end{verbatim}}}}}
\]

Defines:
- getpid, used in chunks 214, 311b, 513e, and 568b.
- getppid, used in chunk 214.
- gettid, used in chunks 214 and 223a.

Uses __NR_getpid 204c, __NR_getppid 204c, __NR_gettid, and syscall1 203c.

Note that we have not implemented corresponding u_getpid, u_gettid and u_getppid functions in the kernel as we normally do; querying the current thread’s IDs is too simple to justify extra functions for that purpose; if we need this information inside a kernel function, we can just use the macros current_pid and current_ppid.

### 6.7.2 The getpsinfo and setpsname System Calls

The getpsinfo system call lets a process read its thread control block (the TCB structure). That way, a non-privileged ps program can show the process list. It is not possible to modify a TCB, but the TCB may contain information that should be kept private. In a security-aware operating system the information must be filtered if some of the data are considered confidential.

\[
\text{\textit{ulix system calls 206e}} \quad \text{\texttt{\textbackslash begin{verbatim} #define __NR_getpsinfo 503 \texttt{\textbackslash end{verbatim}}}}}
\]

Defines:
- __NR_getpsinfo, used in chunk 224.

\[
\text{\textit{syscall prototypes 173b}} \quad \texttt{\textbackslash begin{verbatim} void syscall_getpsinfo (context_t \*r); \texttt{\textbackslash end{verbatim}}}}
\]

\[
\text{\textit{syscall functions 174b}} \quad \texttt{\textbackslash begin{verbatim} void syscall_getpsinfo (context_t \*r) \{ \texttt{\textbackslash end{verbatim} unsigned int retval, pid; \texttt{\textbackslash end{verbatim} // ebx: thread ID \texttt{\textbackslash end{verbatim} // ecx: address of TCB block \texttt{\textbackslash end{verbatim} pid = r->ebx; \texttt{\textbackslash end{verbatim} if (pid > MAX_THREADS || pid < 1) \{ \texttt{\textbackslash end{verbatim} // legal argument? \texttt{\textbackslash end{verbatim} retval = 0; goto end; \texttt{\textbackslash end{verbatim} \} \texttt{\textbackslash end{verbatim} if (thread_table[pid].used == false) \{ \texttt{\textbackslash end{verbatim} // do we have this thread? retval = 0; goto end; \texttt{\textbackslash end{verbatim} \}} \texttt{\textbackslash end{verbatim}}}}}
\]

\[
\text{\texttt{\textbackslash begin{verbatim} \texttt{\textbackslash end{verbatim}}}}
\]
Implementation of Processes

// found a process: copy its TCB
memcpy ((char*)r->ecx, &thread_table[pid], sizeof (TCB));
retval = r->ecx;

end: eax_return (retval);

Defines:
syscall_getpsinfo, used in chunks 223d and 224a.
Uses context_t 142a, eax_return 174a, MAX_THREADS 176a, memcpy 596c, TCB 175, and thread_table 176b.

We also allow processes to set their own name via the setpsname system call. In most cases this happens automatically (because u_execv 228b writes the name into the appropriate field of the TCB entry, see below), but for some cases like the swapper daemon, we want to change the default name.

We also allow processes to set their own name via the setpsname system call. In most cases this happens automatically (because u_execv 228b writes the name into the appropriate field of the TCB entry, see below), but for some cases like the swapper daemon, we want to change the default name.

These functions let user mode programs access the process list:

uint getpsinfo (int pid, TCB* tcb) {
    return syscall3 (__NR_getpsinfo, pid, (uint)tcb);
}

uint setpsname (char *psname) {
    return syscall2 (__NR_setpsname, (uint)psname);
}

Uses __NR_getpsinfo 223c, __NR_setpsname 224d, syscall2 203c, syscall3 203c, and TCB 175.
6.8 ELF Loader

In this section we look at Ulix's `execv` function which is able to load ELF binaries (Executable and Linking Format) [TIS95] from disk.\(^2\) Classically, Unix systems provide several variants of exec functions (`execl`, `execle`, `exec1p`, `execv`, `execvp`, `execve` and `execve`) which differ in the way that arguments for the new program are provided. For the kernel one of these functions is sufficient, all other variants can be supplied by library functions which convert between the various syntaxes.

The standard procedure for launching an application on a Unix machine is to first `fork()` the current process and then load a new program binary in the child process. That way, the parent process remains intact. (Note that non-Unix systems typically provide a different mechanism, for example Windows has a `CreateProcess` function which combines the creation of a new process and the loading of the program; it does not support `fork`.)

6.8.1 ELF File Format

Let's look at a simple ELF binary that we create on a Linux machine. We use assembler code since that allows us to create a very compact binary:

```plaintext
⟨example elf program test.asm 225⟩≡

bits 32
global main

main:
  mov eax, 1
  mov ebx, 42
  int 0x80

Uses main 44b.
```

The equivalent C code would only contain `exit(42)`: these assembler commands make a system call (with system call number 1 which is `__NR_exit`) and the argument 42.

We can assemble this program with `nasm -f elf32 test.asm` which creates `test.o`; then we link it with `gcc test.o -nostdlib -e main44b -o test`, creating the binary `test`.

Let's check that this program works as expected and see what kind of information we can gather about it:

```plaintext
linux$ ./test ; echo $?
42
linux$ file test
```

```plaintext
test: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, BuildID[sha1]=0xa45ecc892186bae9977605e0c3d6757bdef2861b, not stripped
```

\(^2\) Note that there is an alternative implementation of the Ulix ELF loader by Frank Kohlmann that he developed as part of his Bachelor's thesis [Koh13] which was supervised by Hans-Georg Eßer. It is available on the Ulix website and shows more details, however the language is German.
linux$ stat -c "%s" test  # filesize?
631
linux$ readelf -e test
ELF Header:
Magic: 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
Class: ELF32
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Intel 80386
Version: 0x1
Entry point address: 0x80480a0
Start of program headers: 52 (bytes into file)
Start of section headers: 224 (bytes into file)
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 2
Size of section headers: 40 (bytes)
Number of section headers: 6
Section header string table index: 3

Section Headers:

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td>NULL</td>
<td>000000</td>
<td>000000</td>
<td>000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.note.gnu.build-id</td>
<td>NOTE</td>
<td>08048074</td>
<td>000074</td>
<td>000024</td>
<td>0</td>
<td>A</td>
<td>4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>.text</td>
<td>PROGBITS</td>
<td>080480a0</td>
<td>0000a0</td>
<td>0000c</td>
<td>0</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>000024</td>
<td>000034</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>00000000</td>
<td>0001d0</td>
<td>00000000</td>
<td>10</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>.strtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>000250</td>
<td>000027</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>VirtAddr</th>
<th>PhysAddr</th>
<th>FileSiz</th>
<th>MemSiz</th>
<th>Flg</th>
<th>Align</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>0x000000</td>
<td>0x8048000</td>
<td>0x0048000</td>
<td>0x000ac</td>
<td>0x000ac</td>
<td>R E</td>
<td>0x1000</td>
</tr>
<tr>
<td>NOTE</td>
<td>0x000074</td>
<td>0x8048074</td>
<td>0x0048074</td>
<td>0x00024</td>
<td>0x00024</td>
<td>R</td>
<td>0x4</td>
</tr>
</tbody>
</table>

Section to Segment mapping:

<table>
<thead>
<tr>
<th>Segment Sections...</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 .note.gnu.build-id .text</td>
</tr>
<tr>
<td>01 .note.gnu.build-id</td>
</tr>
</tbody>
</table>
The ELF format and the readelf tool (which is available via the binutils package on Linux distributions) are discussed in detail in the book "Professional Linux Kernel Architecture" [Mau08, p. 1241 ff.].

In order to read ELF files we need to understand the two kinds of headers which they contain, the ELF header (Elf32_Ehdr) and the ELF program header (Elf32_Phdr). We have copied the following type definitions from the Linux header file /usr/include/elf.h.

```c
typedef uint16_t Elf32_Half;  
typedef uint32_t Elf32_Word;  
typedef uint32_t Elf32_Addr;  
typedef uint32_t Elf32_Off;  

typedef struct {  
    byte e_ident[16]; // Magic number and other info  
    Elf32_Half e_type; // Object file type  
    Elf32_Half e_machine; // Architecture  
    Elf32_Word e_version; // Object file version  
    Elf32_Addr e_entry; // Entry point virtual address  
    Elf32_Off e_phoff; // Program header table file offset  
    Elf32_Off e_shoff; // Section header table file offset  
    Elf32_Word e_flags; // Processor-specific flags  
    Elf32_Half e_ehsize; // ELF header size in bytes  
    Elf32_Half e_phentsize; // Program header table entry size  
    Elf32_Half e_phnum; // Program header table entry count  
    Elf32_Half e_shentsize; // Section header table entry size  
    Elf32_Half e_shnum; // Section header table entry count  
    Elf32_Half e_shstrndx; // Section header string table index  
} Elf32_Ehdr;

typedef struct {  
    Elf32_Word p_type; // Segment type  
    Elf32_Off p_offset; // Segment file offset  
    Elf32_Addr p_vaddr; // Segment virtual address  
    Elf32_Addr p_paddr; // Segment physical address  
    Elf32_Word p_filesz; // Segment size in file  
    Elf32_Word p_memsz; // Segment size in memory  
    Elf32_Word p_flags; // Segment flags  
    Elf32_Word p_align; // Segment alignment  
} Elf32_Phdr;
```

Defines:
- Elf32_Ehdr, used in chunk 228b.
- Elf32_Phdr, used in chunk 228b.
6.8.2 Implementation of the ELF Loader

The default functions which can launch programs on Unix systems are named exec*, and typically there is a variety of them. They differ in the way that users can provide arguments. For example, on a Linux machine the man pages for exec and execve list the following seven functions:

```c
int execl (const char *path, const char *arg, ...);
int execlp (const char *file, const char *arg, ...);
int execl (const char *path, const char *arg, ..., char *const envp[]);
int execve (const char *file, char *const argv[], char *const envp[]);
int execv (const char *path, char *const argv[]);
int execvp (const char *file, char *const argv[]);
int execvpe (const char *file, char *const argv[], char *const envp[]);
```

The functions with an envp[] argument allow the caller to supply a modified environment (a list of exported variables) which we do not support on Ulix: neither the shell nor other application programs can set or query such environment variables.

The functions execlp, execvp and execvpe need not be called with the absolute path of the program but can also accept a simple program name. In that case they will scan the $PATH variable and search all the listed directories that can contain binaries for the program file. Again, since Ulix does not support environment variables, there is also no $PATH variable.

That leaves only the two basic functions execl and execv. These two differ in how arguments for the program can be supplied: execl takes as many arguments as needed (behind the program path), whereas execv takes only two arguments and the second argument points to a list of arguments. For Ulix we have devised to provide the execv variant, both in the kernel (as u_execv) and in the user mode library (as execv):

```c
int u_execv (char *filename, char *const argv[], memaddress *newstack)
```

Our kernel function takes a third argument newstack that will be filled with the address of the new process' user mode stack. It also always returns (and provides the entry address of the newly loaded program if loading it was successful). Note that the user mode library function execv has a different semantics: it only returns if loading the program failed, otherwise the old program is gone and the loaded program starts.

```c
int u_execv (char *filename, char *const argv[], memaddress *newstack) {
    // returns start address of the loaded binary; or -1 if exec fails
    Elf32_Ehdr elf_header;  Elf32_Phdr program_header;
    (u_execv: check that the executable exists 229a)
    (u_execv: check permissions 580a)  // see chapter on Users and Groups
    (u_execv: prepare arguments on stack 231)
    (u_execv: zero out the memory 232c)
    (u_execv: load executable, return entry address 233b)
}
```

Defines:
- u_execv, used in chunks 228a and 234b.

Uses Elf32_Ehdr 227, Elf32_Phdr 227, and memaddress 46c.
6.8.2.1 Step 1: Checking the Executable File

It takes four steps to load and run the program inside the current process. We start with checking that the file we shall load actually is an ELF binary: We open it, read the ELF header which should be right at the beginning of the file and then check whether it contains the magic string that is used to recognize ELF files.

\[
\begin{align*}
\text{\texttt{\textasciitilde u_execv: check that the executable exists}} & \equiv 229a \\
\text{int fd = u\_open (filename, 0, 0);} & \quad \text{[229a]} \\
\text{if (fd == -1) return -1; \quad // error} & \\
\text{int sz = u\_read (fd, &elf\_header, sizeof (elf\_header));} & \\
\text{// check for ELF header} & \\
\text{if (sz != sizeof (elf\_header) || strncmp (elf\_header.e\_ident, "\x7f" "ELF", 4) != 0) \{} & \\
\text{\quad u\_close (fd);} & \\
\text{\quad return -1;} & \\
\text{\}} \\
\end{align*}
\]

Uses strncmp 594a, u_close 418a, u_open 412c, and u_read 414b.

6.8.2.2 Step 2: Preparing the Stack

The next step is to prepare the stack: Since programs can be called with arguments, we need to push them onto the stack so that when the program initializes, it can find the arguments where they are expected. We allow up to 512 bytes for such arguments, and the user mode stack always starts at the fixed address \texttt{TOP\_OF\_USER\_MODE\_STACK}. If the total length of the arguments is too long, the surplus arguments are lost.

Remember that the \texttt{main()} function of every program has this prototype:

\[
\begin{align*}
\text{\texttt{\textasciitilde main prototype}} & \equiv 229b \\
\text{int main (int argc, char** argv);} & \quad \text{[229b]} \\
\end{align*}
\]

When this function starts it expects to access its arguments on the stack like every other function does. The stack contents have to start with the return address, and then the arguments follow. Since we launch a new program we can start with an empty stack. The first address which can be used is 0xffffffff (as we’ve set \texttt{TOP\_OF\_USER\_MODE\_STACK} to 0xb0000000). We want to reserve 512 bytes for the argument strings.

Let’s assume that you start a program from the shell by issuing the command

\texttt{esser@uli6:/home/esser\$ args This is an example 1 2 3 verylongstring}

(\texttt{args} is a \texttt{Uli6} program that displays the list of all supplied arguments with their addresses.) This would mean that at the start of \texttt{args}, the \texttt{argc} parameter is set to 9, and \texttt{argv} points to an array of strings (i.e., an array of character pointers). What kind of data do we need to store?

- First of all, we need all the strings (\texttt{argv[0]} to \texttt{argv[8]}) which contain the characters that the arguments consist of, plus a null terminator for each argument.
- Then we need the list of addresses of these strings.
- Finally we need a pointer to the start of this list and the number of arguments.
Each address needs four bytes of storage, so in this example we need $9 \cdot 4 = 36$ bytes for the addresses, and the address of the list itself needs another four bytes.

We start with the result and show the output of `args`:

```bash
esser@ulix[6]:/home/esser$ args This is an example 1 2 3 verylongstring argc: 9, &argc: 0xaafffd8, argv: 0xaafffe00, &argv: 0xaafffdfe
len(argv[0]) = 4, &argv[0] = afffde24, argv[0] = args
len(argv[1]) = 4, &argv[1] = afffde29, argv[1] = This
len(argv[6]) = 1, &argv[6] = afffde3e, argv[6] = 2
esser@ulix[6]:/home/esser$
```

We can also request a hex dump of the memory area (thanks to the `hexdump` command in the kernel mode shell, see Chapter 17):

```bash
afffffd8 09 00 00 00 00 fe ff af 24 fe ff af 29 fe ff af ........$...)
affffe08 2e fe ff af 31 fe ff af 34 fe ff af 3c fe ff af ....1...4.<.
affffe18 3e fe ff af 40 fe ff af 42 fe ff af 61 72 67 73 >...@...B...args
affffe28 00 54 68 69 73 00 69 73 00 61 6e 00 65 78 61 6d .This.is.an.exam
affffe38 70 6c 65 00 31 00 32 00 33 00 66 65 72 79 6c 6f ple.1.2.3.verylo
affffe48 6e 67 73 74 72 69 6e 67 00 00 00 00 00 00 00 00 ngstring........
```

Note that the byte order is *little-endian* which means that an integer is stored in RAM with the lower bytes coming first. So, for example, the first four bytes of the second line of that hex dump, `2e fe ff af`, store the address `0xaafffe2e` (and not `0x2effefafe`).

From there we can understand the stack layout and work backwards to arrange the stack that way. Table 6.1 shows a detailed analysis of the stack’s contents. We work with a temporary variable stack which is a pointer to `unsigned int`; we set it to `TOP_OF_USER_MODE_STACK - 512` (which is `0xaafffe00`) so that it points to the beginning of the reserved area. That way we can use pointer arithmetic (`stack--;`) to move to the next address when we want to write (four-byte) addresses to the stack. The statement `*(--stack) = address;` is a push operation: it subtracts 4 from the stack address (pointer arithmetic) and *then* writes address to the new location. The number of arguments (argc) is not known yet, because `execv` accepts a null-terminated array of strings—in the example that is

```bash
[ "args", "This", "is", "an", "example", "1", "2", "3", "verylongstring", 0 ]
```

—so we need to walk through the list to find the number:
### Table 6.1: Analysis of the initial stack of a process after calling `exec()`.

<table>
<thead>
<tr>
<th>Address</th>
<th>Type</th>
<th>Contents</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xaffffdf8 – 0xaffffdfb</td>
<td>int</td>
<td>0x00000009</td>
<td>argc</td>
</tr>
<tr>
<td>0xaffffdc – 0xaffffdf</td>
<td>int</td>
<td>0xfafffe00</td>
<td>&amp;argv</td>
</tr>
<tr>
<td>0xaffffe00 – 0xaffffe03</td>
<td>int</td>
<td>0xfafffe24</td>
<td>&amp;argv[0]</td>
</tr>
<tr>
<td>0xaffffe04 – 0xaffffe07</td>
<td>int</td>
<td>0xfafffe29</td>
<td>&amp;argv[1]</td>
</tr>
<tr>
<td>0xaffffe08 – 0xaffffe0b</td>
<td>int</td>
<td>0xfafffe2e</td>
<td>&amp;argv[2]</td>
</tr>
<tr>
<td>0xaffffe0c – 0xaffffe0f</td>
<td>int</td>
<td>0xfafffe31</td>
<td>&amp;argv[3]</td>
</tr>
<tr>
<td>0xaffffe10 – 0xaffffe13</td>
<td>int</td>
<td>0xfafffe34</td>
<td>&amp;argv[4]</td>
</tr>
<tr>
<td>0xaffffe14 – 0xaffffe17</td>
<td>int</td>
<td>0xfafffe3c</td>
<td>&amp;argv[5]</td>
</tr>
<tr>
<td>0xaffffe18 – 0xaffffe1b</td>
<td>int</td>
<td>0xfafffe3e</td>
<td>&amp;argv[6]</td>
</tr>
<tr>
<td>0xaffffe1c – 0xaffffe1f</td>
<td>int</td>
<td>0xfafffe40</td>
<td>&amp;argv[7]</td>
</tr>
<tr>
<td>0xaffffe20 – 0xaffffe23</td>
<td>int</td>
<td>0xfafffe42</td>
<td>&amp;argv[8]</td>
</tr>
<tr>
<td>0xaffffe24 – 0xaffffe28</td>
<td>String</td>
<td>&quot;args\0&quot;</td>
<td>argv[0]</td>
</tr>
<tr>
<td>0xaffffe29 – 0xaffffe2d</td>
<td>String</td>
<td>&quot;This\0&quot;</td>
<td>argv[1]</td>
</tr>
<tr>
<td>0xaffffe2e – 0xaffffe30</td>
<td>String</td>
<td>&quot;is\0&quot;</td>
<td>argv[2]</td>
</tr>
<tr>
<td>0xaffffe31 – 0xaffffe33</td>
<td>String</td>
<td>&quot;an\0&quot;</td>
<td>argv[3]</td>
</tr>
<tr>
<td>0xaffffe34 – 0xaffffe3b</td>
<td>String</td>
<td>&quot;example\0&quot;</td>
<td>argv[4]</td>
</tr>
<tr>
<td>0xaffffe3c – 0xaffffe3d</td>
<td>String</td>
<td>&quot;1\0&quot;</td>
<td>argv[5]</td>
</tr>
<tr>
<td>0xaffffe3e – 0xaffffe3f</td>
<td>String</td>
<td>&quot;2\0&quot;</td>
<td>argv[6]</td>
</tr>
<tr>
<td>0xaffffe40 – 0xaffffe41</td>
<td>String</td>
<td>&quot;3\0&quot;</td>
<td>argv[7]</td>
</tr>
<tr>
<td>0xaffffe42 – 0xaffffe50</td>
<td>String</td>
<td>&quot;verylongstring\0&quot;</td>
<td>argv[8]</td>
</tr>
<tr>
<td>0xaffffe51 – 0xaffffffff</td>
<td>—</td>
<td>—</td>
<td>(unused)</td>
</tr>
</tbody>
</table>

(u_exec: prepare arguments on stack 231)≡

```c
uint *stack = (uint*) (TOP_OF_USER_MODE_STACK - 512);

// find number of arguments
word argc = 0;
while ((memaddress)(argv+argc) < TOP_OF_USER_MODE_STACK && argv[argc] != 0 )
    argc++;
```

Uses memaddress 46c and TOP_OF_USER_MODE_STACK 159b.

Now that we know the number of arguments, we can reserve space for their addresses. We use two variables in the following loop:

- **target** always points to the memory location where the next argument (string) is to be stored. After each step we add the last argument’s length to it.
- **stack** still points to the start of the reserved 512 bytes. In each step i we write the argument address into the location `stack + i`. Note again that due to pointer arithmetic, `stack + i` is `(int)stack + 4*i`.
6 Implementation of Processes

// copy arguments into the reserved 512 bytes
memaddress target = (memaddress)stack;
memaddress args_start = target;
target += argc*4;
for (int i = 0; i < argc; i++) {
    int size = strlen(argv[i])+1;   // string length plus terminator
    memcpy((void*)target, argv[i], size);   // copy i-th argument
    *(stack + i) = target;   // store its address
    target += size;
}

// finish stack preparation
*--stack = args_start;   // push pointer to argument list
*--stack = argc;   // push number of arguments
*--stack = 0;   // push return address (set to 0)
*newstack = (memaddress)stack;

Finally, we push the arguments for main(int argc, char **argv) and the null return address onto the stack. These will be stored just below the reserved area.

If the main() function of a program simply returns (and does not call exit218a) the normal behavior would be an implicit execution of exit218a. We do not provide this feature. However, we have to store some value on the stack that tells where to return to. The start address of exit218a would be a candidate, but in our Ulix implementation we do not know that address, so we just write 0. If you write an application program that leaves main() via return you will see that it just starts over (or jumps into whatever function was compiled to address 0). Thus, Ulix programs must leave via an explicit exit218a call.

6.8.2.3 Step 3: Clearing the Memory

The process memory will still contain data that was stored there before the process called execv235e. We do not want the new program to be able to read its predecessor’s data, so we delete that data by setting the whole user mode memory to zero:

memset ((void*)address_spaces[current_as].memstart, 0,
       address_spaces[current_as].memend - address_spaces[current_as].memstart);

Uses address_spaces 162b, current_as 170b, and memset 596c.

6.8.2.4 Step 4: Load the Program

Now everything is prepared for loading the program. We walk through the program headers of the ELF file and load the program code. The ELF header may point us to several ELF program headers, so we perform a loop: elf_header.e_phnum tells us how many ELF program headers there are.
Each such ELF program header must be read in separately, and then we have to check its type `program_header.p_type`: If it is `ELF_PT_LOAD`,

```c
#define ELF_PT_LOAD 1
```

then we read program code from the file, otherwise we ignore it.

```c
int phoffset = elf_header.e_phoff;
for (int i = 0; i < elf_header.e_phnum; i++) {
    u_lseek (fd, phoffset + i * elf_header.e_phentsize, SEEK_SET);
    u_read (fd, &program_header, sizeof (program_header));
    if (program_header.p_type == ELF_PT_LOAD) {
        (u_execv: reserve sufficient memory)
        u_lseek (fd, program_header.p_offset, SEEK_SET);
        u_read (fd, (char*)program_header.p_vaddr, program_header.p_filesz);
    }
}
```

For each chunk that we need to load we find all the relevant information in the ELF program header:

- `program_header.p_offset` tells us the offset in the ELF file, so we can `u_lseek` to the right file location,
- `program_header.p_vaddr` contains the virtual address where the chunk is to be placed in memory, and
- `program_header.p_filesz` is the size of the chunk.

With these three values we can directly `u_lseek` and `u_read` the chunk without using an intermediate location.

If the loaded program is too big for the currently reserved memory or has a big BSS area (for zero-initialized variables), the loader must acquire more virtual memory via the `u_sbrk` function. It finds the total amount of required memory in the `p_memsz` element of the program header:

```c
int needed_memsz = program_header.p_memsz;
int current_memsz = address_spaces[current_as].memend
                    - address_spaces[current_as].memstart;
if (needed_memsz > current_memsz) {
    u_sbrk (needed_memsz-current_memsz);
}
```

Uses `address_spaces` 162b, `current_as` 170b, and `u_sbrk` 173a.
6.8.2.5 System Call Handler for `execv`

The system call handler is a little more complicated than usual because it has to deal with two possible situations: loading the program may succeed or fail.

- If it fails, no changes should be made to the current process, and it should receive a return value of −1 from calling `execv`.
- If it succeeds we need to update the process context so that it will (re-)start execution at the start address of the new program. Normally, that is 0. We also update the `cmdline` entry of the TCB.

Thus the function

```c
void syscall_execv (context_t *r);
```

has the following implementation:

```c
void syscall_execv (context_t *r) {
    // generate command line in one string
    char *path = (char*)r->ebx;  // path argument of execv ()
    char **argv = (char**)r->ecx;  // argv argument of execv ()
    int i = 0; char cmdline[CMDLINE_LENGTH] = "";
    while (argv[i] != 0) {
        strncpy (cmdline + strlen(cmdline), argv[i], CMDLINE_LENGTH-strlen(cmdline)-1);
        strncpy (cmdline + strlen(cmdline), " ", CMDLINE_LENGTH-strlen(cmdline)-1);
        i++;
    }
    if (cmdline[strlen(cmdline)-1] == ' ')
        cmdline[strlen(cmdline)-1] = '\0';  // remove trailing blank

    // call u_execv()
    memaddress stack;
    memaddress startaddr = (memaddress) u_execv (path, argv, &stack);  // sets stack
    if (startaddr == -1) eax_return (-1);  // error

    // update context and process commandline
    r->eip = startaddr;  // start running at address e_entry
    r->useresp = (memaddress)stack;  // from ELF header
    r->ebp = (memaddress)stack;
    strncpy (thread_table[current_task].cmdline, cmdline, CMDLINE_LENGTH);
}
```

Defines:
- `syscall_execv`, used in chunks 234a and 235c.

Uses `CMDLINE_LENGTH` 235a, `context_t` 142a, `current_task` 192c, `eax_return` 174a, `memaddress` 46c, `strlen` 594a, `strncpy` 594b, `thread_table` 176b, and `u_execv` 228b.

We have not yet defined the `cmdline` entry in the thread control block; we’ll add it now:
6.8 ELF Loader

\textit{public constants} 46a) +≡
\begin{verbatim}
define CMDLINE_LENGTH 50  // how long can a process name be?
\end{verbatim}

Defines:
CMDLINE_LENGTH, used in chunks 224c, 234b, and 235b.

\textit{more TCB entries} 158c) +≡
\begin{verbatim}
char cmdline[CMDLINE_LENGTH];
\end{verbatim}

Uses CMDLINE_LENGTH 235a.

Finally we register the system call handler and provide a user mode function \texttt{execv} that makes the system call:

\textit{initialize syscalls} 173d) +≡
\begin{verbatim}
install_syscall_handler (__NR_execve, syscall_execv);
\end{verbatim}

Uses \_\_NR\_execve 204c, \texttt{install_syscall_handler} 201b, and syscall\_execv 234b.

\textit{ulixlib function prototypes} 174c) +≡
\begin{verbatim}
int execv (const char *path, char *const argv[]);
\end{verbatim}

\textit{ulixlib function implementations} 174d) +≡
\begin{verbatim}
int execv (const char *path, char *const argv[]) {
    return syscall3 (__NR_execve, (uint)path, (uint)argv);
}
\end{verbatim}

Defines:
execv, used in chunks 191a and 235d.
Uses \_\_NR\_execve 204c and syscall\_3 234b.

6.8.3 User Mode Binaries

We use a linker configuration file \texttt{process.\_ld} to build our user mode applications and make the compiler use it via the command line option \texttt{-T process.\_ld}. Here’s the configuration file:

\textit{Application Linker Config File} 235f) ≡
\begin{verbatim}
OUTPUT_FORMAT("elf32-i386")
ENTRY(main)
virt = 0x00000000;
SECTIONS {
   . = virt;

   .setup : AT(virt) {
      *(.setup)
   }

   .text : AT(code) {
      code = .;
      *(.text)
      *(.rodata*)
      . = ALIGN(4096);
   }

   .data : AT(data) {
      data = .;
      *(.data)
      . = ALIGN(4096);
   }

   .bss : AT(bss) {
      bss = .;
      *(COMMON*)
      *(.bss*)
      . = ALIGN(4096);
   }

end = .;
}
\end{verbatim}

Uses main 44b.
The file tells the compiler to create ELF binaries with virtual addresses starting at address 0. We store the C source code files for our applications in a separate folder (lib-build/tools/) and use the following makefile to automatically compile the binaries and copy them to a folder which will be put onto the data disk image:

```
LD=ld
CC=/usr/bin/gcc-4.4
OBJDUMP=objdump
CCOPTIONS=-nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer \ 
    -fno-function-cse -nostartfiles -mtune=i386 -momit-leaf-frame-pointer
LDOPTIONS=-Tprocess.ld -static -s --pie
OBJECTS = $(patsubst %.c, %, $(wildcard *.c))

all: $(OBJECTS) copy

%.c
 $(CC) $(CCOPTIONS) -g $(LDOPTIONS) $^ ../ulixlib.o -o $@
 $(OBJDUMP) -M intel -D $@ > $@.dump

clean:
  rm $(OBJECTS)

copy:
  cp $(OBJECTS) ../diskfiles/bin/

There's some magic in this makefile: the line OBJECTS = $(patsubst %.c, %, $(wildcard *.c)) searches for all *.c files (with wildcard) and replaces each source filename with the filename without .c (e.g. hexdump.c with hexdump). Then the lines

%.c
 $(CC) $(CCOPTIONS) $(LDOPTIONS) $^ ../ulixlib.o -o $@

tell make the rule for creating binaries from source code files, and the all target gets a list of all the binaries that are to be created. $^ always refers to the source file (e.g. hexdump.c), and $@ refers to the target file (hexdump). Thus the expanded command for hexdump is

```
gcc-4.4 -nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer -fno-function-cse \ 
    -nostartfiles -mtune=i386 -momit-leaf-frame-pointer -T process.ld -static -s --pie \ 
    hexdump.c ../ulixlib.o -o hexdump
```
6.9 Exercises

In the tutorial/05/ folder you find a version of the ULIx kernel which contains the new code for handling system calls and also a sample solution for the keyboard interrupt handler exercise. It is a literate program (ulix.nw).

In this and the following two exercises you will implement and test some system calls. While you work on the solution, try to stick to the literate programming paradigm, i.e., integrate code and documentation into the document.

20. Writing strings with `printf`

The `printf()` function is available inside the kernel, but processes cannot call it. In the restricted tutorial version of ULIx there is user mode `printf()` function. You will now implement a system call handler which lets processes call the kernel’s `printf()` function. To make things easier, the goal is that you can later use a `userprint()` function which accepts exactly one string as an argument. (`printf()` takes a format string and an arbitrary number of further arguments, but that requires more effort and is not necessary for this exercise.)

a) Start with defining a syscall number for the `printf()` system call in the `<constants>` code chunk, e.g.

   ```
   #define __NR_printf 1
   ```

b) Next you write a syscall handler with the prototype

   ```
   void syscall_printf (struct regs *r);
   ```

   which calls the kernel function `printf()`. Make sure that you pass the proper arguments: Which of the registers (reachable via `r->eax`, `r->ebx`, `r->ecx` und `r->edx`) holds the address of the string?

   c) Enter the new syscall handler into the system call table.

d) Write a (user mode) function `void userprint (char *s);` which takes a string as argument and then uses one of the four `syscall*()` functions to perform the system call.

e) Verify that your code works correctly by adding the line

   ```
   userprint ("Testausgabe\n");
   ```

to your main function.

21. Reading Memory Locations with `kpeek`

The goal of this exercise is to let processes look at any (existing) memory location, even those that belong to the kernel. Of course, no proper operating system would supply such a function since it completely breaks all security mechanisms. Still, it can be done and shows you how to access data structures which are invisible in user mode. With some additional code this might be turned into a useful tool that, for example, lets only the system administrator access the memory.
You will need a function `int kpeek (unsigned int address);` which takes an address as argument, reads the byte that is stored at that address (a value between 0 and 255) and returns it. If the address is not available, the function shall return -1 (which is why its type is `int` and not `unsigned char` which would otherwise be the proper type for a byte).

If this was only about writing a kernel function for the task, you could implement `kpeek` like this:

```c
int kpeek (unsigned int address) {
    int page = address / 4096;
    if (pageno_to_frameno (page) == -1)
        return -1;
    else
        return *(char*)address;
}
```

But again, this function would only be usable by the kernel (which is the same problem that we had with `printf` in the previous exercise). Instead you have to implement `kpeek` via a system call. The general steps are the same as for `printf`:

- Assign and `#define` a system call number.
- Implement a syscall handler which contains a variation of the above `kpeek` code, but which performs parameter and return value passing via registers.
- Enter the new handler in the system call table.
- Write a function `kpeek` that uses the new system call (with the help of one of the `syscall*()` functions).

You can check whether your code works properly by adding the following lines to your `main()` function:

```c
unsigned int address = 0xc0000000;
*(char*)(address) = 123;
printf ("Testing kpeek: %d\n", kpeek (address));
```

The middle line writes 123 into the memory location `address`, and the last line should write “Testing kpeek: 123” to the screen. Try the same with an invalid address, e.g.

```c
printf ("Testing kpeek/fail: %d\n", kpeek (0x90000000));
```

(This time you should get a “Testing kpeek/fail: -1” output.)

22. **Writing to Memory Locations with kpoke**

Reading is one side of the coin, writing is the other. Now you’ll add a `kpoke` function that your processes can use to modify the contents of arbitrary kernel memory locations. It has the following prototype:

```c
void kpoke (unsigned int address, unsigned char value);
```
If address is a valid address, the value byte shall be written to that memory location, otherwise the function shall simply return. Again, if this was only about adding functionality to the kernel, the implementation would be as simple as

```c
void kpoke (unsigned int address, unsigned char value) {
    if ( ... ) // check for availability
        *(char*)(address) = value;
    return;
}
```

But again, that does not help a process. Implement `kpoke` by writing a system call handler and test the code with the following lines:

```c
ton unsigned int address = 0xc0000000;
kpoke (address, 123);
printf ("Testing kpoke: %d\n", kpeek (address));
```

This is the same test as in the last exercise, but this time you use `kpoke`. The implementation details are very similar to those of `kpeek`, so this time we don’t provide detailed steps.

Note that for proper testing of the new `printf`, `kpeek` and `kpoke` functions we would need user mode which is not available in this tutorial’s version of the kernel. But you’ll add that feature in the following exercises.

The tutorial/06/ folder contains a version of the ULIX kernel which implements the switch to user mode. Again, it is a literate program (`ulix.mw`).

### 23. User Mode Applications

With this exercise you will create the first user mode application and make it run. As usual: try to write a literate program.

a) **Test program:** First you will write a simple test program for ULIX so that you can see where all of this leads. Your program file `test.c` should only contain a `main()` routine as follows:

```c
int main () {
    printf ("Hello - User Mode!\n");
    for (;;) ; // infinite loop
}
```

You must add the implementation of `printf`: Add one of the `syscall*` functions (`syscall1.c`, `syscall2.c` etc.) to the file. Our simple `printf` implementation accepts only one string argument, it works just like `userprint` (from Exercise 20). You must also `#define` the constant `__NR_printf` (1).

Note: In the user mode program source files you must always place the `main` function at the very top. All other functions that you might want to call from `main` must be declared *above* the `main` function (by writing down the prototype),
but implemented below main. If you do not follow that rule, program execution will start in the wrong function (the one whose implementation comes first). The real Ulix does not suffer from this limitation because it has an ELF binary file loader, and ELF binaries contain the start address in the header; for this exercise we’re keeping things simple.

The Makefile already contains the necessary gcc invocation, you need not change it:

```bash
$(CC) -nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer \ 
-fno-function-cse -nostartfiles -mtune=i386 -momit-leaf-frame-pointer \ 
-T process.ld -static -o test test.c
```

By running `make` you generate the executable binary file `test` from the source code.

b) **Install the disassembler**: You will need a disassembler which translates a binary file back to readable assembler code. Install the disassembler package `x86dis` with the following command:

```
sudo apt-get install x86dis
```

c) Then disassemble the generated binary file `test` by running

```
x86dis -e 0 -s intel < test | sort -u
```

The output should begin as follows:

```
00000000 8D 4C 24 04 lea ecx, [esp+0x4]  
00000004 83 E4 F0 and esp, 0xF0  
00000007 FF 71 FC push [ecx-0x4]  
0000000A 51 push ecx  
0000000B 83 EC 08 sub esp, 0x08  
0000000E 83 EC 0C sub esp, 0x0C  
00000011 68 A2 00 00 00 push 0x000000A2  
00000016 E8 77 00 00 00 call 0x00000092  
...
```

This is the start of the translated `main()` function; the `call` instruction calls the `printf` function.

d) Since we have no filesystem in the current miniature Ulix kernel, we cannot load the program from disk. Instead we use a trick: We write the binary file directly into the kernel and later copy it into user mode memory. If you remember the `start_program_from_disk` function from Section 6.3, you will soon see a similar function `start_program_from_ram` that replaces it in the absence of disk access.

For copying the binary into the kernel we use the tool `hexdump` whose output format you can set via a format string:

```
hexdump -e '8/1 "0x%02X", "" -e '8/1 "\n"' test
```

creates an output of the following form:
(Compare these hexadecimal numbers with the numbers that the disassembler has shown: they are identical.)

You can copy the output directly to the kernel source (and add it to the \( \langle \text{global variables 92b} \rangle \) chunk). Declare and initialize a variable `usermodeprog` like this:

```c
unsigned char usermodeprog[] = {
  0x8D, 0x4C, 0x24, 0x04, 0x83, 0xE4, 0xF0, 0xFF,
  0x71, 0xFC, 0x51, 0x83, 0xEC, 0x08, 0x83, 0xEC,
  0x0C, 0x68, 0xA2, 0x00, 0x00, 0x00, 0xE8, 0x77,
  [...]
  0x6F, 0x64, 0x65, 0x21, 0x0A, 0x00, 0x00, 0x00,
  0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
};
```

You only have to add the first and last line to the `hexdump` output and remove the line with the single asterisk. If there is a line at the end that has only zeroes (0x00), you can delete it.

Now it’s time to write the function which loads the program.

e) Locate the chunk \( \langle \text{kernel main: user-defined tests} \rangle \) in the source code and add `start_program_from_ram ((unsigned int)usermodeprog, sizeof(usermodeprog));` as its new last line (before the terminating `@` line).

The function `start_program_from_ram` (which you’re going to implement) will load and start the program which involves a switch from kernel mode (ring 0) to user mode (ring 3).

24. User Mode Activation

Now you will implement some of the functions required for loading and starting a program. You have already seen how this works in this chapter, but here you can create your own implementation of a program loader (that loads from memory, not disk). Many code parts from the regular Ulix kernel are already present in the source code file.

a) The central function is `start_program_from_ram()`: As already mentioned, it takes the place of `start_program_from_disk()` from the real kernel. A further difference between Ulix and this version is that there is no scheduler; we start one single process and have no multi-tasking.
void start_program_from_ram (unsigned int address, int size) {
    addr_space_id as;
    thread_id tid;
    <<start program from ram: prepare address space and TCB entry>>
    <<start program from ram: load binary>>
    <<start program from ram: create kernel stack>>
    current_task = tid; // make this the current task
    cpu_usermode (BINARY_LOAD_ADDRESS,
                TOP_OF_USER_MODE_STACK); // jump to user mode
}

You need not type in this code, it is already included in ulix.nw.

b) In order to implement the missing chunks, follow these steps:

In ⟨start program from ram: prepare address space and TCB entry⟩ write appropriate
values into the two variables as, tid by calling create_new_address_space163c() and
register_new_tcb188d(). These function are similar to the ones in the real ker-
nel, they’re already contained in the ulix.nw file for this exercise. Note that the
order in which you call these two functions is important: register_new_tcb188d() takes
the address space ID as an argument. Give the new process 64 KByte of
memory and a 4 KByte user mode stack.

As a further task for the first code chunk you need to assign some values to the
TCB elements. Give the new process a PPID (parent process ID) of 0.

The chunk ⟨start program from ram: create kernel stack⟩ is almost identical to
⟨start program from disk: create kernel stack 192a⟩ and is also included in ulix.nw
already.

What’s missing is the code chunk ⟨start program from ram: load binary⟩: Here
you can use memcpy() to copy the usermodeprog array (whose start address and
length you have provided via the address and size parameters) to address 0.

cpu_usermode() is an assembler function that you can find in start.asm.

c) Test your code (calling make and make run)—after the old "Hello World" mes-
sage Ulix should also print the line from the user mode program (“Hello – User
Mode!”).

25. More Features for User Mode

In the final exercise of this chapter you add an input mechanism so that your user
mode programs can interact with the user. The goal is to let the application read text
from the keyboard with readline(). This requires several additions:

a) Write a syscall handler which calls the kernel function kreadline(). Like all
syscall handlers it has the prototype

    void syscall_readline (struct regs *r);
When it is called, r->eax contains the syscall number (which you can ignore), and r->ebx holds the address of a string (that was declared in the user mode program). In the application source file test.c declare a string variable which can hold 256 characters. We do not provide the string length; when calling kreadline you can ask that the returned string be no longer than 256 characters. In the handler you have to enable interrupts before calling kreadline() because the system deactivates them upon handler entry. Add the following line:

```
asm volatile("sti"); // before kreadline()!
```

Assign a syscall number __NR_readline and register the handler function in the syscall table.

b) Now add a function void readline(char *s); to test.c that uses one of the syscall* functions to make the system call. You will have to define the constant __NR_readline in that file. Remember to put function prototypes in front of main() and implementations behind it so that process execution starts with the first command in main().

c) Modify the main() function in test.c so that it continuously reads in text and prints it on the console, like this:

```c
int main () {
    char s[256];
    printf("Hello - User Mode!\n");
    for (;;) {
        printf("> "); readline(s);
        printf("Input was: "); printf(s);
    }
}
```

d) Test your program. After compiling test.c with make you will again have to convert the binary file into an array of hexadecimal numbers (as you did in Exercise 23 d–e) and integrate it in the kernel file ulix.nw. Then call make once more to generate the modified kernel.

The output function of this exercise’s kernel is able to scroll so that you can simply go on printing even after you’ve reached the bottom of the screen. In order to understand how scrolling was implemented, look at the scroll() function and the two places from where it gets called.

e) The scroll() function determines the right memory address to write data to by looking at the VIDEO variable: Search for all the lines in the code that change VIDEO—the variable takes three different values (0xc00b8000, 0xb8000, 0xd00b8000) during system initialization—why is that so? As a reminder, the physical addresses that are used for the text mode framebuffer start at 0xb8000.

f) Copy the code from test.c into the literate program ulix.nw and modify the Makefile so that test.c will be extracted from it. Name the code chunk ⟨test.c⟩. You will need an extra invocation of notangle, analogous to
notangle -Rulix.c ulix.mw >ulix.c

where the option -R is followed by the new chunk name and output redirection writes to the right file. Note that all commands in the Makefile must be indented by a tabulator character.

Search for code that appears identically in both ulix.c and test.c and combine it by creating code chunks which will be written to both files. As a result the literate program will be free of duplicates.
Implementation of Threads

In Chapters 3 and 4 we looked into a fundamental abstraction offered by the operating system: virtual memory. It abstracts physical memory, one of the main hardware resources. The second such resource is processor time, i.e., machine cycles or computation power offered by the CPU. The abstraction which encapsulates processor time in an operating system was traditionally called a process, in newer systems threads have taken over that job. We have just discussed the implementation of processes in the previous Chapter 6.

Up to now, we used the more historic term process in parts of this book instead of the term thread. Briefly spoken, a process is a virtual address space (defined by a page directory and page tables) plus exactly one thread. Thus, the term process alludes to the classical Unix process. Today, modern operating systems offer multiple threads within one virtual address space, and so does Ulx.

Thus, a summary of the differences between classical operating systems that are based on processes and newer systems with threads can be given as follows:

- In classical systems, process management provides a common abstraction for both processor time and memory. Switching from one process to another always means that the used address space changes, too.
- In modern systems, thread management and memory management are decoupled: It is possible to switch from one thread to another without also changing the address space. A process is simply a collection of one or more threads which share a common address space.

Note that some operating systems allow variations of the thread and process concepts which are something in-between. For example, the Linux kernel provides a clone function which can create new processes, new threads and other kinds of tasks which are neither.
7.1 Threads, Teams of Threads and Virtual Processors

A thread\(^1\) can be regarded quite literally as an execution thread within the operating system. Threads are abstractions of processing time, virtual processors. They are implemented by multiplexing virtual processors (the threads) onto the physical processors (CPUs). A thread always has an associated program, i.e., a sequence of machine instructions which it executes. When a thread starts its operation, execution starts at a pre-defined address in this sequence.

Threads and address spaces are two abstractions which are orthogonal but nevertheless closely tied together. Whenever a virtual address space is created, a first thread is also created within the address space. This results in what is often called a process. In most cases (as in early Unix) this is absolutely sufficient to perform all the classical application tasks programmed on top of the operating system, and it is what you have seen when we described the \texttt{u\_fork}\(^2\) implementation of the fork mechanism. However, it sometimes makes sense to create multiple threads within a single address space, as we now explain.

7.1.1 Teams of Threads

We call multiple threads within a single address space a team (or team of threads). Why does it make sense to create multiple threads within one address space? There are several answers to this question.

The first block of answers refers to performance issues:

- If within an application one thread invokes a system call which blocks for an I/O operation to succeed, then the whole application will block if the application is carried on just this one single thread. If more than one thread would carry the application, these other threads could continue to operate, giving the user a better quality of service.

- Also, if an application runs on multiple threads, it is possible to distribute the machine cycles onto physically distinct processors. This (of course) is not an issue in a monoprocessor system. However, in a dual processor system for example an application which is carried only by a single thread will never be able to bring the power of the two CPUs into the application.

The second block of answers refers mainly to software engineering aspects, i.e., the way we write programs.

- Multiple threads within one address space allow to program those applications which contain inherent parallel activities in a much more natural way. The result is a concurrent model of programming which includes both the fields of distributed and parallel programming. Concurrent programming refers to programming multiple independent threads of execution in general.

\(^1\) The theory Sections 7.1 and 7.2 of this chapter are heavily based on the “Threads” chapter of Nehmer and Sturm’s book [NS01].
• Parallel programming on the one side refers rather to more dependent threads, e.g., threads which operate in strongly synchronized “lock-step” mode.

• Distributed programming on the other side refers to concurrent programming where the aspect of geographic distribution plays a role (like in the Internet).

### 7.1.2 Natural Concurrency

Many of today’s operating systems already support multiple threads in one address space and so it is becoming more and more natural to use them. It is especially natural if the application which is implemented already contains inherent concurrency. As an example (taken from Nehmer and Sturm [NS01]) consider a weather reporting application. It consists of a huge database in which new measurements of humidity, temperature etc. are regularly logged from different sensing stations. From this database the application computes in a continuous manner weather reports for different areas of the country using complex weather models. Additionally, the application has a graphical interface through which users can inspect data, query weather reports and visualize measurement data.

Looking at the application from a concurrent programming viewpoint, it has three rather independent streams of activity:

1. The measurement and logging activity of data into the database.
2. The continuous weather prediction and reporting computation.
3. The graphical user interface.

Note that each stream of activity by itself is sequential.

Let’s make things simple and just look at the last two activities: computation and user interface. As both are sequential activities, we can program them separately and enclose each activity within a thread. The pseudocode could look like this:

```plaintext
⟨weather reporting example: thread pseudocode 247⟩≡
void Compute () {  // activity 1: computation
    while (true) {
        // do the actual computation
    }
}

void GUI() {  // activity 2: graphical user interface
    while (true) {
        Event e = ReceiveEvent ();
        ProcessEvent (e);
    }
}

int main () {
    start_thread (Compute ());  // Start concurrent threads
    start_thread (GUI ());
}
```
Note that the sequential activities are encoded within simple sequential functions which are both started within separate threads in the main routine and thereafter run separately. Here we assume that the entire application exits when all of its threads have exited.

How would we program this application traditionally (i.e., without threads)? We would have to split the activities into small slices and run them alternately. Assume we can divide the function Compute() into small parts called ComputeStep. Then after computing such a step we would need to check whether user input must be handled. If yes, we handle it, if not, we compute the next step. The pseudocode could look like this:

```
int main () {
    while (true) {
        ComputeStep ();   // compute the next step
        if (QueryEvent ()) {
            // do we have to process an event?
            e = ReceiveEvent ();
            ProcessEvent (e);
        }
    }
}
```

This approach should also work, but only under the assumption that we can in fact split Compute into ComputeStep. In many cases this is not as easy as it seems, sometimes it might even be impossible. Another disadvantage of the traditional approach is that the computation is interrupted regularly even if there are no events to be processed. In this case the code for QueryEvent should be very efficient so that it doesn’t cost too many CPU cycles. It goes without saying that functions like QueryEvent should not block (e.g., until user input arrives) because this would block the entire application.

There are more downsides of the traditional approach. For example, the program structure without threads determines the reaction time to user input. If ComputeStep may take up to a couple of seconds of execution time, then reaction to user input can also take this time. The execution time of ComputeStep should therefore be rather short to guarantee responsiveness. However, a short execution time implies that the overhead of QueryEvent increases in relation. So we have a non-trivial tradeoff here. Finally, but this is a matter of taste, we find the traditional code much harder to read and understand than the code using threads.

### 7.1.3 Advantages of Concurrent Programming

Threads allow to create an unbounded number of virtual processors, no matter how many physical processors exist in the system. This lets us distribute applications over as many virtual processors as are necessary to serve their inherent concurrency. Threads therefore allow to abstract from the actual number of physical processors in the system and to depart from the traditional sequential programming model. If an application has inherent, natural concurrency, then it should be expressed in the program.

Threads do not only make programs with inherent concurrency easier to read and understand, they also may make the execution of the application more efficient since only
concurrent applications can exploit the power of truly concurrent hardware available in multiprocessor systems. But even on monoprocessor systems a concurrent program can be more efficient than its sequential counterpart because the periods in which one thread is blocked (e.g., due to lack of user input) can be used by other threads more effectively.

### 7.1.4 Virtual vs. Physical Processors

As mentioned above, a thread can be regarded as a *virtual processor*. Therefore, a team of threads can be regarded as a *virtual multiprocessor*. Ideally, every virtual processor is backed up from below by exactly one physical processor and the assignment of virtual to physical processors is fixed. However, the normal case is rather different: Many virtual processors need to be executed on few physical processors. The task of the operating system is to distribute the physical processor cycles as effectively as possible between the virtual processors in a kind of *time division multiplex* mode of operation. This is depicted in Figure 7.1.

As an example, consider the case where one physical processor carries two virtual processors (threads). In this case the threads would be assigned alternately to the physical processor by the operating system. The change from one thread to the other is called a *context switch*. Within the context switch, the execution of the current thread is interrupted, the processor context (registers, stack pointer etc.) is saved somewhere, the processor context of the next thread is loaded from somewhere onto the processor and the next thread then continues execution at the point in its program where it was previously interrupted.

In a sense, the operating system pretends that every thread has exclusive access to the physical processor. During a context switch, the previously running thread is “frozen” and saved somewhere. The next thread is selected and “unfrozen” by loading its state into the CPU. During the times in which they are frozen, threads do not operate. In fact, since they don’t operate, they are not aware that time is passing. After unfreezing the new thread, it continues operation as if it had never been interrupted. This can remotely be compared with becoming unconscious after a knock out in boxing.

![Figure 7.1: The assignment of virtual to physical processors can change over time.](image-url)
If there is more than one candidate for the next running thread, the operating system has to make a choice. The operating system component which is responsible for making this decision is called the scheduler. As we will see later (in Chapter 8) there exist many different strategies to make this scheduling decision.

7.2 Thread Requirements and Thread Types

Before we delve into the implementation, let’s take a closer look at why threads are so useful. They help the operating system reach a higher degree of concurrency for the applications it runs.

7.2.1 Thread Requirements

If an operating system supports threads, it must offer at least two types of functionality: On the one hand, a user should be able to create a new address space with a single thread. On the other hand, the user should be able to assign a new program to this thread. Often these two functionalities are assembled within one single system call offered by the kernel.

To offer more flexibility, it should be possible to create multiple threads within one address space. Good operating systems therefore offer functionality to create a new thread within the same address space at runtime and to assign a new program to this thread.

7.2.2 Utility of Threads

Normally, several threads wait for the same processor to become free. Let’s assume that each thread, once activated, uses \( k \) units of time for completion and would run until it is finished. The first thread starts at once, the second thread after \( k \) units of time, etc. That leads to an average response time of

\[
\frac{1}{n} \sum_{i=1}^{n} (i - 1) \cdot k = \frac{n - 1}{2} \cdot k
\]

units of time [NS01, p. 101]. However, this ignores that threads typically alternate between CPU and I/O bursts:

- A CPU burst is a time range during which a thread uses the CPU, i.e., it is active and executes instructions.
- An I/O burst is a time range during which a thread waits for the completion of an I/O operation that it initiated. The burst begins in the moment where the thread is put on a blocked queue (as a direct result of requesting the I/O operation) and it ends when the I/O operation completes and the thread is moved to the ready queue.

If there is only one single thread, then the system switches between CPU and I/O bursts of that thread. With several threads and a scheduler the situation becomes more complicated since the scheduler can interrupt an active thread in the middle of a CPU burst.
Also, whenever an I/O burst begins, the CPU is reassigned to a different thread (which continues its CPU burst).

If I/O was handled completely asynchronously (i.e., we ignore the times required to process I/O requests), the CPU burst times would lead to an average response time of

\[ \frac{n - 1}{2} \cdot t_{\text{burst}} \]

where \( t_{\text{burst}} \) is the average length of a CPU burst [NS01, p. 102].

Nehmer and Sturm measured the length of bursts and looked at their statistical distribution which resulted in the image shown in Figure 7.2 [NS01, p. 102]. This shows that typically CPU bursts are very short—much shorter than I/O bursts. Using the time that a thread waits for I/O completion for other threads (which continue their CPU bursts) creates a considerable degree of concurrency, even on a monoprocessor. Thus, it is important to pause threads which wait for I/O (which is precisely what we’ve been doing for processes in Chapter 6), and with threads it improves the behavior of processes by allowing CPU-burst threads of a process to continue while I/O-burst threads are blocked. This improves response times for those processes.

![Figure 7.2: Distribution of the CPU burst length: A length of 2 ms occurred most often (as measured by Nehmer and Sturm).](image)

However, once the number of threads becomes very large, too many of them will be in their CPU bursts simultaneously, and then the overall execution speed and responsiveness will shrink.
7.2.3 Types of Threads

Two different types of threads are usually distinguished: kernel-level threads and user-level threads. The “classical” threads (i.e., the threads in Unix processes) are kernel-level threads. The distinction is based on the mode in which the context switch is performed. In kernel-level threads the context switch happens in system mode, in user-level threads it happens in the user mode thread library.

Kernel-level threads can be regarded as virtual processors running directly on physical processors. User-level threads can be regarded as virtual processors running on kernel-level threads. In this sense, a team of kernel-level threads running in the same virtual memory can be regarded as a virtual multiprocessor for user-level threads.

The techniques used to implement and synchronize virtual processors (i.e., kernel-level threads) on physical processors are the same as those used to implement and synchronize virtual processors (user-level threads) on virtual processors (kernel-level threads). Therefore people sometimes speak of a processor hierarchy. Virtual processors run on virtual processors that run on physical processors. (Note that in principle it is even possible to run user-level threads on user-level threads, extending the processor hierarchy.) The multiplexing of higher-level processors to lower-level processors is performed by a software layer (see Figure 7.3). In case of kernel-level threads implemented on physical processors this software layer is the operating system; in case of user-level threads implemented on kernel-level threads this software layer is usually called the user-level threads library.

Figure 7.3: Schematic view of the processor hierarchy. Arrow colors (on one level) express the order of allocation of a virtual (top) or physical (bottom) processor, but the top and bottom arrows have their own times.
In the example shown in Figure 7.3 the green and blue arrows describe the order in which a virtual/physical processor is assigned to a user-level/kernel-level thread. If we assume that the time slices for user and kernel level threads are identical, then the example thread might execute as shown in Table 7.1, but in practice switch times for user-level threads will not be synchronized with switch times for kernel-level threads.

<table>
<thead>
<tr>
<th>Time</th>
<th>KLT</th>
<th>Ult</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n + 1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>n + 2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>n + 3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>n + 4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>n + 5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>n + 6</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>n + 7</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>n + 8</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>n + 9</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>n + 10</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 7.1: Possible order of execution for the threads shown in Figure 7.3.

### 7.2.3.1 Kernel Level Threads

Kernel-level threads are managed in system mode, i.e., the context switch of one kernel-level thread to the next requires some action by the operating system’s scheduler. This is why kernel-level threads are often called heavy-weight threads, because entering and leaving the scheduler incurs a large performance overhead. For example, if the context switch from one kernel-level thread to the next also causes a switch of one virtual memory to another (i.e., the process changes), the effect of caching in the TLB or in any other local cache is destroyed.

### 7.2.3.2 User-Level Threads

User-level threads run on kernel-level threads. From the point of view of user-level threads, kernel-level threads are virtual processors that carry the user-level thread library. In contrast to kernel-level threads, user-level threads are managed entirely in user mode. This implies that the context switch of user-level threads does not incur a context switch of the kernel-level thread that carries it. This in turn implies that there is no switch of virtual memories and no performance penalty. Therefore, user-level threads are often called light-weight threads.

Ulǐx only implements kernel-level threads. We show the code in the following section.
7.3 Implementation of Threads in ULIx

So far, we have been talking about threads all the time, e.g., when we discussed the structure of the thread control block, but all the previous code actually dealt with processes. That changes now: we are about to introduce true threads which can be created inside processes.

7.3.1 Creating Threads Instead of Processes

Creating a kernel-level thread (belonging to an already existing process) is very similar to forking a process. The original idea was to let u_fork perform its usual tasks, but with two exceptions—it should not copy the user mode memory, and it should provide two new stacks (for kernel and user mode) which exist in the same address space. This approach is used in the Linux kernel’s clone function which can generate both new threads and new processes [Lov03, pp. 22–27].

However, our first attempts led to many if-then-else constructions in the u_fork function which reduced the readability of the code, and we also wanted to describe process forking and thread creation separately, so we decided against the combined treatment inside u_fork. Instead we explain thread creation in this section, and we will provide a separate function u_pthread_create which is loosely based on the POSIX user mode function pthread_create.

We’re going to implement (parts of) the POSIX thread API [IEE95] inside the kernel, and the user mode library functions will simply access these kernel functions via system calls.

POSIX threads need a thread identifier of some type pthread_t. We define this public type as a pointer to a kernel-internal data structure whose definition we will not export to user land, so we declare:

```
typedef void *pthread_t;
typedef void pthread_attr_t;  // attributes not implemented
```

As already mentioned, a new thread differs from a new process by sharing its creator’s address space, but it still needs its own kernel and user mode stacks—in the same address space which the calling process currently uses.

Our implementation of the kernel’s

```
int u_pthread_create (pthread_t *thread, const pthread_attr_t *attr,
                     memaddress start_address, void *restrict arg);
```

function looks similar to u_fork.
7.3 Implementation of Threads in Ulix

\( \langle \text{function implementations} \rangle \) +\( \equiv \) \( \langle \text{begin critical section in kernel} \rangle \) \( \langle \text{access the thread table} \rangle \) \( \langle \text{create new TCB} \rangle \) \( \langle \text{fill new TCB} \rangle \) \( \langle \text{create new stacks} \rangle \) \( \langle \text{end critical section in kernel} \rangle \) return 0;

Defines:
\( \text{u_thread_create} \), used in chunks 254b and 258b.
Uses \( \text{current_task} \), \( \text{memaddress} \), \( \text{pthread_attr_t} \), \( \text{pthread_t} \), and \( \text{thread_id} \).

Where \( \text{u_fork} \) starts with creating a fresh address space and reserving a TCB, the thread only needs a TCB:

\( \langle \text{create new TCB} \rangle \equiv \)
\( \langle \text{fill new TCB} \rangle \)
\( \langle \text{create new stacks} \rangle \)

Uses \( \text{address_spaces} \), \( \text{current_as} \), \( \text{register_new_tcb} \), and \( \text{thread_id} \).

We increase the refcount and extra_kstacks elements of the address space which lets us keep track of how often this address space is in use: As long as refcount is non-zero, we must not reuse the address space. extra_kstacks has a similar but slightly different function that we will explain further below.

Entering data in the TCB is similar to the respective step in \( \text{u_fork} \):

\( \langle \text{fill new TCB} \rangle \equiv \)
\( \langle \text{create new stacks} \rangle \)

Uses \( \text{context_t} \), \( \text{current_as} \), \( \text{memset} \), \( \text{t_new} \), \( \text{t_old} \), \( \text{TCB} \), \( \text{thread_table} \), and \( \text{TSTATE_FORK} \).

We need two new TCB entries to mark a thread as new and to store its kernel stack address:

\( \langle \text{more TCB entries} \rangle \equiv \)
We will give only one page of kernel stack memory to each new thread, and we put those just under the regular kernel stack.

Remember that we’ve defined `TOP_OF KERNEL MODE STACK` and `KERNEL STACK PAGES`. We can now calculate `TOP_OF KERNEL MODE STACK – KERNEL STACK PAGES * PAGE_SIZE` to find the lowest possible address that may be used by the kernel stack. In principle we could have our first thread’s kernel stack start just below, but we want to provide a protective buffer of unmapped memory: That way, whenever one of the threads exceeds its kernel stack it will generate a page fault (and in turn the process will be stopped).

We will provide each thread with just one page of kernel stack memory whereas the initial process always gets four pages of them—this is just intended to simplify the clean-up of a process with several threads (Figure 7.4).

![Figure 7.4](image-url)
This leads to the address calculation

\[
\text{TOP}\_\text{OF}\_\text{KERNEL}\_\text{MODE}\_\text{STACK} - \left(\text{KERNEL}\_\text{STACK}\_\text{PAGES} + 2 \times \text{thread}\right) \times \text{PAGE}\_\text{SIZE}
\]

where thread is 1, 2, etc. for the first, second, etc. new thread (and it is 0 for the original kernel thread of the process). For example, for the first new thread (and assuming that \(\text{TOP}\_\text{OF}\_\text{KERNEL}\_\text{MODE}\_\text{STACK} = 0xc0000000\), \(\text{KERNEL}\_\text{STACK}\_\text{PAGES} = 4\)) we get \(0xc0000000 - (4 + 2 \times 1) \times 4096 = 0xbfff\text{fa}000\) which is the start address for the new stack.

We could use the address space’s refcount element in our formula (but would need to subtract 1 since it starts counting with the initial, thread-less process). However, this only works if we assume that a process will create several new threads and then enters a stage in which threads will only terminate until none is left.

If thread creation is more dynamic, with new threads coming and old threads going away, we cannot use this approach because a leaving thread will decrease refcount. So we introduce a new address_space entry extra_kstacks:

\[
\text{extra_kstacks} = \text{TOP}\_\text{OF}\_\text{KERNEL}\_\text{MODE}\_\text{STACK} - \left(\text{KERNEL}\_\text{STACK}\_\text{PAGES} + 2 \times \text{extra_kstacks}\right) \times \text{PAGE}\_\text{SIZE}
\]

which counts the number of extra kernel stacks. In a pure process (without extra threads) its value will always be 0. We will update refcount by increasing and decreasing it as threads come and go, but we will only increase extra_kstacks when we add a new thread. That way, for every new thread we can get a fresh kernel stack.

(As a side effect all extra kernel stacks will continue to exist until the process finally terminates; more about that at the end of this chapter.)

This finally lets us calculate the bottom of the new kernel stack:

\[
\text{bottom of new kernel stack} = \text{TOP}\_\text{OF}\_\text{KERNEL}\_\text{MODE}\_\text{STACK} - \left(\text{KERNEL}\_\text{STACK}\_\text{PAGES} + 2 \times \text{extra_kstacks}\right) \times \text{PAGE}\_\text{SIZE}
\]

Uses address_spaces 162b, current_as 170b, KERNEL_STACK_PAGES 169b, PAGE_SIZE 112a, and TOP_OF_KERNEL_MODE_STACK 159c.

Now we have everything that we need for stack creation. For the new user mode stack we simply increase the process’ heap (via \(\text{u}\_\text{sbrk}\)), and for the kernel stack we reserve a frame and update the address space.

\[
\text{u_pthread_create: create new stacks}
\]

\[
\begin{align*}
&\text{// create user stack} \\
&\text{void *ustack = u_sbrk (PAGE_SIZE);} \\
&\text{memset (ustack, 0, PAGE_SIZE);} \\
&\text{// create kernel stack} \\
&\text{int kstack_frame = request_new_frame ();} \\
&\text{uint kstack_start = \text{bottom of new kernel stack}}; \\
&\text{as_map_page_to_frame (current_as, kstack_start >> 12, kstack_frame);} \\
\end{align*}
\]
Implementation of Threads

```c
uint *STACK = (uint*) (kstack_start+PAGE_SIZE); // top of new stack
t_new->top_of_thread_kstack = STACK;

*(--STACK) = 0x20 | 0x03; // push ss (selector 0x20 | RPL3: 0x03)
*(--STACK) = (uint*)ustack + PAGE_SIZE; // push esp (for user mode)
*(--STACK) = t_old->regs.eflags; // push eflags
*(--STACK) = 0x18 | 0x03; // push cs (selector 0x18 | RPL3: 0x03)
*(--STACK) = start_address; // push eip (for user mode)

```

...}

```c
t_new->esp0 = (memaddress)STACK;
add_to_ready_queue (new_tid);
```

Uses add_to_ready_queue 184b, as_map_page_to_frame 165b, current_as 170b, kstack_frame, memaddress 46c, memset 596c, PAGE_SIZE 112a, request_new_frame 118b, t_new 276c, t_old 276c, top_of_thread_kstack, u_sbrk 173a, and ustack.

Compare the values that we push on the stack to the values we push in the assembler function cpu_usermode that is invoked when the very first process starts: These are the same data, except for the start address which is 0 for the initial process and the address of the thread function in case of a new thread. The reason why we need this stack setup is the same in both cases: When the iret instruction is executed, the stack has to contain the information that lets the system go back to user mode and continue execution at the right address.

### 7.3.2 System Call for Thread Creation

We provide a simplified user mode implementation of thread creation which ignores thread IDs and simply provides the start address of the thread function. This is possible because we will not provide a pthread_join function (that lets a thread wait for the termination of a specific other thread). The system call handler

```c
void syscall_pthread_create (context_t *r);
```

just calls u_pthread_create with the start address in the right place and all other arguments set to NULL.

```c
void syscall_pthread_create (context_t *r) {
    // ebx: address of thread function
    memaddress address = r->ebx;
    u_pthread_create (NULL, NULL, address, NULL);
}
```

Uses context_t 142a, memaddress 46c, NULL 46a, syscall_pthread_create, and u_pthread_create 255a.

The next free system call number is

```c
#define __NR_pthread_create 506
```

Defines: __NR_pthread_create, used in chunk 259.

and we add a syscall table entry:

```c
```

[258a] ⟨syscall prototypes 173b⟩+≡ (202a) ⟨234a 282a⟩

[258b] ⟨syscall functions 174b⟩+≡ (202b) ⟨234b 282c⟩

[258c] ⟨ulix system calls 206e⟩+≡ (205a) ⟨224d 260b⟩
7.3 Implementation of Threads in Ulix

\[\langle \text{initialize syscalls } 173d\rangle +\equiv\]
\[
\text{install_syscall_handler (_NR_pthread_create, syscall_pthread_create);}\]
Uses _NR_pthread_create 258c, install_syscall_handler 201b, and syscall_pthread_create.

For the user mode library we stick with the POSIX prototype

\[\langle \text{ulixlib function prototypes } 174c\rangle +\equiv\]
\[
\text{int pthread_create (pthread_t *thread, const pthread_attr_t *attr,}
\text{ void *address, void *arg);}\]

but as mentioned above, the only thing we pass along is the start address:

\[\langle \text{ulixlib function implementations } 174d\rangle +\equiv\]
\[
\text{int pthread_create (pthread_t *thread, const pthread_attr_t *attr,}
\text{ void *address, void *arg)} \{
\text{return syscall2 (_NR_pthread_create, (memaddress)address);}\}
\]
Uses _NR_pthread_create 258c, memaddress 46c, pthread_attr_t 254a, pthread_t 254a, and syscall2 203c.

### 7.3.3 Terminating Threads

In a complete POSIX thread implementation a thread can call pthread_exit 260e to terminate, and another thread may call pthread_join to wait for that specific thread to finish. Describing pthread_join in this book would be a repetition of the code that we’ve shown when we discussed the process mechanisms provided by waitpid 220d and exit 218a: We would add another blocked queue to the system and move a thread that calls pthread_join to that queue; then the pthread_exit 260e function would check whether there is a thread that waits for this thread and wake it up.

Since no deeper understanding is gained by this repetition, we only provide the

\[\langle \text{function prototypes } 45a\rangle +\equiv\]
\[
\text{void syscall pthread_exit (context_t *r);}\]

function. Note that this means that many multi-threaded code examples will not work with Ulix, but from our explanation it should be clear how you could extend the Ulix code so that it supports threads properly.

There is a special case we need to consider: The last thread of a multi-threaded process has two options for terminating.

- It can call the regular process exit function exit 218a. This will typically be the case if it was the “master process” that executes the main() function. In that case all other threads have already left via pthread_exit 260e.
- It can alternatively call pthread_exit 260e. The man page for pthread_exit 260e states:

  “After the last thread in a process terminates, the process terminates as by calling exit(3) with an exit status of zero; thus, process-shared resources are released and functions registered using atexit(3) are called.” [Lin12b]

So in this special case (which we can detect by checking refcount == 1 in the current address space) we simply call syscall_exit 216b and let it do the work.
void syscall_pthread_exit (context_t *r) {
    if (address_spaces[current_as].refcount == 1) {
        // last thread leaves: use normal exit mechanism
        r->ebx = 0; // set process exit code to 0
        syscall_exit (r); return; // and leave
    }
}

#define __NR_pthread_exit 507

Note that the POSIX prototype for pthread_exit provides an exit value argument which we omit because in our implementation there is no way for another thread to access it.
7.3.3.1 Getting Rid of the Extra Kernel Stacks

In \langle scheduler: free old kernel stacks 169a\rangle we had included a code chunk named \langle remove extra thread kernel stacks 261\rangle and given no further explanation (because at that time we only dealt with processes). Now the time has come to explain how to get rid of the extra kernel stacks.

In the overall kernel stack deletion chunk, we’re in the middle of a loop (over the elements of the kstack_delete_list), and id is the ID of the current address space that we need to delete. In a regular process address_spaces[id].extra_kstacks will be 0, we don’t want to touch such an address space any further. But if the value is larger than 0, we remove the extra pages:

\[
\text{if} \ (\text{address_spaces}[\text{id}].\text{extra_kstacks} > 0) \\
\text{while} \ (\text{address_spaces}[\text{id}].\text{extra_kstacks} > 0) \ {\}
\]

\[
\text{uint stack} = \text{TOP_OF_KERNEL_MODE_STACK} - \\
\text{(KERNEL_STACK_PAGES} + 2 * (\text{address_spaces}[\text{id}].\text{extra_kstacks})) \times \text{PAGE_SIZE};
\]

\[
\text{int frameno} = \text{mmu_p} (\text{id}, \text{stack/PAGE_SIZE});
\]

\[
\text{if} \ (\text{frameno} !=-1) \ \text{release_frame} (\text{frameno});
\]

\[
\text{address_spaces}[\text{id}].\text{extra_kstacks}--; \\
\]

Uses address_spaces 162b, KERNEL_STACK_PAGES 169b, mmu_p 171c, PAGE_SIZE 112a, release_frame 119b, and TOP_OF_KERNEL_MODE_STACK 159c.

(Note that we cannot use release_page 122d because the address space is not active; the current page table does not point to the frames we want to free.)

For determining the start of each kernel stack we use the same formula that we used when we created the stack (see \langle bottom of new kernel stack 257b\rangle).

7.3.4 Thread Synchronization

We will also provide \text{pthread_mutex_*} functions for thread synchronization. You can find the code in Chapter 11.5 (which is part of the \text{Synchronization} chapter).
Scheduling

Every multi-tasking operating system needs a scheduler: It is the primary OS component that allows the quasi-parallel execution of several programs. Scheduling actually encompasses two separate tasks:

- deciding when to switch from one process or thread to another and picking that new task
- and actually performing the task switch (also called context witch).

The first problem is what scheduling strategies are about, and this is where researchers regularly develop new schedulers. In the introductory theory part (Sections 8.1 and 8.2) we look at some classical strategies.

For Unix we will implement the Round Robin scheduling strategy, but picking the next process or thread according to this strategy is rather simple, so the implementation part (Section 8.3) of this chapter focuses on the context switch: Switching from one task to the other without breaking the system is complex.

### 8.1 Monoprocessor Scheduling

For this book we focus on scheduling strategies that work on systems with exactly one CPU: Unix does not support more than one processor. With several CPUs or cores (and even with hyperthreading) things get more interesting, and multiprocessor machines can profit from specialized scheduling strategies (even though most standard schedulers can be adapted to use more than one CPU, as well).
8.1.1 Quality Metrics

Scheduling is one of the best understood parts of operating systems because it has such an important impact on system performance. However, it is not so easy to say what the “best scheduler” is because it depends very much on the definition of quality used in a particular situation. Here are several common quality metrics for scheduling algorithms, the more historical ones are listed first:

- **CPU usage**: The metric of CPU usage is one of the simplest notions of quality in the literature. It basically gives the percentage of time in which the CPU actually executed application instructions (in contrast to operating system instructions or being idle). The CPU usage is important if CPUs are very expensive (as it was in earlier times). Today, the CPU usage of common desktop computers is usually very low, since they are idle most of the time.

- **Throughput**: The throughput of a system usually counts the number of tasks that the system executed per time unit. This metric depends on the definition of “task”. It comes from a time in which computers did batch processing: A number of computation jobs were ready in a physical entry queue (for example in the form of punched cards). The computer then started the processing of these jobs. The throughput counted the number of such jobs that the system could execute per hour (for example).

- **Turnaround time**: The turnaround time of a thread is the time it takes for the thread to be scheduled again. In other words, it is the time between two successive selections of the thread by the scheduler. The turnaround time of the entire system is the average turnaround time of all threads. It can be regarded as a refined throughput metric.

- **Waiting time**: The waiting time of a thread is the average time it has to wait in the ready queue before it is scheduled. This is not the same as throughput since times when the thread is blocked do not count in the waiting time.

- **Response time**: The response time of a thread is the time it takes for the thread to respond to user input. This is similar to the turnaround and waiting time, only that responses to user inputs are counted instead of being scheduled again.

- **Real time**: Finally, a scheduler is real time if it manages to satisfy real time constraints. There is a further differentiation into hard real time and soft real time which is basically about the question whether it is acceptable to occasionally miss a deadline.

8.1.2 Preemptive vs. Non-preemptive Scheduling

There are two main classes of scheduling algorithms: preemptive ones and non-preemptive ones. Roughly speaking, preemptive scheduling algorithms allow that a thread is thrown off the processor even if that thread does not want to be thrown off. In practice, all scheduling algorithms are usually preemptive in order to prevent that threads (accidentally or willingly) monopolize the system.

The precise definition is as follows: A scheduling algorithm is preemptive if an asynchronous interrupt can cause a thread to be taken off from the processor.
### 8.1.3 First-Come First-Served

The most simple approach to scheduling processes is to have a single queue for all ready processes. Whenever the CPU is not busy, the scheduler picks the first process in the queue and lets it compute until it either terminates or blocks. After a process becomes unblocked, it is appended to the end of the queue.

This is a non-preemptive strategy that is called *First Come, First Served (FCFS)* and that can be used on old machines which do not support timer interrupts. The most important problem with this approach is that it requires cooperation of all the running processes: If one process never freely gives up the CPU, it can go on forever.

A further analysis shows that the order in which processes enter the queue influences the average service time (the time between entering the CPU and finishing the calculation) heavily. For example, let’s assume that there are three processes P1, P2 and P3 that simultaneously come into existence. P1 needs 15 units of time, P2 and P3 need four and three units, respectively. Figure 8.1 shows three of the six possible ordering in which the processes can enter the queue (and thus start computing).

![Figure 8.1: The FCFS scheduler’s service time statistics depend heavily on the processes’ order of system entry.](image)

In the first execution sequence, P1, P2 and P3 finish after 15, 19 and 22 units of time, respectively. That means an average service time of \((15 + 19 + 22)/3 \approx 18.7\) units of time.

In the second sequence, the times are 3, 7, 22, leading to \((3 + 7 + 22)/3 \approx 10.7\) units of time, and in the last sequence, times 3, 18, 22 result in \((3 + 18 + 22)/3 \approx 14.3\) units of time. Instead of the service time we could also look at the wait time (which would be the service time minus the burst time of the process) which gives a similar result.

So, if our goal was to minimize the average service time (or wait time), it would make sense to pick the ordering in the middle which sorts processes by their runtime.

### 8.1.4 Shortest Job First

There is a hypothetical strategy that does just that: The *Shortest Job First (SJF)* strategy always picks the job that has the shortest runtime. Thus, of all the possible orderings it will always choose the one in the middle of Figure 8.1.
Why did we call it hypothetical? The problem is that in almost every case the system has no way to find out how long the next CPU burst of some process is going to be. In some rare cases where a system only executes specially prepared applications which announce their next burst length in advance, this strategy might actually be implemented, but for multi-purpose operating systems which run arbitrary programs, it is not possible. Still, it is possible to approximate this strategy. For example the operating system could collect statistical data about each process by monitoring the length of each CPU burst. Then it could calculate averages for all the bursts of a process and order the processes by their average burst times. Programs might change their behavior over time; consider a program that performs heavy calculations on a large set of data: It would start by reading in a big chunk of data (resulting in very short CPU bursts). Once the data are there, it would start the calculation (with very long bursts). Afterwards it might write them back, returning to short bursts. So in order to cater for that variability, it would make sense to discard older statistical burst data and only use the last $N$ burst times for calculating the average.

Also, once a process terminates, the system could store the collected statistical data about it in the filesystem: When starting the program again, it can make a better guess at what is about to happen.

Like FCFS, SJF is a non-preemptive strategy which does not interrupt processes. Both strategies are acceptable in non-interactive systems. But if a machine has a live user sitting in front of the machine who expects that several programs work seemingly in parallel, this is not good enough.

### 8.1.5 Round Robin

The idea behind the *Round Robin (RR)* scheduler is the same as that for FCFS—but with interrupts which force a process off the CPU once a time limit has been reached. The maximum time that a process is allowed to execute is called a *time slice*, and its length is an adjustable parameter of the RR strategy.

Figure 8.2 shows how an RR scheduler would treat the three sample processes from above. At the top you see the order of execution with the time slice set to four units of time, the lower part shows the sequence for a time slice of two units.

![Round Robin scheduler diagram](image)

Figure 8.2: The Round Robin scheduler works with configurable time slices. Here it uses slices of four (top) and two units of time.
If a system already has an FCFS scheduler, it can easily be upgraded to an RR system: Just add a timer handler that checks how long a process has been active; if it exceeds the time slice, call the scheduler.

An open question is: What should the time slice (also called the time slot or the quantum) be? There are two adverse properties which make it non-trivial to make a decision:

- On the one hand, we would like the time slice as small as possible because that guarantees that each process in the ready queue will wait only a short amount of time until it can start running. That is an important property for interactive systems that want to give their users the feeling that “things happen instantaneously”. So this should lead to the rule: Make it short.

- On the other hand, after each time slice a context switch to the next process in the ready queue occurs (which is what we want). The downside is that this switch costs time. Let’s assume that the context switch takes $n$ units of time and that we have chosen the time slice to be the same $n$ units of time. As a consequence the CPU time would be equally distributed between all the processes and the scheduler, resulting in a setup that runs with only half the possible speed because the other half is wasted by the scheduler. Obviously that is bad, so the time slice should be much larger than the context switch time. Here we get the rule: Make it long.

The answer must be some kind of compromise between the two. For interactive systems there is one property that we might be able to observe and that helps us pick the right amount: it is the typical time required to service a user interaction (like a pressed key or mouse button).

We define the average interaction time as follows: Assume that a process is currently blocked because it waits for an I/O event (such as a key being pressed). Once the key is actually pressed, the keyboard handler will move the process to the ready queue. The next time this process runs it will evaluate the character that was read in, and will act on that information somehow. The consequence of the pressed key will become visible, for example, an editor will display the character and move the cursor position, another program might open a menu or perform some other action. Once this observable reaction has occurred we stop the clock: The time between the reactivation of the process and now is the interaction time for this specific interaction. Now make a collection of several representative programs (those that are typically run on the operating system) and for each such program a collection of the typical interactions. For all those interactions measure the interaction time and then calculate the average. Add a few percent to that value to be on the safe side and use that final value as the length of the RR time slice.

Figure 8.3 visualizes why this approach is helpful: It shows the treatment of one interaction by the RR scheduler; once with a time slice that was set as described above, once with a time slice that is just a bit too small. The yellow and green boxes represent the interactive process and a second process, and the striped black box shows the rest of a time slice which remained unused because the interactive process finished handling the interaction and blocked (waiting for the next key stroke). In the top part you can see the behavior that we want: The time slice is slightly larger than the interaction time which
8.1.6 Virtual Round Robin

There is one problem with the RR strategy: It is unfair to I/O-bound processes. (We define a process to be I/O-bound if it performs I/O very often and thusly uses only small parts of each of its time slices, quickly blocking again. The opposite is a CPU-bound process which typically uses up its time slice completely. Obviously there is a gray area between I/O-bound and CPU-bound where is process can be called neither.)

Back to the point: RR treats I/O-bound processes unfairly because they typically use just a tiny fraction of their time slice and then block. Whatever I/O activity they perform, we can assume it to take quite some time (for example, disk access is pretty slow in comparison to CPU instructions, and waiting for a key stroke takes an indefinite amount of time). Once the I/O has completed, RR adds the process to the end of the ready queue. Then it has to wait until all other processes that stand before it in the queue have either used up their time slices or blocked. If you compare the ratio between the needed CPU time (the
time that the process actually executes instructions on the CPU) with the wait time, then I/O-bound processes get a much smaller value than CPU-bound ones.

There is a modification of RR that alleviates this effect, and it is called \textit{Virtual Round Robin (VRR)}. It adds a second, privileged ready queue to the scheduler that only contains processes which had not fully used up their time slices when they ran the last time. If that queue is non-empty, a VRR scheduler will always pick a process from that priority queue when the current process’ time slice runs out. However, it will not grant the newly chosen process a full time slice but only the rest that was not used up the last time.

\subsection*{8.1.7 Priority-Based Scheduling}

We’ve already used the word “priority” when we discussed the extra queue of the VRR scheduler. Priority-based scheduling allows each process to be treated differently by making it more or less important than a process with default settings. This can be implemented in several ways. A priority-based scheduler might give an important process a longer time slice or it might pick it more often than other processes. Depending on how fine-grained priorities can be assigned to processes, there are several choices for handling the processes (see Figure 8.4):

- If the number of different priorities is low, we could manage a separate queue for each priority. In that case the scheduler would always start looking for the next process in the queue with the highest priority. Only if such a queue is empty it would look...
Scheduling
down to the next queue. This can lead to *starvation* when one of the higher queues
never empties, and there are mechanisms to solve that problem, such as increasing
the priority of processes which have been waiting too long.

- Alternatively, if there are too many different priorities and we don’t want the over-
  head of a corresponding queue collection, we can just store the priority as a numerical
  value in the process control block. Then the scheduler must search through the whole
  process table in order to find the (or one) process with the highest priority, and the
  queue is no longer a proper queue because the ordering in the queue is ignored by the
  scheduler (or only recognized if there are several processes with the highest priority).
  Again, such a system can lead to starvation and needs to provide a mechanism that
  prevents this.

Furthermore, priorities can be static or dynamic: A scheduler with *static priorities*
assigns a fixed priority to each process when it is created. It may be changed by a system
call, but otherwise it remains constant throughout the lifetime of the process. When the
scheduler uses *dynamic priorities* it regularly recalculates the priorities based on a set of
rules. For example, such a scheduler may punish or reward a process for some specific
observed behavior. Increasing the priority of processes which have been waiting for a
long time (in order to avoid starvation) is an example for the use of dynamic priorities.

On Unix systems priorities are often expressed with a *nice value*. This does sometimes
lead to confusion, because the “nicer” a process is, the lower is its priority. Unix provides a
*nice* system call that can set an integer value that is roughly in the interval $-20 \ldots 20$. We
write “roughly” because the exact values can differ from one Unix system to another; for
example on a Linux machine the values $-20$ to $19$ are valid, on OS X the range goes from
$-20$ to $20$. The nice value is used by each Unix system to calculate an internal priority,
and often it is not possible to set a process to the highest internal priority by changing its
nice value.

Ulix does not support priorities, but in Exercise 29 you can add that feature to the
kernel.

### 8.1.8 Multi-Level Scheduling

Multi-level schedulers combine the characteristics of two or more other scheduling strate-
gies. An example is a priority scheduler that uses three queues for standard, important
and urgent processes (like the one in Figure 8.4, top) and then handles each of the queues
like a Round Robin scheduler does.

A *Multi-Level Feedback Scheduler* for Ulix has been implemented by Markus Felsner
as part of his Bachelor’s thesis [Fel13] which is available online (in German). The code is
based on Chapter 8 of the textbook by Remzi H. and Andrea C. Arpaci-Dusseau [ADAD14].
8.2 Multiprocessor Scheduling

If a machine has more than one CPU (or the processor has several cores, even virtual ones via hyperthreading), then things get more complicated for the scheduler. The first question is: Where should it run?

It is possible to restrict the whole kernel to run on a single, dedicated CPU that performs all the tasks which require ring 0 permissions. That would include the scheduler which would be activated regularly (via a timer, as on monoprocessor systems), and it could look at the processes running on all the CPUs and decide which processor needs a context switch to a different process. Models like these are also called *Master-Slave-Scheduling* because the dedicated (master) processor controls all the other CPUs and distributes the workload. Such systems are useful for high-performance computing where machines sometimes have a large number of processors and applications require a specific numbers of CPUs to execute program threads simultaneously. In that case a scheduler will let the whole application (which needs to announce its processor requirements beforehand) wait until a sufficient number of slave processors becomes available and then create the requested number of threads and let them execute exclusively on the assigned CPUs. This is called *Gang Scheduling*.

The alternative is that all processors are equals. In that case the operating system will still boot from a single processor, but during initialization it will start copies of the scheduler on each CPU. Those copies could all use the same strategy to find a new process for their local CPU, but special care must be taken so that the process queues remain consistent. For example, if two copies of the scheduler simultaneously look at the front of the ready queue, pick the same process and activate it, that process would execute twice.

Locking and other synchronization tasks become more complex when several CPUs are involved, and there’s also the problem of *cache coherence*: In short, all CPUs have their own local cache and if those caches contain copies of the same memory region, then the system must make sure that changing the memory contents on one CPU invalidates the corresponding cache entries on all other processors.

Another important point that a multiprocessor scheduler must consider is the fact that moving a process from one processor to another one is costly because the old processor may still have parts of the process’ memory in its cache (which would speed up memory access for that process) whereas the new processor’s cache does not contain that memory. Modern operating systems provide the property *CPU affinity* that tells the scheduler to keep a process assigned to a CPU. However, the more constraints are added to a system (such as: process A must always run on CPU X), the harder it becomes to generate an equal load on all processors, and a load balancing algorithm is needed that may decide to move a process to a different CPU (with low load) even though this has some cost.

The short introduction to scheduling on multiprocessors shall suffice for this book since we focus on monoproceessors and *Unix* uses only one CPU as well.
8.3 Implementation of the Ulix Scheduler

Thanks to the forking mechanism of Section 6.5, we can already have more than one process in Ulix—but we still have to write code which lets Ulix switch between several tasks. This section is mainly about the task switch; the scheduling strategy that we use is Round Robin.

Understanding the switch basically means looking at the functions and stacks. When switching from process A to process B, we expect the following to happen:

1. Process A is executing, it runs in user mode, using its user mode stack.
2. A timer interrupt (IRQ 0) occurs. The CPU switches to kernel mode; this also switches the stack to the kernel stack. (Its address is stored in the TSS structure.) The CPU then jumps to the interrupt handler registered for interrupt 0 which does the following:

```
irq0:
push    byte 0
push    byte 32
jmp     irq_common_stub

define irq_common_stub:
pusha
push    ds
push    es
push    fs
push    gs
push    esp
    call    interrupt_handler
    pop     esp
    pop     gs
    pop     fs
    pop     es
    pop     ds
    push    esp
    add     esp, 8
    iret
```

So after pushing 0 (an empty error code) and 32 (that is 32+0, where 0 is the IRQ number) onto the stack, it saves all relevant registers on the stack and then calls `interrupt_handler`, which is a C function:

```
void interrupt_handler (context_t *r) {
    ...
    handler = interrupt_handlers[r->int_no - 32];
    if (handler != NULL) handler (r);
}
```

The generic `interrupt_handler` looks up the correct interrupt service routine for the timer (it calculates `r->int_no-32`, which in this case is `-32-32 = 0`, finds the entry in `interrupt_handlers` (that is `timer_handler`), and then calls it.

3. Next, the `timer_handler` checks whether it is time to call the scheduler and (if so) calls it:

```
if (system_ticks % 5 == 0) {
    ...
    scheduler (r, SCHED_SRC_TIMER);
```
8.3 Implementation of the ULIx Scheduler

The extra argument `SCHED_SRC_TIMER` tells the scheduler that it was called from the timer (which is the standard case).

4. So if it decides to call the scheduler, it enters

```c
void scheduler (context_t *r, int source) {
    ...
}
```

which is the function that we have to implement.

The sequence is similar if the interrupted process was not in user mode but in kernel mode (in most cases because it was executing a system call), but in that case there is no switch from ring 3 to ring 0 since the system is already running in kernel mode. That is reflected by slightly different stack contents since an `iret` (interrupt return) from the timer handler must restore the mode that the process was operating in.

### 8.3.1 Stack Usage

We need to keep track of which stacks are in use and what contents are stored on these stacks. Every process has a private user mode stack and a private kernel mode stack.

1. When the current process runs (in user mode) and a timer interrupt occurs at time $t = 0$, the CPU checks the Task State Segment (TSS) to find the current top of the stack for kernel mode: it is stored in the `ESP0` entry. (It also retrieves the new value for the `SS` register; remember that we have different code/data segment descriptors for user and kernel mode.) It switches to the new stack (by changing the `ESP` and `SS` registers) and pushes the old values of `SS` and `ESP` as well as the contents of `EFLAGS`, `CS` and `EIP` onto the new stack. `EIP` is already set to the address of the next instruction of the process: the one that will be executed once we return to the process. Then it starts executing the interrupt handler code.

The kernel stack now looks like in Figure 8.5.

```
<table>
<thead>
<tr>
<th>31</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>SS</td>
<td></td>
</tr>
<tr>
<td>ESP</td>
<td></td>
</tr>
<tr>
<td>EFLAGS</td>
<td></td>
</tr>
<tr>
<td>CS</td>
<td></td>
</tr>
<tr>
<td>EIP</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 8.5: At $t = 1$ the kernel stack contains these values.
2. Then, at time $t = 1$, the interrupt handler entry function `irq0` (for IRQ 0) pushes 0 and 32 onto the stack and jumps to `irq_common_stub` which pushes `EAX`, `ECX`, `EDX`, `EBX`, `ESP` ($t = 1$), `EBP`, `ESI`, `EDI`, `DS`, `ES`, `FS` and `GS`.

Finally, at time $t = 2$, it pushes the current `ESP` which is now properly set up as the address of the `context_t` structure holding all the registers (see Figure 8.6, also compare this to Figure 5.5 on page 143).

3. When `irq_handler` starts, it expects to find two values on the stack: the return address and one argument. It takes a `context_t` as argument and we have just prepared the stack so that it exactly fits this structure:

```c
typedef struct {
    uint gs, fs, es, ds;
    uint edi, esi, ebp, esp, ebx, edx, ecx, eax;
    uint int_no, err_code;
    uint eip, cs, eflags, useresp, ss;
} context_t;
```
The function can then access the elements of the context_t structure.

4. \texttt{irq_handler} calls the C function \texttt{timer_handler}, and passes the pointer to our context_t structure as its single argument.

5. Lastly, \texttt{timer_handler} calls \texttt{scheduler} (passing that pointer once more).

6. Now, \texttt{scheduler} is executing and it can access the register values which we’ve set up on the stack early in the assembler code and passed down all the way to the scheduler. Since we’ve only passed a pointer all the time (call by reference), the scheduler can modify the register values. After the whole call stack unwinds later and we’re back in \texttt{irq_common_stub}, the pop instructions will write the modified values into the appropriate registers. Note however that somewhere inside \texttt{scheduler} an address space change will occur that will also exchange the current kernel stack with the kernel stack that exists in the new address space which needs to be prepared so that the whole stack unwinding works as if no switch had occurred. That is why we took special care to create the stack properly in the implementation of the forking mechanism.

Let’s first assume that we do not enter the scheduler (because \texttt{system_ticks} % 5 \neq 0). In that case we just return and do not modify anything relevant to scheduling—we expect the current process to continue running, as it does after other interrupt treatments.

If we’ve just entered the \texttt{scheduler}, what does the stack look like right now? In comparison to Figure 8.6, there will be further return addresses and references to the context_t structure on the stack, since each function passes it, but we don’t really have to care because all the important information is available via the pointer \texttt{r} to the context_t.

Note again that at the beginning of the interrupt handling we stored the contents of all registers on the stack, in just the order which conveniently fits the context_t structure definition. We also pass the pointer to this structure to all further functions which get called (when calling \texttt{handler(r)} and then \texttt{scheduler} (\texttt{r})). So within the \texttt{scheduler} we can look at \texttt{r} to see the state as it was before the timer interrupt occurred.

Whether we’ve come from user mode or from kernel mode (e.g. from a process that was executing a system call when the timer interrupt fired) does not matter since all relevant registers have been saved and will be restored.

### 8.3.2 The Implementation

When we schedule, we select the new process and then store all registers (the data that \texttt{r} points to) in the old TCB. Then we switch the address space (the CPU’s pointer to the page directory), and then we load the new TCB contents in the registers. After that we can return.

Our scheduler has the prototype

\[
\texttt{void scheduler (context_t *r, int source);}
\]

which—as expected—takes a context_t pointer as its first argument. We introduce the second argument \texttt{source} because we also call the scheduler from within other kernel func-
scheduling. (Right now we only discuss how it is called from the timer handler, but you have already seen us calling it from syscall_resign which was needed by syscall_waitpid.)

The following two macros will allow the system to temporarily disable (and reenable) scheduling. During system initialization we’ve set scheduler_is_active to false; it is changed to true in the start_program_from_disk function which creates the first process.

```c
scheduler_is_active = true; _set_statusline ("SCH:ON ", 16);
```

Uses _set_statusline 337b and scheduler_is_active 276e.

```c
scheduler_is_active = false; _set_statusline ("SCH:OFF", 16);
```

Uses _set_statusline 337b and scheduler_is_active 276e.

We declare two global variables in the kernel address space which will later come in handy when we have to remember information about the current and next process:

```c
TCB *t_old, *t_new;
```

Defines:
- t_new, used in chunks 210b, 212, 255c, 257c, 277–80, 425a, and 567.
- t_old, used in chunks 210b, 255c, 257c, 277–79, and 425a.
Uses TCB 175.

These are not affected by changing the address space because they are in the region above 0xc0000000 which is identical in all address spaces. This is important! If those variables were defined locally in the scheduler function, they would reside in the kernel stack and get lost when the address space changes. Note that the scheduling code is a critical section, and there are two further exit points in the function at which we explicitly leave the critical section.

```c
void scheduler (context_t *r, int source) {
   (begin critical section in kernel 380a)
   (scheduler implementation 277a)
   (end critical section in kernel 380b)
   return;
}
```

Defines:
- scheduler, used in chunks 216b, 221a, 275, and 342d.
Uses context_t 142a.

Our implementation starts with checking whether there are any zombie processes and tries to get rid of them (see further down). Then it looks at the global variable

```c
int scheduler_is_active = false;
```

Defines:
- scheduler_is_active, used in chunks 206b, 276, 277a, 306d, 311a, 321, 329b, 334b, 335b, 412c, 416b, 509d, 510b, 512c, 518d, 521, 522, 531a, 532d, 545b, 588b, and 589a.

to determine whether it shall try to actually attempt a context switch.
With all obstacles removed, the real scheduling process can begin. The scheduler lets \( t_{old} \) point to the current process and then finds out which process it should switch to, storing the result in \( t_{new} \). Then it performs the context switch, and before returning to the new current process it checks whether that process has any pending signals and whether there is some clean-up to be done for terminated processes. (This last step could be handled elsewhere, for example via one of the \langle timer tasks \rangle.)

\[
\langle \text{scheduler implementation} \rangle \equiv \quad (276d) \quad 277b \quad [277a]
\]

\[
\langle \text{scheduler: check for zombies} \rangle \quad // \text{deal with zombies if we have any}
\]

\[
\text{if} \ (\text{scheduler is active}) \{ \quad // \text{check if we want to run the scheduler}
\]

\[
\langle \text{end critical section in kernel} \rangle
\]

\[
\text{return;}
\]

\[
\}
\]

Uses \text{scheduler is active} \, 276e.

We will implement the code chunk \langle scheduler: check pending signals \rangle in Chapter 14 where we introduce signals. Note that the \langle scheduler: find next process and set \( t_{new} \rangle chunk implements the scheduling logic, whereas all the other code is about technical details of the context switch.

### 8.3.2.1 A Simple Round-Robin Strategy

\text{Ulix} does not attempt any sophisticated scheduling strategy; we will just use a simple Round Robin system. The search for the next process will normally use the next pointer in the current TCB since it will point to the next TCB in the ready queue. However there’s a special case when the scheduler was activated because the current process called \text{waitpid}; in that case the former current process has already been moved to a blocked queue and we need to start over. We can recognize that case by evaluating the source argument and looking at the state of the current process. If it is not \text{TSTATE\_READY} then we must start over (with the first element of the ready queue which is stored in \text{tid} = \text{thread_table}[1].next).

We also might come across the idle process (which has the thread ID 1); if so we skip it in the search of “real” processes that need some work done. (We could have left that process completely out of any queues since it will never block but always be ready.)
Scheduling

\( \langle \text{scheduler: find next process and set } t_\text{new} \rangle \equiv \)

\[
\begin{align*}
\text{thread_id } & \text{ tid;} \\
\text{search: } & \text{ // goto label} \\
\text{if (source == SCHED_SRC_RESIGN &amp;&amp; t_old-&gt;state } &amp; \not= \text{TSTATE_READY) } \{ \\
& \quad \text{// we cannot use the } \rightarrow\text{next pointer} \\
& \quad \text{tid } = \text{thread_table[0].next;} \\
\} \text{ else } \{ \\
& \quad \text{tid } = \text{t_old-&gt;next;} \\
\} \\
& \quad \text{if (tid } = 1\text{) tid } = \text{thread_table[1].next;} \quad \text{// ignore idle process} \\
\end{align*}
\]

Uses SCHED_SRC_RESIGN 343a, t_old 276c, thread_id 178a, thread_table 176b, and TSTATE_READY 180a.

If tid is 0, we have reached the end of the queue—or the queue may be completely empty (in which case we activate the idle process):

\[
\begin{align*}
\text{if (tid } = 0\text{)} & \quad \text{// end of queue reached} \\
& \quad \text{tid } = \text{thread_table[1].next;} \\
\text{if (tid } = 0\text{)} & \quad \text{// still 0? run idle task} \\
& \quad \text{tid } = 1; \quad \text{// idle} \\
& \quad \text{t_new } = \&\text{thread_table[tid]}; \\
& \quad \text{if (tid } > 1\text{ &amp;&amp; (t_new-&gt;addr_space } = 0 \text{ || t_new-&gt;state } &amp; \not= \text{TSTATE_READY}) } \{ \\
& \quad & \quad \text{goto search; } \text{// continue searching} \\
& \quad \} \\
\end{align*}
\]

// found it
Uses t_new 276c, thread_table 176b, and TSTATE_READY 180a.

8.3.2.2 The Context Switch

Before implementing the actual context switch, let’s first observe the following facts:

- We only enter the scheduler (and thus also the context switcher) via timer interrupts or when a function resigns (i.e., freely gives up the CPU by calling resign) or calls waitpid.
- Once we’re running inside the scheduler, we know that the kernel stack has been set up in a way that will allow the system to continue operation of the interrupted process—whether it was running in user mode or kernel mode before the interrupt occurred.
- When we switch the address space, we also switch the kernel stack. However, the stack pointer register ESP will still point to the top of the old process’ kernel stack. We need to remedy that and have it point to the top of the new process’ kernel stack.
- If we switch between threads (within one process) we need not change the address space.

To make the code more readable, we provide some macros which copy values between variables and the ESP, EBP and CR3 registers. They require the use of inline assembler code (see Appendix B.4).
8.3 Implementation of the ULIX Scheduler

(\textit{macro definitions} 35a)+≡ \quad (44a) <222c 340a> \quad [279a]

\begin{verbatim}
#define COPY_VAR_TO_ESP(x) asm volatile("mov %0, %esp" : : "r"(x) )
#define COPY_VAR_TO_EBP(x) asm volatile("mov %0, %ebp" : : "r"(x) )
#define COPY_ESP_TO_VAR(x) asm volatile("mov %esp, %0" : =r"(x) )
#define COPY_EBP_TO_VAR(x) asm volatile("mov %ebp, %0" : =r"(x) )
#define WRITE_CR3(x) asm volatile("mov %0, %%cr3" : : "r"(x) )
\end{verbatim}

Defines:

\begin{itemize}
  \item COPY_EBP_TO_VAR, used in chunk 279c.
  \item COPY_ESP_TO_VAR, used in chunk 279c.
  \item COPY_VAR_TO_EBP, used in chunk 279c.
  \item COPY_VAR_TO_ESP, used in chunk 279c.
  \item WRITE_CR3, used in chunk 279.
\end{itemize}

Now we can present the code that handles the actual context switch. It is only executed if we truly switch, i.e., if \texttt{t_new} \neq \texttt{t_old}. In principle, we would expect something along the lines of

(\textit{scheduler: context switch (simplified version)} 279b)≡ \quad [279b]

\begin{verbatim}
t_old->regs = *r;  // store old: registers
COPY_ESP_TO_VAR (t_old->esp0);    // esp (kernel)
COPY_EBP_TO_VAR (t_old->ebp);    // ebp

current_task = t_new->tid;  // update values of current_{task,as,pd}
current_as = t_new->addr_space;
current_pd = address_spaces[t_new->addr_space].pd;
WRITE_CR3 ( mmu (0, current_pd ));  // activate address space

COPY_VAR_TO_ESP (t_new->esp0);    // restore new: esp
COPY_VAR_TO_EBP (t_new->ebp);    // ebp
*r = t_new->regs;    // registers
\end{verbatim}

but it is a little more complicated since we also have to check whether we do want to change the address space (if not, then we can save some CPU time by omitting that step), and we also need to change the TSS’s esp\textsubscript{0} entry and handle some special cases for newly created processes or threads. So the actual code is a bit more complex:

(\textit{scheduler: context switch} 279c)≡ \quad (277b) 280a> \quad [279c]

\begin{verbatim}
t_old->regs = *r;  // store old: registers
COPY_ESP_TO_VAR (t_old->esp0);    // esp (kernel)
COPY_EBP_TO_VAR (t_old->ebp);    // ebp
current_task = t_new->tid;
if (current_as != t_new->addr_space) {
    // we need to change the address space (switching process, not thread)
    current_as = t_new->addr_space;
    current_pd = address_spaces[t_new->addr_space].pd;
    WRITE_CR3 ( mmu (0, (memaddress)current_pd ));  // activate address space
}
COPY_VAR_TO_ESP (t_new->esp0);    // restore new: esp
COPY_VAR_TO_EBP (t_new->ebp);    // ebp
\end{verbatim}

\begin{itemize}
  \item Uses address_spaces 162b, COPY_EBP_TO_VAR 279a, COPY_ESP_TO_VAR 279a, COPY_VAR_TO_EBP 279a,
  \item COPY_VAR_TO_ESP 279a, current_as 170b, current_pd 105a, current_task 192c, memaddress 46c, mmu 172a,
  \item t_new 276c, t_old 276c, and WRITE_CR3 279a.
\end{itemize}
We must update the TSS structure and enter the address of the kernel stack; that is either \texttt{TOP_OF_KERNEL_MODE_STACK}\textsubscript{159c} (for a pure process or the primary thread of a multi-threaded process, i.e., one with \texttt{pid} \texttt{==} \texttt{tid}) or it is \texttt{t_new}\textsubscript{276c}->\texttt{top_of_thread_kstack} for a non-primary thread (\texttt{pid} \texttt{!=} \texttt{tid}):

\begin{verbatim}
[scheduler: context switch 279c]+\equiv
// set TSS entry esp0 to top of current kernel stack and flush TSS
if (t_new->pid != t_new->tid) {
  // thread kstack information is stored in the TCB
  write_tss (5, 0x10, t_new->top_of_thread_kstack);  // non-primary thread
} else {
  // process kstack is a fixed value
  write_tss (5, 0x10, (void*)TOP_OF_KERNEL_MODE_STACK);  // primary thread
}
tss_flush ();

// show thread ID in status line
if (t_new->tid != 1) {  // ignore switch to idle
  char msg[4]; sprintf (msg, "%03x", t_new->tid);
  _set_statusline (msg, 20);
}
\end{verbatim}

Uses \texttt{t_new}\textsubscript{276c}.

Remember: \texttt{write_tss}\textsubscript{197a} sets the ESP0 element of the TSS structure. It is only used when the CPU switches from ring 3 to ring 0 (in general: whenever it switches from ring 1–3 to ring 0, but ULIx does not use rings 1 and 2), and that means we can always start with an empty kernel stack in those situations: Thus we can always write the address of the stack’s top into that element.

Finally, we need to consider the special case of switching to a freshly created thread: On page 258 we have prepared the new thread so that we can immediately leave with the \texttt{iret} instruction:

\begin{verbatim}
[scheduler: switching to a fresh thread? 280b]+\equiv
*r = t_new->regs;  // restore new: registers
\end{verbatim}

Uses \texttt{_set_statusline} 337b, \texttt{kstack}, \texttt{sprintf} 601a, \texttt{t_new} 276c, \texttt{TCB} 175, \texttt{TOP_OF_KERNEL_MODE_STACK} 159c, \texttt{top_of_thread_kstack}, \texttt{tss_flush} 197c, and \texttt{write_tss} 197a.

In this case the rest of the \texttt{scheduler} function (the code chunks \texttt{⟨scheduler: check pending signals 567b⟩} and \texttt{⟨scheduler: free old kernel stacks 169a⟩}) is not executed, but that is no problem; we can handle that with the next invocation of the scheduler.
8.3 Implementation of the ULIX Scheduler

8.3.2.3 Treating Zombie Processes

We still need to check for zombies: A zombie is a terminated process whose parent process has not yet been able to retrieve the return value (supplied via exit). There are two possible cases:

1. The parent is waiting. This means that it has called waitpid after this process turned into a zombie, because otherwise it would not exist anymore. In that case we deblock the parent process and delete the zombie’s entry in the thread table.

2. The ppid ID of the zombie process was set to 1. That means that its true parent exited without calling waitfor. Here, we can also remove the zombie, and there is nothing else to do: No process is waiting (or will ever be) for that process.

```c
⟨scheduler: check for zombies⟩
for (thread_id pid = 0; pid < MAX_THREADS; pid++) {
  if (thread_table[pid].state == TSTATE_ZOMBIE) {
    thread_id ppid = thread_table[pid].ppid;

    // case 1: parent is waiting
    if ( (thread_table[ppid].state == TSTATE_WAITFOR) &&
         (thread_table[ppid].waitfor == pid) ) {
      deblock (ppid, &waitpid_queue);
      thread_table[pid].state = TSTATE_EXIT;
      thread_table[pid].used = false;
    }

    // case 2: parent ID was set to 1 (idle)
    if ( ppид == 1 ) {
      thread_table[pid].state = TSTATE_EXIT;
      thread_table[pid].used = false;
    }
  }
}
```

Uses deblock, MAX_THREADS, thread_id, thread_table, TSTATE_EXIT, TSTATE_WAITFOR, TSTATE_ZOMBIE, and waitpid_queue.

8.3.3 Letting the init Process Idle

As a last topic for this chapter we discuss the idle process with process ID 1. It starts as the init process, but once it has spawned some other processes, it will turn into the idle process. If it becomes active it shall do nothing. However, if we interpret “nothing” as an empty infinite loop (for(;;);) then the system will always actively spin in this loop whenever no other process is ready—that uses processor power.

There is a better way: The assembler instruction hlt (halt) can stop the CPU until the next interrupt occurs. We provide a system call that executes this instruction and use it in the idle process:
282

void syscall_idle (context_t *r);

#define __NR_idle 505

void syscall_idle (context_t *r) {
  asm ("hlt");
}

install_syscall_handler (__NR_idle, syscall_idle);

In the user mode library we add an idle function:

inline void idle () {
  syscall1 (__NR_idle);
}

When the system starts, start_program_from_disk loads the /init program from disk whose source code we have already shown on page 191. It starts the program /bin/login (via execv) which in turn launches some new processes (/bin/swapper, see page 311, and a few login processes that let users log in on the virtual consoles). When that is done it becomes the idle process via

for (;;) {
  idle (); // if we don't call idle, we will have 100% CPU usage
}
8.4 Exercises

26. Scheduling with the [Esc] Key

The tutorial/07/ folder contains a new version of the mini ULIx kernel which includes the fork() and schedule() functions. Again, it is a literate program (ulix.nw). However the scheduler is never called: Normally the timer handler would have to do that, but it is not part of that kernel version. In this exercise you introduce a manual scheduling that can be initiated by pressing the [Esc] key.

a) Look at the user mode program test.c whose machine code version is already part of the kernel sources. Its main() function creates a new process via fork(), afterwards parent and child write “P” (for parent) or “C” (child) on the screen in infinite loops:

```c
int main () {
    printf ("Hello - User Mode!\n");
    int pid = fork ();
    for (;;) {
        if (pid == 0) printf ("C"); // child
            else printf ("P"); // parent
    }
}
```

Start the ULIx version (using make and make run). You will see that the system displays only “V”s, the child process does not run. (If you compile the same program for Linux and run it, you see alternating sequences of “P”s and “C”s.)

b) The scheduler is implemented in the scheduler() function, just like it is in the real ULIx kernel. You can test it by adding a check for a pressed [Esc] key to the keyboard handler. That key has scan code 1. Locate the keyboard_handler() function and add the following test for scan code 1 right after it was read with inportb():

```c
if (s == 1) {
    scheduler (r, 0);
    return;
}
```

When you recompile and start the kernel, you will see “P”s again. But pressing [Esc] will switch to the child process, so that the output sequence changes to “C”s. Every further time you press [Esc], the process will switch again.

27. Calling the Scheduler from the Timer Handler

Now you add a timer handler which will regularly call the scheduler and thus automatize the multi-tasking.

a) Start with removing the scheduler call from the keyboard handler that you added in the previous exercise; you can simply turn it into a comment.
b) We also need the timer handler to update a global ticks variable so that we can see how long the system has been running. Define it like this:

```c
unsigned long int ticks
```
and initialize it to 0. (Which code chunk to you have to append to in order to declare and initialize variables?) Don’t add this code in earlier definitions of the chunks but create your own section at the end of the document where you put additions to the earlier chunk definitions.

Implement a function `timer_handler` with the same prototype that all the other interrupt handlers have, e.g., the keyboard handler. Its first task is to increment the `ticks` variable. Afterwards call the scheduler:

```c
scheduler (r, 0);
```

With these changes the timer is already functional, but you still have to register it and enable the timer interrupt. Write an addition to the code chunk `(kernel main: initialize system)` where you put appropriate function calls. (You can check how Ulix does that for the keyboard handler.) The interrupt number for the timer is defined as `IRQ_TIMER` (and has the value 0).

Compile and start the modified kernel. Now the output of “V”s und “S”s should switch automatically: Every time the hardware generates a timer interrupt, Ulix will switch between the two processes.

c) As we discussed in this chapter, task switches cost time. It makes sense to let a process run for a longer time before the scheduler takes away the CPU and gives it to another process. You can achieve this behavior by only calling the scheduler if `ticks` is some multiple of a given number, e.g. 5. With

```c
(ticks % 5 == 0)
```

you can check whether `ticks` is a multiple of 5. Call the scheduler only when that condition evaluates to true. That reduces the amount of scheduler invocations (per time) to a fifth of what it was before. You can also test how different values (e.g. 25, 100) change the overall behavior.

d) The current code in `scheduler()` supports exactly two processes which use the TCBs 1 and 2. That makes it rather simple to determine which process comes next. How would you have to modify the function so that it supports exactly four processes instead of two?

---

**Tutorial 8** Until now all exercises used a stripped-down version of the Ulix kernel which evolved from exercise to exercise, mimicking the progress throughout the book. For the following two tasks you will work with real kernel sources. This is not the version that is presented in the book (because the exercises were developed before the Ulix implementation was complete), but it is a usable system with full user mode and filesystem support. You will now add new capabilities to the system.

Use the code that you find in the `/home/ulix/ulix/` folder for the following two exercises.
28. **Boost: Run Only This Process**

The scheduler which is called by `timer_handler` every five ticks chooses the next pro-
cess in the ready queue. In this exercise you give a process a chance to prevent this
switch (so that it can go on longer). By using the function

```c
void boost (int n);
```

it shall be able to set a global variable (global in the Ulx kernel). The timer handler
shall then use one of the ⟨timer tasks 306d⟩ to check whether this value is 0. If not, it
decrements the variable and performs no context switch.

a) Declare a global variable `int boost_count` and initialize it to 0. As in the last
exercise you need to find the appropriate code chunks for these two actions.

b) Move the ⟨timer tasks 306d⟩ code chunk that calls `scheduler()` into your own sec-
section of the literate program file so that you can document the changes to the
chunk. It is this chunk:

```c
<<timer tasks>>
// Every 5 clocks call the scheduler
if (system_ticks % 5 == 0) {
    [...]
    scheduler (r, SCHED_SRC_TIMER); // defined in the process chapter
    [...]
}
```

c) Modify the if clause; the extra condition `boost_count == 0` must also be fulfilled.
You will also need to add an else case which decrements `boost_count`.

d) Write a syscall handler

```c
void syscall_boost (context_t *r);
```

that reads the right processor register (which one is that?) and copies its value to
`boost_count` (if it is positive or 0). Define a syscall number constant `__NR_boost` (us-
ning a syscall number that is not yet in use) and add an `install_syscall_handler201b()`
call to the right code chunk.

e) Write a user mode library function

```c
void boost (int n);
```

that uses `syscall203c()` to make the system call. You can look at the implemen-
tation of `open()` to see how that can be done.

f) Finally, write a small user mode application that lets you test the effect of calling
`boost()`.

(Note that you need not implement a corresponding `u_boost` function which gets
called by `syscall_boost`: It would just contain the one line

```c
void u_boost (int n) { boost_count = n; }
```

so there is no point in writing an extra function for that task.)
29. **Process Prioritization**

All Unix processes (or threads) receive equal treatment by the Unix scheduler because the system does not know priorities. In this exercise you add the classical priority mechanism present in all Unix systems.

a) Add a nice value (int nice) to the thread control block that will hold values between $-20$ and 19, the default value is 0. If a process forks, it will pass the nice value on to the child process.

b) Implement a function int u_setpriority (int nice) that can be used for changing the current process’ nice value to the supplied value. You will also need corresponding functions syscall_setpriority in the kernel and int setpriority (int nice) in the user mode library. (The return value of u_setpriority or setpriority is the new nice value.) The prototype differs from the setpriority() function on Linux systems, but looking at the man page on a Linux box can still be helpful.

c) In the ⟨timer tasks⟩ chunk, modify the code block

```c
if (system_ticks % 5 == 0) {
    [...
    scheduler (r, SCHED_SRC_TIMER); // defined in the process chapter
    [...
```

so that the decision whether the scheduler is called depends on the current process’ nice value (that you can find in thread_table[current_task].nice). You have several options for doing this, but the result should be that a process with a lower nice value receives a longer time slice than a process with a larger nice value.

d) Write a test program that lets you check whether changing the nice values really changes the behavior of the process. You will need at least two processes (one that calls setpriority and one that does not) in order to observe any effect.
Recall what happens every time the machine accesses a memory address, either for receiving the next instruction or for reading or storing data: the address that the processor asks for is a virtual address, and it must first be translated to a physical address by the MMU. Special register CR3 tells the MMU what page directory to use, and that directory reveals the location of a page table which is in turn accessed to (finally) find the physical frame number and calculate the complete physical address by adding the offset (within the page).

In many cases this procedure will fail at some point, typically because either the page directory or the page table contains an entry which tells the MMU that the page does not exist in memory. This means that address translation cannot continue, and the CPU raises a page fault.

Every operating system needs to handle such page faults, the minimum action that is required is either killing a process (which has tried to access an illegal address) or halting the operating system (if the faulting instruction occurred inside code which was not executed on behalf of some process). But we expect our fault handler to be better than that and also handle the following situations:

- **User mode stack grows**: When a user mode program uses recursion or makes extensive use of the stack for other reasons, it will soon cross into a memory range just below the reserved stack space and cause a page fault. In that case the process can rightly expect the stack to grow automatically. Thus, Ulx will check whether a memory area just below the current end of the stack was accessed—and if so, increase the stack by one page. Afterwards, execution of the process can resume.
• Access to page which was paged out: In Sections 9.2 ff. we will introduce a swap file¹ and show code for moving pages of memory to disk and back. That will become necessary when the whole physical memory is in use and there are no further free page frames.

9.1 The Ulix Page Fault Handler

In this section we present the page fault handler that we implemented for Ulix. The handler function has the prototype of all other fault handlers:

\[
\text{function prototypes 45a}\]  
\begin{verbatim}
void page_fault_handler (context_t *regs);
\end{verbatim}

and we originally started by taking code from James Molloy’s kernel tutorial [Mol08, Ch. 6.4.5] (which just gives some information about the faulting reason and address) and then added some features which make it usable in Ulix.

The CR2 register holds the virtual address which caused the page fault, and the err_code element of the context tells us more about why the access to this address failed, so this is the first information we need to gather. Individual bits of err_code describe whether the page that was accessed is present, read-only or only accessible in kernel mode.

Then we check the possible reasons for a page fault and handle them as well as we can:

• First, if the page was paged out to disk (see next section), we bring the page back in. In that case we can simply leave the fault handler with return, and the running process will resume its operation, repeating the instruction that caused the fault at the first attempt. (Actually return jumps back into the generic fault_handler function which then returns to fault_common_stub in start.asm, and the transition back to the process occurs in the assembler code via the iret instruction.)

• Then we check whether the process tried to access an address directly below the user mode stack. That will often occur when a function calls itself recursively. In that case we increase the stack and also return.

These two conditions are recoverable, but there are other situations in which we either have to kill the faulting process or—worst case—permanently disable user mode and jump to a safe kernel function such as the kernel shell that we have included for debugging purposes.

• If a process tried to access an invalid address, and we can neither bring it back by paging in a paged-out page nor can we help the situation by increasing the user mode stack, then it was likely caused by a programming error and we need to kill the process. We check that condition by testing whether the faulting instruction’s address

¹ Note that we use the term “swap file” and not “page file”. Ulix does not implement swapping which is an older technique where whole processes are removed from memory. Instead we move single pages to disk and back, but historically a file or a partition used for swapping was called swap file/swap partition, and that name lives on in paging systems, e. g. in Linux and other Unix-like systems.
is below the kernel’s memory address range \((r->eip < 0xc0000000)\). In that case the system can continue running, minus the killed process.

- Last, we must assume that an error in the kernel caused the page fault. Then there’s nothing to do since the failed instruction is part of the kernel code and there is no simple way to resume after such a problem. We might try to write a memory dump to disk or provide some other means that might help debugging the code, but here we just jump to the kernel shell (from which we cannot return to user mode anymore). If that kernel shell was not available, we would simply halt the machine completely.

Our implementation of the page fault handler

\[
\begin{align*}
\text{void page_fault_handler (context_t *r) } & \text{ \{} \\
\text{ } & \text{\{page fault handler implementation\}} \\
\text{\}}
\end{align*}
\]

\text{Defines: page_fault_handler, used in chunks 151c and 288.}
\text{Uses context_t 142a.}

starts with gathering all the available information:

\[
\begin{align*}
\text{int present} & = !((r->err_code & 0x1)) \text{ // page present?} \\
\text{int rw} & = r->err_code & 0x2; \text{ // attempted to write?} \\
\text{int us} & = r->err_code & 0x4; \text{ // CPU in user-mode (ring 3)?} \\
\text{int reserved} & = r->err_code & 0x8; \text{ // overwritten CPU-reserved bits of page entry?} \\
\text{int id} & = r->err_code & 0x10; \text{ // caused by an instruction fetch?}
\end{align*}
\]

\text{Uses memaddress 46c.}

In the last line from above it checks whether the page was paged out to disk; we will describe that in the following section where we introduce the swap file.

The second benign case of a page fault occurs when the user mode stack needs to grow. We can check that condition by looking at the current end of the user mode stack and calculating whether adding one extra page would solve the problem—if so, we give the process that extra page. Otherwise (if the address is too far away from the stack) we cannot help this process:

\[
\begin{align*}
\text{if } & (\text{faulting_address } \leq \text{TOP_OF_USER_MODE_STACK} \&\& \\
& \text{faulting_address } \geq \\
& \text{TOP_OF_USER_MODE_STACK-address_spaces[current_as].stacksize } - \text{PAGE_SIZE}) \{ \\
\text{\{page fault handler: enlarge user mode stack\}} & \text{ // user mode, stack} \\
\text{return;}
\}
\end{align*}
\]

\text{Uses address_spaces 162b, current_as 170b, PAGE_SIZE 112a, and TOP_OF_USER_MODE_STACK 159b.}
Note that this restricts user mode processes in the way they use the stack. For example, you cannot write a function that takes so many (or so large) arguments that its invocation would increase the stack by more than one page. So depending on the kinds of applications you want to run on Ulix, you might want to modify the above code chunk.

If neither of the first two cases is applicable, then there’s a real problem. With the rest of the code we try to determine how big that problem is: It may be sufficient to kill the current process, or we may have to halt the system. First we write an error message to the screen. Here, we also use the old code chunk ⟨fault handler: display status information 152a⟩ from the generic fault handler that we implemented earlier in Chapter 5.3.

The same chapter also contains the ⟨fault handler: terminate process 152b⟩ chunk that we recycle when we decide to kill the current process. We do that if the faulting address is in the user space of virtual memory, i.e., below 0xc0000000. The old code chunk simply removes the process from the ready queue and calls syscall_exitb. The exit system call handler will then return the process’ resources and launch the scheduler which finally picks another process.

Uses current_as 170b, current_task 192c, hexdump 612c, memaddress 46c, and printf 601a.

(If you are curious why the string "read-only" is split into "re" "ad-only" in the above chunk, read the footnote.)²

Finally, if the page fault was not caused by a process that tried to access an illegal address, then we must assume we’ve come across a kernel bug. There’s no way to recover, because where should the system continue execution? Our last remaining option is to stop the system. Ulix provides a kernel mode shell that can be used for debugging, instead of a real full stop we jump into that function, but the user mode is gone for good.

Uses kernel_shell 610a and printf 601a.

² Sometimes we need to outwit the automatic cross-reference of the literate programming software. For example, it would detect read and misinterpret it as a reference to the read function and thus add an entry to the “Uses:” block. We avoid this by either splitting strings or, in case of comments, adding an underscore: read_ will not be (mis-)detected as read.
9.1.1 Enlarging the Stack

When we notice that the stack’s size is causing the problem, we can grow it. In the above code detection of necessary stack growth uses the fact that the stack grows linearly; we assume that an illegal access to the stack (i.e., to a page that has not been mapped to a frame yet) always occurs directly below the last valid stack page. So, the address has to be below the top of the stack, but above the lowest valid address minus PAGE_SIZE.

We can then simply grow the stack by mapping the next page (remembering that the stack grows downwards) to a fresh frame:

\[
\text{memaddress new_stack} = \text{TOP_OF_USER_MODE_STACK}; \\
\text{new_stack} -= \text{address_spaces[current_as].stacksize}; \\
\text{int pageno} = \text{new_stack} / \text{PAGE_SIZE} - 1; \\
\text{int frameno}; \\
\text{if} \!\! \!\! (\text{frameno} = \text{request_new_frame}()) < 0 \!\!\!\! \{ \\
\text{printf} ("\nERROR: no free frame, cannot grow user mode stack\n"); // error \\
\} \\
\text{as_map_page_to_frame (current_as, pageno, frameno);} // update page table and \\
\text{address_spaces[current_as].stacksize} += \text{PAGE_SIZE}; // TCB stack size entry
\]

Uses address_spaces 162b, as_map_page_to_frame 165b, current_as 170b, memaddress 46c, PAGE_SIZE 112a, printf 601a, request_new_frame 118b, TCB 175, and TOP_OF_USER_MODE_STACK 159b.

Note that this code only works for a thread-less process. If a process consists of several threads, and one of the non-primary threads exceeds its user mode stack, this code will not be executed because it only checks for problems with the primary thread’s stack. Our design does not allow growable stacks for the extra stacks because we have placed those stacks close to each other in virtual memory. We could increase the free spaces between thread stacks to make the problem a little smaller, but in the end there must always be a limit to the threads’ stack sizes—after all we do not know beforehand how many threads a process is going to create.

The U\text{\textregistered}X disk provides

- a fault-mem application which accesses an illegal address (and will subsequently be killed) and
- a recurse program that recursively calls a function (and thus forces stack growth): In early versions of the U\text{\textregistered}X kernel it would eventually run out of memory and then be killed; with the final U\text{\textregistered}X code it can go on rather long because the kernel will start paging out memory in order to free frames.
- There is also a tp program that explicitly pages out a page of its memory and then accesses it (so that it will be brought back in; see the next sections).
### 9.2 The Swap File

ULIX uses a 64 MByte swap file which is stored on the hard disk. This is also just a little less than the maximum file size that our Minix filesystem implementation supports (see Chapter 12): With six direct block addresses, one single indirect block (leading to 256 blocks) and one double indirect block (leading to $256 \times 256$ blocks) as well as a block size of 1 KByte, files can be no larger than 65798 KByte $\approx 64.26$ MByte.

The swap file stores only the contents of pages, all administrative data is kept in memory. 64 MByte allow ULIX to double the available RAM (since it works with a fixed amount of 64 MByte of RAM as well), providing up to 128 MByte of virtual memory to the system and processes.

Internally we will store information about pages which have been written to disk. For each such page we need to know the address space ID and the page number, thus an internal paging record has the following form:

```c
struct paging_entry {
    int as : 10;  // 10 bits for address space, values from [0..1023]
    int pageno : 20;  // 20 bits for the page number
    int used : 1;  // 2 bits for two flags
    int reserved : 1;
} __attribute__((packed));
```

This is just small enough to fit in a 32-bit integer. As the page size is 4 KByte, we need $64 \text{ MByte} / 4 \text{ KByte} = 16384$ such entries:

```c
#define MAX_SWAP_FRAMES 16384
```

If `paging[i].used` is 0 (false), the corresponding swap file entry $i$ is free which fits our initialization of the data structure.

We assume that a swap file `/tmp/swap` of size 64 MByte already exists.
During the system initialization we open this file and keep it open throughout the whole system runtime.

```c
int swap_fd;
```

Defines:

- `swap_fd`, used in chunks 293 and 294.

```c
swap_fd = u_open("/tmp/swap", O_RDWR, 0);
```

We provide two simple functions which write a page to the file and read it back in. Both need to walk through (parts of) the paging array in order to find out whether the page is (already) on the disk and which free entry can be used if that is not the case.

```c
int write_swap_page (int as, int pageno, int frameno);
int read_swap_page (int as, int pageno, int frameno);
```

We give these functions an extra argument `frameno`: If we already know the physical address where a page is stored in memory, we will provide its frame number so that the functions need not calculate it. (Actually we’re not using the feature where `write_swap_page` or `read_swap_page` would manually calculate the frame number. However it would be useful for an enhanced paging mechanism that might page out pages but still keep them in memory as well or page in pages from the swap file and still keep them on the disk. We do neither in our implementation.)

We do not expect these functions to alter a page table of the involved process—that happens in the functions `page_out` and `page_in` which we present in the next section.
if (index == -1 && free_index == -1) return -1; // not found + no free space
if (index == -1 && free_index != -1) {
    index = free_index; // create new entry
    paging[index].used = true;
    paging[index].as = as;
    paging[index].pageno = pageno;
}
// note: if (index != -1) we do not modify paging[]; this is an update

// write to disk
u_lseek (swap_fd, index*PAGE_SIZE, SEEK_SET);
u_write (swap_fd, (char*)PHYSICAL(frameno*PAGE_SIZE), PAGE_SIZE);
return 0; // success
}

Defines:
write_swap_page, used in chunks 293c and 296.
Uses MAX_SWAP_FRAMES 292b, mmu_p 171c, PAGE_SIZE 112a, paging 292c, PHYSICAL 116a, SEEK_SET 469b, swap_fd 293a, u_lseek 418a, and u_write 415a.

Note that u_write 415a will use the buffer cache (see Chapter 13.3), thus calling the function write_swap_page 293d may at first only result in copying a page to a different memory area.

Reading a page back in is simpler because we need not distinguish between updates and initial write operations: When we try to read, the page is either there or it is not. We do not check the case that a requested page might be missing in the swap file because we only call this function when we know that the page must be there.

int read_swap_page (int as, int pageno, int frameno) {
    // get frame number, if it was not supplied
    if (frameno == -1) frameno = mmu_p (as, pageno);
    if (frameno == -1) return -1; // error: page not available

    int index = -1; // get index
    for (int i = 0; i < MAX_SWAP_FRAMES; i++) {
        if (paging[i].used && paging[i].as == as && paging[i].pageno == pageno) {
            index = i; // found the entry!
            break;
        }
    }
    u_lseek (swap_fd, index*PAGE_SIZE, SEEK_SET);
    u_read (swap_fd, (char*)PHYSICAL(frameno*PAGE_SIZE), PAGE_SIZE);
    return 0; // success
}

Defines:
read_swap_page, used in chunk 297.
Uses MAX_SWAP_FRAMES 292b, mmu_p 171c, PAGE_SIZE 112a, paging 292c, PHYSICAL 116a, read 429b, SEEK_SET 469b, swap_fd 293a, u_lseek 418a, and u_read 414b.
9.2.1 Paging Out and In

The `write_swap_page` and `read_swap_page` functions simply copy a page frame from memory to the swap file or vice versa. But real paging requires more than that: we need to modify a page table whenever we remove or add a page. This is what the two functions are for. Some other kernel function (which we will describe soon) makes the decision to remove page X of process Y and then calls `page_out` which in return saves the page (via `write_swap_page`) and updates the relevant process’ page table to indicate that the page is no longer in RAM (but could be gotten from the swap file). When that process tries to access the page the next time, a page fault will occur which must be handled by the page fault handler which brings the needed page back in and lets the process reattempt the last instruction.

Let us first recall the data structure `page_desc` for the page descriptor:

```
typedef struct {
    unsigned int present : 1; // 0
    unsigned int writeable : 1; // 1
    unsigned int user_accessible : 1; // 2
    unsigned int pwt : 1; // 3
    unsigned int pcd : 1; // 4
    unsigned int accessed : 1; // 5
    unsigned int dirty : 1; // 6
    unsigned int zeroes : 2; // 8.. 7
    unsigned int unused_bits : 3; // 11.. 9
    unsigned int frame_addr : 20; // 31..12
} page_desc;
```

If the present bit is set to 0, any attempt to access the page will lead to a page fault. Thus, when we want to page out a page, we can simply reset the page descriptor’s present bit. But how is the fault handler to know whether a “genuine” page fault has occurred (i.e., the page does not exist at all) or whether the kernel paged out the page and is capable of getting it back? The bits 9–11 of the descriptor, called `unused_bits` above, are completely ignored by the MMU. This is our starting point: We use one of these three bits for keeping the paged out state:

```
typedef struct {
    unsigned int present : 1; // 0
    unsigned int writeable : 1; // 1
    unsigned int user_accessible : 1; // 2
    unsigned int pwt : 1; // 3
    unsigned int pcd : 1; // 4
    unsigned int accessed : 1; // 5
    unsigned int dirty : 1; // 6
    unsigned int zeroes : 2; // 8.. 7
    unsigned int unused_bits : 3; // 11.. 9
    unsigned int frame_addr : 20; // 31..12
} page_desc;
```
unsigned int zeroes = 2; // 8..7
unsigned int paged_out = 1; // 9 <- new
unsigned int unused_bits = 2; // 11..10
unsigned int frame_addr = 20; // 31..12
}

new_page_desc;

Defines:
new_page_desc, used in chunks 296 and 297.

Since page_desc and new_page_desc have the same layout, we can cast them into one another without corrupting data. So when we want to find out whether the new paged_out bit is set, we can check that with if ( ((new_page_desc*)pd)->paged_out ) { ... }.

Now we can present the page_out and page_in functions which perform the same calculations as the mmu_p function which we introduced earlier. page_out does four things:

- it calls write_swap_page to do the transfer from memory to disk,
- it resets the page descriptor’s present bit and sets its paged_out bit,
- it invalidates the TLB entry with the invlpg instruction in order to make sure that the next access to this page will cause a page fault (instead of accessing the old frame which may no longer hold the page) [Int08, p. 21] (or [Int11, p. 4-56-4.57]),
- and it releases the frame, thus increasing the free physical memory.

[296] \( \langle \text{function implementations} \rangle + \equiv (44a) \langle 294 \ 297 \rangle \)

int page_out (int as, int pageno) {
    uint pdindex = pageno/1024; uint ptindex = pageno%1024;
    page_directory *pd = address_spaces[as].pd;
    if ( ! pd->ptds[pdindex].present ) {
        return -1; // fail: page table not found
    } else {
        page_table *pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
        if ( pt->pds[ptindex].present ) { // found the page
            new_page_desc *pdesc = (new_page_desc*) &pt->pds[ptindex];
            int frameno = pdesc->frame_addr;
            write_swap_page (as, pageno, frameno); // write to swap file
            pdesc->present = false; // mark page non-present
            pdesc->paged_out = true; // mark page paged-out
            asm volatile ("invlpg %0" : : "m"(*char*)(pageno<<12)) ;
            release_frame (frameno); // mark phys. frame as free
            return 0; // success
        } else {
            return -1; // fail: page not found
        }
    }
}

Defines:
page_out, used in chunks 295a, 299a, and 308c.
Uses address_spaces 162b, new_page_desc 295c, page_directory 103d, page_table 101b, PHYSICAL 116a, release_frame 119b, and write_swap_page 293d.
The \texttt{page\_in} function expects that we try to page in a page which was paged out with \texttt{page\_out} earlier. That is, the page descriptor must exist and have its \texttt{paged\_out} bit set. It will then do the following three things:

\begin{itemize}
\item it reserves a new physical frame which will soon hold the page and writes its address to the page descriptor,
\item it calls \texttt{read\_swap\_page} to do the transfer from disk to memory,
\item and it resets the page descriptor's \texttt{paged\_out} bit,
\end{itemize}

\begin{withcode}
\textbf{function implementations}\quad\textbf{100b)\textsuperscript{+}}\equiv \quad (44a) <296 319d> [297]
\begin{verbatim}
int page_in (int as, int pageno) {
    uint pdindex = pageno/1024;
    uint ptindex = pageno%1024;
    page_directory *pd = address_spaces[as].pd;
    if (!pd->ptds[pdindex].present ) {
        printf ("DEBUG: page_in: page table not present\n");
        return -1; \quad \text{\textit{\negmedspace fail: page table not found}}
    } else {
        page_table *pt = (page_table*) PHYSICAL(pd->ptds[pdindex].frame_addr << 12);
        if (!pt->pds[ptindex].present ) {
            \textit{\texttt{found the page descriptor}}
            new_page_desc *pdesc = (new_page_desc*) &pt->pds[ptindex];
            if (!pdesc->paged_out) {
                printf ("DEBUG: page_in: page 0x%0x not marked paged out!\n", pageno);
                return -1; \quad \text{\textit{\negmedspace fail: page was not paged out}}
            }
            int frameno = request_new_frame (); \quad \text{\textit{\negmedspace reserve a phys. frame}}
            if (frameno == -1) return -1; \quad \text{\textit{\negmedspace fail: no free memory}}
            \textbf{read_swap_page} (as, pageno, frameno); \quad \text{\textit{\negmedspace read from swap file}}
            pdesc->present = true; \quad \text{\textit{\negmedspace mark page present}}
            pdesc->paged_out = false; \quad \text{\textit{\negmedspace mark page not paged-out}}
            pdesc->frame_addr = frameno; \quad \text{\textit{\negmedspace write new phys. frame number}}
            // \texttt{asm volatile}\n            \begin{verbatim}
            __asm__ volatile ("invlpg %0" : : "m"(*char*)(pageno<<12)) ; \quad \text{\textit{\negmedspace not needed}}
            \end{verbatim}
            // \textit{\negmedspace success}
            return 0; \quad \text{\textit{\negmedspace success}}
        } else {
            printf ("DEBUG: page_in: page not found\n");
            return -1; \quad \text{\textit{\negmedspace fail: page not found}}
        }
    }
}
\end{verbatim}
\end{withcode}

Defines: \texttt{page\_in}, used in chunk 298a.
Uses \texttt{address\_spaces 162b, new\_page\_desc 295c, page\_directory 103d, page\_table 101b, PHYSICAL 116a, printf 601a, read\_swap\_page 294, and request\_new\_frame 118b.}

When we page in, we need not invalidate a TLB entry since it does not store information about non-present pages [Int08, p. 21] (or [Int11, p. 4-58]):
If a paging-structure entry is modified to transition the present bit from 0 to 1, no invalidation is necessary. This is because no TLB entry or paging-structure cache entry will be created with information from a paging-structure entry that is marked “not present”. (If it is also the case that no invalidation was performed the last time the present bit was transitioned from 1 to 0, the processor may use a TLB entry or paging-structure cache entry that was created when the present bit had earlier been 1.)

This is all the code we need for paging in and out a page. Now we need to decide when to page out a page and how to pick that page.

### 9.2.2 Letting the Page Fault Handler Page In a Page

We can now add the missing code chunk ⟨page fault handler: check if page was paged out 298a⟩. Recall that the faulting address is stored in faulting_address. All we need to do here is attempt to page in the corresponding page (faulting_address / PAGE_SIZE)—if we are successful we can leave the page fault handler, and the process will re-execute the last instruction.

```
[298a]
⟨page fault handler: check if page was paged out 298a⟩≡
    int pageno = faulting_address / PAGE_SIZE;
    if (page_in (current_as, pageno) == 0) {
        return;   // success, leave fault handler
    }
```

Uses current_as 170b, page_in 297, and PAGE_SIZE 112a.

### 9.2.3 Testing

At this step of the implementation task we should check whether our code works as intended. For that purpose we will provide a temporary system call with which a process can force the kernel to page out a page. (Note that in general it is a bad idea to allow processes to take that kind of control over the memory management.)

A test program will then try to access data in the paged-out page which should in turn have the page fault handler bring the page back. We know that we’re successful if the program executes without errors.

```
[298b]
⟨public constants 46a⟩≡
#define __NR_page_out 508
```

Defines: __NR_page_out, used in chunk 299.

```
[298c]
⟨syscall prototypes 173b⟩≡
void syscall_page_out (context_t *r);
```

(44a 48a) <282b 315> (44a 48a) <282b 315> (44a 48a) <282b 315> (44a 48a) <282b 315> (44a 48a) <282b 315>
9.2 The Swap File

```c
(void) syscall_page_out (context_t *r) {
    // ebx: page number
    eax_return (page_out (current_as, r->ebx));
}
```

Defines:
- syscall_page_out, used in chunks 298c and 299b.

Uses context_t 142a, current_as 170b, eax_return 174a, and page_out 296.

```c
install_syscall_handler (__NR_page_out, syscall_page_out);
```

Uses __NR_page_out 298b, install_syscall_handler 201b, and syscall_page_out 299a.

The user mode program will use the following library function

```c
int lib_page_out (int pageno);
```

which just makes the system call

```c
int lib_page_out (int pageno) { return syscall2 (__NR_page_out, pageno); }
```

Defines:
- lib_page_out, used in chunk 299c.

Uses __NR_page_out 298b and syscall2 203c.

Here is the code for a simple test program:

```c
#include "../ulixlib.h"
char test[4096] __attribute__((aligned (4096)));

int main () {
    printf("Testing paging\n");
    test[5] = 'X';
    unsigned int address = (unsigned int)&test[5];
    printf("test[5] = \"%c\", address = 0x%x\n", test[5], address);
    lib_page_out (address >> 12);
    printf("test[5] = \"%c\", address = 0x%x\n", test[5], address);
    exit (0);
}
```

Now we need to discuss when the operating system should page out a page and which page it should choose. We enter the realm of page replacement strategies—the topic of the following section.
9.3 Page Replacement Strategy

When memory gets full, eventually the system will have to move pages to the disk in order to make room for other processes’ memory demands. Paging out a page (i.e., writing it to disk and releasing the page frame that held the page) and assigning a different process’ page to this page frame is called page replacement. Paging the information in and out of main memory is extremely simple because of the fixed size data chunks—as you have seen in the implementation of `page_in` and `page_out`.

Access to secondary storage is very slow, while access to main memory is rather fast. At time of writing, good hard disks have an average access time of about eight milliseconds, main memory of about eight nanoseconds. This is a difference of $10^6$, i.e., six orders of magnitude. To make this huge difference more evident, assume that access to main memory needs one second. Then the access to secondary storage would have to take $10^6$ seconds, which is roughly 11.5 days, to stay in the same relation.

Well-tuned paging systems can achieve a performance which is very close to the speed of main memory. The decisive parameter is the probability $p$ of not finding the requested information in RAM (i.e., the probability of a page fault). Given that $t_{mm}$ is the time necessary to access main memory and $t_{pf}$ the time to handle a page fault, the average time $t_{vm}$ to access virtual memory using a paging system is:

$$t_{vm} = (1 - p) \cdot t_{mm} + p \cdot t_{pf}$$

Since $t_{pf}$ is dominated by the access time to secondary storage, we need to keep $p$ as low as possible.

The algorithm that decides which page to page out is called a page replacement algorithm, and it implements a page replacement strategy. The chosen strategy is a part of the memory management system’s design, and several choices are available.

One possible choice would be a random selection: Whenever there is need for a free page frame (and none available) just pick any odd page frame and page out its contents. This strategy would not be much good, but we can think of even worse ones, e.g. always pick the very first page frame in the RAM.

The selection process has no consequences on the overall functioning of memory management: Even the worst strategy (and “pick the first page frame” is a good candidate for that) will lead to a working memory management system. However, the selection process decides how efficiently the resulting system behaves.

Before going into details, let us note that there is no direct equivalent to page replacement in filesystems—unless you had another layer of the memory hierarchy that is above disk access, e.g. an automatic tape backup system with a tape robot that can write files to a tape and delete them on disk when disk space gets low. If you had such a setup, you would move from a CPU–cache–RAM–disk memory hierarchy to a CPU–cache–RAM–disk–tape one, and accessing a file currently on tape would cause something that could be called a file access fault, resulting in the system automatically fetching the file back from tape (and keeping the requesting process blocked during all the time until the file becomes available again). Strategies for deciding which files to temporarily transfer from the disk
9.3 Page Replacement Strategy

A way of measuring a page replacement strategy is the average number of page faults that it causes. It is not possible to truly calculate this number, because it depends on so many things, e.g.:

- The absolute memory demands depend on all the processes currently running on a system.
- Even if sample situations (test cases) are created that consist of predefined processes with fixed start times and memory requirements (such as: process will access its page number \( n \) at instruction \( i \)) it is not possible to predict when precisely this process will execute this instruction—scheduling the processes will always result in slightly different orders of execution each time the test case is run.

So all we can do is think of theoretical properties of replacement strategies and, when implementing a strategy, observe its effects on a number of test cases which are tested several times in order to calculate an average number of page faults for each test case. Looking at the design of a strategy will however allow us to make some principle predictions.

9.3.1 Page Locking

Page replacement is a good idea, but some pages must sometimes be protected from being paged out. For example, certain parts of the operating system are so critical that they should never be paged out to secondary storage. The most striking example is the code that contains the interrupt handlers. If a page fault occurs and the code for the page fault handler is not be present, then we are be in big trouble. Also, most parts of the page tables for the kernel should always be present, as well as pages that are located in special frames for memory-mapped I/O. In such cases the pages should be *locked* into their frames and page replacement algorithms should ignore these pages.

*Unix* does not explicitly support page locking, but it considers the upper 1 GByte of each address space as locked: kernel memory will never be paged out, so we only have to deal with the memory of processes which avoids all the problems that otherwise call for page locking.


\section*{9.3.2 Page Replacement Strategies}

We describe a few classical replacement strategies before showing you the implementation in the U\textsc{li\textsubscript{x}} kernel.

\subsection*{9.3.2.1 FIFO Page Replacement}

A simple approach to page replacement is using a FIFO (first in, first out) list that keeps record of pages as they come into memory (either by being newly created, e.g. because a new process was started, or by being brought back in from disk after they had been paged out earlier). For U\textsc{li\textsubscript{x}} we could modify the `as\_map\_page\_to\_frame` and `page\_in` functions in order to keep track of new pages.

The list can grow up to a size that is determined by the number of available page frames in the system’s memory. When this limit is reached, the list will be chopped from the top: The page that is first in the list is removed from the list and also paged out. If the owning process tries to access this (paged-out) page again, a page fault occurs, and the memory manager has to page it back in, adding it at the end of the FIFO list.

This approach is simple because administering a FIFO list is simple, and selecting the next page to be paged out only requires reading the list head and removing it. However it has the problem of totally ignoring that some pages are accessed much more frequently than others. All pages travel from the list end to the list head at equal speed as pages are continuously paged out and back in, and for constantly and frequently used pages this means they will be paged out and in very often. It would make sense to be informed about the access frequency and keep the more frequently used pages in memory all the time, resulting in a much increased overall performance (with less page faults).

\subsection*{9.3.2.2 Second Chance Algorithm}

An attempt to bring the frequency of page access into the FIFO strategy is the introduction of a "second chance": The idea is to set an access bit for a page each time it is accessed by its owning process. This is something that the MMUs of most processors do automatically—which is important because otherwise it would be very hard to detect memory access manually.

The modification of the FIFO strategy is the following:

- A simple FIFO list of all pages works in principle as in the FIFO case.
- The MMU sets bits for each page access, as described above.
- When a page frame has to be freed, the system looks at the first list entry (as before). If that page has its access bit set, it is not paged out, but instead moved to the end of the list, and its access bit is cleared: it gets a second chance.

So the Second Chance algorithm selects the first page in the FIFO list that does not have its access bit set. “Not using the chance” then means that after being spared when first found at the list head, it will travel all the way from the end to the head of the list without being accessed one single time. Then the memory manager will page it out.
9.3.2.3 Clock

The Clock algorithm does the same as the Second Chance algorithm but does not require the reordering of the list (by taking away the head element and appending it to the list). Instead it uses a circular list (where the last element points back to the head) and uses a "clock hand" which points to the current head of the list.

When the algorithm needs to pick the candidate it starts with the list element that the clock hand points to. If its access bit is not set, that page is paged-out and removed from the list; the clock hand turns forward to the next element in the list.

If, however, the access bit is set, the algorithm clears it, moves the clock hand to the next location and starts over. It may eventually come full circle and arrive at the element whose access bit it had just reset; then it will pick that page.

Figure 9.1: The Clock algorithm resets the access bits in the first three entries (pages 26, 4, 72) and picks the fourth entry (page 1) because its access bit is not set.

Figure 9.1 shows an example of the Clock algorithm at work: When it starts, its clock hand points to page 26 (left part of figure). It sees that the page has its access bit set, so it resets it and moves on (clock-wise). The same repeats twice for pages 4 and 72, but when it reaches page 1 which does not have its access bit set, it picks that page as the candidate for removal.

9.3.2.4 Least-Frequently Used

The Second Hand or Clock strategy suffers from the fact that the only observed property of a page is whether it has been accessed recently or not. However, some pages are used much more often than others, and those much-used pages should be avoided when choosing a candidate for paging because they will be used again soon with high probability.

What we would like to have is an access log for each page so that we can pick a page which both was not accessed recently and in general was not accessed a lot further ago. A
true access log (that picks up every single access) is very hard to implement. For example, one could modify all page descriptors so that they cause a page fault. Then every access would generate a page fault, the page fault handler could temporarily grant access to that page and resume the process, only to remove the access permissions as soon as possible. While we would still not register all page accesses (since a process might access the same page several times in short sequence) it would give a good overview of the actual usage patterns. But this would be extremely expensive in terms of CPU time as the system would permanently generate page faults.

We can, however, do something that approximates such an access log: The Least-Frequently Used strategy counts page accesses by regularly checking and resetting the access bit. Every time it notices a set access bit it increments the access counter for that page. From time to time the counters need to be scaled down so that they don’t exceed the limit of their datatype. When the time comes to page out a page, the page with the lowest counter value is chosen.

For ULIx, with its fixed 64 MByte of RAM, this would mean keeping records for up to 16384 pages. If we used the maximum possible amount of RAM (4 GByte) and all its frames were in use, there would be more than a million pages to look after, and we might want to grow or shrink the list dynamically (according to the number of existing pages) so that we don’t waste too much memory for it. Also, the larger the list of pages becomes, the longer it takes to search for a minimum.

### 9.4 Page Replacement Implementation in ULIx

We will use access counters, but not for individual (process/page) pairs, but for hashes of them. This will let us use a fixed-size counter table which need not grow or shrink over time when new processes are created or old ones removed.

Each page can be identified by an \((as, pageno)\) pair. \(pageno\) is a 20 bit number, and \(as\) is a 10 bit number (since we only allow up to \(2^{10}\) address spaces).

We map this to an array index by calling

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno)
\]

Uses \(address\_space\) 161 and \text{hash} 306f.

This index number points to entry \(\text{counter\_table}[\text{index}]\) which stores a used flag and a counter. We will regularly update the counter table ...

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]

\[
\text{index} = \text{hash} ((address\_space \ll 20) \mid pageno);
\]
...and (less often) rescale the counters by halving them if the maximum counter is above some threshold:

\[305a\]
\[
\langle\text{pseudo code for counter rescaling}\rangle
\]
\[
\text{// get maximum count}
\]
\[
\text{themax} = 0;
\]
\[
\text{for (index in 0..maxindex)}\{
\text{\quad if (counter_table[index].used)}\{
\text{\quad \quad themax} = \text{max (themax, counter_table[index].count);}
\text{\quad\}\}
\text{if (themax < THRESHOLD) return; \text{// do nothing}}
\]

\[
\text{// halve all counters}
\]
\[
\text{for (index in 0..maxindex)}\{
\text{\quad if (counter_table[index].used)}\{
\text{\quad \quad counter_table[index].count} /= 2
\text{\quad \quad counter_table[index].count} += 1; \text{// add 1 to avoid 0 value}
\text{\quad\}\}
\}
\]

This automatically leads to some kind of aging: when the maximum reaches the threshold value, all entries will be halved.

Now, picking a page with minimum counter for replacement goes like this:

\[305b\]
\[
\langle\text{pseudo code for picking a page}\rangle
\]
\[
\text{pick} = \text{NULL;}
\]
\[
\text{for (as in used_address_spaces, pageno in user_page_numbers(as))} \{
\text{\quad index} = \text{hash ((address_space} \ll 20\text{) | pageno)};
\text{\quad if (pick==NULL && counter_table[index].used)}\{
\text{\quad \quad // initialize minimum, pick}
\text{\quad \quad pick} = (\text{as, pageno});
\text{\quad \quad themin} = \text{counter_table[index].count;}
\text{\quad\}\}
\text{\quad else} \{
\text{\quad \quad if (counter_table[index].count < themin)}\{
\text{\quad \quad \quad themin} = \text{counter_table[index].count}
\text{\quad \quad \quad pick} = (\text{as, pageno});
\text{\quad \quad\}\}
\text{\quad\}\}
\}
\text{if (pick != NULL) page_out (pick.as, pick.pageno);}
\]

This algorithm does not check whether a page is dirty, i.e., modified. In more advanced paging systems, a page may simultaneously exist in memory and on the disk (for example when it was paged out and paged back in but the swap file entry was not deleted). In that case it would make sense to pick a page which is still on disk and has not been changed in memory since it was brought back in the last time. Since our implementation does not keep pages both in RAM and on disk, we can ignore the dirty flag (or consider every page dirty; we always have to write to disk, whatever page we pick).

Note that the algorithm may pick a wrong page if there are pages with the same hash as the chosen one. In that case we have no way to decide which of those pages had the fewest accesses, but all of them at least had very few accesses, so this is good enough.
Here's the actual implementation. We define the counter table as an array of simple structures:

```c
#define PG_MAX_COUNTERS 1024

#define PG_MAX_COUNTERS, used in chunks 306–8.
```

```c
struct { boolean used; int count; } counter_table[PG_MAX_COUNTERS] = {{ 0 }};
lock paging_lock;
```

```c
Defines:

- `counter_table`, used in chunks 307 and 308.
- `paging_lock`, used in chunks 306–8.
```

We also provide a lock to protect access to that table:

```c
paging_lock = get_new_lock ("paging");
```

```c
Uses get_new_lock 367b, paging 292c, and paging_lock 306b.
```

And we regularly update the table via timer tasks.

```c
if (scheduler_is_active && ((system_ticks % 10) == 0)) {
    // Every 10 ticks (~ 0.1 seconds)
    // page replacement: update counters
}
if (scheduler_is_active && ((system_ticks % 50) == 5)) {
    // Every 50 ticks (~ 0.5 seconds)
    // page replacement: rescale counters
}
```

```c
Uses scheduler_is_active 276e and system_ticks 338a.
```

As mentioned above, we need a hash function for mapping all the possible (address space, page number) combinations onto our array. Hashing is a science in its own right, and we do not attempt to provide a clever or useful hashing algorithm in this book. Instead we implement our hash function in a very simple fashion: We assume that the `val` argument was created from an address space ID as and a page number `pageno` by calculating `(as << 20) | pageno`. Our hash function can then restore the original values via the formulas `as = val >> 20` and `pageno = val & 0b1111111111`. We multiply the address space ID with 32 and add the page number. Since that sum may exceed `PG_MAX_COUNTERS`, we use a modulo operation to make it fit:

```c
int hash (int val, int maxval) {
    // return val % maxval; // ridiculous hash
    return ((val >> 20)*32 + (val & 0b1111111111)) % maxval;
}
```

```c
Defines:

- `hash`, used in chunks 304 and 306–8.
```
The update code does not disable interrupts or use a lock; we do not really care if data are changed while we assemble the statistical data, since a small error in the statistics (which might result from parallel access to a page table entry) will not change the overall behavior.

The double loop over address space IDs and page numbers that we’ve shown above in the *(pseudo code for counter updates)* code chunk turns into a triple loop (over address space IDs, page table descriptors and page descriptors) since we cannot directly access the page descriptor for some page \( n \) without inspecting the right page table (number \( n/1024 \)) first. We only look at the first 768 page tables—beyond that kernel memory starts, and we have decided to never page out memory that belongs to the kernel. That way we need not deal with sticky bits (locked bits) in the page descriptors. Instead, the simple rule is: If a page belongs to process memory, it is a candidate for removal; otherwise not.

```c
if (mutex_try_lock (paging_lock)) {
    for (int as = 1; as < MAX_ADDR_SPACES; as++) {
        if (address_spaces[as].status != AS_FREE) {
            page_directory *pd = address_spaces[as].pd;
            for (int i = 0; i < 100; i++) { // < 768: only work on process memory
                if (pd->ptds[i].present) { // directory entry in use
                    page_table *pt = (page_table*)PHYSCAL ((pd->ptds[i].frame_addr)<<12));
                    for (int j = 0; j < 1024; j++) {
                        if (pt->pds[j].present) { // table entry in use
                            (page replacement: update counter for page i \cdot 1024 + j)*
                        }
                    }
                }
            }
        }
    }
    mutex_unlock (paging_lock);
}
```

Uses address_spaces 162b, AS_FREE 162a, MAX_ADDR_SPACES 158a, mutex_try_lock 366b, mutex_unlock 366c, page_directory 103d, page_table 101b, paging_lock 306b, and PHYSICAL 116a.

For updating the counter for page \( 1024 \cdot i + j \) we look at its page descriptor. If the accessed bit is set

```c
int pageno = i*1024 + j;
int n = pt->pds[j].accessed; // get and ...
pt->pds[j].accessed = false; // reset access flag
int index;
if (n == 1 &&
    (index = hash ((as << 20) | pageno, PG_MAX_COUNTERS)) < PG_MAX_COUNTERS) {
    counter_table[index].used = true;
    counter_table[index].count++;
}
```

Uses counter_table 306b, hash 306f, and PG_MAX_COUNTERS 306a.
The implementation of the rescaling operation is only slightly more complex than the pseudocode:

```c
// get the maximum count
int themax = 0;
if (mutex_try_lock (paging_lock)) {
    for (int index = 0; index < PG_MAX_COUNTERS; index++) {
        if (counter_table[index].used) {
            int val = counter_table[index].count;
            if (val > themax) themax = val;
        }
    }
    if (themax > PG_COUNTER_THRESHOLD) {
        // rescale all counters
        for (int index = 0; index < PG_MAX_COUNTERS; index++) {
            counter_table[index].count /= 2;
            counter_table[index].count += 1; // avoid 0 value
        }
    }
}
mutex_unlock (paging_lock);
```

We still need to define the counter threshold:

```c
#define PG_COUNTER_THRESHOLD 100000
```

Once at least one page has been access counted more than 100 000 times, the counter values will be rescaled.

Finally this is the code which frees a frame. It looks at all the pages in all address spaces, generates the hash and looks up the counter for that hash (if it exists). It initializes pick_as and pick_pageno to the first address space and page number for whose hash it finds a counter and then updates these variables whenever it finds a smaller counter.

```c
addr_space_id pick_as = -1;
int pick_pageno, themin;
while (!mutex_try_lock (paging_lock)) ; // active waiting for lock
for (int as = 1; as < MAX_ADDR_SPACES; as++) {
    if (address_spaces[as].status == AS_USED) {
        page_directory *pd = address_spaces[as].pd;
        for (int i = 0; i < 768; i++) { // < 768: only work on process memory
            if (pd->ptds[i].present) { // directory entry in use
```
Page Replacement Implementation in ULIX

```c
page_table *pt = (page_table*) (PHYSICAL ((pd->ptds[i].frame_addr) << 12));
for (int j = 0; j < 1024; j++) {
    if (pt->pds[j].present) {// table entry in use
        int pageno = i*1024 + j;
        int index = hash ((as << 20) | pageno, PG_MAX_COUNTERS);
        if (pick_as==-1 & counter_table[index].used) {
            // initialize minimum, pick
            pick_as = as;
            pick_pageno = pageno;
            themin = counter_table[index].count;
        } else {
            if (counter_table[index].count < themin) {
                themin = counter_table[index].count;
                pick_as = as;
                pick_pageno = pageno;
            }
        }
    }
}
mutex_unlock (paging_lock);

if (pick_as != -1) {
    mutex_lock (paging_lock);
    page_out (pick_as, pick_pageno);
    mutex_unlock (paging_lock);
} else {
    printf ("\nERROR: cannot pick a page to evict!\n");
}
```

Uses addr_space_id 158b, address_spaces 162b, AS_USED 162a, counter_table 306b, hash 306f, lock 365a,
MAX_ADDR_SPACES 158a, mutex_lock 366a, mutex_try_lock 366b, mutex_unlock 366c, page_directory 103d,
page_out 296, page_table 101b, paging_lock 306b, PG_MAX_COUNTERS 306a, PHYSICAL 116a, pick_pageno,
printf 601a, and themin.

Again, instead of the double loop from the (pseudo code for picking a page 305b) code chunk, we need a triple loop to access all page tables referenced by all page directories for all address spaces.

### 9.4.1 The Swapper Process

We provide two system calls that retrieve the number of free frames and issue a request to free a page:

```c
void syscall_get_free_frames (context_t *r);
void syscall_free_a_frame (context_t *r);
```
void syscall_get_free_frames (context_t *r) {
    // no parameters
    mutex_lock (swapper_lock);   // lock_, see below
eax_return (free_frames);
}

void syscall_free_a_frame (context_t *r) {
    // no parameters
    ⟨page replacement: free one frame 308c⟩
}

#define __NR_get_free_frames 509
#define __NR_free_a_frame 510

Uses __NR_free_a_frame and __NR_get_free_frames.

We also need user mode library functions which can make the two system calls:

int get_free_frames ();
void free_a_frame ();

The swapper process should not work permanently, so we use a trick: We let it block on a lock and add a timer task that unlocks that lock every 0.1 seconds.
The swapper program switches to the last virtual console. In an infinite loop it queries the number of free frames using `get_free_frames`, and that function will block because `syscall_get_free_frames` locks the `swapper_lock`. The function returns after the timer handler releases the lock, so the loop is only executed every 0.1 seconds.

If the number of frames gets too low, the program calls `free_a_frame`.

We will start this swapper process right from the `init` process; it will run with process ID 2. In order to stop arbitrary processes from calling `free_a_frame`, the system call handler should verify that it was called by this process (and no other one).
10

Talking to the Hardware

In this chapter we provide the code which talks to various kinds of hardware. In most cases this will include an interrupt handler which gets called when a device generates a hardware interrupt.

10.1 Keyboard

Ulix does not provide a graphical user interface, and it does not recognize a mouse. Thus, the keyboard is the only available input device. Since there will be up to ten virtual consoles (on which users can log on with different user accounts), we need several keyboard input buffers and keep track of where to store a new character when a key was pressed.

10.1.1 Scan Code Tables

The keyboard interrupt handler must recognize which key was pressed, while also checking if any of the modifier keys (such as shift, control or alt) was held down at the same time.

The array scancode_table316 maps the key codes of a standard US keyboard (as generated by the keyboard controller) to ASCII characters. We started with the code in Bran’s Kernel Development tutorial [Fri05] (the table is on the http://www.osdever.net/bkerndev/Docs/keyboard.htm page), but modified it.

Similarly, scancode_up_table316 holds the characters for the same key codes, but with one of the shift keys pressed. Since we alternated between US and German keyboards during the development of Ulix, we also provide corresponding tables for the German layout which you can find in scancode_DE_table317 and scancode_DE_up_table317.
Figure 10.1: Layout of a US keyboard with additional Windows keys (without the number pad)

Figure 10.2: Layout of a German keyboard with additional Windows keys (without the number pad)

Figure 10.3: Scancodes for the US keyboard; on a German keyboard “<” generates the key code 41.

Figure 10.1 shows the layout of a standard US-American PC keyboard, and Figure 10.2 shows the German layout. In the third figure (Figure 10.3) you can find the key codes.
Both pressing and releasing a key generate a key code (that way the operating system can see whether the user holds a key pressed). The key codes for pressing and releasing any specific key are identical except for the upper bit: If a key was pressed, the key code’s upper bit is unset (0); if it was released it is set (1). Thus `scancode & 0x80` is 0 if the event is a key press event, it is non-zero otherwise. In the latter case `scancode-0x80` (or `scancode & ~0x80`) calculates the key code of the corresponding key press event.

There are some exceptions for newer keys which did not exist on the original PC XT keyboard [IBM83, pp. 1-65–1-69], and they use combinations which are initiated with an escape character (0xe0 = 224 or 0xe1 = 225). Figure 10.3 shows this for the two Windows keys, the (Windows) menu key and the right Alt and Ctrl keys. Note how Left-Alt and Right-Alt (or Left-Ctrl and Right-Ctrl) only differ in that the right keys generate the escape code and then the same code as the corresponding left key, e.g., 29 for Left-Ctrl and 224 / 29 for Right-Ctrl. This way a driver that is unaware of escape codes will just ignore the escape code and interpret the second code (almost) correctly.

In the “Keyboard scan codes” list [Bro09], Brouwer describes the newer keys, too. He also notes:

“The prefix e0 was originally used for the grey duplicates of keys on the original PC/XT keyboard. These days e0 is just used to expand code space. The prefix e1 used for Pause/Break indicated that this key sends the make/break sequence at make time, and does nothing upon release.”

The terms `scan codes` and `key codes` are sometimes used interchangeably, but there are other encodings of key-press and key-release events. We only discuss the key codes that are transmitted by the keyboard controller. They are also called “set 1” or “IBM PC XT” scan codes. A complete overview of “set 1” and “set 2” scan codes can also be found in a Microsoft specification document [Mic00a].

All 0 entries in the map make ULix ignore a key. We also enter 0 in the map for modifier keys (Shift, Ctrl, Alt etc.) since we handle them separately. For the Escape and cursor keys we provide names because we will use them later:

```c
#define KEY_ESC 27
#define KEY_UP 191
#define KEY_DOWN 192
#define KEY_LEFT 193
#define KEY_RIGHT 194
```

Defines:
- `KEY_DOWN`, used in chunks 316 and 317.
- `KEY_LEFT`, used in chunks 316 and 317.
- `KEY_RIGHT`, used in chunks 316 and 317.
- `KEY_UP`, used in chunks 316 and 317.

Uses `KEY_ESC`.

This is the table for the US keyboard:
byte scancode_table[128] = {
  /*  0.. 9 */ 0, KEY_ESC, '1', '2', '3', '4', '5', '6', '7', '8',
  /* 10..19 */ '9', '0', '-', '=', '\b', /* Backspace */
    '\t', /* Tab */ 'q', 'w', 'e', 'r',
  /* 20..29 */ 't', 'y', 'u', 'i', 'o', 'p', '{', '}',
    '\n', /* Enter */ 0, /* Control */
  /* 30..39 */ 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', ';
    /* Backspace */
  /* 40..49 */ '\', ',', ' ', 0, /* Left shift */ ' ', 'z', 'x', 'c', 'v', 'b', 'n',
  /* 50..59 */ 'm', ',', '.', '/', 0, /* Right shift */
    '*', 0, /* Alt */ ' ', /* Space bar */
    0, /* CapsLock */ 0, /* F1 */
  /* 60..69 */ 0, 0, 0, 0, 0, 0, 0, 0, /* F2..F10 */ 0, /* NumLock */
  /* 70..79 */ 0, /* Scroll Lock */ 0, /* Home */ KEY_UP, 0, /* Page Up */
    '-', KEY_LEFT, 0, KEY_RIGHT, '+', 0, /* End */
  /* 80..89 */ KEY_DOWN, 0, /* Page Down */ 0, /* Insert */ 0, /* Delete */
    0, 0, 0, 0, /* F11 */ 0, /* F12 */ 0,
  /* 90..127 */ not defined */
};

byte scancode_up_table[128] = {
  /*  0.. 9 */ 0, KEY_ESC, '!', '@', '#', '$', '%', '^', '&', '*',
  /* 10..19 */ '(', ')', '-', '+', '\b', /* Backspace */
    '\t', /* Tab */ 'Q', 'W', 'E', 'R',
  /* 20..29 */ 'T', 'Y', 'U', 'I', 'O', 'P', '{', '}',
    '\n', /* Enter */ 0, /* Control */
  /* 30..39 */ 'A', 'S', 'D', 'F', 'G', 'H', 'J', 'K', 'L', ':'
    /* Backspace */
  /* 40..49 */ '``', '~', 0, /* Left shift */ ' ', 'Z', 'X', 'C', 'V', 'B', 'N',
  /* 50..59 */ 'M', '<', '>', '?', 0, /* Right shift */
    '*', 0, /* Alt */ ' ', /* Space bar */
    0, /* CapsLock */ 0, /* F1 */
  /* 60..69 */ 0, 0, 0, 0, 0, 0, 0, 0, /* F2..F10 */ 0, /* NumLock */
  /* 70..79 */ 0, /* Scroll Lock */ 0, /* Home */ KEY_UP, 0, /* Page Up */
    '-', KEY_LEFT, 0, KEY_RIGHT, '+', 0, /* End */
  /* 80..89 */ KEY_DOWN, 0, /* Page Down */ 0, /* Insert */ 0, /* Delete */
    0, 0, 0, 0, /* F11 */ 0, /* F12 */ 0,
  /* 90..127 */ not defined */
};

Defines:
  scancode_table, used in chunk 319d.
  scancode_up_table, used in chunk 319d.
Uses KEY_DOWN 315, KEY_ESC, KEY_LEFT 315, KEY_RIGHT 315, and KEY_UP 315.

Ulix does not support German special characters (äöüßÁÖÜß§), so the keys which would
generate those characters are mapped to standard ASCII characters which can then be
entered via two keys, for example, pressing [Ä] or [Shift-Ä] will generate the ' and "
characters. Users can switch between the US and German layouts by pressing [Ctrl-L].
 Defines:  
  scandcode_DE_table, used in chunk 319d.  
  scandcode_DE_up_table, used in chunk 319d.  

Uses KEY_DOWN, KEY_ESC, KEY_LEFT, KEY_RIGHT, KEY_UP, KEY_DOWN,  
KEY_ESC, KEY_LEFT, KEY_RIGHT, KEY_UP, F1, F2, F3, F4, F5, F6, F7,  
F8, F9, F10, F11, F12, and NumLock.

10.1.2 Virtual Consoles

We provide ten virtual consoles (terminals), each of which has its own keyboard buffer.  
Such a buffer can store up to 32 characters—if an application does not react fast enough to  
key-press events, the buffer can become full: in that case further key-presses are lost.
For each buffer we also store the current position (where the next character will be entered) and the last read position (which character was last read):

```
typedef struct {
    char kbd[SYSTEM_KBD_BUFLEN+1];
    int kbd_pos;
    int kbd_lastread;
    int kbd_count;
} terminal_t;
```

The kbd_count field is redundant but makes checking the buffer status simpler.

Terminal 0 is also used as the system terminal. Ulx provides a kernel mode shell that can be activated with Shift-Esc. It always uses the first terminal and keeps its own set of position variables.

```
char *system_kbd = terminals[0].kbd;
int system_kbd_pos;
int system_kbd_lastread;
int system_kbd_count;
```

They need to be initialized when the system boots:
10.1.3 Keyboard Interrupt Handler

The keyboard handler deals with all press and release events:

\[
\begin{align*}
\text{It checks the variable LANG_GERMAN to decide whether it shall use the German or the US keyboard layout:} \\
\text{LANG_GERMAN} = 1; \quad // \text{default: german keyboard}
\end{align*}
\]

The implementation is rather simple, the function is only long because it needs to handle key presses differently when one of the modifier keys (Left-Shift, Right-Shift, Ctrl, Alt) is held while another key is pressed. Other than that, the handler reads a scan code from the keyboard I/O port.

\[
\begin{align*}
\text{and interprets it. For standard keys it looks up the assigned character using one of the scan code tables. Key-release events are ignored unless one of the modifier keys was released: in that case the status of shift_pressed, alt_pressed etc. must be updated. We declare those variables as static in the function so that they keep their values between several invocations of the handler.}
\end{align*}
\]

\[
\begin{align*}
\text{After initializing the keyboard mapping and the states of the modifier keys the real work begins. We read the scan code from the I/O port IO_KEYBOARD. Then we check if the scan code corresponds to a key release event (i.e., the highest bit is set). That situation}
\end{align*}
\]
is only of interest for the modifier keys: If Shift, Ctrl or Alt were released, we update the corresponding static variable and return immediately. The release of regular keys is ignored. We also give the modifier key numbers names to make the code more readable:

```c
#define KEY_CTRL 29
#define KEY_L_SHIFT 42
#define KEY_R_SHIFT 54
#define KEY_ALT 56
```

Defines:
- KEY_ALT, used in chunk 320.
- KEY_CTRL, used in chunk 320.
- KEY_L_SHIFT, used in chunk 320.
- KEY_R_SHIFT, used in chunk 320.

Otherwise we deal with a key press event. To keep things ordered nicely, we start with checking whether one of the modifier keys was pressed: Again, we can update a state variable and return from the handler. If that was not the case, we look up the character in the right scan code table (either `upper_table` or `lower_table`):

```c
byte c = (shift_pressed ? upper_table[scancode] : lower_table[scancode]);
```

Uses `KEY_ALT` 320a, `KEY_CTRL` 320a, `KEY_L_SHIFT` 320a, and `KEY_R_SHIFT` 320a.

Then we check for special key combinations: Alt-0 to Alt-9 let us switch to a different terminal, Ctrl-C kills the current process, Ctrl-L changes the keyboard layout, and Shift-Escape starts the *kernel mode shell* (which can be used for debugging, see Chapter 17).
// Alt-0 to Alt-9: switch terminal
if (alt_pressed && '0' ≤ c && c ≤ '9') {
    vt_activate ((int)((c-'0')+9)%10);  // activate virtual console
    vt_move_cursor ();  // update cursor on new terminal
    return;
}

// Ctrl-C: kill and reset input
if (ctrl_pressed && c == 'c') {
    u_kill (target_pid, SIGKILL);  // kill the process
    return;
}

// Ctrl-L: change keyboard layout
if (ctrl_pressed && c == 'l') {
    switch (LANG_GERMAN) {
        case 0:  LANG_GERMAN = 1;  _set_statusline ("de", 44);  return;
        case 1:  LANG_GERMAN = 0;  _set_statusline ("en", 44);  return;
    }
}

// Shift-Escape: start kernel mode shell
if (shift_pressed && c == KEY_ESC && scheduler_is_active) {
    printf ("\nGoing to kernel shell\n");
    vt_activate (0);  // must run on vt0
    kernel_shell ();
    printf ("returning from kernel shell\n");
    return;
}

With all special cases handled, only the default case remains: If a regular character was entered, we need to store it in one of the keyboard buffers—as long as it is not filled already. So we first check whether the buffer can carry the new character:

Uses _set_statusline 337b, kernel_shell 610a, KEY_ESC, kill 568b, LANG_GERMAN 319b, printf 601a, scheduler_is_active 276e, SIGKILL 562a, target_pid, u_kill 562b, vt_activate 327a, and vt_move_cursor 328a.

Uses cur_vt 326a, scheduler_is_active 276e, SYSTEM_KBD_BUFLEN 318a, terminal_t 318b, and terminals 318c.
We still need to discuss what happens when a process is sleeping (while waiting for input from its terminal). We search the keyboard_queue (that we will define in the following section) for a process which waits for input and uses the currently active terminal cur_vt. If we find one (and we assume that for each terminal at most one process can wait for key entry) we wake it up, i.e., move it to the ready queue using the deblock function:

\[
\text{(keyboard handler: wake sleeping process 322a)} \equiv
\]
\[
\text{thread_id start_pid = keyboard_queue.next;}
\]
\[
\text{if (start_pid != 0)) { // only if the queue is not empty}
\]
\[
\text{thread_id search_pid = start_pid;}
\]
\[
\text{do {}
\]
\[
\text{if (thread_table[search_pid].terminal == cur_vt) {}
\]
\[
\text{deblock (search_pid, &keyboard_queue); break;}
\]
\[
\text{else {}
\]
\[
\text{search_pid = thread_table[search_pid].next;}
\]
\[
\text{}}}
\]
\[
\text{while (search_pid != start_pid & search_pid != 0);}
\]

Uses cur_vt, deblock, keyboard_queue, thread_id, and thread_table.

A Ctrl-C key combination should make the system deliver a SIGKILL signal to the process that uses the current terminal. There may be several such processes; we will pick the first one which has no child process:

\[
\text{(keyboard handler: find active process, set target_pid 322b)} \equiv
\]
\[
\text{int target_pid = 0;}
\]
\[
\text{for (int i = 3; i < MAX_THREADS; i++) {}
\]
\[
\text{if (thread_table[i].used & (thread_table[i].terminal == cur_vt)) {}
\]
\[
\text{int is_candidate = true;}
\]
\[
\text{for (int j = 3; j < MAX_THREADS; j++) {}
\]
\[
\text{if (thread_table[j].used & (thread_table[j].ppid == i)) {}
\]
\[
\text{// thread j has parent i - not a candidate}
\]
\[
\text{is_candidate = false; break; // leave inner loop}
\]
\[
\text{}}}
\]
\[
\text{if (is_candidate) {}
\]
\[
\text{target_pid = i; goto end_of_search;}
\]
\[
\text{}}}
\]
\[
\text{end_of_search: ; // label needs a statement}
\]

Uses cur_vt, MAX_THREADS, target_pid, and thread_table.
During system initialization we register the keyboard handler:

```c
void keyboard_install();
```

We add calling `keyboard_install` to the general chunk that installs interrupt handlers:

```c
keyboard_install();
```

### 10.1.4 The Keyboard Queue

We provide several blocked queues—one for each different reason that a process may block for. Here we define the queue for processes that wait for a keystroke (on their terminal).

```c
blocked_queue keyboard_queue; // processes which wait for a keystroke
```

We must initialize the queue:

```c
initialize_blocked_queue(&keyboard_queue);
```

Now we can provide two functions which read in a character or a whole string:

```c
void kgetch (char *c);
void kreadline (char *s, int maxlength);
```

They are only used for the kernel mode shell, processes have their own way of reading characters from the keyboard; they use the regular file `read` function with the `STDIN_FILENO` file descriptor because their input might be redirected to a file. We will describe this in Chapter 12.

In kernel mode we can just run a loop that waits for a new character to appear in the keyboard buffer.
void kgetch (char *c) {
    int t = thread_table[current_task].terminal;
    if (t < 0 || t > TERMINALS-1) {
        t = 0; printf ("ERROR: terminal not set! setting to 0\n");
    }
    terminal_t *term = &terminals[t];

    *c = 0;
    while (*c == 0) {
        if (term->kbd_count > 0) {
            term->kbd_count--;
            term->kbd_lastread = (term->kbd_lastread+1) % SYSTEM_KBD_BUFLEN;
            *c = term->kbd[term->kbd_lastread];
        } else {
            *c = 0;
        }
    }
}

void kreadline (char *s, int maxlength) {
    char c;
    int pos = 0;
    for (;;) {
        kgetch (&c); // read one character
        if (c == 0x08 && pos > 0) { // backspace
            pos--;
            kputch (c); kputch (' '); kputch (c);
        } else if (c == 'n') { // newline: end of input
            kputch ('\n');
            s[pos] = (char) 0;
            return;
        } else if (c != 0x08 && pos < maxlength) { // other character
            kputch (c);
            s[pos++] = c;
        }
    }
}

void kgetch (char *c)

void kreadline (char *s, int maxlength)
10.2 Terminals

We want Ulx to provide several terminals so that we can run a few login shells and execute programs on them.

Conceptually, providing terminals is not complicated: we need

- memory to store the contents of the terminals – roughly $80 \times 25 \times 2$ bytes per terminal (the size of the textmode video buffer),
- a way to make Ulx switch the active terminal,
- a modification of the write functions so that they will either write to the current terminal or a specified terminal.
- When writes to a terminal occur, the terminal’s screen buffer is updated—if it is the active terminal, the screen is updated at the same time.
- When switching to a different terminal, its screen buffer is copied to the screen.

We start with the required memory. Since Ulx uses the last line on the screen for displaying a status line, we consider it not to be part of any terminal buffer; for example scrolling shall always ignore the last line, and from a process’ point of view the 25th line does not exist. So we can define

```c
#define VT_WIDTH (80)
#define VT_HEIGHT (24)
#define VT_SIZE (VT_WIDTH * VT_HEIGHT * 2)
```

Defines:
- VT_HEIGHT, used in chunk 334.
- VT_SIZE, used in chunks 325–27, 329b, 332b, and 337b.
- VT_WIDTH, used in chunks 329b and 334a.

```c
typedef struct {
    char mem[VT_SIZE];
    int x,y;
} term_buffer;
```

Defines:
- term_buffer, used in chunks 326a, 334b, and 335b.

Uses VT_SIZE 325a.

Two bytes are required for each character; the first one holds the ASCII value of the symbol to be displayed, the second is used for foreground and background colors.

We want the system to use up to ten virtual consoles (numbered from 0 to 9), so we create an array for them:

```c
#define MAX_VT 9
```

Defines:
- MAX_VT, used in chunks 326–28.
term_buffer vt[MAX_VT+1];
int cur_vt = 0;

Defines:
cur_vt, used in chunks 321, 322, 327a, 329b, 332b, 334b, 335b, and 342b.
v_t, used in chunks 326–30, 322b, 334b, and 335b.
Uses MAX_VT 325c and term_buffer 325b.

vt[i].mem is the buffer of console i, and vt[i].x and vt[i].y hold the current
cursor position in console i. We initialize the current terminal to number 0.

To start with proper contents, we initialize each of the ten text buffers with blanks. A
blank character is actually a word with the low byte containing the ASCII value of the
blank symbol (0x20) and the high byte containing the color information (0x0F for white on
black).

#define VT_NORMAL_BACKGROUND (0x0F << 8)
#define VT_BLUE_BACKGROUND (0x1F << 8)
#define VT_RED_BACKGROUND (0x4F << 8)

Defines:
VT_BLUE_BACKGROUND, used in chunks 329b and 609.
VT_NORMAL_BACKGROUND, used in chunks 326c, 329b, and 333–35.
VT_RED_BACKGROUND, used in chunk 609.

int vtno;
word *memptr;
unsigned blank = 0x20 | VT_NORMAL_BACKGROUND;  // blank character
for (vtno = 1; vtno < 10; vtno++) {
    memptr = (word*)vt[vtno].mem;
    memsetw (memptr, blank, VT_SIZE/2);
}
printf ("VT: Initialized ten terminals (press [Alt-1] to [Alt-0])\n"");
Uses memsetw 596c, printf 601a, terminals 318c, vt 326a, VT_NORMAL_BACKGROUND 326b, and VT_SIZE 325a.

Note that we do not initialize the first terminal’s buffer vt[0] because it will obtain a
copy of the current screen when we switch to a different terminal.

We also need a way to tell a process what terminal it runs on, so we add a new TCB entry:

int terminal;

A regular Unix system would allow for a more complex setup, but for Unix we restrict
ourselves to using ten text consoles.

Activating a console via

int vt_activate (int i);

is the simplest of all the operations:
10.2 Terminals

\(\text{function implementations} \ 100b\) +≡ (44a) <324b 328a> [327a]
int vt_activate (int new_vt) {
    if (new_vt < 0 || new_vt > MAX_VT) return -1; // no such console
else {
    memcpy (vt[cur_vt].mem, (void*)VIDEORAM, VT_SIZE); // save old contents
    vt[cur_vt].x = csr_x; vt[cur_vt].y = csr_y;
    memcpy ((void*)VIDEORAM, vt[new_vt].mem, VT_SIZE); // load new contents
    cur_vt = new_vt;
    csr_x = vt[new_vt].x; csr_y = vt[new_vt].y;
    vt_move_cursor ()
    return 0;
}
}

Defines:
vt_activate, used in chunks 321a and 326e.
Uses cur_vt 326a, MAX_VT 325c, memcpy 596c, VIDEORAM 327b, vt 326a, vt_move_cursor 328a, and VT_SIZE 325a.

Here we’re using the address VIDEORAM\[327b\] to access the text mode frame buffer of the graphics card; csr_x and csr_y store the cursor position on the visible terminal. We have not defined the variables yet, so here they are:

\(\text{global variables} \ 92b\) +≡ (44a) <326a 328b> [327b]
uint VIDEORAM = 0xB8000;
byte csr_x = 0; byte csr_y = 0; // Cursor position

Defines:
VIDEORAM, used in chunks 116, 327a, 332b, 334b, 337b, and 342d.

It is initially set to 0xb8000 but changes its value to 0xd00b8000 during system initialization (when we set up paging). We can also use textmemptr\[116c\] which was \#defined as ((word*)VIDEORAM\[327b\]).

With vt_move_cursor\[328a\] we update the cursor location, since it will need to be in a different position on the new terminal. The cursor location can be controlled by sending 80 · \(x + y\) (where \(x\) is the line number and \(y\) is the column number) to the VGA cursor location register. This is a 16 bit value—it must be sent in two chunks. First the control code IO\_VGA\_CURSOR\_LOC\_HIGH\[327c\] is sent to the IO\_VGA\_TARGET\[327c\] port (which signals that the high byte of the cursor location follows), then that high byte is sent to the IO\_VGA\_VALUE\[327c\] port. A similar sequence follows, using IO\_VGA\_CURSOR\_LOC\_LOW\[327c\] and the lower byte.

\(\text{constants} \ 112a\) +≡ (44a) <325c 338d> [327c]
#define IO\_VGA\_TARGET 0x3D4
#define IO\_VGA\_VALUE 0x3D5
#define IO\_VGA\_CURSOR\_LOC\_HIGH 14
#define IO\_VGA\_CURSOR\_LOC\_LOW 15

Defines:
IO\_VGA\_CURSOR\_LOC\_HIGH, used in chunk 328a.
IO\_VGA\_CURSOR\_LOC\_LOW, used in chunk 328a.
IO\_VGA\_TARGET, used in chunk 328a.

\(\text{function prototypes} \ 45a\) +≡ (44a) <326e 329a> [327d]
void vt_move_cursor ();
10 Talking to the Hardware

### Function Implementations

**vt_move_cursor**

```c
void vt_move_cursor()
{
    unsigned position = csr_y * 80 + csr_x;
    // high byte:
    outportb(IO_VGA_TARGET, IO_VGA_CURSOR_LOC_HIGH);
    outportb(0x3D5, position >> 8);
    // low byte:
    outportb(IO_VGA_TARGET, IO_VGA_CURSOR_LOC_LOW);
    outportb(0x3D5, position & 0xff); // low byte
}
```

**Defines:**

- vt_move_cursor, used in chunks 321a, 327, 329b, and 330a.
- Uses IO_VGA_CURSOR_LOC_HIGH, IO_VGA_CURSOR_LOC_LOW, IO_VGA_TARGET, and outportb.

**Let's define what terminal we expect to display kernel messages.** We initialize the variable `KERNEL_VT` to 0 (for the first terminal), though it may later be changed.

```c
short int KERNEL_VT = 0;
```

**Defines:**

- KERNEL_VT, used in chunks 334b and 335b.

**Back to terminal selection, we provide a system call that lets a process choose which terminal to use.**

```c
void syscall_setterm(context_t *r)
{
    int vt = r->ebx; // argument in ebx register
    if (vt<0 || vt[MAX_VT]) return; // check if proper number...
    thread_table[current_task].terminal = vt;
}
```

**Defines:**

- syscall_setterm, used in chunk 328e.
- Uses context_t, current_task, MAX_VT, thread_table, and vt.

**We define the system call number and register the syscall:**

```c
#define __NR_setterm 511
```

**Defines:**

- __NR_setterm, used in chunk 328.

**The user mode library gains a new function as well:**

```c
void setterm(int vt)
{
    syscall2(__NR_setterm, (uint) vt);
}
```

**Defines:**

- setterm, used in chunks 311b and 513e.
- Uses __NR_setterm, syscall2, and vt.
We also need functions to clear the screen, set the cursor and get the current cursor location:

\[
\text{\texttt{void \texttt{vt\_clrscr \(\cdots\)}}} \quad \text{\texttt{void \texttt{vt\_get\_xy char *x, char *y);}}} \\
\text{\texttt{void \texttt{vt\_set\_xy char x, char y);}}}
\]

Clearing the screen means writing a blank character to each location. We need to consider that each character byte is followed by a format byte and—if calling the function from the kernel—we want to format the last line with a blue background so that the status line can be recognized.

\texttt{vt\_clrscr}, just overwrites the terminal buffer of the current process with blank characters and then calls \texttt{vt\_set\_xy} to set the cursor to the top left position. If the current process is also working on the currently visible terminal, the function updates the physical screen as well. (Otherwise the change will only become visible when the user switches to that terminal.)

\[
\text{\texttt{void \texttt{vt\_clrscr \(\cdots\)}}} \\
\text{\texttt{\{ \}}} \\
\text{\texttt{word \texttt{blank} = 0x20 | VT\_NORMAL\_BACKGROUND; \}}} \\
\text{\texttt{word \texttt{blankrev} = 0x20 | VT\_BLUE\_BACKGROUND; \}}} \\
\text{\texttt{int \texttt{process\_term}; \}}} \\
\text{\texttt{if (scheduler\_is\_active) \{ \}}} \\
\text{\texttt{process\_term = thread\_table[current\_task].terminal; \}}} \\
\text{\texttt{word *memptr = (word*)vt[process\_term].mem; \}}} \\
\text{\texttt{memsetw (memptr, blank, VT\_SIZE/2); \}}} \\
\text{\texttt{\}}} \\
\text{\texttt{\}}} \\
\text{\texttt{// \texttt{lines 1-24}}} \\
\text{\texttt{vt\_set\_xy (0, 0); \}}} \\
\text{\texttt{\}}} \\
\text{\texttt{// current terminal? \}}} \\
\text{\texttt{if ((!scheduler\_is\_active) || (scheduler\_is\_active && process\_term == cur\_vt)) \}}} \\
\text{\texttt{memsetw (textmemptr, blank, VT\_SIZE/2); \}}} \\
\text{\texttt{// \texttt{lines 1-24}}} \\
\text{\texttt{\}}} \\
\text{\texttt{// kernel mode? clear status line, set cursor \}}} \\
\text{\texttt{if (!scheduler\_is\_active) \{ \}}} \\
\text{\texttt{memsetw (textmemptr + VT\_SIZE/2, blankrev, VT\_WIDTH); \}}} \\
\text{\texttt{// line 25}} \\
\text{\texttt{csr\_x = csr\_y = 0; \}}} \\
\text{\texttt{vt\_move\_cursor (); \}}} \\
\text{\texttt{\}}} \\
\text{\texttt{\}}}

Defines:
\texttt{vt\_clrscr}, used in chunks 331a, 337c, and 608b.
Uses cur\_vt 326a, current\_task 192c, memsetw 596c, scheduler\_is\_active 276e, textmemptr 116c, thread\_table 176b, vt 326a, VT\_BLUE\_BACKGROUND 326b, vt\_move\_cursor 328a, VT\_NORMAL\_BACKGROUND 326b, vt\_set\_xy 330a, VT\_SIZE 325a, and VT\_WIDTH 325a.
The \texttt{vt\_get\_xy} and \texttt{vt\_set\_xy} read respectively set the \(x\) and \(y\) members of the current terminal's \texttt{term\_buffer} structure. We will only call them from processes, so we need not check for as many special cases as we did in \texttt{vt\_clrscr}. The only condition we have to check is whether we're changing the cursor location of the currently active terminal—then we also need to update the hardware cursor.

\begin{Verbatim}
\begin{function implementations}\end{function implementations}
\begin{verbatim}
void \texttt{vt\_get\_xy}\ ((\texttt{char} \ *x, \ \texttt{char} \ *y)) \{
    \texttt{int} \ \texttt{process\_term} = \texttt{thread\_table[current\_task].terminal};
    \texttt{*x} = \texttt{vt[process\_term].x};
    \texttt{*y} = \texttt{vt[process\_term].y};
\}

void \texttt{vt\_set\_xy}\ ((\texttt{char} \ x, \ \texttt{char} \ y)) \{
    \texttt{int} \ \texttt{process\_term} = \texttt{thread\_table[current\_task].terminal};
    \texttt{vt[process\_term].x} = \texttt{x};
    \texttt{vt[process\_term].y} = \texttt{y};

    \texttt{// current terminal?}
    \texttt{if} (\texttt{process\_term} == \texttt{cur\_vt}) \{
        \texttt{csr\_x} = \texttt{x}; \texttt{csr\_y} = \texttt{y};
        \texttt{vt\_move\_cursor} ();
    \}
\}
\end{verbatim}
\end{function implementations}

Defines:
\texttt{vt\_get\_xy}, used in chunk 331a.
\texttt{vt\_set\_xy}, used in chunks 329 and 331a.
Uses \texttt{cur\_vt}, \texttt{current\_task}, \texttt{thread\_table}, \texttt{vt} and \texttt{vt\_move\_cursor}. We provide three system calls

\begin{Verbatim}
\begin{syscall prototypes}\end{syscall prototypes}
\begin{verbatim}
void \texttt{syscall\_clrscr}\ ((\texttt{context\_t} \ *r));
void \texttt{syscall\_get\_xy}\ ((\texttt{context\_t} \ *r));
void \texttt{syscall\_set\_xy}\ ((\texttt{context\_t} \ *r));
\end{verbatim}
\end{syscall prototypes}

for these functions:

\begin{Verbatim}
\begin{ulix system calls}\end{ulix system calls}
\begin{verbatim}
#define \texttt{\_NR\_clrscr} 512
#define \texttt{\_NR\_get\_xy} 513
#define \texttt{\_NR\_set\_xy} 514
\end{verbatim}
\end{ulix system calls}

Defines:
\texttt{\_NR\_clrscr}, used in chunk 331.
\texttt{\_NR\_get\_xy}, used in chunk 331.
\texttt{\_NR\_set\_xy}, used in chunk 331.

As usual, the system call handlers evaluate the parameters by looking at the registers \texttt{EBX} and \texttt{ECX} (if there are any), then they call the above functions.
void syscall_clrscr (context_t *r) {
    // no parameters, no return value
    vt_clrscr ();
}

void syscall_get_xy (context_t *r) {
    // ebx: address of x position (char)
    // ecx: address of y position (char)
    vt_get_xy ((char*)r->ebx, (char*)r->ecx);
}

void syscall_set_xy (context_t *r) {
    // ebx: x position (char)
    // ecx: y position (char)
    vt_set_xy ((char) r->ebx, (char) r->ecx);
}

And we add those system calls to the system:

install_syscall_handler (__NR_clrscr, syscall_CLRSCR);
install_syscall_handler (__NR_get_xy, syscall_GETXY);
install_syscall_handler (__NR_set_xy, syscall_SETXY);

Uses __NR_CLRSCR, __NR_GETXY, __NR_SETXY, install_syscall_handler, syscall_CLRSCR, syscall_GETXY, and syscall_SETXY.

Via the user mode library we provide the functionality to processes:

void clrscr ( );
void get_xy (char *x, char *y);
void set_xy (char x, char y);

void clrscr ( ) {
    syscall1 (__NR_CLRSCR);
}
void get_xy (char *x, char *y) {
    syscall3 (__NR_GETXY, (int) x, (int) y);
}
void set_xy (char x, char y) {
    syscall3 (__NR_SETXY, (int) x, (int) y);
}

To make life easier for the application programmer (who cannot access the screen memory directly) we also provide functions which allow reading or writing the whole screen (that is: 24 lines of 80 characters; the last line on the 80 × 25 display is reserved for the operating system). For this purpose we implement the read_screen and write_screen functions and let applications call them via system calls.
<function prototypes 45a>+

- void read_write_screen (char *buf, boolean read_flag);
- void read_screen (char *buf);
- void write_screen (char *buf);

<function implementations 100b>+

- void read_write_screen (char *buf, boolean read_flag) {
  // if read_flag == true: read from screen, otherwise write
  int process_term = thread_table[current_task].terminal;
  char *video_address = (char*) vt[process_term].mem;
  if (read_flag) {
    memcpy (buf, video_address, VT_SIZE); // read the screen
  } else {
    memcpy (video_address, buf, VT_SIZE); // write the screen
    // current terminal?
    if (process_term == cur_vt)
      memcpy ((char*) VIDEORAM, video_address, VT_SIZE);
  }
}

void read_screen (char *buf) { read_write_screen (buf, true); }
void write_screen (char *buf) { read_write_screen (buf, false); }

Defines:
- read_screen, used in chunk 333e.
- read_write_screen, used in chunk 332d.
- write_screen, used in chunks 332 and 333.

Uses cur_vt 326a, current_task 192c, memcpy 596c, thread_table 176b, VIDEORAM 327b, vt 326a, and VT_SIZE 325a.

In the system call handlers we call read_write_screen32b instead of read_screen332b and write_screen332b, to save the extra function call:

<unix system calls 206e>+

- #define __NR_read_screen 515
- #define __NR_write_screen 516

Uses __NR_read_screen and __NR_write_screen.

<syscall functions 174b>+

- void syscall_read_screen (context_t *r) {
  // ebx: buffer address
  read_write_screen ((char *) r->ebx, true);
}

- void syscall_write_screen (context_t *r) {
  // ebx: buffer address
  read_write_screen ((char *) r->ebx, false);
}

Defines:
- syscall_read_screen, used in chunk 333a.
- syscall_write_screen, used in chunk 333a.
Uses context_t 142a and read_write_screen 332b.
Terminals

initialize syscalls:

\[
\begin{align*}
\text{install syscall handler} & (\_NR\_read\_screen, \text{syscall\_read\_screen}); \\
\text{install syscall handler} & (\_NR\_write\_screen, \text{syscall\_write\_screen}); \\
\end{align*}
\]

Uses \_NR\_read\_screen, \_NR\_write\_screen, install syscall handler 201b, syscall\_read\_screen 332d, and syscall\_write\_screen 332d.

Again, we add these to the library:

\[\text{ulixlib function prototypes}\]

\[
\begin{align*}
\text{void read\_screen (char \*buf);} \\
\text{void write\_screen (char \*buf);} \\
\end{align*}
\]

\[\text{ulixlib function implementations}\]

\[
\begin{align*}
\text{void read\_screen (char \*buf)} & \{ \text{syscall2 (\_NR\_read\_screen, (uint) buf);} \} \\
\text{void write\_screen (char \*buf)} & \{ \text{syscall2 (\_NR\_write\_screen, (uint) buf);} \}
\end{align*}
\]

Defines:

read\_screen, used in chunk 333e.
write\_screen, used in chunks 332 and 333.
Uses \_NR\_read\_screen, \_NR\_write\_screen, and syscall2 203c.

Applications can use read\_screen332b and write\_screen332b for scrolling. Here's a simple scroll function which scrolls the user mode part of the screen (lines 1–24) one line "up" (that means: the first lines disappears, and all other lines move up one line, leaving one blank line at the bottom)

\[\text{ulixlib function prototypes}\]

\[
\begin{align*}
\text{void scroll\_up (});} \\
\text{void scroll\_down (});}
\end{align*}
\]

\[\text{ulixlib function implementations}\]

\[
\begin{align*}
\text{void scroll\_up ()} & \{
\text{char buffer}[80*25*2]; // we reserve space for 25 (!) lines} \\
\text{word blank = 0x20 | VT\_NORMAL\_BACKGROUND; // blank character} \\
\text{read\_screen ((char*) buffer);} \\
\text{memset ((word*)((char*) buffer + 80*24*2), blank, 80);} \\
\text{write\_screen ((char*) buffer + 160);} \\
\}
\]

\[
\begin{align*}
\text{void scroll\_down ()} & \{
\text{char buffer}[80*25*2]; // we reserve space for 25 (!) lines} \\
\text{word blank = 0x20 | VT\_NORMAL\_BACKGROUND; // blank character} \\
\text{read\_screen ((char*) buffer + 160);} \\
\text{memset ((word*)((char*) buffer), blank, 80);} \\
\text{write\_screen ((char*) buffer);} \\
\}
\]

Defines:

scroll\_up, used in chunk 333d.
Uses memset 596c, read\_screen 332b 333c, VT\_NORMAL\_BACKGROUND 326b, and write\_screen 332b 333c.
For scrolling from inside the kernel, we provide a helper function that can “scroll” any screen-sized chunk of memory. In our case that is an area of 24 lines à 80 characters, each of which is 2 bytes large (24 × 80 × 2 = 3840 bytes), and scrolling it means to move lines 2–24 to lines 1–23 and empty line 24.

```c
void vt_scroll_mem (word *address) {
    word blank = ' ' | VT_NORMAL_BACKGROUND; // space + format
    memcpy (address, address + VT_WIDTH, (VT_HEIGHT-1) * VT_WIDTH * 2);
    memsetw (address + (VT_HEIGHT-1) * VT_WIDTH, blank, VT_WIDTH);
}
```

Defines: vt_scroll_mem, used in chunk 334b.
Uses memcpy 596c, memsetw 596c, VT_HEIGHT 325a, VT_NORMAL_BACKGROUND 326b, and VT_WIDTH 325a.

Note that this function uses pointer arithmetic: address is of type word*, i.e., a pointer to a 16-bit wide integer. That means that when we add e.g. VT_WIDTH 325a to address, the resulting address is actually 2 * VT_WIDTH 325a higher.

```c
void vt_scroll () {
    term_buffer *term;
    short int target_vt;
    if (scheduler_is_active) {
        target_vt = thread_table[current_task].terminal;
        term = &vt[target_vt];
    } else {
        target_vt = KERNEL_VT; // kernel: default write to 0
    }

    if (cur_vt == target_vt && csr_y >= VT_HEIGHT) {
        vt_scroll_mem ((word*)VIDEORAM);
        csr_y = VT_HEIGHT-1;
    }

    if (scheduler_is_active && term->y >= VT_HEIGHT) {
        vt_scroll_mem ((word*)term->mem);
        term->y = VT_HEIGHT-1;
    }
}
```

Defines: vt_scroll, used in chunk 335b.
Uses cur_vt 326a, current_task 192c, KERNEL_VT 328b, scheduler_is_active 276e, term_buffer 325b, thread_table 176b, VIDEORAM 327b, vt 326a, VT_HEIGHT 325a, and vt_scroll_mem 334a.
10.2 Terminals

10.2.1 Terminal Output

The next two functions

\[
\text{\langle function prototypes 45a\rangle} + \equiv
\]

\[
\begin{align*}
\text{void kputch (byte c);} \\
\text{void kputs (char* text);} \\
\end{align*}
\]

write a character or a string to the screen.

The kputch function is based on the scrn.c function of Bran’s kernel tutorial [Fri05] but was modified a lot.

\[
\text{\langle function implementations 100b\rangle} + \equiv
\]

\[
\begin{align*}
\text{void kputch (byte c) \{} \\
& \quad \text{// check if we're writing to current terminal} \\
& \quad \text{term_buffer* term;} \\
& \quad \text{short int target_vt;} \\
& \quad \text{word* where;} \\
& \quad \text{if (scheduler_is_active) \{} \\
& \quad \quad \text{target_vt = thread_table[current_task].terminal;} \\
& \quad \quad \text{term = &vt[target_vt];} \\
& \quad \text{\} else \{} \\
& \quad \quad \text{target_vt = KERNEL_VT; \quad \text{// kernel: default write to 0} } \\
& \quad \} \\
& \quad \text{switch (c) \{} \\
& \quad \quad \text{case '\b': \quad \text{// backspace, move cursor back} } \\
& \quad \quad \quad \text{if (cur_vt == target_vt) \{} \\
& \quad \quad \quad \quad \text{if (csr_x != 0) csr_x--; } \\
& \quad \quad \quad \text{\} } \\
& \quad \quad \quad \text{if (scheduler_is_active) \{} \\
& \quad \quad \quad \quad \text{if (term->x != 0) term->x--; } \\
& \quad \quad \quad \text{\} break; } \\
& \quad \quad \text{\} \\
& \quad \quad \text{case '\r': \quad \text{// carriage return, go back to first column} } \\
& \quad \quad \quad \text{if (cur_vt == target_vt) \{} \\
& \quad \quad \quad \quad \text{csr_x = 0;} \\
& \quad \quad \quad \text{\} } \\
& \quad \quad \quad \text{if (scheduler_is_active) \{} \\
& \quad \quad \quad \quad \text{term->x = 0;} \\
& \quad \quad \quad \text{\} break; } \\
& \quad \quad \text{\} \\
& \quad \quad \text{case '\n': \quad \text{// newline, go to next line, first column} } \\
& \quad \quad \quad \text{if (cur_vt == target_vt) \{} \\
& \quad \quad \quad \quad \text{csr_x = 0; csr_y++; } \\
& \quad \quad \quad \text{\} } \\
& \quad \quad \quad \text{if (scheduler_is_active) \{} \\
& \quad \quad \quad \quad \text{term->x = 0; term->y++; } \\
& \quad \quad \quad \text{\} break;} \\
& \quad \text{\} } \\
& \quad \text{\} } \\
& \quad \text{if (c \geq ' ') \{} \quad \text{// normal character} \\
& \quad \quad \text{if (cur_vt == target_vt) \{} \\
& \quad \quad \quad \text{where = textmemptr + (csr_y * 80 + csr_x);} \\
& \quad \quad \quad \text{\} } \\
& \quad \quad \text{\} } \\
& \quad \text{if (scheduler_is_active) \{} \\
& \quad \quad \text{where = (word*)term->mem + (term->y * 80 + term->x);} \\
\end{align*}
\]

\[
\text{\rangle 44a} < 332a \ 336a \] [335a]
\[
\text{\rangle 44b} < 334b \ 336b \] [335b]
*where = c | VT_NORMAL_BACKGROUND;
   term->x++; }

if (csr_x ≥ 80) { // end of line reached
   if (cur_vt == target_vt) { csr_x = 0; csr_y++; }
   if (scheduler_is_active) { term->x = 0; term->y++; }
 }

vt_scroll (); // scroll if necessary
if (cur_vt == target_vt) { vt_move_cursor (); }

// write to serial console
if (c == '\b') { // backspace
   uartputc ('\b'); uartputc (' '); uartputc ('\b');
} else uartputc (c);

void kputs (char *text) {
   while (*text != 0)
      kputch (*{text++});
}

Defines:
   kputch, used in chunks 324b, 417, 598a, 605b, 611b, and 613b.
   kputs, used in chunks 108, 115d, 121b, 335a, 603, 604b, 608b, and 610–13.
Uses cur_vt 326a, current_task 192c, KERNEL_VT 328b, scheduler_is_active 276e, term_buffer 325b,
   textmemptr 116c, thread_table 176b, uartputc 336b, vt 326a, vt_move_cursor 328a, VT_NORMAL_BACKGROUND 326b,
   and vt_scroll 334b.

For writing to the serial console, kputch335b, uses the helper function

\begin{quote}
\texttt{\textbf{void uartputc (int c);}}
\end{quote}

which sends a character to the I/O port \texttt{IO\_COM1}. This is useful for running Unix in the qemu PC emulator [B+14] which can display serial line output in the terminal window of the host machine, see also Section 10.4 on serial ports.

\begin{quote}
\texttt{\textbf{void uartputc (int c) \{}}
\texttt{  // taken from the xv6 operating system [CKM12], \texttt{uart.c}}
\texttt{  if (!uart[0]) return; \hspace{1em} // leave if we have no first serial port}
\texttt{  // wait until COM1 is ready to receive another byte}
\texttt{  for (int i = 0; i < 128 &amp; !(inportb (IO\_COM1+5) &amp; 0x20); i++)
\texttt{    outportb (IO\_COM1+0, c); \hspace{1em} // write the byte}}
\texttt{\}}
\end{quote}

Defines:
   uartputc, used in chunks 335b, 336a, and 598a.
Uses inportb 133b, IO\_COM1 344a, outportb 133b, and uart 344b.
10.2 Terminals

10.2.2 Status Line Management

The last line on the screen is reserved for the Unix status line. We provide two functions which let the kernel display status messages:

\[
\textit{function prototypes 45a} + = \\
\text{void set_statusline (char *text);} \\
\text{void _set_statusline (char *text, int offset);} \\
\]

There are two functions, \textit{set_statusline} and \textit{_set_statusline}, to set and update the status line. The first function, \textit{set_statusline}, always writes to the start of the status line, whereas \textit{_set_statusline} takes an extra position argument and can be used to update a small location somewhere in the middle of the line.

\[
\textit{function implementations 100b} + = \\
\text{void set_statusline (char *text)} \{ \textit{_set_statusline (text, 0)}; \} \\
\text{void _set_statusline (char *text, int offset)} \{ \\
\text{i = 0; \\
\text{uint videoaddress = VIDEORAM + VT_SIZE+2*offset; } // last line of video \\
\text{while (((*text != 0) && (i < 80)) } \\
\text{ \{ \\
\text{POKE (videoaddress + 2*i, *text); \\
\text{i++; text++; \\
\text{\}} \\
\} \\
\}
\]

 Defines:
- \textit{set_statusline}, used in chunks 276, 280a, 321a, 342b, 343b, and 512b.
- \textit{set_statusline}, used in chunks 337 and 608–10.
Uses \textit{POKE 117, VIDEORAM 327b, and VT_SIZE 325a}.

10.2.3 Initializing the Screen

When we described the kernel initialization in the \textit{main} function, we promised to define the code chunk \textit{\langle setup video \rangle} in this chapter—here it is: We use \textit{vt_clrscr} to clear the screen and \textit{set_statusline} to display the OS name and version.

\[
\textit{setup video} \Rightarrow \\
\text{vt_clrscr \;} \\
\text{set_statusline (UNAME);} \\
\text{printf ("%s Build: %s\n", UNAME, BUILDDATE);} \\
\]

 Uses \textit{BUILDDATE 35a, printf 601a, set_statusline 337b, UNAME 35a, and vt_clrscr 329b}.

Remember that we’ve set \textit{UNAME} and \textit{BUILDDATE} at the very beginning.
10.3 System Timer

clock chip A central hardware component is the clock chip which regularly causes a timer interrupt. By adding a timer handler, the kernel can regularly check whether some administrative action is necessary. The most important action is calling the scheduler: Without the timer handler we would have to live without preemptive multi-tasking.

There are many other tasks for the timer handler, for example we will use it to keep track of time in our system. Every time the timer handler runs, we will increment a system_ticks variable that must be initialized at system start. system_time will be set to Unix time (the seconds since the Unix epoch, 1 January 1970, 00:00:00 UTC) so that we can properly display the date and time and update timestamps in the filesystem.

```c
unsigned int system_ticks = 0; // updated 100 times a second
unsigned int system_time; // unix time (in seconds)
```

Defines: system_ticks, used in chunks 306d, 311a, 342, and 343b. system_time, used in chunks 342c, 343b, 475c, 478b, and 605a.

10.3.1 Setting the Frequency

timer frequency Initially the clock chip is pre-set to a weird frequency (≈ 18.222 Hz), we change that to 100 Hz when the system starts. The function

```c
void timer_phase (int hz);
```

adjusts the timer’s frequency: It first announces that it wants to set the frequency by sending 0x36 to port IO_CLOCK_COMMAND and then sends the lower and the higher eight bits of the divisor 1193180 / hz to the port IO_CLOCK_CHANNEL0 which is responsible for configuring timer 0 (the system timer) [vG94, p. 794].

```c
#define IO_CLOCK_COMMAND 0x43
#define IO_CLOCK_CHANNEL0 0x40
```

with

```c
int divisor = 1193180 / hz; // calculate divisor
outportb (IO_CLOCK_COMMAND, 0x36); // set command byte 0x36
outportb (IO_CLOCK_CHANNEL0, divisor & 0xFF); // set low byte of divisor
outportb (IO_CLOCK_CHANNEL0, divisor >> 8); // set high byte of divisor
```
10.3 System Timer

Defines:

 IO_CLOCK_CHANNEL0, used in chunk 338c.
 IO_CLOCK_COMMAND, used in chunk 338c.

When the kernel runs through the initialization steps, we let it set the frequency and install the timer interrupt handler (whose implementation we will discuss soon) for interrupt number IRQ_TIMER (0). While we’re at it, we also query the current date and time.

\[
\text{install the timer } \equiv \text{timer_phase (100); // set timer to 100 Hz (100 interrupts/second)} \]

install_interrupt_handler (IRQ_TIMER, timer_handler);
enable_interrupt (IRQ_TIMER);

\[
\text{read date and time from CMOS } \equiv \]

Uses enable_interrupt 140b, install_interrupt_handler 146c, IRQ_TIMER 132, timer_handler 342b, and timer_phase 338c.

10.3.2 Reading the Date and Time

Since it is a somewhat related task, we also query the PC’s CMOS chip to find out what date and time it is [vG94, p. 746–747]:

\[
\text{constants } \equiv (44a) \text{ define IO_CMOS_CMD 0x70} \]

define IO_CMOS_DATA 0x71

Defines:

 IO_CMOS_CMD, used in chunks 339d and 552c.
 IO_CMOS_DATA, used in chunks 339d and 552c.

\[
\text{global variables } \equiv \text{unsigned long system_start_time } = 0; \]

Defines:

 system_start_time, used in chunks 340b and 342c.

\[
\text{read date and time from CMOS } \equiv \]

// code adapted from http://wiki.osdev.org/CMOS

outportb (IO_CMOS_CMD, 0); byte second = inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD, 2); byte minute = inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD, 4); byte hour = inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD, 7); byte day = inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD, 8); byte month = inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD, 9); word year = inportb (IO_CMOS_DATA);
outportb (IO_CMOS_CMD, 0x32); word century = inportb (IO_CMOS_DATA);

Uses hour, inportb 133b, IO_CMOS_CMD 339b, IO_CMOS_DATA 339b, and outportb 133b.

The values that the CMOS chip returns are BCD-encoded (binary-coded decimal; each half-byte encodes a decimal digit, from 0 = 0000b to 9 = 1001b), so they have to be converted so that they make sense: To convert one BCD byte into a proper number, take the upper half times 10 ((bcd >> 4) * 10) and add the lower half (bcd & 0x0f):
#define CONVERT_BCD(bcd) (((bcd) >> 4) * 10) + (bcd & 0x0f))

Defines:
CONVERT_BCD, used in chunk 340b.

second = CONVERT_BCD (second); minute = CONVERT_BCD (minute);
hour = CONVERT_BCD (hour); day = CONVERT_BCD (day);
month = CONVERT_BCD (month); century = CONVERT_BCD (century);
year = CONVERT_BCD (year) + 100 * century;

The year is only stored with two digits (e.g., 14 for the year 2014), so we have to add 100 * century. Some CMOS chips return the hour in "12 hour time". For example, they would represent the hour value 23 as 11 and set the highest bit to indicate "pm" time. A formula that can cope with both types of BIOS is

hour = (((hour & 0x0F) + ((hour & 0x70) / 16) * 10)) | (hour & 0x80);  

—we do not use it since qemu returns "24 hour time". We need functions

that convert between Unix time (seconds since 01/01/1970) and a time structure with year, month, day, hour, minute and second. You can skip the implementation of the following two functions since they are neither pretty to look at nor do they tell you anything about operating systems.
Defines:
unixtime, used in chunk 340.
Uses hour and ulong 46b.

The function rev_unixtime is not used in the ULIx kernel at all, however the user mode program ls uses it, so we show it here for completeness. yearlength is a helper function that returns the length of a year (either 342 or 345 for a leap year).

```
short yearlength (short year) {
    int res = 364;
    if ((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0) res++;
    return res;
}

void rev_unixtime (ulong utime, short *year, char *month, char *day, char *hour, char *minute, char *second) {
    char days_per_month[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int days = utime / (60*60*24); char sec = utime % 60;
    char min = (utime/60) % 60; char hou = (utime/(60*60)) % 24;

    int yy = 1970;
    if (days > 15706) { // speed up calculation for 2013 or later
        days -= 15706;
        yy += 43;
    }

    for (;;) {
        int l = yearlength (yy);
        if (days ≥ l) {
            yy++;
            days -= (l+1); // distance between two years is l+1, not l
        } else break;
    }

    int mon = 1;
    for (;;) {
        int l = days_per_month[mon];
        if ((l == 2) && (yearlength (yy) == 365)) l++;
        if (days ≥ l) {
            mon++;
            days -= l;
        } else break;
    }

    days++;
    *year = yy; *month = mon; *day = days; // return results
    *hour = hou; *minute = min; *second = sec;
}
```

Uses hour, min, sec, and ulong 46b.
10.3.3 Implementation of the Timer Handler

This is our timer interrupt handler:

\[
\text{\texttt{\# function prototypes}} \quad \text{\texttt{\small \begin{aligned}
\text{\texttt{\quad \texttt{\small void}}
& \text{\texttt{\small \quad \texttt{\small timer_handler}}
& \text{\texttt{\small \quad \texttt{\small (context_t} *r)};}
\end{aligned}}}}
\]

It executes all the timer tasks (as defined in the code chunk \langle \text{timer tasks} \rangle) and updates the status line.

\[
\text{\texttt{\# function implementations}} \quad \text{\texttt{\small \begin{aligned}
\text{\texttt{\quad \texttt{\small void}}
& \text{\texttt{\small \quad \texttt{\small timer_handler}}
& \text{\texttt{\small \quad \texttt{\small (context_t} *r)} \}\texttt{\small ;}
\end{aligned}}}}
\]

\[
\begin{aligned}
\text{\texttt{\quad \texttt{\small char buf[80]; \quad \texttt{\small // temporary buffer, can be used by all timer tasks}}} \\
\text{\texttt{\quad \texttt{\small \quad \texttt{\small // show current terminal, free frames, current_as}}} \\
\text{\texttt{\quad \texttt{\small \quad \texttt{\small sprintf ((char*})&buf, ”tty%d FF=%04x AS=%04d”, cur_vt, free_frames, current_as);}}} \\
\text{\texttt{\quad \texttt{\small \quad \texttt{\small \quad \texttt{\small _set_statusline ((char*})&buf, 48);}}} \\
\end{aligned}
\]

Defines:
\texttt{timer_handler}, used in chunks 339a and 342a.

Uses \texttt{_set_statusline, context_t, cur_vt, current_as, free_frames, and sprintf}.

10.3.4 Tasks for the Timer

We need the timer handler to do several things which we collect in the \langle \text{timer tasks} \rangle code chunk. The first and easiest task is to modify the system uptime:

\[
\text{\texttt{\texttt{\texttt{\langle \text{timer tasks} \rangle}}}} \\
\text{\texttt{\quad \texttt{\small system_ticks++; \quad \texttt{\small // one more timer interrupt}}} \\
\text{\texttt{\quad \texttt{\small system_time = (uint)(system_ticks/100) + system_start_time; \quad \texttt{\small // frequency: 100 Hz}}} \\
\text{\texttt{\end{aligned}}}}
\]

Uses \texttt{system_start_time}, \texttt{system_ticks}, and \texttt{system_time}.

Next, it calls the scheduler. It also displays a quickly changing progress character in the right top corner of the screen so that users can check that the scheduler is still active. If those signs stop spinning, something has gone wrong.

\[
\text{\texttt{\texttt{\texttt{\langle \text{timer tasks} \rangle}}}} \\
\text{\texttt{\quad \texttt{\small char sched_chars[] = \"/-\\\"; \quad \texttt{\small // scheduler activity}}} \\
\text{\texttt{\quad \texttt{\small static short sched_c = 0; \quad \texttt{\small // next character to display}}} \\
\text{\texttt{\quad \texttt{\small if (system_ticks % 5 == 0) \{}}} \\
\text{\texttt{\quad \texttt{\small \quad \texttt{\small // cycle }\{}\texttt{\small /-\\\}-\texttt{\small to show scheduler calls in upper right corner}}} \\
\text{\texttt{\quad \texttt{\small \quad \texttt{\small POKE (VIDEORAM + 79*2, sched_chars[sched_c]);}}} \\
\text{\texttt{\quad \texttt{\small \quad \texttt{\small sched_c++; sched_c %= 4;}}} \\
\text{\texttt{\quad \texttt{\small \quad \texttt{\small scheduler }\{r, \texttt{\small SCHED_SRC_TIMER}\};}}} \\
\text{\texttt{\quad \texttt{\small \}}} \\
\text{\texttt{\end{aligned}}}}
\]

Uses \texttt{POKE}, \texttt{sched_chars}, \texttt{sched_c}, \texttt{SCHED_SRC_TIMER}, \texttt{scheduler}, \texttt{system_ticks}, and \texttt{VIDEORAM}.
As mentioned earlier, the timer handler does not call scheduler if a resign action is currently active. When it does call the scheduler, it provides a second SCHED_SRC_TIMER argument to indicate that it was the timer who called. (The alternative is that syscall_resign called the scheduler which it announces by using SCED_SRC_RESIGN.)

\begin{verbatim}
#define SCHED_SRC_TIMER 0
#define SCHED_SRC_RESIGN 1
\end{verbatim}

Defines:
SCHED_SRC_RESIGN, used in chunks 216b, 221a, and 278a.
SCHED_SRC_TIMER, used in chunk 342d.

In the status line at the bottom of the screen we display the current time; we want to update this display approximately every other second:

\begin{verbatim}
short int sec,min,hour;
if (system_ticks % 100 == 0) {
  // Every 100 clocks (approx. 1 second)
  hour = (system_time/60/60)%24;    // display the time
  min = (system_time/60)%60;
  sec = system_time%60;
  sprintf ((char*)&buf, "%02d:%02d:%02d", hour, min, sec);
  _set_statusline ((char*)&buf, 72);
}
\end{verbatim}

Uses _set_statusline, hour, min, sec, sprintf, system_ticks, and system_time.

There are only two further places in the book where \textit{timer tasks} gets an addition; we have decided not to place them in this chapter but at the places where the need for them arises. These are the chunks:

- Updating the counters for the page replacement code, p. 306
- Releasing the swapper_lock so that the swapper process can enter the next loop, p. 311

### 10.4 Serial Ports

Ulix supports two serial ports. It uses the first one to copy the regular output to a serial console and also writes kernel debug messages to that console. When running in a PC emulator which can redirect serial ports, these messages can be displayed in the terminal window from which Ulix was started. The Makefile in the bin-build/ directory calls qemu with a -serial option and a pipe into the tee command

```
-qemu -serial mon:stdio | tee ulix.output
```

to simultaneously display the serial output in the terminal window and write it into a log file ulix.output.
The following code is borrowed from the xv6 operating system [CKM12], especially from source files uart.c and console.c. We modified the uartinit function so that it can deal with two serial ports.

Several I/O ports are used for sending data to the serial ports or reading from them, the base port numbers are the following:

```c
#define IO_COM1 0x3f8
#define IO_COM2 0x2f8
```

We use the array `uart` to keep track of available ports.

```c
static int uart[2];  // do we have serial ports?
```

```c
void uartinit (int serport) {
    char *p;
    word io_com, irq;
    switch (serport) {
        case 1: io_com = IO_COM1; irq = IRQ_COM1; break;
        case 2: io_com = IO_COM2; irq = IRQ_COM2; break;
        default: return;
    }

    outportb (io_com+2, 0);  // Turn off the FIFO
    // set 9600 baud, 8 data bits, 1 stop bit, parity off.
    outportb (io_com+3, 0x80);  // Unlock divisor
    outportb (io_com+0, 115200/9600);
    outportb (io_com+1, 0);
    outportb (io_com+3, 0x03);  // Lock divisor, 8 data bits.
    outportb (io_com+4, 0);
    outportb (io_com+1, 0x01);  // Enable receive interrupts.

    // If status is 0xFF, no serial port.
    if (inportb (io_com+5) == 0xFF) { return; }
    uart[serport-1] = 1;

    // Acknowledge pre-existing interrupt conditions; enable interrupts.
    inportb (io_com+2);
    inportb (io_com+0);
    enable_interrupt (irq);
}
```

Defines:
- `uartinit`, used in chunk 345.
- `IO_COM1`, used in chunks 336b, 344c, 346a, and 519d.
- `IO_COM2`, used in chunks 344c, 345c, and 519d.
- `uart`, used in chunks 336b, 339c, 344c, 346b, and 519d.

Uses `enable_interrupt`, `inportb`, and `outportb`.

---

[44a] <333b 363>a
[44b] <339c 363>b
[44c] <342b 345>c
[44d] <343a 363>b
[44e] <344a 344>b
We start the serial port when booting:

\[
\text{\texttt{(setup serial port 345a)} \equiv} \quad (44b) \quad [345a] \\
\text{\quad \texttt{uartinit (1);} }
\]

Uses \texttt{uartinit 344c}.

To simplify disk access, we provide something we call a \textit{serial hard disk} (see Chapter 13.4). Using the second serial port (of a virtual machine) we allow \texttt{Ulix} to connect to an external process which imitates a hard disk controller. \texttt{Ulix} can send simple commands to that process and in return will be served with data (1024 byte sectors) out of a hard disk image file.

\[
\text{\texttt{(function prototypes 45a)} +} \equiv \quad (44a) \quad <342a \ 361a> \quad [345b] \\
\text{\quad \texttt{void uart2putc (int);}}
\]

\[
\text{\texttt{(function implementations 100b)} +} \equiv \quad (44a) \quad <344c \ 361c> \quad [345c] \\
\text{\quad \texttt{void uart2putc (int c) \{}}
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\texttt{// taken from the xv6 operating system [CKM12], uart.c}}}
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\texttt{if (!uart[1]) return; \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad // leave if we have no second serial port}}}
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\texttt{// wait until COM2 is ready to receive another byte}}}
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\texttt{for (int i = 0; i < 128 && !(inportb (IO_COM2+5) & 0x20); i++) ;}}}
\]

\[
\text{\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{\texttt{outportb (IO_COM2+0, c); \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad // write the byte}}}
\]

\[
\text{\texttt{\}}} 
\]

Defines:

\texttt{uart2putc}, used in chunks 345b, 517, 518, and 521a.
Uses \texttt{inportb 133b, IO_COM2 344a, outportb 133b, and uart 344b}.

(This function is almost identical to \texttt{uartputc336b} except that it uses \texttt{IO_COM2} instead of \texttt{IO_COM1}, see page 336.)

\[
\text{\texttt{(setup serial hard disk 345d)} \equiv} \quad (45c) \quad 520a \quad [345d] \\
\text{\quad \texttt{uartinit (2);} }
\]

Uses \texttt{uartinit 344c}.

The interrupt handler for the second serial port will be implemented in Chapter 13.4. We need no handler for the first port because we only write to it.
In previous chapters, we had a look at the basic abstractions implemented by the operating system: virtual memory abstracting physical memory and virtual processors (threads) abstracting physical processors. Virtual processors may now execute concurrent programs in which the concurrent threads often have to interact in some specific way. There are two basic interaction patterns:

- A **competitive interaction pattern** occurs when two threads want to perform the same operation, however only one of them is permitted to do so at the same time. This means that one thread must go first and the other must wait until the first has finished his operation. Classic examples of this interaction pattern are accesses to exclusive resources or critical code sections. In this interaction pattern, the competing threads often don’t know of each other so that some mediator (i.e., the operating system) has to synchronize the threads in a convenient and fair manner.

- A **cooperative interaction model** occurs when two threads know each other and want to exchange information in a well-defined way. This interaction pattern occurs for example in client/server-type systems where one thread requests information which another thread provides.

The question for Ulix is: Which thread synchronization abstractions make sense and how can they be implemented?

Depending on the basic implementation mechanisms, we distinguish between memory-based synchronization abstractions and message-based synchronization abstractions. The most relevant ones for us are the former ones which are based on the availability of shared memory between threads. They can therefore be utilized in those operating systems using service combinations which primarily depend on the availability of shared memory.
Outline

We first present the central abstraction of competitive thread synchronization in Section 11.1. Then we go through different ways of implementing critical sections. While there are some purely software-based methods for achieving mutual exclusion that require no hardware support, they are rarely used; thus we immediately turn to more low-level and more practical synchronization techniques based on special hardware operations in Section 11.2.

We then climb again up the abstraction ladder and look at a higher-level concepts: Semaphores can be regarded as an operating system service which is more usable than low-level hardware. They are treated in Section 11.3. We will discuss the implementation of these concepts in Ulix as we go along. We present a standard implementation of semaphores based on atomic hardware operations.

Then we look at a specialization of semaphores, the mutexes (or locks), and show their implementation in Ulix in Sections 11.4 (for the kernel) and 11.5 (for threads in user mode). Finally, Section 11.6 discusses the important topic of kernel-level synchronization which is needed in situations where interrupt handlers and threads share common data—in those situations we cannot use blocking mutexes or semaphores because an interrupt handler must not block.

11.1 Critical Sections

We start with explaining the central concept: the critical section.

11.1.1 The Case of the Lost List Element

Consider an implementation of a linked list. This could for example be the implementation of the ready queue within the dispatcher (for the real implementation see Section 6.2.2). Imagine a list element consists of the real content of the element together with a pointer to the next list element. Adding an item to the front of the queue is usually implemented like add_to_front does it in the following example program and as is illustrated in Figure 11.1.

![Figure 11.1: Example of adding an element to the front of a linked list.](image-url)
The next pointer of the new element is set to the “old” front of the list. Second, the global list pointer is set to the “new” front of the list. If you follow the final pointer structure you will see that the new list element has been correctly inserted at the front of the list.

The claim is now that the list implementation from above can cause problems if multiple threads try to put different elements into the list at the same time. To see this, consider Figure 11.2. There, two threads \( T_1 \) and \( T_2 \) invoke the implementation of `add_to_front` from above at almost the same time. The scheduling of the two threads is somewhat unfortunate in that \( T_1 \) is interrupted after the first operation, then \( T_2 \) adds its element, and then thread \( T_1 \) can finalize its insertion by executing the second operation. In total there are four pointer assignments, which are reflected in the figure. If you follow the final pointer structure of the ready queue, you will see that one element (namely that of thread \( T_2 \)) has been lost: It is not contained in the list anymore.

The reason for this is the unfortunate scheduling of the machine instructions. Operation 2 of thread \( T_1 \) overwrites the effect of the two operations of thread \( T_2 \), because it implicitly assumes that nothing has happened after it executed its own operation 1. These problems would have been avoided if there were a guarantee that whenever some thread executes operation 1 it can also execute operation 2 without being interrupted.

### 11.1.2 Defining Critical Sections

A **critical section** is a sequence of instructions of a program which access shared resources. In the example above, the linked list is the *shared resource*. Manipulation of shared re-
Figure 11.2: Concurrent threads trying to add two elements to a linked list: The list can lose elements when operations are interleaved in a special way.

sources should be protected by an entry and exit protocol. These should guarantee mutual exclusion between critical sections. This is defined as follows:

**Definition 1 (mutual exclusion)** At any time there is at most one thread executing within its critical section.

Note that critical sections are something very abstract. They have a meaning at almost any level of abstraction, be it operating system, user program or programming language level. When dealing with critical sections it is merely necessary to mark the beginning and the end of the critical sections. The runtime system must then guarantee that no two critical sections at the same level of abstraction are executed concurrently.

In the following code examples, we mark beginning and end of critical sections with the two macros `ENTER_MUTEX` and `EXIT_MUTEX`. This is an abbreviation for entering and exiting mutual exclusion. So if we write our list operation from above again, we should mark the critical section in the following way:

```c
void add_to_front (element **first, element *e) {
    ENTER_MUTEX ();
    e->next = *first;  // operation 1
    *first = e;       // operation 2
    EXIT_MUTEX ();
}
```

We will learn about many ways to implement critical sections in this chapter. For the time being, imagine a global token which must be acquired before a thread can enter its critical section.

What we want to achieve is the behavior that you can see in Figure 11.3: Assume that there are two threads which share a resource, e.g., a memory location in the process that both threads belong to. Both threads contain code that performs an update on that memory address: It reads the value stored at the address, performs some calculation and
then writes back a new value to the same location. The whole code range from reading it in to writing it back is the critical section, and we want to make sure that they cannot overlap, turning the code block into an atomic action.

One of the threads (in the figure, it is thread A) will first reach the entry point of its critical section. Before it enters, it calls \texttt{ENTER\__MUTEX}. Since at that time no other thread is in its own critical section, it can enter. Shortly afterwards, the other thread (thread B) also arrives at the entry point of its critical section: It must not pass, because thread A is still executing inside the critical section. Since it cannot continue, it will block.

After some time has passed, thread A finishes the work in the critical section and calls \texttt{EXIT\__MUTEX}. Now we can let thread B pass and enter its critical section. Later it finishes the work and also leaves, calling \texttt{EXIT\__MUTEX}, too.

You can think of the mutex as some global token that only one of the threads can possess and which is required to enter the critical section. Application programmers arrive at this situation all the time when they write multi-threaded programs, they use the synchronization features that the operating system provides, and our task is to implement this mechanism.

Figure 11.3: Simple example of mutual exclusion between two threads. Using a global token, one thread has to wait until the other thread returns the token to enter its critical section.
11.2 Hardware-based Synchronization

In this section we will consider synchronization based on explicit hardware support. We will look at the simplest thinkable mechanism first and then at more refined ways which are based on special CPU operations.

11.2.1 Disabling Interrupts

The simplest way to achieve mutual exclusion on a single CPU is to switch off the interrupts, which is also often called interrupt masking. Every modern multi-purpose CPU which has an interrupt mechanism allows to disable certain or all interrupts. The effect is that the interrupt handler is not invoked when the interrupt is signaled. Hence, asynchronous interrupts, which are usually the source for non-atomicity in critical sections, can be effectively eliminated.

Conceptually, every CPU should offer instructions like `INTERRUPTS_OFF` and `INTERRUPTS_ON`. We have already defined code chunks ⟨disable interrupts 47a⟩ and ⟨enable interrupts 47b⟩ which perform these tasks on an Intel x86 processor via the assembler instructions cli (clear interrupt flag) and sti (set interrupt flag).

Whenever a kernel programmer needs to ensure mutual exclusion of a critical section in system mode, he will now have to write the following.

\[
\text{⟨disable interrupts 47a⟩} \\
\text{// critical section} \\
\text{⟨enable interrupts 47b⟩}
\]

Note that masking interrupts can only be performed in system mode. (If normal programs could invoke the interrupt masking operations in user mode then they could monopolize the CPU.) Also, interrupts should only be disabled for relatively short periods of time. Otherwise, interrupts which are only flagged for a certain period of time like asynchronous I/O interrupts could be missed, leading to a possible lost wakeup (discussed later in Section 11.6.4.6). So overall, disabling interrupts is only advisable for rather short code sections within the kernel. Another disadvantage of this mechanism is that it only works for monoprocessor systems since turning off interrupts on one CPU does not affect the code executed on another CPU which could access shared memory data structures concurrently.

A trick makes is possible to improve the situation slightly: Using a global bit `busy` as a lock, we can extend the duration within a critical section without losing interrupts. The idea is to use the global lock bit as an indication whether some thread is within the critical section and just use interrupt masking to access this bit. The entry and exit protocols `ENTER_MUTEX` and `EXIT_MUTEX` for critical sections can then be programmed as follows.
11.2 Hardware-based Synchronization

Example: mutual exclusion using global lock bit and interrupt masking

```c
global boolean busy = false; // no thread in critical section

void ENTER_MUTEX() {
    disable interrupts 47a
    while (busy == true) {
        enable interrupts 47b
        NOP; // briefly leave interrupts on
disable interrupts 47a
    }
    busy = true; // I am in the critical section
    enable interrupts 47b
}

void EXIT_MUTEX() {
    busy = false; // I've left the critical section
}
```

Two processes wishing to enter their critical sections will “race” for the lock bit during the entry protocol. The process which is able to switch off interrupts first will be able to grab the lock bit (in case it is free). If it is not free, some other process is in its critical section. So we have to turn on the interrupts at least for a short period of time to allow that process to interrupt and exit the critical section.

11.2.2 Using Special Hardware Instructions

Most processors today offer machine instructions which are specially tailored towards synchronization so that it can be achieved without having to mess around with the interrupts. The most common such instructions are either called test-and-set or lock. They are designed in such a way that mutual exclusion can be achieved by “grabbing a token”—that is similar to the use of the global lock bit above.

11.2.2.1 Test-and-Set

Assume you have a global lock bit `busy` which is initially false. The test-and-set instruction takes two arguments: the first is the name (or address) of the global lock bit, the second is a local variable. Invoking `Test-and-Set (&busy, &local)` then results in the following two actions performed as one atomic (i.e., uninterrupted) operation:

1. The value of `locked_bit` is copied into `local` (“test”), and
2. the `locked_bit` is set to true (“set”).

In pseudocode this can be expressed as:

```c
void Test-and-Set (*busy, *local) {
    *local = *busy;
    *busy = true;
}
```
The idea of this operation is that after it has been performed you can safely check whether you have “grabbed the token” or not. If you have grabbed the token, then the result of the operation (i.e., the value stored in local) should be false since it reflects the value of busy before it was set to true.

As an example for a “real” Test-and-Set operation, here is the description of the gcc built-in function __sync_lock_test_and_set [Int01, section 7.4.5, p. 61] which can be used in the way described above. The semantics of the function is as follows:

\[
\begin{align*}
\langle \text{compiler-internal functions} \rangle \equiv \\
\text{int } \_\text{sync\_lock\_test\_and\_set} (\text{int *variable, int value}) \{ \\
\quad \text{int tmp = *variable;} \quad \text{// save old value} \\
\quad \quad \text{*variable = value;} \quad \text{// set new value} \\
\quad \text{return tmp;} \quad \text{// return old value} \\
\}
\end{align*}
\]

The compiler (and in the end the processor) guarantees that all of this is executed atomically.

### 11.2.2.2 Lock Instruction

Another common machine instruction you can find is called Lock. Our presentation here follows Nehmer and Sturm [NS01] who introduce it in the form of a boolean function which implicitly refers to the global lock bit busy.

In pseudocode, Lock does the following:

```c
boolean Lock () { 
    tmp = busy;
    busy = true;
    return tmp;
}
```

In effect, Lock does the same as Test-and-Set in that it copies the value of busy before it is set to true and then returns this value. Note again that all this is done in an uninterruptible way.

### 11.2.2.3 Spin Locks

Now we can implement ENTER_MUTEX and EXIT_MUTEX without having to use privileged hardware instructions. We first have a look at the implementations based on Test-and-Set.

```c
void ENTER_MUTEX() { 
    repeat { 
        Test-and-Set (&busy, &local);
    } until (local == false);
}
```
void EXIT_MUTEX() {
    busy = false;
}

Here is the implementation based on Lock.

(example: synchronization using Lock 355)
void ENTER_MUTEX() {
    while (Lock () == true) ; // loop over empty instruction
}

void EXIT_MUTEX() {
    busy = false;
}

Note that both implementations do not require privileged machine instructions, thus such a mechanism could be implemented completely in user mode, for example as part of a library that supplies service functions for threads.

The construction using Test-and-Set and Lock in the preceding examples is called a spin lock. In such a spin lock, threads waiting to enter their critical section must “spin” in a loop until they are allowed to enter. A spin lock is a form of busy waiting which is often encountered in low-level synchronization. Busy waiting however is a very inefficient form of waiting since CPU cycles are used up without actually contributing to any form of computation. Imagine how many machine instructions a 3 GHz CPU spinning in a loop could have donated to some computation. So similar to interrupt masking, spin locks are only allowed if critical sections are relatively short. The advantage of spin locks over interrupt masking however is that they can be performed without switching to kernel mode.

11.2.3 Monoprocessor vs. Multiprocessor Synchronization

To achieve mutual exclusion on a monoprocessor system, it is sufficient to turn interrupts off when entering and turning them on again when leaving the critical section. We will illustrate this strategy for performing mutual exclusion within Ulix in Section 11.6. As noted above, however, simply turning interrupts off is not sufficient on a multiprocessor system because disabling interrupts on one CPU does not prevent another CPU from accessing a shared data structure. In a multiprocessor system we additionally have to use spin locks. The strategy is as follows:

1. First we achieve local mutual exclusion per CPU by disabling interrupts.
2. Then we go into a spin lock to achieve global mutual exclusion over all CPUs.

Is achieving mutual exclusion at the lowest level in multiprocessor systems necessarily as complicated as this? Find out yourself by solving exercise 30.

It is generally advisable to avoid busy waiting whenever possible. At the lowest level of abstraction (e.g., synchronizing CPUs on the hardware level) busy waiting cannot be totally avoided. However, on higher levels of abstraction it generally can be avoided using more abstract synchronization primitives like semaphores.
11.2.4 Nested Critical Sections

In processor systems that support multiple *interrupt levels* turning off interrupts is not as easy as it may seem because interrupt handlers (and therefore critical sections) can be invoked in a nested fashion.

Assume for example, an interrupt handler at interrupt level 3 is executed on a CPU. During execution of that handler all interrupts at level 3 or lower are disabled, however, an interrupt at higher priority 5 may kick in and its handler be executed. It disables interrupts up to level 5. However, when this interrupt handler returns, it would be a bad idea to enable interrupts altogether. It rather must *restore the interrupt level* that was active *before* the interrupt occurred.

A similar situation happens if (on purpose or by accident) one critical section is declared within another such as in the following code example:

```c
f() {
    // higher level critical section
    ENTER_MUTEX();
    g();
    EXIT_MUTEX();
}

g() {
    // lower level critical section
    ENTER_MUTEX();
    // do something critical
    EXIT_MUTEX();
}
```

If we use interrupt masking as synchronization primitive and these markers are nested, it must be assured that `EXIT_MUTEX` enables interrupts only when it is called in `f()` and not in `g()`. The general rule is: The `EXIT_MUTEX` must restore the interrupt level that was active before its corresponding `ENTER_MUTEX` was called. In systems such as Ulix that do not distinguish interrupt levels (i.e., interrupts are either on or off completely), inner critical sections are superseded by outer critical sections, i.e., the outermost critical section disables interrupts and finally enables them again. All inner critical sections do not change anything with the interrupt settings.

We now show how to realize such a “nestable” `ENTER_MUTEX` and `EXIT_MUTEX` in code. We assume a system (such as Ulix) that does not distinguish interrupt levels (i.e., interrupts are either on or off completely). In such environments storing the prior interrupt level burns down to storing a counter representing the nesting level. We use a global variable to store the current level of nesting.
11.2 Hardware-based Synchronization

\[
\text{global variables (unused) 357a} \equiv \\
\text{int if_nested_level = 0;} \\
\text{Defines: if_nested_level, used in chunk 357.}
\]

Whenever we enter and exit a critical section, we flag that code appropriately. Here is a first implementation that assumes that interrupts are on when calling the chunk for the first time. It disables the interrupts and increments the nesting level. Note that disabling interrupts using cli is idempotent. We only turn interrupts back on again if the nesting level has reached its initial value again.

\[
\text{nestable begin critical section (first version) 357b} \equiv \\
\text{disable interrupts 47a} // invoke cli \\
\text{if_nested_level++;} \\
\text{Uses if_nested_level 357a.}
\]

\[
\text{nestable end critical section (first version) 357c} \equiv \\
\text{if_nested_level--;} \\
\text{if (if_nested_level == 0) { \\
\text{enable interrupts 47b} // invoke sti 
\text{}}}
\]

\[
\text{nestable begin critical section 357d} \equiv \\
\{ // create scope for scope-local variable eflags \\
\text{int eflags; } \\
\text{asm volatile ( \\
\text{"pushf \n" // push EFLAGS} \\
\text{"cli \n" // disable interrupts} \\
\text{"movl (%%esp), %0 \n" // copy to eflags variable} \\
\text{"addl $4, %%esp \n" // restore stack pointer : "} \\
\text{"=r"(eflags)) ; \\
\text{if (if_nested_level == 0)} \\
\text{if_state = (eflags >> 9) & 1; // bit 9 of EFLAGS is IF} \\
\text{if_nested_level++;} 
\}
\]

\[
\text{nestable end critical section 357e} \equiv \\
\text{if_nested_level--;} \\
\text{if (if_nested_level == 0 \&\& if_state == 1) { \\
\text{enable interrupts 47b} 
\text{}}}
\]

\[
\text{In exercise 31 we discuss the case of nestable critical sections on multiprocessor systems.}
\]
### 11.3 Semaphores

As seen above in Section 11.2.2, the simplest way of implementing blocking is *busy waiting*. However, that is a rather inefficient way to implement blocking as it consumes CPU cycles that could have been used for threads that are ready to run. We can avoid it by offering the right synchronization abstractions within the operating system. The operations `ENTER_MUTEX` and `EXIT_MUTEX` can, for example, then directly influence the state of threads. This is depicted in Figure 11.4 where three threads are scheduled onto two CPUs. In the example, thread A enters its critical section while running on CPU 1 and thread B running on CPU 2 calls `ENTER_MUTEX`. Because A is already in its critical section, B must block. Instead of waiting actively in a loop, it could go to sleep (change its state to `blocked`) and allow a different ready-to-run thread C to run on CPU 2.

The most popular abstraction for synchronization in operating systems is the *semaphore*. The name stems from a special type of signal used in railway systems. There, a critical section is a single track railway line. At any time, at most one train is allowed to run on such a line and so entering and exiting this part is governed by special signals. Note that designing proper semantics for such signals is not as easy at it seems because the signals at both ends must be synchronized. For example, it must be ensured that of two trains concurrently approaching the signals from opposite ends only one is allowed to pass. Also, leaving the critical section on one end must allow another follow-up train waiting at the opposite end to enter the track, too.

Inspired by real-world semaphores, Edsger W. Dijkstra introduced semaphores as a synchronization abstraction in his “THE” operating system in 1968 [Dij68]. The name “THE” stands for “Technische Hogeschool Eindhoven” (Eindhoven University of Technol-
ogy) where Dijkstra was a professor at that time. Ever since, Dijkstra evolved into one of the most prominent and fascinating figures in computer science. Not only did he influence many of today’s programming languages through his work on the language ALGOL (Algorithmic Language), but he also invented a lot of clever algorithms (like the famous shortest path algorithm for graphs).

11.3.1 Semantics of Semaphores

A semaphore is an operating system abstraction offering two primitive operations called P and V. The operation P (which can be read as “pass”, originally: dutch prolaag, probeer te verlagen; try to reduce) [Wik, Dij] is invoked by a thread when it wishes to enter its critical section. Conversely, the operation V (which can be read as “leaVe”, originally: dutch verhogen, increase) is invoked when a process leaves its critical section.

A semaphore guarantees \textit{k-mutual exclusion}. The formal statement of this concept can be defined as follows.

\textbf{Definition 2 (k-mutual exclusion)} If all threads properly encapsulate their critical section with \textit{P} and \textit{V}, then the semaphore guarantees that at most \textit{k} threads are in their critical sections at the same time.

The concept of \textit{k}-mutual exclusion is a generalization of simple \textit{mutual exclusion} for which \textit{k} = 1. The actual value of \textit{k} must be passed to the semaphore upon initialization.

It is possible to break down \textit{k}-mutual exclusion to specific semantics of the individual semaphore operations \textit{P} and \textit{V} as follows:

\textbf{Definition 3 (semantics of \textit{P} and \textit{V})} Assume semaphore \textit{S} is initialized with \textit{k}. Then the operations \textit{P} and \textit{V} on \textit{S}, written \textit{P(S)} and \textit{V(S)}, have the following meaning:

- \textit{P(S)} blocks in case exactly \textit{k} threads have passed \textit{P(S)} without passing \textit{V(S)}.
- \textit{V(S)} deblocks a thread which is blocked at a \textit{P(S)} in case such a thread exists.

For \textit{k} = 1, the operations \textit{P} and \textit{V} therefore clearly resemble the semantics of a mechanism necessary to protect a single track railway line. More generally, they resemble the semantics for protection signals of a \textit{k}-track railway line segment.

11.3.2 Single Mutual Exclusion

The notion of \textit{k}-mutual exclusion for \textit{k} = 1 is often simply called \textit{mutual exclusion} or \textit{mutex} for short. The name “mutex” is also used for the semaphore that protects a simple critical section where at most one thread is allowed to enter. Such a critical section can be implemented easily with semaphores as follows.

\texttt{(example: classical mutual exclusion with semaphores 359)≡}

\texttt{Semaphore Mutex = 1; // initialization}
\texttt{// code of the thread}
\texttt{P (Mutex); // enter critical section}
11 Synchronization

// critical section
V (Mutex); // leave critical section
// continued code of the thread

Incidentally, this is exactly what we would need if we want to implement ENTER_MUTEX and EXIT_MUTEX without busy waiting at the operating system level. The question of course is: Can we implement semaphores without busy waiting?

11.3.3 Initialization of Semaphores

We now look at a simple implementation of semaphores at the operating system level. These semaphores are consequently called kernel level semaphores. Semaphores at that level basically encapsulate a counter and a list of threads. This list can be thought of as being at the same level as the ready queue in the dispatcher. In fact, it implements one type of blocked queue in the system (see Section 6.2.1.3).

We first declare the semaphore type, a structure consisting of a counter and a queue. All declarations and functions on this type of semaphore are prefixed with kl_ to identify them as kernel level semaphores and clearly separate them from user level semaphores. We allow for additional “implementation” fields at the end of the semaphore structure which are not of interest for the general idea of semaphores.

\[
\text{typedef struct} \{
\text{int counter;}
\text{blocked_queue bq;}
\} \text{kl_semaphore;}
\]

Defines:
kl_semaphore, used in chunks 361–63 and 391a.
Uses blocked_queue 183a.

The structure kl_semaphore_{360a} is the internal representation of semaphores. In later code we will refer to semaphores by a unique identifier instead of a pointer to such a structure. Basically, this identifier will serve as a pointer into a global semaphore table implemented later.

\[
\text{typedef int kl_semaphore_id;}
\]

Defines:
kl_semaphore_id, used in chunks 361c, 362, 364, and 391a.

The function get_new_semaphore_{364b}(k) returns the identifier of a new semaphore initialized with k. The return value \(-1\) is used as an error code meaning that something went wrong during allocation. This usually means that the internal table is full, which is a bad sign.

We also provide a function to release a semaphore. Releasing it implies that all threads which may be blocked on that semaphore are deblocked.

Here are the prototypes:
11.3 Semaphores

11.3.4 Implementing P and V

The idea of the implementation of P and V is as follows. The counter of the semaphore represents the remaining “potential” of the semaphore, i.e., the number of threads which are still allowed to pass without being blocked. To actually block and deblock threads, we can simply use the operations provided by the kernel level dispatcher (see Section 6.2.1.1 for the general description and Section 6.2.2.3 for the implementation).

We will name the functions wait_semaphore (for P) and signal_semaphore (for V):

```c
void wait_semaphore(kl_semaphore_id sid);
void signal_semaphore(kl_semaphore_id sid);
```

The idea of the operation P is to check the remaining potential of the semaphore and block in case the potential is used up. To understand the condition under which a thread is blocked (counter < 0), remember that a semaphore initialized with 1 allows one thread to pass. Since the counter is decremented before the check, a condition counter < 1 would not be correct.

Note that we mark the body of the implementations of P and V as critical sections. To see why, you should explore the case of interrupts happening after the counter manipulation and the test of the counter value (or solve exercise 32). Since semaphores are synchronization techniques on a higher level of abstraction than hardware mechanisms, it would also be counter-intuitive if we could implement them without referring to any lower-level synchronization primitive.

```c
void wait_semaphore(kl_semaphore_id sid) {
    kl_semaphore sem = ⟨semaphore structure with identifier sid⟩;
    ⟨begin critical section in kernel⟩
    sem.counter--;
    if (sem.counter < 0) {
        block (&sem.bq, TSTATE_LOCKED);
        ⟨resign⟩
    }
    ⟨end critical section in kernel⟩
}
```

Defines:
- wait_semaphore, used in chunks 361b and 391a.
Uses kl_semaphore 360a, kl_semaphore_id 360b, and TSTATE_LOCKED 180a.

The operation V is slightly simpler than P because there is no danger of a context switch. The function just checks whether a thread is still blocked on the semaphore queue and deblocks this thread if there is one.
11 Synchronization

```c
void signal_semaphore (kl_semaphore_id sid) {
    kl_semaphore sem = {semaphore structure with identifier sid};
    begin critical section in kernel
    sem.counter++;
    if (sem.counter < 1) {
        blocked_queue *bq = &sem.bq;
        thread_id head = bq->next;
        if (head != 0) {
            deblock (head, bq);
        }
    }
    end critical section in kernel
}
```

Defines:
- `signal_semaphore`, used in chunk 391a.
Uses `blocked_queue` 183a, `deblock` 186b, `kl_semaphore` 360a, `kl_semaphore_id` 360b, and `thread_id` 178a.

The above implementation of semaphores is probably the simplest one but still leaves some room for variation. For example, the implementation of the semaphore queue can be performed in different ways, e.g. as a simple FIFO queue, but priority queues can be used, too. The FIFO processing order is likely the one which is implicitly assumed most often when using semaphores, because it guarantees that the thread which has waited longest is deblocked first.

11.3.5 User-Level Semaphores

Until now, we have discussed the implementation of kernel-level semaphores, i.e., semaphores that have the power to block and deblock kernel-level threads. If you are implementing a user-mode thread library to realize user-level threads, you also may need a synchronization abstraction such as semaphores. You could use kernel-level semaphores for this purpose, but it is also possible to realize synchronization primitives that are tailored to handle user-level threads: user-level semaphores.

To recapitulate the difference between kernel-level threads and user-level threads, have a look again at Figure 7.3 on page 252. Kernel-level threads are virtual processors for user programs, and it is even possible to map a user program to multiple kernel-level threads (resulting in a virtual multiprocessor). But the power of such a virtual multiprocessor can only be unfolded if the user mode program supports a multiprogramming abstraction in user mode, such as a user-level thread library.

User-level semaphores are semaphores that block and deblock user-level threads. Their design and implementation is equivalent to those at kernel level, however, they are implemented in user mode, i.e., one step up the abstraction hierarchy. User-level semaphores have counters, blocked queues etc. in a similar way as kernel-level semaphores. However, these counters are simple integers in user space and the queues are the queues manipulated in user space by the thread library. To build user-level semaphores, you can therefore simply take the implementation of kernel-level semaphores and copy them into your user
program, at least in large parts. The only notable difference is the implementation of critical sections. This will be discussed later when we discuss the general notion of kernel synchronization.

### 11.3.6 Semaphores in Ulix

The main effort to implement kernel level semaphores has already been done above. Here we fill in the final gaps.

Although semaphores (like threads and virtual address spaces) can be allocated and freed, we want to implement them without dynamic memory. Thus semaphores are held in a large semaphore table called `kl_semaphore_table`. It is an array of `kl_semaphore` structures.

```c
const int MAX_SEMAPHORES = 1024;
#define MAX_SEMAPHORES 1024

kl_semaphore_table[MAX_SEMAPHORES];
```

Defines:
- `MAX_SEMAPHORES`, used in chunks 363 and 364b.

There’s a maximum number of semaphores that can be allocated in the kernel. Since both used and unused semaphores are held in a table, we need additional information to distinguish both. So each semaphore has a counter and a queue, but it also has an additional field storing the semaphore state. The value `false (0)` means that the semaphore entry is free.

```c
boolean used;
```

Now it’s also clear how we can initialize the fields.

```c
for (int i = 0; i < MAX_SEMAPHORES; i++) {
    kl_semaphore_table[i].counter = 0;
    initialize_blocked_queue (&kl_semaphore_table[i].bq);
    kl_semaphore_table[i].used = false;
}
```

Uses `initialize_blocked_queue 183c`, `kl_semaphore_table 363b`, and `MAX_SEMAPHORES 363a`.

Since we didn’t mention the semaphore table earlier, we need to fill in the mapping between the semaphore identifier `sid` and the semaphore structure in the table.

```c
kl_semaphore_table[sid]
```

Uses `kl_semaphore_table 363b`. 
Finally, we have to implement the two functions get_new_semaphore\textsubscript{364b} for acquiring and release_semaphore\textsubscript{364c} for releasing semaphores. Allocation is done in a round robin fashion (like in the FIFO allocation scheme for pages). We use a counter next\_kl\_semaphore\textsubscript{364a} to point to the next semaphore entry in the table which can be allocated.

\begin{equation}
\begin{aligned}
\text{kl\_semaphore\_id} &\text{ next\_kl\_semaphore} = 0; \\
\text{defines: next\_kl\_semaphore, used in chunk 364b.}
\end{aligned}
\end{equation}

To allocate a new semaphore we check the next table entry and use it if it is free. While looking for a free entry in the table we use a check counter to catch the case where the semaphore table is full.

\begin{equation}
\begin{aligned}
\text{kl\_semaphore\_id} &\text{ get\_new\_semaphore (int k) } \\
\text{ int } &\text{ check} = \text{ MAX\_SEMAPHORES; } \\
\text{ while } &\text{ (kl\_semaphore\_table[next\_kl\_semaphore].used == true) } \\
\text{ next\_kl\_semaphore} &\text{ = (next\_kl\_semaphore + 1) } \% \text{ MAX\_SEMAPHORES; } \\
\text{ check} &\text{--; } \\
\text{ if (check} &\text{ ≤ 0) return -1; }
\end{aligned}
\end{equation}

\begin{equation}
\begin{aligned}
\text{kl\_semaphore\_table[next\_kl\_semaphore].used} &\text{ = true; } \\
\text{kl\_semaphore\_table[next\_kl\_semaphore].counter} &\text{ = k; } \\
\text{initialize\_blocked\_queue } &\text{(}\&\text{kl\_semaphore\_table[next\_kl\_semaphore].bq); } \\
\text{return next\_kl\_semaphore; }
\end{aligned}
\end{equation}

Uses initialize\_blocked\_queue 183c, kl\_semaphore\_id 360b, kl\_semaphore\_table 363b, MAX\_SEMAPHORES 363a, and next\_kl\_semaphore 364a.

Releasing a semaphore is a little tricky. Just resetting the state field in the semaphore table is not enough since threads may be blocked in the semaphore queue. These threads must be released to the ready queue.

\begin{equation}
\begin{aligned}
\text{void } &\text{ release\_semaphore (kl\_semaphore\_id s) } \\
\text{ kl\_semaphore\_table[s].used} &\text{ = false; } \\
\text{ while } &\text{ (front\_of\_blocked\_queue (kl\_semaphore\_table[s].bq) ! = 0) } \\
\text{ thread\_id } &\text{ t = front\_of\_blocked\_queue (kl\_semaphore\_table[s].bq); } \\
\text{ remove\_from\_blocked\_queue } &\text{ (t, } \&\text{kl\_semaphore\_table[s].bq); } \\
\text{ add\_to\_ready\_queue } &\text{ (t); }
\end{aligned}
\end{equation}

Defines: release\_semaphore, used in chunk 361a.

Uses add\_to\_ready\_queue 184b, front\_of\_blocked\_queue 185b, kl\_semaphore\_id 360b, kl\_semaphore\_table 363b, remove\_from\_blocked\_queue 186a, and thread\_id 178a.
11.4 **ULIX Locks**

Locks could be treated as a special case of semaphores (which are initialized to 1), but we have decided to provide a separate implementation for kernel locks and a user mode interface.

For locks, we use the following type which resembles the definition of the semaphore type `kl_semaphore`:

```c
typedef struct {
    short int l;       // the lock
    boolean used;      // are we using this lock?
    blocked_queue bq;  // queue for this lock
    char lockname[20]; // name
} lock_t;
```

\[type definitions\]

Defines:
- `lock`, used in chunks 306b, 308c, 310f, 366–69, 371a, 373c, 509a, 516b, 530a, 547b, and 552c.
- `lock_t`, used in chunk 365c.

Uses `blocked_queue` 183a.

As we make a lot of use of locks in the kernel, we provide the `lockname` field so that we can generate more helpful debugging output.

We allow up to 1024 locks

```c
#define MAX_LOCKS 1024
```

\[constants\]

Defines:
- `MAX_LOCKS`, used in chunks 365c, 367b, and 606.

and reserve a table for them—as with the semaphores, we want to avoid dynamic allocation of memory in the kernel.

```c
lock_t kernel_locks[MAX_LOCKS];
```

\[global variables\]

Defines:
- `kernel_locks`, used in chunks 367b, 368, and 606.

Uses `lock_t` 365a and `MAX_LOCKS` 365b.

We reserve the first lock (`kernel_locks[0]`) for locking the lock table itself.

11.4.1 **Locking, Unlocking and Just Wishing**

We provide three functions that can be used by user mode processes or threads (via corresponding library functions that we introduce in the next section):

```c
void mutex_lock (lock lockvar);
boolean mutex_try_lock (lock lockvar);
void mutex_unlock (lock lockvar);
```
mutex_lock and mutex_unlock have the expected behavior: The first function tries to acquire the lock, and if that fails, it will block the active process and place it on the queue that belongs to this lock. The unlocking function returns the lock and wakes the first waiting process. In addition to these, mutex_try_lock does what its name implies: It tries to acquire the lock, but does not block if that goes wrong. It always returns, and its return value indicates whether the lock was acquired or not. If it failed, it returns false. So programs can use this to attempt to get a lock, but they have to check the return value and must not enter the critical section, if false was returned.

Regarding the placement of (begin critical section in kernel 380a) see exercises 34 and 35.

366a) \( \text{function implementations 100b)} \) \( \equiv \) \( \text{44a) <364c 366b>} \)
void mutex_lock (lock lockvar) {
  if (current_task == 0) { return; } // no process
  \( \langle \text{begin critical section in kernel 380a} \rangle \)
  while (lockvar->l == 1) {
    block (&(lockvar->bq), TSTATE_LOCKED); // put process to sleep
    \( \langle \text{resign 221d} \rangle \)
  }
  lockvar->l = 1;
  \( \langle \text{end critical section in kernel 380b} \rangle \)
}
Defines:
- mutex_lock, used in chunks 308c, 310a, 367b, 368, 371a, 509d, 510b, 512b, 516d, 517c, 520c, 530, and 549d.
- Uses current_task 192c, lock 365a, and TSTATE_LOCKED 180a.

366b) \( \text{function implementations 100b)} \) \( \equiv \) \( \text{44a) <366a 366c>} \)
boolean mutex_try_lock (lock lockvar) {
  \( \langle \text{begin critical section in kernel 380a} \rangle \)
  int tmp = lockvar->l; lockvar->l = 1;
  \( \langle \text{end critical section in kernel 380b} \rangle \)
  return (tmp == 0);
}
Defines:
- mutex_try_lock, used in chunks 307, 308, and 371a.
- Uses lock 365a.

For unlocking, we reset lockvar->l to 0 and then check whether we can wake up a waiting thread:

366c) \( \text{function implementations 100b)} \) \( \equiv \) \( \text{44a) <366b 367b>} \)
void mutex_unlock (lock lockvar) {
  if (current_task == 0) { return; } // no process
  if (lockvar->l == 0) {
    debug_printf ("NOTICE: unlocking unlocked LOCK: %s\n", lockvar->lockname);
  }
  \( \langle \text{begin critical section in kernel 380a} \rangle \)
  lockvar->l = 0;
  \( \langle \text{end critical section in kernel 380b} \rangle \)
  // wake a process
  blocked_queue *bq = &(lockvar->bq);
  thread_id head = bq->next;
if (head != 0) { // If one thread is waiting, deblock and resign
deblock (head, bq);
}
@end critical section in kernel 380b
}

Defines:
mutex_unlock, used in chunks 307, 308, 311a, 365d, 367b, 368, 371a, 509d, 510b, 512b, 516d, 517c, 520c, 530, and 549e.
Uses blocked_queue 183a, current_task 192c, deblock 186b, debug_printf 601d, lock 365a, and thread_id 178a.

If you compare the mutex_lock and mutex_unlock functions with wait_semaphore and signal_semaphore you will notice a big similarity. The main difference is that semaphores are more general, thus allowing $k$-mutual exclusion, whereas locks can only be used for single-mutual exclusion. However, in none of the ULIx code we came across a situation where a semaphore with $k > 1$ would have been useful. So now, if we want to achieve mutual exclusion between kernel level threads, we can simply acquire a kernel lock. This strategy is more elegant than using hardware mechanisms directly and also more efficient on multi-processor systems where we can avoid effort spent spinning in spin locks.

11.4.1.1 Lock Administration

Finally, we need to provide functions that allow to create a new lock and release it when it is no longer needed. They have these prototypes:

\[ \text{lock get_new_lock (char *name);} \]
\[ \text{void release_lock (lock l);} \]

The get_new_lock function has an argument via which we can give the lock a name. If you enable the kernel mode shell, you can type locks to see a list of all the locks, with their names and the threads on their blocked queues.

\[ \text{lock get_new_lock (char *name) {} \}
\]
\[ \text{mutex_lock (kernel_locks); // lock the list of kernel locks, we use kernel_locks[0] for (int i = 1; i < MAX_LOCKS; i++) { \}
\]
\[ \text{if (!kernel_locks[i].used) { \}
\]
\[ \text{kernel_locks[i].used = true; \}
\]
\[ \text{initialize_blocked_queue (&kernel_locks[i].bq); // initialize blocked queue \}
\]
\[ \text{strncpy (kernel_locks[i].lockname, name, 20); \}
\]
\[ \text{mutex_unlock (kernel_locks); // unlock access to list \}
\]
\[ \text{return &kernel_locks[i]; \}
\]
\[ \text{mutex_unlock (kernel_locks); \}
\]
\[ \text{return NULL; \}
\]
11 Synchronization

For freeing a lock we set the used entry to false and unlock all threads on the blocked list (if there are any):

```c
void release_lock (lock l) {
    mutex_lock (kernel_locks); // lock the list of kernel locks
    l->used = false;
    blocked_queue *bq = &(l->bq);
    thread_id head = bq->next;
    while (head != 0) {
        thread_id next = thread_table[head].next;
        deblock (head, bq);
        head = next;
    }
    mutex_unlock (kernel_locks);
}
```

11.5 Pthread Mutexes for Threads

In order to provide user space programs with mutexes, it is not necessary to interface the kernel—the code that we used for the implementation of kernel locks would also work in user mode since it requires no privileged instructions. However, we want to queue threads which try to acquire a mutex, and that is a task for the kernel. So instead of duplicating parts of the already existing locking code, we provide a user mode interface to the kernel functions get_new_lock, mutex_lock, mutex_unlock, and release_lock.

If you search for POSIX mutex functions on a Linux machine, you will find several functions, including the following ones:

- `pthread_mutex_init(3)` - create a mutex
- `pthread_mutex_lock(3)` - lock a mutex
- `pthread_mutex_trylock(3)` - attempt to lock a mutex without blocking
- `pthread_mutex_unlock(3)` - unlock a mutex
- `pthread_mutex_destroy(3)` - free resources allocated for a mutex

Our implementation only supports the essential features, so for example, you cannot use mutex attributes. We only include the corresponding argument in the function call so that the functions have a similar look and feel as the regular POSIX functions. You've already seen the same comment when you looked at the implementation of POSIX threads.
Also note that our kernel locks are valid globally and can be used across process borders. That means that in a Ulx program a process can create a mutex and then fork; afterwards both processes can be synchronized via the mutex. POSIX mutexes forbid this.

### 11.5.1 Creating a New Mutex

Before we start, we define two types that simplify attempts to port programs to Ulx:

```c
typedef int pthread_mutex_t;
typedef int pthread_mutexattr_t;
```

Defines:

- `pthread_mutex_t`, used in chunks 369–73.
- `pthread_mutexattr_t`, used in chunks 369c, 370d, and 373.

For mutex creation we implement the function

```c
int u_pthread_mutex_init (pthread_mutex_t *restrict mutex,
                        const pthread_mutexattr_t *restrict attr);
```

that reserves a fresh kernel lock via `get_new_lock` and returns the memory address of the lock data structure. This serves as a unique identifier for the lock when used in user space. (Since that address is in the kernel’s private memory range, it will also be valid across process borders; see our earlier comment about cross-process use of mutexes.)

Since kernel locks have names, the function generates a name that consists of the string "lock, pid=" and the thread ID. Note that this is not unique if the same thread creates several mutexes.

```c
int u_pthread_mutex_init (pthread_mutex_t *restrict mutex,
                        const pthread_mutexattr_t *restrict attr) {
    char s[20];
    sprintf ((char*)s, "lock, pid=%d", thread_table[current_task].pid);
    lock tmp = get_new_lock (s);
    if (tmp != NULL) {
        *mutex = (pthread_mutex_t)tmp;
        return 0; // success
    } else {
        thread_table[current_task].error = EAGAIN;
        return -1; // error
    }
}
```

Defines:

- `u_pthread_mutex_init`, used in chunks 369b and 370d.
Uses `current_task` 192c, `EAGAIN` 370a, `get_new_lock` 367b, `lock` 365a, `NULL` 46a, `pthread_mutex_t` 369a, `pthread_mutexattr_t` 369d, `sprintf` 601a, and `thread_table` 176b.
If `get_new_lock` was unsuccessful, we set the error field in the TCB to `EAGAIN` (it can be queried via the `errno` macro in the process).

We add a system call:

```c
#define EAGAIN 35

#define __NR_pthread_mutex_init 517

void syscall_pthread_mutex_init (context_t *r) {
    // ebx: mutex id
    // ecx: attributes, not implemented
    eax_return ( u_pthread_mutex_init ( pthread_mutex_t*r->ebx,
                                            pthread_mutexattr_t*r->ecx ) );
}
```

We define the three locking and unlocking functions

```c
int u_pthread_mutex_lock (pthread_mutex_t *mutex);
int u_pthread_mutex_trylock (pthread_mutex_t *mutex);
int u_pthread_mutex_unlock (pthread_mutex_t *mutex);
```

which “convert” POSIX-compliant mutexes into kernel mutexes and call the `mutex_lock`, `mutex_try_lock`, and `mutex_unlock` functions. If `u_pthread_mutex_trylock` cannot acquire the mutex via `mutex_try_lock`, it will set the error field of the TCB to `EBUSY` (as expected by the POSIX standard) and return −1. The other two functions cannot fail, they simply return.
11.5 Pthread Mutexes for Threads

(function implementations 100b)+≡

int u_pthread_mutex_lock (pthread_mutex_t *mutex) {
    lock l = (lock)*mutex;
    mutex_lock (l);
    return 0;
}

int u_pthread_mutex_trylock (pthread_mutex_t *mutex) {
    lock l = (lock)*mutex;
    if (mutex_try_lock (l))
        return 0; // success
    else {
        thread_table[current_task].error = EBUSY;
        return -1; // error
    }
}

int u_pthread_mutex_unlock (pthread_mutex_t *mutex) {
    lock l = (lock)*mutex;
    mutex_unlock (l);
    return 0;
}

Defines:
  u_pthread_mutex_lock, used in chunk 372a.
  u_pthread_mutex_trylock, used in chunk 372a.
  u_pthread_mutex_unlock, used in chunks 370f and 372a.
Uses current_task 192c, EBUSY 371b, lock 365a, mutex_lock 366a, mutex_try_lock 366b, mutex_unlock 366c,
  pthread_mutex_t 369a, and thread_table 176b.

(error constants 370a)+≡

#define EBUSY 16 // device / resource busy

Defines:
  EBUSY, used in chunk 371a.
Uses busy.

Again, we add system calls for the new functions:

(ulix system calls 206e)+≡

#define __NR_pthread_mutex_lock 518
#define __NR_pthread_mutex_unlock 519
#define __NR_pthread_mutex_trylock 526

Defines:
  __NR_pthread_mutex_lock, used in chunks 372b and 373e.
  __NR_pthread_mutex_trylock, used in chunk 372b.
  __NR_pthread_mutex_unlock, used in chunks 372b and 373e.

(syscall prototypes 173b)+≡

void syscall_pthread_mutex_lock (context_t *r);
void syscall_pthread_mutex_trylock (context_t *r);
void syscall_pthread_mutex_unlock (context_t *r);
They simply evaluate the mutex ID (found in the EBX register) and return the result by setting the EAX register:

```c
void syscall_pthread_mutex_lock (context_t *r) {
    // ebx: mutex id
    eax_return ( u_pthread_mutex_lock ((pthread_mutex_t*)r->ebx) );
}

void syscall_pthread_mutex_trylock (context_t *r) {
    // ebx: mutex id
    eax_return ( u_pthread_mutex_trylock ((pthread_mutex_t*)r->ebx) );
}

void syscall_pthread_mutex_unlock (context_t *r) {
    // ebx: mutex id
    eax_return ( u_pthread_mutex_unlock ((pthread_mutex_t*)r->ebx) );
}
```

Defines:  
syscall_pthread_mutex_lock, used in chunk 372b.  
syscall_pthread_mutex_unlock, used in chunks 371d and 372b.  
Uses context_t 142a, eax_return 174a, pthread_mutex_t 369a, syscall_pthread_mutex_trylock, u_pthread_mutex_lock 371a, u_pthread_mutex_trylock 371a, and u_pthread_mutex_unlock 371a.

### 11.5.3 Destroying a Mutex

Finally, we need to be able to destroy a mutex when it is no longer needed.

```c
void syscall_pthread_mutex_destroy (context_t *r);
```

Defines:  
syscall_pthread_mutex_destroy, used in chunk 372c.  
Uses context_t 142a, eax_return 174a, pthread_mutex_t 369a, and u_pthread_mutex_destroy 373c.

```c
#define __NR_pthread_mutex_destroy 520
```

Defines:  
__NR_pthread_mutex_destroy, used in chunk 373.
11.5 Pthread Mutexes for Threads

\[\begin{align*}
\text{initialize syscalls} &\equiv (44b) <372b 416c> [373a] \\
\text{install syscall handler} &\equiv (\_NR\_pthread_mutex\_destroy, syscall\_pthread_mutex\_destroy); \\
\text{Uses }\_NR\_pthread_mutex\_destroy 372e, \text{ install syscall handler 201b, and syscall\_pthread_mutex\_destroy 372d.}
\end{align*}\]

Here is the implementation of the kernel function

\[\begin{align*}
\text{function prototypes} &\equiv (44a) <370f 405d> [373b] \\
\text{int } u\_pthread\_mutex\_destroy &\equiv (\text{pthread\_mutex\_t }\ast\text{mutex}); \\
\text{function implementations} &\equiv (44a) <371a 406> [373c] \\
\text{int } u\_pthread\_mutex\_destroy &\equiv (\text{pthread\_mutex\_t }\ast\text{mutex}) \\
&\quad \text{lock } l = (\text{lock})*\text{mutex}; \\
&\quad \text{release\_lock}(l); \\
&\quad \text{return } 0;
\end{align*}\]

Defines: 
\(u\_pthread\_mutex\_destroy\), used in chunks 372d and 373b.

\(u\_pthread\_mutex\_destroy\) uses lock 365a, pthread\_mutex\_t 369a, and release\_lock 368.

### 11.5.4 The Library Functions

As usual we provide a set of library functions that let user mode processes make these system calls:

\[\begin{align*}
\text{ulixlib function prototypes} &\equiv (48a) <333d 429a> [373d] \\
\int &\text{pthread\_mutex\_init } (\text{pthread\_mutex\_t }\ast\text{mutex}, \\
&\quad \text{const pthread\_mutex\_attr\_t }\ast\text{attr}); \\
\int &\text{pthread\_mutex\_lock } (\text{pthread\_mutex\_t }\ast\text{mutex}); \\
\int &\text{pthread\_mutex\_unlock } (\text{pthread\_mutex\_t }\ast\text{mutex}); \\
\int &\text{pthread\_mutex\_destroy } (\text{pthread\_mutex\_t }\ast\text{mutex});
\end{align*}\]

\[\begin{align*}
\text{ulixlib function implementations} &\equiv (48b) <333e 429b> [373e] \\
\int \text{pthread\_mutex\_init } &\equiv (\text{pthread\_mutex\_t }\ast\text{mutex}, \\
&\quad \text{const pthread\_mutex\_attr\_t }\ast\text{attr}) \\
&\quad \text{return syscall3 (\_NR\_pthread\_mutex\_init, (unsigned int)mutex, (unsigned int)attr);}}
\end{align*}\]

\[\begin{align*}
\int \text{pthread\_mutex\_lock } &\equiv (\text{pthread\_mutex\_t }\ast\text{mutex}) \\
&\quad \text{return syscall2 (\_NR\_pthread\_mutex\_lock, (int)mutex);}}
\end{align*}\]

\[\begin{align*}
\int \text{pthread\_mutex\_unlock } &\equiv (\text{pthread\_mutex\_t }\ast\text{mutex}) \\
&\quad \text{return syscall2 (\_NR\_pthread\_mutex\_unlock, (int)mutex);}}
\end{align*}\]

\[\begin{align*}
\int \text{pthread\_mutex\_destroy } &\equiv (\text{pthread\_mutex\_t }\ast\text{mutex}) \\
&\quad \text{return syscall2 (\_NR\_pthread\_mutex\_destroy, (int)mutex);}}
\end{align*}\]
11.5.5 Testing

In order to test the synchronization of threads, you can run the /bin/thread program. It starts three threads, two of which add to or subtract from a shared variable. After 250 additions and 250 subtractions, the variable should have the initial value. The program accepts a parameter: If you start it as thread 0 it will work without synchronization (and return random results due to the two threads concurrently entering their critical sections). If, however, you start it as thread 1 it will use a pthread_mutex_t to protect those sections and consistently return the correct result.

11.6 Kernel Synchronization

Until now, we have dealt with a couple of synchronization mechanisms that are suitable for different levels of abstraction:

1. Hardware-based mechanisms, such as interrupt masking and spin locks, that can establish mutual exclusion on a monoprocessor or multiprocessor system.
2. Kernel-level semaphores that can be used to block and deblock kernel level threads. Kernel-level locks are a special instance of such semaphores.
3. User-level semaphores that can be used to block and deblock user-level threads.

So far, our discussion of synchronization has focused on threads and with semaphores and locks we have provided nice abstractions to synchronize them. Whenever we need to prevent a thread from accessing some resource we simply block it; later when the resource becomes available again, we unblock such a thread so that it can continue execution. This works for both threads in user mode and in kernel mode.

However, there is other code in the kernel which is not executed on behalf of some particular thread: Interrupt handlers are activated whenever an interrupt occurs, and while such a handler function runs in the context of the thread which was active when the interrupt was signaled, it is not related to that thread in any manner. Getting this right is called kernel synchronization.

Kernel synchronization is a messy business because machine operations run in kernel mode and interrupts and context switches make code and execution sequences unintuitive. At its core, kernel synchronization deals with correctly implementing critical sections at the lowest level (the kernel). This section deals with kernel synchronization and discusses how this issue is solved in ULIx.
11.6 Kernel Synchronization

11.6.1 Overview

In general, synchronization may be needed on different levels of abstraction within an operating system. Figure 11.5 (a modified version of Figure 7.3 on page 252) shows those levels; the dotted black lines represent situations where shared data may be used, thus requiring synchronization. Differing from Figure 7.3, it also includes an interrupt handler which always runs in kernel mode.

We now discuss the different synchronization issues in context from top (user-level threads) to bottom (CPUs).

11.6.1.1 Synchronizing User-Level Threads

First we consider the following case: Two user-level threads which are mapped to one or multiple kernel-level threads may use the same data and need to perform mutual exclusion. In Figure 11.5, for example, Process 1 runs on one kernel-level thread (virtual monoprocessor) and Process 2 runs on two kernel-level threads (virtual multiprocessor). In both cases, the user-level thread library has to take care of synchronization since the kernel does not recognize the two threads as separate entities.

Figure 11.5: Synchronization in user and kernel mode. Arrow colors (on one level) express the order of allocation of a virtual (top) or physical (bottom) processor; black dotted arrows show where synchronization can be supported between user-level threads (1), between kernel-level threads (2), between kernel-level threads and interrupt handlers (3), and between two CPUs (4).
The mechanism of choice to synchronize user-level threads are user-level semaphores. They can be used to block and deblock user-level threads for synchronization. And if the thread library runs on a single kernel-level thread and the library does not support a signal mechanism (a “user-level interrupt” mechanism, see Chapter 14), only one concurrent activity will update the critical thread library data structures at any time. So while there might be critical sections in the thread (library) code, the entry and exit protocols may be empty.

However, if user-level threads can be interrupted (using signals for example) or if user-level threads run on a virtual multiprocessor (multiple kernel-level threads), we have to ensure mutual exclusion of concurrent activities in the threads library again. For the case of “interruptible” user-level threads (via signals), we need to ensure that any invocation of a signal handler does not violate mutual exclusion. A valid method would be to “mask” signals (i.e., ensure that during certain times user-defined signal handlers are not executed). This situation is analogous to interrupt-masking at lower levels.

For the case of a virtual multiprocessor (multiple kernel-level threads), signal masking is not enough since signals target kernel-level threads and so if one kernel-level thread masks signals another can still receive them and invoke a signal handler, thus violating mutual exclusion. So how could mutual exclusion be achieved here?

A naive approach would be to use a spin lock in addition to signal masking. Since the necessary machine instructions like Test-and-Set are not privileged, the user-level thread library can use them to achieve “global” mutual exclusion. However, spin locks imply busy waiting, so the question is: Can we do better?

Fortunately, the answer is yes: We can use a kernel-level semaphore, or more precisely a kernel lock as follows. During initialization of the thread library we allocate a kernel lock. Whenever a user-level thread wishes to run exclusively, if will lock the mutex and proceed. When it finishes its critical section, it releases the lock. To see why this works, consider again Figure 11.5 and look at user-level thread 3 and user-level thread 4 in the top right. Both threads might be running on different virtual processors (kernel-level thread 2 and kernel-level thread 3, the green assignment). Now assume that user-level thread 3 enters a critical section and acquires the lock. If user-level thread 4 attempts to enter its own critical section, it will try to acquire the same lock, but because it is taken, then kernel-level thread 3 is blocked. It is automatically deblocked when user-level thread 3 releases the lock. Busy waiting is completely avoided!

Remember however, that the above solution still needs to take care of user-level “interrupts” (signals). If signal handlers can be run at any time, also note that kernel locks are not useful for mutual exclusion if the thread library is running on a virtual monoprocessor. If one of the user-level threads acquires the lock, is interrupted by a signal handler and switches to another thread that also tries to acquire the lock, the kernel-level thread (i.e., the entire user-level thread library) would block and the situation could never be resolved.

Ulxix (like most other systems) does not implement the mapping of several user-level threads to one kernel-level thread, thus we need not consider this case—the user-mode functions \texttt{pthread_mutex \_lock} and \texttt{pthread_mutex \_unlock} use system calls to call the kernel functions \texttt{u\_pthread_mutex \_lock} and \texttt{u\_pthread_mutex \_unlock}.
11.6.1.2 Synchronizing Kernel-Level Threads

Now we consider the case where two kernel-level threads wish to synchronize because they use the same data. This is case (2) in Figure 11.5. The method of choice here is obviously to use kernel-level semaphores, or more specifically (for mutual exclusion) kernel locks. If such locks can be acquired and released atomically, kernel locks can be used to achieve mutual exclusion between kernel-level threads, just as discussed previously for the virtual multiprocessor that runs a user-level thread library.

The property we need to ensure, however, is in fact the atomicity of lock acquisition. Remember that kernel-level threads may be interrupted at any time and a context switch might schedule another kernel-level thread. So the problem of achieving mutual exclusion between kernel-level threads can be reduced to the problem of ensuring the atomicity of the locking procedure. Unfortunately, we cannot use kernel mutexes for this since this is the mechanism we are trying to implement.

We have handled exactly this case in the previous sections where we have discussed the implementation of \texttt{u_pthread_mutex_lock} and \texttt{u_pthread_mutex_unlock}. The trick to solve this problem was to declare the main parts of \texttt{mutex_lock}, \texttt{mutex_try_lock} and \texttt{mutex_unlock} as critical sections in the kernel, but we have not yet discussed how to actually implement this. However, we have all necessary ingredients ready to deal with it now. This is what kernel synchronization is all about.

Many operating systems also support synchronization across processes which is also possible in \texttt{ULiX} if two processes share a \texttt{pthread_mutex_t} variable; note however that the POSIX standard forbids this (see Section 11.5). Instead Unix systems use \textit{named semaphores} for cross-process synchronization which are not implemented in \texttt{ULiX}.

To summarize synchronization issues up to now—and as partially shown in Figure 11.6,

- in user mode, threads can call the user-mode library functions \texttt{pthread_mutex_lock} and \texttt{pthread_mutex_unlock} to acquire kernel level mutexes that can be used for inter-thread synchronization,
- in kernel mode, threads can also use the kernel functions \texttt{u_pthread_mutex_lock} and \texttt{u_pthread_mutex_unlock} for the same purposes,
- and these are (again) wrappers for the general kernel-internal synchronization functions \texttt{mutex_lock} and \texttt{mutex_unlock}.
- Besides \texttt{mutex_lock} and \texttt{mutex_unlock}, the \texttt{ULiX} kernel also supports synchronization with semaphores via the \texttt{wait_semaphore} and \texttt{signal_semaphore} functions which are not accessible from user mode applications.

11.6.1.3 Synchronizing the Kernel

Kernel synchronization deals with cases (3) and (4) in Figure 11.5. The main task of kernel synchronization is to achieve mutual exclusion of concurrent activities in the kernel. This is a challenge because concurrent activities are natural in operating systems, let they be introduced through interrupts or through multiple CPUs.
The case of interrupts, case (3) in Figure 11.5, is probably the toughest issue to solve because it happens in all operating systems:

An interrupt handler may share data with a process which has made a system call. As we will show on the following pages, we cannot use the same mutex/locking approach as for inter-thread synchronization, instead we will need to revert to hardware-based mechanisms discussed in Section 11.2. In fact, Ulix uses the simplest approach to synchronize the kernel: It realizes a non-interruptible kernel. What this means will be discussed shortly.

The case of multiprocessing, case (4) in Figure 11.5, is equally tough. When more than one CPU (or CPU core) is used, the situation becomes more complicated because code will be executed truly simultaneously on those CPUs or cores. But as we will see, if case (3) has been solved conceptually, it is not too hard to extend this solution to case (4). Furthermore, since Ulix does not support more than one CPU or core, we do not need to work out this case in code anyway.

### 11.6.2 Minimizing the Size of Critical Sections: Interruptible Kernels

We now turn to the central problem of implementing critical sections in the kernel, i.e., achieving mutual exclusion at the lowest layer. Until now, we declared critical sections
using the markers \(\langle \text{begin critical section in kernel } 380a \rangle\) and \(\langle \text{end critical section in kernel } 380b \rangle\). A kernel built in this way is called an interruptible kernel. Such kernels allow concurrent activities within the kernel, but only in some parts. For a monoprocessor system this means that interrupts may be enabled during the execution of some kernel code. (It does not mean that interrupts are always on.) For multiprocessor systems, spin locks are needed in addition to disabling the interrupts (see Section 11.2).

Building correct interruptible kernels boils down to finding all critical sections and ensuring that the entry and exit protocols to these sections are implemented correctly. Both problems are non-trivial and have caused much misery in the history of operating systems:

- It is easy to spot some critical sections, but it is hard to overlook no critical section. Overlooking a critical section (i.e., failing to mark it correctly) usually causes hard to diagnose system faults because bugs are usually the result of non-reproducible race conditions (so called Heisenbugs).

- It is easy to “play safe” and mark all possible candidates for critical sections as such, following the strategy: if in doubt, then it’s a critical section. But this results in rather large critical sections, and large critical sections cause their own problems (inefficiency being one). So the challenge is to declare “minimal” critical sections.

A typical approach is to let system call handlers be interruptible while disabling (all or some) interrupts during the execution of interrupt handlers.

These challenges have been debated by operating system designers for many decades. They effectively ask the question of the size of critical sections. The more code allows interrupts, the better the performance and responsiveness of the system can become, but at the same time complexity of synchronization increases.

Interestingly, there is a very simple synchronization strategy that works in most cases and allows you to start off with a system which is inefficient yet correctly synchronized: The most simple approach is to generally forbid interrupts while executing kernel code, thereby “maximizing” the size of critical sections in the kernel. We call such an implementation a non-interruptible kernel. For a single-CPU, single-core machine this means that all kernel code can expect to remain in control and keep the CPU until it either finishes or blocks (in a system call handler).

Only when the system runs in user mode (i.e., executes the application code of some thread), interrupts can occur. Note that this means that whenever a thread makes a system call (and thus transitions to kernel mode via the system call interface), interrupts will be disabled until the system call function completes its work and returns to user mode. In such systems, there will be no need to synchronize interrupt handlers and system call handlers.

Note however that system call handlers still have to consider synchronization because a thread executing such a system call handler may decide to block (for example, in order to wait for a disk operation to complete). In that case control will transfer to a different thread which might call some other or the same system call handler, thus possibly accessing the same kernel data structures. Therefore, the begin of a critical section might happen in thread A while the end of a critical section might happen in thread B (after a context switch). This needs to be considered.
Fortunately, the Intel CPU lets developers decide whether interrupts are automatically
disabled upon entering some handler by providing both interrupt gates (which auto-disable
interrupts) and trap gates (which do not), see also Section 6.4.1.

11.6.3 Ulix as a Non-Interruptible Kernel

We have decided to make the Ulix kernel non-interruptible which greatly reduces the
demand for synchronization. For completeness, Section 11.6.4 will describe the issues
arising with interruptible kernels and illustrate them with examples from the Ulix code.

As mentioned above, the Intel CPU lets developers decide whether interrupts are auto-
matically disabled upon entering some handler by providing both interrupt gates and trap
gates. We have used an interrupt gate in our system call implementation in Section 6.4.1.
So we can safely assume that interrupts are off whenever we execute code in kernel mode.
Since Ulix does not support multiple CPUs, there is nothing more to do.

Now, finally, we can show the entry and exit protocols for the critical sections in the
kernel. Recall that we marked them using the code chunks ⟨begin critical section in kernel
380a⟩ and ⟨end critical section in kernel 380b⟩. Since we have a non-interruptible kernel
and switch off interrupts using an interrupt gate, there is no further necessity for addi-
tional synchronization. Therefore, their implementations are empty.

[380a]  ⟨begin critical section in kernel 380a⟩ ≡
(168d 169a 184–87 209c 216b 219c 221a 255a 260a 276d 361c 362 366 391 392 416b 521a 530 539c 540c 545b 548b 551 580c 581)
   // do nothing

[380b]  ⟨end critical section in kernel 380b⟩ ≡
(168d 169a 184–87 212 216b 219c 221a 255a 260a 276d 277a 280b 361c 362 366 391 392 416b 521b 531a 545b 580c 581)
   // do nothing

The interested reader might ask: Why didn’t we omit these markers from the beginning
if they are empty anyway? There are two answers to this question:

1. Omitting these markers would have avoided the discussion (and identification) of
critical sections, which would have avoided some nice intellectual challenges.

2. Keeping these markers allows for a future evolution of Ulix into an interruptible
kernel.

In such a future Ulix version with an interruptible kernel these chunks would be im-
plemented using ⟨disable interrupts 47a⟩ and ⟨enable interrupts 47b⟩, or with the nestable
versions.

11.6.4 Illustrating Problems of Interruptible Kernels

In this section we describe some relevant problems that arise and discuss the additional
care that needs to be taken when the kernel is interruptible. Note that the situation would
become even more complex if several CPUs (or cores) were supported. Wherever possible,
we use code of Ulix to illustrate the problems.
11.6.4.1 Example: Accessing a Keyboard Buffer

We begin with an introductory example for the synchronization problems that arise from using an interruptible kernel. Consider the way that reading from the keyboard works for a thread:

- A thread calls the `read()` function (using file descriptor `STDIN_FILENO = 0`) which in turn makes a system call. This enters kernel mode and leads to the execution of `syscall_read()` which in turn will make an other system call that activates the system call handler `syscall_readchar()`.

- `syscall_readchar()` determines the current terminal `term`, reads a character from its associated buffer `term->kbd[]` and updates the current position and number of characters in the buffer (`term->kbd_lastread` and `term->kbd_count`). Note that those two changes modify a global data structure.

- On the other hand, the keyboard interrupt handler, `keyboard_handler()`, updates the same data structure: it determines the active terminal (the one currently displayed on the screen), enters the character’s ASCII code in the corresponding buffer and also modifies the buffer’s current position and number of contained characters.

If both accesses originated from threads, we could simply use a list of mutexes (one for each terminal) to lock these structures, but we cannot use the locking mechanism in the interrupt handler since that might put it to sleep if a thread had just acquired the lock right before the keyboard interrupt occurred. Thus, the obvious attempt of using a lock list `keyboard_buffer_lock[]` and implementing mutual exclusion via

\[
\begin{align*}
\text{void syscall_readchar (context_t *r)} & \{ & \text{[381a]} \\
\text{char c; } & \text{[381a]} \\
\text{int t = thread_table[current_task].terminal; } & \text{[381a]} \\
\text{terminal_t *term = &terminals[t]; } & \text{[381a]} \\
\text{// get character, return 0 if there is no new character in the buffer} & \text{[381a]} \\
\text{mutex_lock (keyboard_buffer_lock[t]); } & \text{[381a]} \\
\text{if (term->kbd_count > 0) } & \text{[381a]} \\
\text{\{ } & \text{[381a]} \\
\text{\quad term->kbd_count--; } & \text{[381a]} \\
\text{\quad term->kbd_lastread = (term->kbd_lastread+1) \% SYSTEM_KBD_BUFLEN; } & \text{[381a]} \\
\text{\quad c = term->kbd[term->kbd_lastread]; } & \text{[381a]} \\
\text{\\} } & \text{[381a]} \\
\text{mutex_unlock (keyboard_buffer_lock[t]); } & \text{[381a]} \\
\text{// ... } & \text{[381a]}
\end{align*}
\]

and

\[
\begin{align*}
\text{terminal_t *term = &terminals[cur_vt]; } & \text{[381b]} \\
\text{mutex_lock (keyboard_buffer_lock[cur_vt]); } & \text{[381b]} \\
\text{if (term->kbd_count < SYSTEM_KBD_BUFLEN) } & \text{[381b]} \\
\text{\{ } & \text{[381b]} \\
\text{\quad if (ctrl_pressed && c >= 'a' && c <= 'z') c -= 96; } & \text{[381b]} \\
\text{\quad term->kbd[term->kbd_pos] = c; } & \text{[381b]}
\end{align*}
\]
term->kbd_pos = (term->kbd_pos + 1) % SYSTEM_KBD_BUFLEN;
term->kbd_count++;
if (scheduler_is_active) { (keyboard handler: wake sleeping process 322a) }
}
mutex_unlock (keyboard_buffer_lock[cur_vt]);

This cannot work. However, note that the situation is not symmetrical: Whereas two threads will run alternately, this is not true for an interrupt handler which will interrupt the flow of control in a thread, but not vice versa. Thus it will be sufficient to prevent the interrupt handler from running at all while we access the data structures in a system call handler—we can simply disable interrupts before the critical section and re-enable them afterwards.

This implements a synchronization model that is called one-sided synchronization. Since disabling interrupts disturbs the overall functioning of the operating system (e.g., by delaying the next scheduling decision) we should limit this to very short time spans. Figure 11.7 shows the difference between two-sided inter-thread synchronization and the synchronization of system call and interrupt handlers.

**Synchronization Between Threads**

Thread 1

```c
mutex_lock (l);
// access...
mutex_unlock (l);
```

Thread 2

```c
mutex_lock (l);
// access...
mutex_unlock (l);
```

**Synchronization Between System Call and Interrupt Handlers**

Thread / System Call

```c
⟨disable interrupts⟩
// access...
⟨enable interrupts⟩
```

Interrupt Handler

```c
// interrupts are off
// access...
```

Figure 11.7: Mutex-based synchronization of threads (top) is an example of two-sided synchronization; data which is shared between system call and interrupt handlers is handled via a one-sided synchronization method (bottom; i.e., disabling interrupts in the system call handler).

A working implementation of syscall_readchar() thus looks as follows:
11.6 Kernel Synchronization

\[383a\]  
void syscall_readchar (context_t *r) {  
    char c;  
    int t = thread_table[current_task].terminal;  
    terminal_t *term = &terminals[t];  
    
    // get character, return 0 if there is no new character in the buffer  
    ⟨disable interrupts 47a⟩  // critical section starts here  
    if (term->kbd_count > 0) {  
        term->kbd_count--;  
        term->kbd_lastread = (term->kbd_lastread+1) % SYSTEM_KBD_BUFLen;  
        c = term->kbd[term->kbd_lastread];  
    }  
    ⟨enable interrupts 47b⟩  // critical section ends here  
    // ...

and that is also the way it could be implemented in Ulix (see page 416). The corresponding critical section in the interrupt handler would need no protection because there it would be impossible for the system call handler’s code to interfere with the running interrupt handler.

11.6.4.2 Restoring Old Interrupt States

As discussed in Section 11.2.4, marking critical sections needs to be done with care since nested critical sections can cause premature enabling of interrupts. This can be avoided by using “nestable” entry and exit protocols for critical sections, as discussed. However, sometimes interrupts might have been turned off by other means (e.g., the interrupt gate) and so we might not even know whether interrupts were enabled before we wish to disable them. For example, deblock_186b calls remove_from_blocked_queue_186a and add_to_ready_queue_184b, and deblock_186b itself is both called from the interrupt handler keyboard_handler_39q and from the syscall handler syscall_kill_565c via u_kill_562b and a further helper function (see Figure 11.8).

Thus, for a Ulix version with an interruptible kernel, we might enter deblock_186b with interrupts on or off, depending on which function calls it.

In those situations we want to restore the original state afterwards. Thus, we need to save the current state of the interrupt flag (IF) which is bit 9 of the EFLAGS register before we disable interrupts. We can use a global variable

\[383b\]  
boolean if_state;  // state of the interrupt flag (IF)

Defines:  
    if_state, used in chunks 357 and 384.

for storing the state because interrupts will always be off after reading the state and until it is restored.

The following two new code chunks save and disable interrupts and restore them, respectively:
(a) Keyboard interrupt occurs
keyboard_handler ()
deblock ()
- interrupts are off
- needs to modify thread queues
(b) Process sends a signal
syscall_kill ()
u_kill ()
wake_waiting_parent_process ()
deblock ()
- interrupts are on
- needs to modify thread queues

Figure 11.8: Two ways to enter the deblock functions with interrupts off or on.

[384a] (save and disable interrupts 384a)≡
{ // create scope for scope-local variable eflags
  int eflags;
  asm volatile (   "pushf \n" // push EFLAGS
                     "cli \n" // disable interrupts
                     "movl (%esp), %0 \n" // copy to eflags variable
                     "addl $4, %esp \n" // restore stack pointer
                 : "=r"(eflags) );
  if_state = (eflags >> 9) & 1; // bit 9 of EFLAGS is IF
}
Uses if_state 383b.

[384b] (restore interrupts 384b)≡
if (if_state == 1) {
  (enable interrupts 47b)
}
Uses if_state 383b.

The functions add_to_blocked_queue\textsubscript{185c} and remove_from_blocked_queue\textsubscript{186a} as well as add_to_ready_queue\textsubscript{184b} and remove_from_ready_queue\textsubscript{184c} which handle the blocked queues and the ready queue could use this feature.

11.6.4.3 Finding Critical Sections

As mentioned above, identifying critical sections is one of the major problems in interruptible kernels. It needs much experience to do this correctly, and doing it minimally is more an art than a science. A best-practice approach is to invest much discipline during development and for every new data structure investigate thoroughly whether this data
structure is critical and, if yes, who accesses it. These accesses must be declared as critical sections.

We will now discuss these points using Ulix code. For example, look at all the interrupt handlers of Ulix and search for data structures which are accessed elsewhere. For an interruptible kernel we would have to make sure that interrupts are disabled whenever such an access (outside interrupt handlers) occurs. Luckily, the number of handlers is small—there are only five interrupt handlers:

- `keyboard_handler` served as our initial example and has already been dealt with.
- `timer_handler` periodically activates the scheduler—we will discuss it below.

The next three handlers deal with filesystem activity; you will only fully understand the following discussion after reading Chapters 12 (Filesystems) and 13 (Disk I/O).

- `ide_handler` modifies the global `hd_buf` buffer and the `hd_direction` variable and deblocks a thread that has been waiting for the completion of a hard disk action. Other than from the IDE handler, any access to these data will originate from a system call handler initiated by a thread trying to read from or write to disk.

Since the implementation of the filesystem code does not allow multiple parallel accesses to the disk (once a transfer has been initiated, all other threads block on a mutex `hd_lock`), no thread will access the data before a current transaction has completed. Similarly, the IDE controller will only generate an interrupt after a (single) thread has made a system call that accesses the disk.

- `floppy_handler` calls `fdc_wakeup` which manipulates an entry in the thread list (like `ide_handler`, it deblocks a thread that has been waiting for the floppy drive’s action to complete).

Floppy access is also serialized for threads using an `fdc_lock` variable (similar to `hd_lock`). `fdc_timer` (which is called by the timer handler) checks the lock state of `fdc_lock` but does not change it.

- `serial_hard_disk_handler` also deblocks a thread which caused the recently finished action of the serial hard disk. Again, there is a `serial_disk_lock` variable which is used to serialize parallel accesses to the serial disk device.

For all hard disk, floppy disk and serial disk accesses the following order of execution is enforced:

1. A thread initiates disk access which leads to acquisition of one of the locks `hd_lock`, `fdc_lock`, or `serial_disk_lock`. Eventually the thread will block (and wait for the disk operation to complete).
2. One or more interrupts are generated by the controller, thus the corresponding interrupt handler will be executed. When Ulix detects that the requested operation has completed, it will wake up the thread.
3. When the scheduler selects the thread, it will unlock the lock. Only then can the system start the next disk access (if other threads are waiting).
Thus, we do not need one-sided synchronization in the disk I/O code. Access to different devices (e.g., a hard disk and a floppy disk) can happen in parallel.

Note that the ready and blocked queues are manipulated by these interrupt handlers. These are critical regions because those queues are also modified via system calls (when a new process or thread is created or exits or when a process asks to wait for termination of a child process). Thus the functions

- `u_fork` (forks a process; called by `syscall_fork`)
- `syscall_exit` (exits a process)
- `u_pthread_create` (creates a new thread in the current process)
- `syscall_waitpid` (makes a process wait for termination of a child)

(in an interruptible kernel) would have to disable interrupts while accessing the thread table in order to ensure that they cannot be interrupted by one of the interrupt handlers. (`syscall_pthread_exit` uses `syscall_exit` to make the thread terminate.)

The timer handler calls the scheduler which also modifies the thread list. Since it runs with deactivated interrupts it cannot conflict with the other functions that modify this list.

An earlier version of the U*n* code (up to release 0.12) used a `thread_list_lock` mutex for controlling access to the thread list (e.g., when creating a new thread or removing one from the list). However, the scheduler (which is sometimes started from a thread when it exits or yields, but typically runs on behalf of the timer interrupt handler) must not block and thus cannot use the mutex.

### 11.6.4.4 Dealing With Complex Handlers

In more complex operating systems handling an interrupt can become time-intensive. Since disabling interrupts should be limited to short time spans (see above), in those cases the approach of splitting the handler code into a “first-level interrupt handler” and a “second-level interrupt handler” can help.

1\textsuperscript{st}-level handler

- **(top half)**
  - The first-level interrupt handler (sometimes called top half) is registered as the regular handler and runs with other interrupts disabled. It performs only the most important tasks (e.g., it acknowledges the interrupt and saves volatile data from a device’s internal buffer). As a last step it creates the lower half of the handler, re-enables interrupts and terminates.

2\textsuperscript{nd}-level handler

- **(bottom half)**
  - The second-level interrupt handler (sometimes: bottom half) is not part of the interrupt handler and runs while interrupts are enabled. It is somewhat similar to a kernel thread but without an address space of its own (it only uses the kernel’s memory). The bottom half could be activated by the scheduler (in that case bottom halves would need to have a higher priority than regular threads) or some other mechanism could be used for making sure that the bottom halves run as early as possible—as long as no other top half has to be executed because new interrupts occurred.

The Linux kernel uses this concept and uses the terms *top half* (for the first-level handler) and *bottom half or tasklet* (for the second-level handler) [Lov03, pp. 81–106]. Tasklets
in the Linux 2.6 kernel can have one of two priorities where the higher-priority tasklets are always executed before the lower-priority ones and both run before the next thread is scheduled. The top half of the interrupt handler registers a tasklet handler. Those tasklet handlers must not block (just like the top halves, they cannot be scheduled, so it is not possible for them to use locks or semaphores), and they must be able to cope with being interrupted.

### 11.6.4.5 Spurious Interrupts

A spurious interrupt is an interrupt whose occurrence is faulty and unexpected. Yet, on physical hardware it is a problem that needs to be dealt with. For example, the Intel 8259 PIC which Ulinx uses allows the discovery of a spurious interrupt:

“In both the edge and level triggered modes the IR inputs must remain high until after the falling edge of the first INTA. If the IR input goes low before this time a DEFAULT IR7 will occur when the CPU acknowledges the interrupt. This can be a useful safeguard for detecting interrupts caused by spurious noise glitches on the IR inputs. To implement this feature the IR7 routine is used for ‘clean up’ simply executing a return instruction, thus ignoring the interrupt. If IR7 is needed for other purposes a default IR7 can still be detected by reading the ISR. A normal IR7 interrupt will set the corresponding ISR bit, a default IR7 won’t.” [Int88, p. 18](ISR is the In-Service Register.) Thus it is sufficient to modify handlers for interrupt numbers 7 (coming from the first PIC) and 15 (from the second PIC). In case of an interrupt from the first PIC, no action is required; especially it is not necessary to acknowledge the interrupt. However, if the interrupt comes from the second PIC, the first PIC (and only the first PIC) needs to be sent the acknowledgement.

Certain spurious interrupts can be detected by the system because interrupts usually provide services with prior demands. So if there is an interrupt which has no obvious demand, then it probably is spurious. Other types of spurious interrupts are harder to detect. Since Ulinx uses neither interrupt 7 (first parallel port) nor 15 (secondary IDE controller), we fortunately need not deal with this situation.

### 11.6.4.6 Lost Wakeup

Blocking processes until an interrupt occurs introduces a problem that is called lost wakeup. Lost wakeups can be caused by bad programming or unfortunate circumstances.

For example, if the interrupt that should deblock a thread is received before the thread is actually blocked, the deblock operation is not triggered and the thread might never get deblocked.

Lost wakeup situations are hard to deal with in practice since it is difficult to distinguish the case where a wakeup is lost or merely very slow. Lost wakeups usually result in deadlocks or threads being blocked forever. So in principle, techniques to discover deadlocks or timeouts on the waiting times of threads can help resolve this issue.
The same problem can occur with signals (see Chapter 14) which might be sent to a process which is not yet waiting for them.

Lost wakeups could occur in a U\texttt{Lix} version with an interruptible kernel whenever a process or thread is blocked and the reason for blocking can disappear before the block operation completes. There are several such potential situations in the interruptible version’s U\texttt{Lix} code, but they cannot occur in our non-interruptible version. We have, however, marked the critical sections in some places so that these cases are already dealt with (preparing again for the transition to an interruptible kernel).

As before, some of the following explanations will only make sense once you have read later chapters, so we suggest you return here after (for example) reading Chapter 13.5 which discusses the hard disk controller.

- \texttt{syscall\_waitpid} blocks a process so that it can wait for a child process to terminate. In the interruptible version of U\texttt{Lix}, running this function with interrupts disabled would solve the possible situation of the child exiting after the parent entered \texttt{syscall\_waitpid} but before it blocked.

- hard disk access (see Chapter 13.5): before sending a request to the disk controller, in the interruptible version of U\texttt{Lix}, the code would disable the interrupts. It would then check whether the request has already been completed and potentially block, re-enabling the interrupts after the thread has been moved to the blocked queue. This concerns the functions \texttt{write\_sector\_hd} and \texttt{read\_sector\_hd}.

- floppy disk access (see Chapter 13.6): this is analogous to the hard disk case, but disabling and enabling interrupts would occur in different functions.

The functions \texttt{fdc\_read\_sector} and \texttt{fdc\_write\_sector} call \texttt{fdc\_command} which would disable interrupts and send a control sequence to the floppy controller. Then \texttt{fdc\_command} calls \texttt{wait\_fdc\_interrupt} which in turn calls \texttt{fdc\_sleep} where interrupts would be re-enabled after blocking the thread.

- serial disk access (see Chapter 13.4): only read access can block, and this situation would be handled in the same way as hard disk access: before sending a read request to the serial disk, interrupts would be disabled.

- \texttt{syscall\_readchar} (as discussed above) reads from a keyboard buffer and blocks if that is empty. Here, interrupts would be turned off between checking the buffer and calling \texttt{block}.

- \texttt{mutex\_lock} blocks when the mutex is already held elsewhere. Since this function could be called in situations where interrupts may already be disabled, it saves the current state (interrupts on or off) and restores it before returning (when the lock has been acquired; see Section 11.6.4.2).

In all these cases, when a thread blocks, interrupts would be re-enabled after moving a thread to a blocked queue via \texttt{block}. 

11.7 Further Reading

An excellent collection of synchronization problems (and solutions) is “The Little Book of Semaphores” by Allen B. Downey [Dow08], which is freely available on the publisher’s website. It presents both classical and less well-known problems, and solutions are not simply given, but developed step by step—passing through several failed attempts and explaining why they failed.

For details on the POSIX synchronization functions, take a look at “Programming with POSIX Threads” by David R. Butenhof [But97]. Its chapter 3 (which deals with synchronization) is freely available online.

“The Art of Multiprocessor Programming” by Maurice Herlihy and Nir Shavit [HS12] is a very thorough introduction to the synchronization problems, especially those that occur on multiprocessor machines. As we mentioned earlier in Chapter 8.2, things get much more complicated when more than one CPU is used. It contains a lot of code that exemplifies the presented theory. That code is also available online.

A detailed discussion of top and bottom halves (tasklets) in the Linux kernel can be found in chapter 6 of Robert Love’s kernel development book [Lov03].

11.8 Exercises

30. Multiprocessor Kernel Synchronization

This exercise deals with the implementation of critical sections on multiprocessor systems, i.e., systems with multiple CPUs or CPUs with multiple cores. Assume you have a kernel in which critical sections are correctly labelled with ENTER_MUTEX and EXIT_MUTEX. For simplicity, assume that this holds for all interrupt handlers and all system calls. Assume further that two kernel level threads A and B are running on the system and that at least two CPUs execute these threads.

For each of the following cases discuss whether one of the following two situations can arise:

a) violation of mutual exclusion, i.e., threads A and B concurrently execute critical sections.

b) deadlock, i.e., at least one thread will never be scheduled again.

If one of these conditions can occur, construct a schedule of the system that ends in the condition being true. If the conditions can never occur, argue why it is impossible.

- Case 1: single CPU, ENTER_MUTEX and EXIT_MUTEX are empty.
- Case 2: single CPU, ENTER_MUTEX disables the interrupts on that CPU, ENTER_MUTEX enables the interrupts again.

• Case 3: single CPU, ENTER_MUTEX contains a spin lock on a global bit called busy and assigns 1 when the lock is taken. EXIT_MUTEX assigns 0 to busy.

• Case 4: single CPU, ENTER_MUTEX and EXIT_MUTEX are implemented as follows, using the definition of Lock from Section 11.2.2.2 (page 354):

\[ \text{Byte busy = false;} \]
\[ \text{ENTER_MUTEX} \{ \]
\[ \langle \text{disable interrupts 47a} \rangle \]
\[ \text{while (Lock()) == true);} \]
\[ \} \]
\[ \text{EXIT_MUTEX} \{ \]
\[ \text{busy = false;} \]
\[ \langle \text{enable interrupts 47b} \rangle \]
\[ \} \]

Uses busy, ENTER_MUTEX, and EXIT_MUTEX.

• Case 5: single CPU, ENTER_MUTEX and EXIT_MUTEX are implemented as follows:

\[ \text{Byte busy = false;} \]
\[ \text{ENTER_MUTEX} \{ \]
\[ \langle \text{disable interrupts 47a} \rangle \]
\[ \text{while (Lock()) == true);} \]
\[ \langle \text{enable interrupts 47b} \rangle \]
\[ \} \]
\[ \text{EXIT_MUTEX} \{ \]
\[ \langle \text{disable interrupts 47a} \rangle \]
\[ \text{busy = false;} \]
\[ \langle \text{enable interrupts 47b} \rangle \]
\[ \} \]

Uses busy, ENTER_MUTEX, and EXIT_MUTEX.

• Case 6: two CPUs, ENTER_MUTEX and EXIT_MUTEX are implemented as in case 1.
• Case 7: two CPUs, ENTER_MUTEX and EXIT_MUTEX are implemented as in case 2.
• Case 8: two CPUs, ENTER_MUTEX and EXIT_MUTEX are implemented as in case 3.
• Case 9: two CPUs, ENTER_MUTEX and EXIT_MUTEX are implemented as in case 4.
• Case 10: two CPUs, ENTER_MUTEX and EXIT_MUTEX are implemented as in case 5.

31. Nestable Critical Sections on Multiprocessor Systems

Does the implementation of nestable critical sections presented in chunks \( \langle \text{nestable begin critical section 357d} \rangle \) and \( \langle \text{nestable end critical section 357e} \rangle \) work on multiprocessor systems?

Does it work if disabling the interrupts is accompanied by a spin lock?
32. **Kernel Level Semaphores as Critical Sections**

Consider the implementation of `wait_semaphore` and `signal_semaphore` in Section 11.3.4 and assume it were not declared as a critical section (i.e., interrupts remain on during execution of the code apart from within dispatcher operations). Is the implementation then still correct? Try to construct an example where two threads use a single semaphore, a context switch occurs and semaphore semantics are violated.

33. **Implementing Kernel Level Semaphores**

Consider the following implementation of `wait_semaphore` and `signal_semaphore` that use atomic counter manipulations instead of non-atomic ones as in the actual implementation:

```c
void wait_semaphore (kl_semaphore_id sid) {
    kl_semaphore sem = ⟨semaphore structure with identifier sid⟩;
    ⟨begin critical section in kernel⟩
    int count = __sync_sub_and_fetch (&sem.counter, 1); // atomic "--sem.counter"
    if (count < 0) {
        block (&sem.bq, TSTATE_LOCKED);
        ⟨resign 221d⟩
    }
    ⟨end critical section in kernel⟩
}

void signal_semaphore (kl_semaphore_id sid) {
    kl_semaphore sem = ⟨semaphore structure with identifier sid⟩;
    ⟨begin critical section in kernel⟩
    int count = __sync_add_and_fetch (&sem.counter, 1); // atomic "++sem.counter"
    if (count < 1) {
        blocked_queue *bq = &sem.bq;
        thread_id head = bq->next;
        if (head != 0) {
            deblock (head, bq);
        }
    }
    ⟨end critical section in kernel⟩
}
```

Uses `__sync_add_and_fetch`, `__sync_sub_and_fetch`, `blocked_queue`, `deblock`, `kl_semaphore`, `kl_semaphore_id`, `signal_semaphore`, `thread_id`, `TSTATE_LOCKED`, and `wait_semaphore`.

The atomic counter manipulation is done using the gcc compiler’s built-in functions `__sync_add_and_fetch` and `__sync_sub_and_fetch` [Int01, section 7.4.1, p. 59] that require that we add the option `-march=i586` to the compiler flags (CFLAGS). The implementation of those functions is semantically equivalent to

```c
int __sync_sub_and_fetch (int *variable, int value) {
    *variable -= value;
}
```
return *variable;
}

int __sync_add_and_fetch (int *variable, int value) {
*variable += value;
return *variable;
}

Defines:
__sync_add_and_fetch, used in chunk 391a.
__sync_sub_and_fetch, used in chunk 391a.

but performs all steps atomically.

The question is whether this implementation is in any aspect better than the one given in Section 11.3.4. Would it change anything if the body of the function were not protected as a critical section? (The latter question is an extension of exercise 32.)

34. Locks as Critical Sections

Consider the following variation of the implementation of mutex_lock366a. The only difference is that the (begin critical section in kernel 380a) is moved to within the loop.

void mutex_lock (lock lockvar) {
    if (current_task == 0) { return; } // no process
    while (lockvar->l == 1) {
        (begin critical section in kernel 380a)
        block (&(lockvar->bq), TSTATE_LOCKED); // put process to sleep
        (resign 221d)
    }
    lockvar->l = 1;
    (end critical section in kernel 380b)
}

Is this implementation correct? Does it matter if the kernel were interruptible?

35. Locks as Critical Sections (Variation)

Here is another variation of the mutex_lock366a implementation. It uses the gcc-internal function __sync_lock_test_and_set354a discussed in Section 11.2.2.1 to guarantee that testing and setting are performed atomically.

void mutex_lock (lock lockvar) {
    if (current_task == 0) { return; } // no process
    while (__sync_lock_test_and_set (&(lockvar->l), 1) != 0) {
        (begin critical section in kernel 380a)
        block (&(lockvar->bq), TSTATE_LOCKED); // put process to sleep
        (resign 221d)
    }
    (end critical section in kernel 380b)
}

Does this improve the correctness?
In this chapter we describe how operating systems store files on hard disks and floppy disks. The central concept for organizing directories and files is the filesystem: it is an abstract description of the required data structures.

In Chapter 13 we will look at what is needed to actually talk to a physical drive, but for now let’s just assume that there is some mechanism which enables us to read and write “blocks”: these are small chunks of disk storage into which we partition a disk—quite similar to the way that we’ve split memory into page frames.

We start with an overview of filesystem concepts in Section 12.1 that briefly discusses CP/M, FAT, NTFS and Unix filesystems. Section 12.2 explains the concept of mounting devices to mount points which is used on Unix operating systems. After this short theory block we jump right into the implementation details: First, Section 12.3 shows how the virtual filesystem (VFS) is organized in Ulix. It provides some abstractions which allow the OS to locate and use files on media which are formatted with various filesystems. In Section 12.4 we present the new system calls that can be used by user mode programs.

Then, in Sections 12.5 and 12.6 we introduce the Minix filesystem and show its implementation in Ulix.

Section 12.7 presents a second filesystem (for accessing device files in /dev), so you will see that the virtual filesystem layer is actually put to good use.

Finally, Section 12.8 gives a very short overview of the directory hierarchy that Ulix uses (which is modeled after other Unix systems).
12.1 Introduction to Filesystems

Every operating system needs to support one or more filesystems—at least one is required so that the OS can store and retrieve data, load programs from the disk and enable them to access files. Support for more than one filesystem makes sense when the OS wants to read media from other systems, e.g. FAT-formatted media which can be used on most of today’s systems.

Why is there no single filesystem which all operating systems could agree on? Surely, this would make the cooperation of diverse systems much simpler.

If we want to understand why every OS has its own idea of how to store files (and possibly directories) on disk we have to look back to the beginnings of external storage.

Card Readers Early computers used punch cards (see Figure 12.1) to store jobs and the associated data: a card reader would be filled with a stack of such cards, the first set of cards contained the binary program to be run, and the following cards held the data. The system would read all these cards into memory and then run the job. After completion, the results of the computation would be punched on empty cards or sent to a connected printer. Data organization in such a card stack was strictly serial, so there was no concept of a filesystem: the first card(s) would describe how many program and data cards would follow and where to store their contents in RAM.

Figure 12.1: Punch cards were used to store program code and data.

Tape The next step was the introduction of magnetic tape drives. These live on until today (as backup media, or third-level storage), and they are also strictly serial: Typically tape drives can be sent a `rewind` command to move to the beginning of the tape and `read` and `write` commands to read or write the tape sequentially. While it is possible to store more than one file on a tape, accessing the third file requires skipping the first two ones—which can only be achieved by reading them first. Thus, tapes have no filesystem, either. Today, when people use a tape drive for backup, they typically
generate an archive file (which may or may not contain a central listing of the files) and write this single archive file onto the tape. The Unix \texttt{tar} program (\texttt{tar}: tape archive) writes a file header for each contained file and then the file’s data; then the next file header and data follow. The created archive file is written raw to the tape.

\textbf{Disk} The seriality of storage was changed with the introduction of disks: it does not matter whether you think of floppy or hard disks, all of them allow \textit{random access}, that means you can store several files on one disk and read any file (or part of a file) without looking at other data as long as you know where to find the file on the disk. Thus, disks need some kind of \textit{directory} information so that the machine can look up files on a previously unseen disk. The question of how and where to store this directory information on the disk defines (most of) a filesystem, and many OS developers have had their own ideas about the organization of files on a disk. Early machines were not meant to be compatible with machines of other manufacturers, and this led to various incompatible filesystems. We’ll look at some examples in the next section.

\subsection{12.1.1 Simple Filesystems}

We assume throughout the rest of this chapter that a disk is divided into \textit{blocks} of some fixed size (typically 512 or 1024 bytes), and these blocks can be read and written using some kind of \texttt{readblock} and \texttt{writeblock} commands which the disk drive controller supports. We’re not going to delve into the details of how a hard disk is organized into platters, cylinders, sectors and tracks; instead we will assume that there is a logical ordering of blocks and that the controller allows to access these \textit{logical blocks} directly via some mechanism that the OS can use.

\subsubsection{12.1.1.1 Contiguous Filesystems}

As long as you want to write files to a disk only \emph{once} (and then continue using it in read-only-mode), organizing files is rather simple: Assume you want to store 150 files on the disk. You can then reserve one or more blocks at the beginning of the disk for a central directory—just enough space to store filename, starting block and file length for each file (see Table 12.1). Then write each file to the disk, starting at the first unused block, and update the directory afterwards.

All files are stored \textit{contiguously}, which means that you can later read them sequentially as long as you know where to start reading (see Figure 12.2). Opening and reading files then means just looking up the filename in the directory and starting to read at the start block number which is stored next to the filename.

The ISO-9660 filesystem [ECM87] which is used for compact discs works in a similar way, so this concept is still in use today. When we first implemented a disk driver for Ulix we used this approach as a quick hack to create a read-only filesystem.

Why is there a problem with this kind of organization? Imagine you want to enable write-support for such a filesystem. As long as you only modify blocks inside a file, ev-
Table 12.1: This is a simple directory of a contiguous filesystem. It stores both the number of blocks as well as the actual file size which might be less than a multiple of the block size (1024). The number of blocks could also be calculated from the file size.

<table>
<thead>
<tr>
<th>Filename</th>
<th>Size</th>
<th>First Block</th>
<th>Block Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>somefile.txt</td>
<td>3301</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>otherfile.txt</td>
<td>49152</td>
<td>6</td>
<td>48</td>
</tr>
<tr>
<td>test.txt</td>
<td>11147</td>
<td>54</td>
<td>11</td>
</tr>
</tbody>
</table>

Figure 12.2: Organization of disk blocks in a contiguous filesystem: The T blocks hold the table of contents, and A, B and C represent the three sample files.

everything is fine: In the same fashion used for reading a file, you can calculate the block number from the first block and the file offset and then change the right data block on the disk to update the file’s contents. But what about appending to a file? Here’s where we run into problems. Once you reach the end of the last block of a file, you cannot go on, since the next block on the disk belongs to a different file. You would first have to move all blocks of the following file to an unused region on the disk (and update the directory) before you can continue to write new blocks for the first file. But such a procedure will leave holes in the disk: unused regions which may be used for new files but more often than not will be too small to store a complete file inside. This is called external fragmentation.

The solution to this problem is to not store just a starting block and the file length, but a collection of block numbers: with such a method any free block may be used to store file contents, and the order of the blocks is irrelevant. Also, files can then be spread all over the disk instead of being contiguous.

12.1.1.2 Non-contiguous Filesystems

Giving up the contiguousness makes file storage more flexible, but reading and writing become harder: once you start reading or writing to a file, you can only read/write until the end of the current block; to continue you need to first look up the block number of the next data block (which can be found in some kind of directory, see Table 12.2 and Figure 12.3). And again, this information is only good until the end of that block, when you need to look up the next block number.
12.1 Introduction to Filesystems

<table>
<thead>
<tr>
<th>Filename</th>
<th>Size</th>
<th>Block List</th>
</tr>
</thead>
<tbody>
<tr>
<td>somefile.txt</td>
<td>3301</td>
<td>2, 3, 16, 44</td>
</tr>
<tr>
<td>otherfile.txt</td>
<td>49152</td>
<td>4, 5, ..., 15, 17, 18, ..., 43, 55, 56, ..., 63</td>
</tr>
<tr>
<td>test.txt</td>
<td>11147</td>
<td>45, 46, ..., 54, 64</td>
</tr>
</tbody>
</table>

Table 12.2: This is the directory of a non-contiguous filesystem. Instead of start block and block count it stores a block list for each entry.

![Figure 12.3: Organization of disk blocks in a non-contiguous filesystem: Files can use any set of available blocks.](image)

Most modern filesystems work in this way, so the question remains where to store the block numbers.

The new flexibility comes at a cost: reading a disk sequentially is cheap because the read/write heads are always positioned properly for reading the next block. When you allow a file to spread over the disk, the disk must seek to the next block which costs time: it slows down reading. A disk that holds lots of files which are spread over the disk blocks in this fashion is called fragmented, and many operating systems provide defragmenting tools which reorganize the files so that all blocks of a file occur in-order on the disk. However, a freshly defragmented disk becomes fragmented again once you start deleting files and writing new ones.

### 12.1.1.3 CP/M

An early example for non-contiguous filesystems is that of CP/M, an operating system which was popular in the 70s and early 80s. Actually, there was a variety of non-compatible CP/M filesystems, but they were at least all similar.

CP/M’s filesystem was flat, i.e., it did not implement the concept of subdirectories. There was one central directory that held the information about all files on the disk. For bringing some order into a big list of files, CP/M introduced the concept of a user number: each file’s metadata contained a number between 0 and 15 (standard: 0) and by changing the current user number to \( n \) the internal \( \text{DIR} \) command would only show files with that user number \( n \). That did not protect a user’s files from access by another user, but it allowed cooperative and trusted users to share a disk.
Block numbers for each file were stored in the directory entry (which was 32 bytes long, with 16 bytes reserved for block numbers). If a file used more data blocks than one directory entry could address, another directory entry for the same file was created (and called an extent). One directory entry could hold either 16 or eight block numbers, depending on the total size of the disk: early floppy disks stored \( \approx 180 \) KByte, and the blocksize was 1 KByte, thus on those floppies one byte was enough to reference a block [JL83, pp. 19 ff.].

CP/M files have three attributes which may be set or unset: ‘read-only’, ‘system’ and ‘file changed’. These are stored in the highest bits of three of the filename bytes—filenames must be made of ASCII characters which use only seven bits of each byte. CP/M filenames follow the “8.3” convention which was later picked up by MS-DOS: the first eight characters named the file, the other three characters were used for the file extension which defined the filetype (e.g. COM, C, PAS). Lower-case letters were forbidden. The dot in a filename such as TEST.COM was not stored in the directory entry, and you could not create names like X.ENDING, since the filename and extension were treated separately (with their eight characters / three characters limits).

### 12.1.1.4 MS-DOS FAT

The FAT filesystem (File Allocation Table) of MS-DOS and Windows uses a different approach for storing the block numbers. Instead of blocks, disks are divided into clusters, and the size of a cluster depends on the filesystem size. For example, with FAT16 the cluster size is 512 bytes for filesystem sizes up to 32 MByte, it is 32 KByte for filesystems with a size between 1 GByte and 2 GByte [Mic00b].

There are three variants of FAT named FAT12, FAT16 and FAT32. For example, a FAT16 directory entry maps a filename to the first data cluster of the file. Further clusters can be found via traversing a linked list, the file allocation table: it contains one 16-bit entry for each cluster, and such an entry has one of the following values:

- **0x0000**: Free cluster.
- **0x0002–0xFFFF**: points to the next cluster in the linked list.
- **0xFFF0–0xFFFF**: Reserved.
- **0xFFF7**: Bad cluster (cannot be used).
- **0xFFF8–0xFFFF**: This is the last cluster of the file. (Microsoft operating systems only use the 0xFFFF value.)

(The size of the cluster number (16 bit) is what gives FAT16 its name. For FAT12 and FAT32 there are similar conventions, e.g. **0x0FFFFFFF** (28 1-bits) marks the end of the list on FAT32, and **0xFFF** (twelve 1-bits) on FAT12 media. As Microsoft’s specification [Mic00b] notes, FAT32 is actually “FAT28” since only the lower 28 bits of a cluster number are interpreted.)

This means that it is impossible to quickly access the end of a large file, for example a file with FAT16’s maximum size (which uses 65522 clusters) requires reading 65522 FAT entries in order to determine the last cluster number.
12.1.2 Advanced Filesystems

Leaving FAT and other classical filesystems behind, we now look at two more advanced specimens: the NTFS filesystem which was introduced with Microsoft Windows NT and the Unix way of storing files.

12.1.2.1 NTFS

The New Technology Filesystem (NTFS) is a successor to both FAT and the High Performance Filesystem (HPFS) that was developed for OS/2 by Microsoft and IBM. When the two companies ended the cooperation on OS/2, IBM kept on using HPFS, and Microsoft developed Windows NT which introduced NTFS. The central data structure of an NTFS volume is the Master File Table (MFT) which contains entries for each file and each directory, including itself (since the MFT is also a file). Filename and data are attributes of a file. The filename is always part of the MFT entry, and the file data may also be if they are small enough. Otherwise, the data are external to the MFT entry. In that case, NTFS stores information about one or more cluster runs: A cluster run is a contiguous set of clusters, identified by a starting cluster and the number of clusters. Each such cluster run description is encoded so that is uses as few bytes as possible, it starts with a header byte in which the high half-byte gives the size (in bytes) of the following start cluster number, and the low half-byte tells how many bytes are used to describe the number of clusters (they follow behind the first encoded number).

As an example, 32 EF CD AB 02 01 (all numbers are hexadecimal) describes a cluster run with the following properties:

- 32: three bytes for the start cluster number
- 32: two bytes for the cluster count number
- EF CD AB: little-endian encoding of the start cluster number, 0xabcd
- 02 01: little-endian encoding of the cluster count number, 0x0102

This means that the cluster starts at cluster 0xabcd and ends at cluster 0xabcdef0 (= 0xabcd + 0x0102 − 1).

If this is the only cluster (i.e., the file is contiguous or non-fragmented), a further 0 byte ends the description, otherwise the encoded form of the next cluster run follows; in any case the last cluster run description is followed by a 0 byte, ending the entry. This may degenerate into a classical list of used clusters where the cluster count number is always 1, but that is not normally the case.

In comparison to FAT, access to an arbitrary cluster is much faster because all the information that is needed to find the n-th cluster of a file is stored directly in the MFT entry, whereas on a FAT volume half the cluster chain must be inspected (on average).

NTFS allows a file to have several names; it can store more than one filename attribute in an MFT entry. For example, besides the regular name (as seen on Windows) files can also have a short filename for compatibility with old MS-DOS applications.

Directories on NTFS are also files (again with an MFT entry) and their file entries are organized as B-trees. Directory entries map a name to the MFT entry of the associated
file, but also (redundantly) store the file size and time stamps that are also available via each file’s MFT entry [Cus94, p. 28].

The organization into directories that map filenames to MFT entries somewhat resembles the Unix filesystem structure where directories map filenames to inodes (see below).

NTFS provides several interesting features, for example it performs journaling and allows more than one standard data stream: The file contents are considered the default data stream, but there can be further, named data streams which can be accessed via a filename:streamname syntax. These extra streams could be used to implement versioning of files (keeping several old version of a file).

### 12.1.2.2 Unix Filesystems

Where CP/M and MS-DOS link all data to a filename as the significant identifier in a directory entry, Unix filesystems work differently: They assign numbers to files, not names. These numbers are called inode numbers and they point to entries in an inode table. Each inode (index node) stores metadata about a file, such as ownership, access permissions, file creation and modification dates and some kind of pointers to the file’s data blocks.

Files get a name by writing that name and an inode number into a special directory file. As an inode may occur several times in such directory files, any file can have more than one name. Thus, filenames are not unique.

Like a CP/M directory entry, a Unix inode has a few fields which contain block numbers and lead to the first data blocks of a file. However, in order to support large files, a different mechanism is needed because the inode has a fixed (and small) size. We have already discussed in Section 3.2.4.9 (p. 81) that Unix filesystems work with several layers of indirection—how many depends on the concrete filesystem. With single indirection one or more of the inode fields point to blocks which do not contain data but other block numbers. Double indirection introduces an extra layer of indirection, and triple indirection goes yet one step further (Figure 12.4).

We will not discuss the general Unix filesystem characteristics in this overview since you will see all the details in the upcoming sections which implement one of its variants (the Minix filesystem).

### 12.2 Mounting: the Unix Way to Access Many Volumes

Some operating systems dedicate a “drive identifier” to each volume that is in use. CP/M was one of the earlier systems with a huge user base, and they used drive letters (A, B) to access more than one floppy drive. The idea was copied by MS-DOS and has been kept alive to this day, with Windows drive letters always starting at C (A and B are still reserved for floppy drives). Figure 12.5 shows how Windows accesses three volumes via three drive letters.

In the Unix world drive letters are unknown. Instead, Unix combines all volumes into one huge Unix directory tree (Figure 12.6). This means that parts of the tree can represent
the contents of several volumes. The process of adding a new volume (and thus enlarging
the tree) is called *mounting*. The root of the newly included filesystem will not appear as
the root of the overall tree, but as some node in the middle of the new tree. That node is
called the *mount point*.

Originally, Unix used regular files as mount points. The idea was to look at the current
directory tree (before the mount operation), then pick a leaf in this tree (regular files
cannot have children in the tree) and attach the root of the new volume to this leaf [RT74].

Modern Unix-type systems use directories as mount points which can have the effect
of hiding files if the directory chosen as a mount point is not empty (i.e., it is not a leaf of
the tree).

A mount operation can be undone: the system can *unmount* a volume which removes
the volume’s sub-tree from the directory tree. This is only possible when there are no
open files on the volume. Unix systems support mounting and unmounting via *mount* and
*umount* system calls (and library functions of the same names), whereas the corresponding
command line tools are called *mount* and *umount* (without the “n” letter of *umount*).

When mounting a volume, it is also possible to provide *mount options* which are specific
to this volume, and special mount options may be available which depend on the filesystem. A classical mount option is *read only* which completely forbids write operations to
Figure 12.5: Windows uses separate drive letters for each volume.

Figure 12.6: Unix systems integrate all volumes in one tree. In this example, /, /mnt/dvd/ and /mnt/disk2/ are mount points.
the volume. This is sometimes required by the device type (e.g. in case of a DVD), and at other times it is just so desired by the user; for example, computer forensic examiners need to mount media in read-only mode so that all data will remain in their original state.

## 12.3 The Ulix Virtual Filesystem

Ulix shall use a virtual filesystem (VFS). That means that several real filesystems might be used, and the VFS provides an abstraction so that generic functions such as `open`, `read`, and `write` may be used for accessing these.

There are several layers, as you can see in Figure 12.7:

- Starting at the lowest level, we need code that can interact with the controllers to which floppy or hard disk drives are connected. They will work with blocks of data, i.e., they read or write a whole block (and not single bytes) each time. Chapter 13 will present the code necessary to talk to the devices and perform the data transfers between memory and disk.

- One level above, we provide generic block read/write functions that will work with any supported device. They take a device ID as a parameter but otherwise let the next level ignore specifics of the device in use.

- Yet higher in the driver hierarchy are the logical filesystem drivers. Ulix only provides one implementation of such a driver (we support the Minix filesystem), but we write the code in such a way that we (or you) could easily add additional drivers. The logical level is concerned with the organization of files, directories and metadata of a volume: this is what this chapter is mainly about. We will give an introduction to all the details of the Minix filesystem design.

- On the highest level there are the virtual file system functions. They work with pathnames for accessing files or directories, and such pathnames will be translated into (device, local pathname) pairs. If you split the mount point from an absolute path, the remainder is the local (absolute) path on the volume which is mounted on that mount point. This is also the code for which we provide a user mode interface via system calls.

Whenever a file is being accessed, we want Ulix to take the following steps:

1. Calculate the absolute path of the file. Note that a filename may already be given as an absolute path (e.g. `/usr/bin/ps`), but it may also be given as a relative path (e.g. `../ps`). In the latter case we construct the absolute path from the current working directory and the relative path.

2. Scan the mount table to find out on which filesystem the file is located. This will return two values: a pointer to the filesystem and a path which is local to this filesystem. In case of `/mnt/tmp/file.txt` this may lead to the number 1 (standing for filesystem number 1 which is mounted on `/mnt`) and the path (`/tmp/file.txt`) within that filesystem.
Operating System

Virtual file system driver

Minix FS driver

FAT driver (not implem.)

NTFS driver (not implem.)

Generic block device driver (disk access)

FDD driver

HDD driver

“serial hard disk” driver

FDD controller

HDD controller

serial port

Figure 12.7: The Virtual Filesystem has a layered design. For accessing files, the kernel and processes use functions at the virtual filesystem level, and they will be translated through the layers until they are finally turned into floppy or hard disk access function calls.

3. Depending on the service function which was called (e.g. open), find a registered function that can talk to this kind of filesystem (e.g. mx_open for a Minix filesystem or fat_open for a DOS/FAT filesystem) and call it.

The filesystem-specific functions should assume that they can access the filesystem as a large, consecutive block of data. As already mentioned, Unix will provide generic functions readblock_{506b} and writeblock_{507c} which can be used to access the raw data, be they on a disk partition, a floppy disk or inside RAM.
We will restrict the number of mounts to 16. A *mount table entry* looks like this:

```c
typedef struct {
    char mountpoint[256];
    short fstype;  // filesystem type, e. g. Minix, device filesystem
    short device;  // e. g. DEV_FD0, DEV_HDA
    short mount_flags;  // always 0; we will not use mount flags
} mount_table_entry;
```

Defines:
- `mount_table_entry`, used in chunk 405b.

We do not provide `mount` and `umount` functions in the kernel, but instead work with a fixed table. This could easily be remedied since those functions would just add or remove an entry to the following array and perform some checks:

```c
mount_table_entry mount_table[16] = {
    { "", FS_MINIX, DEV_HDA, 0 },
    { "/mnt/", FS_MINIX, DEV_FD1, 0 },
    { "/tmp/", FS_MINIX, DEV_HDB, 0 },
    { "/dev/", FS_DEV, DEV_NONE, 0 },
    { { 0 } }
};
short current_mounts = 4;  // how many FSs are mounted?
```

Defines:
- `current_mounts`, used in chunks 406, 408c, and 492.
- `mount_table`, used in chunks 406, 408, and 492.

Uses `DEV_FD1 508a`, `DEV_HDA 508a`, `DEV_HDB 508a`, `DEV_NONE 508a`, `FS_DEV 410a`, `FS_MINIX 410a`, and `mount_table_entry 405a`.

(The constants `DEV_HDA 508a`, `DEV_HDB 508a` and `DEV_FD1 508a` identify the first two hard disks and the second floppy disk; we will define them in the next chapter.) The information in such an entry corresponds roughly to the data you can observe in `/etc/mtab` on a Linux system:

```plaintext
/dev/sda2 / ext3 rw,errors=remount-ro 0 0
```

(This line shows the device filename, the mount point, the filesystem type, the mount options (in this case: read-write, remount as read-only in case of errors) and dump and filesystem check options which we will not deal with in ULIX.)

A mount table entry is unused if its `fstype` element is 0. The mount flag value 0 calls for standard mount options. (In our case that means readable and writeable, though we do not provide alternatives such as read-only.) The following helper function

```c
void print_mount_table();
```

will be called during system startup and show the mount table:
The fs_names array contains strings describing the filesystems; its definition follows on page 410.

We will provide two concrete implementations of filesystems:

- a **Minix filesystem** implementation which describes how to (logically) read and write Minix-formatted media
- and a **/dev filesystem**, similar to Linux, which provides information about known devices (such as /dev/fd0 for the first floppy drive).

The architecture will be such that it is possible to add support for other filesystems, for example FAT (from MS-DOS/Windows). Since Ulxix belongs to the Unix family, we will provide abstract Unix filesystem features (such as symbolic and hard links, user and group information, classical Unix access permissions and timestamps) and have to map them to the data which are available in a concrete filesystem (on a disk).

The other layer is the hardware: It shall be possible to use all supported filesystems on any kind of device for which there are blockread and blockwrite functions. Thus, when Ulxix tries to open a file and read from it, the system will start with executing the virtual u_open or u_read function, then call (for example) Minix-related mx_open or mx_read functions and finally end in calls to the hardware-specific readblock functions. The overall process of executing fd = open("/mnt/tmp/test") with the second floppy drive mounted on /mnt/ is shown in Figure 12.8.
On the hardware side, URIX will provide drivers for floppy disks and hard disks. Thus, there will be support for Minix-formatted floppy disks and hard disks.

### 12.3.1 Finding the Device and Local Path

Let's look at a possible scenario: Assume we have two floppy disks (fda, fdb) and two RAM disks (ram0, ram1) mounted like this:

- /dev/fda on / (minix)
- /dev/fdb on /home (minix)
- /dev/ram0 on /mnt (minix)
- /dev/ram1 on /home/ramtest (minix)
Since the path `/home/ramtest` does not exist before fd0 has been mounted on `/home/`, the second RAM disk\(^1\) must have been mounted *after* the second floppy disk. Thus, if we assume that we store the mount information in the order in which it was created by mount, we can search the mount table backwards, starting with the last entry, and compare each mount point to the leading characters of the absolute path name:

\[
\text{split_mountpoint (} \text{mount_table[mount_entry].mountpoint, path, localpath}\text{)};
\]

\[
\text{if (strlen (localpath) == 0) } \{
\text{// empty string}
\text{localpath[0] = '/'; localpath[1] = 0;}
\}
\]

\[
\text{*dev = mount_table[mount_entry].device;}
\]
\[
\text{*fs = mount_table[mount_entry].fstype;}
\]
\[
\text{return 0;}
\]

¹ This version of U11X does not implement RAM disks; if you want to see U11X RAM disk support, you can read Liviu Beraru’s bachelor thesis [Ber13].
That is really all there is to do. Every call of this function (with a syntactically correct absolute path) must be successful, however that does not mean that the path truly exists: Other functions must check whether the local part of the path (localpath) is available on the device—but we know where to look now.

We need to implement the two helper functions string_starts_with (which is similar to strcmp) and split_mountpoint:

```c
int string_starts_with (char *str, char *prefix) {
    if (strlen (prefix) > strlen (str)) { return false; } // cannot be a sub-string
    while (*prefix != '0') {
        if (*prefix++ != *str++) { return false; } // found different character
    }
    return true; // parsed all of prefix; match!
}
```

Defines: string_starts_with, used in chunks 408c and 409a.
Uses strlen 594a.

The function split_mountpoint expects that the path string does in fact start with mountpoint. It does not check this property but only removes as many characters as necessary:

```c
void split_mountpoint (char *mountpoint, char *path, char *localpath) {
    // input: mountpoint, e.g. /home/
    // path, e.g. /home/user/file.txt
    // output: localpath, e.g. /user/file.txt
    int len = strlen (mountpoint);
    strncpy (localpath, path+len-1, 256);
}
```

Defines: split_mountpoint, used in chunk 408d.
Uses g, strlen 594a, and strncpy 594b.

### 12.3.2 Constants for Filesystems

We declare some constants for the filesystems which are (or might be) supported by Ulix: In most cases we will work with FS_MINIX since we provide a full implementation of the Minix filesystem. The device filesystem FS_DEV is also available, but FS_ERROR and FS_FAT will only cause errors if they occur anywhere. We have included an FS_FAT constant because we use it occasionally for explaining how FAT support could be added to Ulix.
12.3.3 Global File Descriptors

We will allow all filesystem drivers to manage their own sets of file descriptors since they may use them as an index into a private table. Thus, when we open several files on Minix-formatted media, file descriptors 0, 1, 2, 3, etc. will be in use. If another filesystem driver (e.g. one for FAT) exists, it may use the same numbers.

Obviously just passing those file descriptor numbers to the calling process (or kernel function) would create chaos with some numbers being used twice or more often. We avoid this problem by granting each filesystem a range of 256 numbers. For each filesystem `filesys` and a filesystem-local file descriptor `localfd` we calculate the global file descriptor via \( fd = (filesys \ll 8) + localfd \).

Open Minix files will have file descriptors in the range 256–511, and FAT files would have descriptors in the range 512–767. When one of the functions `u_read`, `u_write` etc. is called, it expects a global file descriptor as argument. By reversing the above calculation via \( filesys = fd >> 8; localfd = fd \& 0xff; \) we can easily find out which filesystem function we need to call and which local file descriptor we have to provide it.

So, in order to clear the terminology, here is an overview of the three types of file descriptors we will use:

**Global File Descriptor**: A global file descriptor is used by the kernel to identify a unique open file—across all processes (and the kernel itself).

**Local File Descriptor**: Each subsystem (such as the Minix or /dev subsystem) uses its own set of file descriptors. These are also global in that they are not associated with any specific process, but no generic filesystem function is meant to use them.

**Process File Descriptor**: Each process keeps a list of its own file descriptors (in its thread table entry). Those descriptors are mapped to global file descriptors via those entries. They only make sense when seen from a process’ point of view.
12.3.4 Opening a File

We start with an implementation of the \texttt{u\_open} function which—in case of a file on a Minix filesystem—will call \texttt{mx\_open}. All filesystem related functions on the virtual filesystem layer will have a \texttt{u\_} prefix so that we can distinguish them from the user mode library functions with the same names (e.g. \texttt{u\_open} and \texttt{u\_read} inside the kernel, \texttt{open} and \texttt{read} in the user mode library).

The prototype for \texttt{u\_open} almost follows the Unix standards. We do not allow the optional third argument of the POSIX standard,

\begin{verbatim}
int open (const char *pathname, int flags);
int open (const char *pathname, int flags, mode_t mode);
\end{verbatim}

but instead expect a third parameter \texttt{open\_link} that we will use to decide whether \texttt{u\_open} shall follow symbolic links or just open the link file itself. This option will not be available to the corresponding system call since we do not want processes to manually handle symbolic links.

When creating a file, Ulx always sets the standard access permissions 644; they can later be modified with \texttt{u\_chmod}. Thus the \texttt{u\_open} prototype is:

\begin{verbatim}
(int prototypes 411a)≡
 int u\_open (const char *path, int oflag, int open\_link);
\end{verbatim}

Before we start with the function implementation we note that there will be a recurring pattern in many of the following functions: Many of them start with converting a path argument into an absolute path via \texttt{relpath\_to\_abspath}, and then they look up the device, the filesystem type and the local path via \texttt{get\_dev\_and\_path}. They will always use variables named \texttt{localpath}, \texttt{abspath}, \texttt{device} and \texttt{fs}. Thus, we create two code chunks for the variable declarations and the function calls:

\begin{verbatim}
(VFS functions: declare default variables 411d)≡
char localpath[256], abspath[256];
short device, fs;
\end{verbatim}

\begin{verbatim}
(VFS functions: make absolute path, get device, fs and local path 411e)≡
if (*path !='/')
 relpath_to_abspath (path, abspath);
else
 strcpy (abspath, path, 256);
 get_dev_and_path (abspath, &device, &fs, (char*)&localpath);
\end{verbatim}
void relpath_to_abspath (const char *relpath, char *abspath);

if (strlen (thread_table[current_task].cwd) > 1) {
    // combine cwd and relpath, add "/" in the middle
    strncpy (abspath, thread_table[current_task].cwd, 256);
} else {
    strncpy (abspath, "", 256);
}

strncpy (abspath + strlen (abspath) + 1, relpath, 256 - strlen (abspath) - 1);
abspath[strlen (abspath)] = '/';

int u_open (char *path, int oflag, int open_link) {
    (VFS functions: declare default variables)
    (VFS functions: make absolute path, get device, fs and local path)
    (u_open: handle symlink)

    if (scheduler_is_active) {
        (u_open: check permissions)       // see user/group chapter
    }

    int fd;
    switch (fs) {
        case FS_MINIX:
            fd = mx_open (device, localpath, oflag);
            if (fd == -1) return -1;       // error (opening failed)
            else return (fs << 8) + fd;
        case FS_FAT:
            return -1;                    // not implemented
        case FS_DEV:
            fd = dev_open (localpath, oflag);
            if (fd == -1) return -1;       // error (opening failed)
            else return (fs << 8) + fd;
        case FS_ERROR:
            return -1;                    // error (wrong FS)
        default:
            return -1;                    // error (wrong FS)
    }
}

Defines:
    u_open, used in chunks 190c, 229a, 293b, 411, 420, 426b, 488a, and 582a.
Uses dev_open 495c, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_open 464b,
and scheduler_is_active 276e.
This is all there is to do: the function first looks into the mount table in order to determine on which device the file resides and what filesystem that device is formatted with. Since get_dev_and_path\textsubscript{408c} also calculates the local absolute path within the filesystem, \texttt{u_open}\textsubscript{412c} can immediately call \texttt{mx_open}\textsubscript{464b} or \texttt{dev_open}\textsubscript{495c} which do the real work. If we were to add support for the Linux Ext3 filesystem, we would modify this code by adding a case for \texttt{fs == FS_EXT3}:

\begin{verbatim}
\textbf{adding Ext3 support 413a)≡}
\begin{verbatim}
case FS_EXT3:
    fd = ext3_open (device, localpath, oflag);
    if (fd == -1) return -1; // error (opening failed)
    else return (fs << 8) + fd;
\end{verbatim}
\end{verbatim}

Note that we need not deal with special cases such as non-existing files or files which do not have the necessary access permissions in this code: This will be handled in the filesystem-specific functions, such as \texttt{mx_open}\textsubscript{464b}. If these return an error code (instead of a file descriptor), the result is just forwarded to the caller of \texttt{u_open}\textsubscript{412c}.

In case that the file to open is a symbolic link, we need to follow the link. We do this by reading the link file and using the path found there:

\begin{verbatim}
\textbf{\langle u_open: handle symlink 413b⟩≡}
\begin{verbatim}
struct stat st;
char link[256];
u_stat (abspath, &st);
if ( ((st.st_mode & S_IFLNK) == S_IFLNK) && (open_link == FOLLOW_LINK)) {
    // open (how?), read_, then u_open (symlink)
    int link_fd = u_open (abspath, O_RDONLY, DONT_FOLLOW_LINK); // open link file
    u_read (link_fd, link, 256);
    u_close (link_fd);
    return u_open (link, oflag, FOLLOW_LINK); // recursion
}
\end{verbatim}
\end{verbatim}

This calls \texttt{u_open}\textsubscript{412c} recursively, and our simple function does not check the recursion level. Thus a simple sequence of \texttt{ln -s xyz xyz} and \texttt{cat xyz} in some directory will force \texttt{u_open}\textsubscript{412c} into an infinite recursion (and crash the system when the kernel stack exceeds its boundary).

To see this mechanism in operation, consider that there is a file \texttt{file} and we have two symbolic links, \texttt{a} and \texttt{b}, where \texttt{a} points to \texttt{b} and \texttt{b} points to \texttt{file}. Trying to open \texttt{a} will cause the following recursion:

\begin{verbatim}
\textbf{\langle example for recursive u_open calls 413c⟩≡}
\begin{verbatim}
\texttt{u_open ("/a", oflag, FOLLOW_LINK) // open "a", called from somewhere
st.st_mode == S_IFLINK // "a" is a symlink
u_open ("/a", O_RDONLY, DONT_FOLLOW_LINK) // open it read-only and...
u_read () -> "/b" // read the contents: it's "b"
u_close ("/a") // close "a"
\texttt{u_open ("/b", oflag, FOLLOW_LINK) // open "b" (return its retval)
st.st_mode == S_IFLINK // "b" is also a symlink
u_open ("/b", O_RDONLY, DONT_FOLLOW_LINK) // open it RO and...
u_read () -> "/file" // read the contents: it's "file"
\end{verbatim}
\end{verbatim}
12.3.5 Reading, Writing and Other Operations

The operations on open files are even simpler because we don’t have to find out what filesystem to use: we get that information via the global file descriptor. For example, the `u_read` function extracts the filesystem type and the local file descriptor from the global file descriptor (just like we have already described earlier); then it branches and calls one of the `*_read` functions.

```c
int u_read (int fd, void *buf, int nbyte);  // file descriptor
int u_write (int fd, void *buf, int nbyte);  // file descriptor
int u_close (int fd);                     // file descriptor
int u_lseek (int fd, int offset, int whence);  // file descriptor
int u_unlink (const char *path);           // file descriptor
int u_link (const char *path, const char *path2);  // file descriptor
int u_symlink (const char *path, const char *path2);  // file descriptor
int u_truncate (const char *path, int length);  // file descriptor
int u_ftruncate (int fd, int length);         // file descriptor
int u_readlink (char *path, char *restrict buf, size_t bufsize);  // file descriptor
```

The `u_read` function simply requires the following code:

```c
int u_read (int fd, void *buf, int nbyte) {
  if (fd < -100) {  // file descriptor
    return -1;     // file descriptor
  }
  switch (fs) {
    case FS_MINIX: return mx_read (localfd, buf, nbyte);  // file descriptor
    case FS_FAT: return -1;  // file descriptor
    case FS_DEV: return dev_read (localfd, buf, nbyte);  // file descriptor
    case FS_ERROR: return -1;  // file descriptor
    default: return -1;  // file descriptor
  }
}
```

Defines:
- `u_read`, used in chunks 190c, 229a, 233b, 294, 413c, 420c, 426b, and 582a.
- `dev_read` 496d, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, and mx_read 470b.

Uses `dev_read` 496d, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, and mx_read 470b.

```c
if (fd < 0) return -1;  // file descriptor
int fs = fd >> 8;  // file descriptor
int localfd = fd & 0xff;  // file descriptor
```
(Again we create a separate code chunk for this check and the calculation which turns a global file descriptor into a (filesystem, local file descriptor) pair since we will reuse it several times.)

Negative file descriptors with \( fd < -100 \) deal with the special cases of standard input/output and pipes (though pipes have not been implemented in this version); we will describe that case soon. The code for writing is almost identical to that of \( u\_read_{414b} \):

\[
\begin{enumerate}
\item \textit{function implementations}_{100b} \equiv \text{\textbackslash function implementations}_{414a} \text{\textbackslash function implementations}_{426b} \equiv \]
\[
\begin{aligned}
&\text{int } u\_write (\text{int } fd, \text{void } *buf, \text{int } nbyte) \{} \\
&\quad \text{if } (fd < -100) \{ \langle \text{write: standard I/O and pipes}_{417} \rangle \} \\
&\quad \langle \text{VFS functions: turn } fd \text{ into } (fs, \text{localfd}) \text{ pair or fail}_{414c} \rangle \\
&\quad \text{switch (fs) } \{ \\
&\quad\quad \text{case FS\_MINIX: return mx\_write } (\text{localfd, buf, nbyte}); \\
&\quad\quad \text{case FS\_FAT: return -1; } \langle \text{not implemented}_{414d} \rangle \\
&\quad\quad \text{case FS\_DEV: return dev\_write } (\text{localfd, buf, nbyte}); \\
&\quad\quad \text{case FS\_ERROR: return -1; } \langle \text{error}_{414e} \rangle \\
&\quad\quad \text{default: return -1; } \rangle \\
&\} \\
\end{aligned}
\]

\[
\begin{aligned}
\text{Defines: } \\
&u\_write, \text{used in chunks 293d, } 414a, \text{and } 426b. \\
\text{Uses dev\_write}_{497}, \text{FS\_DEV}_{410a}, \text{FS\_ERROR}_{410a}, \text{FS\_FAT}_{410a}, \text{FS\_MINIX}_{410a}, \text{and } \text{mx\_write}_{474c}. \\
\end{aligned}
\]

We define the file descriptors for stdin, stdout and stderr (as they are valid inside processes) and kernel-internal negative values for the three standard I/O streams:

\[
\begin{aligned}
&\langle \text{public constants}_{46a} \rangle \equiv \text{\textbackslash define } STDIN\_FILENO \text{ 0} \\
&\langle \text{define } STDIN\_FILENO \text{ 0} \rangle \equiv \text{\textbackslash define } STDOUT\_FILENO \text{ 1} \\
&\langle \text{define } STDOUT\_FILENO \text{ 1} \rangle \equiv \text{\textbackslash define } STDERR\_FILENO \text{ 2} \\
&\langle \text{define } STDERR\_FILENO \text{ 2} \rangle \equiv \\
&\langle \text{constants}_{112a} \rangle \equiv \text{\textbackslash define } DEV\_STDIN \text{ (-101)} \\
&\langle \text{define } DEV\_STDIN \text{ (-101)} \rangle \equiv \text{\textbackslash define } DEV\_STDOUT \text{ (-102)} \\
&\langle \text{define } DEV\_STDOUT \text{ (-102)} \rangle \equiv \text{\textbackslash define } DEV\_STDERR \text{ (-103)} \\
&\langle \text{define } DEV\_STDERR \text{ (-103)} \rangle \equiv \\
&\langle \text{define } dev\_write, \text{used in chunks 190a, } 416d, 417, \text{and } 421b. \\
&\text{DEV\_STDIN, used in chunks 190a, } 416d, 417, \text{and } 421b. \\
&\text{DEV\_STDOUT, used in chunks 190a, } 416d, 417, \text{and } 421b. \\
\end{aligned}
\]

In order to read from standard input, we make a system call which in turn will execute \texttt{syscall\_readchar}_{416b}, our function for reading from the keyboard:

\[
\begin{aligned}
&\langle \text{unix system calls}_{206e} \rangle \equiv \text{\textbackslash define } \_NR\_readchar \text{ 525} \\
&\langle \text{define } \_NR\_readchar \text{ 525} \rangle \equiv \\
&\langle \text{define } \_NR\_readchar, \text{used in chunk 416c} \rangle \\
\end{aligned}
\]
The implementation lets the current process block if no new character is available in the keyboard buffer:

```c
void syscall_readchar (context_t *r) {
    char c;
    int t = thread_table[current_task].terminal;
    terminal_t *term = &terminals[t];

    // get character, return 0 if there is no new character in the buffer
    if (term->kbd_count > 0) {
        term->kbd_count--;
        term->kbd_lastread = (term->kbd_lastread+1) % SYSTEM_KBD_BUFLEN;
        c = term->kbd[term->kbd_lastread];
    } else {
        c = 0;
        if ((current_task > 1) && scheduler_is_active) {
            block (&keyboard_queue, TSTATE_WAITKEY);
        }
    }
}
```

Defines:
- `syscall_readchar`, used in chunk 416.

Reading from standard output or standard error is not allowed and causes an error. Also there are no pipes in this Unix version, but the idea is to let the kernel create an internal buffer for each pipe and associate two negative file descriptors with it.
__asm__ ("\n.proiq: 
.intel_syntax noprefix; 
mov eax, 525; 
int 0x80; 
mov %0, ebx; 
.att_syntax"
 ): 
"=r"(u)
);
c = (byte) u;
((byte*) buf)[i] = c;
break;
case DEV_STDOUT:
case DEV_STDERR:
printf "((ERROR: reading from stdout or stderr)\n"
return (-1); // error, cannot read from output

default: return (-1); // pipes not implemented yet
}
Uses DEV_STDERR 415c, DEV_STDIN 415c, DEV_STDOUT 415c, and printf 601a.

Similarly, writing to standard output or standard error dumps data on the terminal using the kputch function, whereas writing to standard input (or pipes) is forbidden:

\langle write: standard I/O and pipes 417\rangle 

byte c;
switch (fd) {
case DEV_STDIN:
printf "((ERROR: writing to stdin)\n"
return (-1); // error, cannot write to input

case DEV_STDOUT:
case DEV_STDERR:
for (int i = 0; i < nbyte; i++) {
c = ((char*)buf)[i];
if (c > 31 || c == '\n' || c == 0x08) {
kputch (c); // regular characters: 32..255, \n, \b
} else {
kputch ('^'); kputch (c+64); // control characters: <32
}
}
default: return (-1); // pipes not implemented yet
}
Uses DEV_STDERR 415c, DEV_STDIN 415c, DEV_STDOUT 415c, kputch 335b, and printf 601a.

Closing an open file or performing a seek operation work like reading, in that u_close and u_lseek simply call the corresponding mx_* or dev_* functions:
int u_close (int fd) {
    (VFS functions: turn fd into (fs, localfd) pair or fail 414c)
    switch (fs) {
        case FS_MINIX: return mx_close (localfd);
        case FS_FAT: return -1; // not implemented
        case FS_DEV: return dev_close (localfd);
        case FS_ERROR: return -1; // error
        default: return -1;
    }
}

int u_lseek (int fd, int offset, int whence) {
    (VFS functions: turn fd into (fs, localfd) pair or fail 414c)
    switch (fs) {
        case FS_MINIX: return mx_lseek (localfd, offset, whence);
        case FS_FAT: return -1; // not implemented
        case FS_DEV: return dev_lseek (localfd, offset, whence);
        case FS_ERROR: return -1; // error
        default: return -1;
    }
}

The implementation of u_unlink is similar to that of u_open since in both cases we get a pathname as argument:

int u_unlink (const char *path) {
    (VFS functions: declare default variables 411d)
    (VFS functions: make absolute path, get device, fs and local path 411e)
    switch (fs) {
        case FS_MINIX: return mx_unlink (device, localpath);
        case FS_FAT: return -1; // not implemented
        case FS_DEV: return -1; // no unlink support in device FS
        case FS_ERROR: return -1; // error
        default: return -1;
    }
}

On the other hand, creating a new hard link with u_link requires some extra work: The function takes two pathnames, one of which is for a name that does not yet exist. It must check that both reside on the same volume because hard links cannot cross volumes.
The rest is similar, but we work without the switch statement since hard links only exist on Unix filesystems (of which Unix only supports Minix):

\[\text{(function implementations 100b)}\]  
\[
\text{int u\_link (const char *path, const char *path2) \{}
\]
\[
\text{char localpath[256], abspath[256]; short device, fs;}
\]
\[
\text{char localpath2[256], abspath2[256]; short device2, fs2;}
\]
\[
\text{char dir2[256], base2[256], localdir2[256];}
\]
\[
\text{if (*path != '/') relpath\_to\_abspath (path, abspath); \quad \text{ // source}}
\]
\[
\text{else \quad \text{strncpy (abspath, &device, &fs, (char*)&localpath);}}
\]
\[
\text{if (*path2 != '/') relpath\_to\_abspath (path2, abspath2); \quad \text{ // target}}
\]
\[
\text{else \quad \text{strncpy (abspath2, path2, 256);}}
\]
\[
\text{splitpath (abspath2, dir2, base2); \quad \text{ // get dirname}}
\]
\[
\text{get_dev_and_path (dir2, &device2, &fs2, (char*)&localdir2);}
\]
\[
\text{if (device != device2) return -1; \quad \text{ // error: link across volumes}}
\]
\[
\text{if (fs != FS\_MINIX) return -1; \quad \text{ // error: not Minix}}
\]
\[
\text{strncpy (localpath2, localdir2, 256); \quad \text{ // localpath2 = localdir2}}
\]
\[
\text{int len = strlen(localpath2);} \quad \text{ // special case "/"}
\]
\[
\text{localpath2[len] = '/'; \quad \text{ // localpath2 += "/"}}
\]
\[
\text{strncpy (localpath2 + len + 1, base2, 256); \quad \text{ // localpath2 += base2}}
\]
\[
\text{return mx\_link (device, localpath, localpath2);}
\]
\[
\text{}}
\]
\[
\text{Defines:}
\]
\[
\text{u\_link, used in chunk 426b.}
\]
\[
\text{Uses dirname 455b, FS\_MINIX 410a, get\_dev\_and\_path 408c, mx\_link 480a, relpath\_to\_abspath 412b, splitpath 455a, strlen 594a, and strncpy 594b.}
\]

The \text{u\_symlink} function only checks the target since it is allowed to write invalid paths (and paths to different volumes) into a symbolic link:

\[\text{(function implementations 100b)}\]  
\[
\text{int u\_symlink (const char *path, const char *path2) \{}
\]
\[
\text{char localpath2[256], abspath2[256]; short device2, fs2;}
\]
\[
\text{if (*path2 != '/') relpath\_to\_abspath (path2, abspath2); \quad \text{ // target}}
\]
\[
\text{else \quad \text{strncpy (abspath2, path2, 256);}}
\]
\[
\text{get_dev_and_path (abspath2, &device2, &fs2, (char*)&localpath2);}
\]
\[
\text{if (fs2 != FS\_MINIX) return -1; \quad \text{ // error: not Minix}}
\]
\[
\text{return mx\_symlink (device2, (char*)path, localpath2);}
\]
\[
\text{}}
\]
\[
\text{Defines:}
\]
\[
\text{u\_symlink, used in chunk 426b.}
\]
\[
\text{Uses FS\_MINIX 410a, get\_dev\_and\_path 408c, mx\_symlink 484b, relpath\_to\_abspath 412b, and strncpy 594b.}
\]

For truncating a file we only need to think about the \text{u\_ftruncate} variant which works on an open file; the other function can just open the file and call \text{u\_ftruncate}:

\[\text{(function implementations 100b)}\]  
\[
\text{int u\_ftruncate (const char *path, const char *path2) \{}
\]
\[
\text{char localpath2[256], abspath2[256]; short device2, fs2;}
\]
\[
\text{if (*path2 != '/') relpath\_to\_abspath (path2, abspath2); \quad \text{ // target}}
\]
\[
\text{else \quad \text{strncpy (abspath2, path2, 256);}}
\]
\[
\text{get_dev_and_path (abspath2, &device2, &fs2, (char*)&localpath2);}
\]
\[
\text{if (fs2 != FS\_MINIX) return -1; \quad \text{ // error: not Minix}}
\]
\[
\text{return mx\_ftruncate (device2, (char*)path, localpath2);}
\]
\[
\text{}}
\]
12.3.6 Detect Terminals

Sometimes a process needs to find out whether it is reading from or writing to a file or one of the standard I/O streams. For that purpose it can call the user mode function \texttt{isatty}.
12.3 The UliX Virtual Filesystem

which returns true if the file descriptor is connected to a terminal. In case of files or pipes it returns false. In order to implement the kernel function

\[\text{boolean } \text{u_isatty}(\text{int } \text{fd});\]

we simply check whether the file descriptor is DEV_STDIN, DEVSTDOUT, or DEV_STDERR:

\[\text{boolean } \text{u_isatty}(\text{int } \text{fd}) \{\]
\[\text{return } ((\text{fd} == \text{DEV_STDIN}) \| (\text{fd} == \text{DEVSTDOUT}) \| (\text{fd} == \text{DEV_STDERR}));\]
\]

Uses DEV_STDERR, DEV_STDIN, and DEV_STDOUT.

12.3.7 Status

The u_stat function fills a struct stat entry with the status information about a file which can be queried with mx_stat or dev_stat:

\[\text{int } \text{u_stat}(\text{const char } \text{*path}, \text{struct stat } \text{*buf});\]

\[\text{int } \text{u_stat}(\text{const char } \text{*path}, \text{struct stat } \text{*buf}) \{\]
\[\langle \text{VFS functions: declare default variables}\rangle\]
\[\langle \text{VFS functions: make absolute path, get device, fs and local path}\rangle\]
\[\text{switch } (\text{fs}) {\]
\[\text{case } \text{FS_MINIX}: \text{return } \text{mx_stat}(\text{device}, \text{localpath}, \text{buf});\]
\[\text{case } \text{FS_FAT}: \text{return } -1; \quad \text{// not implemented}\]
\[\text{case } \text{FS_DEV}: \text{return } \text{dev_stat}(\text{localpath}, \text{buf});\]
\[\text{case } \text{FS_ERROR}: \text{return } -1; \quad \text{// error}\]
\[\text{default: } \text{return } -1;\]
\[\}
\]

Defines:
- u_stat, used in chunks 413b, 420c, 421c, 426b, 432e, 576, and 577.
- dev_stat 499d, FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_stat 490a, and stat 429b 489b.

12.3.8 Directories

We also need to handle directories: it is possible to create and remove them via the

\[\text{int } \text{u_mkdir}(\text{const char } \text{*path}, \text{int } \text{mode});\]
\[\text{int } \text{u_rmdir}(\text{const char } \text{*path});\]

functions and to read their entries via u_getdent. The u_mkdir and u_rmdir implementations are just rewrites of u_open and u_unlink:
int u_mkdir (const char *path, int mode) {
  (VFS functions: declare default variables 411d)
  (VFS functions: make absolute path, get device, fs and local path 411e)
  switch (fs) {
    case FS_MINIX: return mx_mkdir (device, localpath, mode);
    case FS_FAT: return -1; // not implemented
    case FS_DEV: return -1; // not allowed
    case FS_ERROR: return -1; // error
    default: return -1;
  }
}

int u_rmdir (const char *path) {
  (VFS functions: declare default variables 411d)
  (VFS functions: make absolute path, get device, fs and local path 411e)
  switch (fs) {
    case FS_MINIX: return mx_rmdir (device, abspath, localpath);
      // two path args
    case FS_FAT: return -1; // not implemented
    case FS_DEV: return -1; // no rmdir support in device FS
    case FS_ERROR: return -1; // error
    default: return -1;
  }
}

int u_getdent (const char *path, int index, struct dir_entry *buf) {
  (VFS functions: declare default variables 411d)
  (VFS functions: make absolute path, get device, fs and local path 411e)
  switch (fs) {
    case FS_MINIX: return mx_getdent (device, localpath, index, buf);
    case FS_FAT: return -1; // not implemented
    case FS_DEV: return dev_getdent (localpath, index, buf);
    case FS_ERROR: return -1; // error
    default: return -1;
  }
}

Defines:
  u_mkdir, used in chunk 426b.
  u_rmdir, used in chunks 421e and 426b.
Uses FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_mkdir 487a, and mx_rmdir 488a.

For reading a directory we provide the

int u_getdent (const char *path, int index, struct dir_entry *buf);

function which reads single entries and writes them into a struct dir_entry buffer:
12.4 System Calls for File Access

We have already discussed the three types of file descriptors which are used in Unix. All the functions of the virtual filesystem layer which you have seen so far use global file descriptors which uniquely identify an open file across all processes. But this information should be hidden from individual processes—after all, operating systems are all about abstraction, and from a process’ point of view only its own open files are relevant, thus a process can expect that its internal file descriptor numbers do not depend on the activities of other processes (or the kernel). Also, we want to support the Unix tradition of reserving file descriptor numbers 0, 1 and 2 for the standard I/O streams.

Figure 12.9: Relationship between the various file descriptors.
The u_* functions in the kernel use global file descriptors which do not recognize process affiliation. Since we do not want to export these numbers to the processes, we must convert them to process file descriptors.

We provide the following two functions gfd2pfd and pfd2gfd which convert global to process and process to global file descriptors. Figure 12.9 shows the relationship between process, local and global file descriptors.

Processes may open up to 16 files; we add a new field to the TCB which stores that many file descriptors.

If we start with a process file descriptor, the translation is simpler: we just look it up in the process' file descriptor table.

Turning a global file descriptor into a process one is a bit more complicated: if there is no mapping (yet), we need to find a free place in the process' descriptor list files. But we start with searching for a mapping:

```c
int pfd2gfd (int pfd) {
    int gfd;
    if (pfd == -1) return -1;
    thread_id pid = thread_table[current_task].pid;
    if (pfd >= 0 && pfd < MAX_PFD)
        return thread_table[pid].files[pfd];
    else return -1;
}
```

Uses current_task 192c, MAX_PFD 424b, pfd2gfd, thread_id 178a, and thread_table 176b.

(Note that thread_table[176b][current_task[192c].files[pfd] may also be -1 which is the standard value for an unused local file descriptor.)

```c
int gfd2pfd (int gfd) {
    int pfd;
    if (gfd == -1) return -1;
    thread_id pid = thread_table[current_task].pid;
    for (pfd = 0; pfd < MAX_PFD; pfd++) {
        if (thread_table[pid].files[pfd] == gfd)
            return pfd;
    }
    return -1;
}
```
// found none, create it
for (pfd = 0; pfd < MAX_PFD; pfd++) {
    if (thread_table[pid].files[pfd] == -1) {
        thread_table[pid].files[pfd] = gfd;
        return pfd;
    }
}
// no free entry
return -1; // error
}

Uses current_task 192c, gfd2pfd, MAX_PFD 424b, thread_id 178a, and thread_table 176b.

We need to copy file descriptors when we create a new process: The following chunk completes the TCB initialization in the u_fork function.

(u_fork: copy the file descriptors 425a)≡
  if (t_new->pid != t_old->pid) {
    for (int pfd = 0; pfd < MAX_PFD; pfd++) {
      int gfd = t_old->files[pfd];
      if (gfd ≥ 0)
        t_new->files[pfd] = u_reopen (gfd); // get new gfd
      else
        t_new->files[pfd] = gfd;          // use old gfd (stdio)
    }
  }

Uses MAX_PFD 424b, t_new 276c, t_old 276c, and u_reopen.

The function u_reopen creates a copy of a file descriptor. We cannot simply give the newly forked process access to the same (globally visible) file descriptor because the current read/write position in the file is associated with that descriptor. If one of the two processes changes that position, the modification must not be visible in the other process.

(function prototypes 45a)≡
  int u_reopen (int fd);

(function implementations 100b)≡
  int u_reopen (int fd) {
    (VFS functions: turn fd into (fs, localfd) pair or fail 414c)
    switch (fs) {
      case FS_MINIX: return (fs << 8) + mx_reopen (localfd);
      case FS_FAT:    return -1;               // not implemented
      case FS_DEV:    return -1;
      case FS_ERROR:  return -1;               // error
      default:        return -1;
    }
  }

Uses FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, mx_reopen 468b, and u_reopen.

As usual, the real work is done my mx_reopen 468b. (There is no such function for the device filesystem.)
We can now implement the system calls:

We can now implement the system calls:

```c
void syscall_open (context_t *r);
void syscall_stat (context_t *r);
void syscall_close (context_t *r);
void syscall_read (context_t *r);
void syscall_write (context_t *r);
void syscall_lseek (context_t *r);
void syscall_isatty (context_t *r);
void syscall_mkdir (context_t *r);
void syscall_rmdir (context_t *r);
void syscall_getdent (context_t *r);
void syscall_truncate (context_t *r);
void syscall_ftruncate (context_t *r);
void syscall_link (context_t *r);
void syscall_unlink (context_t *r);
void syscall_symlink (context_t *r);
void syscall_readlink (context_t *r);
```

For syscall_getdent we will use the __NR.readdir syscall number but it should be noted that readdir accesses directory entries differently (and U_{IX} does not implement readdir).

```c
void syscall_open (context_t *r) {
    eax_return ( gfd2pfd (u_open ((char*) r->ebx, r->ecx, 0)) );
}

void syscall_stat (context_t *r) {
    eax_return ( u_stat ((char*) r->ebx, (struct stat*) r->ecx) );
}

void syscall_getdent (context_t *r) {
    // ebx: path, ecx: index, edx: dir_entry buffer
    eax_return ( u_getdent ((char*) r->ebx, r->ecx, (struct dir_entry*) r->edx) );
}

void syscall_close (context_t *r) {
    // ebx: fd
    int pfd = r->ebx;
    thread_id pid = thread_table[current_task].pid;
    r->eax = u_close (pfd2gfd (pfd)); // close (globally)
    if (pfd ≥ 0 & pfd < MAX_PFD)
        thread_table[pid].files[pfd] = -1; // close (locally)
}

void syscall_read (context_t *r) {
    // ebx: fd, ecx: *buf, edx: nbytes
    eax_return ( u_read (pfd2gfd (r->ebx), (byte*) r->ecx, r->edx) );
}

void syscall_write (context_t *r) {
    // ebx: fd, ecx: *buf, edx: nbytes
    eax_return ( u_write (pfd2gfd (r->ebx), (byte*) r->ecx, r->edx) );
}
```
void syscall_lseek (context_t *r) {
    // ebx: fd, ecx: offset, edx: whence
    eax_return ( u_lseek (pf2gfd (r->ebx), r->ecx, r->edx) );
}

void syscall_isatty (context_t *r) {
    // ebx: file descriptor
    eax_return ( pf2gfd (u_isatty (r->ebx)) );
}

void syscall_mkdir (context_t *r) {
    // ebx: name of new directory, ecx: mode
    eax_return ( u_mkdir ((char*)r->ebx, r->ecx) );
}

void syscall_rmdir (context_t *r) {
    // ebx: name of directory that we want to delete
    eax_return ( u_rmdir ((char*)r->ebx) );
}

void syscall_truncate (context_t *r) {
    // ebx: filename, ecx: length
    eax_return ( u_truncate ((char*)r->ebx, r->ecx) );
}

void syscall_ftruncate (context_t *r) {
    // ebx: file descriptor, ecx: length
    eax_return ( u_ftruncate (pf2gfd (r->ebx), r->ecx) );
}

void syscall_link (context_t *r) {
    // ebx: original name, ecx: new name
    eax_return ( u_link ((char*)r->ebx, (char*)r->ecx) );
}

void syscall_unlink (context_t *r) {
    // ebx: pathname
    eax_return ( u_unlink ((char*)r->ebx) );
}

void syscall_symlink (context_t *r) {
    // ebx: target file name, ecx: symbolic link name
    eax_return ( u_symlink ((char*)r->ebx, (char*)r->ecx) );
}

void syscall_readlink (context_t *r) {
    // ebx: file name
    // ecx: buffer for link target
    // edx: buffer length
    eax_return ( u_readlink ((char*)r->ebx, (char*)r->ecx, r->edx) );
}

Defines:
syscall_close, used in chunk 428a.
syscall_ftruncate, used in chunk 428a.
syscall_isatty, used in chunk 428a.
syscall_link, used in chunk 428a.
syscall_lseek, used in chunk 428a.
syscall_mkdir, used in chunk 428a.
defines: ulix system calls

As a last step we create syscall table entries for the new system call handlers:

\[\text{initialize syscalls} 173d\] +\= (44b) <416c 434a>
\[
\begin{align*}
    \text{install syscall handler} & (\_NR\_open, \text{ syscall\_open})\; ; \\
    \text{install syscall handler} & (\_NR\_stat, \text{ syscall\_stat})\; ; \\
    \text{install syscall handler} & (\_NR\_close, \text{ syscall\_close})\; ; \\
    \text{install syscall handler} & (\_NR\_read, \text{ syscall\_read})\; ; \\
    \text{install syscall handler} & (\_NR\_write, \text{ syscall\_write})\; ; \\
    \text{install syscall handler} & (\_NR\_lseek, \text{ syscall\_lseek})\; ; \\
    \text{install syscall handler} & (\_NR\_isatty, \text{ syscall\_isatty})\; ; \\
    \text{install syscall handler} & (\_NR\_mkdir, \text{ syscall\_mkdir})\; ; \\
    \text{install syscall handler} & (\_NR\_rmdir, \text{ syscall\_rmdir})\; ; \\
    \text{install syscall handler} & (\_NR\_readdir, \text{ syscall\_getdent})\; ; \\
    \text{install syscall handler} & (\_NR\_truncate, \text{ syscall\_truncate})\; ; \\
    \text{install syscall handler} & (\_NR\_ftruncate, \text{ syscall\_ftruncate})\; ; \\
    \text{install syscall handler} & (\_NR\_link, \text{ syscall\_link})\; ; \\
    \text{install syscall handler} & (\_NR\_unlink, \text{ syscall\_unlink})\; ; \\
    \text{install syscall handler} & (\_NR\_symlink, \text{ syscall\_symlink})\; ; \\
    \text{install syscall handler} & (\_NR\_readlink, \text{ syscall\_readlink})\; ; \\
\end{align*}
\]

Uses \_NR\_close 204c, \_NR\_ftruncate, \_NR\_isatty 428b, \_NR\_link 204c, \_NR\_lseek 204c, \_NR\_mkdir 204c, \_NR\_open 204c, \_NR\_read 204c, \_NR\_readdir 204c, \_NR\_readdir 204c, \_NR\_rmdir 204c, \_NR\_stat 204c, \_NR\_symlink 204c, \_NR\_ftruncate 204c, \_NR\_isatty 204c, \_NR\_write 204c, install\_syscall\_handler 201b, syscall\_close 426b, syscall\_ftruncate 426b, syscall\_getdent, syscall\_isatty 426b, syscall\_link 426b, syscall\_lseek 426b, syscall\_mkdir 426b, syscall\_open 426b, syscall\_read 426b, syscall\_readlink 426b, syscall\_rmdir 426b, syscall\_stat, syscall\_symlink 426b, syscall\_truncate 426b, syscall\_unlink 426b, and syscall\_write 426b.

We need a system call number for \_NR\_isatty 426b:

\[\text{ulix system calls} 206e\] +\= (205a) <415d 493c>
\[
\begin{align*}
    \#define \_NR\_isatty 521 \\
\end{align*}
\]

Defines:

\_NR\_isatty, used in chunks 428a and 429b.
12.4 System Calls for File Access

12.4.1 Library Functions

Here we define the user mode library functions for our collection of new system calls:

\[\text{ulixlib function prototypes} \text{174c} \cdots \text{429a}\]

- `int open` (const char *path, int oflag, ...);
- `int stat` (const char *path, struct stat *buf);
- `int close` (int fildes);
- `int read` (int fildes, void *buf, size_t nbyte);
- `int write` (int fildes, const void *buf, size_t nbyte);
- `int lseek` (int fildes, int offset, int whence);
- `boolean isatty` (int fd);
- `int mkdir` (const char *path, int mode);
- `int rmdir` (const char *path);
- `int getdent` (const char *path, int index, struct dir_entry *buf);
- `int ftruncate` (int fd, int length);
- `int truncate` (const char *path, int length);
- `int link` (const char *path1, const char *path2);
- `int unlink` (const char *path);
- `int symlink` (const char *path1, const char *path2);
- `int readlink` (char *path, char *buf, int bufsize);

\[\text{ulixlib function implementations} \text{174d} \cdots \text{429b}\]

- `int open` (const char *path, int oflag, ...) {
  return syscall3 (__NR_open, (uint)path, oflag); }
- `int stat` (const char *path, struct stat *buf) {
  return syscall3 (__NR_stat, (uint)path, (uint)buf); }
- `int close` (int fildes) { return syscall2 (__NR_close, fildes); }
- `int read` (int fd, void *buf, size_t nbyte) {
  return syscall4 (__NR_read, fd, (uint)buf, nbyte); }
- `int write` (int fd, const void *buf, size_t nbyte) {
  return syscall4 (__NR_write, fd, (uint)buf, nbyte); }
- `int lseek` (int fildes, int offset, int whence) {
  return syscall4 (__NR_lseek, fildes, offset, whence); }
- `boolean isatty` (int fd) { return syscall2 (__NR_isatty, fd); }
- `int mkdir` (const char *path, int mode) {
  return syscall3 (__NR_mkdir, (uint)path, mode); }
- `int rmdir` (const char *path) {
  return syscall2 (__NR_rmdir, (uint)path); }
- `int getdent` (const char *path, int index, struct dir_entry *buf) {
  return syscall4 (__NR_readdir, (uint)path, index, (uint)buf); }
int ftruncate (int fd, int length) {
    return syscall3 (__NR_ftruncate, fd, length); }

int truncate (const char *path, int length) {
    return syscall3 (__NR_truncate, (uint)path, length); }

int link (const char *path1, const char *path2) {
    return syscall3 (__NR_link, (uint)path1, (uint)path2); }

int unlink (const char *path) {
    return syscall2 (__NR_unlink, (unsigned int)path); }

int symlink (const char *path1, const char *path2) {
    return syscall3 (__NR_symlink, (uint)path1, (uint)path2); }

int readlink (char *path, char *buf, int bufsize) {
    return syscall4 (__NR_readlink, (uint)path, (uint)buf, bufsize); }

Defines:
close, used in chunks 467b and 585b.
lseek, used in chunk 498a.
mkdir, used in chunk 618.
open, used in chunks 411a, 414c, 467b, 475a, and 585b.
read, used in chunks 294, 431, 432b, 456, 475a, 477b, 490a, 503, 543a, 552c, and 585b.
stat, used in chunks 420c, 421d, 426b, 432e, 489, 490, 499, 576, 577c, and 608a.
write, used in chunks 35b, 123b, 170c, 199, 204, 213b, 429a, 431, 460b, 475a, 525a, 528a, 539c, 543a, 575, 598c, and 624.

Uses __NR_close 204c, __NR_ftruncate, __NR_isatty 428b, __NR_link 204c, __NR_lseek 204c, __NR_mkdir 204c, __NR_open 204c, __NR_read 204c, __NR_readdir 204c, __NR_readlink 204c, __NR_rmdir 204c, __NR_stat 204c, __NR_symlink 204c, __NR_truncate 204c, __NR_unlink 204c, __NR_write 204c, dir_entry 490b, size_t 46b, syscall2 203c, syscall3 203c, and syscall4 203b.

12.4.2  Reading from Standard Input

The functions

\[ \text{int ureadline (char *s, int maxlength, boolean echo)}; \]
\[ \text{byte ureadchar ()}; \]

use read29b, with the standard input file descriptor STDIN_FILENO415b, to read one or more characters. Writing to standard output will be handled by ulixlib_printchar598c which just write29b, s to the standard output (via file descriptor STDOUT_FILENO415b) and is implemented where we discuss the printf601a function.

ureadline431 takes three arguments: a buffer, a maximum length and an echo flag. If the length parameter is negative then pressing [Enter] to complete the input will not cause the newline character to be displayed. If echo is not set, output will be disabled completely which is useful for password queries: The /bin/login and /bin/su programs use that feature.
int ureadline (char *s, int maxlength, boolean echo) {
   // if maxlength is negative, dont print \n at the end
   char print_newline = 1;
   if (maxlength < 0) {
      print_newline = 0;
      maxlength = -maxlength;
   }
   int pos=0;
   for (;;) {
      startlabel:
      if (pos < 0) { printf ("ERROR: pos < 0\n") ; return; }
      byte c = 0;
      int nbytes = read (STDIN_FILENO, &c, 1); // read one char. from stdin
      if (nbytes == 0) return -1;
      if (c == 0 || c == 27 || c > 190) // Esc, cursor and other keys
         goto startlabel;
      if (c == 3) { // Strg-C, kill command
         pos = 0; s[0] = 0;
         if (echo) printf ("\n");
         return 0;
      }
      if (c == 4 && pos == 0) { // Strg-D in first column
         strncpy (s, "exit", 5);
         if (echo) printf ("exit\n");
         return 0;
      }
      if ((c == 0x08) && (pos>0)) { // backspace
         pos--;
         if (echo) write (STDOUT_FILENO, "\010 \010", 3);
      } else if ( c == '\n' ) { // newline, end of input
         if ((print_newline == 1) && echo) write (STDOUT_FILENO, "\n", 1);
         s[pos] = '\0';
         return 0;
      } else if ( (c != 0x08) && (pos < maxlength) ) { // other character
         if (echo) write (STDOUT_FILENO, &c, 1);
         s[pos++] = c;
      }
   };
}

Defines:
ureadline, used in chunks 214, 430, 432a, and 586b.
Uses kill 568b, printf 600, printf 601a, read 429b, STDIN_FILENO 415b, STDOUT_FILENO 415b, strncpy 594b, and write 429b.
Since `getS` is a traditional Unix function for reading from standard input, we provide it as a macro that uses `ureadline` with a maximum string length of 9999 characters. Note that using `getS` is deprecated since an application cannot control the length of the input which is likely to cause problems when the reserved buffer overflows due to overly long input.

The `ureadchar` function reads just one single character. It is used by the `/bin/vi`, `/bin/keys` and `/bin/hexdump` programs:

```c
char *u_getcwd (char *buf, int size)
{
    strncpy (buf, thread_table[current_task].cwd, size);
    return buf;
}
```

12.4.3 Working Directory, Relative Paths

We want processes to have a "current working directory", so we add an entry to the thread control block structure:

```c
char cwd[256];
```

To query and set this value, we will need two functions `u_getcwd` and `u_chdir` which can also be accessed by user mode functions `getcwd` and `chdir` via system calls.

Now `u_getcwd` just copies a string, while `chdir` needs to check whether the argument is a valid directory:

```c
char *u_getcwd (char *buf, int size) {
    strncpy (buf, thread_table[current_task].cwd, size);
    return buf;
}
```
return 0; // already at root directory
// change to ..
strncpy (abspath, thread_table[current_task].cwd, 256);
splitpath (abspath, dir, base);
strncpy (thread_table[current_task].cwd, dir, 256);
return 0;
}

// check relative/absolute path
if (*path != '/') relpath_to_abspath (path, abspath);
else  strncpy (abspath, path, 256);

// check if abspath is directory
struct stat st;
u_stat (abspath, &st);
if ((st.st_mode & S_IFDIR) == S_IFDIR) {
  strncpy (thread_table[current_task].cwd, abspath, 256);
  return 0;
} else {
  return -1; // error
}
}

As usual, we define and register system call functions ...

\[\textworldsetup{fontsize=10pt}
\langle \text{syscall prototypes} \rangle
\]

void syscall_getcwd (context_t *r);
void syscall_chdir (context_t *r);

\[\textworldsetup{fontsize=10pt}
\langle \text{syscall functions} \rangle
\]

void syscall_getcwd (context_t *r) {
  // ebx: buffer for directory
  // ecx: maximum length of path
  eax_return ( u_getcwd ((char*)r->ebx, r->ecx) );
}

void syscall_chdir (context_t *r) {
  // ebx: new directory
  eax_return ( u_chdir ((char*)r->ebx) );
}

As usual, we define and register system call functions ...
install_syscall_handler (__NR_getcwd, syscall_getcwd);
install_syscall_handler (__NR_chdir, syscall_chdir);

Uses __NR_chdir 204c, __NR_getcwd 204c, install_syscall_handler 201b, syscall_chdir 433b, and syscall_getcwd 433b.

...and provide user mode library functions:

\[
\text{\textbf{ulixlib function prototypes}}
\]

\[
\text{\textbf{ulixlib function implementations}}
\]

\[
\text{\textbf{ulixlib function prototypes}}
\]

\[
\text{\textbf{ulixlib function implementations}}
\]

Defines:
getcwd, used in chunk 434b.
Uses __NR_chdir 204c, __NR_getcwd 204c, syscall2 203c, and syscall3 203c.
12.5 The Minix Filesystem

We have chosen to use the Minix [Tan87] filesystem as the native filesystem for ULIX for two reasons:

- While Minix has all the properties of a Unix filesystem, it is very simple. That also means that explaining and implementing its function is fit for an introductory book. There is no redundancy (for example, Minix does not create backup copies of the superblock, like other Unix filesystems do), so the code does not become complex.
- Minix was the first filesystem that Linux used, and it is still supported. Thus, on a Linux machine the commands `mkfs.minix` and `fsck.minix` are available for creating and checking Minix volumes. The last tool is especially helpful because it allowed us to check the correctness of our implementation: Whenever earlier revisions of the ULIX code wrote wrong data to the volume, `fsck.minix` detected that.

Conceptually, the Minix filesystem uses data structures which can also be found in all other Unix filesystems:

**Superblock:** A superblock contains general information about the whole filesystem. For example it tells how large the volume is. In the Minix case it also lists the maximum numbers of data blocks and inodes as well as where the first data block starts (after the metadata).

**Inode:** For every file on the volume there is an inode (index node) that describes the file. You can find the file size, owner and group IDs, access permissions, timestamps and some pointer to the data blocks in an inode. How these data are organized and how exactly the data blocks can be found after inspecting the inode depends on the specific filesystem.

**Directory:** In order to have a hierarchic filesystem, directories are used. In all Unix filesystems a directory is a simple file that maps filenames to inode numbers. The version of the Minix filesystem that we will look at uses 32 bytes for each directory entry: 30 bytes for the file name and two bytes for a 16-bit inode number. Note again: Directories are files, too (if of a special kind), so when we create a new directory we also need a new inode that describes this directory (file). A freshly created filesystem already contains the root directory.

**Bitmaps:** Inodes are reserved on the volume when it is created. While it would be possible to scan the inode table for a free inode, this would take too long on larger filesystems. So there is also a bitmap that holds a bit for each inode that indicates if it is free or not. Similarly there is a second bitmap which describes the free/used status of the data blocks.

We will first look at the Minix filesystem by creating one on a Linux machine and trying to understand some of its properties. For that purpose we format a floppy image with the Minix filesystem. We create a 1440 KByte image file with `dd`:
### Filesystems

[436a] \(\text{create 1.4 MB disk file 436a}\) ≡

```
$ dd if=/dev/zero of=minixfs.img bs=1k count=1440
1440+0 records in
1440+0 records out
1474560 bytes (1.5 MB) copied, 0.219079 s, 6.7 MB/s
```

and then format it with `mkfs.minix`:

[436b] \(\text{format the disk image with minix fs 436b}\) ≡

```
$ /sbin/mkfs.minix -2 minixfs.img
480 inodes
1440 blocks
Firstdatazone=34 (34)
Zonesize=1024
Maxsize=2147483647
```

The option `-2` creates a version 2 filesystem with “long” filenames (up to 30 characters; the option `-n 14` would enable “short” filenames that have only up to 14 characters). As a next step, `hexdump` will show that there is not much data on a freshly formatted Minix filesystem. (We removed lines that displayed only `0x00` bytes from the output.)

[436c] \(\text{look at the image with hexdump 436c}\) ≡

```
$ hexdump -C minixfs.img
00000400 e0 01 00 00 01 00 01 00 22 00 00 00 ff ff ff 7f |........".......|
00000410 78 24 01 00 a0 05 00 00 00 00 00 00 00 00 00 00 |x$...............
00000800 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000830 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000840 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |................|
* 00000c00 03 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000ca0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000cb0 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |................|
* 00001000 ed 41 02 00 e8 03 e8 03 40 00 00 00 66 89 eb 53 |.A.......@..f..S|
00001010 66 89 eb 53 66 89 eb 53 22 00 00 00 00 00 00 00 |f$F..S".........|
00008800 01 00 2e 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00008820 01 00 2e 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00008830 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00008840 00 00 2e 62 61 64 62 6c 6f 63 6b 73 00 00 00 00 |...badblocks....|
```

We cannot interpret the output without knowledge of the internal data structures; we will explain them in the next section where we present the implementation.

Next we ask `fsck.minix` to display as much information as it can. The `file` command also recognizes the file type:

[436d] \(\text{fsck on an empty minix filesystem 436d}\) ≡

```
$ /sbin/fsck.minix -sfv minixfs.img
Forcing filesystem check on minixfs.img
480 inodes
1440 blocks
Firstdatazone=34 (34)
Zonesize=1024
Maxsize=2147483647
Filesystem state=1
Filesystem state=1
```

```
name=fsck.minix
```
Now we want to see what happens when we write a file onto that filesystem. For that purpose we mount the image and then create a file. We then print a new hexdump; the output only shows the changed or new lines:

```
⟨write file to disk image 437⟩
$ sudo mount -o loop minixfs.img /mnt
$ sudo echo "Hello World" > /mnt/hello.txt
$ sudo umount /mnt
$ hexdump -C minixfs.img  # only changed lines are shown
00000800 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000c00 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00001000 ed 41 02 00 e8 03 e8 03 60 00 00 00 00 00 00 00 |.A......`......S|
00001010 d8 8b eb 53 d8 8b eb 53 22 00 00 00 00 00 00 00 |...S...S".......|
00001040 a4 81 01 00 e8 03 e8 03 0c 00 00 00 d8 8b eb 53 |...............S|
00001050 d8 8b eb 53 d8 8b eb 53 23 00 00 00 00 00 00 00 |...S...S#.......|
00008840 02 00 68 65 6c 6c 6f 2e 74 78 74 00 00 00 00 00 |..hello.txt.....|
00008c00 48 65 6c 6c 6f 20 57 6f 72 6c 64 0a 00 00 00 00 |Hello World.....|
$ /sbin/fsck.minix -sfv minixfs.img
[...]
2 inodes used (0%)
36 zones used (2%)

1 regular files
1 directories
[...]
```

So what happened here? First of all, fsck.minix tells us that one more inode and one more zone are used and that we have one regular file. Zones are blocks (of size 1 KByte); the original Minix filesystem implementation (on Minix) allows zones to have a larger size, but the Linux mkfs.minix tool cannot create such volumes. So from now on, whenever you read “zone”, think “block”.

As you can see, the filename hello.txt and the file contents Hello World show up in the new hexdump, and some of the other locations have new values, for example at addresses 0x800 and 0xc00 the bytes have changed from 0x03 to 0x07. In binary these numbers are 0b1001110 and 0b10000111: One bit was flipped from 0 to 1 in both locations. We will soon see that these newly set bits refer to a new inode and a new zone—which makes sense since we created a new file which is so small that it fits in one block.

We perform another test (with a freshly dd’ed and mkfs.minix-formatted filesystem) and populate it with more than one file:
We copy the file testfile1.txt (6144 bytes, hex.: 0x1800, exactly six blocks) onto the filesystem.

With sed -e 's/file1/file2/' < testfile1.txt > testfile2.txt we create a slightly modified copy (testfile2.txt),

and then use ln to create a hard link (Hardlink.txt)

and ln -s to create a symbolic link Symlink.txt (of testfile1.txt).

When listing the filesystem’s root directory with ls, we get:

```
$ ls -il
2 -rw-r--r-- 2 root root 6144 2014-06-04 23:32 Hardlink.txt
2 -rw-r--r-- 2 root root 6144 2014-06-04 23:32 testfile1.txt
3 -rw-r--r-- 1 root root 6144 2014-06-04 23:32 testfile2.txt
$ ls -ild /mnt
1 drwxr-xr-x 2 root root 192 2012-06-04 23:33 /mnt
```

Every inode has an internal number. The output shows that the inodes with numbers 1–4 are in use (see first column of the ls output). Looking at the image again with hexdump, would reveal the root directory’s table of contents and the contents of the two files.

The following blocks are in use:

- The root directory / uses block 34.
- The file testfile1.txt uses blocks 35–40.
- The file testfile2.txt uses blocks 41–46.
- The symbolic link Symlink.txt uses block 47. (Symbolic links need data blocks as well!)
- No further blocks are in use, the hard link is just a further entry in the root directory.

The **inode bitmap** starts at position 0x800 in the image file and has the following contents which we display with our bindump tool (see p. 630) that works like hexdump but displays bytes as binary numbers:

```
# bindump -r < minixfs.img
[...]
00000800 11110000 00000000 00000000 00000000 00000000 00000000 00000000 ........
00000808 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ........
00000810 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ........
[...]
```

Those five ones represent inode numbers 0–4—however, there is no inode 0. This is what the **zone bitmap** (that starts at offset 0xc00 in the image) looks like:
The 15 set bits refer to block numbers 33–47. Block 33 is used by the inodes and is not a data block! So this represents 14 used data blocks (which fits what we told you above: The 14 blocks 34–46 are in use.)

In comparison, when looking at a freshly created (empty) Minix filesystem (with no files and an empty root directory) the inode and zone bitmaps look like this:

```
# bindump -r < minix-empty.img
[...]
00000800 11000000 00000000 00000000 00000000 00000000 00000000 00000000 ........
00000808 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ........
[...]
00000c00 11000000 00000000 00000000 00000000 00000000 00000000 00000000 ........
00000c08 00000000 00000000 00000000 00000000 00000000 00000000 00000000 ........
[...]
```

Two inodes (with numbers 0, 1) and two blocks (numbers 33, 34) are marked as used. Inode 0 does not exist, and inode 1 stores information about the root directory. A directory is never empty, since it always contains . and .. entries.

It is now time to properly introduce the data structures that Minix filesystems use; they will shed light on the hexdump we’ve shown earlier.

### 12.6 The Ulix Implementation of the Minix Filesystem

There are five variants of the Minix filesystem which differ in the sizes of inodes and block numbers (leading to different maximum file sizes) and the maximum length of filenames (14, 30 or 60). For any Minix filesystem image you can find out its version. Our implementation allows access to version 2 of the filesystem with a filename length of up to 30 characters. The Linux tool `mkfs.minix` will create a Minix version 1 filesystem (with 30-character filenames and a theoretical maximum file size of 256 MByte) by default, but this can be changed by supplying the options `-v` or `-2` (for version 2 with an approximate maximum filesize of 2 GByte) or `-3` (for version 3). For version 1 and 2, the filename length can be set to 14 with the `-n 14` option which reduces the size of a directory entry from 32 bytes to 16 bytes, whereas version 3 only supports a filename length of 60 characters. This leads to the characteristics shown in Table 12.3.

The standard zone size of a Minix filesystem is 1 KByte (1024 bytes). It is possible to increase this size to $1024 \times 2^n$ bytes (for some $n$), but not with the Linux tool `mkfs.minix`. The first block in the filesystem contains the boot sector (which we will ignore), the second block is the `superblock` which contains the setup information of a specific filesystem, including two magic bytes which tell the filesystem version number.
<table>
<thead>
<tr>
<th>Minix version</th>
<th>inode number size</th>
<th>directory entry size</th>
<th>entries per block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1 (filenames: 14)</td>
<td>2 bytes</td>
<td>16 bytes</td>
<td>64</td>
</tr>
<tr>
<td>Version 1 (filenames: 30)</td>
<td>2 bytes</td>
<td>32 bytes</td>
<td>32</td>
</tr>
<tr>
<td>Version 2 (filenames: 14)</td>
<td>2 bytes</td>
<td>16 bytes</td>
<td>64</td>
</tr>
<tr>
<td>Version 2 (filenames: 30)</td>
<td>2 bytes</td>
<td>32 bytes</td>
<td>32</td>
</tr>
<tr>
<td>Version 3 (filenames: 60)</td>
<td>4 bytes</td>
<td>64 bytes</td>
<td>16</td>
</tr>
</tbody>
</table>

Table 12.3: Characteristics of the Minix filesystem versions.

Since the Minix filesystem is the standard filesystem for Unix, we will always work with KByte-sized blocks—even in functions that work on a lower level. We define:

```c
#define BLOCK_SIZE 1024
```

Defines:
- `BLOCK_SIZE`, used in chunks 451a, 453b, 471–73, 475, 476b, 480c, 484e, 496d, 497, 508–10, 515a, 518b, 521a, and 582a.

### 12.6.1 The Minix Superblock

Every Unix filesystem has a superblock: that is a block which stores global information about a specific volume, and it must always be visited upon first interaction with a volume. It is created when the volume is formatted. In the case of Minix its contents are immutable; they remain the same over the lifetime of that volume.

We start our Minix filesystem implementation with a look at the superblock. All functions will appear inside the `<minix filesystem implementation 420c>` code chunk:

```c
struct minix_superblock {
    uint16_t s_ninodes;  // 2 bytes
    uint16_t s_nzones;   // 2 bytes
    uint16_t s imap_blocks; // 2 bytes
    uint16_t s zmap_blocks; // 2 bytes
    uint16_t s firstdatazone; // 2 bytes
    uint16_t s log_zone_size; // 2 bytes
    uint32_t s max size; // 4 bytes
    uint16_t s_magic; // 2 bytes
    uint16_t s_state; // 2 bytes
    uint32_t s_zones; // 4 bytes
};
```

Defines:
- `minix_superblock`, used in chunks 443b, 448, and 492.

The `uint16_t` and `uint32_t` types are defined in `/usr/include/stdint.h` (on a Linux system). They are 16 bit and 32 bit wide unsigned integers, respectively. Thus, the superblock only uses 24 bytes.
When you copy the superblock into a `struct minix_superblock` variable, you can check (or print) the values; an analysis of these values is a first step towards properly accessing the filesystem. The `s_magic` field tells what version of the Minix filesystem was used when the medium was formatted. There are five possible cases:

- Minix v1 (14 characters per filename): 0x137F
- Minix v1 (30 characters per filename): 0x138F
- Minix v2 (14 characters per filename): 0x2468
- Minix v2 (30 characters per filename): 0x2478
- Minix v3 (60 characters per filename): 0x4D5A (but stored elsewhere since the v3 superblock has a different layout)

Without checking the version it is impossible to access the filesystem since the versions differ in size and content of the inodes and the directory entries. A version 1 superblock stores the number of blocks in the `s_nzones` entry, whereas a version 2 superblock uses the `s_zones` entry. The unused value is set to 0.

As mentioned before, instead of blocks, Minix uses “zones” as the smallest allocatable unit. Typically a zone contains just one block, but theoretically a zone may consist of a collection of blocks if `s_log_zone_size` is non-zero; the following formula expresses the relationship between zone size and block size:

\[
\text{zone size} = \text{block size} \times 2^{s\_log\_zone\_size}
\]

(With a default setting of `s_log_zone_size = 0` that formula reads: `zone size = block size`, since \(2^0 = 1\).) In our implementation we only support the case where block size = zone size.
size = 1024, and so we will often use the terms zone and block interchangeably. Some filesystems use the name cluster to discuss smallest allocatable units, thus a zone is also a cluster.

The s_imap_blocks and s_zmap_blocks fields note how many blocks are reserved for the inode bitmap and the zone bitmap. Those bitmaps store a single bit for each inode or data block, respectively, and the bits show whether an inode/data block is free (0) or occupied (1). s_ninodes is the number of inodes from which we can calculate the size of the inode table.

The inode bitmap and the zone bitmap follow directly after the superblock (see Figure 12.10). After that the filesystem contains the inode table and finally the data blocks.

The zone bitmap starts with a 1 bit (which does not represent any zone); the second bit (bit 1) refers to the first data zone that contains (the start of) the filesystem’s root directory. Blocks before the data blocks area are not available for data storage, thus they are not represented in the zone bitmap.

An inode of a Minix (version 2) filesystem has the following form:

```c
struct minix2_inode {
    // external minix2 inode
};

Define:
minix2_inode, used in chunks 451–53, 456, 457b, 461d, 466–68, 475c, 478–80, 484c, 487a, 488a, 490a, 589d, 607a, and 610d.
```

We’re placing the minix2_inode entries in a separate code chunk since we will later define another inode data structure for the in-memory management of open files; there we will also need these fields.

Thus, we can calculate the size of an inode: it uses sizeof(struct minix2_inode) = 24 + 4 × 10 = 64 bytes, which lets 1024/64 = 16 inodes fit inside one inode table block. When we look at a 1.44 MByte floppy disk that was formatted with the Minix version 2 filesystem (using mkfs.minix -v), we find that the superblock contains the following values:

```c
s_ninodes: 480
s_nzones: 0
s_imap_blocks: 1
s_zmap_blocks: 1
s_firstdatazone: 34
s_log_zone_size: 0
s_max_size: 2147483647
s_magic: 9336
s_state: 1
s_zones: 1440
```
That tells us:

- There are 480 inodes. With 16 inodes per block the inode table requires \(480/16 = 30\) blocks.
- The inode bitmap consists of only 480 bits (= 60 bytes) and fits in one block, with the rest of the block remaining unused. During filesystem creation the unused bits are filled with 1s.
- 1440 blocks require 1440 bits (= 180 bytes) for the zone bitmap which (again) fit in one block. This bitmap also fills the unused bits with 1s.

This leads to the layout shown in Table 12.4.

<table>
<thead>
<tr>
<th>Blocks</th>
<th>Usage</th>
<th>Absolute Bytes</th>
<th>Absolute Bytes (hex)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>unused (boot sector)</td>
<td>0–1023</td>
<td>0x0000 – 0x03ff</td>
</tr>
<tr>
<td>1</td>
<td>superblock</td>
<td>1024–2047</td>
<td>0x0400 – 0x07ff</td>
</tr>
<tr>
<td>2</td>
<td>inode bitmap</td>
<td>2048–3071</td>
<td>0x0800 – 0x0bff</td>
</tr>
<tr>
<td>3</td>
<td>zone bitmap</td>
<td>3072–4095</td>
<td>0x0c00 – 0x0fff</td>
</tr>
<tr>
<td>4–33</td>
<td>inode table (30 blocks)</td>
<td>4096–34815</td>
<td>0x1000 – 0x87ff</td>
</tr>
<tr>
<td>34–1439</td>
<td>data blocks (zones)</td>
<td>34816–…</td>
<td>0x8800 –…</td>
</tr>
</tbody>
</table>

Table 12.4: The layout of a Minix version 2 floppy disk formatted with `mkfs.minix -v`.

From calculating the sizes of the bitmaps and the inode table we know that the first data block is block 34, but this information is also stored (redundantly) in the superblock’s `s_firstdatazone` field.

We provide a function which extracts specific values from the superblock, e.g., the number of inodes. Since we do not want to write several similar functions we combine this in one function called `mx_query_superblock` that expects a device identifier and a constant which refers to some specific property:

```c
int mx_query_superblock (int device, char index)
{
    byte block[1024];
    struct minix_superblock *sblock;
    readblock (device, 1, (byte*)block); // superblock = block 1
    sblock = (struct minix_superblock*) &block;
    switch (index) {
        case MX_SB_NINODES: return sblock->s_ninodes;
        case MX_SB_NZONES: return sblock->s_nzones;
        case MX_SB_IMAP_BLOCKS: return sblock->s_imap_blocks;
        case MX_SB_ZMAP_BLOCKS: return sblock->s_zmap_blocks;
        case MX_SB_FIRSTDATAZONE: return sblock->s_firstdatazone;
        case MX_SB_LOG_ZONE_SIZE: return sblock->s_log_zone_size;
        case MX_SB_MAX_SIZE: return sblock->s_max_size;
    }
}
```
case MX_SB_MAGIC: return sblock->s_magic;
case MX_SB_STATE: return sblock->s_state;
case MX_SB_ZONES: return sblock->s_zones;
default: return -1; // error
}

Defines:
mx_query_superblock, used in chunks 443a, 445, 451a, and 492.
Uses minix_superblock 440c and readblock 506b.

These constants can be declared as follows:

```
⟨constants 112a⟩+≡
enum { MX_SB_NINODES, MX_SB_NZONES, MX_SB_IMAP_BLOCKS, MX_SB_ZMAP_BLOCKS,
      MX_SB_FIRSTDATAZONE, MX_SB_LOG_ZONE_SIZE, MX_SB_MAX_SIZE,
      MX_SB_MAGIC, MX_SB_STATE, MX_SB_ZONES }
```

### 12.6.2 Zone and Inode Bitmaps

**Going where?**

Now that we know how to access the superblock, our next task is to deal with the zone and inode bitmaps properly. This requires some fiddling to extract or modify single bits of a byte.

We need ways to access single bits in the inode and zone bitmaps. Reading is simple, because we only need to find the right byte and then perform a bit-shift operation, followed by a modulo operation to isolate a specific bit.

Let’s start with the inode bitmap: One block stores 1024 bytes = 8192 bits, and the inode bitmap begins in block 2. Thus if i is the number of the bit we want to read, we must read in block \(2 + \frac{i}{8192}\). Inside that block we need to read byte number \(\frac{i}{8192} \mod 8\).

```
byte mx_get_imap_bit (int device, int i) {
    byte block[1024];
    byte thebyte;
    readblock (device, 2 + i/8192, (byte*)&block);
    thebyte = block[\(\frac{i}{8192}\) % 8];
    return (thebyte >> (i%8)) % 2;
}
```

Defines:
mx_get_imap_bit, used in chunk 451a.
Uses readblock 506b.

For the zone map, we need to consider that the inode map (which is located just before it) may be larger than one block (if we have more than 8192 inodes). We can query the superblock to find out how many blocks are used.

Again, the zone map may be larger than one block, so we have to find out which block we need to read via a similar calculation.
### 12.6 The Unix Implementation of the Minix Filesystem

#### Implementation of the Minix Filesystem

```c
(byte mx_get_zmap_bit (int device, int i) {
    byte block[1024];
    byte thebyte;
    unsigned int zmap_start = 2 + mx_query_superblock (device, MX_SB_IMAP_BLOCKS);
    readblock (device, zmap_start + i/8192, (byte*)&block);
    thebyte = block[(i%8192)/8];
    return (thebyte >> (i%8)) % 2;
}
```

Uses `mx_query_superblock` 443b and `readblock` 506b.

In order to fetch the bit number `i` of an integer number `n` we use the formula `(n >> i) % 2` which performs a right shift (by `i` positions) and then cuts out the lowest bit with `% 2`.

We also need to set and clear individual bits. Since the code for accessing and changing a bit is almost identical for setting and clearing, we write two functions `mx_set_clear_*` which can both set and clear; they are called by the four `mx_set_*` and `mx_clear_*` functions with appropriate arguments:

```c
(void mx_set_clear_imap_bit (int device, int i, int value) {
    byte block[1024];
    byte thebyte;
    readblock (device, 2 + i/8192, (byte*)&block);
    (set bit i from block block to value 445c)
    writeblock (device, 2 + i/8192, (byte*)&block);
});
```

```c
(void mx_set_clear_zmap_bit (int device, int i, int value) {
    byte block[1024];
    byte thebyte;
    unsigned int zmap_start = 2 + mx_query_superblock (device, MX_SB_IMAP_BLOCKS);
    readblock (device, zmap_start + i/8192, (byte*)&block);
    (set bit i from block block to value 445c)
    writeblock (device, zmap_start + i/8192, (byte*)&block);
});
```

Defines:
- `mx_set_clear_imap_bit`, used in chunk 446.
- `mx_set_clear_zmap_bit`, used in chunk 446.

Uses `mx_query_superblock` 443b, `readblock` 506b, and `writeblock` 507c.

where setting the bit from a block looks like this in both cases:

```c
(set bit i from block block to value 445c)\equiv
```
If the shift (<<), modulo (\%), bitwise “and” (&), bitwise “or” (|) and bitwise negation (~) operations seem like magic to you, consider the following example calculations:

- For clearing bit 3 of 00101100:

  byte = 00101100
  1<<3 = 00001000
  ~(1<<3) = 11110111
  00101100 & 11110111 = 00100100

- For setting bit 3 of 00100100:

  byte = 00100100
  1<<3 = 00001000
  00100100 | 00001000 = 00101100

This should explain setting and clearing a bit in a single byte well enough. The extra code which reads and writes block[(i%8192)/8] is necessary because we do not deal with a single byte but an array of such bytes.

Now the mx_set_* and mx_clear_* functions simply provide the right value (0 or 1) to the more general mx_set_clear_* function:

```c
#define mx_clear_imap_bit (device, i) \{
    mx_set_clear_imap_bit (device, i, 0);
}\n
#define mx_clear_zmap_bit (device, i) \{
    mx_set_clear_zmap_bit (device, i, 0);
}\n
#define mx_set_zmap_bit (device, i) \{
    mx_set_clear_zmap_bit (device, i, 1);
}\n
#define mx_request_inode (device, i) \{
    mx_request_inode (device, i);
}\n
#define mx_request_block (device, i) \{
    mx_request_block (device, i);
}\n
void mx_set_imap_bit (int device, int i) \{
    mx_set_clear_imap_bit (device, i, 1);
}\n
void mx_clear_imap_bit (int device, int i) \{
    mx_set_clear_imap_bit (device, i, 0);
}\n
void mx_set_zmap_bit (int device, int i) \{
    mx_set_clear_zmap_bit (device, i, 1);
}\n
void mx_clear_zmap_bit (int device, int i) \{
    mx_set_clear_zmap_bit (device, i, 0);
}\n```

Defines:
- `mx_clear_imap_bit`, used in chunk 483.
- `mx_clear_zmap_bit`, used in chunks 477, 481, 482, and 484e.
- `mx_set_zmap_bit`, used in chunk 447a.
Uses `mx_set_clear_imap_bit` 445b and `mx_set_clear_zmap_bit` 445b.

Note: An optimized implementation will not do this calculation every single time, instead when mounting the filesystem, we should copy the superblock to memory and also memorize where the zone bitmap starts. Early versions of U\textsc{ux} suffered from very slow disk access because reading blocks was not buffered—this led to actually reading the superblock from disk whenever we wanted to query the zone bitmap. With buffered read operations this is no longer a problem but still highly inefficient. It is, however, the simplest implementation and thus easy to grasp. Yet, you will see on the next pages that we did not stick with it because a slightly optimized version improved performance a lot.

Requesting a free inode or a free block means searching the corresponding bitmap for a zero bit. We start with the simple implementation of two `mx_request_inode()` and `mx_request_block` functions which just loop over the whole bitmaps and check the bits with `mx_get_*_bit`. If all inodes or blocks are in use, these functions return -1.

Note that (as described above) the zone bitmap does not start with an entry for block 0, but with a fixed 1 entry, followed by the bit that describes the first data zone. In order to query the state of data zone n we need to call `mx_get_zmap_bit` (device, n-s_firstdatazone +1). If this is unclear, go back to the example filesystem where s_firstdatazone is 34, then evaluate the expression for n=34.
\[ \text{(old minix filesystem implementation 447a)} \]

```c
int mx_request_inode (int device) {
    int no_inodes = mx_query_superblock (device, MX_SB_NINODES); // floppy: 480
    for (int i = 0; i < no_inodes; i++) {
        if (mx_get imap_bit (device, i) == 0) {
            // found a free inode
            mx_set imap_bit (device, i); // mark as used
            return i;
        }
    }
    return -1; // found nothing
}
```

```c
int mx_request_block (int device) {
    int no_zones = mx_query_superblock (device, MX_SB_ZONES); // floppy: 1440
    int first_data = mx_query_superblock (device, MX_SB_FIRSTDATAZONE);
    for (int i = 0; i < no_zones - first_data - 2; i++) {
        if (mx_get zmap_bit (device, i) == 0) {
            mx_set zmap_bit (device, i); // mark as used
            return i + first_data - 1; // floppy example: i+33
        }
    }
    return -1; // found nothing
}
```

While the above implementations of `mx_request_inode` and `mx_request_block` are correct, they are also highly inefficient. Thus, we will provide a second implementation which has a better performance. Normally, we do not focus on performance issues, but these functions are very slow which makes writing a new file unbearable.

We start with a helper function

\[ \text{(function prototypes 45a)} \]

```c
int findZeroBitAndSet (byte *block, int maxindex);
```

which finds the first 0 bit in a block, changes it to 1 and returns its (bit) position

\[ \text{(minix filesystem implementation 420c)} \]

```c
int findZeroBitAndSet (byte *block, int maxindex) {
    int i, j;
    byte b;
    for (i = 0; i < 1024; i++) {
        b = block[i];
        if (b != 0xFF) {
            // at least one bit in this byte is 0, find the first one
            for (j = 0; j < 8; j++) {
                if (((b >> j) % 2 == 0) // bit is 0  
                    && (i*8 + j < maxindex)) // bit position is ok
                    {
                    block[i] = b | (1 << j); // set bit
                    return i*8 + j;
                }
            }
        }
    }
    // not found
    return -1;
}
```
We need to provide a maxindex argument since the last block of the bitmap may not always be fully used. In the floppy example from above, only 480 bits of the inode bitmap and only less than 1440 bits of the inode bitmap have to be considered.

The implementations of `mx_request_inode` and `mx_request_block` that we actually use start with manually reading the superblock (since they need to access several of its entries). The optimization is reached via checking a whole block for a 0 bit with the helper function:

```c
int mx_request_inode (int device) {
    byte block[1024];
    struct minix_superblock *sblock;
    readblock (device, 1, (byte*)block); // superblock = block 1
    sblock = (struct minix_superblock*) &block;

    int no_inodes = sblock->s_ninodes; // floppy: 480
    int imap_start = 2;

    int i, index;
    for (i = 0; i < sblock->s_imap_blocks; i++) { // all IMAP blocks
        readblock (device, imap_start + i, (byte*)&block);
        index = findZeroBitAndSet ((byte*)&block, no_inodes);
        if (index != -1) { // found one!
            writeblock (device, imap_start + i, (byte*)&block);
            return i*8192 + index;
        }
    }
    return -1; // found nothing
}

int mx_request_block (int device) {
    byte block[1024];
    struct minix_superblock *sblock;
    readblock (device, 1, (byte*)block); // superblock = block 1
    sblock = (struct minix_superblock*) &block;

    int no_zones = sblock->s_zones; // floppy: 1440
    int zmap_start = 2 + sblock->s_imap_blocks;
    int zmap_blocks = sblock->s_zmap_blocks;
    int data_start = sblock->s_firstdatazone;
```
int i, index;
for (i = 0; i < zmap_blocks; i++) {
    // all ZMAP blocks
    readblock (device, zmap_start + i, (byte*) &block);
    index = findZeroBitAndSet ((byte*) &block, no_zones);
    if (index != -1) {
        // found one!
        writeblock (device, zmap_start + i, (byte*) &block);
        return i*8192 + index + data_start - 1;  // convert to zone number
    }
}
return -1;  // found nothing
};

Defines:
mx_request_block, used in chunks 454a, 476, and 477.
mx_request_inode, used in chunk 478b.
Uses findZeroBitAndSet 447c, minix_superblock 440c, readblock 506b, and writeblock 507c.

Note that mx_request_inode simply returns the bit position of a free entry. On the other hand, mx_request_block returns a block number which is not identical to the bit position since the zone bitmap holds no bits for the early blocks in the filesystem.

12.6.3 Reading and Writing Inodes

We’re now able to query the superblock and read and write the two bitmaps. Our next goal is to create (empty) files.

Since empty files use no data blocks, this requires being able to read and write inodes.

Creating a new (empty) file consists of the following steps:

1. Reserve an inode with mx_request_inode.
2. Write the inode.
3. Create an entry in the file’s directory, i.e., create the (filename → inode number) mapping.

Before we start, remember how pointer arithmetic works; we will sometimes use memcpy to move data from a block around. If that block is declared as char block[1024]; and you want to write a 32 byte chunk to position 512, then the code

```c
offset = 512; size = 32;
memcpy (&block + offset, &data, size);
```

will fail. On the other hand, if the block was declared via char *block; then the similar code

```c
offset = 512; size = 32;
memcpy (block + offset, &data, size);
```

works as expected. The following example program offset-test.c shows the difference:
```c
#include <stdio.h>

int main () {
    char block[1024]; char *block2 = (char*) &block; char data[] = "Test";
    int size = sizeof(data); int offset = 512; long diff;

    printf("&block: \%p \n", &block);
    printf("&block + offset: \%p \n", &block + offset);
    diff = (long)(&block+offset)-(long)&block;
    printf("difference: \%ld \n", diff);

    printf("block2: \%p \n", block2);
    printf("block2 + offset: \%p \n", block2 + offset);
    diff = (long)(block2+offset)-(long)block2;
    printf("difference: \%ld \n", diff);
}
```

generates the following output:

```
$ ./offset-test
&block: 0x7ffff31802b0
&block + offset: 0x7ffff32002b0
difference: 524288 // that is 512 x 1024 !
block2: 0x7ffff31802b0
block2 + offset: 0x7ffff31804b0
difference: 512 // that's what we want
```

In the first attempt, `&block` creates a pointer to `block` that "knows" the size of `block` (which is a whole kilobyte). When adding the offset (512) the program actually adds that offset multiplied with the size (resulting in a 512 KByte offset). So in order to perform correct pointer arithmetic, it is necessary to first perform a cast to a `(char*)` pointer; otherwise you would access wrong memory areas (see also Appendix A.5).

We continue the implementation with two functions

```c
int mx_read_inode (int device, int i, struct minix2_inode *inodeptr);
int mx_write_inode (int device, int i, struct minix2_inode *inodeptr);
```

which copy an inode from disk to memory or vice versa. They shall return 0 when an error occurs and the inode number i otherwise—this lets us write code of the form if (!mx_read_inode(...)) { /* error */ } . Trying to read an unused inode shall also generate an error.

Since reading and writing an inode are similar tasks we write a combined function

```c
int mx_read_write_inode (int device, int i, struct minix2_inode *inodeptr, int wr_flag);
```

which can do both; the flag `wr_flag` decides about the direction.
The only statement that needs some explanation is the calculation of blockno which queries the superblock twice to find out about the layout of the filesystem; the first block of the inodetable is placed behind the zone bitmap. Block 2 is where the inode bitmap starts, and adding mx_query_superblock 443b (device, MX_SB_IMAP_BLOCKS) and mx_query_superblock 443b (device, MX_SB_ZMAP_BLOCKS) brings us right behind the zone bitmap. We need to add i / inodesperblock in order to pick the right block within the inodetable.

As explained above, mx_read_inode 451b and mx_write_inode 452a simply call the function mx_read_write_inode 451a with wr_flag set to 0 or 1:

```c
int mx_read_write_inode (int device, int i, struct minix2_inode *inodeptr, int wr_flag) {
  i--; // first inode is No. 1, but has position 0 in table
  if ((i < 0) || (i ≥ mx_query_superblock (device, MX_SB_NINODES))) {
    return 0; // illegal inode number
  }
  if (mx_get_imap_bit (device, i+1) == 0) {
    return 0; // attempt to read unused inode; forbidden
  }
  const int inodesize = sizeof (struct minix2_inode);
  const int inodesperblock = BLOCK_SIZE / inodesize;
  int blockno = i / inodesperblock + 2
    + mx_query_superblock (device, MX_SB_IMAP_BLOCKS)
    + mx_query_superblock (device, MX_SB_ZMAP_BLOCKS);
  int blockoffset = i % inodesperblock;
  // we need to read the block, even if this is a write operation
  byte block[1024];
  readblock (device, blockno, (byte*)&block);
  byte *addr = (byte*)&block; // add offset, beware of pointer arithmetic
  addr += blockoffset * inodesize;
  if (!wr_flag) {
    memcpy (inodeptr, addr, inodesize);
  } else {
    memcpy (addr, inodeptr, inodesize);
    writeblock (device, blockno, (byte*)&block); // write whole block to disk
  }
  return (i+1); // return original number
}
```

Defines:
- `mx_read_write_inode`, used in chunks 450–52.
- `BLOCK_SIZE`, `memcpy`, `minix2_inode`, `mx_get_imap_bit`, `mx_query_superblock`, `readblock`, and `writeblock`.

The only statement that needs some explanation is the calculation of blockno which queries the superblock twice to find out about the layout of the filesystem; the first block of the inodetable is placed behind the zone bitmap. Block 2 is where the inode bitmap starts, and adding mx_query_superblock 443b (device, MX_SB_IMAP_BLOCKS) and mx_query_superblock 443b (device, MX_SB_ZMAP_BLOCKS) brings us right behind the zone bitmap. We need to add i / inodesperblock in order to pick the right block within the inodetable.

As explained above, mx_read_inode 451b and mx_write_inode 452a simply call the function mx_read_write_inode 451a with wr_flag set to 0 or 1:
12.6.4 Directory Entries

Reserving and writing inodes is only the first half of creating a new file; we also need to create directory entries (in the directory where we want to place a file). Our goal is to write a function `mx_write_link` that takes an inode number and a pathname and creates the link between them.

That task consists of several smaller tasks: We need to be able to read and write single entries in the directory (file), find a free entry in the directory, split a pathname (such as `/home/user/dir/file`) into its base name (file) and its directory name `/home/user/dir`. The directory name will lead us to the directory where we need to place the link.

Let’s look at the task step by step. In order to read the root directory of a volume, we need to know that Minix always uses inode number 1 for it, and it also starts counting inodes with 1. Thus, inode 1 is the first (not the second) inode, stored at position 0 of the inode table, but the corresponding bitmap entry is bit 1 (not 0), see further below.

Looking at the root directory’s inode brings up a special case of file access (before reading the first regular file): Directories are a special kind of file which map filenames to inode numbers. We can read a directory block by block, and the first zone number is stored in `i_zone[0]`. We just need to know how to interpret the data: A directory file is an array of structures of type `minix_dir_entry`:

```c
typedef struct minix_dir_entry {
    uint16_t inode;
    char name[30];
} minix_dir_entry;
```

Each such entry has a size of $2 + 30 = 32$ bytes and starts with the 16-bit inode number, followed by the filename. Such filenames are normally null-terminated (as is standard for all strings on Unix systems), but when a filename uses the maximum allowed size of 30 characters, there is no space for the terminating `\0` character. Thus, simply copying an entry with `strcpy` will fail on filenames with maximum length. When dealing with Minix filenames internally, they should be stored in a `char[31]` string whose last byte is manually set to `\0`.

A block can hold $1024/32 = 32$ such directory entries. If there are more than 32 entries in a directory, an additional block is used (whose block number can be found in the next
The Uôû Implementation of the Minix Filesystem

Now we start the work on the function mx_write_link that can add a (filename → inode number) mapping to a directory. As already mentioned, directories are special files and the root directory has the inode number 1. Each of the associated data blocks contains 32 directory entries of type struct minix_dir_entry (since $2^5 = 1024$). An unused entry has the inode field set to 0. We provide two functions

$$\begin{align*}
\text{int } \text{mx_read_dir_entry} & (\text{int device, int inodenr, int entrynr,} \\
& \text{struct minix_dir_entry *entry})
\end{align*}$$

$$\begin{align*}
\text{int } \text{mx_write_dir_entry} & (\text{int device, int inodenr, int entrynr,} \\
& \text{struct minix_dir_entry *entry})
\end{align*}$$

for reading or writing individual entries of a directory.

We will use the same trick that we applied to reading and writing inodes by providing a common function mx_read_write_dir_entry that can do both and decides via an extra flag wr_flag whether it shall read or write. The other arguments are the device, the inode number of the directory (file) and a pointer to a struct minix_dir_entry variable.

Since only 32 entries fit in one block, we might have to reserve a new block and enter its location in the directory’s inode; then we can go on writing entries in the new block. Similarly, when we later delete entries, we might want to remove additional blocks that are no longer needed.

In each inode’s i_zone array only the first seven entries refer directly to data blocks, so using these we can work with up to $7 \cdot 32 = 224$ directory entries. After that an indirection block must be used, but we will restrict our Minix implementation to a maximum of 224 entries for a directory.

```c
int mx_read_write_dir_entry (int device, int inodenr, int entrynr, 
struct minix_dir_entry *entry, int wr_flag) { 
    if (entrynr ≥ 32 * 7) { // 7 direct blocks, 32 entries per block 
        return false; 
    }
    struct minix2_inode inode; 
    mx_read_inode (device, inodenr, &inode); // read directory inode
    int blockno; 
    blockno = inode.i_zone[entrynr/32]; // number of block that holds the entry 
    if (blockno == 0) {
        if (wr_flag) { // reserve a block and map it in the directory inode 
            return false; 
        } 
    }
    char block[1024]; 
    readblock (device, blockno, (byte*)&block); 
    int offset = (32*entrynr) % BLOCK_SIZE;
```
if (!wr_flag) {
    memcpy (entry, ((char*)&block)+offset, 32);  // reading
    return (entry->inode != 0);                     // true if entry non-empty
} else {
    memcpy (((byte*)&block)+offset, entry, 32);   // writing
    writeblock (device, blockno, (byte*)&block);
    return true;
};

int mx_read_dir_entry (int device, int inodenr, int entrynr, 
                       struct minix_dir_entry *entry) {
    return mx_read_write_dir_entry (device, inodenr, entrynr, entry, false);
};

int mx_write_dir_entry (int device, int inodenr, int entrynr, 
                        struct minix_dir_entry *entry) {
    return mx_read_write_dir_entry (device, inodenr, entrynr, entry, true);
};

Defines:
mx_read_dir_entry, used in chunks 456, 462a, 480c, and 490d.
mx_write_dir_entry, used in chunks 453a, 456, and 480c.

Uses BLOCK_SIZE 440a, memcpy 596c, minix2_inode 442a, minix_dir_entry 452b, mx_read_inode 451b, 
readblock 506b, and writeblock 507c.

If the block which should hold the directory entry does not yet exist (and we’re trying 
to write to it), we create it and enter it in the directory inode:

reserve a block and map it in the directory inode 454a)≡

(453b)

blockno = mx_request_block (device);
char empty_block[1024] = { 0};
writeblock (device, blockno, (byte*)&empty_block);
inode.i_zone[(entrynr/32) = blockno;
mx_write_inode (device, inodenr, &inode);  // update directory inode

Uses mx_request_block 448, mx_write_inode 452a, and writeblock 507c.

For dealing with pathnames we will sometimes need two helper functions: dirname and basename, can be used to split a path into a directory (path) and a file or directory name, for example dirname ("/usr/bin/vi") = "/usr/bin" and basename ("/usr/bin/vi") = "vi". This works similarly for relative paths, and the special case of a pathname x without any slashes is handled by dirname ("x") = "." and basename ("x") = "x".

We will also recycle these functions in the user mode library, since it does not matter whether the kernel or a program wants to split a pathname. Instead of parsing the path in two separate functions, we write a combined function splitpath and call that one from basename and dirname.

public function prototypes 454b)≡

(44a 48a) 593>

void splitpath (const char *path, char *dirname, char *basename);
char *basename (char *path);
char *dirname (char *path);
12.6 The UNIX Implementation of the Minix Filesystem

\[\text{public function implementations}] \equiv \]
\[
\text{void splitpath (const char *path, char *dirname, char *basename) \{
\text{if (strlen (path) == 1 \\&\& path[0] == '/') // special case "/"
\text{strncpy (dirname, "/", 1); strncpy (basename, "/", 1); return;
\}
\text{char p[256]; strncpy (p, path, 256); // work on copy
int pos = strlen (p) - 1;
if (p[pos] == '/') // strip trailing '/'
\text{for (;;) \{ // search for / (from back to front
pos--;}
\text{if (pos == -1) \{ // no single slash found
strncpy (dirname, ".", 2); strncpy (basename, p, 256); return;
\}
\text{if (p[pos] == '/') \{
\text{if (pos==0)
strncpy (dirname, "/", 2); // special case "/
else {
\text{memcpy (dirname, p, pos);
dirname[pos] = 0; // remove trailing '/'
\}
\text{strncpy (basename, p + pos + 1, 30);
return;
\}
\}
\}
\}
\]
\text{Defines: splitpath, used in chunks 419a, 432e, 454–56, 480c, 487a, 488a, and 577.
Uses basename 455b, dirname 455b, memcpy 596c, strlen 594a, and strncpy 594b.}

In the implementations of basename455b and dirname455b we declare bname and dname as static so that they are not stored on the stack; that way we can return a pointer.

\[\text{public function implementations}] \equiv \]
\[
\text{char *basename (char *path) \{
static char bname[30]; static char dname[256];
\text{splitpath (path, dname, bname); return (char *)bname;}
\}
\]
\[
\text{char *dirname (char *path) \{
static char bname[30]; static char dname[256];
\text{splitpath (path, dname, bname); return (char *)dname;}
\}
\]
\text{Defines: basename, used in chunks 455a and 577b.
dirname, used in chunks 419a, 455a, 456, and 577.
Uses splitpath 455a.}

The implementation of the
\[function prototypes] \equiv \]
\[
\text{void mx_write_link (int device, int inodenr, const char *filename);}
\]
function is complex:

- First, we check whether the directory already contains an entry for the filename—it is not possible to have the same filename twice in a directory.
- Then we split the path into the directory name and the base filename.
- We locate the inode that belongs to the directory file using `mx_pathname_to_ino` (a function that we still have to implement; this follows a few pages later).
- Then we read all the directory entries (using `mx_read_dir_entry` from above) until we find a free entry. If we don’t, we have to abort because the directory is full.
- Once we’ve found a free entry, we prepare a directory entry and write it to the free location.
- As a last thought, we must not forget to increase the link count of the inode: It counts how many links to the inode exist. For a freshly created file we could always set that value to 1, but we will also use this function when we create a hard link.
- Finally we update the size of the directory (it may have grown by 32 bytes unless we’ve found a free entry between other, used entries) and write back the modified directory inode.

```c
void mx_write_link (int device, int inodenr, const char *path) {
    if (mx_file_exists (device, path)) { // check if filename already exists
        printf ("ERROR: filename %s exists!\n", path); return;
    }

    struct minix_dir_entry dentry; struct minix2_inode inode;
    char dirname[256]; char filename[30];
    splitpath (path, dirname, filename);
    int dir_inode_no = mx_pathname_to_ino (device, dirname);
    // find free location and enter it
    mx_read_inode (device, dir_inode_no, &inode); // read directory inode
    for (int i = 0; i < 32 * 7; i++) {
        mx_read_dir_entry (device, dir_inode_no, i, &dentry);
        if (dentry.inode == 0 || i * 32 >= inode.i_size) {
            dentry.inode = inodenr; // found an empty entry
            memcpy ((char*)dentry.name, filename, 30);
            mx_write_dir_entry (device, dir_inode_no, i, &dentry);
            mx_increase_link_count (device, inodenr); // link count for file
            if (inode.i_size < 32*(i+1)) {
                mx_read_inode (device, dir_inode_no, &inode); // must read again
                inode.i_size = 32*(i+1);
                mx_write_inode (device, dir_inode_no, &inode);
            }
            return; // success
        }
    }
    printf ("ERROR: no free entry in directory\n"); // search failed
}
```
12.6 The UNIX Implementation of the Minix Filesystem

Defines:
mx_write_link, used in chunks 455c, 478b, and 480a.

Uses dirname 455b, memcpy 596c, minix2_inode 442a, minix_dir_entry 452b, mx_file_exists 479b,
mx_increase_link_count 457b, mx_pathname_to_ino 461d, mx_read_dir_entry 453b, mx_read_inode 451b,
mx_write_dir_entry 453b, mx_write_inode 452a, printf 601a, read 429b, and splitpath 455a.

In the last few lines we need to read the directory inode from disk again (even though
we did so just nine lines ago, but calling mx_write_dir_entry 453b, may have modified it if a
new block was added to the directory—we must not overwrite this change.

This function uses mx_increase_link_count 457b(), which adds 1 to the number of links for
a given inode:

\(\text{function prototypes 45a}) \equiv \)

\[
\text{int mx_increase_link_count (int device, int inodenr);} \]

It simply reads an inode, increments the i_nlinks entry and writes it back:

\(\text{minix filesystem implementation 420c}) \equiv \)

\[
\text{int mx_increase_link_count (int device, int inodenr) {}
\text{struct minix2_inode inode;}
\text{mx_read_inode (device, inodenr, &inode);} \]
\[
iinode.i_nlinks++; \]
\[
\text{mx_write_inode (device, inodenr, &inode);} \]
\[
\text{return inode.i_nlinks;} \}
\]

Defines:
mx_increase_link_count, used in chunks 456 and 457a.

Uses minix2_inode 442a, mx_read_inode 451b, and mx_write_inode 452a.

12.6.5 The i_mode Entry of the Inode

Each inode contains an i_mode entry that describes the file type and the access permissions. Since we will need to query this information in some of the following functions, we provide a few standard constants that make it easier to check for a specific property.

\(\text{public constants 46a}) \equiv \)

\[
\text{#define S_IRWXU 0000700 } // \text{RWX mask for owner}
\text{#define S_IRUSR 0000400 } // \text{R for owner}
\text{#define S_IWUSR 0000200 } // \text{W for owner}
\text{#define S_IXUSR 0000100 } // \text{X for owner}
\text{#define S_IRWXG 0000070 } // \text{RWX mask for group}
\text{#define S_IRGRP 0000040 } // \text{R for group}
\text{#define S_IWGRP 0000020 } // \text{W for group}
\text{#define S_IXGRP 0000010 } // \text{X for group}
\text{#define S_IRWXO 0000007 } // \text{RWX mask for other}
\text{#define S_IROTH 0000004 } // \text{R for other}
\text{#define S_IWOTH 0000002 } // \text{W for other}
\text{#define S_IXOTH 0000001 } // \text{X for other}
\]
Some of these constants can be used for direct comparisons, for example, in order to check whether a file has the read access bit for the file owner set, you could check whether \((i\_mode \& S\_IRUSR) != 0\). We assume that you’re aware of the standard access permissions on Unix systems—if not, here’s a brief summary:

The standard access permissions encompass nine bits grouped in three groups of three bits each. The first group describes the permissions granted to the file owner who may or may not read, write or execute the file. In the output of the `ls -l` command, these are represented by the characters in the second to fourth column and shown as `rwx` (or some of those letters replaced with `-` if a permission is not set. For example, `rw-` in that place says: owner can read and write, but not execute (those are the standard settings for a document file). The next group describes the corresponding permissions for the members of the group that the file belongs to, and the third group shows permissions for all other users. There are two further interesting bits which can be set: The Set User ID bit, when set on an executable file, changes the effective user ID of a program to the file owner, regardless of who started it. Similarly the Set Group ID bit sets the effective group ID to the file’s group. Owner and group are also stored in the inode (in the `i_uid` and `i_gid` fields).

The last block of constants can be used for identifying the type of a file. This does not refer to properties like “Word document” and “C source file”, but to whether a file is a “classical” file or something else, like a directory, a block or character device file (only the block variety is implemented in Uinx, see Section 12.7), a symbolic link or a socket (not available, either). Since a file cannot be in more than one of these categories, you can check for a specific file type with an expression like \((i\_mode \& S\_IFMT) == S\_IFDIR\). This is an example for using a mask (to mask out the irrelevant bits of the `i_mode` field): We used `S_IFMT` to remove the access permissions. Similarly, `S_IRWXU`, `S_IRWXG` and `S_IRWXO` are masks for the owner, group and others parts of the permissions.

We will explain this in more detail in Chapter 15 where we discuss users and groups and also add permission checking code to the filesystem functions from this chapter.
12.6 The UNIX Implementation of the Minix Filesystem

12.6.6 Opening and Closing Files

We’re getting closer to actually opening files. First, we need to introduce some new data structures, including an internal inode representation and a file status structure. Next up is a function that gets the inode number when given a path. Then we can start work on the implementation of the \texttt{mx\_open} function.

The goal is to provide the kernel functions which will be called when a process uses the standard filesystem functions \texttt{open}, \texttt{read}, \texttt{write} and \texttt{close}. In the Minix subsystem we need the functions \texttt{mx\_open}, \texttt{mx\_read}, \texttt{mx\_write} and \texttt{mx\_close} (and they are called by the corresponding \texttt{u\_} functions).

We have to introduce some new data structures that will help the kernel stay aware of the states of open files:

\textbf{Internal Inode:} This will be an enhanced copy of the inode (as it is stored on the filesystem), but with extra elements, e.g. a reference counter \texttt{refcount} that takes notice of how often the associated file is opened. Whenever we open a file we reserve such an internal inode. All changes to the inode will first be made in the internal copy only; when closing the file we will write the information back to disk. (For immediate writing there will be an \texttt{mx\_sync} function.) One of the new fields is \texttt{clean}: It is set to 1 as long as no changes were made to the internal inode. A change resets it, and calling \texttt{my\_sync} will set it again (after saving the changes to disk).

\begin{verbatim}
struct int_minix2_inode {
  // fields from the external inode
  int ino;    // inode number
  unsigned int refcount; // how many users?
  unsigned short clean;  // 0: changed; 1: unchanged (as on disk)
  short device;  // file resides on which device?
};
\end{verbatim}

Defines:
\begin{itemize}
  \item \texttt{int\_minix2\_inode}, used in chunks 459c, 460a, 464d, 467–71, 473a, 475a, 476b, 484e, and 607b.
\end{itemize}

We create an internal inode table that can store up to 256 records on open files:

\begin{verbatim}
#define MAX_INT_INODES 256

struct int_minix2_inode mx_inodes[MAX_INT_INODES] = {{ 0 }};
\end{verbatim}

Defines:
\begin{itemize}
  \item \texttt{MAX\_INT\_INODES}, used in chunks 459c, 460a, 464d, 467–71, 473a, 475a, 476b, 484e, and 607b.
  \item \texttt{mx\_inodes}, used in chunks 462c, and 464d.
\end{itemize}

\textbf{Local File Descriptor:} The LFD is a non-negative integer returned by \texttt{mx\_open} which the other \texttt{mx\_*} functions use for accessing an open file. It has the same function as...
the process file descriptor in user mode programs, however the local file descriptor is valid in the whole Minix subsystem, whereas each process counts file descriptors separately. When we later allow processes to work with files (via system calls), we will map process-local file descriptors to global file descriptors which are more general than the local ones. For a reminder of how global, local and process file descriptors are connected, go back to Section 12.3.3 (p. 410).

File Status: This is a structure that points to an internal inode. (If there's a null pointer, then this specific file status structure is not in use.) Additionally, it stores the current read/write position and the access mode (see below).

\[
\begin{aligned}
\text{struct mx_filestat} &= \{ \\
\text{\hspace{1em} struct int_minix2_inode *int_inode; } \\
\text{\hspace{1em} int pos; } \\
\text{\hspace{1em} short mode; } \\
\}\;
\end{aligned}
\]

Defines: $\text{mx_filestat}$, used in chunks 461b, 467–70, 475a, and 484e. Uses $\text{int_minix2_inode}$ 459a.

Thus, if a file is opened twice, there will be one internal inode, referenced by the (Minix-subsystem-)local file descriptor, and two $\text{mx_filestat}$ structures, since access mode and read/write position may be different.

We support the following modes for opening:

\[
\begin{aligned}
\text{#define O_RDONLY} &= 0x0000 \quad \text{// read only} \\
\text{#define O_WRONLY} &= 0x0001 \quad \text{// write only} \\
\text{#define O_RDWR} &= 0x0002 \quad \text{// read and write} \\
\text{#define O_APPEND} &= 0x0008 \quad \text{// append mode} \\
\text{#define O_CREAT} &= 0x0200 \quad \text{// create file}
\end{aligned}
\]

Defines: $\text{O_RDONLY}$, used in chunks 190c, 420c, 475a, 488a, 579c, and 582a. $\text{O_WRONLY}$, used in chunks 420a, 470c, 484b, 487a, and 579c. $\text{O_RDWR}$, used in chunks 293b and 579c. $\text{O_CREAT}$, used in chunks 464c, 484b, 487a, 495c, and 576d.

- $\text{O_RDONLY}$ and $\text{O_WRONLY}$ are used when the file shall be used exclusively for reading or writing, respectively.
- Using the $\text{O_RDWR}$ mode allows read and write access.
- The mode $\text{O_APPEND}$ can be supplied in addition to $\text{O_WRONLY}$ (by calculating the mode as $\text{O_WRONLY | O_APPEND}$). In that case all write operations append to the file, and $\text{lseek}$ calls are ignored.
- $\text{O_CREAT}$ allows the creation of new files. Trying to open a non-existing file without $\text{O_CREAT}$ will fail. On the other hand, using $\text{O_CREAT}$ with an already existing file does not change anything (specifically: it does not truncate the file).
Status List: This is an array that holds 256 file status entries, so we allow the Minix subsystem to open up to 256 files simultaneously—the same limit holds for all subsystems since we defined the global file descriptors in a way that allows the local component of the number to lie between 0 and 255.

```c
#define MX_MAX_FILES 256
```

Defines:
- `MX_MAX_FILES`, used in chunks 461b, 463a, 467–70, 475a, 484e, and 607b.

We need a function that looks up a filename in the directory and returns the inode number. Assume we want to look up the path `/etc/passwd`: We can assume that this is an absolute path (because the virtual filesystem layer already took care of that). Then we scan the path until we reach the next `/` (or the string terminator `\0`) which gives us a directory to search for. We can then look up its inode and continue the search with the next directory element. Eventually we reach the last part of the pathname and return its inode number.

The search begins in the volume's root directory that always has inode number 1 on a Minix filesystem.

```c
int mx_pathname_to_ino (int device, const char *path);
```

```c
int mx_pathname_to_ino (int device, const char *path) {
    struct minix2_inode dirinode, inode;
    struct minix_dir_entry dentry;
    char subpath[31]; // maximum name length: 30
    char searchbuf[256];
    char *search = (char*)searchbuf;
    strncpy (search, path, 256); // do not modify original path
    int dirinode_no = 1; // inode number of / directory
    int next_dirinode_no;
    short final = 0; // final = 1 if looking at final part

    search++;
    if (*search == '\0') { return 1; } // searching for / : inode 1
    while (*search != '\0') {
        mx_pathname_to_ino: search loop
        }
    return next_dirinode_no;
};
```

Defines:
- `mx_pathname_to_ino`, used in chunks 456, 461c, 464c, 479, 480, 484c, 487a, 490, and 589d.
- `minix2_inode`, `minix_dir_entry`, and `strncpy`.
Inside the loop we pick the next sub-path and perform the inode lookup. We know that we’re done when search points to the ‘\0’ character (the end of the pathname):

```c
int i = 0;
while (*search != '\0' && *search != '/') {
    subpath[i] = *search;
    search++; i++;
}
subpath[i] = '\0'; // terminate subpath string
```

if (*search == '\0') final = 1; // looking at final part of path

```c
next_dirinode_no = -1;
for (i = 0; i < 32*7; i++) { // max. 32 * 7 entries
    mx_read_dir_entry (device, dirinode_no, i, &dentry);
    if (dentry.inode != 0) {
        if (strequal (dentry.name, subpath)) {
            next_dirinode_no = dentry.inode; // found it!
            break; // leave for loop
        }
    }
}
```

// now next_dirinode_no is either -1 (not found) or points to next step
if (next_dirinode_no == -1) { return -1; } // not found!

```c
dirinode_no = next_dirinode_no;
if (*search != '\0') search++;
else break; // finished, leave while loop
```

We also need two helper functions that give us the index of a free `mx_inodes` and a free `mx_status` entry. The code is similar: We loop over the respective array and check whether an entry is free. `mx_inodes` is free if its refcount element is 0; the file status entry `mx_status` is free if its int_inode element is a NULL pointer.

```c
for (int i = 0; i < MAX_INT_INODES; i++) {
    // returns internal inode no.
    if (mx_inodes[i].refcount == 0) return i;
}
```

Defines: `mx_get_free_inodes_entry`, used in chunk 464d.

Uses `MAX_INT_INODES` and `mx_inodes`.
We're about to show the implementation of the \( \text{mx\_open} \) function that will use many of the functions we've already discussed (and also some new ones). Figure 12.11 shows the function call graph for \( \text{mx\_open} \), so you can see that opening a file is a rather complex task.

Figure 12.11: Minix subsystem functions called by \( \text{mx\_open} \).
12.6.6.1 mx_open

We define two variables

\[
\begin{align*}
\text{int count_open_files} &= 0; \quad \text{// number of open files} \\
\text{int count_int_inodes} &= 0; \quad \text{// number of internal inodes in use}
\end{align*}
\]

\[\text{⟨global variables 92b⟩} + \quad \text{int count_open_files} = 0; \quad \text{// number of open files} \quad \text{int count_int_inodes} = 0; \quad \text{// number of internal inodes in use}\]

Defines:
- \text{count_int_inodes}, used in chunks 466c and 467b.
- \text{count_open_files}, used in chunks 464d, 464c, and 467b.

We start with checking whether the file exists—if it does not, but the \text{O_CREAT} flag was used, we will call \text{mx_creat_empty_file} to make a new file. In both cases \text{ext_ino} is set to the number of the external inode.

\[
\text{short file_already_open = false;}
\]

\[\text{⟨minix filesystem implementation 420c⟩} + \quad \text{int mfd} = \text{mx_get_free_status_entry}();
\]

\[\text{⟨mx_open 464c⟩} \quad \text{int int_ino} = -1; \quad \text{// number of internal inode for this file} \quad \text{int i}; \quad \text{if (count_open_files == 0)} \quad \text{⟨mx_open 464c⟩} \quad \text{int_ino} = 0; \quad \text{// first file to be opened}\]

\[\text{⟨mx_open 464c⟩} + \quad \text{int ext_ino} = \text{mx_pathname_to_ino} (\text{device}, \text{path}); \quad \text{// file not found} \quad \text{if (ext_ino == -1)} \quad \text{// file not found and no O_CREAT}\]

\[\text{⟨mx_open 464c⟩} + \quad \text{if ((oflag} \quad \text{O_CREAT} \quad \text{!= 0)} \quad \text{// file not found and no O_CREAT}\]

\[\text{⟨mx_open 464c⟩} + \quad \text{else} \quad \text{return} \quad (-1); \quad \text{// file not found and no O_CREAT}\]

\[\text{Uses \text{mx_creat_empty_file} 478b, \text{mx_pathname_to_ino} 461d, and \text{O_CREAT} 460b.}\]

In \text{int_ino} we will store the index into the internal inode table. The file may already be open, because another process (or even the same one) opened it earlier. In that case a valid internal inode is in place and can be recycled; otherwise we create a fresh one. We find out if the file is open by checking the \text{ino} and \text{device} fields of all our \text{mx_inodes} array entries.
} else {
    for (i = 0; i < MAX_INT_INODES; i++) {
        if (mx_inodes[i].ino == ext_ino && mx_inodes[i].device == device) {
            // same inode number and same device: this is the same file!
            file_already_open = true;
            int_ino = i;
            break;
        }
    }

    // reached end of the loop: file is not open
    if (int_ino == -1) int_ino = mx_get_free_inodes_entry();
}

if (int_ino == -1) {
    return -1;  // error: no free internal inode available
}

struct int_minix2_inode *inode = &mx_inodes[int_ino];
Uses count_open_files 464a, int_minix2_inode 459a, MAX_INT_INODES 459c, mx_get_free_inodes_entry 462c,
mx_get_free_status_entry 463a, and mx_inodes 459c.

Now int_ino is either set to 0 (we’re just opening the first file), to an index of an
already existing internal inode or to the index of a fresh internal inode (provided by
mx_get_free_inodes_entry 462c) and inode points to that entry.

mfd is the local file descriptor (an index into the mx_status 461b file status table. We can
start filling that entry:

\[
\begin{align*}
&\text{mx_status[mfd].int_inode }= \text{inode}; \\
&\text{mx_status[mfd].pos }= 0; \\
&\text{mx_status[mfd].mode }= \text{oflag};
\end{align*}
\]

if (file_already_open) { (mx_open case: file already open 466b) }
else { (mx_open case: file not open 466a) }

if (oflag & O_APPEND != 0)
    mx_status[mfd].pos = inode->i_size;  // append: set pos to end of file
Uses mx_status 461b and O_APPEND 460b.

mx_status[461b][mfd].pos is set to the current read/write position—normally that is 0 when
freshly opening a file, but if the O_APPEND 460b flag was given, we set it to the file size so
that writing will begin at the end of the file. In mx_status[461b][mfd].mode we remember the
oflag argument of the mx_open 464b call. This will later be used to determine whether it is
acceptable to read from or write to the file. The current read/write position and the mode
field are our reasons for having separate mx_status[461b][] entries, since both can differ for
several opening operations on the file.

Now there are two cases that we need to treat differently. If the file is not yet open, we
copy the inode from disk to memory. We can simply use the mx_read_inode 451b function
because we declared the two inode types (on-disk inode: struct minix2_inode 442a, internal
inode: struct \texttt{int\_minix2\_inode} so that they both start with the same fields—that lets us cast the pointer to the internal inode to a normal inode pointer; here are both types for a quick comparison:

\begin{verbatim}
struct minix2_inode {
    uint16_t i_mode;
    uint16_t i_nlinks;
    uint16_t i_uid;
    uint16_t i_gid;
    uint32_t i_size;
    uint32_t i_atime;
    uint32_t i_mtime;
    uint32_t i_ctime;
    uint32_t i_zone[10];
};

struct int_minix2_inode {
    uint16_t i_mode;
    uint16_t i_nlinks;
    uint16_t i_uid;
    uint16_t i_gid;
    uint32_t i_size;
    uint32_t i_atime;
    uint32_t i_mtime;
    uint32_t i_ctime;
    uint32_t i_zone[10];
    int ino;
    uint32_t refcount;
    uint16_t clean;
    short device;
};
\end{verbatim}

\[466a\] \(\langle\text{mx\_open case: file not open}\ 466a\rangle\equiv\)

\begin{verbatim}
// copy diskinode[ext\_ino] to mx\_inodes[int\_ino]
mx\_read\_inode (device, ext\_ino, (struct minix2\_inode*) inode);
inode->ino = ext\_ino;    // number of external inode
inode->device = device;  // what device is the file on?
inode->refcount = 1;      // one user
inode->clean = true;      // inode is clean (just copied from disk)
\end{verbatim}

Uses \texttt{minix2\_inode 442a, mx\_inodes 459c, and mx\_read\_inode 451b.}

If the file is already open, we have less work: We simply increase the inode's refcount field, because the inode is gaining an additional user:

\[466b\] \(\langle\text{mx\_open case: file already open}\ 466b\rangle\equiv\)

\begin{verbatim}
inode->refcount++;        // file is opened once more
\end{verbatim}

We set clean to true and refcount to 1, and we take note of the external inode and the device. Note that it is important to remember what device the file resides on because the inode number alone is not enough to identify a file when more than one filesystem is mounted.

When opening fails we return -1 (earlier in the code). Otherwise we increment the counters for open files and (possibly) used inodes and return the new local file descriptor. (The variable \texttt{mfd} that we use in this function is short for “Minix file descriptor”.)

\[466c\] \(\langle\text{mx\_open 464c}\rangle\equiv\)

\begin{verbatim}
count\_open\_files++;
if (!file\_already\_open) count\_int\_inodes++;
return mfd;
\end{verbatim}

Uses \texttt{count\_int\_inodes 464a} and \texttt{count\_open\_files 464a.}
Note that we do not check access permissions—this is handled one layer further up in the virtual filesystem.

### 12.6.6.2 mx_close

Closing an open file with

```c
int mx_close (int mfd);
```

is a comparatively simple task. We set the `int_inode` pointer in the file status entry to `NULL` and decrement `count_open_files`. We also check if this was the last user of the internal inode—if that is true and the internal inode is not clean, we write it back to disk. We don’t have to explicitly mark the inode as unused—setting its `refcount` to 0 does the job since that property is what we use to find a free one.

```c
int mx_close (int mfd) {
if (mfd < 0 || mfd ≥ MX_MAX_FILES) return -1; // wrong mfd number
struct mx_filestat *st = &mx_status[mfd];
struct int_minix2_inode *inode = st->int_inode;
if (inode == NULL) return -1; // no open file
short device = inode->device;

// close file
inode->refcount--;
st->int_inode = NULL;

if (inode->refcount == 0) { // usage count down to 0? Then synchronize inode
if (inode->clean == 0) {
   int ext_ino = inode->ino;
   mx_write_inode (device, ext_ino, (struct minix2_inode*) inode);
}
count_int_inodes--;
}

count_open_files--;
return 0;
}
```

Defines: `mx_close`, used in chunks 418a, 467a, 484b, and 487a.
Uses `close 429b, count_int_inodes 464a, count_open_files 464a, int_minix2_inode 459a, minix2_inode 442a, mx_filestat 460a, MX_MAX_FILES 461a, mx_status 461b, mx_write_inode 452a, NULL 46a, and open 429b.

### 12.6.6.3 Helpers: mx_reopen and mx_sync

We provide an `mx_reopen` function that is used when file descriptors are duplicated by `fork` and it is then called by `u_reopen`:
It makes a copy of the \texttt{mx\_status} entry so that the original process and its child can work with different values for the read/write position—if we simply let the child process use the same \texttt{mx\_status} entry, every read or write operation would also update the position for the other process. \texttt{u\_reopen} also increments the usage counter of the file; when one of the two processes closes (its copy of) the file, the counter is reset to the original value.

The \texttt{mx\_sync} function saves changes to the internal inode by writing it back to disk; after that it sets the clean flag.

12.6.6.4 \texttt{mx\_lseek}

Seeking is a very simple operation: Since the file is open we know its size and the current read/write position; so
The Minix Filesystem

will only check if the request makes sense and then update the internal inode. As usual, we support the following three SEEK_* constants for the whence parameter which decide how the offset is to be interpreted:

```
define SEEK_SET 0  // absolute offset
define SEEK_CUR 1  // relative offset
define SEEK_END 2  // EOF plus offset
```

Defines:
SEEK_CUR, used in chunks 469c and 498a.
SEEK_END, used in chunks 293b, 469c, and 498a.
SEEK_SET, used in chunks 233b, 293, 294, 469c, and 498a.

When the file is opened in append mode, we must not change the position; otherwise we either set the position, add the offset to the current location or add it to the end of file position. For the last two cases negative values are OK.

```
int mx_lseek (int mfd, int offset, int whence) {
    if (mfd < 0 || mfd ≥ MX_MAX_FILES) return -1;  // wrong mfd number
    struct mx_filestat *st = &mx_status[mfd];
    struct int_minix2_inode *inode = st->int_inode;
    if (inode == NULL) return -1;  // no open file
    if (whence < 0 || whence > 2) return -1;  // wrong lseek option
    if ((st->mode & O_APPEND) != 0) return st->pos;  // append mode, ignore lseek

    switch (whence) {
        case SEEK_SET: st->pos = offset; break;  // set absolute
        case SEEK_CUR: st->pos += offset; break;  // relative to current loc.
        case SEEK_END: st->pos = inode->i_size + offset;  // relative to EOF
    }

    if (st->pos < 0) st->pos = 0;  // sanity check
    return st->pos;
}
```

Defines:
mx_lseek, used in chunks 418a and 469a.

Uses int_minix2_inode 459a, mx_filestat 460a, MX_MAX_FILES 461a, mx_status 461b, NULL 46a, O_APPEND 460b, SEEK_CUR 469b, SEEK_END 469b, and SEEK_SET 469b.

12.6.7 Reading and Writing

With all these preparations we can now approach the read and write operations which work on open files.
12.6.7.1 mx_read

We start with the function

\[
\text{int mx_read (int mfd, void *buf, int nbyte);}
\]

that reads \(nbyte\) bytes from an open file identified by \(mfd\) into a buffer \(buf\).

Since \(mx_read\) is a bit longer, we use a code chunk \(\langle \text{mx_read} \rangle\) for displaying the code:

\[
\text{int mx_read (int mfd, void *buf, int nbyte) \{ }
\]

\[
\text{\langle \text{mx_read} \rangle}
\]

\[
\text{\}}
\]

Defines:

- \(\text{mx_read}\), used in chunks 414b and 470a.

We start with the usual variable initialization so that we have access to both the internal inode and the file status entry. If \(mfd\) has an invalid value or we attempt to read a file in write-only or append mode, we return \(-1\) at once.

\[
\text{if (mfd < 0 || mfd \geq MX_MAX_FILES) return -1; // wrong mfd number}
\]

\[
\text{struct mx_filestat *st = \&mx_status[mfd];}
\]

\[
\text{struct int_minix2_inode *inode = st->int_inode;}
\]

\[
\text{short device = inode->device;}
\]

\[
\text{if (inode == NULL) return -1; // no open file}
\]

\[
\text{if (st->mode == O_WRONLY || st->mode == O_APPEND)}
\]

\[
\text{\quad return -1; // reading is forbidden}
\]

Uses \(\text{int_minix2_inode}\), \(\text{mx_filestat}\), \(\text{MX_MAX_FILES}\), \(\text{mx_status}\), \(\text{NULL}\), \(O_WRONLY\), and \(O_APPEND\).

Next we look at the current read/write position and determine which logical blocks of the file must be read—even if we want just a single byte from the file, we must read a whole block since that’s the only way that we can access the hardware. (With “logical block” we mean the enumeration of the file’s blocks. A file always starts with logical block 0 (unless it is empty).

Note that as a worst case, even reading two bytes can result in reading two blocks, if those bytes are placed precisely on a block boundary.

\[
\text{int startbyte = st->pos;}
\]

\[
\text{if (startbyte \geq inode->i_size) \{ return 0; \} // nothing to read}
\]

\[
\text{int endbyte = st->pos + nbyte - 1;}
\]

\[
\text{if (endbyte \geq inode->i_size) \{}
\]

\[
\text{nbyte -= (endbyte - inode->i_size + 1);}
\]

\[
\text{endbyte = inode->i_size - 1;}
\]

With \(\text{startbyte}\) and \(\text{endbyte}\) set, we can easily calculate the logical blocks:
We need to loop over all the logical blocks and read them. In order to read a logical block \( \text{curblock} \) from the file we must find out where it is located on the device (i.e., find the physical block where it is stored); we put the lookup of that block number into a separate function

\[
\text{int fileblocktozone (int device, int blockno, struct int_minix2_inode *inode);}\]

that we will implement afterwards.

\[
\text{int read_bytes = 0;}\]
\[
\text{while (nbyte > 0) { } }\]
\[
\text{int zone = fileblocktozone (device, curblock, inode);} \quad \text{// where is the block?}\]
\[
\text{if (zone == -1) { } }\]
\[
\text{printf ("ERROR, fileblocktozone() = -1\n");}\]
\[
\text{return -1;}\]
\[
\text{ };\]

\[
\text{byte block[BLOCK_SIZE]; readblock (device, zone, (byte*) block);}\]
\[
\text{int offset, length;}\]
\[
\text{if (curblock == startblock) { } }\]
\[
\text{offset = startbyte \% BLOCK_SIZE;}\]
\[
\text{length = MIN (nbyte, BLOCK_SIZE - offset);}\]
\[
\text{ }\]
\[
\text{else { } }\]
\[
\text{offset = 0;}\]
\[
\text{length = MIN (nbyte, BLOCK_SIZE);}\]
\[
\text{ }\]
\[
\text{ }}\]
\[
\text{memcpy (buf, block+offset, length);}\]
\[
\text{nbyte -= length;}\quad \text{buf += length;}\]
\[
\text{read_bytes += length;}\quad \text{curblock++;}\]
\[
\text{st->pos += length;} \quad \text{// update current location in inode}\]
\[
\text{ }\]
\[
\text{return read_bytes;} \quad \text{// return the read bytes, might be != nbyte}\]

Uses BLOCK_SIZE 440a, fileblocktozone 473a, memcpy 596c, MIN 471d, printf 601a, and readblock 506b.

This code uses the MIN 471d macro that we have not defined yet:

\[
\#define \text{MIN(a,b) \{(a) \leq (b) ? (a) : (b)}\]
\[
\#define \text{MAX(a,b) \{(a) \geq (b) ? (a) : (b)}\]

Defines:

MIN, used in chunks 471c, 475c, 496d, and 497.
Single and double indirection in the Minix filesystem

```c
struct minix2_inode {
    uint16_t i_mode;
    uint16_t i_nlinks;
    uint16_t i_uid;
    uint16_t i_gid;
    uint32_t i_size;
    uint32_t i_atime;
    uint32_t i_mtime;
    uint32_t i_ctime;
    uint32_t i_zone[10];
};
```

Legend:
- Points to data block
- Points to (single/double/triple) indirection block

Figure 12.12: A Minix inode stores seven direct block numbers and two block numbers for single and double indirection blocks; i_zone[9] is unused.

Now we need to show how to translate a logical block number to a physical block number (or zone number). This may require looking at single and double indirection blocks. The Minix V2 filesystem uses four byte long integers as zone addresses, so one indirection block has space for `BLOCK_SIZE / 4 = 256` such addresses. We’ll define this number as a constant:

```
#define BLOCKADDRESSES_PER_BLOCK (BLOCK_SIZE / 4)
```

Writing fileblocktozone is straightforward if we know how single and double indirection blocks are organized. Figure 12.12 once more shows how Minix uses indirect blocks. You have already seen a bigger version of that figure on page 82, here we only show the single and double indirection that we implement for U\text{L}\text{IX}; triple indirection was
not part of the original implementation in Minix either, but the inode entry \texttt{i\_zone[9]} is reserved for triple indirection.

Thus, the physical block numbers of the first seven logical blocks (number 0–6) can be found directly in the inode. For the 256 blocks with numbers 7–262 we must load the indirection block (whose address is in \texttt{i\_zone[7]}, and any block number beyond 262 requires us to first load the double indirection block (via \texttt{i\_zone[8]}) and then search for the address of the right single indirection block, so locating such a block always requires reading two indirection blocks which is one of the reasons why it is so helpful to use a caching mechanism: When reading consecutive blocks beyond block number 262, the indirection blocks will remain in the cache once the first of those blocks has been accessed.

\begin{verbatim}
(minix filesystem implementation 420c)+\equiv
\text{int fileblocktozone(int device, int blockno, struct int_minix2_inode *inode) }
\{ int zone; int *zone_ptr; byte indirect_block[BLOCK_SIZE];
    if (blockno < 7) {
        // the first 7 zone numbers (0..6) are in the inode:
        zone = inode->i_zone[blockno];
    } else if (blockno >= 7 && blockno < 7+BLOCKADDRESSES_PER_BLOCK) {
        // inode holds the address of an indirection block
        (fileblocktozone: single indirect 473b)
    } else {
        // inode holds the address of a double indirection block
        (fileblocktozone: double indirect 474a)
    }
    return zone;
\}
\end{verbatim}

Defines:
\texttt{fileblocktozone}, used in chunks 471c, 475c, and 484e.
Uses \texttt{BLOCK\_SIZE 440a, BLOCKADDRESSES\_PER\_BLOCK 472, and int\_minix2\_inode 459a.}

For the two indirection cases (singly indirect, doubly indirect) we provide two code chunks which are increasingly complex. Single indirection works like this:

\begin{verbatim}
(fileblocktozone: single indirect 473b)≡
\text{int indirect_zone = inode->i_zone[7];
    if (indirect_zone == 0) {
        return -2;  // no indirection block found
    }
    readblock(device, indirect_zone, (byte *) indirect_block);
    zone_ptr = (int *) indirect_block;
    zone_ptr += (blockno - 7);
    zone = *zone_ptr;
}
\end{verbatim}

Here we set \texttt{zone}\_ptr to the start address of the loaded indirection block. Then we need to add an offset to find the right block number inside that block. We don’t add \texttt{blockno} but \texttt{blockno - 7} because the first seven block addresses are already found in the inode and not repeated in the indirection block which starts with the block number of block 7.

Resolving a double indirection works similarly, but consists of two steps.
• First, \((\text{blockno} - 7 - \text{BLOCKADDRESSES\_PER\_BLOCK}) / \text{BLOCKADDRESSES\_PER\_BLOCK})\) is the index into the first indirection block—instead of \(\text{blockno} - 7\) (as in the single indirection case) we must also subtract \(\text{BLOCKADDRESSES\_PER\_BLOCK}\) since the first \(7 + \text{BLOCKADDRESSES\_PER\_BLOCK}\) blocks can be found via the direct addresses and the single indirection block. Then we also need to divide by \(\text{BLOCKADDRESSES\_PER\_BLOCK}\) as each address in the first indirection block points to a whole block of addresses.

• In the second step, we take \((\text{blockno} - 7) \mod \text{BLOCKADDRESSES\_PER\_BLOCK}\) as an index into the second indirection block: Note that the equation

\[
(\text{blockno} - 7 - \text{BLOCKADDRESSES\_PER\_BLOCK}) \mod \text{BLOCKADDRESSES\_PER\_BLOCK} = (\text{blockno} - 7) \mod \text{BLOCKADDRESSES\_PER\_BLOCK}
\]

holds; the left side is the original formula, but \(n \mod n = 0\) for all \(n\).

\[
\begin{align*}
\text{double indirect zone} & = \text{inode->i\_zone[8]}; \\
\text{if (double indirect zone == 0) \{ return -2; // no double indirection block found \}} \\
\text{readblock} \ (\text{device, double indirect zone, \{byte *\} indirect block}); \\
\text{zone ptr} & = \text{(int \*) indirect block}; \\
\text{zone ptr} & += (\text{blockno} - 7 - \text{BLOCKADDRESSES\_PER\_BLOCK}) / \text{BLOCKADDRESSES\_PER\_BLOCK}; \\
\text{int indirect zone} & = *\text{zone ptr}; \\
\text{readblock} \ (\text{device, indirect zone, \{byte *\} indirect block}); \\
\text{zone ptr} & = \text{(int \*) indirect block}; \\
\text{zone ptr} & += (\text{blockno} - 7) \mod \text{BLOCKADDRESSES\_PER\_BLOCK}; \\
\text{zone} & = *\text{zone ptr}; \\
\end{align*}
\]

Uses \text{BLOCKADDRESSES\_PER\_BLOCK} and \text{readblock}.

12.6.7.2 \textit{mx\_write}

The \textit{mx\_write} function works similar to \textit{mx\_read}, but it must also read blocks which are only partly modified so that writing the block does not erase the parts which are not modified.

\[
\text{int mx\_write} \ (\text{int mfd, void *buf, int nbyte});
\]

The structure is the same as in \textit{mx\_read}, and again we use a separate code chunk \textit{mx\_write}.

\[
\begin{align*}
\text{int mx\_write} \ (\text{int mfd, void *buf, int nbyte}) \{
\text{\{mx\_write 475a\}} \\
\}
\end{align*}
\]

Defines:
\textit{mx\_write}, used in chunks 415a, 474b, 484b, and 487a.
We start with some checks:

\[
\text{\texttt{mx_write}}\text{\texttt{475a}}
\]

\[
\text{\texttt{475c}} \quad \text{if} (\text{mfd < } 0 \text{ || mfd } \geq \text{ MX_MAX_FILES}) \text{ return -1; // wrong mfd number}
\]

\[
\text{\texttt{475a}} \quad \text{\text{\texttt{struct mx_filestat *st} = &\text{\texttt{mx_status}[mfd];}}}
\]

\[
\text{\texttt{475a}} \quad \text{\text{\texttt{struct int_minix2_inode *inode = st->int_inode;}}}
\]

\[
\text{\texttt{475a}} \quad \text{\text{\texttt{short device = inode->device;}}}
\]

\[
\text{\texttt{475a}} \quad \text{\text{\texttt{if (inode == NULL) \text{ return -1; // no open file}}}\text{\texttt{475b}}}
\]

\[
\text{\texttt{475b}} \quad \text{\text{\texttt{if (st->mode == O_RDONLY) return -1; // cannot write to read-only file}}}\text{\texttt{475c}}
\]

Uses \texttt{int_minix2_inode 459a, mx_filestat 460a, MX_MAX_FILES 461a, mx_status 461b, NULL 46a, O_RDONLY 460b, open 429b, read 429b, and write 429b.}

The calculation of start and end positions (and blocks) is the same as well, but without the checks: writing does not require the data to be available.

\[
\text{\texttt{mx_write}}\text{\texttt{475a}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int startbyte = st->pos;}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int endbyte = startbyte + nbyte - 1;}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int startblock = startbyte / BLOCK_SIZE;}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int endblock = endbyte / BLOCK_SIZE;}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int curblock = startblock;}}
\]

Uses \texttt{BLOCK_SIZE 440a.}

The code for actually writing the blocks is just a little more complex than that of the \texttt{mx_read} function:

\[
\text{\texttt{mx_write}}\text{\texttt{475a}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{byte block[BLOCK_SIZE];}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int offset, length;}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int written_bytes = 0;}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{while (nbyte > 0) \{}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{int zone = fileblocktozone (device, curblock, inode);}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{if (zone == -2 || zone == 0 \{}}
\]

\[
\text{\texttt{475c}} \quad \text{\texttt{zone = mx_create_new_zone (device, curblock, inode); // block doesn't yet exist}}}\text{\texttt{475d}}
\]

\[
\text{\texttt{475d}} \quad \text{\texttt{\}}}\text{\texttt{475e}}
\]

\[
\text{\texttt{475e}} \quad \text{\texttt{if (zone == -1) \text{ return -1;}}}\text{\texttt{475f}}
\]

\[
\text{\texttt{475f}} \quad \text{\texttt{if (curblock == startblock) \{}}
\]

\[
\text{\texttt{475f}} \quad \text{\texttt{offset = startbyte % BLOCK_SIZE;}}
\]

\[
\text{\texttt{475f}} \quad \text{\texttt{length = MIN (nbyte, BLOCK_SIZE - offset);}}\text{\texttt{475g}}
\]

\[
\text{\texttt{475g}} \quad \text{\texttt{\}}}\text{\texttt{475h}}
\]

\[
\text{\texttt{475h}} \quad \text{\texttt{else \{}}
\]

\[
\text{\texttt{475h}} \quad \text{\texttt{offset = 0;}}
\]

\[
\text{\texttt{475h}} \quad \text{\texttt{length = MIN (nbyte, BLOCK_SIZE);}}\text{\texttt{475i}}
\]

\[
\text{\texttt{475i}} \quad \text{\texttt{\}}}\text{\texttt{475j}}
\]

\[
\text{\texttt{475j}} \quad \text{\texttt{if (offset ! = 0 || length ! = BLOCK_SIZE) \{}}
\]

\[
\text{\texttt{475j}} \quad \text{\texttt{\}}}\text{\texttt{475k}}
\]

\[
\text{\texttt{475k}} \quad \text{\texttt{\}}}\text{\texttt{475l}}
\]

\[
\text{\texttt{475l}} \quad \text{\texttt{memcpy (block+offset, buf, length);}}
\]

\[
\text{\texttt{475l}} \quad \text{\texttt{writeblock (device, zone, (byte*) block);}}
\]

\[
\text{\texttt{475l}} \quad \texttt{}}
\]
nbyte -= length;(buf += length;written_bytes += length; curblock++;

st->pos += length;
if (st->pos > inode->i_size) inode->i_size = st->pos; // update size
}

inode->i_mtime = system_time; // update mtime
mx_write_inode (device, inode->ino, (struct minix2_inode*)inode);
return written_bytes;

Uses BLOCK_SIZE 440a, fileblocktozone 473a, memcpy 596c, MIN 471d, minix2_inode 442a, mx_create_new_zone 476b,
mx_write_inode 452a, readblock 506b, system_time 338a, and writeblock 507c.

The function uses

476a) \langle function prototypes 45a \rangle +\equiv \langle 44a \rangle <474b 478a>
int mx_create_new_zone (int device, int blockno, struct int_minix2_inode *inode);

which requests a new block and inserts it in the inode’s block list: The argument blockno is the logical block number (as seen from the file). In the function, zone is the physical block (zone) number that is assigned to the logical block.

476b) \langle minix filesystem implementation 420c \rangle +\equiv \langle 440b \rangle <474c 478b>
int mx_create_new_zone (int device, int blockno, struct int_minix2_inode *inode) {
int zone = mx_request_block (device); // new data block
if (zone == -1) {
printf("ERROR: cannot reserve data block; disk full\n");
return -1;
}
int indirect_zone, double_indirect_zone;
int *zone_ptr;
byte indirect_block[BLOCK_SIZE] = { 0 };
byte double_indirect_block[BLOCK_SIZE] = { 0 };
if (blockno < 7) {
\langle create new zone: direct 477a \rangle
} else if (blockno >= 7 && blockno < 7+BLOCKADDRESSES_PER_BLOCK) {
\langle create new zone: single indirect 477b \rangle
} else {
\langle create new zone: double indirect 477c \rangle
}
return zone;
}

Defines:
mx_create_new_zone, used in chunks 475c and 476a.
Uses BLOCK_SIZE 440a, BLOCKADDRESSES_PER_BLOCK 472, int_minix2_inode 459a, mx_request_block 448,
and printf 601a.

with the following three cases for direct, indirect and double indirect blocks—they are similar to the three cases in the fileblocktozone473a function. If we deal with a direct zone, we can directly enter the zone number in the inode:
12.6 The ULIx Implementation of the Minix Filesystem

\[\text{create new zone: direct 477a}\] ≡

// the first 7 zone numbers (0..6) are in the inode:
inode->i_zone[blockno] = zone;

In case of single indirection, we may have to create the indirection block which requires another call of \text{mx_request_block}\text{448}.

\[\text{create new zone: single indirect 477b}\] ≡
indirect_zone = inode->i_zone[7];

// if there is no indirection block yet, create it
if (indirect_zone == 0) {
  // no indirection block found
  indirect_zone = mx_request_block (device);  // data block for indirections
  if (indirect_zone == -1) {
    mx_clear_zmap_bit (device, zone);  // undo reservation of data block
    return -1;
  }
  inode->i_zone[7] = indirect_zone;
} else {
  // indirection block exists: read it
  readblock (device, indirect_zone, (byte *) indirect_block);
}

zone_ptr = (int *) indirect_block;
zone_ptr += (blockno - 7);
*zone_ptr = zone;  // write information about new data block
writeblock (device, indirect_zone, (byte *) indirect_block);

Uses \text{mx_clear_zmap_bit 446}, \text{mx_request_block 448}, \text{read 429b}, \text{readblock 506b}, and \text{writeblock 507c}.

Finally, in the case of double indirection, the worst that can happen is that we need to create both the first and the second indirection block. In both cases (single and double indirection) we use the same offset calculations as in \text{fileblocktozone}\text{473a}().

\[\text{create new zone: double indirect 477c}\] ≡

double_indirect_zone = inode->i_zone[8];

// if there is no double indirection block yet, create it
if (double_indirect_zone == 0) {
  // no double indirection block found
  double_indirect_zone = mx_request_block (device);  // data block for 2x indir.
  if (double_indirect_zone == -1) {
    mx_clear_zmap_bit (device, zone);  // undo reservation of data block
    return -1;
  }
  inode->i_zone[8] = double_indirect_zone;
} else {
  // indirection block exists: read it
  readblock (device, double_indirect_zone, (byte *) double_indirect_block);
}
zone_ptr = (int *) double_indirect_block;
zone_ptr += (blockno - 7 - BLOCKADDRESSES_PER_BLOCK) / BLOCKADDRESSES_PER_BLOCK;
indirect_zone = *zone_ptr;

// if there is no indirect block yet, create it
if (indirect_zone == 0) {
    // no indirect block found
    indirect_zone = mx_request_block (device); // data block for indirections
    if (indirect_zone == -1) {
        mx_clear_zmap_bit (device, zone); // undo reservation of data block
        return -1;
    }
}

// write to first level indirection block
*zone_ptr = indirect_zone;
writeblock (device, double_indirect_zone, (byte *) double_indirect_block);
} else {
    // indirection block exists: read it
    readblock (device, indirect_zone, (byte *) indirect_block);
}

zone_ptr = (int *) indirect_block;
zone_ptr += (blockno - 7) % BLOCKADDRESSES_PER_BLOCK;

*zone_ptr = zone; // write information about new data block
writeblock (device, indirect_zone, (byte *) indirect_block);

What’s still missing is a way to create a new (empty) file. The function

\[
\text{int \ mx\_creat\_empty\_file (int device, const char *path, int mode);}\]

requests a new inode, fills it with the appropriate data and writes it to disk. Then it writes a link into the directory that will hold the file.

```c
int mx_creat_empty_file (int device, const char *path, int mode) {
    int inodenr = mx_request_inode (device);
    struct minix2_inode inode = { 0 };
    inode.i_size = 0;
    inode.i_atime = inode.ictime = inode.ictime = system_time;
    inode.i_uid = thread_table[current_task].uid;
    inode.i_gid = thread_table[current_task].gid;
    inode.i_nlinks = 0;
    inode.i_mode = S_IFREG | mode;
    mx_write_inode (device, inodenr, &inode);
    mx_write_link (device, inodenr, path); // create directory entry
    return inodenr;
}
```
12.6 The Unix Implementation of the Minix Filesystem

Defines:
- `mx_creat_empty_file`, used in chunks 464c and 478a.
- `current_task`, `gid`, `minix2_inode`, `mx_request_inode`, `mx_write_inode`, `mx_write_link`, `S_IFREG`, `system_time`, `thread_table`, and `uid`.

12.6.8 Linking and Unlinking

Unix systems have no delete or erase system calls for files—instead there is an `unlink` system call which removes a directory entry (it deletes the link from a filename to an inode in that directory). Only if the last name was deleted, `unlink` will also delete the file, which means freeing all data blocks and the inode.

The opposite operation is creating a hardlink: This creates a new name (a new link from a filename to an inode in some directory).

Both operations modify an inode's link count: That is where the filesystem keeps track of how many names were given to a file.

We will start by showing two helper functions which can check whether a file exists and whether it is a directory, then we implement the `link` operation since it is the simpler one (of `link` and `unlink`).

\[(\text{function prototypes})\]
\[
\begin{align*}
\text{boolean } & \text{mx_file_exists (int device, const char *path);} \\
\text{boolean } & \text{mx_file_is_directory (int device, const char *path);} \\
\text{int } & \text{mx_link (int device, const char *path1, const char *path2);} \\
\end{align*}
\]

\[(\text{minix filesystem implementation})\]
\[
\begin{align*}
\text{boolean } & \text{mx_file_exists (int device, const char *path) \{} \\
& \text{if (mx_pathname_to_ino (device, path) == -1) return false;} \\
& \text{return true;} \\
\end{align*}
\]

\[
\begin{align*}
\text{boolean } & \text{mx_file_is_directory (int device, const char *path) \{} \\
& \text{int ino = mx_pathname_to_ino (device, path);} \\
& \text{if (ino == -1) return false; \hspace{1cm} // does not exist} \\
& \text{struct minix2_inode inode;} \\
& \text{mx_read_inode (device, ino, &inode);} \\
& \text{if ((inode.i_mode & S_IFDIR) == 0) return false; \hspace{1cm} // no directory} \\
& \text{return true;} \\
\end{align*}
\]

Defines:
- `mx_file_exists`, used in chunks 456 and 480.
- `mx_file_is_directory`, used in chunk 480a.
- `minix2_inode`, `mx_pathname_to_ino`, `mx_read_inode`, `S_IFDIR`, and `mx_write_inode`.

12.6.8.1 `mx_link`

The `mx_link` function checks both paths and writes the link. Note that this implementation lets users create hard links of directories which is normally forbidden. We do check
the condition but only print a warning because it is interesting to “play” with hard-linked directories.

```c
int mx_link (int device, const char *path1, const char *path2) {
    // check path1 exists
    if (!mx_file_exists (device, path1)) return -1; // does not exist

    // check path1 is not a directory
    if (mx_file_is_directory (device, path1)) {
        printf ("ln: warning: %s is a directory. This option will be removed.\n");  
    }

    // check path2 does NOT exist
    if (mx_file_exists (device, path2)) {
        return -1; // path2 already exists; no forced linking
    }

    // everything ok now
    int ino = mx_pathname_to_ino (device, path1);
    mx_write_link (device, ino, path2); // updates link count
    return 0;
}
```

Defines:
- `mx_link`, used in chunks 419a and 479a.
Uses `mx_file_exists` 479b, `mx_file_is_directory` 479b, `mx_pathname_to_ino` 461d, `mx_write_link` 456, and `printf` 601a.

### 12.6.8.2 `mx_unlink`

Unlinking is similar as long as at least one filename (one link) remains. If none remains, the data blocks of the file and the inode must be freed.

```c
int mx unlink (int device, const char *path);
```

```c
int mx_unlink (int device, const char *path) {
    char ind_block [BLOCK_SIZE], double_ind_block [BLOCK_SIZE];
    struct minix2_inode inode;

    // check if path exists
    if (!mx_file_exists (device, path)) {
        printf ("rm: file does not exist\n");
        return -1; // error: path does not exist
    }

    // get inodes of file and directory
    int inodenr = mx_pathname_to_ino (device, path);
    struct minix2_inode inode;
```
mx_read_inode (device, inodenr, &inode);
char dir[256], base[30];
splitpath (path, dir, base);                  // split path into dir and base
int dir_inodenr = mx_pathname_to_ino (device, dir);

// delete entry in directory
boolean found = false;
for (int i = 0; i < 32*7; i++) {
    mx_read_dir_entry (device, dir_inodenr, i, &dentry);
    if (dentry.inode==inodenr && strequal (dentry.name, base)) {
        dentry.inode = 0;
        memset (dentry.name, 0, 30);
        found = true;
        mx_write_dir_entry (device, dir_inodenr, i, &dentry);
        break;   // search finished
    }
}
if (found==false) { return -1; }   // error: not found in directory

inode.i_nlinks--;             // one name less
if (inode.i_nlinks == 0) { (free this inode) }
mx_write_inode (device, inodenr, &inode);
return 0;
}

Defines:
mx_unlink, used in chunks 418b and 480b.
Uses BLOCK_SIZE 440a, memset 596c, minix2_inode 442a, minix_dir_entry 452b, mx_file_exists 479b,
        mx_pathname_to_ino 461d, mx_read_dir_entry 453b, mx_read_inode 451b, mx_write_dir_entry 453b,
mx_write_inode 452a, printf 601a, splitpath 455a, and strequal 596a.

We must take care of the case when the last link has been removed—then we have an
inode with a reference count of 0, and that means, the file is truly to be deleted: We need
to mark its data blocks as free (including an indirection block, if it exists) and also mark
the inode as free. We show this action in four separate steps:

\[ \langle \text{free this inode} \rangle \equiv \]  \hspace{1cm} (480c)  \hspace{1cm} [481a]
\langle \text{free this inode: (1) direct blocks} \rangle \equiv \hspace{1cm} (481a)  \hspace{1cm} [481b]
\langle \text{free this inode: (2) single indirect blocks} \rangle \equiv \hspace{1cm} (481a)  \hspace{1cm} [481b]
\langle \text{free this inode: (3) double indirect blocks} \rangle \equiv \hspace{1cm} (481a)  \hspace{1cm} [481b]
\langle \text{free this inode: (4) inode itself} \rangle \equiv \hspace{1cm} (481a)  \hspace{1cm} [481b]

for (int i = 0; i < 7; i++) {
    if (inode.i_zone[i] != 0) {
        mx_clear_zmap_bit (device, inode.i_zone[i] - 33); // mark data block as free
        inode.i_zone[i] = 0;
    }
}

Uses mx_clear_zmap_bit 446.
For single indirection, we clear both the zone map entries for the indirection block itself and all the blocks it points to.

\[ (\text{free this inode: (2) single indirect blocks} 482a) \equiv \]

\[
\begin{align*}
\text{if} (\text{inode}.i\_zone[7] \neq 0) &\{ \\
\text{readblock} &\text{(device, inode}.i\_zone[7], \text{ind} \_ \text{block}); \\
\text{unsigned int} &\ast \text{zoneno}; \\
\text{zoneno} &\equiv (\text{unsigned int}*)\text{ind} \_ \text{block}; \quad \text{// cast to uint*} \\
\text{int} &\text{count} = 0; \\
\text{while} (*\text{zoneno} \neq 0 \&\& \text{count} < 256) &\{ \\
\text{mx\_clear\_zmap\_bit} &\text{(device, *zoneno - 33)}; \quad \text{// mark data block as free} \\
\text{zoneno} &++; \\
\text{count} &++;
\} \\
\text{mx\_clear\_zmap\_bit} &\text{(device, inode}.i\_zone[7] - 33); \quad \text{// mark indir. block as free} \\
\text{inode}.i\_zone[7] &\equiv 0;
\}
\end{align*}
\]

Uses \texttt{mx\_clear\_zmap\_bit} 446 and \texttt{readblock} 506b.

And in case of double indirection, there are even more blocks to mark as free:

\[ (\text{free this inode: (3) double indirect blocks} 482b) \equiv \]

\[
\begin{align*}
\text{if} (\text{inode}.i\_zone[8] \neq 0) &\{ \\
\text{readblock} &\text{(device, inode}.i\_zone[8], \text{double} \_ \text{ind} \_ \text{block}); \\
\text{unsigned int} &\ast \text{ind\_zoneno}; \\
\text{ind\_zoneno} &\equiv (\text{unsigned int}*)\text{double} \_ \text{ind} \_ \text{block}; \quad \text{// cast to uint*} \\
\text{int} &\text{count1} = 0; \\
\text{int} &\text{count2}; \\
\text{while} (*\text{ind\_zoneno} \neq 0 \&\& \text{count1} < 256) &\{ \\
\text{readblock} &\text{(device, *ind\_zoneno, ind} \_ \text{block}); \\
\text{unsigned int} &\ast \text{zoneno}; \\
\text{zoneno} &\equiv (\text{unsigned int}*)\text{ind} \_ \text{block}; \quad \text{// cast to uint*} \\
\text{count2} &\equiv 0; \\
\text{while} (*\text{zoneno} \neq 0 \&\& \text{count2} < 256) &\{ \\
\text{mx\_clear\_zmap\_bit} &\text{(device, *zoneno - 33)}; \quad \text{// mark data block as free} \\
\text{zoneno} &++; \\
\text{count2} &++;
\} \\
\text{mx\_clear\_zmap\_bit} &\text{(device, *ind\_zoneno - 33)}; \quad \text{// mark indir. block as free} \\
\text{ind\_zoneno} &++; \\
\text{count1} &++;
\}
\end{align*}
\]

\[
\begin{align*}
\text{mx\_clear\_zmap\_bit} &\text{(device, inode}.i\_zone[8] - 33); \quad \text{// mark double ind. block free} \\
\text{inode}.i\_zone[8] &\equiv 0;
\}
\end{align*}
\]

Uses \texttt{mx\_clear\_zmap\_bit} 446 and \texttt{readblock} 506b.
12.6 The Ulix Implementation of the Minix Filesystem

Last, we free the inode:

\[
\begin{align*}
&\text{free this inode: (4) inode itself} \quad \text{(481a)} \\
&\text{mx_clear_imap_bit (device, inodenr)}; \quad \text{(483)} \\
\end{align*}
\]

Uses \text{mx_clear_imap_bit} 446.

12.6.8.3 \text{mx}\_symlink

Unix systems also know a second type of link, the \textit{symbolic} or \textit{soft link} (short: \textit{symlink}). While this name reminds of the (hard) links for which we have just provided the implementation, and even the same Unix tool (\textit{ln}) handles both hard and soft links, these two link types have nothing in common.

A symbolic link is a special file which contains a path name as data. When you disable the treatment of symbolic links on a Unix system and try to read such a file, all you get is the stored path name: Figure 12.13 shows how Ulix displayed the content of a symbolic link before symlinks were implemented: \texttt{ulix.symlink} was created by executing

\texttt{ln -s ulix.h ulix.symlink}

on a Linux system which had loop-mounted the Minix filesystem: As you can see from the letter l at the start of the file entry, Ulix already recognized the file type but did not know better than to output the contents of the file's first data block.

\begin{verbatim}
Ulix-i386 0.08 Build: Mon Jul 15 22:30:00 CEST 2013 -
Physical RAM (64 MB) mapped to 0x00000000-0x83FFFFFF.
Initialized ten terminals (press [Alt-1] to [Alt-0])
PDC: fdd is 1.44M, fdb is 1.44M
Starting five shells on tty0..tty4. Type exit to quit.
Ulix Usermode Shell. Commands: help, ps, fork, ls, cat, head, cp, diff, sh, hexdump, kill, loop, test, brk, cd, ln, rm, pwd, touch, exit
Press [Shift+Esc] to launch kernel mode shell (reboot to get back here)
esser@ulix(2):/$ ls

directory / is in inode 1
  1 drwxr-xr-x 3 1000 1000 192 ..
  1 drwxr-xr-x 3 1000 1000 192 ...
  2 -rw-r--r-- 1 1000 1000 16384 sh
  3 -rw-r--r-- 1 1000 1000 3794 ulix.h
  4 drwxr-xr-x 2 1000 1000 128 sub
  6 lrwxrwxrwx 1 1000 1000 6 ulix.symlink

esser@ulix(2):/$ hexdump ulix.symlink
absolute path: ulix.symlink
  0000  75 6c 69 70 2c 60 68 0a
esser@ulix(2):/$ ulix.h

Ulix-i386 0.08 tty0 FF=3B70 AS=0001 00:00:21
\end{verbatim}

Figure 12.13: Ulix version 0.08 displays the contents of a symbolic link—in that version symbolic links were not yet implemented. (The image was inverted for better readability.)

As you can see, creating a symlink is easy: We just write the link target into a data block and mark the file as symlink:
12 Filesystems

```c
int mx_symlink (int device, char *path1, char *path2);
```

12.6.9 Truncating Files

Sometimes is is necessary to truncate a file, i. e., to reduce its file size by cutting off everything after a given offset. A special case is deleting all the content (setting the file size to 0). For emptying the file, we could simply delete and recreate it, but that might give the new version a different inode number, and also that is impossible with an open file.

On other Unix systems, the truncate functions can also grow a file (by supplying a `length` argument that is larger than the current size); we do not support this feature.

```c
int mx_ftruncate (int mfd, int length);
```

```c
int mx_ftruncate (int mfd, int length) {
    if (mfd < 0 || mfd >= MX_MAX_FILES) return -1;  // wrong mfd number
    struct mx_filestat *st = &mx_status[mfd];
    struct int_minix2_inode *inode = st->int_inode;
    if (inode == NULL) return -1;  // no open file
    // ... truncating code...
    return 0;
}
```
short device = inode->device;

if (inode->i_size ≤ length) return -1; // attempt to grow the file

// calculate blocks to delete
int last_kept_byte = length - 1;
int firstblock;
if (length == 0) firstblock = 0;
else firstblock = last_kept_byte / BLOCK_SIZE + 1;
int lastblock = inode->i_size / BLOCK_SIZE - 1;

if (lastblock ≥ firstblock) { // any blocks to delete?
    for (int i = firstblock; i ≤ lastblock; i++) {
        // delete block
        int zone = fileblocktozone (device, i, inode);
        mx_clear_zmap_bit (device, zone);
    }
}

// check indirection blocks
if (lastblock > 6 && inode->i_zone[7] != 0) {
    ⟨mx_ftruncate: free single indirection block⟩
}
if (lastblock > 262 && inode->i_zone[8] != 0) {
    ⟨mx_ftruncate: free double indirection block⟩
}

// reset size and write changed inode
inode->i_size = length;
inode->clean = false; // inode was changed
return 0;

Defines:
mx_ftruncate, used in chunks 420b and 484d.
Uses BLOCK_SIZE 440a, fileblocktozone 473a, int_minix2_inode 459a, mx_clear_zmap_bit 446, mx_filestat 460a,
MX_MAX_FILES 461a, mx_status 461b, and NULL 46a.

We do not implement the ⟨mx_ftruncate: free single indirection block⟩ and ⟨mx_ftruncate: free double indirection block⟩ code chunks since they are basically a rewrite of corresponding chunks in fileblocktozone. Instead of looking up a zone number, it must be set to 0. Thus, when we truncate a file, single and double indirection blocks remain in use (and linked by the inode); they will however be destroyed when the file is finally deleted, and they will also be reused when the file grows again.
12.6.10 Making and Removing Directories

What’s left is code for creating and deleting directories. Similar to symlinks, directories are just a special type of file, and we already know how to modify existing directories.

Deleting a directory means to free the data blocks that had been used by it and to remove its entry in the super-directory (the directory which is one step closer to the filesystem root and contains it)—this task is already handled by the `mx_unlink` function, so we need no further code in the kernel. Actually, we could provide an `mx_rmdir` function which is simpler than `mx_unlink` since directories must not be hard-linked: if we remove a directory, we always remove its inode.

For making a directory, we could do this as the logical three-step-procedure that is involved:

- create the (empty) directory,
- within the new directory, create a hard link . to itself,
- and also create a hard link ..—either to the super-directory or to / in case of the root directory /.

But the kernel will never create a root directory (that’s handled by the user mode tool `mkfs.minix`)

Also, all new directories look identical except for the two hard links . and .., so we will keep our task simple by defining what the contents of a new directory look like (data-block-wise) and how to update the inode numbers in the . and .. entries.

The “.” character (dot) has the ASCII value 46, or 0x2E in hexadecimal code. A hexdump of the data area of a Minix filesystem shows the following contents for an empty directory:

```
$ hexdump -C /tmp/minixdata.img
[...]
00013000 08 00 2e 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00013010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00013020 01 00 2e 2e 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00013030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
* [...]

This shows a directory on the top level with inode number 8. The first two lines contain the directory entry for . (pointing to itself: 08 00) and the last two lines hold the directory entry for .. (pointing to the root directory which has inode number 1 (01 00)).

The size of such an entry is 64:

```
$ ls -ld /mnt/minix/empty
drwxr-xr-x 2 esser esser 64 Jul 17 16:13 /mnt/minix/empty
```

The code for creating a directory with

```
(int mx_mkdir (int device, const char *path, int mode);
```
is as follows:

```c
int mx_mkdir (int device, const char *path, int mode) {
    struct minix_dir_entry entry = { 0);
    char dir[256]; char base[256];
    int dir_inodenr, new_inodenr;
    struct minix2_inode inode;

    splitpath (path, dir, base); // split path into dir and base
    dir_inodenr = mx_pathname_to_ino (device, dir);

    // create new file
    int fd = mx_open (device, path, O_CREAT | O_WRONLY); mx_close (fd);
    new_inodenr = mx_pathname_to_ino (device, path);

    // enter "." and "." inode numbers
    fd = mx_open (device, path, O_WRONLY);
    memset (&entry, 0, 32); memcpy (entry.name, ".", 2); entry.inode = new_inodenr;
    mx_write (fd, &entry, 32);
    memset (&entry, 0, 32); memcpy (entry.name, '..', 3); entry.inode = dir_inodenr;
    mx_write (fd, &entry, 32);
    mx_close (fd);

    // fix inode (make it type directory, set nlinks to 2)
    mx_read_inode (device, new_inodenr, &inode);
    inode.i_mode = S_IFDIR | (mode & 0777); // set mode
    inode.i_uid = thread_table[current_task].euid; // set UID
    inode.i_gid = thread_table[current_task].egid; // set GID
    inode.i_nlinks = 2;
    mx_write_inode (device, new_inodenr, &inode);

    // update link count of directory above
    mx_read_inode (device, dir_inodenr, &inode);
    inode.i_nlinks++;
    mx_write_inode (device, dir_inodenr, &inode);
    return 0;
}
```

 Defines:
 `mx_mkdir`, used in chunks 422a and 486.
 Uses `current_task` 192c, `egid` 573a, `euid` 573a, `memcpy` 596c, `memset` 596c, `minix2_inode` 442a, `minix_dir_entry` 452b, `mx_close` 467b, `mx_open` 464b, `mx_pathname_to_ino` 461d, `mx_read_inode` 451b, `mx_write` 474c, `mx_write_inode` 452a, `O_CREAT` 460b, `O_WRONLY` 460b, `S_IFDIR` 457c, `splitpath` 455a, and `thread_table` 176b.

 Deleting a directory via

```c
int mx_rmdir (int device, const char *fullpath, const char *path);
```

 is only allowed if . and .. are its only entries. If so, we first unlink those two and then remove the directory (file). Note that the function expects two path arguments, the full
(global) path and the device-local path. Our implementation does not use the device-local path, but new implementations of similar *rmdir functions for other filesystems might do so.

```
int mx_rmdir (int device, const char *fullpath, const char *path) {
    // check relative/absolute path
    if (*path != '/') relpath_to_abspath (fullpath, abspath);
    else strncpy (abspath, fullpath, 256);

    // check if directory exists
    int fd = u_open ((char*)fullpath, O_RDONLY, 0);
    if (fd == -1) { return -1; }
    u_close (fd);

    // split path into dir and base
    splitpath (abspath, dir, base);

    // save current working directory
    u_getcwd ((char*)cwd, 256);

    if (mx_directory_is_empty (device, path)) {
        if (u_chdir (dir) == -1) { return -1; }
        if (u_unlink (".")) == -1) { u_chdir (cwd); return -1; }
        if (u_unlink ("..") == -1) { u_chdir (cwd); return -1; }
        if (u_unlink (fullpath) == -1) { u_chdir (cwd); return -1; }

        u_chdir (cwd); // restore old current working directory
        return 0;
    }
    printf ("Directory not empty\n");
    return -1; // not empty
}
```

Defines:
  mx_rmdir, used in chunks 422a and 487b.

Uses cwd, minix2_inode 442a, minix_dir_entry 452b, mx_directory_is_empty 489a, O_RDONLY 460b, printf 601a, relpath_to_abspath 412b, splitpath 455a, strncpy 594b, u_chdir 432e, u_close 418a, u_getcwd 432e, u_open 412c, and u_unlink 418b.

We still need the function mx_directory_is_empty which could check the file size of a directory: if it is 64 then the directory should contain only the . and .. entries. Our

```
boolean mx_directory_is_empty (int device, const char *path);
```
function expects that the provided path argument is already a (device-local) absolute path, i.e., it starts with a slash. It does not use the size check but queries individual directory entries instead because we don’t always update the directory size when we delete files.

```c
boolean mx_directory_is_empty (int device, const char *path) {
    int count = 0; // number of entries
    struct dir_entry d = { 0 };
    for (int i = 0; i < 32 * 7; i++) {
        if (mx_getdent (device, path, i, &d) == 0 && d.inode != 0)
            count++;
    }
    return (count == 2);
}
```

Defines:
- `mx_directory_is_empty`, used in chunk 488.
- `dir_entry` and `mx_getdent`.

The function uses `mx_getdent()` which we present in the following section.

### 12.6.11 Listing a Directory

In order to retrieve the information stored in the inode of a file, all Unix systems offer a `stat()` function which fills a status structure with content. What that structure looks like depends on the flavor of Unix; for ULIIX we use the `struct stat` definition as it is shown in the Linux man page `stat(2)`, but without the `st_blksize` and `st_blocks` entries:

```c
struct stat {
    unsigned int  st_dev;  // ID of device containing file
    unsigned short st_ino; // inode number
    unsigned short st_mode; // protection
    unsigned short st_nlink; // number of hard links
    unsigned short st_uid;  // user ID of owner
    unsigned short st_gid;  // group ID of owner
    unsigned short st_rdev; // device ID (if special file)
    unsigned int  st_size; // total size, in bytes
    unsigned int  st_atime; // time of last access
    unsigned int  st_mtime; // time of last modification
    unsigned int  st_ctime; // time of last status change
};
```

Defines:
- `stat`, used in chunks 420c, 421d, 426b, 432e, 489, 490, 499, 576, 577c, and 608a.

We have not prefixed the type name with `mx_` because we use it for all supported filesystems. However, there are several `*stat` functions that retrieve the data—one for each supported filesystem. As part of the Minix filesystem implementation, we provide the

```c
int mx_stat (int device, const char *path, struct stat *buf);
```

function which simply locates the inode and copies the data into a `struct stat` variable:
int mx_stat (int device, const char *path, struct stat *buf) {
    struct minix2_inode inode;
    int ino = mx_pathname_to_ino (device, path);
    if (ino == -1) return -1;    // error
    buf->st_dev  = device;              buf->st_rdev = 0;
    buf->st_ino  = ino;
    mx_read_inode (device, ino, &inode); // read the inode
    buf->st_mode = inode.i_mode;              buf->st_nlink = inode.i_nlinks;
    buf->st_uid  = inode.i_uid;               buf->st_gid  = inode.i_gid;
    buf->st_size = inode.i_size;              buf->st_atime = inode.i_atime;
    buf->st_ctime = inode.i_ctime;            buf->st_mtime = inode.i_mtime;
    return 0;
}

#define mx_stat, used in chunks 421d and 490d.
Uses minix2_inode 442a, mx_pathname_to_ino 461d, mx_read_inode 451b, read 429b, and stat 429b 489b.

We define the generic directory entry structure struct dir_entry_490b that is similar to the Minix structure struct minix_dir_entry_432b, but allows longer filenames.

struct dir_entry {
    word inode;   // inode number
    byte filename[64]; // filename
};

The function

int mx_getdent (int device, const char *path, int index, struct dir_entry *buf);

fills a struct dir_entry_490b buffer with the data found in the Minix inode on disk via the mx_read_dir_entry_453b function.

int ret = mx_stat (device, path, &s);
if (ret == -1) return -1;    // error does not exist

if (index*32 ≥ s.st_size) return -1; // index out of bounds

int ino = mx_pathname_to_ino (device, path);
if (ino == -1) return -1;    // error: not a directory

ret = mx_read_dir_entry (device, ino, index, &d);
if (ret == -1) return -1;    // error: no such entry in directory
buf->inode = d.inode;
strncpy ((char*)buf->filename, d.name, 30);
d.name[30] = 0;    // terminate string
return 0;        // success
}

Defines:
mx_getdent, used in chunks 422c, 489a, and 490c.

Uses dir_entry 490b, minix_dir_entry 452b, mx_pathname_to_ino 461d, mx_read_dir_entry 453b, mx_stat 490a, stat 429b 489b, and strncpy 594b.

12.6.12 Filesystem Information: df

In order to implement a df (disk free) application we need a method to query the number of free blocks on a filesystem. As we will only support this for the Minix filesystem, we do not provide a generic layer (as part of the virtual filesystem) but directly write a Minix-specific function. It will fill struct diskfree_query structures:

```
struct diskfree_query {
    int device;       // device ID (is set before calling mx_diskfree)
    int size;         // size of filesystem, in blocks
    int used;         // number of used blocks
    int free;         // number of free blocks (redundant; == size-used)
    char name[10];    // name (such as "/dev/hda")
    char mount[256];  // mount point
    char fstype[10];  // filesystem name, e.g. "minix"
};
```

Defines:
diskfree_query, used in chunks 491–93.

The goal is to implement

```
void mx_diskfree (struct diskfree_query *query);
```

Uses diskfree_query 491a and mx_diskfree 492.

which will use the helper function

```
int count_zeros (byte *block, int maxcount) {
    int count = 0;
    for (int i = 0; i < (maxcount+7)/8; i++) {
        if (block[i] == 0) { count += 8; }
        else {
            for (int j = 0; j < 8; j++) {
                if (i*8 + j < maxcount && (block[i] >> j) % 2 == 0) count++;
            }
        }
    }
    return count;
}
```
which counts the number of zero bits in a given block, but only up to the bit position specified by the maxcount parameter. (We can slightly optimize the counting by checking whether a byte is 0, in that case we can add 8 to the counter; this assumes that maxcount is always a multiple of 8.)

mx_diskfree\textsubscript{492} takes the device element of the structure as an argument, reads all zone map blocks of that device and counts the contained zeros. That gives us the number of free blocks. The other values are taken from the mount table or the superblock (or we calculate them).

\begin{verbatim}
void mx_diskfree (struct diskfree_query *query) {
    int device = query->device;
    struct minix_superblock sblock;
    char block[1024];
    query->size = mx_query_superblock (device, MX_SB_ZONES);
    unsigned int nblocks = mx_query_superblock (device, MX_SB_ZONES);
    unsigned int zmap_start = 2 + mx_query_superblock (device, MX_SB_IMAP_BLOCKS);
    unsigned int free_blocks = 0;
    for (int i = 0; i < mx_query_superblock (device, MX_SB_ZMAP_BLOCKS); i++) {
        readblock (device, zmap_start + i, (byte*)&block);
        if ((i+1)*8192 < query->size)
            free_blocks += count_zeros ((byte*)&block, 8192);
        else
            free_blocks += count_zeros ((byte*)&block, query->size - i*8192);
    }
    query->free = free_blocks;
    query->used = query->size - free_blocks;

    // find device name
    switch (device) {
    case DEV_HDA: strcpy (query->name, "/dev/hda", 10); break;
    case DEV_HDB: strcpy (query->name, "/dev/hdb", 10); break;
    case DEV_FD0: strcpy (query->name, "/dev/fd0", 10); break;
    case DEV_FD1: strcpy (query->name, "/dev/fd1", 10); break;
    default: strcpy (query->name, "unknown", 10); break;
    }

    // find mount point
    boolean mounted = false;
    for (int i=0; i<current_mounts; i++) {
        if (mount_table[i].device == device) {
            strcpy (query->ftype, fs_names[mount_table[i].ftype], 10);
            strcpy (query->mount, mount_table[i].mountpoint, 255);
            mounted = true;
            break;
        }
    }
}
\end{verbatim}
\[
\begin{align*}
\text{if (!mounted) }
&\text{ strncpy (query->fstype, "none", 10);} \\
&\text{ strncpy (query->mount, "none", 10);} \\
\end{align*}
\]

Defines: 
\begin{itemize}
\item \texttt{mx_diskfree}, used in chunks 491 and 493b.
\end{itemize}

Uses \texttt{count_zeros 491c, current_mounts 405b, DEV_FDO 508a, DEV_FDO1 508a, DEV_HDA 508a, DEV_HDB 508a, diskfree_query 491a, fs_names 410b, minix_superblock 440c, mount_table 405b, mx_query_superblock 443b, readblock 506b, and strncpy 594b.}

We provide a system call handler
\begin{itemize}
\item \texttt{syscall prototypes 493a}\[493a\]
\item \texttt{syscall functions 493b}\[493b\]
\end{itemize}

\[
\begin{align*}
\text{void syscall_diskfree (context_t *r);} \\
\text{void syscall_diskfree (context_t *r)} \{ \\
  \text{// ebx: address of diskfree query structure} \\
  \text{mx_diskfree (}(\text{struct diskfree_query}*)r\rightarrow\text{ebx});} \\
\}
\end{align*}
\]

Defines: 
\begin{itemize}
\item \texttt{syscall_diskfree}, used in chunk 493.
\item \texttt{context_t 142a, diskfree 493f, diskfree_query 491a, and mx_diskfree 492.}
\end{itemize}

and register it:
\begin{itemize}
\item \texttt{ulix system calls 206e}\[205a\]
\item \texttt{initialize syscalls 173d}\[44b\]
\end{itemize}

\[
\begin{align*}
\text{#define \_NR_diskfree 522} \\
\text{install_syscall_handler (\_NR_diskfree, syscall_diskfree);} \\
\text{install_syscall_handler 201b, and syscall_diskfree 493b.}
\end{align*}
\]

User mode applications can then ask for the information by writing a value into the device field of a \texttt{diskfree_query 491a} structure and calling
\begin{itemize}
\item \texttt{ulixlib function prototypes 174c}\[48a\]
\item \texttt{ulixlib function implementations 174d}\[48b\]
\end{itemize}

\[
\begin{align*}
\text{void diskfree (struct diskfree_query *query);} \\
\text{void diskfree (struct diskfree_query *query) \{} \\
  \text{syscall2 (\_NR_diskfree, (unsigned int)query);} \\
\}
\end{align*}
\]

Defines: 
\begin{itemize}
\item \texttt{diskfree}, used in chunk 493b.
\end{itemize}

Uses \texttt{\_NR_diskfree 493c, diskfree_query 491a, and syscall2 203c.}
12.7 The /dev Filesystem

This section provides an interface to block devices (via /dev/fd0, /dev/fd1, /dev/hda and /dev/hdb) and the physical memory (via /dev/kmem). We have not designed the code in a way that would easily allow extensions to other device classes (however, a third or fourth hard disk would be easy to add).

The /dev filesystem must be mounted on /dev/, so the filesystem itself has only the root directory and the following five files which reside inside:

```
const char *path, int oflag)
```

```
short dev;
int pos;
short mode;
```

We will simulate behavior of the Minix filesystem for our /dev filesystem so that the function which inspects a directory works with the /dev/ directory as well.
and allow up to 32 simultaneously open files:

```c
#define MAX_DEV_FILES 32
```

Defines:
- `MAX_DEV_FILES`, used in chunks 495, 496b, and 499b.

So, similar to the `mx_status` array, we declare a `dev_status` array that holds the same kind of information:

```c
struct dev_filestat dev_status[MAX_DEV_FILES] = { { 0 } };
```

Defines: `dev_status`, used in chunks 495–99.

Uses `dev_filestat` and `MAX_DEV_FILES`.

The `dev_open` function is much simpler than `mx_open` because we know exactly which files can be opened. We identify the inode number with the index into the table `dev_directory` (plus 1, as we start counting inodes at number 1).

```c
int dev_open (const char *path, int oflag) {
    if ((oflag & O_CREAT) != 0) return -1; // cannot create
    int i, dev_inode = -1;

    // get the inode number
    for (i = 0; i < 7; i++) {
        if (strequal (path+1, dev_directory[i].name)) {
            // found!
            dev_inode = dev_directory[i].inode; // which is always i...
            break;
        }
    }
    if (dev_inode == -1) return -1; // not found

    // find free file descriptor
    int fd = -1;
    for (i = 0; i < MAX_DEV_FILES; i++) {
        if (dev_status[i].dev == 0) { fd = i; break; }
    }
    if (fd == -1) return -1; // no free file descriptor

    switch (dev_inode) {
        case DEV_FD0_INODE : dev_status[fd].dev = DEV_FD0; break;
        case DEV_FD1_INODE : dev_status[fd].dev = DEV_FD1; break;
        case DEV_HDA_INODE : dev_status[fd].dev = DEV_HDA; break;
        case DEV_HDB_INODE : dev_status[fd].dev = DEV_HDB; break;
        case DEV_KMEM_INODE : dev_status[fd].dev = DEV_KMEM; break;
        default: dev_status[fd].dev = -1;
    }
    dev_status[fd].pos = 0;
```
```c
dev_status[fd].mode = oflag;
return fd;
}
```

Defines:
- `dev_open`, used in chunks 412c and 494c.

Uses `dev_directory` 494b, `DEV_FD0` 508a, `DEV_FD0_INODE` 494a, `DEV_FD1` 508a, `DEV_FD1_INODE` 494a, `DEV_HDA` 508a, `DEV_HDA_INODE` 494a, `DEV_HDB` 508a, `DEV_HDB_INODE` 494a, `DEV_KMEM` 508a, `DEV_KMEM_INODE` 494a, `dev_status` 495b, `MAX_DEV_FILES` 495a, `O_CREAT` 460b, and `strequal` 596a.

Closing a device file via

```c
⟨function prototypes 45a⟩+≡
int dev_close (int fd);
```

is much simpler:

```c
⟨function implementations 100b⟩+≡
int dev_close (int fd) {
  if (fd ≥ 0 && fd < MAX_DEV_FILES && dev_status[fd].dev != 0) {
    dev_status[fd].dev = 0;
    return 0; // success
  } else {
    return -1; // fail
  }
}
```

Defines:
- `dev_close`, used in chunks 418a and 496a.

Uses `dev_status` 495b and `MAX_DEV_FILES` 495a.

Finally we need `dev_read` 496d, `dev_write` 497 and `dev_lseek` 498a functions:

```c
⟨function prototypes 45a⟩+≡
int dev_read (int fd, char *buf, int nbyte);
int dev_write (int fd, char *buf, int nbyte);
int dev_lseek (int fd, int offset, int whence);
```

The implementation of `dev_read` 496d and `dev_write` 497 is similar to the one of `mx_read` 470b and `mx_write` 474c (which you have seen earlier), but it is simpler: the new functions need not access an inode in order to retrieve block numbers.

There is a special case for reading from memory via `/dev/kmem` which does not require any block read/write operations at all: For memory access we can simple call `memcpy`.

```c
⟨function implementations 100b⟩+≡
int dev_read (int fd, char *buf, int nbyte) {
  ⟨dev filesystem: check if fd is a proper file descriptor 499b⟩
  int startbyte = dev_status[fd].pos;
  int devsize = dev_size (dev_status[fd].dev);
  if (startbyte ≥ devsize) { return 0; } // nothing to read
  int endbyte = dev_status[fd].pos + nbyte - 1;
  if (endbyte ≥ devsize) {
    nbyte -= (endbyte - devsize + 1); endbyte = devsize - 1;
  }

  // special case /dev/kmem: direct memcpy()
```
if (dev_status[fd].dev == DEV_KMEM) {
    memcpy (buf, (char*) (PHYSICAL (startbyte)), nbyte);
    dev_status[fd].pos += nbyte;
    return nbyte;
}

int readbytes = 0; int offset, length;
int startblock = startbyte / BLOCK_SIZE; int curblock = startblock;
while (nbyte > 0) {
    byte block[BLOCK_SIZE];
    readblock (dev_status[fd].dev, curblock, (byte*) block);
    if (curblock == startblock) {
        offset = startbyte % BLOCK_SIZE; length = MIN (nbyte, BLOCK_SIZE - offset);
    } else {
        offset = 0; length = MIN (nbyte, BLOCK_SIZE);
    }
    memcpy (buf, block + offset, length);
    nbyte -= length; buf += length;
    readbytes += length; curblock++;
}
return readbytes;
}

int dev_write (int fd, char *buf, int nbyte) {
    int startbyte = dev_status[fd].pos;
    int devsize = dev_size (dev_status[fd].dev);
    if (startbyte >= devsize) { return 0; } // nothing to write
    int endbyte = dev_status[fd].pos + nbyte - 1;
    if (endbyte >= devsize) {
        nbyte -= (endbyte - devsize + 1); endbyte = devsize - 1;
    }

    // special case /dev/kmem: direct memcpy()
    if (dev_status[fd].dev == DEV_KMEM) {
        memcpy ((char*) (PHYSICAL (startbyte)), buf, nbyte);
        dev_status[fd].pos += nbyte;
        return nbyte;
    }
int written_bytes = 0; int offset, length;
int startblock = startbyte / BLOCK_SIZE; int curblock = startblock;
while (nbyte > 0) {
    byte block[BLOCK_SIZE];
    if (curblock == startblock) {
        offset = startbyte % BLOCK_SIZE; length = MIN (nbyte, BLOCK_SIZE - offset);
    } else {
        offset = 0; length = MIN (nbyte, BLOCK_SIZE);
    }
    if (offset != 0 || length != BLOCK_SIZE) {
        // writing a partial block -- read it first!
        readblock (dev_status[fd].dev, curblock, (byte*) block);
    }
    memcpy (block + offset, buf, length);
    writeblock (dev_status[fd].dev, curblock, (byte*) block);
    nbyte -= length; buf += length;
    written_bytes += length; curblock++;
    dev_status[fd].pos += length;
}
return written_bytes;
}

Defines:
dev_write, used in chunks 415a and 496c.
Uses BLOCK_SIZE 440a, DEV_KMEM 508a, dev_size 499a, dev_status 495b, memcpy 596c, MIN 471d, PHYSICAL 116a, readblock 506b, and writeblock 507c.

Seeking is also simple:

[498a] (function implementations 100b)+≡ (44a) <497 499a>
int dev_lseek (int fd, int offset, int whence) {
    (dev filesystem: check if fd is a proper file descriptor 499b)
    if (whence < 0 || whence > 2)
        return -1; // wrong lseek option
    if (whence == SEEK_END && offset > 0)
        return -1; // cannot seek beyond end of device
    switch (whence) {
        case SEEK_SET: dev_status[fd].pos = offset; break;
        case SEEK_CUR: dev_status[fd].pos += offset; break;
        case SEEK_END: dev_status[fd].pos = dev_size (dev_status[fd].dev) + offset;
    }
    return dev_status[fd].pos;
}

Defines:
dev_lseek, used in chunk 418a.
Uses dev_size 499a, dev_status 495b, lseek 429b, SEEK_CUR 469b, SEEK_END 469b, and SEEK_SET 469b.

We have used a function

[498b] (function prototypes 45a)+≡ (44a) <496c 499c>
long dev_size (int dev);
which returns the size of the drive; here is its implementation:

\[
\text{long dev_size (int dev) \{ switch (dev) \{ case DEV_FD0 : return fdd_type[fdd[0].type].total_sectors * 512; \ / fd0 case DEV_FD1 : return fdd_type[fdd[1].type].total_sectors * 512; \ / fd1 case DEV_HDA : return hd_size[0] * 512; \ / hda case DEV_HDB : return hd_size[1] * 512; \ / hdb case DEV_KMEM : return MEM_SIZE; \ / kmem default : return -1; \ / error \} }
\]

Defines:
- dev_size, used in chunks 496–99.
- Uses DEV_FD0, DEV_FD1, DEV_HDA, DEV_HDB, DEV_KMEM, fdd, fdd_type, hd_size, and MEM_SIZE.

In both functions for reading and writing we check whether a valid file descriptor was supplied and return -1 if not:

\[
\text{if (fd < 0 || fd ≥ MAX_DEV_FILES) return -1; \ / bad file descriptor if (dev_status[fd].dev == 0) return -1; \ / file not open}
\]

Uses dev_status 495b and MAX_DEVFILES 495a.

Last, we supply functions for querying a file and reading a directory entry which are called from u_stat and u_getdent in the virtual filesystem layer.

\[
\text{int dev_stat (const char *path, struct stat *buf);} \text{int dev_getdent (const char *path, int index, struct dir_entry *buf);}
\]

dev_stat compares the local path (which is expected to be /fd0, /fd1, /hda, /hdb or /kmem) against the list of known device names and fills the struct stat buffer:

\[
\text{int dev_stat (const char *path, struct stat *buf) \{ int devices[] = \{-1, 0, 0, DEV_FD0, DEV_FD1, DEV_HDA, DEV_HDB, DEV_KMEM \}; int dev_inode; for (int i = 0; i < 7; i++) \{ // get the inode number if (strequal (path+1, dev_directory[i].name)) \{ // found! dev_inode = dev_directory[i].inode; // which is always i... break; \} \} if (dev_inode == -1) return -1; // not found}
\]

// buf->st_dev = 0; // no device, /dev is a virtual FS buf->st_dev = devices[dev_inode];
buf->st_rdev = 0;
buf->st_ino  = dev_inode;
if (dev_inode > 2)
    buf->st_mode = S_IFBLK | 0600;  // block device; we have no char. devices
else
    buf->st_mode = S_IFDIR | 0600;  // directory
buf->st_nlink = 1;  buf->st_uid  = 0;
buf->st_gid  = 0;  buf->st_size  = dev_size(devices[dev_inode]);
buf->st_atime = 0;  buf->st_ctime = 0;
buf->st_mtime = 0;
return 0;
}

Defines:
    dev_stat, used in chunk 421d.
Uses dev_directory 494b, DEV_FD0 508a, DEV_FD1 508a, DEV_HDA 508a, DEV_HDB 508a, DEV_KMEM 508a, dev_size 499a,
    S_IFBLK 457c, S_IFDIR 457c, stat 429b 489b, and strequal 596a.

And dev_getdent, uses an entry in the dev_directory 494b, table into the buffer (of type struct dir_entry 490b).

(int dev_getdent (const char *path, int index, struct dir_entry *buf) {
    if (index < 0 || index > 6) return -1;  // no such entry

    buf->inode = dev_directory[index].inode;
    strncpy (buf->filename, dev_directory[index].name, 5);
    return 0;
}

Defines:
    dev_getdent, used in chunk 422c.
Uses dev_directory 494b, dir_entry 490b, and strncpy 594b.

12.8 Default Contents of the Filesystem

In the last two sections we describe the directories and files that you can find on the UNIX system disks and suggest two books which discuss other filesystems in depth.

Figure 12.14 shows the general tree structure of the virtual filesystem which is organized in a similar way as most Unix filesystems. The /bin directory contains executable programs, /etc is reserved for configuration files (currently the only one is /etc/passwd), /dev is the mount point for the device filesystem, and /home contains the home directories of system users, it is already populated with two home directories that “belong” to us (the authors of this book). They correspond to identical user names in the /etc/passwd file (with the passwords set to “xyz”). The administrator root has the home directory /root; it is standard practice to place it directly in the root directory (/) and not below /home.
12.9 Further Reading

If you are interested in further details about Unix filesystem implementations, we suggest you take a look at the following books:


12.10 Exercises

36. Sparse Files

Sparse files are “files with holes”: They have large areas which only contain 0x00 bytes. Storing all those zeros on disk is a waste of disk space (and also of time for initially writing them). Modern filesystems support a special treatment of sparse files where information about these holes is stored in the inode or in a separate block that is linked from the inode. The Minix filesystem does not support this kind of treatment, but you can modify it so that it does.

Modify the file access functions so that a block address of $-1 \ (0xFFF)$ is interpreted as a reference to a sparse area, i.e., an area completely filled with zeros. No blocks need to be allocated for such areas. If a direct block address is $-1$, the whole (logical) block is considered to consist of zeros. If an indirect block address is $-1$, it means that there is no indirection block, but the space that could be addressed via the indirection...
block (up to 256 blocks = 256 KByte) are assumed to contain zeros.

Every read access to a sparse area shall return a block of zeros, but when writing to a sparse area you need to allocate a block and change the block number from \(-1\) to the new zone number. If data is written in the middle of a sparse indirection block, you need to allocate another block that then serves as indirection block with one zone number pointing to the new non-zero data block (and all other zone numbers set to \(-1\)).

Change the `mx_write` function so that it recognizes whether a whole sparse block is being written (or whether the result of the current write operation is a block full of zeros)—if so, add a new sparse area and release the block that is no longer needed.

You will need to make some changes to the system calls and user mode library functions so that you can test the behavior.

37. **Completion of the `mx_ftruncate` Function**

Our implementation of `mx_ftruncate` is incomplete: It does not provide code for the \(<mx_ftruncate: free single indirection block>\) and \(<mx_ftruncate: free double indirection block>\) code chunks. Add those chunks (or modify the function otherwise) and try to do that in an optimized way that uses as few individual inode or block write operations as possible.
In the last chapter you saw the implementations of both the Minix and the FAT filesystem—but what we have not discussed so far is how to actually talk to the hardware: we used functions readblock and writeblock to read or write kilobyte-sized chunks of data from the disk, and in this chapter you will see how to implement this.

### 13.1 Block and Character Devices

Classically, devices are split into two categories: block devices and character devices. The difference lies in the amounts of data which are transferred with every single request. A typical character device is the keyboard: each key-press generates an interrupt and the amount of data transferred is (typically) two bytes. On the other hand, disks (both floppy and hard disks) transfer whole chunks of data (512 bytes or larger quantities). The controllers for floppy and hard disks do not provide the functionality to read/write single bytes from/to the disk, but can only handle those larger chunks. That has consequences for code which wants to change a single byte in a disk file: The chunk containing this byte must first be read into memory, then modified and finally rewritten to disk.

Block devices can be accessed in two ways: in the classical approach drivers used the processor's in and out instructions for every single byte that was to be transferred. Reading a 512-byte-sized chunk of data from the disk would basically look like this:

```plaintext
out iport1, sector ; request data from the disk
out iport2, READ_CMD
mov reg1, memory ; set up target address, length of data
mov reg2, iport3
mov reg3, length ; for "rep"
rep insl ; read data (loop)
```
The rep prefix in the final statement uses the length argument stored in some register to repeat the insl instruction length times and auto-increment the memory address so that all the bytes coming from the disk will be stored in consecutive memory positions.

This approach requires the CPU to do a lot of work since it has to deal with every single byte that is to be transferred. It is better to use direct memory access (DMA) which allows the disk controller to store the data in memory without bothering the CPU.

In this chapter we will provide implementations of three device drivers:

• We start with a driver for a device that doesn’t exist but can easily be simulated: the “serial disk”. This driver assumes that a disk is connected to the serial port of the machine. The drive accepts read/write requests and sends or receives single bytes of such a request via the serial port. Every byte sent by the disk will generate an interrupt (and we need to provide an interrupt handler which will then read the newly-arrived byte via an in instruction), every byte we want to send to the disk must be sent explicitly via an out instruction.

• The second driver uses the classical (non-DMA) approach for accessing hard disks. It is easy to implement; a request for a 512-bytes-chunk is sent to the controller, the controller reads the data from the disk and then generates one interrupt. The interrupt handler must then read all 512 bytes from the controller. We use this to access the hard disks on the machine.

• The third driver uses DMA to talk to a floppy drive: Reading a 512-bytes-chunk also starts with requesting it from the controller, but the transfer happens in the background. When it’s finished, the controller generates an interrupt, and the interrupt handler only needs to acknowledge it and tell the (suspended) process that its data have arrived.

This collection of drivers thus introduces three very different approaches for talking to mass media controllers.

### 13.2 Device Selection

We will provide two generic functions

\[
\text{void readblock (int device, int blockno, char *buffer);} \\
\text{void writeblock (int device, int blockno, char *buffer);} \\
\]

which read kilobyte-sized blocks from all the devices we support. As a naming convention for devices we use the Unix concept of major and minor device IDs—this lets us break down device IDs into a device class (the major device number) and the specific device of a class (the minor device number). We use the same numbers as Linux (for floppies and hard disks):

• Floppy drives have the major number 2, we support two drives /dev/fd0 and /dev/fd1 with minor numbers 0 and 1.
• Hard disks have the major number 3, we support two drives /dev/hda and /dev/hdb with minor numbers 0 and 64 (the numbers 1–63 and 65–127 are reserved for partitions of the first and second hard disk, respectively, but we do not implement partition support for hard disks).

• The serial disk has major number 42. There is only one of this kind: /dev/sdisk has minor number 0.

We combine major and minor numbers in one 16 bit wide device number via
\[ \text{device} = \text{major} \ll 8 + \text{minor}. \]
That is: the upper eight bits of a device number contain the major number, and the lower eight bits contain the minor number.

With that formula we can also calculate major and minor numbers from a given device number:
\[ \text{major} = \text{device} \gg 8 \quad \text{and} \quad \text{minor} = \text{device} \& 0xff. \]

This leads to the major, minor and device numbers shown in Table 13.1. Note that we do not provide devices for the serial ports or the keyboard even though they are also used by Ulix—they would be examples for the class of character devices.

We provide the following three functions to do the calculations:

\[ \begin{align*}
\&\text{word makedev (byte major, byte minor);} \\
\&\text{byte devmajor (word device);} \\
\&\text{byte devminor (word device);} \\
\end{align*} \]

They just use the formulas which we have already described above:

\[ \begin{align*}
\&\text{word makedev (byte major, byte minor) \{ return ((major \ll 8) + minor); \}} \\
\&\text{byte devmajor (word device) \{ return (device \gg 8); \}} \\
\&\text{byte devminor (word device) \{ return (device \& 0xff); \}} \\
\end{align*} \]

Defining some constants makes our life easier in the following implementations:

<table>
<thead>
<tr>
<th>device file</th>
<th>major</th>
<th>minor</th>
<th>device</th>
</tr>
</thead>
<tbody>
<tr>
<td>/dev/fd0</td>
<td>2</td>
<td>0</td>
<td>0x0200 = 512</td>
</tr>
<tr>
<td>/dev/fd1</td>
<td>2</td>
<td>1</td>
<td>0x0201 = 513</td>
</tr>
<tr>
<td>/dev/hda</td>
<td>3</td>
<td>0</td>
<td>0x0300 = 768</td>
</tr>
<tr>
<td>/dev/hdb</td>
<td>3</td>
<td>64</td>
<td>0x0340 = 832</td>
</tr>
<tr>
<td>/dev/kmem</td>
<td>4</td>
<td>0</td>
<td>0x0400 = 1024</td>
</tr>
<tr>
<td>/dev/sdisk *)</td>
<td>42</td>
<td>0</td>
<td>0x2a00 = 10752</td>
</tr>
</tbody>
</table>

Table 13.1: Ulix supports these devices. *) The device file /dev/sdisk is not available.
Disk I/O

\[\text{constants}\ 112a\] + \equiv

\#define MAJOR_FD 2
\#define MAJOR_HD 3
\#define MAJOR_KMEM 4
\#define MAJOR_SERIAL 42

Defines:
MAJOR_FD, used in chunks 506b and 507b.
MAJOR_HD, used in chunks 506b and 507b.
MAJOR_SERIAL, used in chunks 506b and 507b.

The generic readblock506b and writeblock507c functions calculate the major and minor numbers from the device number and then call the appropriate reading or writing function for the correct device class:

\[\text{function implementations}\ 100b\] + \equiv

void readblock (int device, int blockno, char *buffer) {
    // check buffer
    if (buffer_read (device, blockno, buffer) == 0) { return; }

    // read from disk
    byte major = devmajor (device);
    byte minor = devminor (device);
    switch (major) {
        case MAJOR_HD:   readblock_hd (minor/64, blockno, buffer); break;
        case MAJOR_FD:   readblock_fd (minor,   blockno, buffer); break;
        case MAJOR_SERIAL: readblock_serial ( blockno, buffer); break;
        default: return;
    }

    // update buffer
    buffer_write (device, blockno, buffer, BUFFER_CLEAN);
}

Defines:
readblock, used in chunks 443–45, 448, 451a, 453b, 471c, 473–75, 477, 482, 492, 496d, and 497.
Uses BUFFER_CLEAN 510a, buffer_read 509d, buffer_write 510b, devmajor 505b, devminor 505b, MAJOR_FD 506a,
MAJOR_HD 506a, MAJOR_SERIAL 506a, readblock_fd 550d, readblock_hd 531b, and readblock_serial 522e.

The case selection is straightforward: depending on the major number, readblock506b calls either readblock_hd 531b (for hard disk access), readblock_fd 550d (for the floppy disks), or readblock_serial 522e (for the serial disk), and we will provide implementations of those functions in the following chapters.

The readblock506b function also calls buffer_read 509d and buffer_write 510b, which we have not discussed yet—these two functions provide access to a system-wide disk cache which stores the contents of disk blocks so that they need not be read again when they are requested a second time. We will introduce the buffer mechanism in the next section. The short explanation for the above code is this: readblock506b first checks whether the requested block is already in the cache. If so, no disk access is necessary, and the function can return immediately. Otherwise one of the readblock_* functions takes care of the block transfer from disk to memory, and after that the freshly-read block is stored in the
cache. (The BUFFER_CLEAN argument states that the buffer’s copy of the block is identical to the disk’s version.) Note that the memory device (DEV_KMEM) is not supported by readblock or writeblock: it is no block device; the u_read and u_write functions use memcpy to access the memory.

Writing a block comes in two variations: We first show the writeblock_raw function which is the direct counterpart to readblock: it has the same case selection and forwards the writing task to the writeblock_hd (for hard disks), writeblock_fd (for floppies) and writeblock_serial (serial disk) functions. When that is done, it also updates the buffer’s copy of the block (if it is already cached).

```c
#define UPDATE_BUF 1
#define DONT_UPDATE_BUF 0

void writeblock_raw (int device, int blockno, char *buffer, char flag) {
    byte major = devmajor (device);
    byte minor = devminor (device);
    switch (major) {
        case MAJOR_HD: writeblock_hd (minor/64, blockno, buffer); break;
        case MAJOR_FD: writeblock_fd (minor, blockno, buffer); break;
        case MAJOR_SERIAL: writeblock_serial (blockno, buffer); break;
        default: break;
    }
}

// update buffer cache (if it is in the cache)
if ((flag == UPDATE_BUF) && (buffer_contains (device, blockno)))
    buffer_write (device, blockno, buffer, BUFFER_CLEAN);
```

However, in order to increase the disk performance, Ulix will not write blocks to disk immediately. Instead, we will always call the following function (writeblock) which simply copies the data into the cache and marks it as dirty. At regular intervals the kernel will check whether there are dirty blocks and write them to disk.

```c
void writeblock (int device, int blockno, char *buffer) {
    buffer_write (device, blockno, buffer, BUFFER_DIRTY);
}
```

However, in order to increase the disk performance, Ulix will not write blocks to disk immediately. Instead, we will always call the following function (writeblock) which simply copies the data into the cache and marks it as dirty. At regular intervals the kernel will check whether there are dirty blocks and write them to disk.
Note that when calling `readblock_hd` or `writeblock_hd`, we pass `minor/64` as argument which turns the only supported values for `minor` (0 and 64) into 0 and 1 for the two hard disks.

If you want to add a driver for a different kind of media (e.g. CD-ROM or DVD-ROM drives) all you need to do is develop `readblock_XX` and `writeblock_XX` functions for this new category, define a new `MAJOR_XX` constant and add the new case to the implementations of `readblock` and `writeblock_raw`.

We define device constants for the five devices we plan to use regularly and another one that is used for error checking:

```
#define DEV_HDA 0x300 // disk /dev/hda
#define DEV_HDB 0x340 // disk /dev/hdb
#define DEV_FD0 0x200 // floppy /dev/fd0
#define DEV_FD1 0x201 // floppy /dev/fd1
#define DEV_KMEM 0x400 // memory /dev/kmem
#define DEV_NONE 0 // no device
```

Defines:
- `DEV_FD0`, used in chunks 406, 492, 495c, and 499.
- `DEV_FD1`, used in chunks 405b, 406, 492, 495c, and 499.
- `DEV_HDA`, used in chunks 405b, 406, 492, 495c, 499, 607a, and 610d.
- `DEV_HDB`, used in chunks 405b, 406, 492, 495c, and 499.
- `DEV_KMEM`, used in chunks 495–97 and 499.
- `DEV_NONE`, used in chunk 405b.

### 13.3 A Simple Buffer Cache

In early versions of U1ix, the `readblock` and `writeblock` functions directly accessed the drive controllers which made even simple things such as displaying the contents of the root directory very slow, since many blocks were read over and over again.

Using a buffer cache can dramatically speed up disk access (to blocks which have already been read) by buffering them in memory. For our simple system it does not take much, we provide a buffer that can store 512 blocks:

```
#define BUFFER_CACHE_SIZE 512
```

Defines:
- `BUFFER_CACHE_SIZE`, used in chunks 509–12.

Buffer entries store the buffer and some additional information: the device and block numbers (in order to identify which block is cached), an access counter and a dirty flag:

```
struct buffer_entry {
    char buf[BLOCK_SIZE];
    int dev;       // from what device? (~1 if free)
    int blockno;  // block number of buffered block (~1 if free)
    byte count;   // how often was it read?
    byte dirty;   // true if not written to disk
};
```
13.3 A Simple Buffer Cache

Defines:
- buffer_entry, used in chunk 509a.
Uses BLOCK_SIZE 440a.

The cache is just an array of buffer entries:

\[
\begin{aligned}
\langle \text{global variables} \rangle &+\equiv \\
\text{struct buffer_entry buffer_cache[BUFFER_CACHE_SIZE];} \\
\text{lock buffer_lock;}
\end{aligned}
\]

Defines:
- buffer_cache, used in chunks 509–12.
- buffer_lock, used in chunks 509, 510b, 512b, and 606.
Uses BUFFER_CACHE_SIZE 508b, buffer_entry 508c, and lock 365a.

and the kernel lock buffer_lock 509a protects against parallel access attempts.

Here's how we initialize the buffer cache at system start:

\[
\begin{aligned}
\langle \text{initialize system} \rangle &+\equiv \\
\text{memset (buffer_cache, 0, sizeof (buffer_cache));} \\
\text{for (int i = 0; i < BUFFER_CACHE_SIZE; i++)} \\
\quad \text{buffer_cache[i].blockno =} \\
\quad \text{buffer_cache[i].dev = -1;} \\
\quad \text{buffer_cache[i].dirty = 0;}
\end{aligned}
\]

buffer_lock = get_new_lock ("disk buffer");

Uses buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_lock 509a, get_new_lock 367b, and memset 596c.

Next we need code for entering data into and extracting it from the buffer cache; we write three functions

\[
\begin{aligned}
\langle \text{function prototypes} \rangle &+\equiv \\
\text{int buffer_write (int dev, int blockno, char *block, char dirtyflag);} \\
\text{int buffer_read (int dev, int blockno, char *block);} \\
\text{boolean buffer_contains (int dev, int blockno);} \\
\end{aligned}
\]

The functions for reading and writing buffer entries have the same signatures as the readblock 506b and writeblock_raw 507b functions.

Reading is the simpler task, so we start with that:

\[
\begin{aligned}
\langle \text{function implementations} \rangle &+\equiv \\
\text{int buffer_read (int dev, int blockno, char *block) } \\
\quad \{ \\
\quad // don't use the buffer before the scheduler is up \\
\quad \quad \text{if (!scheduler_is_active) } \{ \text{return -1;} \} \quad // -1 signals: must be read from disk \\
\quad \quad \text{mutex_lock (buffer_lock);} \\
\quad \quad \text{// check if buffer cache holds the requested block} \\
\quad \quad \text{int pos = -1; } \quad // \text{position in the cache} \\
\quad \quad \text{for (int i = 0; i < BUFFER_CACHE_SIZE; i++)} \\
\quad \quad \quad \text{if ((buffer_cache[i].dev == dev) && (buffer_cache[i].blockno == blockno)) } \{ \\
\quad \quad \quad \quad \text{// found it!} \\
\quad \quad \quad \quad \text{pos = i;} \\
\quad \quad \quad \text{break;} \\
\quad \quad \}\}
\end{aligned}
\]
if (pos == -1) { mutex_unlock (buffer_lock); return -1; } // not found

// we found it: copy the contents, update the counter
memcpy (block, buffer_cache[pos].buf, BLOCK_SIZE);
if ((int)buffer_cache[pos].count < 254) { buffer_cache[pos].count++; }
mutex_unlock (buffer_lock); return 0; // success

Defines:
buffer_read, used in chunk 506b.
Uses BLOCK_SIZE 440a, buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_lock 509a, memcpy 596c,
mutex_lock 366a, mutex_unlock 366c, and scheduler_is_active 276e.

Writing to the buffer is a little more complicated—if there is no entry for the block we want to write. Otherwise it’s pretty much the same:

[510a]  ⟨constants 112a⟩+≡ (44a) <508b 515b>
#define BUFFER_CLEAN 0
#define BUFFER_DIRTY 1

Defines:
BUFFER_CLEAN, used in chunks 506b and 507b.
BUFFER_DIRTY, used in chunk 507c.

[510b]  ⟨function implementations 100b⟩+≡ (44a) <509d 512b>
int buffer_write (int dev, int blockno, char *block, char dirtyflag) {
  // don't use the buffer before the scheduler is up
  if (!scheduler_is_active) { return 0; }
  mutex_lock (buffer_lock);
  // check if buffer cache already holds the requested block
  int pos = -1; // position in the cache
  for (int i = 0; i < BUFFER_CACHE_SIZE; i++) {
    if ((buffer_cache[i].dev == dev) && (buffer_cache[i].blockno == blockno)) {
      pos = i; break; // found it!
    }
  }

  // if not found, create it
  if (pos == -1) { ⟨buffer cache: find or create free entry; sets pos 511a⟩ }

  // copy the contents, update the counter
  if ((pos ≥ 0) && (pos < BUFFER_CACHE_SIZE)) {
    memcpy (buffer_cache[pos].buf, block, BLOCK_SIZE);
    if ((int)buffer_cache[pos].count < 254)
      buffer_cache[pos].count++;
    buffer_cache[pos].dirty = dirtyflag;
  }
  mutex_unlock (buffer_lock);
  return 0; // success
}

Defines:
buffer_write, used in chunks 506, 507, and 509c.
Uses BLOCK_SIZE 440a, buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_lock 509a, memcpy 596c,
mutex_lock 366a, mutex_unlock 366c, and scheduler_is_active 276e.
The obvious difference is that writing to the buffer cache always succeeds because we either update an existing entry or create a new entry. Creating a new one is not a problem as long as there remain free entries:

\[
\begin{align*}
\text{(buffer cache: find or create free entry; sets pos 511a)} \equiv \\
\text{pos} &= -1; \quad \text{// new search} \\
\text{for (int } i = 0; \ i < \text{BUFFER_CACHE_SIZE}; \ i++) \
\text{ if (buffer_cache[i].dev == -1) \\
\text{ \ \ \ \ \ \ \ pos = i; \ break; \quad \text{// this one is free}} \\
\text{}} \\
\text{if (pos == -1) \{} \quad \text{// we found no free entry} \\
\text{\ \ \ \ \ \ \ (buffer cache: free an entry; sets pos 511b)} \\
\text{}} \\
\text{buffer_cache[pos].dev = dev;} \\
\text{buffer_cache[pos].blockno = blockno;} \\
\text{buffer_cache[pos].count = 0;}
\end{align*}
\]

Uses buffer_cache 509a and BUFFER_CACHE_SIZE 508b.

This code prepares the buffer cache entry by setting its dev and blockno members. The memset 506c command above would also zero out the buffer’s contents, but this is not needed since it will be overwritten immediately.

Finally we need to say how to find an entry when all entries are in use. This asks for a replacement strategy and we’ll implement a simple “least often used” strategy.

\[
\begin{align*}
\text{(buffer cache: free an entry; sets pos 511b)} \equiv \\
\text{begin_buffer_search: \quad \text{// find first clean entry}} \\
\text{pos} &= -1; \\
\text{for (int } i = 0; \ i < \text{BUFFER_CACHE_SIZE}; \ i++) \\
\text{ \ \ \ \ \ \ \ if (buffer_cache[i].dirty == false) \\
\text{ \ \ \ \ \ \ \ \ \ \ pos = i; \ break; \quad \text{// end loop}} \\
\text{if (pos == -1) \{} \quad \text{// all buffers are dirty} \\
\text{ \ \ \ buffer_sync (0);} \\
\text{ \ \ \ goto begin_buffer_search;} \\
\text{}} \\
\text{int least_used_val = buffer_cache[pos].count;} \\
\text{for (int i = pos+1; \ i < \text{BUFFER_CACHE_SIZE}; \ i++) \\
\text{ \ \ \ if (buffer_cache[i].count < least_used_val \&\& buffer_cache[i].dirty == false) \\
\text{ \ \ \ \ \ \ \ \ \ \ // this entry is clean and was accessed less often} \\
\text{ \ \ \ least_used_val = buffer_cache[i].count;} \\
\text{ \ \ \ pos = i; \quad \text{// update candidate}} \\
\text{}}
\end{align*}
\]

Uses buffer_cache 509a, BUFFER_CACHE_SIZE 508b, buffer_sync 512b, and least_used_val.
When we want to force a synchronization of the buffer (i.e., writing dirty entries to disk and thereby making them clean), we call the `buffer_sync` function

```
void buffer_sync (boolean lock_buffer);
```

which takes one argument indicating whether the `buffer_lock` needs to be acquired:

```
void buffer_sync (boolean lock_buffer) {
    _set_statusline ("[B]", 34);
    if (lock_buffer) mutex_lock (buffer_lock);
    for (int i = 0; i < BUFFER_CACHE_SIZE; i++) {
        if (buffer_cache[i].dirty == true) {
            writeblock_raw (buffer_cache[i].dev, buffer_cache[i].blockno,
                           (char*)buffer_cache[i].buf, DONT_UPDATE_BUF);
            buffer_cache[i].dirty = false;
        }
    }
    if (lock_buffer) mutex_unlock (buffer_lock);
    _set_statusline ("[ ]", 34);
}
```

Defines:

- `buffer_sync`, used in chunks 511–13.

Uses `_set_statusline` 337b, `buffer_cache` 509a, `BUFFER_CACHE_SIZE` 508b, `buffer_lock` 509a, `DONT_UPDATE_BUF` 507a, `mutex_lock` 366a, `mutex_unlock` 366c, and `writeblock_raw` 507b.

We also add a function `buffer_contains` which lets us query whether a specific block is currently buffered:

```
boolean buffer_contains (int dev, int blockno) {
    // don't use the buffer before the scheduler is up
    if (!scheduler_is_active) { return false; }

    // check if buffer cache holds this block
    for (int i = 0; i < BUFFER_CACHE_SIZE; i++) {
        if (((buffer_cache[i].dev == dev) && (buffer_cache[i].blockno == blockno)) {
            return true; // found it!
        }
    }
    return false;
}
```

Defines:

- `buffer_contains`, used in chunk 507b.

Uses `buffer_cache` 509a, `BUFFER_CACHE_SIZE` 508b, and `scheduler_is_active` 276e.

In order to let the user synchronize the buffer cache (before shutting down the Ulix machine), we provide a `syscall_sync` system call:

```
void syscall_sync (context_t *r);
```
A Simple Buffer Cache

 syscall functions 174b

```c
void syscall_sync (context_t *r) {
    // this syscall takes no arguments
    buffer_sync (1);  // with lock
}
```

Defines: syscall_sync, used in chunks 512d and 513b.
Uses buffer_sync 512b and context_t 142a.

initialize syscalls 173d

```c
install_syscall_handler (__NR_sync, syscall_sync);
```

Uses __NR_sync 204c, install_syscall_handler 201b, and syscall_sync 513a.

The system call will be available via the user mode library function

ulixlib function prototypes 174c

```c
void sync ();
```

that we implement here: It provides no arguments, so we use syscall1 203c.

ulixlib function implementations 174d

```c
void sync () { syscall1 (__NR_sync); }
```

Defines: sync, used in chunk 513c.
Uses __NR_sync 204c and syscall1 203c.

Syncing will be done by the following swapper process which needs to be started during system initialization:

lib-build/tools/swapper.c 311b

```c
#include "../ulixlib.h"
int main () {
    int pid = getpid ();
    if (pid != 2) { printf ("swapper: don't start_ me manually.\n"); exit (1); }
    setterm (9); setpsname ("[swapper]");
    int init_frames = get_free_frames ();
    int last_free_frames;
    int free_frames = init_frames;
    unsigned int counter = 0;
    #define THRESHOLD (init_frames - 500)
    for (;;) {
        last_free_frames = free_frames;
        free_frames = get_free_frames ();
        if (free_frames != last_free_frames) {
            printf ("%d.%d] swapper: %d free frames. threshold = %d."
                    , pid, counter++, free_frames, THRESHOLD);
            if (free_frames < THRESHOLD) {
                printf ("calling free_a_frame (%d < %d)\n", free_frames, THRESHOLD);
                free_a_frame ();
            } else {
                printf ("\n");
            }
        }
    }
```
We launch that program from the init process which guarantees that it will always have process ID 2; we also prevent that process against being killed.

13.4 Serial Hard Disk

Talking to device controllers requires knowledge of the protocols which those controllers understand. In later sections you will see how this is done for floppy and hard disk controllers, but we start with an example that is easier to understand, though it introduces a “device” that does not exist in real life: the serial hard disk. We provide support for a hypothetical disk which is connected to a serial port, and we can only make it work in an emulated machine (or with a second PC which takes the part of the serial disk).

Testing this code requires that you

- run Ulx in a virtual machine which supports two serial ports and that you
- add an external program which connects to the (virtual) second serial port, accepting commands and sending data back and forth.

It takes a little more than that, though, since we want to emulate the “normal” behavior of a disk controller. In real life, transfers use DMA (direct memory access). At the lowest level, the disk driver creates a DMA_READ or DMA_WRITE message and sends it to the controller. By itself, neither of these is a blocking action, since the disk controller will handle the transfer of data from the hard disk to memory (reading) or from memory to the disk (writing) independently of the CPU which continues executing. However, the process which initiated the transfer must be blocked anyway, since reading from the disk will take a while (and writing might not be safe if it continued and possibly changed the data which are currently written). After completion the disk controller creates an interrupt, the corresponding interrupt handler starts and puts the process back into the ready queue.

The actual DMA transfers work with physical memory addresses, so code using DMA must always know where data is or will be stored in physical memory.

Our serial hard disk works differently, it uses in and out commands to read or write single bytes through the serial port, and it can use virtual memory. In a simple implementation of this method the process would never block, it would just take a while to send or receive the data, and the scheduler would switch back and forth between this process and others.

In order to emulate “proper” disk controller behavior we take the following steps:

- Each time that a process starts a disk read/write operation, we create a special buffer for this transfer (which knows what data to send in what direction) and put it in a
disk queue; we then block the process. We limit the queue size so that no more than 100 processes may create an entry at the same time (the 101st process would fail and exit).

- Sending data via the serial port can happen immediately, while receiving depends on the other side (our external process). When the other side sends a byte, it causes an interrupt for the serial port, and inside the Unix interrupt handler we fill a different buffer with that byte. So when “reading” in the timer handler, we don’t actually talk to the serial port, but instead just copy data from one buffer to another.
- We do not allow a read and a write operation at the same time, since this would overcomplicate matters. Instead at each moment, we either read a whole sector, write one or do not access the serial disk at all.
- We add extra functions for non-blocking data transfer, because when we let the kernel (not processes) access the disk, we have nothing that we can block. Since this is the easier type of transfer, we start with it.

### 13.4.1 Kernel Code for the Serial Disk

We start with defining the buffer (which we create as a ring buffer):

```c
struct serial_disk_buffer_entry {
    int pid; // process ID; -1 if kernel
    short status; // New, Transfer, Finished, see BUF_STAT_*
    short direction; // 100 = read_, 101 = write
    unsigned int secno; // sector number
    memaddress address; // memory address (in process' address space)
    byte sector[BLOCK_SIZE]; // 1024 bytes
};
```

Defines:
- `serial_disk_buffer_entry`, used in chunks 516, 517c, and 520c.
Uses `BLOCK_SIZE` 440a and `memaddress` 46c.

```c
#define BUF_STAT_NEW 0
#define BUF_STAT_TRANSFER 1
#define BUF_STAT_FINISHED 2
#define BUF_READ 100
#define BUF_WRITE 101
#define SER_BUF_SIZE 100
```

Defines:
- `BUF_READ`, used in chunks 517c, 518d, 520c, and 522e.
- `BUF_STAT_FINISHED`, used in chunks 518 and 521a.
- `BUF_STAT_NEW`, used in chunk 516d.
- `BUF_WRITE`, used in chunks 517c, 518d, 520c, and 522e.
- `SER_BUF_SIZE`, used in chunks 516, 518, and 521a.

The buffer is just an array with `SER_BUF_SIZE` 515b buffer entries, and we mark its current use with two integers which remember its current start and end:
[516a] \( \langle \text{global variables} \ 92b \rangle + \equiv \) (44a) \(<509a \ 516b >

\begin{verbatim}
struct serial_disk_buffer_entry serial_disk_buffer[SER_BUF_SIZE];
int serial_disk_buffer_start = 0;  // initialize start and end of buffer usage
int serial_disk_buffer_end = 0;   // interval in use is [start_, end],
\end{verbatim}

\[ \equiv \]

\begin{verbatim}
// [0,0] is empty
\end{verbatim}

Defines:
serial_disk_buffer, used in chunks 516d, 517c, 519d, and 520c.
serial_disk_buffer_end, used in chunks 516d, 517c, and 520c.
serial_disk_buffer_start, used in chunks 516–21.

Uses SER_BUF_SIZE 515b and serial_disk_buffer_entry 515a.

This way we can always check whether the buffer is empty by testing if the two variables
serial_disk_buffer_start_{516a} and serial_disk_buffer_end_{516a} are equal.

The buffer shall be protected by a lock:

[516b] \( \langle \text{global variables} \ 92b \rangle + \equiv \) (44a) \(<516a \ 517a >

\begin{verbatim}
lock serial_disk_lock;
\end{verbatim}

Defines:
serial_disk_lock, used in chunks 516, 517c, and 520c.
Uses lock 365a.

[516c] \( \langle \text{initialize kernel global variables} \ 184d \rangle + \equiv \) (44b) \(<363d >

\begin{verbatim}
serial_disk_lock = get_new_lock("serial disk");
\end{verbatim}

Uses get_new_lock 367b and serial_disk_lock 516b.

Next we provide a function with which we can enter a new entry in the buffer:

[516d] \( \langle \text{function implementations} \ 100b \rangle + \equiv \) (44a) \(<512c \ 517b >

\begin{verbatim}
int serial_disk_enter (int pid, short direction, uint secno, uint address) {

mutex_lock (serial_disk_lock);

// check if buffer is full
if ( (serial_disk_buffer_end+1) % SER_BUF_SIZE == serial_disk_buffer_start ) {

mutex_unlock (serial_disk_lock);
return -1;  // fail
}

struct serial_disk_buffer_entry *entry;
entry = &serial_disk_buffer[serial_disk_buffer_end];
entry->status = BUF_STAT_NEW; entry->pid = pid;
entry->direction = direction; entry->secno = secno;
entry->address = address;
short tmp = serial_disk_buffer_end;
serial_disk_buffer_end = (serial_disk_buffer_end+1) % SER_BUF_SIZE;
mutex_unlock (serial_disk_lock);
return tmp;  // tell the caller what entry number we used
}
\end{verbatim}

Defines:
serial_disk_enter, used in chunks 518d and 522e.

Uses BUF_STAT_NEW 515b, mutex_lock 366a, mutex_unlock 366c, SER_BUF_SIZE 515b, serial_disk_buffer 516a,
serial_disk_buffer_end 516a, serial_disk_buffer_entry 515a, serial_disk_buffer_start 516a,
and serial_disk_lock 516b.
13.4 Serial Hard Disk

13.4.1.1 Non-Blocking Read/Write Operations

Now it is time to provide the non-blocking functions for reading and writing. We combine them in one function which does the appropriate thing, based on the buffer entry’s direction field. The function takes no arguments since it finds all the necessary information in the buffer entry.

\[
\begin{align*}
\text{volatile int serial_disk_reader = 0; } & \quad \text{// are we currently reading?} \\
\end{align*}
\]

Defines:
- `serial_disk_reader`, used in chunks 518, 519, and 521.

When we want to send a sector number (as part of a request) we have to split it into bytes; a sector number is a 32 bit wide integer, so four bytes are needed:

\[
\begin{align*}
\text{void serial_disk_send_sector_number (uint secno) {} } & \quad \text{// lowest byte} \\
& \quad \text{secno /= 256; uart2putc ((byte)(secno % 256));} \\
& \quad \text{// 2nd lowest byte} \\
& \quad \text{secno /= 256; uart2putc ((byte)(secno % 256));} \\
& \quad \text{// 3rd lowest byte} \\
& \quad \text{secno /= 256; uart2putc ((byte)(secno % 256));} \\
& \quad \text{// highest byte} \\
\end{align*}
\]

Defines:
- `serial_disk_send_sector_number`, used in chunks 518 and 521.

Uses `uart2putc`.

The next function reads or writes a buffer.

\[
\begin{align*}
\text{int serial_disk_non_blocking_rw ()} & \quad \text{// we don't block} \\
& \quad \text{if (serial_disk_buffer_start == serial_disk_buffer_end)} \{ \\
& \quad \quad \text{mutex_unlock (serial_disk_lock); return -1; } \quad \text{// buffer is empty} \\
& \quad \}
\end{align*}
\]

Defines:
- `serial_disk_non_blocking_rw`, used in chunk 518d.

Uses `BUF_READ` and `BUF_WRITE`.

Writing is the simpler task: we only send the write command and the data via the serial port; we need not wait for a response since the serial port controller will not send one.
Disk I/O

(serial disk: write a buffer)\(^{518a}\)

\[
\text{uart2putc} (\text{CMD\_PUT}); \ \text{serial\_disk\_send\_sector\_number} (\text{entry}\rightarrow\text{secno}); \\
\text{byte} *\text{addressptr} = (\text{byte}*)\text{(entry}\rightarrow\text{address}); \\
\text{for} (\text{int} i = 0; i < 1024; i++) \{ \\
\quad \text{uart2putc} (*\text{addressptr}); \text{addressptr}++; \\
\}
\]

\text{entry}\rightarrow\text{status} = \text{BUF\_STAT\_FINISHED}; \\
\text{serial\_disk\_buffer\_start}++; \\
\text{serial\_disk\_buffer\_start} \%= \text{SER\_BUF\_SIZE};

Uses \text{BUF\_STAT\_FINISHED} 515b, \text{CMD\_PUT} 519a, \text{SER\_BUF\_SIZE} 515b, \text{serial\_disk\_buffer\_start} 516a, \\
\text{serial\_disk\_send\_sector\_number} 517b, and \text{uart2putc} 345c.

\text{CMD\_PUT}\(^{519a}\) will be defined soon; along with \text{CMD\_GET}\(^{519a}\) it is used to tell the serial disk \\
whether we initiate a write or read operation.

Reading is more complicated and requires the help of an interrupt handler; in the non- 
blocking implementation we do not put processes to sleep while a read operation is active. 
Instead we simply wait for its completion by repeatedly using the CPU instruction \text{hlt}.

(serial disk: read a buffer)\(^{518b}\)

\[
\text{uart2putc} (\text{CMD\_GET}); \ \text{serial\_disk\_send\_sector\_number} (\text{entry}\rightarrow\text{secno}); \\
\text{serial\_disk\_reader} = 1; \quad \text{// we're in read mode,} \\
\text{// this value will be changed in the IRQ handler} \\
\text{while} (\text{serial\_disk\_reader} == 1) \text{asm} ("\text{hlt}"); \quad \text{// wait for data} \\
\text{entry}\rightarrow\text{status} = \text{BUF\_STAT\_FINISHED}; \\
\text{serial\_disk\_buffer\_start}++; \\
\text{serial\_disk\_buffer\_start} \%= \text{SER\_BUF\_SIZE}; \\
\text{// copy buffer to target memory location} \\
\text{memcpy} ((\text{char}*)\text{(entry}\rightarrow\text{address}), (\text{char}*)\text{(entry}\rightarrow\text{sector}), \text{BLOCK\_SIZE});
\]

Uses \text{BLOCK\_SIZE} 440a, \text{BUF\_STAT\_FINISHED} 515b, \text{CMD\_GET} 519a, \text{memcpy} 596c, \text{SER\_BUF\_SIZE} 515b, \\
\text{serial\_disk\_buffer\_start} 516a, \text{serial\_disk\_reader} 517a, \text{serial\_disk\_send\_sector\_number} 517b, \\
and \text{uart2putc} 345c.

Next we combine our functions to provide non-blocking read and write functions for 
the kernel (\text{nb} is short for “non-blocking”):

(function prototypes)\(^{45a}\)

\[
\text{void readblock\_nb\_serial} (\text{int} \text{secno}, \text{char} *\text{buf}); \\
\text{void writeblock\_nb\_serial} (\text{int} \text{secno}, \text{char} *\text{buf});
\]

(function implementations)\(^{100b}\)

\[
\text{void readblock\_nb\_serial} (\text{int} \text{secno}, \text{char} *\text{buf}) \{ \\
\quad \text{int} \text{pid}; \text{if} (\text{scheduler\_is\_active}) \text{pid} = \text{current\_task}; \text{else} \text{pid} = -1; \\
\quad \text{serial\_disk\_enter} (\text{pid}, \text{BUF\_READ}, \text{secno}, (\text{uint})\text{buf}); \\
\quad \text{serial\_disk\_non\_blocking\_rw} (); \\
\}
\]

\[
\text{void writeblock\_nb\_serial} (\text{int} \text{secno}, \text{char} *\text{buf}) \{ \\
\quad \text{int} \text{pid}; \text{if} (\text{scheduler\_is\_active}) \text{pid} = \text{current\_task}; \text{else} \text{pid} = -1; \\
\quad \text{serial\_disk\_enter} (\text{pid}, \text{BUF\_WRITE}, \text{secno}, (\text{uint})\text{buf}); \\
\quad \text{serial\_disk\_non\_blocking\_rw} (); \\
\}
\]
13.4 Serial Hard Disk

Defines:
- writeblock_nb_serial, used in chunk 518c.

Uses BUF_READ 515b, BUF_WRITE 515b, current_task 192c, scheduler_is_active 276e, serial_disk_enter 516d, and serial_disk_non_blocking_rw 517c.

These are the commands which we can send to the external controller process:

\[\text{⟨serial-hd/serial-hd-controller.h 519a⟩= (519b) [519a]}\]

\[
\begin{align*}
\text{#define CMD_STAT} & \quad 1 \quad \text{// status query} \\
\text{#define CMD_GET} & \quad 2 \quad \text{// GET a block (1024 bytes)} \\
\text{#define CMD_PUT} & \quad 3 \quad \text{// PUT a block (1024 bytes)} \\
\text{#define CMD_TERM} & \quad 99 \quad \text{// terminate controller}
\end{align*}
\]

Defines:
- CMD_GET, used in chunks 518b and 521a.
- CMD_PUT, used in chunk 518a.

We use them both in the UUT code as well as in the controller program.

\[\text{⟨constants 112a⟩= (44a) 〈515b 521c〉 [519b]}\]

\[\langle\text{serial-hd/serial-hd-controller.h 519a}\rangle\]

13.4.1.2 The Interrupt Handler

The interrupt handler serial_hard_disk_handler_{519d} copies a byte from the serial port into the buffer, and if the buffer is full, it resets the serial_disk_reader_{517a} variable to indicate that a whole block (of 1024 bytes) has been transferred.

\[\text{⟨global variables 92b⟩= (44a) 〈517a 522a〉 [519c]}\]

\[\begin{align*}
\text{char} & \quad \text{serial_hard_disk_buffer}[1024]; \\
\text{int} & \quad \text{serial_hard_disk_pos} \quad = \ 0; \\
\text{boolean} & \quad \text{serial_hard_disk_blocks} \quad = \text{false};
\end{align*}\]

Defines:
- serial_hard_disk_blocks, used in chunks 517c, 519d, and 520c.
- serial_hard_disk_buffer, used in chunk 519d.
- serial_hard_disk_pos, used in chunk 519d.

We will also have to read from the second serial port, so we provide a uart2getc function which reads a single character from that port. There is no corresponding uartgetc function for the first port, but it would look identical, except for using uart_{344b}[0] and IO_COM1_{344a} instead of uart_{344b}[1] and IO_COM2_{344a}:

\[\text{⟨function implementations 100b⟩= (44a) 〈518d 520c〉 [519d]}\]

\[\begin{align*}
\text{static int uart2getc ()} \{ \\
\quad \text{if (!uart[1])} \{ \text{return -1; } \} \\
\quad \text{if (!inportb (IO_COM2+5) & 0x01)}) \{ \text{return -1; } \} \\
\quad \text{return inportb (IO_COM2+0);}
\}
\]

\[
\text{void serial_hard_disk_handler (context_t *r)} \{ \\
\quad \text{char c = uart2getc ();} \\
\quad \text{serial_hard_disk_buffer[serial_hard_disk_pos++] = c;} \\
\quad \text{if (serial_hard_disk_pos == 1024) } \{ \\
\quad \quad \text{serial_hard_disk_pos = 0; }
\}
\]
// copy buffer to proper serial hard disk buffer
memcpy ( & (serial_disk_buffer[serial_disk_buffer_start].sector),
         &serial_hard_disk_buffer, 1024 );
serial_disk_reader = 0; // reading a sector is finished
if (serial_hard_disk_blocks) { ( serial hard disk: wake process 522c ) }
}

Defines:
serial_hard_disk_handler, used in chunk 520a.
Uses context_t 142a, inportb 133b, IO_COM2 344a, memcpy 596c, serial_disk_buffer 516a, serial_disk_buffer_start 516a, serial_disk_reader 517a, serial_hard_disk_blocks 519c, serial_hard_disk_buffer 519c, serial_hard_disk_pos 519c, and uart 344b.

Finally we enter this interrupt handler in the handler list and enable the interrupt.

[520a] (setup serial hard disk 345d) += (45c) <345d
    install_interrupt_handler (IRQ_COM2, serial_hard_disk_handler);
    enable_interrupt (IRQ_COM2);
Uses enable_interrupt 140b, install_interrupt_handler 146c, IRQ_COM2 132, and serial_hard_disk_handler 519d.

Note that we’re executing a code chunk (serial hard disk: wake process 522c) if we’re currently working on a request for which blocking was enabled. We explain this in the next subsection.

13.4.1.3 Blocking Read/Write Operations

For a multitasking system it is inacceptable to work with blocking I/O operations, at least for processes. We will now implement the blocking read and write functions. For writing there is no difference (because the transfer commands to the serial port finish immediately), but for reading we will put the calling process to sleep until a whole block of data has been read. The function

[520b] (function prototypes 45a) += (44a) <518c 522d>
    int serial_disk_blocking_rw ();
looks just like serial_disk_no_blocking_rw with two differences:
• It sets serial_hard_disk_blocks 519c to true (to indicate that we want to block),
• and in the switch expression it uses a fresh code chunk for reading.

[520c] (function implementations 100b) += (44a) <519d 522e>
    int serial_disk_blocking_rw () {
        mutex_lock (serial_disk_lock);
        serial_hard_disk_blocks = true; // we block
        if (serial_disk_buffer_start == serial_disk_buffer_end) {
            mutex_unlock (serial_disk_lock); return -1; // buffer is empty
        }
        struct serial_disk_buffer_entry *entry;
        entry = &serial_disk_buffer[serial_disk_buffer_start];
        switch (entry->direction) {
            case BUF_WRITE: ( serial disk: write a buffer 518a ); break;
        }
    }
case BUF_READ: ⟨serial disk: read a buffer and block 521a⟩; break;
default: mutex_unlock (serial_disk_lock); return -1;
}
mutex_unlock (serial_disk_lock);
return 0;
}

Defines:
serial_disk_blocking_rw, used in chunks 520b and 522e.
Uses BUF_READ 515b, BUF_WRITE 515b, mutex_lock 366a, mutex_unlock 366c, serial_disk_buffer 516a,
serial_disk_buffer_end 516a, serial_disk_buffer_entry 515a, serial_disk_buffer_start 516a,
serial_disk_lock 516b, and serial_hard_disk_blocks 519c.

The difference between the ⟨serial disk: read a buffer 518b⟩ and the following new code chunk is that we don’t do busy waiting (as above) but the process to sleep. Only one line was changed (marked with [*]).

⟨serial disk: read a buffer and block 521a⟩≡ (520c) [521a]
uart2putc (CMD_GET);
⟨begin critical section in kernel 380a⟩
serial_disk_send_sector_number (entry->secno);
serial_disk_reader = 1; // we're in read mode,
// this value will be changed in the IRQ handler
while (serial_disk_reader == 1) {
⟨serial disk: put process to sleep 521b⟩  // [*]
entry->status = BUF_STAT_FINISHED;
serial_disk_buffer_start++;
serial_disk_buffer_start %= SER_BUF_SIZE;
// copy buffer to target memory location
memcpy ((char*)entry->address, (char*)entry->sector, BLOCK_SIZE);
}⟨end critical section in kernel 380b⟩

Uses BLOCK_SIZE 440a, BUF_STAT_FINISHED 515b, CMD_GET 519a, memcpy 596c, SER_BUF_SIZE 515b,
serial_disk_buffer_start 516a, serial_disk_reader 517a, serial_disk_send_sector_number 517b,
and uart2putc 345c.

In the non-blocking code we simply executed the assembler instruction hlt in the loop, we actively waited for the transfer to complete. Here we put the process to sleep:

⟨serial disk: put process to sleep 521b⟩≡ (521a) [521b]
if (scheduler_is_active) {
// we access thread table; interrupts are off
block (&serial_disk_queue, TSTATE_WAITSD);
⟨end critical section in kernel 380b⟩
⟨resign 221d⟩
} else {
⟨end critical section in kernel 380b⟩
}

Uses scheduler_is_active 276e, serial_disk_queue 522a, and TSTATE_WAITSD 521c.

We define the new state TSTATE_WAITSD 521c and the serial_disk_queue 522a blocked queue:

⟨constants 112a⟩≡ (44a) <519b 525a> [521c]
#define TSTATE_WAITSD 12

Defines:
TSTATE_WAITSD, used in chunks 521b and 564c.
The queue must also be initialized:

```c
initializeBlockedQueue (&serial_disk_queue);
```

Since we need to wake the process up when the block was transmitted, we add the wake-up call to the interrupt handler. We've included the following code chunk in the serial_hard_disk_handler function:

```c
if (scheduler_is_active) {
    int tid;
    if ((tid = serial_disk_queue.next) != 0)
        deblock (tid, &serial_disk_queue);
}
```

Defines:
- `readblock_serial`, used in chunk 506b.
- `writeblock_serial`, used in chunks 507b and 522d.

Uses `BUF_READ`, `BUF_WRITE`, `current_task`, `scheduler_is_active`, and `serial_disk_blocking_rw`.

### 13.4.2 The External Controller Process

The controller is a simple program that opens a TCP socket to talk to the serial port of the emulated PC that executes Ulix. The functions `readsect` and `writesect` transfer individual
blocks. The program only reacts to requests that come from the ULIX machine. In that way it simulates the behavior of a disk controller.

```c
#include <fcntl.h> // open()
#include <sys/types.h> // socket()
#include <sys/socket.h> // socket()
#include <netinet/in.h> // socket()
#include <unistd.h> // close()
#include <string.h> // bzero()
#include <stdio.h>
#include <unistd.h> // lseek: SEEK_SET
#include "serial-hd-controller.h"

int socks; // socket descriptor for ULIX connection
int fd = -1; // file descriptor
int numsec = -1; // number of sectors (1024 bytes) in disk image
byte sector[BLOCK_SIZE];

void readsocket (byte *buf, short len) {
    // We use this instead of recv(), since recv() does not always
    // read the expected number of bytes.
    int total = 0;
    while (total < len) {
        total += recv (socks, buf+total, len-total, 0);
    }
}

void openfile () { fd = open ("minix1.img", O_RDWR); numsec = 2880; }
void closefile () { close (fd); fd = -1; numsec = -1; }

void readsect (int i) {
    lseek (fd, i*BLOCK_SIZE, SEEK_SET); // get sector from disk image
    int res = read (fd, &sector, BLOCK_SIZE);
    send (socks, &sector, BLOCK_SIZE, 0); // send it to ULIX
}

void writesect (int i) {
    readsocket ((byte*) &sector, BLOCK_SIZE); // get sector from ULIX
    lseek (fd, i*BLOCK_SIZE, SEEK_SET); // write it to disk image
    write (fd, &sector, BLOCK_SIZE);
}

int main () {
    openfile (); // open disk image
    socks = socket (AF_INET, SOCK_STREAM, 0); // connect to localhost:4444
    struct sockaddr_in serveraddr;
    bzero (&serveraddr, sizeof (serveraddr));
```
In order to connect the external process to qemu (running Ulix), we start qemu as follows:

\[
\text{qemu} -m 64 -fda ulix-fd0.img -d cpu_reset -s -serial mon:stdio \ 
-serial tcp::4444,server
\]

Note that there are two `-serial` arguments; the first one connects COM1 with the terminal from which qemu was started; the second one connects COM2 with a TCP server on port 4444. That's the one our external program is going to connect to.

The final release of Ulix does not use the serial hard disk any more because its dependency on the external controller program made using the system uncomfortable. In the following two sections we present our hard disk and floppy disk drivers.
13.5 The Hard Disk Controller

As mentioned before, we will let our hard disk driver use non-DMA data transfer, called **PIO** (programmed input/output). The code in this section is based on the IDE driver code of the xv6 operating system [CKM12].

### 13.5.1 Sending Commands to the Controller

Communication with a device always needs to follow strict protocols, this holds for the hard disk controller, too. The following description is full of technical details about controller-internal registers and the ports used to access them, it also contains some assembler code. If you want to skip this, here’s a summary: in this subsection we’ll define two code chunks ⟨ide: read sector sector on device hd 527b⟩ and ⟨ide: write sector sector on device hd 527c⟩ which can be used for sending the controller the commands for initiating the transfer and for copying a sector from memory to the controller’s internal memory. The other direction (from the controller’s memory to RAM) will be dealt with inside the interrupt handler which we’ll discuss in one of the following subsections.

We’ll define some constants which will be used in the following code: The interrupt number for the (first) IDE controller is 14 (IRQ_IDE). The controller accepts two commands for reading (0x20; IDE_CMD_READ) and writing (0x30; IDE_CMD_WRITE), and when we query the controller’s status, there are four possible results which we’ll be prepared to handle (busy, data ready, device fault and error):

```c
#define IDE_CMD_READ 0x20 // read from disk, with retries
#define IDE_CMD_WRITE 0x30 // write to disk, with retries
#define IDE_CMD_IDENT 0xec // identify disk
#define IDE_BSY 0x80 // 0b10000000 (bit 7), device busy
#define IDE_DRDY 0x40 // 0b01000000 (bit 6), device ready
#define IDE_DF 0x20 // 0b00100000 (bit 5), drive fault
#define IDE_ERR 0x01 // 0b00000001 (bit 0), error
```

Defines:
IDE_BSY, used in chunk 533b.
IDE_CMD_IDENT, used in chunk 534b.
IDE_CMD_READ, used in chunk 527b.
IDE_CMD_WRITE, used in chunk 527c.
IDE_DF, used in chunk 533b.
IDE_DRDY, used in chunk 533b.
IDE_ERR, used in chunk 533b.
IRQ_IDE, used in chunk 534b.

In order to talk to the controller we use the ports 0x1f0 – 0x1f7 (the port numbers can be found in Seagate’s ATA Interface Reference Manual [Sea93, p. 13]), we give them names to make things easier:

```c
// IDE output
#define IO_IDE_SEC_COUNT 0x1f2 // sector count register (read_/write_)
#define IO_IDE_SECTOR 0x1f3 // (32 bits in 0x1f3..0x1f6)
#define IO_IDE_DISKSEL 0x1f6 // disk select and upper 4 bits of sector no.
```
#define IO_IDE_COMMAND 0x1f7 // command register
#define IO_IDE_DEVCTRL 0x3f6 // device control register

// IDE input
#define IO_IDE_DATA 0x1f0 // data (read/write)
#define IO_IDE_STATUS 0x1f7 // status register (identical to command reg.)

Defines:
IO_IDE_COMMAND, used in chunks 527 and 534b.
IO_IDE_DATA, used in chunks 532 and 534b.
IO_IDE_DEVCTRL, used in chunk 526.
IO_IDE_DISKSEL, used in chunks 526 and 534b.
IO_IDE_SEC_COUNT, used in chunk 526.
IO_IDE_SECTOR, used in chunk 527a.
IO_IDE_STATUS, used in chunks 532–34.

Note that IO_IDE_COMMAND and IO_IDE_STATUS are the same port number (0x1f7), but depending on the type of access, they refer to different registers: When reading that port, we access the status register, and when writing, we access the command register.

The read and write commands are specified in the “AT Attachment Interface for Disk Drives” document [Lam94, p. 40]; 0x20 and 0x30 are the read/write commands which trigger data transfer with retries (in case of errors); there are also further commands (0x21, 0x31) which trigger corresponding reads or writes without retries. The kernel can send a command by writing the required value into the controller’s command register via the IO_IDE_COMMAND port 0x1f7 (outb IDE_CMD_READ, 0x1f7).

The status values represent the bit positions 7 (0x80 = 2^7), 6 (0x40 = 2^6), 5 (0x20 = 2^5) and 0 (0x01 = 2^0) of the status register [Lam94, p. 34], see Table 13.2.

The read and write commands can make the disk read/write several sectors with one command. To start such a read or write operation, we need to tell the controller three things:

- How many sectors shall be read/written? This information must be stored in the sector count register which is accessible via the IO_IDE_SEC_COUNT port (0x1f2). We will always read or write just a single sector:

```c
(id: read/write sector on device hd 526)  526
    idewait (0);
    outportb (IO_IDE_DISKSEL, 0xe0 | (hd<<4)); // select disk
    outportb (IO_IDE_DEVCTRL, 0);               // generate interrupt
    outportb (IO_IDE_SEC_COUNT, 1);             // one sector
```

Uses idewait 533b, IO_IDE_DEVCTRL 525b, IO_IDE_DISKSEL 525b, IO_IDE_SEC_COUNT 525b, and outportb 133b.

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BSY</td>
<td>DRDY</td>
<td>DWF</td>
<td>DSC</td>
<td>DRQ</td>
<td>CORR</td>
<td>IDX</td>
<td>ERR</td>
</tr>
</tbody>
</table>

Table 13.2: The status register of the IDE controller.
• Via the four ports 0x1f3–0x1f6, we can specify the 28 bits of a sector number (four bits are used for selecting the drive, making the target register 32 bits wide):

\[
\begin{align*}
\langle ide: \text{read/write sector on device hd 526}\rangle &+≡ \langle ide: \text{read/write sector on device hd 526}\rangle \\
\text{outportb (IO_IDE_SECTOR, sector } & \& \text{ 0xff)}; \\
\text{outportb (IO_IDE_SECTOR+1, (sector } & \gg 8) \& \text{ 0xff)}; \\
\text{outportb (IO_IDE_SECTOR+2, (sector } & \gg 16) \& \text{ 0xff)}; \\
\text{outportb (IO_IDE_SECTOR+3, ((sector } & \gg 24) \& \text{ 0x0f}) | ((0xe + hd) \ll 4) );}
\end{align*}
\]

Uses IO_IDE_SECTOR 525b and outportb 133b.

• Finally, we send the IDE_CMD_READ525a or IDE_CMD_WRITE525a command to the controller via the IO_IDE_COMMAND525b port 0x1f7:

\[
\begin{align*}
\langle ide: \text{read sector on device hd 527b}\rangle &≡ \langle ide: \text{read sector on device hd 526}\rangle \\
\text{outportb (IO_IDE_COMMAND, IDE_CMD_READ});
\end{align*}
\]

Uses IDE_CMD_READ 525a, IO_IDE_COMMAND 525b, and outportb 133b.

\[
\begin{align*}
\langle ide: \text{write sector on device hd 527c}\rangle &≡ \langle ide: \text{write sector on device hd 526}\rangle \\
\text{outportb (IO_IDE_COMMAND, IDE_CMD_WRITE)};
\end{align*}
\]

Uses IDE_CMD_WRITE 525a, IO_IDE_COMMAND 525b, and outportb 133b.

Using 28 bits for the sector number allows us to access \(2^{28} = 268,435,456\) sectors, thus the maximum disk size is \(2^{28} \times 512 = 137,438,953,472\) bytes (128 GByte) which should be enough for most UNIX uses ... This is called LBA28 (Logical Block Addressing, 28 bits). In 2002, ATA-6 [McL02], the sixth version of the ATA standard, introduced LBA48 which uses 48 bits to specify sector numbers and allows for much larger disks.

Our IDE driver will not use DMA transfer but instead copy the bytes with in and out operations, directly talking to the controller.

Now we’re able to send read/write commands to the hard disk controller, and it will start servicing those requests immediately. But what happens when the disk has read the sector contents and copied them into the controller’s internal memory? We need to fetch the data.

The controller is helpful in that it will tell us when it finished its work: it will raise the IRQ_IDE132 interrupt (number 14), and our interrupt handler must then copy the data from the controller to RAM.

We do this by directly reading the data via the controller’s data register (i.e., by reading from IO_IDE_DATA525b). The CPU instruction insl can read four bytes (a 32 bit value) in one go. For reading the contents of a whole sector we would need \(512/4 = 128\) of these insl instructions, but the processor has a way of repeating this command automatically: if we add a rep prefix to insl, we get repeated executions of insl without manually writing a loop.

The logic of rep insl requires us to fill some registers with proper values:

• The memory address (our buffer) goes into the EDI register (after each step, EDI will be incremented by 4),
• the number of repetitions must be stored in the ECX register (after each step, ECX will be decremented; the loop continues while ECX ≠ 0),
• and the port number must be stored in the (16 bit) DX register.

Thus, in C-like pseudo code, rep insl does the following:

[528a] ⟨pseudo code for rep insl 528a⟩≡

while (%ecx != 0) {
  *(%edi) = inportl (%ecx); // read 4 bytes, write them to *(%edi)
  %edi += 4; // update target memory
  %ecx--; // decrement counter
}

In the inline assembler language the registers ECX, DX and EDI can be accessed using "c", "d" and "D", respectively (see Appendix B for an introduction to gcc inline assembler). rep can either auto-increment or auto-decrement the target address with each step; in order to make it increment (like we want), we need to set the direction flag of the EFLAGS register to 0 using the machine instruction cld (clear direction flag).

The following function definition sets everything up properly and executes rep insl:

[528b] ⟨function prototypes 45a⟩≡

static inline void repeat_inportsl (int port, void *addr, int cnt);

[528c] ⟨function implementations 100b⟩≡

static inline void repeat_inportsl (int port, void *addr, int cnt) {
  asm volatile ("cld \n"
    "rep insl" : 
    ";=D" (addr), ";=c" (cnt) :
    ";d" (port), ";0" (addr), ";1" (cnt) :
    "memory", "cc");
}

Defines:
repeat_inportsl, used in chunks 528b, 532d, and 534b.

With "0" and "1" we refer to the first two registers used, that is, "D" (EDI) and "c" (ECX); see also Appendix B.4.

For the other direction, we provide a repeat_outportsl function which looks almost identical:

[528d] ⟨function prototypes 45a⟩≡

static inline void repeat_outportsl (int port, void *addr, int cnt);

[528e] ⟨function implementations 100b⟩≡

static inline void repeat_outportsl (int port, void *addr, int cnt) {
  asm volatile ("cld \n"
    "rep outsl" : 
    ";=S" (addr), ";=c" (cnt) :
    ";d" (port), ";0" (addr), ";1" (cnt) :
    "cc");
}

Defines:
repeat_outportsl, used in chunks 528d and 532b.
Instead of EDI, the outsl instruction expects the ESI register to contain the memory address, which is why we wrote "=S(addr)" instead of "=D(addr)".

### 13.5.2 The Blocked Queue

We define a harddisk_queue which will contain processes waiting for a hard disk access operation to finish:

```
// processes which wait for the hard disk
```

Defines:
- harddisk_queue, used in chunks 529b, 531a, 532d, and 564c, and 606.

Uses blocked_queue 183a.

and we have to initialize this queue:

```
initialize_blocked_queue (&harddisk_queue);
```

Processes that access a hard disk drive will wait on this queue, and the interrupt handler will wake up these processes when the operation was completed.

### 13.5.3 Reading and Writing

Now we use the code presented so far to create two functions

```
void readblock_hd (int hd, int blockno, char *buffer);
void writeblock_hd (int hd, int blockno, char *buffer);
```

which read and write a complete block (1024 bytes). They will use a buffer hd_buf which can contain one sector (512 bytes) and must be protected by a lock. We also declare a global variable hd_direction which we set to HD_OP_READ or HD_OP_WRITE when we initialize a read or write operation.

```
#define HD_OP_READ 0
#define HD_OP_WRITE 1
#define HD_OP_NONE -1
```

Defines:
- HD_OP_NONE, used in chunks 530 and 532.
- HD_OP_READ, used in chunks 530c and 532.
- HD_OP_WRITE, used in chunks 530d and 532.
- HD_SECSIZE, used in chunks 530 and 532.

We also need some global variables: hd_buf is a buffer that can store one sector, hd_lock is used for locking disk access, and hd_direction will always be set to one of the HD_OP_* constants to indicate the current transfer direction.
Disk I/O

(global variables 92b) +≡
  char hd_buf[HD_SECSIZE];
  lock hd_lock;
  char hd_direction;

Defines:
  hd_buf, used in chunks 530 and 532.
  hd_direction, used in chunks 530 and 532d.
  hd_lock, used in chunk 530.
Uses HD_SECSIZE 529d and lock 365a.

We initialize the lock at system start-up:

(initialize system 45b) +≡
  hd_lock = get_new_lock ("hard disk");
Uses get_new_lock 367b and hd_lock 530a.

The functions readblock_hd531b and writeblock_hd531b will transfer 1 KByte blocks of data. Since the hard disk controller defaults to transferring 512-bytes-sized sectors, we first provide functions for reading and writing such sectors.

(function implementations 100b) +≡
  void readsector_hd (int hd, int sector, char *buffer) {
    mutex_lock (hd_lock);
    hd_direction = HD_OP_READ;
    ⟨begin critical section in kernel 380a⟩
    ⟨ide: read sector sector on device hd 527b⟩
    while (hd_direction == HD_OP_READ) { ⟨ide: put process to sleep 531a⟩ }
    ⟨ide: read data from the controller 532a⟩
    memcpy (buffer, hd_buf, HD_SECSIZE);
    hd_direction = HD_OP_NONE;
    mutex_unlock (hd_lock);
  }
Uses hd_buf 530a, hd_direction 530a, hd_lock 530a, HD_OP_NONE 529d, HD_OP_READ 529d, HD_SECSIZE 529d, memcpy 596c, mutex_lock 366a, and mutex_unlock 366c.

For writing, the sequence of events is slightly different; we first transfer the data to the controller and then put the process to sleep, letting it wait for the transfer to finish:

(function implementations 100b) +≡
  void writesector_hd (int hd, int sector, char *buffer) {
    mutex_lock (hd_lock);
    hd_direction = HD_OP_WRITE;
    memcpy (hd_buf, buffer, HD_SECSIZE);
    ⟨begin critical section in kernel 380a⟩
    ⟨ide: write sector sector on device hd 527c⟩
    ⟨ide: write data to the controller 532b⟩
    while (hd_direction == HD_OP_WRITE) { ⟨ide: put process to sleep 531a⟩ }
    hd_direction = HD_OP_NONE;
    mutex_unlock (hd_lock);
  }
Defines:
  writesector_hd, used in chunk 531b.
Uses hd_buf 530a, hd_direction 530a, hd_lock 530a, HD_OP_NONE 529d, HD_OP_WRITE 529d, HD_SECSIZE 529d, memcpy 596c, mutex_lock 366a, and mutex_unlock 366c.
Why do we put the process to sleep anyway? While the PIO transfer between system RAM and the controller’s memory wastes some time (and a DMA would save that waste), the actual transfer between controller and disk takes a lot longer—this transfer is what the process must wait for to complete.

- When the process reads from the disk, we cannot get around waiting anyway: the data will not be available before the transfer completes. Putting the process to sleep guarantees that execution of the process only continues after the data have been read (and stored in the buffer that this process has set up for reading).
- In the case of writing, we could in principle let the process continue immediately after sending the data off to the controller. The process need not wait for the controller-to-disk transfer to finish. But if it issued another write request immediately, that would get in the way of the previous one. To make things simpler, we block the process until the write operation is done.

When we put the process to sleep we use the waiting state TSTATE_WAITHD defined on page 180.

```c
if (scheduler_is_active) {
    // interrupts are off; we access the thread table
    block (&harddisk_queue, TSTATE_WAITHD);
    end critical section in kernel
    resign
} else {
    end critical section in kernel
}
```

Uses harddisk_queue, scheduler_is_active, and TSTATE_WAITHD.

The block read/write functions are now implemented as follows:

```c
void readblock_hd (int hd, int blockno, char *buffer) {
    readsector_hd (hd, blockno*2, buffer);
    readsector_hd (hd, blockno*2+1, buffer + HD_SECSIZE);
}

void writeblock_hd (int hd, int blockno, char *buffer) {
    writesector_hd (hd, blockno*2, buffer);
    writesector_hd (hd, blockno*2+1, buffer + HD_SECSIZE);
}
```

Defines:
- readblock_hd, used in chunk 506b.
- writeblock_hd, used in chunks 507b and 529c.

After a read operation has finished (and the controller has generated an interrupt) we can copy the read data from the controller’s memory to the buffer. That transfer happens in the interrupt handler, here we only let the process wait for an interrupt.
(ide: read data from the controller)\]
\[
\text{idewait}(0);\]
\[
inportb(IO_{IDE\_STATUS}); // read status, ack irq\]
Uses idewait 533b, inportb 133b, and IO_{IDE\_STATUS} 525b.

Writing is similar:

(ide: write data to the controller)\]
\[
inportb(IO_{IDE\_STATUS}); // read status, ack irq\]
\[
\text{repeat\_outportsl(IO}_{IDE\_DATA}, \text{hd\_buf, HD\_SECSIZE / 4});\]
\[
inportb(IO_{IDE\_STATUS}); // read status, ack irq\]
Uses hd\_buf 530a, HD\_SECSIZE 529d, inportb 133b, IO_{IDE\_DATA} 525b, IO_{IDE\_STATUS} 525b, and repeat\_outportsl 528e.

Now the only missing bit is the interrupt handler which will only acknowledge the interrupt and possibly wake up a waiting process.

### 13.5.4 Interrupt Handler

The interrupt handler for the IDE controller

\[
\text{void ide\_handler(context\_t *r);}\]

will be executed whenever the controller finishes an operation and signals the CPU. What it has to do then depends on the transfer direction:

- In case of a write operation, \text{writesector\_hd} had filled the buffer, copied the data from there into the controller's memory and asked the controller to start the write operation onto the disk. So when that is finished, the whole operation is completed, and the interrupt handler only has to wake up the waiting process.

- The situation is different during a read operation: In that case the \text{readsector\_hd} function had only asked the controller to read the data from disk and store them in the controller's memory. When the controller signals completion, the sector is waiting there (in the controller memory) to be retrieved. Thus, the interrupt handler must copy the data to system memory. Once it has finished that, it can also wake up the waiting process.

\[
\text{void ide\_handler(context\_t *r)}\}
\[
\text{switch (hd\_direction) \{}\]
\[
\text{case HD\_OP\_READ: repeat\_inportsl \text{(IO\_IDE\_DATA, hd\_buf, HD\_SECSIZE / 4)}; hd\_direction = HD\_OP\_NONE; break;}\]
\[
\text{case HD\_OP\_WRITE: hd\_direction = HD\_OP\_NONE; break;}\]
\[
\text{case HD\_OP\_NONE: printf("Funny IDE interrupt -- no request waiting\n"); return;}\]
if (scheduler_is_active) {
    int tid;
    if (tid = harddisk_queue.next != 0)
        deblock (tid, &harddisk_queue);  // wake up process
}

The last function we need to discuss in the context of reading form or writing to disk is

\[\text{int idewait (int checkerr)};\]

which waits if the IDE controller is not yet ready to receive the next command. It checks the \text{IO_IDE_STATUS} register’s flags DRDY (device ready; bit 6) and BSY (busy; bit 7) and loops until the controller is ready and not busy.

\[\text{int idewait (int checkerr)};\]

\[\begin{align*}
\text{for (;;) \{} \\
\quad \text{r = inportb (IO_IDE_STATUS);} \\
\quad \text{if ((r \& (IDE_BSY | IDE_DRDY)) == IDE_DRDY) break; \// ready, not busy}
\end{align*}\]

\[\text{if (checkerr \&\& (r \& (IDE_DF | IDE_ERR)) != 0) \{} \\
\quad \text{return -1;} \\
\text{\} else \{} \\
\quad \text{if (current_task > 1) \{} \text{\// see comment}
\end{align*}\]

\[\text{return 0;}\]

The last function we need to discuss in the context of reading form or writing to disk is

\[\text{int idewait (int checkerr)};\]

\[\begin{align*}
\text{for (;;) \{} \\
\quad \text{r = inportb (IO_IDE_STATUS);} \\
\quad \text{if ((r \& (IDE_BSY | IDE_DRDY)) == IDE_DRDY) break; \// ready, not busy}
\end{align*}\]

\[\text{if (checkerr \&\& (r \& (IDE_DF | IDE_ERR)) != 0) \{} \\
\quad \text{return -1;} \\
\text{\} else \{} \\
\quad \text{if (current_task > 1) \{} \text{\// see comment}
\end{align*}\]

\[\text{return 0;}\]

\[\text{\} \}\]
13 Disk I/O

[534a] ⟨global variables 92b⟩+≡
ulonglong hd_size[2] = {−1, −1};

Defines:
hd_size, used in chunks 499a and 534b.
Uses ulonglong 46b.

record. If a disk is not available we keep the −1 value.

The ata_init function selects a disk by sending the encoded disk number to the
IO_IDE_DISKSEL port and then asks for identification by sending the IDE_CMD_IDENT command to port IO_IDE_COMMAND. The answer is 512 bytes long and copied into a buffer using the repeat_inportsl function. The disk size is then assembled from four bytes and written to the right hd_size array entry (and displayed in the boot messages).

We also use this function to install the interrupt handler.

[534b] ⟨function implementations 100b⟩+≡

void ata_init () {
    // detect installed hard disks
    word buf[512]; short drivecount = 0;
    char *names[2] = { "hda", "hdb" };
    printf ("ATA: ");
    for (int disk = 0; disk < 2; disk++) {
        outportb (IO_IDE_DISKSEL, 0xe0 | (disk<<4)); // select disk
        for (int i = 0; i < 1000; i++) {
            if (inportb (IO_IDE_STATUS) != 0) {
                drivecount++;
                outportb (IO_IDE_COMMAND, IDE_CMD_IDENT); // identify!
                repeat_inportsl (IO_IDE_DATA, buf, 256); // 512 bytes = 256 words
                hd_size[disk] = (ulonglong)buf[100] + ((ulonglong)buf[101])<<16
                                + ((ulonglong)buf[102])<<32 + ((ulonglong)buf[103])<<48;
                if (drivecount > 1) printf (", ");
                printf ("%s (%d KByte)", names[disk],
                        hd_size[disk]/2); // 512-byte sectors!
                break;
            }
        }
    }
    printf ("\n");
    outportb (IO_IDE_DISKSEL, 0xe0 | (0<<4)); // select disk 0

    // install the interrupt handler
    install_interrupt_handler (IRQ_IDE, ide_handler);
    enable_interrupt (IRQ_IDE);
}

Defines:
ata_init, used in chunk 45c.
Uses enable_interrupt 140b, hd_size 534a, IDE_CMD_IDENT 525a, ide_handler 532d, inportb 133b,
install_interrupt_handler 146c, IO_IDE_COMMAND 525b, IO_IDE_DATA 525b, IO_IDE_DISKSEL 525b,
IO_IDE_STATUS 525b, IRQ_IDE 132 525a, outportb 133b, printf 601a, repeat_inportsl 528c, and ulonglong 46b.
13.6 The Floppy Controller

In the previous sections you have already seen two ways to talk to a device controller:

- We accessed the “serial hard disk” by sending individual bytes across the serial port. In one direction they contained controller commands and data (sectors to be written on the disk), in the other direction only data (sectors read from the disk). Every single received byte caused an interrupt, and so the sector had to be assembled byte by byte.

- For the IDE controller we used some kind of block transfer where we copied a sector between the PC’s memory and the controller’s internal memory. For that purpose we used a global buffer, though that was not strictly necessary; we could have used the memory location that the read/write functions use for storing the sector, i. e., memory that belongs to a process.

The transfer between controller memory and the actual disk was performed by the controller itself, and we had to wait for that transfer to complete. This type of data transfer is called PIO transfer (Parallel I/O).

Now we show you a third way that uses DMA transfer (Direct Memory Access). Here we need to work with a global buffer and we also need to know the physical address of that buffer because the floppy controller will access it directly—it cannot use the MMU to translate a virtual address. Once the controller has been told what to do, the data transfer from or to that buffer happens automatically, no further activity by the CPU is required. That is possible because the controller can access the memory bus (just like the CPU does). This is most interesting in case of a read operation: Our code only has to tell the controller that it shall read a certain sector from the floppy, and the next time the controller generates an interrupt, the data will already be stored in the buffer—we need not call repeat_inports1528c or an equivalent instruction, as in our hard disk driver.

(Note that the IDE controller also supports DMA transfers. We have decided to let it work in PIO mode so that you can see both approaches at work. However, PIO transfers increase the load on the CPU, so if performance was your goal, you would have to replace the PIO code with DMA code.)

13.6.1 Talking to the Controller

The floppy controller has several ports that can be used to communicate with it; either for sending it a command or data or for reading data. These ports are the following:

```c
#define IO_FLOPPY_OUTPUT 0x3f2 // digital output register (DOR)
#define IO_FLOPPY_STATUS 0x3f4 // main status register (MSR)
#define IO_FLOPPY_COMMAND 0x3f5 // command/data register
#define IO_FLOPPY_RATE 0x3f7 // configuration control register
```

Defines:
- `IO_FLOPPY_COMMAND`, used in chunks 536b and 537a.
- `IO_FLOPPY_OUTPUT`, used in chunks 544, 551a, and 552c.
- `IO_FLOPPY_RATE`, used in chunks 542c and 552c.
- `IO_FLOPPY_STATUS`, used in chunks 536b and 537a.
In most cases we will use the function \texttt{fdc\_out} to send a command to the controller and read in the results with \texttt{fdc\_getresults}. These functions work as follows:

```c
void fdc_out (byte data);
int fdc_getresults ();
```

Before we can write to the controller we need to check whether it is ready. We read the status from the status register via the \texttt{IO\_FLOPPY\_STATUS} port. We are only interested in the highest two bits of the status register that tell us whether it is ready (bit 7) and whether it is prepared for a write operation (bit 6). So we mask the returned status value with \texttt{(FLOPPY\_MASTER \texttt{\mid} FLOPPY\_DIRECTION)}:

```c
#define FLOPPY\_DIRECTION 0b01000000 // bit 6 of status reg.
#define FLOPPY\_MASTER 0b10000000 // bit 7 of status reg.
```

If only bit 7 is set in the resulting value, then we know that the controller is ready and expects a write operation. Then we can send the byte to the \texttt{data/command register} via the \texttt{IO\_FLOPPY\_COMMAND} port; otherwise we loop until the status changes to what we need.

Sometimes a single byte (that was sent to the controller) constitutes a complete command, but often we need to send a sequence. The controller can tell from the first byte how many more bytes follow. When the command is complete, the controller executes it and generates a result that may consist of a sequence of bytes as well.

If the loop completes without managing to send the byte, the controller needs to be reset. We store that information in the

```c
static volatile int fdc\_need\_reset = 0;
```
variable and return.

We read the result in the following function:

```c
int fdc_getresults () {
    int i, results = 0;
    if (fdc_need_reset) { printf ("exit\n"); return 0; }

    for (i = 0; i < 30000; i++) {
        byte status = inb_delay (IO_FLOPPY_STATUS) & FLOPPY_NEW_BYTE;
        if (status == FLOPPY_MASTER) return true; // results are complete
        if (status != FLOPPY_NEW_BYTE) continue;
        if (results == MAX_FLOPPY_RESULTS) break;
        fdc_results[results++] = inb_delay (IO_FLOPPY_COMMAND);
    }

    fdc_need_reset = true; printf ("FDC: reply error\n"); return false;
}
```

Defines:
- `fdc_getresults`, used in chunks 540d, 548b, and 551.
- `fdc_need_reset` 536d, `fdc_results` 537d, `FLOPPY_MASTER` 536c, `FLOPPY_NEW_BYTE` 537b, `inb_delay` 538b, `IO_FLOPPY_COMMAND` 535, `IO_FLOPPY_STATUS` 535, `MAX_FLOPPY_RESULTS` 537c, and `printf` 601a.

If the last byte has been read, the status changes to `FLOPPY_MASTER` 536c and our function can return. We check whether the status is `FLOPPY_NEW_BYTE` 537b (i.e., the bits 4, 6 and 7 are set which indicates that the controller is busy, we’re reading from the controller and it is ready to have us query it). If this is not yet the case, we repeat until the status changes to `FLOPPY_NEW_BYTE` 537b. Then we can read the new byte via the `IO_FLOPPY_COMMAND` 535 port.

If we exceed the maximum number of bytes that we expect the controller to send, we cancel the operation and set the `fdc_need_reset` 536d flag.

### Constants

- `FLOPPY_CONTROLLER_BUSY` 0b00010000 // bit 4 of status reg., busy
- `FLOPPY_NEW_BYTE` (FLOPPY_MASTER | FLOPPY_DIRECTION | FLOPPY_CONTROLLER_BUSY)

Defines:
- `FLOPPY_NEW_BYTE`, used in chunk 537a.
- `busy`, `FLOPPY_DIRECTION` 536c, and `FLOPPY_MASTER` 536c.

When we read the results, we store them in the `fdc_results` 537d buffer:

```c
#define MAX_FLOPPY_RESULTS 0x07
```

Defines:
- `MAX_FLOPPY_RESULTS`, used in chunk 537.

### Global Variables

```c
byte fdc_results[MAX_FLOPPY_RESULTS];
```

Defines:
- `fdc_results`, used in chunks 537a, 540d, 548b, and 551b.
- `MAX_FLOPPY_RESULTS` 537c.
The functions

```c
void outb_delay (word __port, byte __value);
byte inb_delay (word __port);
```
do the same as `outportb()` and `inportb()`, but they execute an extra `outb` to an unused port (0xE0) in order to create a short delay. It does not matter which value is sent to the port, so they just send al (but could use any other value):

```c
#define defines

/* **** FROM proc/i386.h *********/

int fdc_command (int cmd, int drive, int track, int sector);
```

We have to pick the address manually because there are limitations on which memory areas can be used for DMA transfers.

The central function of the floppy driver which also handles the DMA setup is:

```c
static char *fdc_buf = (char *)0x9a800;
```
It takes four parameters: cmd is set to either FLOPPY_READ or FLOPPY_WRITE to indicate the transfer direction, and the last three parameters describe the sector in terms of the physical layout of a floppy disk.

```c
#define FLOPPY_READ 0xe6
#define FLOPPY_WRITE 0xc5
```

Defines:
- FLOPPY_READ, used in chunks 543a and 549c.
- FLOPPY_WRITE, used in chunks 540d and 550b.

`fdc_command` first sets the three variables

```c
static int fdc_drive, fdc_track, fdc_head;
```

Defines:
- fdc_drive, used in chunks 539c, 540c, 544, 547d, 548b, and 551b.
- fdc_head, used in chunks 539, 540, and 548b.
- fdc_track, used in chunks 539, 540, and 548b.

...to the corresponding argument values; they will also be accessed by other functions of the floppy driver, so using global variables we can avoid passing these around as parameters.

Then the function resets the controller (if needed), starts the motor, lets the drive seek to the right track (we have to do that manually) and initiates the DMA transfer (see the `fdc_transfer` chunk):

```c
int fdc_command (int cmd, int drive, int track, int sector) {
    fdc_drive = drive;
    fdc_track = track;
    fdc_head = sector / current_fdd_type->sectors;
    int fdc_sector = sector % current_fdd_type->sectors + 1;

    fdc_ticks_till_motor_stops = 3 * HZ;

    begin critical section in kernel
    // will be re-enabled in fdc_read_/write_sector
    for (int err = 0; err < MAX_FLOPPY_ERRORS; err++) {
        if (fdc_need_reset) fdc_reset();
        (fdc_start_motor 544a)
        if (!fdc_seek ()) continue;
        (fdc_transfer 540c)
        switch (transfer_status) {
            case -1: printf ("FDC: disk in drive %d is write protected\n", fdc_drive);
                return 0;
            case 0: continue;
            case 1: return 1;
        }
    }
    return 0;
}
```
Defines:
- fdc_command, used in chunks 538d, 549c, and 550b.

Uses current_fdd_type 541c, fdc_drive 539b, fdc_head 539b, fdc_need_reset 536d, fdc_reset 551a, fdc_seek, fdc_ticks_till_motor_stops 546c, fdc_track 539b, HZ 540a, MAX_FLOPPY_ERRORS 540b, printf 601a, and write 429b.

\[\text{[540a]}\]
\[\text{(constants 112a)}\] \(\equiv\) \(\text{(44a) } <539a \ 540b>\)
\#define HZ 100 \ // frequency of the timer

Defines:
- HZ, used in chunks 539c and 547a.

We allow up to eight floppy errors before we fail:

\[\text{[540b]}\]
\[\text{(constants 112a)}\] \(\equiv\) \(\text{(44a) } <540a \ 541a>\)
\#define MAX_FLOPPY_ERRORS \(0x08\)

Defines:
- MAX_FLOPPY_ERRORS, used in chunk 539c.

With all the information available we can initiate the transfer which means sending a longer sequence of bytes to the command/data register. We first tell the controller that we want to transfer in DMA mode (and not in PIO mode) which requires another command sequence shown in \(\text{fdc dma init 543a}\).

\[\text{[540c]}\]
\[\text{(fdc transfer 540c)}\] \(\equiv\) \(\text{(539c) } 540d\rangle\)

\begin{verbatim}
int transfer_status = 0;  // will be set to 1 when successful
int sectors;  // number of transmitted sectors
\(\text{begin critical section in kernel 380a})\)

if (!fdc_need_reset && current_fdd->motor && current_fdd->calibrated) {
    \text{[fdc dma init 543a]}
    fdc_mode (cmd);
    fdc_out (cmd);  fdc_out (fdc_head << 2 | fdc_drive);
    fdc_out (fdc_track);  fdc_out (fdc_head);  fdc_out (fdc_sector);
    fdc_out (current_fdd_type->sectorsize);  // 2: 512 bytes/sector
    fdc_out (current_fdd_type->sectors);  // end of track
    fdc_out (current_fdd_type->gap);  // gap length
    fdc_out (FLOPPY_DTL);  // data length
\end{verbatim}

Uses current_fdd 541c, current_fdd_type 541c, fdc_drive 539b, fdc_head 539b, fdc_mode 542c, fdc_need_reset 536d, fdc_out 536b, fdc_track 539b, and FLOPPY_DTL 541a.

We need not terminate the sequence because the controller knows when it has received a complete command. So we can immediately continue by waiting for the answer (via wait_fdc_interrupt 547d) and call fdc_getresults 537a to check whether the transfer was successful:

\[\text{[540d]}\]
\[\text{(fdc transfer 540c)}\] \(\equiv\) \(\text{(539c) } <540c}\)

\begin{verbatim}
if (!fdc_need_reset && !wait_fdc_interrupt () && fdc_getresults ()) {
    if (cmd == FLOPPY_WRITE && fdc_results[1] & WRITE_PROTECTED) {
        fdc_out (FLOPPY_SENSE);
        fdc_getresults ();
        transfer_status = -1;
    } else if ((fdc_results[0] & TEST_BITS) != TRANSFER_OK ||
\end{verbatim}
```c
fdc_results[1] || fdc_results[2]) {
    current_fdd->calibrated = 0;
    transfer_status = 0;
} else {
    sectors = (fdc_results[3] - fdc_track) * current_fdd_type->sectors * 2
      + (fdc_results[4] - fdc_head) * current_fdd_type->sectors
      + fdc_results[5] - fdc_sector;
    if (sectors == 1) transfer_status = 1;  // success
}
}
}

Uses current_fdd; 541c, current_fdd_type, 541c, fdc_getresults, 537a, fdc_head, 539b, fdc_need_reset, 536d,
fdc_out, 536b, fdc_results, 537d, fdc_track, 539b, FLOPPY_WRITE, 539a, TEST_BITS, 548c,
TRANSFER_OK, 541a, wait_fdc_interrupt, 547d, and WRITE_PROTECTED, 541a.

⟨constants 112a⟩ +=
  (44a) 〈540b 542a〉  [541a]
#define FLOPPY_DTL 0xFF
#define TRANSFER_OK 0x00
#define WRITE_PROTECTED 0x02

Defines:
  FLOPPY_DTL, used in chunk 540c.
  TRANSFER_OK, used in chunk 540d.
  WRITE_PROTECTED, used in chunk 540d.

Both when sending the request and when checking whether the sector was successfully read we need the device information which declares the physical properties of the disk drive: In recent years only 3.5" drives with a formatted capacity of 1440 KByte have been built into PCs (if at all), but older machines used 5.25" drives with a 1200 KByte capacity. We store the information that we need to tell the controller in fdd_type; 541c:

⟨type definitions 91⟩ +=
  (44a) 〈515a 541b〉  [541b]
typedef struct {
    int total_sectors, tracks, sectors, sectorsize, trackstep, rate, gap, spec1;
} struct_fdd_type;

typedef struct {
    int present, calibrated, motor, current_track, type;
} struct_fdd;

Defines:
  struct_fdd, used in chunk 541c.
  struct_fdd_type, used in chunk 541c.

⟨global variables 92b⟩ +=
  (44a) 〈539b 544d〉  [541c]
char *fdd_drive_name[6] = {
  "not installed", "360 KByte (not supported)",
  "1200 KByte",  "720 KByte (not supported)",
  "1440 KByte",  "2880 KByte (not supported)"};

struct_fdd_type fdd_type[2] = {
  { 80*15*2, 80, 15, 2, 0, 0, 0x1B, 0x0F },  /* 1.2M */
  { 80*18*2, 80, 18, 2, 0, 0, 0x1B, 0xCF }  /* 1.44M */
};
```
struct_fdd_type *current_fdd_type;
struct_fdd fdd[2] = {{ 0, 0, 0, INVALID_TRACK, 0 }, { 0, 0, 0, INVALID_TRACK, 0 }};
int fdds_in_use[2] = { 0, 0 };
struct_fdd *current_fdd;

Defines:
current_fdd, used in chunks 540, 544a, 548b, and 551b.
current_fdd_type, used in chunks 539, 540, 542c, 543a, 548b, and 549d.
fdd, used in chunks 499a, 544, 547a, 549d, 551a, and 552c.
fdd_drive_name, used in chunk 552c.
fdd_type, used in chunks 499a and 549d.

Uses INVALID_TRACK 542a, struct_fdd 541b, and struct_fdd_type 541b.

[542a] ⟨constants 112a⟩+≡ (44a) <541a 542d>
#define INVALID_TRACK -1

Defines:
INVALID_TRACK, used in chunks 541c and 551b.

When we initialize the system, we will detect the available floppies and write the information into fdd used in Section 13.6.7.

We have been using an fdc_mode function that tells the controller what kind of drive it has to access, but we have not shown its implementation yet. It uses fdc_out but also writes to the configuration control register via the IO_FLOPPY_RATE port:

[542b] ⟨function prototypes 45a⟩+≡ (44a) <538d 545a>
void fdc_mode ();

[542c] ⟨function implementations 100b⟩+≡ (44a) <539c 545b>
void fdc_mode () {
    fdc_out (FLOPPY_SPECIFY);
    fdc_out (current_fdd_type->spec1);
    fdc_out (FLOPPY_SPEC2);
    outb_delay (IO_FLOPPY_RATE, current_fdd_type->rate & ~0x40);
}

Defines:
fdc_mode, used in chunks 540c and 542b.

DMA controller For setting up the DMA transfer we need to talk to the DMA controller which uses its own ports for configuring:

[542d] ⟨constants 112a⟩+≡ (44a) <542a 542e>
#define FLOPPY_SPECIFY 0x03
#define FLOPPY_SPEC2 0x06

Defines:
FLOPPY_SPEC2, used in chunk 542c.
FLOPPY_SPECIFY, used in chunk 542c.

[542e] ⟨constants 112a⟩+≡ (44a) <542d 543b>
#define IO_DMA0_INIT 0x0A // single channel mask register
#define IO_DMA0_MODE 0x0B // mode register
#define IO_DMA0_FLIPFLOP 0x0C // flip-flop reset register

DMA controller
The important bit about the following code chunk is that we tell the DMA controller which chunk of memory it shall use as buffer for the DMA data transfer. The controller only accepts 24 bit wide physical addresses. We tell it our buffer address \texttt{fdc_buf} by sending the lowest 16 bits to \texttt{IO_DMA_ADDR_2} and the highest eight bits to \texttt{IO_DMA_PAGE_2}. That stores the lower 16 bits in the controller’s \textit{address register} and the eight extra bits in the \textit{page register}. The reason for this separate treatment is compatibility: Older DMA controllers only supported 16-bit addresses. The amount of bytes to read or write must be written to the \textit{count register} via \texttt{IO_DMA_COUNT_2}. It takes a 16-bit value which requires two \texttt{outb} commands.

\begin{verbatim}
⟨fdc dma init 543a⟩≡
int count = 1 << (current_fdd_type->sectorsize + 7);  // = 512
int mode;
if (cmd == FLOPPY_READ)
  mode = DMA_READ_MODE;
else
  mode = DMA_WRITE_MODE;
// prepare read operation
outb_delay (IO_DMA0_INIT, FLOPPY_CHANNEL | 4);  // disable DMA channel
outb_delay (IO_DMA0_FLIPFLOP, 0);  // clear DMA ch. flipflop
outb_delay (IO_DMA0_MODE, mode | FLOPPY_CHANNEL);  // set DMA ch. mode (r/w)
// set count, address and page registers
outb_delay (IO_DMA0_COUNT, (byte)(count-1));  // count
outb_delay (IO_DMA0_COUNT, (byte)((count-1) >> 8));
outb_delay (IO_DMA0_ADDR, (byte)(unsigned)fdc_buf);  // address, bits 0..7
outb_delay (IO_DMA0_ADDR, (byte)((unsigned)fdc_buf >> 8));  // bits 8..15
outb_delay (IO_DMA0_PAGE, (unsigned)fdc_buf >> 16);  // page, bits 16..23
outb_delay (IO_DMA0_INIT, FLOPPY_CHANNEL);  // enable DMA channel
\end{verbatim}

Uses current_fdd_type 541c, DMA_READ_MODE 542e, DMA_WRITE_MODE 542e, \texttt{fdc_buf} 538c, FLOPPY_CHANNEL 543b, FLOPPY_READ 539a, IO_DMA0_FLIPFLOP 542e, IO_DMA0_INIT 542e, IO_DMA0_MODE 542e, IO_DMA_ADDR_2 542e, IO_DMA_COUNT_2 542e, IO_DMA_PAGE_2 542e, outb_delay 538b, read 429b, and write 429b.
13.6.3 Starting and Stopping the Motor

In comparison to the hard disk controller, the floppy controller needs a lot of help to get things right. For example, it is necessary to manually turn the floppy motor on and off. The code chunks \(\langle\text{fdc start motor 544a}\rangle\) and \(\langle\text{fdc stop motor 544b}\rangle\) send the right commands:

\[
\begin{align*}
\text{if (!current_fdd->motor)} \{ \\
&\quad \text{outb_delay (IO_FLOPPY_OUTPUT, FLOPPY_CONTROLLER_ENABLE | FLOPPY_DMAINT_ENABLE | fdc_drive | (16 \ll fdc_drive))}; \\
&\quad \text{current_fdd->motor = 1;} \\
&\quad \text{fdd[!fd} \text{c_drive].motor = 0;} \\
&\quad \text{for (int i = 0; i < 500000; i++)} \{ // delay } \\
\}
\]

Uses current_fdd 541c, fdc_drive 539b, fdd 541c, FLOPPY_CONTROLLER_ENABLE 544c, FLOPPY_DMAINT_ENABLE 544c, IO_FLOPPY_OUTPUT 535, and outb_delay 538b.

\[
\begin{align*}
\text{outb_delay (IO_FLOPPY_OUTPUT,} \\
&\quad \text{FLOPPY_CONTROLLER_ENABLE | FLOPPY_DMAINT_ENABLE | fdc_drive);} \\
&\quad \text{fdd[0].motor = fdd[1].motor = 0;} \\
\]

Uses fdc_drive 539b, fdd 541c, FLOPPY_CONTROLLER_ENABLE 544c, FLOPPY_DMAINT_ENABLE 544c, IO_FLOPPY_OUTPUT 535, and outb_delay 538b.

\[
\begin{align*}
\#define \text{FLOPPY_CONTROLLER_ENABLE} \ 0x04 \\
\#define \text{FLOPPY_DMAINT_ENABLE} \ 0x08
\]

Defines: 
FLOPPY_CONTROLLER_ENABLE, used in chunks 544 and 551a.
FLOPPY_DMAINT_ENABLE, used in chunks 544 and 551a.

13.6.4 Handling Floppy Interrupts

We define a floppy_queue which will contain processes waiting for a floppy access operation to finish:

\[
\begin{align*}
\text{blocked_queue floppy_queue; // processes which wait for the floppy}
\end{align*}
\]

Defines: 
floppy_queue, used in chunks 544–46, 564c, and 606.
Uses blocked_queue 183a.

and we have to initialize this queue:

\[
\begin{align*}
\text{initialize_blocked_queue (&floppy_queue);} \\
\end{align*}
\]

Uses floppy_queue 544d and initialize_blocked_queue 183c.
Processes that access a floppy disk drive will wait on this queue, and the interrupt handler will wake up these processes when the operation was completed.

For example, when reading from an open file, we will call `fdc_read_sector` which in turn calls `fdc_command`.

The last function potentially calls `fdc_reset`, and it will call further functions all of which call `wait_fdc_interrupt`.

`wait_fdc_interrupt` actually puts the process to sleep via `fdc_sleep` until an interrupt occurs.

Here is the implementation of the

```c
void fdc_sleep ()
{
    if ((current_task > 1) && scheduler_is_active) {
        // block process
        fdc_is_busy = true;
        // access thread table
        block (&floppy_queue, TSTATE_WAITFLP);
        // resign
    }
    fdc_is_busy = false;
};
```

**Defines:**
- `fdc_sleep`, used in chunks 545a and 547d.
- `current_task`, `fdc_is_busy`, `floppy_queue`, `scheduler_is_active`, and `TSTATE_WAITFLP`.

It uses the global

```c
short int fdc_is_busy = false;
```

**Defines:**
- `fdc_is_busy`, used in chunk 545b.

When we need to wake up a process that has been waiting for the floppy drive, we use the

```c
void fdc_wakeup ()
```

**Function:** At any given time there can only be one active floppy operation, because (as you will see in the implementation of `fdc_read_sector` and `fdc_write_sector`), all read and write operations are critical sections, protected by the lock `fdc_lock`. So for `fdc_wakeup` we just wake the first process in the queue as there can only be one.
void fdc_wakeup () {
    thread_id tid = floppy_queue.next;
    if (tid != 0) deblock (tid, &floppy_queue);
}

void floppy_handler (context_t *r) {
    fdc_timeout = false;
    if (!fdc_waits_interrupt)
        fdc_need_reset = 1;  // unexpected floppy interrupt, reset controller
    fdc_waits_interrupt = false;
    fdc_wakeup ();
}

int fdc_ticks = 0;
int fdc_ticks_till_motor_stops = 0;
boolean fdc_timeout = false;
boolean fdc_waits_interrupt = false;

void fdc_timer (){
}

We use two counters fdc_ticks and fdc_ticks_till_motor_stops: The first one starts counting when we have started a read or write operation. If it does not finish within two seconds (200 ticks) we abort the current operation (and fail). The second counter makes sure that the motor is stopped three seconds after the last operation completed. We don’t turn the motor off immediately because further requests might follow. The flags fdc_timeout and fdc_waits_interrupt show whether our two seconds have run out and whether we’re waiting for an interrupt.

The fdc_timer function will be called from the timer handler, so we need to declare it here:
The floppy timer has two objectives: It checks whether a timeout has occurred (and cancels the current operation) and it checks if the motor has been running too long:

\[
\begin{align*}
\text{void fdc_timer ()} & \{ \\
& \quad \text{if (fdc_waits_interrupt && ++fdc_ticks > HZ * 2) } \\
& \quad \text{fdc_waits_interrupt = false;} \\
& \quad \text{fdc_timeout = true;} \\
& \quad \text{fdc_wakeup ()}; \\
& \} \text{ else if ((fdd[0].motor | fdd[1].motor) &&} \\
& \quad \text{!(fdc_lock->l) && !--fdc_ticks_till_motor_stops) } \\
& \quad \{ \\
& \quad \text{} \texttt{fcn stop motor 54ab} \\
& \quad \}
\end{align*}
\]

Defines:
\text{fdc_timer}, used in chunk 546.

Uses \text{fdc_lock 547b}, \text{fdc_ticks 546c}, \text{fdc_ticks_till_motor_stops 546c}, \text{fdc_timeout 546c},
\text{fdc_waits_interrupt 546c, fdc_wakeup 546a, fdd 541c, and HZ 540a.}

It only stops the motor if the lock \text{fdc_lock 547b} is not held. We have not declared it yet, but already mentioned it twice:

\[
\begin{align*}
\text{int wait_fdc_interrupt ()} & \{ \\
& \quad \text{fdc_ticks = 0;} \quad \text{\texttt{// reset the wait time}} \\
& \quad \text{fdc_waits_interrupt = true;} \quad \text{\texttt{// yes, we wait}} \\
& \quad \text{fdc_sleep ()}; \\
& \quad \text{if (fdc_timeout) } \\
& \quad \text{\{ } \\
& \quad \text{\quad fdc_need_reset = 1;} \\
& \quad \text{\quad printf ("FDC: drive %d timeout\n", fdc_drive);} \\
& \quad \text{\}} \\
& \quad \text{return fdc_timeout;} \\
& \}
\end{align*}
\]

Defines:
\text{wait_fdc_interrupt}, used in chunks 540d, 547c, 548b, and 551.

Uses \text{fdc_drive 539b, fdc_need_reset 536d, fdc_sleep 545b, fdc_ticks 546c, fdc_timeout 546c},
\text{fdc_waits_interrupt 546c, and printf 601a.}
Reading and Writing

Reading and writing require that we first move the drive head to the right location. This is performed by the

\[\text{function prototypes}\]  
\[
\text{int fdc\_seek();}
\]

function which calculates the physical parameters and sends them to the controller, using the FLOPPY\_SEEK function.

\[\text{function implementations}\]  
\[
\begin{array}{l}
\text{int fdc\_seek()}
\end{array}
\]

\[
\begin{array}{l}
\text{if (fdc\_need\_reset} ||
\text{(!current\_fdd->calibrated && !fdc\_recalibrate())) return false;}
\end{array}
\]

Via fdc_getresults we check whether the seek operation was successful: In that case the highest five bits of fdc_results[0] will be 00100b.

\[\text{constants}\]  
\[
\begin{array}{l}
\text{#define TEST\_BITS 0b11111000 // 0xf8}
\end{array}
\]

\[
\begin{array}{l}
\text{#define SEEK\_OK 0b00010000 // 0x20}
\end{array}
\]

The commands FLOPPY\_SEEK and FLOPPY\_SENSE perform the seek operation and request status information from the floppy drive which is required after every command.
13.6 The Floppy Controller

\(\text{constants 112a} \equiv \) (44a) \(\langle 548c \ 550a \rangle \ [549a]\)

\#define FLOPPYSEEK 0x0f
\#define FLOPPYSENSE 0x08

Defines:

FLOPPYSEEK, used in chunk 548b.
FLOPPYSENSE, used in chunks 540d, 548b, and 551.

With seeking completed, we’re ready to read or write.

\(\text{function prototypes 45a} \equiv \) (44a) \(\langle 548a \ 550c \rangle \ [549b]\)

int fdc_read_sector (int device, int block, char *buffer);
int fdc_write_sector (int device, int block, char *buffer);

These functions read and write 512 byte sized sectors:

\(\text{function implementations 100b} \equiv \) (44a) \(\langle 548b \ 550b \rangle \ [549c]\)

int fdc_read_sector (int device, int block, char *buffer) {
    \(\text{fdc: prepare read/write sector 549d}\)
    result = fdc_command (FLOPPYREAD, device, ctrack, csector); // will sleep
    if (result) {
        memcpy ((void *)buffer, PHYSICAL(fdc_buf), FD_SECSIZE);
    }
    \(\text{fdc: finish read/write sector 549e}\)
}

Defines:

fdc_read_sector, used in chunk 550d.

Uses csector, ctrack, FD_SECSIZE 550a, fdc_buf 538c, fdc_command 539c, FLOPPYREAD 539a, memcpy 596c, and PHYSICAL 116a.

with

\(\text{fdc: prepare read/write sector 549d} \equiv \) (549c 550b) \ [549d]\)

int spt; // sectors per track
int ctrack, csector;
int result;

mutex_lock (fdc_lock);
    current_fdd = &fdd[device];
    current_fdd_type = &fdd_type[current_fdd->type];

    spt = current_fdd_type->sectors * 2; // 36 ??
    ctrack = block / spt;
    csector = block % spt;

Uses csector, ctrack, current_fdd 541c, current_fdd_type 541c, fdc_lock 547b, fdd 541c, fdd_type 541c,
mutex_lock 366a, and spt.

and

\(\text{fdc: finish read/write sector 549e} \equiv \) (549c 550b) \ [549e]\)

mutex_unlock (fdc_lock);
if (result) return FD_SECSIZE;
else return 0;

Uses FD_SECSIZE 550a, fdc_lock 547b, and mutex_unlock 366c.
defines FD_SECSIZE, used in chunks 549 and 550.

Writing is similar, but the order of calling fdc_command and copying the buffer contents is reversed; also fdc_command supplies the argument FLOPPY_WRITE instead of FLOPPY_READ, and the memcpy operation works the other way round:

```c
int fdc_write_sector (int device, int block, char *buffer) {
    memcpy (PHYSICAL (fdc_buf), (void *)buffer, FD_SECSIZE);
    result = fdc_command (FLOPPY_WRITE, device, ctrack, csector); // will sleep
    return result;
}
```

Defines: fdc_write_sector, used in chunks 549b and 550d.

Uses csector, ctrack, FD_SECSIZE, fdc_buf, FLOPPY_WRITE and memcpy.

Since we will always read or write whole blocks (1 KByte) we add readblock_fd and writeblock_fd functions:

```c
void readblock_fd (int device, int blockno, char *buffer);
void writeblock_fd (int device, int blockno, char *buffer);
```

which simply call the sector functions twice:

```c
void readblock_fd (int device, int blockno, char *buffer) {
    fdc_read_sector (device, blockno*2, buffer);
    fdc_read_sector (device, blockno*2 + 1, buffer + FD_SECSIZE);
}

void writeblock_fd (int device, int blockno, char *buffer) {
    fdc_write_sector (device, blockno*2, buffer);
    fdc_write_sector (device, blockno*2 + 1, buffer + FD_SECSIZE);
}
```

Defines: readblock_fd, used in chunk 506b.

writeblock_fd, used in chunks 507b and 550c.

Uses FD_SECSIZE, fdc_read_sector, and fdc_write_sector.

### 13.6.6 Resetting and Recalibrating

Two further functions:

```c
void fdc_reset ();
int fdc_recalibrate ();
```
are required for our floppy driver implementation. \texttt{fdc_reset} is called when too many errors have occurred; in that case it asks the controller to reset so that a new attempt can be started.

```c
#define fdc_reset (void) fdc_reset()

int fdc_recalibrate () {
    if (fdc_need_reset) return 0;
    fdc_out (FLOPPY_RECALIBRATE);
    fdc_out (fdc_drive);
    if (fdc_need_reset || wait_fdc_interrupt ()) return 0;
    fdc_out (FLOPPY_SENSE);
    if (!fdc_getresults()) return 0;
    if (fdc_results[0] & TEST_BITS) !:-SEEK_OK || fdc_results[1])
        goto bad_recalibration;

    current_fdd->current_track = INVALID_TRACK;
    return current_fdd->calibrated = 1;

    bad_recalibration:
    printf("FDC: can't recalibrate\n");
    fdc_need_reset = 1;
    return 0;
}
```
13 Disk I/O

Defines:
fdc_recalibrate, used in chunk 548b.

Uses current_fdd 541c, fdc_drive 539b, fdc_getresults 537a, fdc_need_reset 536d, fdc_out 536b,
fdc_results 537d, FLOPPY_RECALIBRATE 552a, FLOPPY_SENSE 549a, INVALID_TRACK 542a, printf 601a,
SEEK_OK 548c, TEST_BITS 548c, and wait_fdc_interrupt 547d.

\[\text{constants 112a}\] \(\equiv\) (44a) <550a 579b>  
#define FLOPPY_RECALIBRATE 0x07

Defines:  
FLOPPY_RECALIBRATE, used in chunk 551b.

13.6.7 Floppy Driver Initialization

As with the hard disk driver, we also need to initialize the floppy driver when the system boots. This happens in the

\[\text{function prototypes 45a}\] \(\equiv\) (44a) <550e 553a>  
void fdc_init();

function which initializes the locks, gathers the information about available floppy drives from the CMOS, enters them in the data structures (and displays them on the screen) and installs the interrupt handler for the floppy interrupt. It is comparable to the ata_init function.

\[\text{function implementations 100b}\] \(\equiv\) (44a) <551b 553b>  
void fdc_init() {  
fdc_lock = get_new_lock("fdc");  // initialize lock

outb_delay(IO_CMOS_CMD, 0x10);  // read floppy status from CMOS
int fdd_type_byte = inb_delay(IO_CMOS_DATA);

int type; printf("FDC: ");  // enter and display data
for (int i = 0; i < 2; i++) {
  // check floppy drive i
  if (i == 0) type = fdd_type_byte >> 4;  // upper 4 bits
  else type = fdd_type_byte & 0x0F;  // lower 4 bits
  if ((fdd[i].present = (type == 2 || type == 4 || type == 5)))
    fdd[i].type = fdc_map_type(type);
  printf("fd%d (%s)%s", i, fdd_drive_name[type], (i==0) ? ", " : 

if (fdd[0].present || fdd[1].present) {  // enable floppy handler
  install_interrupt_handler (IRQ_FDC, floppy_handler);
  enable_interrupt (IRQ_FDC);
  outportb (IO_FLOPPY_RATE, 0);  // FDC Reset
  outportb (IO_FLOPPY_OUTPUT, 12);  // enable DMA, disable Reset
} }
13.6 The Floppy Controller

Defines:
- fdc_init, used in chunks 45c and 552b.

Uses enable_interrupt 140b, fdc_lock 547b, fdc_map_type 553b, fdd 541c, fdd_drive_name 541c,
- floppy_handler 546b, get_new_lock 367b, inb_delay 538b, install_interrupt_handler 146c, IO_CMOS_CMD 339b,
- IO_CMOS_DATA 339b, IO_FLOPPY_OUTPUT 535, IO_FLOPPY_RATE 535, IRQ_FDC 132, lock 365a, outb_delay 538b,
- outportb 133b, printf 601a, and read 429b.

It uses the helper function

\[
\begin{align*}
\text{int fdc_map_type (int t);}
\end{align*}
\]

that converts the floppy drive type (as seen in the CMOS) into an index into the fdd_type541c[]
table which contains description of the drive characteristics.

\[
\begin{align*}
\text{int fdc_map_type (int t) }
\{ \\
\quad \text{switch (t) }
\{ \\
\quad \quad \text{case 2: return 0; } \quad \text{// 1.2 MByte drive} \\
\quad \quad \text{case 4: return 1; } \quad \text{// 1.44 MByte drive} \\
\quad \quad \text{default: return -1; }
\}
\}
\end{align*}
\]

Defines:
- fdc_map_type, used in chunks 552c and 553a.

Credits

As a final note we want to give credit to Tudor Hulubei who wrote the Thix Operating
System [Hul95] and published the source code under the GPL 2 license. The whole floppy
code in Section 13.6 is based on his floppy driver implementation, though we have removed
a lot of the original code. For example, Thix uses several disk buffers so that floppy requests
can be queued.
Signals are a classical Unix mechanism which allows a simple kind of messaging: processes can send signals to other processes which makes them either terminate or call a registered signal handler. In many ways these signals are very similar to interrupts, but while an interrupt handler can only be set up in kernel mode (and serves the whole operating system), signal handlers can be set up in user mode and belong to just one process.

The similarity between signals and interrupts goes even further: Interrupts exist in two varieties, synchronous (e.g. interrupts caused by accessing a bad memory address or trying to execute an unknown CPU instruction) and asynchronous (e.g. raised by a device, such as the timer or a floppy or hard disk controller), and the same holds for signals: a synchronous signal is caused by the process itself (again access to a bad memory address is an example—in that case it causes an interrupt first and the interrupt handler sends a corresponding signal to the process), but most signals are asynchronous (sent by a different process).

Some functionality is available in both worlds (interrupts and signals), take for example a timer: the computer’s timer chip generates regular timer interrupts which are asynchronous events and invoke the kernel’s timer interrupt handler. Besides other things, this handler checks whether one of the processes has registered a (process-private) timer—and if so, generates an alarm signal. When the process is scheduled the next time, instead of continuing its normal execution it enters its alarm signal handler and treats the asynchronous signal.

An example for synchronous events in both worlds is bad memory access. When a process tries to access a virtual address which is not available (because the page tables do not map it to some physical address), a page fault is generated, so the CPU jumps into the page fault (interrupt) handler. That one checks the reason (the only acceptable reason being that the page was paged out to disk). If that is not the case, the process must be terminated. To achieve this goal, the page fault handler sends the process a SIGSEGV (segmentation fault signal)
violation) signal. When the process is scheduled again, it will normally abort, though it may have registered a SIGSEGV signal handler to deal with such a situation.

For the memory example, consider the following program `segfault.c`:

```
int main () {
    char *adr = (char *)0;
    char c = *adr;
    putchar (c);
}
```

and its execution via the debugger `gdb`:

```
$ gcc -g segfault.c
$ gdb a.out
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
(gdb) run
Starting program: /tmp/a.out
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400508 in main () at segfault.c:3
 3 char c = *adr;
```

### 14.1 Use Cases for Signals

What are signals good for? In this section we show you three examples which demonstrate the range of application of signals.

**Program Error:** In many cases programs may run into a problem when they try to perform an action that the CPU will not allow, for example when trying to access an invalid memory address. This is normally an indication that the program is faulty, and the best action will be to terminate the process. However, instead of checking every memory address in the program before it is accessed, a developer may decide to rely on an error handler that will be called if such behavior is detected. So a solution could be to write an error handler (and install it as the signal handler for the signal that will be generated) that resets the program to some initial state and starts over. Then, when an error occurs, the processor jumps into the fault handler which will send a signal to the process. When the process continues, it will execute the signal handler.

**Inter Process Communication:** Processes that work together somehow, often need to communicate. Unix systems provide special mechanisms for sending complex messages, but if only some kind of “ping” is required to tell another process that a certain condition has been reached, then a signal can serve that purpose.

**Voluntary Abort:** A process may decide to terminate itself when it recognizes an error condition. Instead of calling `exit` with an error return value, it can use `abort` to send itself a `SIGABRT` signal.
14.2 Signals in Classical Unix Systems

In classical Unix systems, processes can use the `kill` system call to deliver a signal to an arbitrary process (as long as both have the same owner or the signal-sending process belongs to `root`) or the `raise` system call to send a signal to themselves.

Every Unix system knows a few standard signals, with signal numbers typically ranging from 0 to 31. While some signals have standard signal numbers (such as 9 and 15 for `SIGKILL` and `SIGTERM`), the POSIX standard does not require signals to use standard values; it only asks for signal names to be defined:

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Default Action</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>(default action: abort)</td>
<td>Process abort signal. The signal could be sent by the process itself (see above), by a different process or by the kernel.</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>(default action: abort)</td>
<td>Unix systems normally supply an alarm clock mechanism. By defining a timer, this signal will be sent when the requested timeout occurs.</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>(default action: abort)</td>
<td>The process tried to access an invalid address. This is similar to <code>SIGSEGV</code>, but the latter deals with forbidden access to a valid address (an address that requires kernel privileges).</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>(default action: ignore)</td>
<td>When a child process terminates, stops (<code>SIGSTOP</code>) or continues (<code>SIGCONT</code>), its parent process is notified with this signal.</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>(default action: continue)</td>
<td>A process that was stopped (via <code>SIGSTOP</code>) continues execution.</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>(default action: abort)</td>
<td>Originally the acronym FPE stood for <em>Floating Point Exception</em> and the signal was raised when the process caused the FPU (Floating Point Unit) to perform a faulting calculation, such as a division by zero. Today <code>SIGFPU</code> is used for all kinds of arithmetic errors.</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>(default action: abort)</td>
<td>The Hangup signal tells a process that its controlling terminal is gone. On a modern Unix system this often refers to a closed terminal window, traditionally it occurred when the connection of a dumb terminal device to the machine (via a serial line or a dial-in connection) was lost.</td>
</tr>
<tr>
<td>SIGILL</td>
<td>(default action: abort)</td>
<td>The process tried to execute an illegal instruction (e.g., one that requires a newer processor with an extended instruction set).</td>
</tr>
</tbody>
</table>
| SIGINT      | (default action: abort) | Pressing `[Ctrl-C]` generates this signal. It is possible to write a signal handler which may decide to ignore `[Ctrl-C]`. Note that Unix terminates

---

¹ see http://pubs.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html
processes when this key combination is pressed.

**SIGKILL** (default action: abort)

This is the aggressive KILL signal that cannot be intercepted by a signal handler. It terminates the process at once.

**SIGPIPE** (default action: abort)

The process wrote to a pipe that has no reader.

**SIGQUIT** (default action: abort)

This is similar to **SIGINT**. Some systems write a core dump, in addition to what is caused by **SIGINT**.

**SIGSEGV** (default action: abort)

The process tried to access a memory location for which it lacks the required privileges, see **SIGBUS**.

**SIGSTOP** (default action: stop)

The signal stops a process. It will remain blocked until it receives **SIGCONT**. The signal cannot be intercepted by a handler.

**SIGTERM** (default action: abort)

The process is asked to terminate. It can install a signal handler for this signal which allows it to write in-memory data to files or perform other final actions. The signal can also be ignored.

**SIGTSTP** (default action: stop)

This is a variant of **SIGSTOP** and allows the installation of a signal handler.

**SIGTTIN** (default action: stop)

The process has no terminal but tried to read from a non-redirected standard input.

**SIGTTOU** (default action: stop)

The process has no terminal but tried to write to a non-redirected standard output or standard error output.

**SIGUSR1, SIGUSR2** (default: abort)

These signals can be used by application developers for their own purposes.

**SIGPOLL** (default action: ignore)

When new data appear on a process’ standard input, this signal is raised. Network sockets can also generate this signal.

**SIGPROF** (default action: abort)

A special timer that counts CPU time which was spent in this process (or in the kernel, but for this process) generates this signal. It is similar to **SIGALRM** but that one uses real time. Also compare with **SIGVTALRM**.

**SIGSYS** (default action: abort)

The process tried to execute a system call with an unknown system call number.

**SIGTRAP** (default action: abort)

This signal is raised when a process is run in a debugger and a breakpoint was reached.

**SIGURG** (default action: ignore)

For systems that support networking, this signal indicates that data
have arrived on a socket which require urgent treatment.

**SIGVTALRM** (default action: abort)

A special timer that counts CPU time which was spent in this process (but not the kernel) generates this signal. It is similar to **SIGALRM** but that one uses real time. Also compare with **SIGPROF**.

**SIGXCPU** (default action: abort)

This can be used if a process is only granted a certain amount of CPU time before it is terminated. The signal is sent a bit earlier so that the process can finish its work.

**SIGXFSZ** (default action: abort)

The process has tried to create a file that is larger than allowed.

Table 14.1 shows how these signal names are mapped to signal numbers on some standard systems.

The information in the table was gathered from the following sources:

- `kill -l` on Linux (kernel 3.0.0) and OS X (Darwin kernel 10.8.0),
- Minix `signal.h` header file; “n/a” (not available) means: these system calls have not been implemented in Minix, but the numbers were assigned because the POSIX standard requires Unix implementations to define them; [http://faculty.qu.edu.qa/rriley/cmpt507/minix/signal_8h-source.html](http://faculty.qu.edu.qa/rriley/cmpt507/minix/signal_8h-source.html).

Over these five operating systems, only the signals SIGHUP (1), SIGINT (2), SIGQUIT (3), SIGILL (4), SIGTRAP (5), SIGABRT (6), SIGFPE (8), SIGKILL (9), SIGSEGV (11), SIGPIPE (13), SIGALRM (14) and SIGTERM (15) have common numbers.


### 14.3 Implementation of Signals in ULIX

In order to implement signals the following two sets of functionalities are normally required:

- Methods which let a process register *signal handlers* (via a `signal` system call) and decide which signals to block (by setting the *signal mask* via a `sigprocmask` system call). (ULIX does not support changing the signal mask.)
- Methods to deliver signals to processes and have the process react accordingly: for delivering, we need to implement the `kill` system call, and making the process execute (and return from) the signal handler requires changes to the scheduling code.

We will allow each process to define signal handlers for 32 signals (0–31), so we need space for 32 addresses, and we need to store $2 \times 32$ bits in each process for pending signals and blocked signals: A signal handler is stored as its address:
<table>
<thead>
<tr>
<th>Signal</th>
<th>Linux</th>
<th>OS X</th>
<th>Minix</th>
<th>FreeBSD</th>
<th>Solaris</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>7</td>
<td>10</td>
<td>7</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>17</td>
<td>20</td>
<td>17</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>18</td>
<td>19</td>
<td>n/a, 18</td>
<td>19</td>
<td>25</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>19</td>
<td>17</td>
<td>n/a, 19</td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>20</td>
<td>18</td>
<td>n/a, 20</td>
<td>18</td>
<td>24</td>
</tr>
<tr>
<td>SIGTIN</td>
<td>21</td>
<td>21</td>
<td>n/a, 22</td>
<td>21</td>
<td>26</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>22</td>
<td>22</td>
<td>n/a, 23</td>
<td>22</td>
<td>27</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>10</td>
<td>30</td>
<td>10</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>12</td>
<td>31</td>
<td>12</td>
<td>31</td>
<td>17</td>
</tr>
<tr>
<td>SIGPOLL *)</td>
<td>29</td>
<td>23</td>
<td>–</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>SIGPROF</td>
<td>27</td>
<td>27</td>
<td>25</td>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>31</td>
<td>12</td>
<td>–</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>SIGURG</td>
<td>23</td>
<td>16</td>
<td>–</td>
<td>16</td>
<td>21</td>
</tr>
<tr>
<td>SIGVTALRM</td>
<td>26</td>
<td>26</td>
<td>24</td>
<td>26</td>
<td>28</td>
</tr>
<tr>
<td>SIGXCPU</td>
<td>24</td>
<td>24</td>
<td>–</td>
<td>24</td>
<td>30</td>
</tr>
<tr>
<td>SIGXFSZ</td>
<td>25</td>
<td>25</td>
<td>–</td>
<td>25</td>
<td>31</td>
</tr>
</tbody>
</table>

Table 14.1: Standard signals on five Unix systems.

*) Linux, Mac OS and FreeBSD call the SIGPOLL signal SIGIO.

[560a] *(public elementary type definitions 45e) +≡

typedef void (*sighandler_t)(int);

Defines:
sighandler_t, used in chunks 560b, 561a, 566, and 568b.

and 32 bits fit precisely in an unsigned long integer, so we can add two variables sig_pending and sig_blocked for storing those bits:

[560b] *(more TCB entries 158c) +≡

sighandler_t sighandlers[32];
unsigned long sig_pending;
unsigned long sig_blocked;

Uses sighandler_t 560a.
For every signal number $i$ and each state of a process there are several possibilities:

- The signal is blocked; in that case the $i$-th bit in sig_blocked is 1, and the value in sighandlers[$i$] is irrelevant, because the signal will cause the $i$-th bit of sig_pending to be set.
- The signal is not blocked and the default action shall take place; in that case the corresponding bit in sig_blocked is 0 and sighandlers[$i$] is SIG_DFL.
- The signal is not blocked, but the process ignores this signal; in that case the corresponding bit in sig_blocked is 0 and sighandlers[$i$] is SIG_IGN.
- The signal is not blocked and a signal handler handler() has been installed; in that case the corresponding bit in sig_blocked is 0 and sighandlers[$i$] is handler.

We use the SIG_DFL and SIG_IGN values from a Linux system (on a 32 bit Linux system their definitions can be found in /usr/include/asm-generic/signal-defs.h):

```c
#define SIG_DFL ((sighandler_t)0) // default signal handling
#define SIG_IGN ((sighandler_t)1) // ignore signal
#define SIG_ERR ((sighandler_t)-1) // error code
```

Defines:
- SIG_DFL, used in chunks 562b and 567b.
- SIG_IGN, used in chunks 562b and 567b.
- SIG_ERR, used in chunk 566b.

Uses sighandler_t 560a and signal 568b.

This assumes that 0 and 1 can never be the addresses of a signal handler function.

We will not implement queues for signals; if the same process receives the same signal more than once before the scheduler activates it the next time, then the extra signal(s) will be lost (which is also what other Unix implementations do). Thus, our internal u_kill function is rather simple:

```c
int u_kill (int pid, int signo);
```

will set errno (more precisely: the TCB element error) to one of the following error constants if something goes wrong:

```c
#define EPERM 1 // not permitted
#define ESRCH 3 // no such process
#define EINVAL 22 // invalid argument
```

Defines:
- EINVAL, used in chunks 562b and 565c.
- EPERM, used in chunks 562b and 565c.
- ESRCH, used in chunks 562b and 565c.

Now we present the implementation. We have taken the signal numbers from an OS X system (and renamed SIGIO to SIGPOLL):
The u_kill function first tests a few error conditions and leaves immediately if it finds one of them: Signal numbers must be between 0 and 31, the target process must exist and must neither have PID 1 (init/idle) or 2 (swapper) nor login as command name.

Then it checks whether SIG_DFL is set as signal handler; depending on the signal it changes the signal number to SIGKILL (for killing the process) or SIGSTOP (for stopping it). After that it treats the three special cases of the SIGKILL, SIGSTOP and SIGCONT signals which cannot be blocked or ignored.

Finally it checks whether the signal shall be ignored; if the signal is neither ignored nor blocked, it sets the pending bit in the target TCB’s sig_pending field and returns.
14.3 Implementation of Signals in Ulx

```c
signo == SIGINT  || signo == SIGPIPE  || signo == SIGTERM ||
signo == SIGUSR1 || signo == SIGUSR2 || signo == SIGPROF ||
signo == SIGSYS || signo == SIGTRAP || signo == SIGVTALRM ||
signo == SIGXCPU || signo == SIGXFSZ) {
    // default: abort, send SIGKILL
    printf("Replacing signal \%d with kill signal (9)\n", signo);
    signo = SIGKILL;  // no handler? kill it
} else if (signo == SIGTTIN || signo == SIGTTOU || signo == SIGTSTP) {
    // default: stop, send SIGSTOP
    printf("Replacing signal \%d with kill signal (9)\n", signo);
    signo = SIGSTOP;  // no handler? kill it
}
}
```

We treat three special cases directly in the `u_kill` function:

- **The SIGSTOP** signal:
  ```c
  case SIGSTOP: {  ⟨u_kill: special cases 563a⟩  };  // cannot ignore/block
                  ⟨u_kill: remove thread from queue 564c⟩
  tcb->state = TSTATE_STOPPED;
  if (pid == current_task) {
      ⟨resign 221d⟩  // enter scheduler
  }
  return 0;
  }
  ```

- **The SIGCONT** signal:
  ```c
  case SIGCONT: {  ⟨u_kill: special cases 563a⟩  +;  // sets TSTATE_READY
                    ⟨add_to_ready_queue 184b⟩  ⟨TSTATE_STOPPED 180a⟩
  if (tcb->state == TSTATE_STOPPED) {  // sets TSTATE_READY
      add_to_ready_queue (pid);  // sets TSTATE_READY
  } // else ignore
  return 0;
  }
  ```

```
#define u_kill, used in chunks 321a, 561b, and 565c.
Uses EINVAL 561c, EPERM 561c, ESRCH 561c, kill 568b, login 584c, printf 601a, set_errno 206b, SIG_DFL 561a,
SIG_IGN 561a, SIGABRT 562a, SIGALRM 562a, SIGBUS 562a, SIGFPE 562a, SIGHUP 562a, SIGILL 562a, SIGINT 562a,
SIGKILL 562a, signal 568b, SIGPIPE 562a, SIGPROF 562a, SIGSTOP 562a, SIGSYS 562a, SIGTERM 562a, SIGTRAP 562a,
SIGTSTP 562a, SIGTTIN 562a, SIGTTOU 562a, SIGUSR1 562a, SIGUSR2 562a, SIGVTALRM 562a, SIGXCPU 562a,
SIGXFSZ 562a, strncmp 594a, TCB 175, and thread_table 176b.
```

We treat three special cases directly in the `u_kill` function:

- **The SIGSTOP** signal:
  ```c
  case SIGSTOP: {  ⟨u_kill: special cases 563a⟩  };  // cannot ignore/block
                  ⟨u_kill: remove thread from queue 564c⟩
  tcb->state = TSTATE_STOPPED;
  if (pid == current_task) {
      ⟨resign 221d⟩  // enter scheduler
  }
  return 0;
  }
  ```

- **The SIGCONT** signal:
  ```c
  case SIGCONT: {  ⟨u_kill: special cases 563a⟩  +;  // sets TSTATE_READY
                    ⟨add_to_ready_queue 184b⟩  ⟨TSTATE_STOPPED 180a⟩
  if (tcb->state == TSTATE_STOPPED) {  // sets TSTATE_READY
      add_to_ready_queue (pid);  // sets TSTATE_READY
  } // else ignore
  return 0;
  }
  ```
The SIGKILL signal:

\[ \langle \text{u_kill: special cases} \rangle + \langle \text{u_kill: remove thread from queue} \rangle \]
\[ \text{case SIGKILL:} \]
\[ \quad \langle \text{u_kill: remove thread from queue} \rangle \]
\[ \quad \text{tcb->used} = \text{false}; \]
\[ \quad \text{tcb->state} = \text{TSTATE_EXIT}; \]
\[ \quad \langle \text{u_kill: write kill message} \rangle \]
\[ \quad \text{wake_waiting_parent_process(pid);} \]
\[ \quad \langle \text{enable scheduler} \rangle \]
\[ \quad \text{if (pid == current_task)} \{
\quad \quad \langle \text{resign} \rangle \quad \text{// enter scheduler} \}
\quad \text{return 0;} \]

Uses current_task 192c, SIGKILL 562a, TSTATE_EXIT 180a, and wake_waiting_parent_process 217a.

We want to notify the user that the process was killed, so we write a “Killed” message onto the terminal that the task uses. For that purpose we need to temporarily change the value of thread_table[current_task].terminal (since printf uses that value to find its target) and restore it after writing:

\[ \langle \text{u_kill: write kill message} \rangle \]
\[ \quad \text{int tmp_term} = \text{thread_table[current_task].terminal;} \]
\[ \quad \text{thread_table[current_task].terminal} = \text{thread_table[pid].terminal;} \]
\[ \quad \text{printf("'nKilled\n");} \]
\[ \quad \text{thread_table[current_task].terminal} = \text{tmp_term;} \]

Uses current_task 192c, printf 601a, and thread_table 176b.

In order to remove the thread from a queue we can only guess what queue it might be on since we do not have a list of all available queues; for example, every lock has its separate queue, and locks can be generated on the fly. We check the standard blocked queues (waiting for a child process, a keyboard, floppy or hard disk event) and the ready queue:

\[ \langle \text{u_kill: remove thread from queue} \rangle \]
\[ \quad \text{switch (tcb->state)} \{
\quad \quad \text{case TSTATE_READY: remove_from_ready_queue (pid);} \quad \text{break;} \]
\[ \quad \quad \text{case TSTATE_WAITFOR: remove_from_blocked_queue (pid, &waitpid_queue);} \quad \text{break;} \]
\[ \quad \quad \text{case TSTATE_WAITKEY: remove_from_blocked_queue (pid, &keyboard_queue);} \quad \text{break;} \]
\[ \quad \quad \text{case TSTATE_WAITFLP: remove_from_blocked_queue (pid, &floppy_queue);} \quad \text{break;} \]
\[ \quad \quad \text{case TSTATE_WAITHD: remove_from_blocked_queue (pid, &harddisk_queue);} \quad \text{break;} \]
\[ \quad \quad \text{case TSTATE_WAITSD: remove_from_blocked_queue (pid, &serial_disk_queue);} \quad \text{break;} \]
\[ \quad \quad \text{default: printf("cannot remove process %d (state: %d) from blocked" \n", pid, tcb->state);} \]
\[ \}

Uses floppy_queue 544d, harddisk_queue 529a, keyboard_queue 323d, printf 601a, remove_from_blocked_queue 186a, remove_from_ready_queue 184c, serial_disk_queue 522a, TSTATE_READY 180a, TSTATE_WAITFLP 180a, TSTATE_WAITFOR 180a, TSTATE_WAITHD 180a, TSTATE_WAITKEY 180a, TSTATE_WAITSD 521c, and waitpid_queue 218b.
Note that we need not (and do not) perform any checks in this function: \( u_{\text{kill}} \) can be called by the kernel itself (which may send any signal to any process), but it cannot be called directly by a process. Sending by a process requires using a system call, and the system call handler will check whether the process is allowed to send the signal to the target process before calling \( u_{\text{kill}} \).

It is also classical for a process to send a signal to itself; that is what the \( \text{raise} \) function does. We will not implement it specifically inside the kernel, but in the user mode library: \( \text{raise}(\text{sig}) \) is the same as \( \text{kill}(\text{getpid}, \text{sig}) \).

Here’s the code for the system call handler:

```c
#include <unistd.h>

void syscall_kill (context_t *r) {
    if (!thread_table[target_pid].used) { // check if target process exists
        set_errno (ESRCH);
        retval = -1; goto end;
    }

    if (signo < 0 || signo > 31) { // check if signal is in range 0..31
        set_errno (EINVAL);
        retval = -1; goto end;
    }

    // check if current process may send a signal
    if ((thread_table[current_task].euid == 0) ||
        (thread_table[target_pid].euid == thread_table[current_task].euid)) {
        retval = u_kill (target_pid, signo);
    } else {
        set_errno (EPERM);
        retval = -1;
    }

    end: r->eax = retval;
}
```

Defines:
- syscall_kill, used in chunk 565.

Uses context_t 142a, current_task 192c, EINVAL 561c, EPERM 561c, ESRCH 561c, euid 573a, raise 568b, set_errno 206b, signal 568b, target_pid, thread_table 176b, and u_kill 562b.
We only allow sending a signal if either the sender’s owner has user ID 0 or if sender and recipient have the same owner.

If sender and receiver are the same process, we have a raise operation, and in that case we will jump into the scheduler: we do not want the current process to continue execution since it might have sent a SIGKILL signal to itself.

(Note that we cannot use the eax_return macro in this function because we may or may not call ⟨resign⟩.)

How can a process declare a signal handler? It just defines a function void *handler (int) (of type sighandler_t) and makes a signal syscall. The internal function for entering a system call is the following:

```c
sighandler_t u_signal(int sig, sighandler_t func) {
    sighandler_t old_func;
    if (sig >= 0 && sig < 32 &&
        sig != SIGKILL && sig != SIGSTOP &&
        sig != SIGCONT) {
        old_func = thread_table[current_task].sighandlers[sig];
        thread_table[current_task].sighandlers[sig] = func;
    } else {
        old_func = SIG_ERR; // wrong signal number
    }
    return old_func;
}
```

Defines: u_signal, used in chunk 566.
Uses current_task 192c, SIG_ERR 561a, SIGCONT 562a, sighandler_t 560a, SIGKILL 562a, signal 568b, SIGSTOP 562a, and thread_table 176b.

The function sets the new signal handler and returns the address of the old handler (or 0 if there was none); that way the process can keep a copy of the old handler address in order to restore it at a later point.

As usual, we need to provide a system call so that a process can access this function.

```c
void syscall_signal(context_t *r) {
    // ebx: signal number
    // ecx: address of signal handler
    int signo = r->ebx;
    sighandler_t func = (sighandler_t)r->ecx;
    func = u_signal(signo, func);
    eax_return(func);
}
```

It performs the already well-known transfers of register values to arguments and of the return value to the EAX register:
14.3 Implementation of Signals in Ulx

Defines:
  syscall_signal, used in chunks 566c and 567a.

Uses context_t 142a, eax_return 174a, sighandler_t 560a, and u_signal 566b.

\(\langle \text{initialize syscalls 173d}\rangle \equiv \quad (44b) <565a \ 583b> \quad [567a]\)

\[\begin{align*}
&\text{install syscall_handler (__NR_signal, syscall_signal);} \\
\end{align*}\]

Uses __NR_signal 204c, install_syscall_handler 201b, and syscall_signal 566d.

Finally, we need to add code to the scheduler: it needs to check for pending signals and—if there are any—launch the registered handler function or execute the standard action. Running a handler can be achieved by modifying the process’ stack. We loop over the set of possible signal numbers (0–31) and check the bits in sig_pending. (Remember that t_new points to the newly chosen process’ TCB.)

Note that we only modify the stack (and the EIP value) if the process is currently running in user mode (i.e., t_new->regs.eip < 0xc0000000), because otherwise we would change the kernel stack (making the signal handler run with kernel privileges).

\(\langle \text{scheduler: check pending signals 567b}\rangle \equiv \quad (277b) \quad [567b]\)

\[\begin{align*}
&\text{for (int signo = 0; signo < 32; signo++) } \{ \\
&\quad \text{if ((t_new->sig_pending & (1<<signo)) != 0) } \quad \text{// signal is pending} \\
&\qquad \&\& \text{t_new->regs.eip < 0xc0000000} \} \quad \text{// and thread is in user mode} \\
&\quad \text{if (t_new->sighandlers[signo] == SIG_DFL) } \{ \\
&\qquad \text{; } \quad \text{// default action, cannot happen} \\
&\text{\quad } \} \quad \text{else if (t_new->sighandlers[signo] == SIG_IGN) } \{ \\
&\text{\quad ; } \quad \text{// ignored, should not happen} \\
&\text{\quad } \} \quad \text{else } \{ \\
&\text{\quad // handler exists} \\
&\text{\quad \langle \text{modify process to execute signal handler 567c}\rangle } \\
&\text{\quad } \} \\
&\text{t_new->sig_pending &= ~(1<<signo); } \quad \text{// remove bit} \\
&\text{\quad break; } \quad \text{// only one handler at a time} \\
&\text{\}} \\
\end{align*}\]

Uses SIG_DFL 561a, SIG_IGN 561a, and t_new 276c.

When a handler exists, we modify both the (user mode) stack and the EIP register.

\(\langle \text{modify process to execute signal handler 567c}\rangle \equiv \quad (567b) \quad [567c]\)

\[\begin{align*}
&\text{\quad // Note: t_new->regs has already been copied to } r \\
&\text{\quad memaddress oldeip = r->eip;} \\
&\text{\quad r->eip = (memaddress)t_new->sighandlers[signo];} \\
&\text{\quad // push signal number and oldeip on user mode stack} \\
&\text{\quad POKE_UINT (r->useresp, signo); } \quad \text{// overwrites old RET address} \\
&\text{\quad r->useresp -= 4;} \\
&\text{\quad POKE_UINT (r->useresp, oldeip); } \quad \text{// writes new RET address} \\
\end{align*}\]

Uses memaddress 46c, POKE_UINT 117, and t_new 276c.
14.3.1 Library Functions

Now we can provide the user mode library functions for the two new system calls; we also define a \texttt{raise} function which sends a signal to the own process:

\begin{verbatim}
(ulixlib function prototypes 174c) +≡
int kill (int pid, int signo);
int raise (int signo);
sighandler_t signal (int sig, sighandler_t func);
\end{verbatim}

\begin{verbatim}
(ulixlib function implementations 174d) +≡
int kill (int pid, int signo) {
  return syscall3 (__NR_kill, pid, signo);
}
int raise (signo) {
  return kill (getpid (), signo);
}
sighandler_t signal (int sig, sighandler_t func) {
  return (sighandler_t)syscall3 (__NR_signal, sig, (memaddress)func);
}
\end{verbatim}

Defines:
\begin{itemize}
  \item \texttt{kill}, used in chunks 321a, 431, and 562b.
  \item \texttt{raise}, used in chunk 565c.
  \item \texttt{signal}, used in chunks 561a, 562b, 565c, 566b, and 568a.
\end{itemize}

Uses \texttt{__NR_kill} 204c, \texttt{__NR_signal} 204c, \texttt{getpid} 223b, \texttt{memaddress} 46c, \texttt{sighandler_t} 560a, and \texttt{syscall} 203c.

14.3.2 Example Program

We end this chapter with an example program that you can also find on the Ulix disk image: \texttt{/bin/sigtest} forks, registers two signal handlers in the child and sends two signals twice from the parent process. Both processes otherwise print sequences of “p”s or “c”s to show that they are active. The expected behavior is that the output is interrupted with four messages from the two signal handlers.

\begin{verbatim}
#include "../ulixlib.h"

void handler1 (int sig);
void handler2 (int sig);
void waste_time ();

int main (int argc, char *argv[]) {
  int i; int pid = fork ();
  if (pid == 0) { // child
    signal (5, handler1); // register handler 1
    signal (6, handler2); // register handler 2
    for (i = 0; i < 40; i++) { printf("c"); waste_time (); }
    exit (0);
  }
\end{verbatim}
} else { // parent
    for (i = 0; i < 20; i++) { printf("p"); waste_time(); }
    kill (pid, 5);
    kill (pid, 6);
    for (i = 0; i < 22; i++) { printf("p"); waste_time(); }
    printf("--done
");
    exit (0);
}

void handler1 (int sig) { printf("\nHandler 1\n"); }
void handler2 (int sig) { printf("\nHandler 2\n"); }

void waste_time (){
    long int i, z;
    for (i=0L; i<1000000L; i++)  z = i*i - (i+1)*(i+1);
}
Unix and all Unix-derived operating systems are multi-user-capable. There are configuration files which contain the information about all known users, and there is also a list of groups that users can be members of. Each user has a standard group but may—additionally—have the membership of one or more further groups. The id command lists the user ID, a corresponding user name, the standard group ID and name and a list of all additional group memberships. The following is an example from a Linux installation:

```
$ id
uid=1000(esser) gid=1000(esser) groups=1000(esser),24(cdrom),25(floppy),29(audio),
   30(dip),44(video),46(plugdev)
```

Internally, the systems use only the numerical IDs; tools which display user and group names will look them up using functions such as getpwnam or getgrnam. The inode of each file stores a user ID and a group ID, the first one expresses file ownership by the specific user who uses this user ID, whereas the second one establishes an additional group ownership. Members of a group may (or may not) have access rights to a file which has the corresponding group ID. The group is sometimes called the owner group or group owner, both variants mean the same thing.

Classical Unix systems can associate nine access permissions with every file and every directory: for files, these are the three basic rights to read (r), write (w) or execute (x) a file, for directories the interpretation is listing, modifying and searching/entering a directory (where “searching” means getting the inode number of a file in the directory and “entering” means setting the current working directory to a folder). Each of these three permissions can be granted or denied to the file owner, all members of one specific group of users and all other users. This leads to nine permissions typically represented as a nine-character string of the form rwxrwxrwx where missing rights are expressed by exchanging a letter
with a minus character. For example, the permission string `rwxr-x---` for a file lets the file owner read, write, and execute that file (`rwx`); it lets members of the file’s owner group read and execute (but not write) it (`r-x`), and other users (those who neither are the owner nor belong to the owner group) cannot access the file at all (`---`).

There are a few more attributes which can be set for files and directories which have specific effects:

**SUID:** If the *Set User ID bit* (SUID) is set for an executable file, the *effective user ID* (EUID) is set to the ID of the file owner. For example, the `passwd` tool uses this feature: It may be called by any user (who wants to change his own password), but it needs administrator privileges to modify the password file which is non-writeable for ordinary users. In order to allow this, the `passwd` program is set to belong to the `root` user and has the SUID bit set. When a regular user starts `passwd`, the effective user ID of the process is set to 0 (the user ID of `root`), and write access to the password file is granted.

In the directory listing of an executable file with a set SUID bit, the first `x` in the permissions string is replaced with an `s` to show this. It is also possible to set the SUID bit on a non-executable file which will show up with a capital `S`, however this is useless.

**SGID:** The *Set Group ID bit* (SGID) has a function that is similar to the SUID bit’s, however it changes the *effective group ID* (EGID). It appears as a capital `S` in the group permissions block.

On some Unix systems, a non-group-executable file which has the SGID bit set is marked for *mandatory locking* (for the Linux OS, see https://www.kernel.org/doc/Documentation/filesystems/mandatory-locking.txt).

**Sticky Bit:** The *sticky bit* appears as a `t` letter in the last position of the permissions string (replacing the `x` which shows the world-executable state). Similar to the difference between `s` and `S`, if a file is set to be sticky but not world-executable, it appears as a capital `T`.

The effect of a set sticky bit depends on the Unix variant. For example, the Linux man page for `chmod` says: “For directories, it prevents unprivileged users from removing or renaming a file in the directory unless they own the file or the directory; this is called the restricted deletion flag for the directory, and is commonly found on world-writable directories like `/tmp`.”

On traditional Unix systems, a sticky bit on an executable file caused the system to keep the program code in memory after termination of a process, with the idea that it did not have to be reloaded when the same program was executed again. That is where the term “sticky” comes from.

**access control list (ACL):** Many modern Unix-like systems provide further access mechanisms through *extended attributes* or *access control lists* (ACLs). This is a feature which was not available in classical Unix, and we will not implement it for Ulx, either.
15.1 Users and Groups in Ulx

In order to implement the user and group concepts in Ulx, each thread control block needs four new entries: a user ID (uid573a), a group ID (gid573a) and effective user and group IDs (euid573a, egid573a) plus a third set called real user ID and real group ID (ruid573a, rgid573a):

\[
\langle \text{more TCB entries 158c}\rangle + \equiv
\]

- word uid; // user ID
- word gid; // group ID
- word euid; // effective user ID
- word egid; // effective group ID
- word ruid; // real user ID
- word rgid; // real group ID

Defines:

- egid, used in chunks 487a, 573b, 576–78, 580–83, and 587d.
- euid, used in chunks 487a, 565c, 573b, 576–78, 580–83, 587d, and 588b.
- gid, used in chunks 478b, 573b, 580–84, and 587d.
- rgid, used in chunks 573b and 582a.
- ruid, used in chunks 573b and 582a.
- uid, used in chunks 478b, 573b, 580–85, and 587d.

The purpose of the real user and group IDs is to always remember which user started the process: it will (normally) not change over a process’ lifetime whereas uid573a, gid573a and the effective IDs can be changed by a running process using setuid584c, setgid584c, seteuid584c and setegid584c functions; we will explain soon why we need to keep track of so many IDs.

0 is a special ID, both for users and groups. It belongs to the root user and root group, respectively. The root user is the system administrator who can override all permission settings (e.g. open files for which no read permissions have been set at all). All other IDs have no special meaning, though it is standard on many systems to reserve IDs below 100 (or below 1000) for system services with regular user IDs starting at 100 (or 1000). The default setting of Ulx processes is to run with root privileges. The function start_program_from_disk159 sets all four IDs to 0:

\[
\langle \text{start program from disk: set uid, gid, euid, egid 573b}\rangle \equiv
\]

- thread_table[tid].uid = 0;
- thread_table[tid].gid = 0;
- thread_table[tid].euid = 0;
- thread_table[tid].egid = 0;
- thread_table[tid].ruid = 0;
- thread_table[tid].rgid = 0;

Uses egid573a, euid573a, gid573a, rgid573a, ruid573a, thread_table176b, and uid573a.

The Linux man page for setreuid states:

“POSIX.1 does not specify all of [the] possible ID changes that are permitted on Linux for an unprivileged process. For setreuid(), the effective user ID can be made the same as the real user ID or the saved set-user-ID, and it is unspecified whether unprivileged processes may set the real user ID to the real user ID, the effective user ID or the saved set-user-ID. For setregid(), the real group ID can be changed to the value of the saved set-group-ID, and
the effective group ID can be changed to the value of the real group ID or the saved set-group-ID. The precise details of what ID changes are permitted vary across implementations.

For URIX we will take a simplified approach with the following rules:

- Processes can invoke the `setuid` and `setgid` system calls to change their user and group IDs (`uid_{573a}`, `gid_{573a}`), and this will also set the effective user/group ID to the same value. These functions shall only succeed if the process had an `uid_{573a}` value of 0 or if the desired new ID is identical to either the current user ID or the current effective user ID.

- Processes can invoke the `seteuid` and `setegid` system calls to change their effective user and group IDs (`euid_{573a}`, `egid_{573a}`). This will not change the user/group ID (`uid_{573a}`, `gid_{573a}`). These functions shall only succeed if the process had an `uid_{573a}` value of 0 or if the desired new effective ID is identical to either the current user ID or the current effective user ID.

- The `login` system call which expects a user ID and a password allows changing the real and effective user and group IDs as long as the correct password was supplied. `login` is also the only system call which sets the real user and group IDs, thus allowing the system to keep track of which user initially started a process.

- Whenever file access is checked, the system looks at the effective IDs to establish the permissions of the current process.

The consequence of these rules is that changing, for example, the user ID and effective user ID via `setuid_{584c}` is a permanent change which cannot be undone, whereas a modification of only the effective user ID via `seteuid_{584c}` can be followed up with another `seteuid_{584c}` call that sets it back to the original value. Once `uid_{573a}` and `euid_{573a}` have the same non-zero value, there is only one way to go back to a different ID or effective ID: it has to make a `login` system call which reperforms authorization against the password database.

Whenever a process forks, the new process inherits all IDs from its parent.

### 15.1.1 Checking Permissions

Before we delve into the implementation of `login_{584c}` and the `set*id` functions, let's look at how the `open_{429b}` and `execv_{235e}` functions use the effective IDs to test whether access can be granted or not.

Checking whether a user may access a file seems to be a simple task: The OS just needs to look up the file’s inode and check the file owner, group and permissions stored in the corresponding fields. But this is only half of the work we need to do because there is also the issue of getting into the directory which holds the file—after all, if the directory does not provide sufficient read permissions, it is forbidden to find the inode number that belongs to a filename entry.

The access rules are also different for opening (existing) files and newly creating files.
• In order to *access* an existing file (either for reading or writing), the user must have read and execute permissions on all directories which are passed on the way to file, starting with the root directory `/.

• In order to *create* a new file, the same permissions are needed, and the user must also have write permission for the last directory (in which the file will be created).

**ULIX** will first check whether the file already exists. It will then follow the path from either the existing file or from the target directory, all the way up to the root directory and test for each directory whether the needed permissions are available. We can do this in a loop which repeatedly uses `splitpath` to strip the last element of the current path.

Pseudo code for this loop looks like this:

\[\langle\text{pseudo code for checking permissions}\rangle\]

```
curpath = abspath;
step = 0;
for (; ;) {
    // check current path
    if (step == 0 && fileexists (abspath)) {
        // can we access the (existing) file?
        ok = check (curpath, oflag);
    } else
    if (step == 1 && !fileexists (abspath)) {
        // can we create the file, i.e., write
        // to the target directory?
        ok = check (curpath, "rwx");
    } else {
        ok = check (curpath, "rx");
    }
    if (!ok) return false;
    if (curpath == "/") return true;
    // access denied
    // move to upper directory
    lastpath = curpath;
    splitpath (lastpath, curpath, tmp);
    step++;
}
```

Note that it would be more efficient to directly implement the access checks in the loop inside the `mx_open` function of the Minix subsystem which traverses the path down from the root to the file, but we did not want to discuss access rights when we presented the code for opening a file. Also, our method is independent of the filesystem (e.g. Minix). But it does more than duplicate the efforts of walking down the path, so it would be unacceptable for production systems.

In each step both user, group and world access permissions need to be considered: for example, if user permissions allow access to the first directory, group permissions allow access to the second directory and world permissions allow access to the third directory, then that is an acceptable sequence. Only if none of the directory permissions allow access (somewhere in the path), access must be denied.
We start with the implementation of

\( \langle \text{function prototypes } 45a \rangle + \equiv \)

\[
\text{boolean fileexists (char *abspath);} \\
\]

for which we can use the \( u_{\text{stat}} \) function:

\( \langle \text{function implementations } 100b \rangle + \equiv \)

\[
\text{boolean fileexists (char *abspath) } \\
\text{struct stat tmp; } \\
\text{// stat info will not be used} \\
\text{return (u_{\text{stat}} (abspath, &tmp) != -1); } \quad // -1 \text{ means: does not exist} \\
\]

Defines:

\( \text{fileexists, used in chunk 576a.} \)

Uses \( \text{stat 429b 489b} \) and \( u_{\text{stat}} 421d. \)

For checking the permissions on any level, we write a function

\( \langle \text{function prototypes } 45a \rangle + \equiv \)

\[
\text{boolean check_access (char *path, word euid, word egid, word mode);} \\
\]

which evaluates the owner, group and world access permissions, depending on the provided user and group IDs (\( euid_{573a} \) and \( egid_{573a} \)):

\( \langle \text{function implementations } 100b \rangle + \equiv \)

\[
\text{boolean check_access (char *path, word euid, word egid, word mode) } \\
\text{struct stat st; } \\
\text{int res = u_{\text{stat}} (path, &st); } \quad // \text{get file permissions} \\
\text{if (res == -1) \\
\text{&& (mode & O\_CREAT) == 0) } \\
\text{set_errno (ENOENT); } \quad // \text{file not found} \\
\text{return false;} \\
\]

\[
\text{if (res == -1) \\
\text{&& (mode & O\_CREAT) != 0) } \\
\text{(check_access special case: create file 577b)} \\
\]

\[
\text{if (euid == st.st_uid) } \\
\text{// case 1: user owns the file} \\
\text{res = check_perms (CHECK\_USER, mode, st.st_mode);} \\
\text{else if (egid == st.st_gid) } \\
\text{// case 2: group matches owner group} \\
\text{res = check_perms (CHECK\_GROUP, mode, st.st_mode);} \\
\text{else } \\
\text{// case 3: world access?} \\
\text{res = check_perms (CHECK\_WORLD, mode, st.st_mode);} \\
\]

\[
\text{if (!res) set_errno (E\_ACCES); } \\
\text{return res;} \\
\]

Defines:

\( \text{check\_access, used in chunk 577c.} \)

Uses \( \text{CHECK\_GROUP 579b, check\_perms 579c, CHECK\_USER 579b, CHECK\_WORLD 579b, E\_ACCES 577a, egid 573a, ENOENT 577a, euid 573a, O\_CREAT 460b, set\_errno 206b, stat 429b 489b, and u\_stat 421d.} \)
(The function check_perms checks just one possible way of getting access, e.g. via the owner permissions. We will describe it soon.)

```
#error constants 370a) +≡
#define ENOENT 2 // No such file or directory
#define EACCES 13 // Permission denied
```

Defines:
ENOENT, used in chunks 576 and 577.
EACCES, used in chunks 576d and 577b.

There’s also one special case we need to consider: when we create a new file, it does not yet exist, and so u_stat will return −1. For file creation we have to check the access permissions of the directory in which the new file is to be placed.

```
⟨check_access special case: create file 577b⟩≡
char dirname[256], basename[256];
splitpath (path, dirname, basename);
res = u_stat (dirname, &st); // get directory permissions
if (res == -1) {
    set_errno (ENOENT); // directory not found
    return false;
}
if (euid == st.st_uid) {
    // case 1: user owns the directory
    res = (((st.st_mode >> CHECK_USER) & 0x7) == 0x7); // 7: rwx
} else if (egid == st.st_gid) {
    // case 2: group matches owner group
    res = (((st.st_mode >> CHECK_GROUP) & 0x7) == 0x7);
} else {
    // case 3: world access?
    res = (((st.st_mode >> CHECK_WORLD) & 0x7) == 0x7);
}
if (!res) set_errno (EACCES);
return res;
```

Uses basename 455b, CHECK_GROUP 579b, CHECK_USER 579b, CHECK_WORLD 579b, dirname 455b, EACCES 577a, egid 573a, ENOENT 577a, euid 573a, set_errno 206b, splitpath 455a, and u_stat 421d.

The u_open function calculates the absolute path of the file it shall open and stores it in abspath; the requested mode for opening is held in the function’s oflag parameter. It can then call u_stat to read its access permissions as well as the permissions of the directories involved:

```
⟨u_open: check permissions 577c⟩≡
boolean access_ok = false;
word euid = thread_table[current_task].euid;
word egid = thread_table[current_task].egid;
struct stat st;

if (euid == 0) {
    access_ok = true; // user root can do anything
} else {
```
// loop over the directories
char old_dirname[256]; char dirname[256]; char rest[256];
strncpy (dirname, abspath, 256);
for (;;) {
    strncpy (old_dirname, dirname, 256);
    splitpath (old_dirname, dirname, rest); // ignore rest
    u_stat (dirname, &st);
    access_ok = ⟨u_open: access to directory is ok 578⟩;
    if (!access_ok) {
        set_errno (EACCES);
        return -1;
    }
    if (strlen (dirname) == 1) break; // reached root directory
}
// finally: check file permissions
access_ok = check_access (abspath, euid, egid, oflag);
}
if (!access_ok) {
    return -1; // wrong permissions
}
Uses check_access 576d, current_task 192c, dirname 455b, EACCES 577a, egid 573a, euid 573a, set_errno 206b,
splitpath 455a, stat 429b 489b, strlen 594a, strncpy 594b, thread_table 176b, and u_stat 421d.

A process may only access a directory if it can read and "execute" it, and that’s possible if one of the following three conditions is fulfilled:

• the process’ effective user (determined by the euid field) owns the file and the user read and execute bits (0500) are set in the access permissions,

• the process’ effective user does not own the file, the process’ effective group (determined by the egid field) is the file’s owner group and the group read and execute bits (0050) are set in the access permissions,

• or neither the user and group fields of the file match the effective user or group, but the world permissions allow read and execute access (0005).

Thus, checking whether the process may access a directory (read and execute, 1 + 4 = 5) can be done with the following code:

[578] ⟨u_open: access to directory is ok 578⟩≡
⟨u_open: access to directory is ok 578⟩≡
⟨u_open: access to directory is ok 578⟩≡
⟨u_open: access to directory is ok 578⟩≡
⟨u_open: access to directory is ok 578⟩≡
⟨u_open: access to directory is ok 578⟩≡

( // user may have access, r-x------- ?
   (((euid == st.st_uid) && (st.st_mode & 0500) == 0500)) ||
   // group may have access (if wrong user), ----r-x--- ?
   (((euid != st.st_uid) && (egid == st.st_gid) && (st.st_mode & 0050) == 0050)) ||
   // others may have access (wrong user, group), --------r-x ?
   (((euid != st.st_uid) && (egid != st.st_gid) && (st.st_mode & 0005) == 0005))
)

Uses egid 573a and euid 573a.

Now we need to provide the function
\(\langle \text{function prototypes} \rangle \)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)

\[
\begin{align*}
\text{boolean check_perms (short what, word req_mode, word perms);}
\end{align*}
\]

which accepts one of the three constants

\(\langle \text{constants} \rangle \)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)

\[
\begin{align*}
#define \text{CHECK_USER} 6 \\
#define \text{CHECK_GROUP} 3 \\
#define \text{CHECK_WORLD} 0
\end{align*}
\]

Defines:

\text{CHECK_GROUP}, used in chunks 576d and 577b.
\text{CHECK_USER}, used in chunks 576d and 577b.
\text{CHECK_WORLD}, used in chunks 576d and 577b.

and a requested mode \(\text{req_mode}\) and the file permissions \(\text{perms}\). The lowest two bits of \(\text{req_mode}\) are either \(00\) (in case of \(\text{O_RDONLY}\)), \(01\) (in case of \(\text{O_WRONLY}\)) or \(10\) (in case of \(\text{O_RDWR}\)). We can check them by looking at \(\text{req_mode} \& 0x3\).

File access permissions can be found in the lowest nine bits of \(\text{perms}\).

- If we want to check world permissions (for “others”), we look at the lowest three bits, \(\text{perms} \& 0x7\).
- For the group permissions we can first right-shift \(\text{perms}\) so that we drop the lowest three bits, that is, we look at \((\text{perms} >> 3) \& 0x7\).
- Finally, for the owner permissions, we need a right-shift of six bits, which gives us \((\text{perms} >> 6) \& 0x7\).

By setting \text{CHECK_USER}\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\), \text{CHECK_GROUP}\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\), and \text{CHECK_WORLD}\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\) to the necessary amount of shifting (6, 3, 0), we can do this automatically:

\(\langle \text{function implementations} \rangle \)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)\(\rightarrow\)

\[
\begin{align*}
\text{boolean check_perms (short what, word req_mode, word perms) \{} \\
\hspace{1cm} \text{boolean req_read = ((req_mode & 0x3) == \text{O_RDONLY}) | ((req_mode & 0x3) == \text{O_RDWR});} \\
\hspace{1cm} \text{boolean req_write = ((req_mode & 0x3) == \text{O_WRONLY}) | ((req_mode & 0x3) == \text{O_RDWR});} \\
\hspace{1cm} \text{word check = (perms >> what) \& 0x7;} \\
\hspace{1cm} \text{if (req_read && ((check & 0x4) != 0x4)) return false; // read perm. failure} \\
\hspace{1cm} \text{if (req_write && ((check & 0x2) != 0x2)) return false; // write perm. failure} \\
\hspace{1cm} \text{set_errno (0);} \\
\hspace{1cm} \text{return true; // both are ok} \\
\text{\}}
\end{align*}
\]

Defines:

\text{check_perms}, used in chunks 576d and 579a.
Uses \text{O_RDONLY} 460b, \text{O_WRONLY} 460b, \text{O_RDWR} 460b, and \text{set_errno} 206b.

Note that some of this behavior is not obvious: for example, consider a user with user ID 1000 and group ID 1000 who wants to open the following file in \(\text{O_RDWR}\) mode:

\[
-r--rw-r-- 1000 1000 filename
\]

The file belongs to him and also to his group, but the owner permissions do not contain the right to write to the file. Access will be denied in this case, even though the group permissions would allow writing. In this case it just makes no sense that the owner’s
write access is not enabled in the permission string. The user would first have to fix this
situation via chmod.

The u_execv function must also check whether it may load an application: this re-
quires that the read and executable flags are set for the owner, group or others. We’re
leaving this as an exercise to you.

// TO DO, see "Exercises" section.

15.1.2 Changing User and Group IDs

Now we can deal with the login and setid system calls. As usual we start with the
kernel functions which do the real work:

All of these functions shall return 0 if they were successful and −1 otherwise. The
implementations are simple, we only have to follow the rules described earlier:

int u_setuid (word uid); // set user ID
int u_setgid (word gid); // set group ID
int u_seteuid (word euid); // set effective user ID
int u_setegid (word egid); // set effective group ID
int u_login (word uid, char *pass);
The implementations of `u_seteuid` and `u_setegid` are almost identical to the above code, they just skip setting the `uid` or `gid` elements of the TCB:

```c
int u_seteuid (word uid) {
    TCB *t = &thread_table[current_task];
    if (t->uid == 0 || uid == t->uid || uid == t->euid) {
        t->euid = uid; // set the EUID (only!)
        return 0; // success
    } else {
        return -1; // failure
    }
}

int u_setegid (word gid) {
    TCB *t = &thread_table[current_task];
    if (t->uid == 0 || gid == t->gid || gid == t->egid) {
        t->egid = gid; // set the EGID (only!)
        return 0; // success
    } else {
        return -1; // failure
    }
}
```

The `u_login` function is just a little more complicated: In a real Unix system it would look up the password hash stored in `/etc/passwd`, `/etc/shadow` or some similar file, calculate the hash from the password that was provided as the second argument, compare the two hashes and then decide on setting all user and group IDs (including the real user and group IDs `ruid` and `guid`). Since we don’t want to include a hash function in the ULIx source code, we just store the plaintext password in the file. We also restrict the password file size to 1024 bytes since the ULIx kernel has no advanced functions for line reading; we read one block of data and parse it byte by byte.


```c
int u_login (word uid, char *pass) {
    TCB *t = &thread_table[current_task];
    char passwords[BLOCK_SIZE];
    char *words[128];
    int fd = u_open("/etc/passwd", O_RDONLY, 0);
    if (fd == -1) return -1;  // fail: no password database
    int size = u_read(fd, &passwords, BLOCK_SIZE);
    u_close(fd);
    int pos; int index = 0;  // position in words array
    words[index++] = (char*)&passwords;  // split
    for (pos = 1; pos < size; pos++) {
        if (passwords[pos] == ':' || passwords[pos] == '\n') {
            passwords[pos] = 0;  // terminate string
            words[index++] = ((char*)&passwords)+pos+1;
        }
    }
    for (int i = 0; i < index/5; i++) {  // search
        if ((atoi(words[5*i+2]) == uid) && strequal(words[5*i+1], pass)) {  // password matches
            int gid = atoi(words[5*i+3]);  // get group ID
            u_chdir(words[5*i+4]);  // make home directory the cwd
            t->uid = t->euid = t->ruid = uid;
            t->gid = t->egid = t->rgid = gid;  // success
            return 0;
        }
    }
    return -1;  // fail
}
```

Defines:
`u_login`, used in chunk 583a.

Uses `atoi` 595, `BLOCK_SIZE` 440a, `current_task` 192c, `cwd` egid 573a, euid 573a, gid 573a, 0_RDONLY 460b, `passwd` 584d, `passwords` 584d, `rgid` 573a, `ruid` 573a, `strequal` 596a, `TCB` 175, `thread_table` 176b, `u_chdir` 432e, `u_close` 418a, `u_open` 412c, `u_read` 414b, and `uid` 573a.

As usual we need to provide system calls for these functions:

```c
void syscall_setuid (context_t *r);
void syscall_setgid (context_t *r);
void syscall_seteuid (context_t *r);
void syscall_setegid (context_t *r);
void syscall_login (context_t *r);
```
15.1 Users and Groups in Ulx

(\textit{syscall functions} \texttt{174b}) +\equiv (\texttt{202b} \triangleleft \texttt{566d} \triangleright \texttt{587d}) \left[ \texttt{583a} \right]

\begin{verbatim}
void syscall_setuid (context_t *r) {
    // ebx: uid
    eax_return ( u_setuid (r->ebx) );
}

void syscall_setgid (context_t *r) {
    // ebx: gid
    eax_return ( u_setgid (r->ebx) );
}

void syscall_seteuid (context_t *r) {
    // ebx: euid
    eax_return ( u_seteuid (r->ebx) );
}

void syscall_setegid (context_t *r) {
    // ebx: egid
    eax_return ( u_setegid (r->ebx) );
}

void syscall_login (context_t *r) {
    // ebx: uid, ecx: password
    eax_return ( u_login (r->ebx, (char*)r->ecx) );
}
\end{verbatim}

Defines:
- syscall_login, used in chunk 583b.
- syscall_setgid, used in chunk 583b.
- syscall_seteuid, used in chunk 583b.
- syscall_setegid, used in chunk 583b.
- syscall_setuid, used in chunks 582b and 583b.

Uses context_t 142a, eax_return 174a, egid 573a, euid 573a, gid 573a, u_login 582a, u_setegid 581,
u_seteuid 581, u_setgid 580c, u_setuid 580c, and uid 573a.

and enter the new handler function in the system call table:

\begin{verbatim}
\langle \textit{initialize syscalls} \texttt{173d}) +\equiv (\texttt{44b} \triangleleft \texttt{567a} \triangleright \texttt{587e}) \left[ \texttt{583b} \right]

install_syscall_handler (__NR_setuid32, syscall_setuid);
install_syscall_handler (__NR_seteid32, syscall_seteuid);
install_syscall_handler (__NR_setreuid32, syscall_setreuid32);
install_syscall_handler (__NR_setregid32, syscall_setregid32);
install_syscall_handler (__NR_login, syscall_login);

\end{verbatim}

Uses __NR_login 584a, __NR_setegid32 204c, __NR_setreuid32 204c, __NR_setregid32 204c, __NR_setuid32 204c,
install_syscall_handler 201b, syscall_login 583a, syscall_setegid 583a, syscall_seteuid 583a,
syscall_setregid 583a, and syscall_setuid 583a.

(Note that the syscall numbers __NR_setreuid32 204c and __NR_setregid32 204c do not really
match the corresponding functions (seteuid 584c and setegid 584c), but they are the closest
candidates, so we chose them instead of reserving new numbers. We must, however, declare the system call number __NR_login 584a;
(public constants 46a) +≡

#define __NR_login 523

Defines:
__NR_login, used in chunks 583b and 584c.

The user mode library functions just make the system calls:

(ulixlib function prototypes 174c) +≡

int setuid (word uid); // set user ID
int setgid (word gid); // set group ID
int seteuid (word euid); // set effective user ID
int setegid (word egid); // set effective group ID
int login (word uid, char *pass);

(ulixlib function implementations 174d) +≡

int setuid (word uid) { return syscall2 (__NR_setuid32, uid); }
int setgid (word gid) { return syscall2 (__NR_setgid32, gid); }
int seteuid (word uid) { return syscall2 (__NR_setreuid32, uid); }
int setegid (word gid) { return syscall2 (__NR_setregid32, gid); }
int login (word uid, char *pass) {
    return syscall3 (__NR_login, uid, (unsigned int) pass); }

Defines:
login, used in chunks 191a and 562b.
Uses __NR_login 584a, __NR_setgid32 204c, __NR_setegid32 204c, __NR_setreuid32 204c, __NR_setuid32 204c, gid 573a, syscall2 203c, syscall3 203c, and uid 573a.

In order to let user mode programs look up /etc/passwd entries, we implement some functions in the library which fill data structures of the following type:

(ulixlib type definitions 584d) ≡

struct passwd {
    char pw_name[32]; // user name
    char pw_passwd[32]; // password
    word pw_uid; // user ID
    word pw_gid; // group ID
    char *pw_gecos; // long name (ULIX: unused)
    char pw_dir[32]; // home directory
    char *pw_shell; // shell (ULIX: unused)
};

Defines:
passwd, used in chunks 582a and 585.

The functions

(ulixlib function prototypes 174c) +≡

int getpwnam_r (const char *name, struct passwd *pwd,
    char *buffer, int bufsize, struct passwd **result);
int getpwuid_r (word uid, struct passwd *pwd,
    char *buffer, int bufsize, struct passwd **result);

have to read the password file, search it for the username (or user ID) and then fill the supplied data structure. Since both reading the file and storing the data in the structure are the same in both functions, we use two code chunks that deal with these tasks.
15.1 Users and Groups in Ulx

\textit{ulixlib function implementations} 174d +\equiv 48b <584c 587b> [585a]

\begin{verbatim}
int getpwnam_r (const char *name, struct passwd *pwd,
    char *buffer, int bufsize, struct passwd **result) {
    \textit{(get password entry: read password file into passwords and parse it) 585b}
    int i; for (i = 0; i < index/5; i++) {
        if (strequal (words[5*i], name)) { \textit{\langle get password entry: fill target buffers 586a\rangle}
            return 0; \textit{\langle success\rangle}
        }
    }
    return -1; \textit{\langle fail\rangle}
}

int getpwuid_r (word uid, struct passwd *pwd,
    char *buffer, int bufsize, struct passwd **result) {
    \textit{(get password entry: read password file into passwords and parse it) 585b}
    int i; for (i = 0; i < index/5; i++) {
        if (atoi (words[5*i+2]) == uid) { \textit{\langle get password entry: fill target buffers 586a\rangle}
            return 0; \textit{\langle success\rangle}
        }
    }
    return -1; \textit{\langle fail\rangle}
}
\end{verbatim}

Uses \texttt{atoi 595, getpwnam_r, getpwuid_r, passwd 584d, strequal 596a, and uid 573a.}

with

\begin{verbatim}
#define PASSWD_SIZE 1024
char passwords[PASSWD_SIZE] = "passwords"; char *words[128];
int fd = open ("/etc/passwd", O_RDONLY);
if (fd == -1) {
    printf ("Cannot open /etc/passwd\n");
    return -1; \textit{\langle fail: no password database\rangle}
}

int size = read (fd, (char*)passwords, PASSWD_SIZE);
if (size == -1) {
    printf ("Cannot read /etc/passwd, fd = %d\n", fd);
    return -1; \textit{\langle fail: cannot read from password database\rangle}
}
close (fd);

int index = 0; \textit{\langle position in words array\rangle}
words[index++] = (char*)passwords;
int pos; for (pos = 1; pos < size; pos++) {
    if (passwords[pos] == ':' || passwords[pos] == '\n') {
        passwords[pos] = 0; \textit{\langle terminate string\rangle}
        words[index++] = ((char *)&passwords)+pos+1;
    }
\end{verbatim}
15.1.3 The su Program

Using the login function we can create a simple su implementation which reads in the password and tries to log in as root. It launches a new shell that runs with root privileges. When that shell is left, control returns to the original shell.

```c
#include "../ulixlib.h"

int main () {
    char password[32]; printf("Enter root password: ");
    ureadline ((char*)&password, sizeof(password)-1, false); // no echo
    printf("\n");
    if (login (0, password) == -1) { printf("Login failed\n"); exit (1); }

    // exec shell
    char *args[] = { "/bin/sh", 0 }; execv (args[0], args);
}
```

15.1.4 The getuid() and getgid() Functions

Processes sometimes need to query the user and group IDs with which they are running. For this purpose they can use the following functions:

```c
word getuid ();
word geteuid ();
word getgid ();
word getegid ();
```
Since such functions are never needed in the kernel (where functions can simply look at the thread control block), we provide a simplified mechanism for quick access to these IDs:

```c
#define QUERY_UID 0
#define QUERY_EUID 1
#define QUERY_GID 2
#define QUERY_EGID 3
#define __NR_query_ids 524
```

Uses `__NR_query_ids`, `QUERY_EGID`, `QUERY_EUID`, `QUERY_GID`, `QUERY_UID`, and `syscall2`.

The system call handler in the kernel directly returns the queried value:

```c
void syscall_query_ids (context_t *r) {
    // ebx: type of ID
    switch (r->ebx) {
    case QUERY_UID:  eax_return (thread_table[current_task].uid);
    case QUERY_EUID: eax_return (thread_table[current_task].euid);
    case QUERY_GID:  eax_return (thread_table[current_task].gid);
    case QUERY_EGID: eax_return (thread_table[current_task].egid);
    default:         eax_return (-1);
    }
}
```

Defines: `syscall_query_ids`, used in chunk 587.

Uses `context_t 142a, current_task 192c, eax_return 174a, egid 573a, euid 573a, gid 573a, QUERY_EGID, QUERY_EUID, QUERY_GID, QUERY_UID, thread_table 176b, and uid 573a.

15.1.5 Changing Owner, Group and Permissions

All Unix systems provide system calls and library functions which allow users to change the owner, group and access permissions of any file or directory for which they have write access; Ulix is no different. We start with the kernel-internal functions:
Note how \texttt{u\_chown} accepts negative arguments for the owner and group: if one or both of them are $-1$, they are ignored. Thus,

- \texttt{u\_chown(file, 1000, 0)} will set file’s UID to 1000 and the GID to 0,
- \texttt{u\_chown(file, 500, -1)} will only set the UID to 500,
- \texttt{u\_chown(file, -1, 0)} ignores the UID argument and sets the GID to 0 and
- \texttt{u\_chown(file, -1, -1)} does nothing at all.

The implementation looks similar to other functions which take a path name as an argument; we start with converting the path into an absolute path and checking which device with which filesystem the file resides on. Then we call filesystem-specific functions. As ULIx only supports the Minix filesystem for disk drives and we do not want /dev entries to change, the only option is to call \texttt{mx\_chown}:

```
[588a] \langle function prototypes 45a\rangle + \equiv (44a) <580b 589b>
    int u\_chown(const char *path, short owner, short group);
    int u\_chmod(const char *path, word mode);
```

The implementation of \texttt{u\_chmod} looks almost identical to \texttt{u\_chown}, except that the number of arguments taken and passed to \texttt{mx\_chown} or \texttt{mx\_chmod} is different and regular users are allowed to change the access permissions (but not the ownership or owner group), so we don’t need the check for the root user:

```
[588b] \langle function implementations 100b\rangle + \equiv (44a) <582a 589a>
    int u\_chown(const char *path, short owner, short group) {
        char localpath[256], abspath[256];
        short device, fs;

        // only root may change file ownership / group
        if (scheduler_is_active && thread_table[current_task].euid != 0) return -1;

        // check relative/absolute path
        if (*path != '/') relpath_to_abspath(path, abspath);
        else strncpy(abspath, path, 256);
        get_dev_and_path(abspath, &device, &fs, (char*)&localpath);
        switch (fs) {
            case FS_MINIX: return mx\_chown(device, localpath, owner, group);
            case FS_FAT: return -1; // not possible (and FAT is not implemented)
            case FS_DEV: return -1; // not allowed
            case FS_ERROR: return -1; // error
            default: return -1;
        }
    }
```

Defines:
- \texttt{u\_chown}, used in chunks 588a and 590b.
Uses current_task 192c, euid 573a, FS\_DEV 410a, FS\_ERROR 410a, FS\_FAT 410a, FS\_MINIX 410a,
  get\_dev\_and\_path 408c, mx\_chown 589d, relpath\_to\_abspath 412b, scheduler\_is\_active 276e, strncpy 594b,
  and thread\_table 176b.
15.1 Users and Groups in Unix

\[\text{function implementations 100b}\] +

\[
\text{int u_chmod (const char *path, word mode) \{ \\
\text{char localpath[256], abspath[256];} \\
\text{short device, fs;} \\
\}
\]

// check relative/absolute path
if (*path != '/') relpath_to_abspath (path, abspath);
else       strncpy (abspath, path, 256);

if (scheduler_is_active) {
    \[\text{u_chmod: check permissions 591c}\] \ // see user/group chapter
}

get_dev_and_path (abspath, &device, &fs, (char*)&localpath);
switch (fs) {
    case FS_MINIX: return mx_chmod (device, localpath, mode);
    case FS_FAT: return -1; \ // not possible, no FAT implementation
    case FS_DEV: return -1; \ // not allowed
    case FS_ERROR: return -1; \ // error
    default: return -1;
}

Defines:
\text{u_chmod, used in chunk 590b.}

Uses \text{FS_DEV 410a, FS_ERROR 410a, FS_FAT 410a, FS_MINIX 410a, get_dev_and_path 408c, mx_chmod 589d, relpath_to_abspath 412b, scheduler_is_active 276e, and strncpy 594b.}

We only implement \text{chown, chgrp and chmod} for the Minix filesystem. Here are the corresponding \text{mx_} \ast \text{ functions}

\[\text{function prototypes 45a}\] +

\[
\text{int mx_chown (int device, const char *path, short owner, short group);} \\
\text{int mx_chmod (int device, const char *path, word mode);} \\
\]

which use a more general function

\[\text{function prototypes 45a}\] +

\[
\text{int mx_chinode (int device, const char *path, short owner,} \\
\text{short group, short mode);} \\
\]

that is able to change user ID, group ID and permissions in one step. It is called by both \text{mx_chown 589d} and \text{mx_chmod 589d} and will only modify the fields for which the provided values are \(\neq -1: \)

\[\text{function implementations 100b}\] +

\[
\text{int mx_chown (int device, const char *path, short owner, short group) \{ \\
\text{return mx_chinode (device, path, owner, group, -1); \ // change UID or GID, not mode} \\
\}} \\
\]

\[
\text{int mx_chmod (int device, const char *path, word mode) \{ \\
\text{return mx_chinode (device, path, -1, -1, mode); \ // change mode, not UID or GID} \\
\}} \\
\]
int mx_chinode (int device, const char *path, short owner,
    short group, short mode) {
    int ext_ino = mx_pathname_to_ino (device, path);
    if (ext_ino == -1) {
        printf ("file not found: %s\n", path);
        return -1;    // file not found
    }

    struct minix2_inode inode;
    mx_read_inode (device, ext_ino, &inode);
    if (owner != -1) inode.i_uid = owner;  // change owner (if != -1)
    if (group != -1) inode.i_gid = group;  // change group (if != -1)
    if (mode != -1)   // change mode (if != -1)
        inode.i_mode = (inode.i_mode & ~07777) | (mode & 07777);
    mx_write_inode (device, ext_ino, &inode);
    return 0;
}

We provide two system call handlers:

[590a] ⟨syscall prototypes 173b⟩+≡
    void syscall_chown (context_t *r);
    void syscall_chmod (context_t *r);

[590b] ⟨syscall functions 174b⟩+≡
    void syscall_chown (context_t *r) {
        // ebx: path, ecx: owner, edx: group
        eax_return ( u_chown ((char *)r->ebx, r->ecx, r->edx) );
    }

    void syscall_chmod (context_t *r) {
        // ebx: path, ecx: new mode
        eax_return ( u_chmod ((char *)r->ebx, r->ecx) );
    }

Defines:
    syscall_chmod, used in chunk 590c.
    syscall_chown, used in chunk 590.
Uses context_t 142a, eax_return 174a, u_chmod 589a, and u_chown 588b.

    (which we need to initialize)

[590c] ⟨initialize syscalls 173d⟩+≡
    install_syscall_handler (__NR_chown, syscall_chown);
    install_syscall_handler (__NR_chmod, syscall_chmod);
Uses __NR_chmod 204c, __NR_chown 204c, install_syscall_handler 201b, syscall_chmod 590b,
    and syscall_chown 590b.
and also two user mode library functions chown, and chmod (the chgrp program will use the chown function):

\[\text{<ulixlib function prototypes 174c> + } \sum \]
int chown (const char *path, short owner, short group);
int chmod (const char *path, short mode);

and

\[\text{<ulixlib function implementations 174d> + } \sum \]
int chown (const char *path, short owner, short group) {
    return syscall4 (__NR_chown, (unsigned int)path, owner, group);
}

int chmod (const char *path, short mode) {
    return syscall3 (__NR_chmod, (unsigned int)path, mode);
}

Defines:
chown, used in chunk 591a.
Uses __NR_chmod 204c, __NR_chown 204c, syscall3 203c, and syscall4 203b.

15.2 Exercises

38. The code chunk (u_execv: check permissions 580a) is empty: When executing a program, this version of the ULIx kernel does not check whether the user is allowed to run the program. In general, users can run programs if they can read them and also have an execute ...

Implement the empty code chunk and test your checks against some binaries which have or do not have appropriate access permissions.

39. The u_chmod function also needs to check whether it may change the access permissions. Fill the following code chunk:

\[\text{<u_chmod: check permissions 591c> } \equiv \]

// TO DO

and test that you can only change permissions of files for which you have write access.
Small Standard Library

Some standard functions which are normally included with an operating system must be provided by us; here’s a list of functions that are often used. Some of these functions will be part of both the kernel and the user mode library since features like formatting and printing a string are needed in both environments.

Looking at the implementations of these functions will add nothing new to your understanding of operating system concepts—that is why they have been moved to this late chapter. Following the literate programming concept that the document is the program, we decided to include them here even though they could have been moved to a separate code file. We will only provide few comments on these functions.

Some of these functions have not been implemented by us but were copied from online resources. In those cases, the first line after the function name lists the source.

### 16.1 Strings

Let’s start with some string functions which compare, copy and convert strings to numbers:

```c
size_t strlen (const char *str);
int  strcmp (const char *str1, const char *str2);
int  strncmp (const char *str1, const char *str2, uint n);
char *strncpy (char *dest, const char *src, size_t count);
char *strcpy (char *dest, const char *src);
int  atoi (char *s);
int  atoi8 (char *s);
```
The function `strlen` returns the length of a null-terminated string, `strcmp` and `strncmp` compare two strings and return 0 if they are equal and -1 or 1 if the first string is lexicographically smaller than the second one or vice versa. The `strncmp` variant stops comparing after `n` characters have been seen.

```c
size_t strlen (const char *str) {
    size_t retval;
    for (retval = 0; *str != '\0'; str++) retval++;
    return retval;
}

int strcmp (const char *s1, const char *s2) {
    while (*s1 != '\0' && *s1 == *s2) {
        s1++; s2++;
    }
    byte b1 = (*byte *) s1;
    byte b2 = (*byte *) s2;
    return ((b1 < b2) ? -1 : (b1 > b2));
}

int strncmp (const char *s1, const char *s2, uint n) {
    if (n == 0) { return 0; } // nothing to compare? return 0
    while (n-- > 0 && *s1 == *s2) {
        if (n == 0 || *s1 == '\0') { return 0; } // equality
        s1++; s2++;
    }
    byte b1 = (*byte *) s1;
    byte b2 = (*byte *) s2;
    return ((b1 < b2) ? -1 : (b1 > b2));
}
```

Defines:
- `strcmp`, used in chunk 596a.
- `strlen`, used in chunks 232a, 234b, 408, 409, 412b, 419a, 455a, 484b, 577c, 641d, and 642a.
- `strncmp`, used in chunks 229a, 562b, and 641e.

Uses `size_t 46b`.

The `strcmp` and `strncmp` functions copy a null-terminated string. While the first of the two will potentially go on copying forever if the source string is not terminated, the second function stops copying after `count` bytes. Note that if `strncpy` fills the whole target string (buffer), that string will not be null-terminated which can cause problems when correct termination is not checked and otherwise enforced.

```c
char *strcpy (char *dest, const char *src) {
    char *ret = dest;
    while ((*(dest++) = *(src++)) != '\0') ;
    return ret;
}
```
char *strncpy (char *dest, const char *src, size_t count) {
    // like memcpy (see next section), but copies only until first \0 character
    const char *sp = (const char *)src;
    char *dp = (char *)dest;
    for (; count != 0; count--) {
        *dp = *sp;
        if (*dp == 0) break;
        dp++; sp++;
    }
    return dest;
}

#define: strcpy, used in chunks 640 and 642b.
#define: strncpy, used in chunks 224c, 234b, 367b, 409c, 411e, 412b, 419, 431, 432e, 455a, 461d, 488a, 490d, 492, 500, 577c, 586a, 588b, 589a, 593, and 641.
Uses size_t 46b.

atoi converts a string into an integer value. It goes on reading until the first non-digit occurs, so it can also be used to convert strings like "123 KByte" to 123. It does not recognize negative values; for example, trying to convert the string "-1234" will lead to a result of 0 as the first character is found to be a non-digit.

```c
int atoi (char *s) {
    int res = 0;
    while ( ('0' ≤ *s) && (*s ≤ '9') ) {
        res = res*10 + (*s-'0');
        s++;
    }
    return res;
}
```

```c
int atoi8 (char *s) {  // same as atoi, but with octal numbers
    int res = 0;
    while ( ('0' ≤ *s) & (s ≤ '7') ) {
        res = res*8 + (*s-'0');
        s++;
    }
    return res;
}
```

Defines:
atoi, used in chunks 582a, 585a, and 586a.
atoi8 is not a standard function. It works like atoi but expects the string to contain an octal number instead of a decimal number.

We define two macros strequal (strings are equal) and strdiff (strings are different) which use strcmp:

```c
#define strequal (str1, str2) (!strcmp(str1, str2))
#define strdiff (str1, str2) (strcmp(str1, str2))
```
16 Small Standard Library

[596a] *(public macro definitions 596a)* \(\equiv\) 

\#define strequal(s1,s2) (!strcmp((s1),(s2)))
\#define strdiff(s1,s2) (strcmp((s1),(s2)))

 Defines:
 strequal, used in chunks 432e, 462a, 480c, 495c, 499d, 582a, 585a, 608b, 610a, and 631.
 Uses strcmp 594a.

 They are more intuitive to use than the standard (in-)equality comparisons via strcmp 594a. Wherever we need to compare strings in this book, we only use our strequal 596a and strdiff 596a functions. However, if you want to integrate other code in Unix or one of the user mode programs, the default strcmp 594a function is available.

16.2 Memory

The standard functions memcpy 596c, memset 596c and memsetw 596c compare two chunks of memory and fill a memory area with a byte or word constant:

[596b] *(public function prototypes 454b)* \(\equiv\) 

void *memcpy (void *dest, const void *src, size_t count);
void *memset (void *dest, char val, size_t count);
word *memsetw (word *dest, word val, size_t count);

[596c] *(public function implementations 455a)* \(\equiv\)

void *memcpy (void *dest, const void *src, size_t count) {
    const char *sp = (const char *)src;
    char *dp = (char *)dest;
    for (; count != 0; count--)
        *dp++ = *sp++;
    return dest;
}

void *memset (void *dest, char val, size_t count) {
    char *temp = (char *)dest;
    for ( ; count != 0; count--) *temp++ = val;
    return dest;
}

word *memsetw (word *dest, word val, size_t count) {
    word *temp = (word *)dest;
    for ( ; count != 0; count--) *temp++ = val;
    return dest;
}

 Defines:
 memcpy, used in chunks 190a, 209b, 223e, 232a, 327a, 332b, 334a, 449, 451a, 453b, 455a, 456, 468b, 471c, 475c, 487a, 496d, 497, 509d, 510b, 518b, 519d, 521a, 530, 549c, 550b, and 597a.
 memset, used in chunks 100c, 103b, 112, 121c, 122a, 164, 166a, 197a, 211a, 232c, 255c, 257c, 480c, 487a, and 509b.
 memsetw, used in chunks 326c, 329b, 333e, 334a, and 609.
 Uses size_t 46b.
Only in the kernel we provide the

```
#define memcpy_debug(dest, src, count) \ 
    debug_printf("DEBUG: memcpy() called in line %d\n", __LINE__); \ 
    memcpy (dest, src, count);
```

macro which creates a debug message and calls `memcpy`.

## 16.3 Formatted Output

We use a small implementation of the standard functions `printf` and `sprintf` that is dual-licensed under the LGPL and the BSD license and available from [http://www.menie.org/georges/embedded/](http://www.menie.org/georges/embedded/)—there is no need to reinvent the wheel. We modified the code so that `printf` can also handle the ‘o’ format character for octal numbers and we changed the code indentation to match the style of the other code in this book. There were also some minor modifications that enable the functions to use the ULIx output functions. (Thankfully the function already knew how to print numbers to any base; we just copied the code for ‘x’ and changed the base 16 to 8.)

The code was also changed a bit to make it shorter, and we have turned the leading comments into normal text to make them better readable.

```c
/* Copyright 2001, 2002 Georges Menie (http://www.menie.org) */

This program is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA */

/* `putchar` is the only external dependency for this file, if you have a working `putchar`, just remove the following define. If the function should be called something else, replace `outbyte(c)` by your own function call. */

## 16.3.1 printf in the Kernel

The kernel uses the functions

```
int printf (const char *format, ...);
int sprintf (char *out, const char *format, ...);
```
and we have to define `putchar` as `kputch` so that `printf` will call the right character output function.

```
#define putchar(c) kputch(c)
```

```c
static void printchar(char **str, int c) {
    if ((int)*str == -1) {
        // debug output: goes to qemu serial console via uartputc
        if (c == 0x100) { // backspace
            uartputc('\b'); uartputc(' '); uartputc('\b');
        } else {
            uartputc(c);
        }
    } else if (str) {
        **str = c; ++(*str);
    } else {
        (void) putchar(c);
    }
}
```

```
(public printf implementation 599a)
```

Defines:
- `printchar`, used in chunks 599 and 600.
- `putchar`, used in chunk 556.

Uses `kputch` and `uartputc`.

The `printchar` function uses `uartputc` to write output (only) to the serial port for debugging purposes if `str` is `null`. `kputch` always writes to the serial port, too.

### 16.3.2 printf in the User Mode Library

For the library, we use the same `printf` code and only need to modify the implementation of `putchar` and `printchar`. That is why we can provide most of the code via the `printf implementation` chunk. Here we also define the `ulixlib_printchar` function for printing single characters that writes to the standard output descriptor (1).

```
int ulixlib_printchar(byte c);  
```

```
#define putchar(c) ulixlib_printchar(c)
```

```c
static void printchar(char **str, int c) {
    if (str) {
        **str = c; ++(*str);
    } else {
        (void) putchar(c);
    }
}
```

```
(public printf implementation 599a)
```
16.3 Formatted Output

Defines:
- `printchar`, used in chunks 599 and 600.
- `putchar`, used in chunk 556.
- `ulinlib_printchar`, used in chunk 598b.
Uses `STDOUT_FILENO` 415b and `write` 429b.

16.3.3 The Generic Implementation

### ⟨public printf implementation 599a⟩ ≡

```c
#define PAD_RIGHT 1
#define PAD_ZERO 2

static int prints (char **out, const char *string, int width, int pad) {
    register int pc = 0, padchar = ' ';  
    if (width > 0) {
        register int len = 0; register const char *ptr;
        for (ptr = string; *ptr; ++ptr) ++len;
        if (len ≥ width) width = 0;
        else width -= len;
        if (pad & PAD_ZERO) padchar = '0';
    }
    if (!((pad & PAD_RIGHT))) {
        for (; width > 0; --width) { printchar (out, padchar); ++pc; }
    }
    for (; *string ; ++string) { printchar (out, *string); ++pc; }
    for (; width > 0; --width) { printchar (out, padchar); ++pc; }
    return pc;
}
```

Defines:
- `PAD_RIGHT`, used in chunk 600.
- `PAD_ZERO`, used in chunks 599b and 600.
- `prints`, used in chunks 599b and 600.
Uses `printf` 598c 598d.

The `printf` functions deals with 32-bit integers whose textual representation fits inside a 34-byte buffer:

### ⟨public printf implementation 599a⟩ +≡

```c
#define PRINT_BUF_LEN 34

static int printi (char **out, int i, int b, int sg, int width, int pad, int letbase) {
    char print_buf[PRINT_BUF_LEN];
    register char *s; register int t, neg = 0, pc = 0; register unsigned int u = i;
    if (i == 0) {
        print_buf[0] = '0'; print_buf[1] = '\0';
        return prints (out, print_buf, width, pad);
    }
    if (sg & b == 10 & & i < 0) { neg = 1; u = -i; }
    s = print_buf + PRINT_BUF_LEN-1; *s = '\0';
    while (u) {
        t = u % b; if (t ≥ 10) t += letbase - '0' - 10;
```

---
*--s = t + '0'; u /= b;

if (neg) {
    if (width & (pad & PAD_ZERO)) {
        printchar (out, '-'); ++pc; --width;
    } else {
        *--s = '-';
    }
}

return pc + prints (out, s, width, pad);

Defines:
printi, used in chunk 600.
Uses PAD_ZERO 599a, printchar 598a 598c, and prints 599a.

We have modified the print function so that it recognizes the %o (octal) and %b (binary) format words and prints numbers with base 8 or base 2, respectively.

\[\text{(public printf implementation 599a) + }\]

\[
\text{static int print (char **out, int *varg) \{ }
\text{register int width, pad; register int pc = 0;}
\text{register char *format = (char *)(*varg++); register char *s; char scr[2];}
\text{for (; *format != 0; ++format) }
\text{if (*format == '%') }
\text{++format; width = pad = 0;}
\text{if (*format == '\0') break;}
\text{if (*format == '-') goto outlabel;}
\text{if (*format == '-') \{ ++format; pad = PAD_RIGHT; \}}
\text{while (*format == '\0') \{ ++format; pad |= PAD_ZERO; \}}
\text{for (; *format >= '0' && *format <= '9'; ++format) }
\text{width *= 10; width += *format - '0';}
\text{switch (*format) }
\text{case 's': s = *((char **varg++);}
\text{pc += prints (out, s ? s: "(null)", width, pad); continue;}
\text{case 'd': pc += printi (out, *varg++, 10, 1, width, pad, 'a'); continue;}
\text{case 'o': pc += printi (out, *varg++, 8, 0, width, pad, 'a'); continue;}
\text{case 'b': pc += printi (out, *varg++, 2, 0, width, pad, 'a'); continue;}
\text{case 'x': pc += printi (out, *varg++, 16, 0, width, pad, 'a'); continue;}
\text{case 'X': pc += printi (out, *varg++, 16, 0, width, pad, 'A'); continue;}
\text{case 'u': pc += printi (out, *varg++, 10, 0, width, pad, 'a'); continue;}
\text{case 'c': // char are converted to int then pushed on the stack}
\text{scr[0] = *varg++; scr[1] = '\0';}
\text{pc += prints (out, scr, width, pad); continue;}
\} else { }
\text{outlabel:}
\text{printchar (out, *format); ++pc; }
\}

if ( (int)out != -1 & out) **out = '\0';
return pc; \}
Defines:
  print, used in chunks 431, 601, and 626b.
Uses PAD_RIGHT 599a, PAD_ZERO 599a, printchar 598a 598c, printi 599b, and prints 599a.

\[ \langle \text{public printf implementation 599a} \rangle \equiv \]
\[
\begin{align*}
\text{int printf (const char *format, ...)} & \{ \\
& \quad \text{register int *varg = (int *)(&format); return print (0, varg);}
\}
\end{align*}
\]

\[
\begin{align*}
\text{int snprintf (char *out, const char *format, ...)} & \{ \\
& \quad \text{register int *varg = (int *)(&format); return print (&out, varg);}
\}
\end{align*}
\]

Defines:
  printf, used in chunks 45d, 151c, 152a, 164, 165a, 168d, 170e, 191a, 201d, 214, 239, 291, 293b, 297, 299e, 308c, 311b, 321a, 324a, 326c, 337c, 340b, 349, 406, 416d, 417, 431, 450a, 456, 471c, 476b, 480, 488a, 513e, 532d, 534b, 536b, 537a, 539c, 547d, 551, 552c, 562b, 564, 585b, 589d, 603–8, 610–14, 626a, and 639c.
  sprintf, used in chunks 280a, 342b, 343b, 369c, 597b, and 608b.
Uses print 600.

The debug_printf function

\[ \langle \text{function prototypes 45a} \rangle \equiv \]
\[
\begin{align*}
\text{int debug_printf (const char *format, ...)} & \;
\end{align*}
\]

exists only in the kernel, it is similar to printf if the DEBUG macro is set but passes the target argument −1 instead of 0 so that output will go only to the serial port (and not to the ULIx screen). If DEBUG is not set, it does nothing.

Since writing many messages makes the system a bit slower, debug output is disabled by default:

\[ \langle \text{macro definitions 35a} \rangle \equiv \]
\[
// \#define DEBUG
\]

\[ \langle \text{function implementations 100b} \rangle \equiv \]
\[
\begin{align*}
\text{ifdef DEBUG} & \{ \\
& \quad \text{int debug_printf (const char *format, ...) } \{ \\
& \quad \quad \text{register int *varg = (int *)(&format); return print ((char**)−1, varg);}
& \quad \}
& \}
\end{align*}
\]

\[ \text{#else} \]
\[
\text{inline int debug_printf (const char *format, ...) } \{ \text{return 0; } \} // \text{do nothing}
\]
\[ \text{#endif} \]

Defines:
  debug_printf, used in chunks 277b, 366c, 597a, and 601b.
Uses print 600.
17.1 The Kernel Mode Shell

In cases when user mode does not work or when we need to look at kernel structures that are not accessible from user mode, we can launch a simple kernel shell kernel_shell that provides a few commands which are helpful for debugging.

This code is even less interesting than the one in the previous chapter unless you want to modify Unix and run into problems that require some debugging.

For most internal commands CMD of the kernel shell we provide corresponding functions named ksh_command_CMD which will be executed.

You can enter the kernel shell by pressing [Shift-Escape], and you can leave it with exit which brings you back to user mode. When the system detects a fault from which it cannot recover it will also drop you in the kernel mode shell, but in that case returning to user mode is not possible.

The test command displays addresses of the current page directory and page table and a hexdump of the frame table.

```c
void ksh_command_test () {
    kputs("current_pd as INT: ");
    printbitsandhex(*(uint*)(current_pd)); kputs("\n");
    kputs("current_pd->ptds[0].frame_addr.: ");
    printbitsandhex(current_pd->ptds[0].frame_addr<<12); kputs("\n");
    kputs("current_pt as INT: ");
    printbitsandhex(*(uint*)(current_pt)); kputs("\n");
    kputs("address of current_pd: "); printf("%08x\n",(uint)current_pd);
    kputs("address of current_pt: "); printf("%08x\n",(uint)current_pt);
```

kputs("size of current_pd: ");
kprintf("%08x\n", sizeof(*current_pd));
kputs("size of current_pt: ");
printf("%08x\n", sizeof(*current_pt));
kputs("address of frame table: ");
printf("%08x\n", (uint)ftable);
kputs("hexdump ftable\n");
hexdump((uint&)place_for_ftable, ((uint&)place_for_ftable) + 1);

};

Defines:

ksh_command_test, used in chunk 608b.

Uses current_pd 105a, current_pt 105a, ftable 112c, hexdump 612c, kputs 335b, place_for_ftable 112c,

printbitsandhex 612a, and printf 601a.

mem displays similar data, but for specific page table entries.

[604a] 〈global variables 92b〉+≡ (44a) <547b

extern memaddress stack_first_address, stack_last_address;

Uses memaddress 46c, stack_first_address 95a, and stack_last_address 95a.

[604b] 〈function implementations 100b〉+≡ (44a) <603 605a>

void ksh_command_mem () {
    kputs("kernel_pd as INT: ");
    printbitsandhex(*(int*)(&kernel_pd)); kputs("\n");
    kputs("kernel_pd.ptds[0].frame_addr: ");
    printbitsandhex(kernel_pd.ptds[0].frame_addr<<12); kputs("\n");
    kputs("kernel_pd.ptds[768].frame_addr: ");
    printbitsandhex(kernel_pd.ptds[768].frame_addr<<12); kputs("\n");
    kputs("kernel_pd.ptds[831].frame_addr: ");
    printbitsandhex(kernel_pd.ptds[831].frame_addr<<12); kputs("\n");
    kputs("kernel_pd.ptds[832].frame_addr: ");
    printbitsandhex(kernel_pd.ptds[832].frame_addr<<12); kputs("\n");
    kputs("kernel_pd.ptds[833].frame_addr: ");
    printbitsandhex(kernel_pd.ptds[833].frame_addr<<12); kputs("\n");
    kputs("kernel_pt as INT: ");
    printbitsandhex(*(int*)(&kernel_pt)); kputs("\n");
    kputs("address of kernel_pd: ");
    printf("%08x\n", (uint)&kernel_pd);
    kputs("address of kernel_pt: ");
    printf("%08x\n", (uint)&kernel_pt);
    kputs("stack_first_address: ");
    printf("%08x\n", (uint)&stack_first_address);
    kputs("stack_last_address: ");
    printf("%08x\n", (uint)&stack_last_address);
    kputs("free_frames: "); printf("%d\n", free_frames);
    uint esp; asm volatile("mov %esp, %0: =r\n(esp));
    kputs("ESP: "); printf("%08x\n", esp);
}

Defines:

ksh_command_mem, used in chunk 608b.

Uses free_frames 112b, kernel_pd 105a, kernel_pt 105a, kputs 335b, printbitsandhex 612a, printf 601a,

stack_first_address 95a, and stack_last_address 95a.
time and uname show the current time and the ULIx version string.

```c
void ksh_command_time () {
    short int hour, min, sec;
    hour = (system_time/60)/60 % 24; min = (system_time/60) % 60; sec = system_time % 60;
    printf("The time is %02d:%02d:%02d.
", hour, min, sec);
}
```

```c
void ksh_command_uname () { printf("%s; Build: %s 
", UNAME, BUILDDATE); }
```

**Defines:**
- `ksh_command_time`, used in chunk 608b.
- `ksh_command_uname`, used in chunk 608b.

**Uses:**
- `BUILDDATE`, `hour`, `min`, `printf`, `sec`, `system_time`.

`div0` causes a division by zero fault.

```c
void ksh_command_div0 () {
    int zero = 0; int i = 10 / zero; kputch (i);  // Test for exception
}
```

**Defines:**
- `ksh_command_div0`, used in chunk 608b.

**Uses:**
- `kputch`.

`hexdump` prints a hex dump of the specified address range. Since kernel shell commands do not take parameters, the addresses have to be changed in the source code in order to show a different range.

```c
void ksh_command_hexdump () {
    int as = current_as;
    activate_address_space (10);
    hexdump (0xaффffdf8, 0xaффffdf8 + 128);  // modify this to show other regions
    activate_address_space (as);
}
```

**Defines:**
- `ksh_command_hexdump`, used in chunk 608b.

**Uses:**
- `activate_address_space`, `current_as`, and `hexdump`.

`ps` shows the process list.

```c
void ksh_command_ps () {
    int i;
    printf (" TID PID PPID ESP EIP EBP ESP\0 AS State "
            "Exi Cmdline\n");
    for (i=0; i<MAX_THREADS; i++) {
        if (thread_table[i].used) {
            printf("%4d %4d %4d %08x %08x %08x %08x %2d %5s %3d %s\n",
                    thread_table[i].tid,
                    thread_table[i].pid,
                    thread_table[i].ppid,
                    thread_table[i].regs.esp,
                    thread_table[i].regs.eip,
                    thread_table[i].regs.ebp,
                    thread_table[i].regs.ebp,
               );
        }
    }
}
```
thread_table[i].esp0,        thread_table[i].addr_space,
  state_names[thread_table[i].state], thread_table[i].exitcode,
  thread_table[i].cmdline);
}
)
}
}

void ksh_print_queue (char *name, blocked_queue *bq) {
  printf ("%s: ", name);
  int pid = bq->next;
  while (pid != 0) {
    printf ("%d, ", pid);
    pid = thread_table[pid].next;
  }
  printf ("\n");
}

void ksh_command_queues () {
  printf ("Queues: \n");
  printf ("ready: ");
  int pid = 0;
  while ((pid = thread_table[pid].next) != 0) printf ("%d, ", pid);
  printf ("\n");
  ksh_print_queue ("keyboard", &keyboard_queue);
  ksh_print_queue ("harddisk", &harddisk_queue);
  ksh_print_queue ("floppy",  &floppy_queue);
  ksh_print_queue ("waitpid",  &waitpid_queue);
  ksh_print_queue ("buffer",   &buffer_lock->bq));
}

void ksh_command_locks () {
  for (int i = 1; i < MAX_LOCKS; i++) {
    if (kernel_locks[i].used) {
      ksh_print_queue (kernel_locks[i].lockname, &kernel_locks[i].bq);
    }
  }
}

Defines:
  ksh_command_ps, used in chunk 608b.
Uses MAX_THREADS 176a, printf 601a, state_names 180b, and thread_table 176b.

  queues shows all blocked queues and the processes or threads on those queues (as well as
the ready queue). The ksh_command_queues function uses the ksh_print_queue helper
function which displays a single queue. Similarly, locks shows all processes or threads
waiting for a lock.

[606] (function implementations 100b) +

(44a) <605d 607a>

void ksh_print_queue (char *name, blocked_queue *bq) {
  printf ("%s: ", name);
  int pid = bq->next;
  while (pid != 0) {
    printf ("%d, ", pid);
    pid = thread_table[pid].next;
  }
  printf ("\n");
}

void ksh_command_queues () {
  printf ("Queues: \n");
  printf ("ready: ");
  int pid = 0;
  while ((pid = thread_table[pid].next) != 0) printf ("%d, ", pid);
  printf ("\n");
  ksh_print_queue ("keyboard", &keyboard_queue);
  ksh_print_queue ("harddisk", &harddisk_queue);
  ksh_print_queue ("floppy",  &floppy_queue);
  ksh_print_queue ("waitpid",  &waitpid_queue);
  ksh_print_queue ("buffer",   &buffer_lock->bq));
}

void ksh_command_locks () {
  for (int i = 1; i < MAX_LOCKS; i++) {
    if (kernel_locks[i].used) {
      ksh_print_queue (kernel_locks[i].lockname, &kernel_locks[i].bq);
    }
  }
}

Defines:
  ksh_command_locks, used in chunk 608b.
  ksh_command_queues, used in chunk 608b.
Uses blocked_queue 183a, buffer_lock 509a, floppy_queue 544d, harddisk_queue 529a, kernel_locks 365c,
keyboard_queue 323d, MAX_LOCKS 365b, printf 601a, thread_table 176b, waitpid 220d, and waitpid_queue 218b.
inode displays the hex dump of an inode. The device and number must be set directly in the source code.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define DEV_HDA 0
#define MX_MAX_FILES 1024
#define MX_MINIMUM_PAGESIZE 1024
#define MX_MINIMUM_FRAME_SIZE 4096
#define MX_MINIMUM_PAGE_SIZE 4096

void ksh_command_inode ()
{
    struct minix2_inode in;
    int dev = DEV_HDA;
    int ino = 79;
    int res = mx_read_inode(dev, ino, &in);
    printf("mx_read_inode(%d, %d) returns %d\n", dev, ino, res);
    if (res != 0) {
        hexdump((uint)&in, (uint)&in+sizeof(struct minix2_inode));
        printf("size: %d, blocks: ", in.i_size);
        for (int i = 0; i < 10; i++) printf("%d, ", in.i_zone[i]);
        printf("\n");
    }
}
```

Defines: ksh_command_inode, used in chunk 608b.
Uses DEV_HDA 508a, hexdump 612c, minix2_inode 442a, mx_read_inode 451b, and printf 601a.

lsf displays the list of open Minix files.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define INT_MINIX2_INODE 459a
#define MX_MAX_FILES 461a
#define MX_STATUS 461b
#define NULL 46a
#define printf 601a

void ksh_command_lsf ()
{
    for (int i = 0; i < MX_MAX_FILES; i++) {
        struct int_minix2_inode *inode = mx_status[i].int_inode;
        if (inode != NULL) {
            printf("mfd=%d inode-addr=%08x size=%d\n", i, (unsigned int)inode, inode->i_size);
        }
    }
}
```

Defines: ksh_command_lsf, used in chunk 608b.
Uses int_minix2_inode 459a, MX_MAX_FILES 461a, mx_status 461b, NULL 46a, and printf 601a.

longhelp explains (some of) the commands in the kernel shell.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define EX "ex" "it return to user mode\n"
#define TE "te" "st\n"
#define PF "pfault, div0 test faults\n"
#define MM "mem show memory (frames, pages) info\n"
#define ST "st" "at\n"
#define UN "uname show Ulix version\n"
#define HEX "hex" "dump show hex dump of some memory area\n"
#define CL "clear clear the screen\n"
#define GF "gf, g" "p, gp1k get a frame, a page, 1000 pages\n"
#define RP "rp release page\n"
#define BD "bdump\n"
#define ML "malloc test kernel malloc\n"

void ksh_command_longhelp ()
{
    printf("ex" "it return to user mode\n"
"te" "st\n"
"pfault, div0 test faults\n"
"mem show memory (frames, pages) info\n"
"st" "at\n"
"uname show Ulix version\n"
"hex" "dump show hex dump of some memory area\n"
"clear clear the screen\n"
"gf, g" "p, gp1k get a frame, a page, 1000 pages\n"
"rp release page\n"
"bdump\n"
"malloc test kernel malloc\n"
```
"time" show time
"cloneas <n>" clone address space (argument: size)
"listas" show address spaces
"ps" process list
"ps" process list
"disable" disable scheduler
"enable" (re-)enable scheduler

); }

As previously mentioned, ksh_command_longhelp, used in chunk 608b.
Uses faults 148a, g, and printf 601a.

The ksh_run_command function tests whether the command that was entered is known.
If so, it executes one of the ksh_command_* functions (or directly performs some action).

[608a] (constants 112a)+≡ (44a) <579b

#define SHELL_COMMANDS "help, ex" "it, test, div0, mem, stat, uname, "\n"hexdump, clear, gf, gp, rp, gp1k, bdump, malloc, time, listas, "\n"init, exec, testdisk, enable, longhelp, ps, queues, lsof"

Defines:
SHELL_COMMANDS, used in chunks 608b and 610a.
Uses gp 92b, hexdump 612c, and stat 429b 489b.

[608b] (function implementations 100b)+≡ (44a) <607c 609>

void ksh_run_command (char *s) {
  if (strequal(s, "help") ) {
    printf ("Commands: %s \n", SHELL_COMMANDS);
  }
  else if (strequal(s, "uname") ) {
    ksh_command_uname();
  }
  else if (strequal(s, "test") ) {
    ksh_command_test();
  }
  else if (strequal(s, "div0") ) {
    ksh_command_div0();
  }
  else if (strequal(s, "hexdump") ) {
    ksh_command_hexdump();
  }
  else if (strequal(s, "clear") ) {
    vt_clrscr();
  }
  else if (strequal(s, "mem") ) {
    ksh_print_page_table();
  }
  else if (strequal(s, "mem2") ) {
    ksh_command_mem2();
  }
  else if (strequal(s, "ps") ) {
    ksh_command_ps();
  }
  else if (strequal(s, "queues") ) {
    ksh_command_queues();
  }
  else if (strequal(s, "locks") ) {
    ksh_command_locks();
  }
  else if (strequal(s, "longhelp") ) {
    ksh_command_longhelp();
  }
  else if (strequal(s, "enable") ) {
    enable scheduler 276a
  }
  else if (strequal(s, "disable") ) {
    disable scheduler 276b
  }
  else if (strequal(s, "stat") ) {
    list_address_spaces();
  }
  else if (strequal(s, "time") ) {
    ksh_command_time();
  }
  else if (strequal(s, "lsof") ) {
    ksh_command_lsof();
  }
  else if (strequal(s, "inode") ) {
    ksh_command_inode();
  }
  else if (strequal(s, "gf") ) {
    uint newframeid = request_new_frame();
    printf ("New frame ID: %d\n", newframeid);
  }
  else if (strequal(s, "gp") ) {
    /* uint* page = */ request_new_page();
    // kputs (", Page @ "); printf ("%08x\n", (uint)page);
  }
  else if (strequal(s, "rp") ) {
    // ksh_command_rp();
  }
  else { /* The command is not recognized */
    printf ("Command not recognized\n");
  }
}
printf ("releasing page range 0xc03fe..0xc07e6 \n");  
release_page_range (0xc03fe,0xc07e6);
} else if ( strequal (s, "gp1k") ) {
    char buf[20]; uint *page;
    for (int i = 0; i < 1024; i++) {
        sprintf ((char*)&buf, "Create: %d ", i); set_statusline ((char*)&buf);
        page = request_new_page ();
    }
} else if ( strequal (s, "gp10k") ) {
    char buf[20]; uint *page;
    for (int i = 0; i < 10; i++) {
        sprintf ((char*)&buf, "Create: %d ", i); set_statusline ((char*)&buf);
        page = request_new_pages (1024);
    }
} else if ( strequal (s, "") ) { return; } // no command
else { printf ("Error: >&s<- no such command\n", s); }
}

Defines:
    ksh_run_command, used in chunk 610a.
Uses gp 92b, hexdump 612c, kputs 335b, ksh_command_div0 605b, ksh_command_hexdump 605c, ksh_command_inode 607a,  
ksh_command_locks 606, ksh_command_longhelp 607c, ksh_command_lsof 607b, ksh_command_mem 604b, ksh_command_ps 605d, ksh_command_queues 606, ksh_command_test 603, ksh_command_time 605a, ksh_command_uname 605a, ksh_print_page_table 613b, list_address_spaces 171a, printf 601a, release_page_range 123d, request_new_frame 118b, request_new_page 120a, request_new_pages 120b, set_statusline 337b, SHELL_COMMANDS 608a, sprintf 601a, strequal 596a, and vt_clscr 329b.

The two statusline_* functions change the color of the status line so that it is always obvious whether you are using a regular shell (blue background) or the kernel shell (red).

\begin{verbatim}
(function implementations 100b) +==
void statusline_red () {
    // make status line red
    memsetw (textmemptr + 24 * 80, 0x20 | VT_RED_BACKGROUND, 80);
}

void statusline_blue () {
    // make status line blue
    memsetw (textmemptr + 24 * 80, 0x20 | VT_BLUE_BACKGROUND, 80);
    set_statusline (UNAME);
}
\end{verbatim}

Defines:
    statusline_blue, used in chunk 610a.
    statusline_red, used in chunk 610a.
Uses memsetw 596c, set_statusline 337b, textmemptr 116c, UNAME 35a, VT_BLUE_BACKGROUND 326b, and VT_RED_BACKGROUND 326b.

Finally, this is the kernel shell. It reads in a command and calls ksh_run_command.
### 17.2 A System Call That Displays an Inode

For testing the Minix filesystem implementation we provide a system call that reads a Minix inode from the disk and displays it. It also shows the first seven entries of the i_zone[] array (which contain the direct block numbers).

```c
void syscall_print_inode (context_t *r)
{
    int ino = r->ebx; // requested inode
    printf ("syscall; ino = %d\n", ino);
}
```

Uses context_t 142a and syscall_print_inode.
struct minix2_inode in;
mx_read_inode (DEV_HDA, ino, &in);
printf("i_mode: 0%o\n", in.i_mode);
printf("i_nlinks: %d\n", in.i_nlinks);
printf("i_size: %d\n", in.i_size);
printf("i_zone: [" ];
  for (int i = 0; i < 7; i++) printf("%d, ", in.i_zone[i]); printf("]\n");
}

Uses context_t 142a, DEV_HDA 508a, minix2_inode 442a, mx_read_inode 451b, printf 601a,
and syscall_print_inode.

⟨initialize syscalls 173d⟩+≡ (44b) +590c [611a]
install_syscall_handler (777, syscall_print_inode);
Uses install_syscall_handler 201b and syscall_print_inode.

The system call should not be used for regular programs, instead the stat function
is intended to return information about a file.

## 17.3 Printing the Page Directory

The following function prints parts of the page directory.

⟨function implementations 100b⟩+≡ (44a) <610a 612a> [611b]
void print_page_directory () {
  int i;
  kputs ("The Page Directory:\n");
  for (i = 700 ; i<800 ; i++) {
    if (current_pd->ptds[i].present ) {
      printf("%04d ", i);
      printf("%08x\n", current_pd->ptds[i].frame_addr);
    }
  }
}

unsigned int z=(unsigned int)current_pd;
printf("hexdump for %08x\n", z);
hexdump (z,z+128);
kputch (\n');
}

Uses current_pd 105a, hexdump 612c, kputch 335b, kputs 335b, and printf 601a.

## 17.4 Helper Functions for Printing

Some functions that belong to the kernel mode shell use the following helper functions to
print number in binary and hexadecimal format and to print a hex dump.

⟨function prototypes 45a⟩+≡ (44a) <610b 612b> [611c]
void printbitsandhex (uint i);
17.5 Printing the Frame Table and Page Table

Here's a function for displaying the current page tables.

Since we want to output information from the free frame list (we will use `test_frame` to check frame states), we write a simple function that can print status information for a memory region (going from `start` to `end`):

```c
void ksh_print_page_table_helper (unsigned sta, unsigned end, unsigned used) {
    if (used) { kputs ("Used: "); }
    else { kputs ("Free: "); }
    printf ("%05x-%05x %5d-%5d (%5d frames)\n", sta, end, sta, end, end-sta+1);
}
```

Defines:
- `ksh_print_page_table_helper`, used in chunk 613c.
Uses `kputs` 335b and `printf` 601a.
The following function prints the frame and page tables:

\[
\begin{align*}
\text{(function prototypes 45a)} &\equiv \\
\text{void ksh_print_page_table ();}
\end{align*}
\]

\[
\begin{align*}
\text{(function implementations 100b)} &\equiv \\
\text{void ksh_print_page_table ()} \\
&\quad \{ \\
&\quad \quad \text{unsigned int cr3;} \\
&\quad \quad \text{(print frame table 613c)} \\
&\quad \quad \text{kputch ('\n');} \\
&\quad \quad \text{(print page table 614a)} \\
&\quad \quad \quad \text{\_asm\_ volatile\_('mov %cr3, %0': "=r\"(cr3));} \\
&\quad \quad \text{printf ("cr3: %08x\n", cr3);} \\
&\quad \}
\end{align*}
\]

Defines:

ksh_print_page_table, used in chunks 608b and 613a.
Uses kputch 335b and printf 601a.

\[
\begin{align*}
\text{(print frame table 613c)} &\equiv \\
&\quad \text{kputs ("Current Frame Info:\n");} \\
&\quad \text{unsigned int frameno = 0;} \\
&\quad \text{unsigned int totalfree = NUMBER_OF_FRAMES; // total number of free frames} \\
&\quad \text{unsigned int test = test_frame (frameno); // check first frame} \\
&\quad \text{for (unsigned int i = 1; i < NUMBER_OF_FRAMES; i++)} \\
&\quad \quad \text{if (test_frame (i) != test)} \\
&\quad \quad \quad \text{ksh_print_page_table_helper (frameno, i-1, test);} \\
&\quad \quad \quad \text{if (test) totalfree -= (i-frameno);} \\
&\quad \quad \quad \text{test = 1-test;} \\
&\quad \quad \quad \text{frameno = i;} \\
&\quad \}; \\
&\quad \text{ksh_print_page_table_helper (frameno, NUMBER_OF_FRAMES-1, test);} \\
&\quad \text{if (test) totalfree -= (NUMBER_OF_FRAMES-frameno);} \\
&\quad \text{printf ("Total free frames: %6d\n", totalfree);} \\
&\quad \text{printf ("Value of free_frames: %6d\n", free_frames);} \\
\end{align*}
\]

Uses free_frames 112b, kputs 335b, ksh_print_page_table_helper 612d, NUMBER_OF_FRAMES 112a, printf 601a, test_frame 114a, and totalfree.

The output of (print frame table 613c) looks like this:

Current Frame Info:

<table>
<thead>
<tr>
<th>Used: 0x00000000-0x000003FF</th>
<th>0000000-0001023</th>
<th>(001024 frames)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Free: 0x00000400-0x000007FE</td>
<td>0001024-0002046</td>
<td>(0001023 frames)</td>
</tr>
<tr>
<td>Used: 0x000007FF-0x00004000</td>
<td>0002047-0002047</td>
<td>(0000001 frames)</td>
</tr>
<tr>
<td>Free: 0x00000800-0x00000400</td>
<td>0002048-0016384</td>
<td>(0014337 frames)</td>
</tr>
<tr>
<td>Total free frames: 0015359</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following code for the page table seems overly complicated because we want to print mappings of ranges and not each single mapping of a page to a frame in order to
save space in the output (and keep it readable). We use a variable started to memorize whether we're right now in a mapped region while skipping through the page tables.

\[
\begin{align*}
\text{boolean started=false; } & \\
\text{int save_i=0; int save_f=0; } & \\
\text{unsigned int start_i=0; unsigned int start_f=0; } & \\
\text{for (unsigned int i = 0; i < 1024*1024; i++) { } } & \\
\text{frameno = mmu_p (current_as, i); } & \\
\text{if (frameno == -1) { } } & \\
\text{if (started) { } } & \\
\text{if (!started) { } } & \\
\text{else { } } & \\
\text{if (i-start_i != frameno-start_f) { } } & \\
\text{save_i = i; save_f = frameno; } & \\
\text{save_i = i; save_f = frameno; } & \\
\end{align*}
\]

Uses current_as, mmu_p, and printf.

This is just the code for formatting the output:

\[
\begin{align*}
\text{printf ("PTEs 0x%05x..0x%05x -> frames 0x%05x..0x%05x (%5d pages)\n", } & \\
\text{start_i, save_i, start_f, save_f, save_i-start_i+1); } & \\
\end{align*}
\]

The output of \langle print page table \rangle will look like this:

Current Paging Info:
PTEs 0x00000000..0x000003FF -> frames 0x00000000..0x000003FF (0001024 pages)
PTEs 0x000C0000..0x000C03FF -> frames 0x00000000..0x000003FF (0001024 pages)
PTEs 0x000D0000..0x000D3FFF -> frames 0x00000000..0x00003FFF (0016384 pages)
18

The ULIX Build Process

You have almost reached the end of the book—now we describe the whole process that is needed in order to turn a literate program (the ulix-book.nw file) into a booting operating system disk image and some other files needed for execution of the system.

18.1 Required Software

If you want to build ULIX yourself, you need several tools which might not be installed on your machine. Check that the following requirements are fulfilled:

- **Linux operating system**
  Any 32-bit version of Linux will work, provided that you can install the correct version of the C compiler (see next point). There was also one positive report of a developer using FreeBSD, and ULIX can also be compiled on Mac OS X, but that requires some more work (see Section 18.1.1). In principle a 64-bit Linux system should work as well, but that would require some extra work because in default 64-bit installations the compiler cannot create 32-bit binaries.

- **GNU C compiler, version 4.4**
  The ULIX sources can be compiled with older or newer versions of the GNU C compiler gcc (https://gcc.gnu.org/), but when we experimentally picked a different version than 4.4, the resulting kernel did not work. This is likely caused by different code optimization. We successfully used GCC 4.4.5 on a 32-bit version of Debian Linux 6.0.1 (Squeeze, https://www.debian.org/releases/squeeze/). If it turns out that your compiler version cannot compile ULIX and you do not have the option to install GCC 4.4, then you will need to download the development environment, see Section 18.2.
NASM assembler
You need the nasm assembler (http://nasm.us/). On two development machines (Debian Linux 6.0.1 and OS X 10.6.8) nasm -v displayed the following version strings:
Debian: NASM version 2.08.01 compiled on Jun 2 2010
OS X: NASM version 0.98.40 (Apple Computer, Inc. build 11) compiled on May 18 2009

Both versions worked well, but others should, too, because the assembler must always produce the same object files from the code: Assembler code will not be optimized.

NoWEB
You have to install the noweb package (http://www.cs.tufts.edu/~nr/noweb/) which can extract the C and assembler source code files and the makefiles from the literate program ulix-book.nw. On a Debian Linux machine you can type apt-get install noweb.

\LaTeX, the Xe\LaTeX\ variant
In order to reproduce a PDF version of this book, you will need the Xe\LaTeX\ variant of the \LaTeX\ document preparation system (http://www.xelatex.org/, http://www.latex-project.org/). Depending on your Linux distribution, installing \LaTeX\ might not lead to a full installation (that contains Xe\LaTeX). On Debian Linux apt-get install texlive-xetex should fetch and install the required packages. You will also need a noweb package for Xe\LaTeX\ that is available from the Ulix project website via

wget http://ulixos.org/files/0.12/noweb.sty

You can instead use the default noweb.sty file that might be installed on your machine, but then the layout will look a bit different.

mtools
Install the mtools package (http://www.gnu.org/software/mtools/) if it is not present yet. (You can check by typing mtools in the shell; if you get a “Command not found” error, you need to install it.) On Debian systems apt-get install mtools finds the right package.

qemu
You will also need the qemu PC emulator (http://www.qemu.org/). While Ulix might run on other Intel-x86-based hardware, we only tested it in the qemu and Bochs PC emulators, and of those two only qemu was able to boot and run it. We also successfully used the Q program on OS X (http://www.kju-app.org/) which is a GUI for qemu; the package contains a qemu version, so installing Q is enough for running Ulix.

18.1.1 Toolchain on Mac OS X

It is possible to compile Ulix on an Apple Mac, but the information in this section will not be fully applicable if you use a newer version of OS X. However, it might still be helpful for finding the right files for your setup. We used Mac OS X 10.6.8 and started with installing a GCC Cross Compiler as documented in http://www.fanofblitzbasic.de/prettyos/PrettyOSMacOSX.pdf). We downloaded the file http://www.fanofblitzbasic.de/prettyos/
18.1 Required Software

i586-elf-binutils-gcc-macos.zip and unpacked it. (Note: When we attempted to re-download the file during the final preparation stage of this book, the website was offline. We could not find that PDF file or the cross compiler archive elsewhere, but on http://wiki.osdev.org/Talk:GCC_Cross-Compiler#On_Mac_OS_X_Lion the creation of a cross-compiler is discussed, so that site might help you. In the end you will need a gcc version called i586-elf-gcc that creates ELF-i386 binaries.)

We also had to install GMP and MPFR which could be automated using the port tool (https://www.macports.org/). (On newer OS X versions port is replaced by brew; http://brew.sh/.)

port install gmp
port install mpfr

Then we set some links:

```bash
ln -s /opt/local/var/macports/software/mpfr/3.0.0-p8_0/opt/local/lib/libmpfr.4.dylib /usr/local/lib/libmpfr.1.dylib
ln -s /opt/local/var/macports/software/gmp/5.0.1_0/opt/local/lib/libgmp.3.dylib /usr/local/lib/libgmp.3.dylib
```

The nasm assembler was installed by default; it is also available via the MacPorts package collection.

18.1.2 Other Useful Tools

You might find the following tools helpful though we have not used all of them for the development of Ulix.

- **All in one boot disk**, http://rescup.winbuilder.net/bootdisk/
  This is a FAT-formatted GRUB boot disk (with other tools on there, e.g., a free DOS clone and tools). It is useful because you can use the mtools utilities to copy a new Ulix kernel to the disk by typing
  ```bash
  mcopy -i bootdisk.img kernel.img ::kernel.img
  ```
  (the first : is a “drive letter” used for talking to the disk image referenced by -i).
  We modified the boot disk so that it has only one menu entry to boot /ulix.bin, and we removed the contents of the TOOLS directory that provided DOS tools such as fdisk.exe or ntfsdos.exe.

- **mfstool**, http://mfstool.sourceforge.net/
  The mfstool can access Minix filesystem images. It works on Linux, Mac OS and other Unix versions. For example,
  ```bash
  mfstool dir minix1.img
  ```
  displays the contents of the root directory in the Minix filesystem image minix1.img. However, the version we tested had problems with writing files to an image. It was good for reading files or listing directories, though.
18.2 Downloading the Development Environment

If you run into problems with your installed version of the development tools, you can either attempt to fix them or simply download a virtual machine image for the VirtualBox virtualization program (https://www.virtualbox.org/) that contains a Debian Linux 6.0.1 installation and the ULIX sources. It is distributed as an ova appliance file (Open Virtualization Format) that you can import in VirtualBox using the File / Import Appliance menu entry. Visit the http://ulixos.org/files/0.12/ova/ directory and read the instructions in readme.txt which contain updated information about the installation process.

18.3 Bootstrapping: How to Start

Assuming that you have a development environment with all the needed tools installed and the ULIX noweb source code file ulix-book.nw in your home directory, you can start by extracting the needed files from the noweb source file.

Create a directory ulix somewhere in your home directory and change into it with cd. The directory must be empty. Move the literate program ulix-book.nw that you can download with

```
wget http://ulixos.org/files/0.12/ulix-book.nw
```

into that folder and execute the command

```
notangle ulix-book.nw | sh
```

That command will extract the following root chunk ("618") of the document which contains a simple shell script that in turn creates some directories and makefiles. You will also need to download and decompress the disk images and some additional files that help with the PDF file generation. If you don’t press [Ctrl-C], the script will do that for you automatically.

The Makefiles are intended to work on a Debian Linux 6.0.1 system that has all the required tools installed. If they do not work, you might want to inspect the Makefile files in bin-build/, lib-build/ and tex-build/ which are extracted from the (bin-build/Makefile 620b), (lib-build/Makefile 622) and (tex-build/Makefile 623b) code chunks (see below).

```
#!/bin/bash
# This is the ULIX source code extractor
litprog=ulix-book.nw
files=$( ls -l | wc -l )
```
if [ $files != 1 ]; then
  echo "~/ulix directory is not empty, it must contain only ulix-book.nw. Aborting."
  exit
fi

for dir in bin-build lib-build/tools lib-build/diskfiles/bin mountpoint tex-build do
  mkdir -p ${dir}
done

for file in Makefile bin-build/Makefile lib-build/Makefile lib-build/process.ld lib-build/tools/Makefile/Makefile lib-build/process/ld/Makefile lib-build/tools/process.ld lib-build/tools/Makefile lib-build/tools/process.ld tex-build/Makefile module.nw lib-build/init.c do
  notangle -R${file} -t8 ${litprog} > ${file}
done
chmod a+x bin-build/assembler-parser.py tex-build/filter-uses.py

webroot="http://ulixos.org/files/0.12"
echo "You can download the disk images if you don't have them yet:
" echo "cd to bin-build/ and type:" echo " wget ${webroot}/ulix-fd0.img.gz" echo " wget ${webroot}/ulix-fd1.img.gz" echo " wget ${webroot}/ulix-hda.img.gz" echo " wget ${webroot}/ulix-hdb.img.gz" echo "Then uncompress them with" echo " gunzip *.gz" echo "Similarly, change to tex-build/ and type:" echo " wget ${webroot}/noweb.sty.gz" echo " wget ${webroot}/grep-patterns.gz" echo "and uncompress them as well." echo echo "This script will download all files for you if you don't press Ctrl-C" echo -n "in the next eight seconds... " for (( i=1; i<9; i++ )); do echo -n ${i}...; sleep 1; done echo "" echo "Downloading files" cd bin-build
for image in fd0 fd1 hda hdb; do echo wget ${webroot}/ulix-$image.img.gz
  wget ${webroot}/ulix-$image.img.gz
  gunzip ulix-$image.img.gz
done
cd ../tex-build
for file in noweb.sty grep-patterns; do echo wget ${webroot}/${file}.gz
  wget ${webroot}/${file}.gz
  gunzip ${file}.gz
done
echo 'Done. Type "make" to build the kernel, type "make run" to run it in qemu.'
18.3.1 **Makefiles**

We start with the makefiles which control the build processes for the kernel, the user mode library, the applications and the PDF document (this book).

The development root folder contains the following makefile that will make kernel, library and tools (when you execute `make`) and start Ulix in the qemu emulator when you type `make run`. With `make pdf` you can create the PDF file (if XƎL A TEX is installed).

```
[620a] ⟨Makefile 620a⟩≡
  all: tools bin
  pdf: ulix-book.nw  
    make -C tex-build
  bin: ulix-book.nw  
    make -C bin-build
  run:  
    make -C bin-build run

[620b] ⟨bin-build/Makefile 620b⟩≡
  OS=Linux
  LD=ld
  CC=/usr/bin/gcc-4.4
  OBJDUMP=objdump
  CFLAGS=-O0 -m32 -mstackrealign
  HDA_IMG=ulix-hda.img
  HDB_IMG=ulix-hdb.img
  FD0_IMG=ulix-fd0.img
  FD1_IMG=ulix-fd1.img
  ASM=nasm
  ASMFLAGS=-f elf
  TEXSRC_FILE=../ulix-book.nw
  TEXSRC_MODULE_FILE=../module.nw
  EXTRACT_FILES=ulix.c start.asm ulix.ld

  all: build

  build: extract parse asm compile linking objdump mtools

  extract:
    notangle -L -Rulix.c $(TEXSRC_FILE) > ulix.c; true
    notangle -Rstart.asm $(TEXSRC_FILE) > start.asm
```

The bin-build/ directory is used for compiling the kernel source file ulix.c, assembling the Assembler source file start.asm and linking the generated object files to create the kernel binary ulix.bin.
notangle -Rulix.ld $(TEXSRC_FILE) > ulix.ld
notangle -L -Rmodule.c $(TEXSRC_MODULE_FILE) > module.c
notangle -L -Rmodule.h $(TEXSRC_MODULE_FILE) > module.h

parse:
  mv ulix.c ulix.c.pre
  ./assembler-parser.py ulix.c.pre ulix.c
  sed -ie "s/Mon Nov  2 17:33:51 CET 2015/'date'/" ulix.c

asm:
  mv module.c module.c.pre
  ./assembler-parser.py module.c.pre module.c
  $$(ASM) $$(ASMFAGS) -o start.o start.asm

compile:
  $$(CC) $$(CFLAGS) -fno-stack-protector -std=c99 -g -nostdlib -nostdinc -fno-builtin -I./include -c -o module.o module.c
  $$(CC) $$(CFLAGS) -fno-stack-protector -std=c99 -g -nostdlib -nostdinc -fno-builtin -I./include -c -o ulix.o -aux-info ulix.aux ulix.c

linking:
  $$(LD) $$(LDFLAGS) -T ulix.ld -o ulix.bin *.o

mtools:
  mcopy -o -i $(FD0_IMG) ulix.bin ::

objdump:
  $$(OBJDUMP) -M intel -D ulix.bin > ulix.dump
  cat ulix.dump | grep -e '^[^ ]* <' | sed -e 's/<//' -e 's/>://' > ulix.sym

clean:
  rm -f ./*.o ./*.c ./*.h ./*.pre ./ulix.bin ./ulix.aux ./ulix.ce
  rm -f ./ulix.dump* ./asm* ./objdump* ./sym

run:
  qemu -m 64 -rtc base=localtime -boot a -fda $(FD0_IMG) -fdb $(FD1_IMG) -hda $(HDA_IMG) -hdb $(HDB_IMG) -d cpu_reset -s -serial mon:stdio |
  tee ulix.output

nolog:
  qemu -m 64 -rtc base=localtime -boot a -fda $(FD0_IMG) -fdb $(FD1_IMG) -hda $(HDA_IMG) -hdb $(HDB_IMG) -d cpu_reset

The lib-build/ directory is used for compiling the user mode library (from its source files ulixlib.c and ulixlib.h) and the user mode applications in lib-build/tools/. The generated Unix binaries will be placed in lib-build/diskfiles/ and then copied to the hard disk image file bin-build/ulix-hda.img.

You might want to set up your Linux system so that you can run sudo without entering a password, otherwise making the files in this directory will ask for your password.
The ULIX Build Process

\[622\] \langle \text{lib-build/Makefile 622} \rangle \equiv
\begin{align*}
&\text{OS=Linux} \\
&\text{LD=ld} \\
&\text{CC=/usr/bin/gcc-4.4} \\
&\text{OBJDUMP=objdump}
\end{align*}

\begin{align*}
&\text{NOWEBFILE=../ulix-book.nw} \\
&\text{ROOTDISK=../bin-build/ulix-hda.img}
\end{align*}

\begin{align*}
&\text{CCOPTIONS=-nostdlib -ffreestanding -fforce-addr -fomit-frame-pointer} \\
&\text{-fno-function-cse -nostartfiles -mtune=i386 -momit-leaf-frame-pointer -O0} \\
&\text{CCASMOPTIONS=-fverbose-asm -masm=intel} \\
&\text{LDOPTIONS=-static -s}
\end{align*}

\textit{all: build}

\textit{build: extract compile image}

\textit{extract:}
\begin{align*}
&\text{notangle \(-L \Rulixlib.c < $(NOWEBFILE) > ulixlib.c ; true} \\
&\text{notangle \(-L \Rulixlib.h < $(NOWEBFILE) > ulixlib.h ; true}
\end{align*}

\textit{compile:}
\begin{align*}
&\text{$(CC) (CCOPTIONS) \-g $(CCTESTOPTIONS) \-c ulixlib.c} \\
&\text{$(CC) (CCOPTIONS) $(CCTESTOPTIONS) \-c init.c} \\
&\text{# link it with linker script "process.ld"} \\
&\text{$(LD) $(LDOPTIONS) \-T process.ld \-o init init.o ulixlib.o} \\
&\text{touch tools/*/c} \\
&\text{make \(-C tools}
\end{align*}

\textit{image:}
\begin{align*}
&\text{sudo mount \(-o loop $(ROOTDISK) ../mountpoint} \\
&\text{cp init ../mountpoint/} \\
&\text{sudo umount ../mountpoint}
\end{align*}

\textit{clean:}
\begin{align*}
&\text{rm \-f ./*.o}
\end{align*}

We’ve already shown the \langle \text{lib-build/tools/Makefile 236} \rangle code chunk when we introduced ELF binaries.

\section{18.3.2 Linker Configuration Files}

The two code chunks \langle \text{lib-build/process.ld 191b} \rangle and \langle \text{lib-build/tools/process.ld 623a} \rangle contain process.ld files which are used to configure the behavior of the GNU linker \texttt{ld}. One of those files belongs in the \texttt{lib-build/} folder and is only used for creating the flat binary file \texttt{init}, the other one belongs in \texttt{lib-build/tools/} and is used for linking all the regular programs which are ELF binaries. (The code chunk \langle \text{lib-build/process.ld 191b} \rangle was already
shown in Chapter 6.3.)

\[\text{lib-build/tools/process.ld} \equiv \]
\[
\text{OUTPUT_FORMAT("elf32-i386")}
\]
\[
\text{ENTRY(main)}
\]
\[
\text{virt = 0x00000000;}
\]
\[
\text{SECTIONS}
\]
\[
\begin{align*}
. & = \text{virt;} \\
\text{setup} & : \text{AT(virt)} \{ \\
\text{.} & = \text{.;}
\end{align*}
\]
\[
\begin{align*}
\text{.} & = \text{ALIGN(4096);} \\
\text{text} & : \text{AT(code)} \{ \\
\text{code} & = \text{.;}
\end{align*}
\]
\[
\begin{align*}
\text{.} & = \text{ALIGN(4096);} \\
\text{data} & : \text{AT(data)} \{ \\
\text{data} & = \text{.;}
\end{align*}
\]
\[
\begin{align*}
\text{.} & = \text{ALIGN(4096);} \\
\text{bss} & : \text{AT(bss)} \{ \\
\text{bss} & = \text{.;}
\end{align*}
\]
\[
\begin{align*}
\text{.} & = \text{ALIGN(4096);} \\
\text{.} & = \text{ALIGN(4096);} \\
\text{.} & = \text{ALIGN(4096);} \\
\text{end} & = \text{.;}
\end{align*}
\]

\text{tex-build/} \text{is the folder in which you can recreate the PDF file of this book.}

\[\text{tex-build/Makefile} \equiv \]
\[
\text{TEX=xelatex -8bit -shell-escape}
\]
\[
\text{all:}
\]
\[
\begin{align*}
\text{nodefs} & -\text{auto cee tmp.nw} | \text{sort -u > noweb.defs} \\
\text{grep } & -v -f \text{grep-patterns noweb.defs} > \text{noweb.filtered.defs} \\
\text{noweave} & -\text{indexfrom noweb.filtered.defs} -\text{delay tmp.nw} > \text{tmp.tex.in} \\
\text{./filter-uses.py} & < \text{tmp.tex.in} > \text{tmp.tex} \\
\text{sed} & -i "s/Mon Nov 2 17:33:51 CET 2015/`LANG=C date`/" \text{tmp.tex} \\
\text{sed} & -i "s/\(z/a/g\)/tmp.tex \\
\text{sed} & -i "s/\(z/a/g\)/tmp.tex \\
\text{\$(TEX)} & \text{tmp} \\
\text{bibtex} & \text{tmp} \\
\text{makeindex} & \text{tmp.idx} \\
\text{noindex} & \text{tmp} \\
\text{\$(TEX)} & \text{tmp} \\
\text{\$(TEX)} & \text{tmp} \\
\text{mv} & \text{tmp.pdf} ..улос-book.pdf
\end{align*}
\]
\[
\text{clean:}
\]
\[
\text{rm} -f ./tmp.*
\]

(Note that the three \text{sed} commands are shown wrong in this code chunk, the first one replaces \text{SCRIPTBUILD} with today's date, the second and third ones replace \text{<} and \text{>} with \text{≤} and \text{≥} which you cannot see here because the transformation was also applied to those lines. Extracting the Makefile gives you a correct file.)
18.3.3 The Assembler Pre-Parser

The following Python program performs transformations of a simplified inline assembler syntax to the regular syntax (as expected by the GNU C compiler).

```python
#!/usr/bin/python

""
This Parser replaces code of the following form:

```asm
starta: mov eax, 0x1001 // comment
mov ebx, 'A' // more comment
int 0x80
```

with code that looks like this:

```asm
asm ("\n    .intel_syntax noprefix; \
    starta: mov eax, 0x1001; \n    mov ebx, 'A'; \n    int 0x80; \n    .att_syntax; \
    ");
```

It also understands asm volatile. What it cannot cope with is variable / register usage. Note that it does not change the number or position of code lines.

```
from sys import argv, exit
if len(argv)<3:
    print ("Error: give input and output filenames")
    exit (1)
infilename = argv[1]
outfilename = argv[2]

global ReplaceMode
ReplaceMode = False

def count_leading_blanks (line):
    counter = 0
    while line and (line[0] == " "):  
        counter+=1
        line = line[1:]
    return counter

def remove_trailing_blanks (line):
    if (line == ")
        while (line != ") and (line[1] == "):  
            line = line[1:
        return line
```

```python
from sys import argv, exit
if len(argv)<3:
    print ("Error: give input and output filenames")
    exit (1)
infilename = argv[1]
outfilename = argv[2]

global ReplaceMode
ReplaceMode = False

def count_leading_blanks (line):
    counter = 0
    while line and (line[0] == " "):  
        counter+=1
        line = line[1:]
    return counter

def remove_trailing_blanks (line):
    if (line == ")
        while (line != ") and (line[1] == "):  
            line = line[1:
        return line
```
def transform(line):
global ReplaceMode
if ReplaceMode:
    if "}" in line:
        # reached the end; skip this line
        blanks = count_leading_blanks(line)
        line = (blanks+2) * " " + '.att_syntax; ");'
        ReplaceMode = False
        return line
    else:
        # do something to the line
        if '//' in line:
            # remove comment
            pos = line.find("//")
            line = line[:pos]
        line = remove_trailing_blanks(line)
        line = line + "; \"
        return line
else:
    # do something to the line
    if '//' in line:
        # remove comment
        pos = line.find("//")
        line = line[:pos]
    line = remove_trailing_blanks(line)
    line = line + "; \"
    return line

def process(line):
global ReplaceMode
line = line[:-1]
if ReplaceMode:
    # we're already in ReplaceMode, working on assembler
    line = transform(line)
else:
    # we're in normal C mode, check for asm {
    if ("asm volatile{" in line) or ("asm volatile {" in line):
        blanks = count_leading_blanks(line)
        line = blanks * " " + 'asm volatile (.intel_syntax noprefix; \"
        ReplaceMode = True
    elif ("asm{" in line) or ("asm{" in line):
        blanks = count_leading_blanks(line)
        line = blanks * " " + 'asm (.intel_syntax noprefix; \"
        ReplaceMode = True
    return line

infile = file(infilename, "r")
outfile = file(outfilename, "w")
EndOfFile = False
for line in infile:
    line = process(line)
    outfile.write(line +"
"
infile.close()
outfile.close()
18.3.4 Creating Modules with module.nw

The file module.nw is intended for students who want to work on a ULIX-related project but create their own literate program document. From the file two code chunks are extracted, resulting in bin-build/module.c and bin-build/module.h. The C file will also be compiled, and the resulting object file module.o is linked with the other kernel object files.

```plaintext
(module.nw 626a)≡

<<module.c>>=
#include "module.h"
void initialize_module () {} 
@

<<module.h>>=
void initialize_module();
extern int printf(const char *format, ...);
@

The module.c function must provide an initialize_module function which will be called during kernel initialization.
```

18.3.5 Pretty Printing for the Book

The filter-uses.py script enables the limited pretty-printing that we have used in this book. It replaces brackets ({}[]), exclamation marks, #include and #define statements and C and Assembler comments with highlighted versions.

```plaintext
tex-build/filter-uses.py 626b)≡
#!/usr/bin/env python

import fileinput
from re import sub

deletemode = False
codemode = False
asmmode = False
breakmode = False
nosyntaxmode = False

for line in fileinput.input():
    line = line[:-1]  # remove \n
if line == "%nouse":
    deletemode = True
    # print "% DELETE MODE ON"

if "%BEGIN ASM CHUNK" in line:
    asmmode = True

if "%END ASM CHUNK" in line:
    asmmode = False
```
if "%BEGIN NOSYNTAX" in line:
    nosyntaxmode = True

if "%END NOSYNTAX" in line:
    nosyntaxmode = False

if "%BREAK BEFORE DEFINES" in line:
    breakmode = True

if "nwendcode" in line:
    codemode = False

if breakmode and "nwindexdefn" in line:
    line = sub (r"\nwindexdefn", r"\pagebreak\nwindexdefn", line)
    breakmode = False

if not nosyntaxmode:
    if codemode == True:
        if asmmode == False:
            # highlight C comments
            line = sub (r"//.*"$, r"\{\green\emph\{1}\}", line)
            line = sub (r"/\*", r"\{\green\emph{/\*", line)
            line = sub (r"*/", r"*/\}\}", line)
        else:
            # highlight ASM comments
            line = sub (r";.*"$, r"\{\green\emph\{1}\}", line)
            line = sub (r":\", r"\{\lightblue{}\}", line)
            line = sub (r"\":\", r"\{\lightblue{}\}", line)
            line = sub (r";\!"$, r"\{\red{}\}", line)
            line = sub (r"!;\!"$, r"\{\red{}\}", line)
            line = sub (r";\{\{\}, r"\{\red{}\}\}", line)
            line = sub (r"\{\{\}, r"\{\red{}\}\}", line)
            for keyword in (r"#define", r"#include"):
                line = sub (keyword, r"\emph{+keyword+r}"$, line)

    if "nwenddeflinemarkup" in line:
        codemode = True

    if deletemode and "nwidentuses" in line:
        line = sub (r"nwidentuses.*nwindexuse", r"nwindexuse", line)
        deletemode = False

    print line

Uses print 600.
For those readers who might want to modify the build process, we give some more details in the following sections.

### 18.4 Directory Hierarchy and Makefiles

In the `ulix/` directory you find several files and directories after the initial build process:

- `ulix.pdf` is a PDF version of this book. It will only be generated if you have `XeLaTeX` installed (which is not a requirement for simply testing ULIX).
- `bin-build/` contains the kernel source files `ulix.c` and `start.asm` which are compiled and linked into the kernel binary with `gcc`, `nasm` and `ld`.
- `lib-build/` holds the user mode library source files `ulixlib.c` and `ulixlib.h` as well as a sub-directory for the user mode applications.
- `lib-build/tools/` is the place where the application source files reside. All of those are C programs.
- `lib-build/tools/diskfiles/` collects files which shall be placed in the ULIX root disk image file (that is located in `bin-build/minixdata.img`). The image `bin-build/ulix-fd0.img` only contains the boot loader GRUB and the ULIX kernel (and no other data).
- `tex-build/` is used for generating the PDF documentation. It also contains some extra `LaTeX` files which are not part of a standard `LaTeX` distribution, such as the `noweb` package (`noweb.sty`).
- The ULIX source root directory (where you placed `ulix-book.nw`) and all `-build` directories contain makefiles which can be executed by simply changing to the directory and calling `make`. They do also provide some options for partial builds or cleanup operations (see next section).

### 18.5 Making and Booting

In this section we provide some further details about the build process and the ways to execute ULIX.

#### 18.5.1 User Mode Applications

In order to compile a user mode program, place its source code file in the `lib-build/tools` folder. We assume that the source file is called `myprogram.c`. If you simply want to check whether it compiles, type `make myprogram` (while in the `lib-build/tools` directory), that will generate a `myprogram` binary. For installing it in the disk image, change the directory to `lib-build` and type `make`. You can then test the program by entering the ULIX development root directory and typing `make run`.

Note that each user mode program must start with the line
#include "../ulixlib.h"

which includes the Ulix library headers.

If you also made changes to the library (by modifying the ulix-book.nw file) you need to call make twice in lib-build.

### 18.5.2 The Disk Images

Ulix uses four disk image files:

- **ulix-fd0.img**: This file is FAT-formatted and contains the boot loader GRUB and the Ulix kernel file ulix.bin. It is updated whenever you rebuild the kernel.
- **ulix-fd1.img**: This is a Minix-formatted floppy image that is mounted on the /mnt directory when Ulix boots. It does not contain any relevant files, so you can reformat it with mkfs.minix -2 ulix-fd1.img.
- **ulix-hda.img**: This disk image is used as the first hard disk, even though it has the layout of a 1.44 MB floppy disk. It is the root disk, i.e., it is mounted to /. You can reformat it, but then you need to reinstall the init program and the other applications (via make in the lib-build directory).
- **ulix-hdb.img**: The 100 MByte hard disk image is mounted on the /tmp directory and holds the 64 MByte swap file /tmp/swap that Ulix uses for paging out page frames. It is required, but you can also add other files to it.

### 18.5.3 Alternative Boot Options

If you look at the Makefile in bin-build, you will notice that there is another make target besides run which also starts the PC emulator: By typing make nolog you can start qemu without the option that gathers the serial line output, displays it in the terminal and also writes it to the log file ulix.output.

For experiments, you can add further make targets which use modified options.

### 18.5.4 Informative Files

When you compile the kernel, the files ulix.sym and ulix.dump are created. The first one contains a listing of symbols with their addresses, and the second one contains the generated assembler code, also with addresses. When you modify Ulix and the system hangs because of invalid memory access or some other fault, the fault handler will display the faulting address. You can then use these files to check where the error occurred.

### 18.5.5 Manually Inspecting the C Files

If you want to have a look at the C files which are extracted from ulix-book.nw because you prefer to see functions in a complete version (instead of the chunk-based presentation in this book), you will notice that the files are garbled with hundreds of source line modifiers of the form
They allow the C compiler to show the line number in `ulix-book.nw` (instead of the line number in the current C file) when printing error messages. In order to get rid, `untangle` the C file without the `-L` option, i.e., run

```
notangle -Rulix.c ../ulix-book.nw > ulix.c
```

instead of

```
notangle -L -Rulix.c ../ulix-book.nw > ulix.c
```

### 18.5.6 Other Emulators

Early versions of Ulix were also compatible with the Bochs PC emulator which has a comfortable graphical debugger (if you install the right version of Bochs). However, the current version does not boot on the Bochs machine.

You could also try to use Ulix with virtualization software (such as VirtualBox or VMware Workstation), but we have not tested that.

### 18.6 Online Resources: the ulixos.org Website

We already mentioned the website as the download resource for all the files we have discussed above. You may find updated information in a `readme.txt` file in the `http://ulixos.org/files/0.12/` directory. Also check the start page, `http://ulixos.org/`, for information about new Ulix releases.

### 18.7 Tools

The last section is not strictly related to the build process. Here we merely present the `bindump` tool that was mentioned in the Minix implementation chapter (Chapter 12.5).

#### 18.7.1 bindump

Similar to `hexdump`, here’s an implementation of `bindump`. The tool has an option `-r` which reverses the output order of 8-bit-strings (bytes; e.g. `10100000` instead of `00000101`). `bindump` accepts no filename, you must use it as a filter (e.g. `bindump -r < image.img`).

The tool was helpful during the early implementation phase of the Minix filesystem since it allowed to print the inode and zone bitmaps in a readable form. It is not automatically extracted from the book sources, but you can copy and paste its code from `ulix-book.nw` if you want to use it, too.
631

18.7 Tools

⟨bindump source code 631⟩≡

[631]

// bindump.c
// use as filter:
//
bindump < image.img
//
bindump -r < image.img
//
cat image.img | bindump

(for regular output, lower bits on the right)
(for reversed output, lower bits on the left)

#include <stdio.h>
int rev;

// reverse output?

void binwrite (byte c) {
unsigned int v = (unsigned int)c;
int i;
for (i = 7; i > -1; i--) {
if (rev == 0) printf ("%d", (v>>i)%2);
else
printf ("%d", (v>>(7-i))%2);
};
printf (" ");
}

// regular output
// reversed output

void bindump (byte *bytes, int offset, int num) {
int i; byte c;
printf ("%08x ", offset);
for (i = 0; i < num; i++) binwrite (bytes[i]);
printf (" ");
for (i = 0; i < num; i++) {
c = bytes[i];
if ((c > 31) && (c < 128)) printf ("%c",c);
else
printf (".");
};
printf ("\n");
};
int main (int argc, char *argv[]) {
byte buf[8]; int count; int pos = 0;
rev = 0;
// Test if option -r is set:
if ((argc > 1) && (strequal (argv[1], "-r"))) rev = 1;
do {
count = read (0, &buf, 8);
if (count > 0) bindump ((byte*)&buf, pos, count);
pos += 8;
} while (count > 0);
return 0;
}

// reverse?
// reverse!


Where to Go Now?

You’ve done it: you finished the book (unless you skipped to this chapter early), and that means you’ve seen the whole source code of the Ulix operating system. Now you know how a Unix-like system works internally, and that tells you a lot about how most other systems function. Of course, Ulix differs a lot from Linux or Windows, but many of the differences are about hardware support (systems intended for practical purposes need lots of drivers for all sorts of devices), performance, stability, failure handling and of course the list of provided features.

However, there is one important topic that you have not seen in this book at all: Operating systems for multi-core (or multi-processor) architectures are more complex since they have to handle a lot more parallelism; after all, on such systems several cores or CPUs execute instructions at the same time, and it may happen that two or more processes simultaneously make a system call or run a faulting instruction. Similarly the scheduler may be required to pick a new process on several cores at the same time. This has many consequences for the operating system code which needs to be protected better against the typical problems that parallelism causes.

So if you want to understand why Linux or Windows is able to use your quad-core machine so efficiently, you need to go on reading.


Looking at the Linux sources (or those of one of the free BSD versions) could be a next step, though that would require lots of time. If you’re more interested in Windows, Microsoft used to provide a stripped-down but very well-documented version of the Windows 2003 Server kernel, called the “Windows Research Kernel” (WRK) which was available to instructors via Microsoft’s Academic Alliance website and was later moved
to http://www.microsoft.com/resources/sharedsource/Licensing/researchkernel.mspx, however that section of the website has been moved once more and we are currently unaware of any WRK download resources—perhaps someone in your faculty still has a copy of it.

If you want to test your understanding of the Ulx code (and have already worked on the exercises) you might want to continue with a bigger project. Here are some suggestions for improvements of the Ulx kernel:

- **Enable partition support:** Currently Ulx treats hard disks like floppies, i.e., unpartitioned. Understanding either the classical MBR or the new GUID partition tables (GPT) and adding code to Ulx so that partitions can be accessed via /dev/hda1, /dev/hda2 etc. is not too complicated but will still require some time to get it right.

- **Add network support:** Ulx would get closer to being a proper Unix system if it could access the network. The task would be twofold: a) Write a hardware driver for a standard network adapter (e.g. the one that is provided as a virtual network card by qemu), and b) Write or port a TCP/IP stack to Ulx.

- **Port some interesting user mode applications to Ulx.** For example, there is a simple implementation of a vi clone which can do very limited editing of text files that are no longer than 23 lines (because it does not support scrolling). You could take this code and build it into a proper editor.

If you’re able to read the German language, you can also have a look at the publication list on the Ulx website: There are links to several Bachelor’s theses which describe the implementations of various Ulx components.

As a closing remark, we’d like to rephrase what we wrote in the foreword: We hope that you’ve found this book interesting and helpful for gaining some understanding of operating system concepts. We believe that our approach of presenting the whole source code of a simplified Unix system in the literate programming style is unique and worthwhile. If you agree (or disagree), then please drop us a note and tell us how the Ulx book worked for you.
Introduction to C

In this chapter we give you a very short introduction to programming with C—and we expect that you have some previous knowledge of one of the object-oriented successors of C, such as C++ or C#.

A.1 No Classes, no Objects

The most important difference between C and the other languages is that C is no object-oriented language. It knows neither classes nor objects. This means that you have to change your way of conceptually thinking of code: Where you have been used to define a class and implement methods that can manipulate objects of that class, this is not possible with C. Instead you need to write functions, and these functions are independent of specific “data objects”. If you want to store data, you declare variables, and you must provide a function with that variable (while calling it).

Imagine a string class that has a reverse function. If you have an object s, you might put the statement s.reverse(); in your code and expect that this changes the order of the characters in the string s. In C, you could implement a function reverse() which has the following prototype:

\[
\begin{align*}
\text{Example function 635} &\equiv \\
\text{void reverse (char *arg)}; \\
\end{align*}
\]

You would then call the function by using the statement reverse (s);. (We will explain why the argument is written char *arg in the next section.)

C is a typed language, and it does not allow you to overload its functions. So, by continuing the above example, if you also had a list class in an object-oriented language, you might find that that list class also provides a reverse() method, using the same name.
Then, if you had a string s and a list l, you could use the syntactically identical method invocations s.reverse(); and l.reverse(); to have both objects reversed—even though the technical details of the methods’ implementations might differ a lot.

It is not possible to have two C functions of the same name, so in this situation the best solution would be to write two functions with appropriate names, such as reverse_string() and reverse_list().

### A.2 Data Types, Arrays and Pointers

Since classes are not available, C needs to provide an alternative for declaring user-defined complex data types (which have several simpler elements, similar to member fields of an object). The C keyword `struct` is used for defining a `structure`. For example, the following definition declares a complex number that consists of two real numbers:

```c
typedef struct {
    float re;
    float im;
} complex;
```

This creates the same kind of structure, but via the `typedef` keyword you assign the name `complex` to that structure. Then, you can declare variables by simply stating `complex c;`.

In both cases, you can access the fields of the variable c with a syntax that is similar to the one that C++ and Java use for member access: the `dot notation`. Typing `c.re` will give you the real component of the number, and `c.im` is the imaginary component.

Often several instances of a variable are needed, and for this purpose C provides `arrays`. If you want `cnumbers[]` to be an array that can hold 20 complex numbers, you would write

```c
struct complex cnumbers[20];
```

or

```c
complex cnumbers[20];
```

(depending on whether you chose the first or second method of defining the new type). You can then access the 20 individual entries of the array by putting the index in square brackets. Note that C starts counting at 0, thus valid index numbers for the example array range from 0 to 19: `cnumbers[0]` is the first number, and `cnumbers[19]` is the last. Getting
the real and imaginary parts is done via the dot notation again, so `cnumbers[0].re` is the real part of the first complex number.

If you want to add all the complex numbers in the `cnumbers[]` array and store the sum in the `sum` variable, you could write the following loop:

```c
complex sum = { 0, 0 };  // set sum.re = sum.im = 0
int i;
for (i = 0; i < 20; i++) {
    sum.re += cnumbers[i].re;
    sum.im += cnumbers[i].im;
}
```

(x += y; is a short form for x = x + y; and the first line shows how you can initialize a structure without using the field names.)

It is impossible to directly add two complex numbers, because you cannot create your own version of the `+` or `+=` operator—the following loop cannot be expressed in C:

```c
complex sum = { 0, 0 };  // set sum.re = sum.im = 0
int i;
for (i = 0; i < 20; i++) {
    sum += cnumbers[i];  // ! cannot do that
}
```

You could, however, write a function `addto()` that takes two complex numbers and adds the second one to the first one:

```c
void addto (complex *c1, complex *c2) {
    *c1.re += *c2.re;
    *c1.im += *c2.im;
}
```

and then rewrite the add loop as

```c
complex sum = { 0, 0 };  // set sum.re = sum.im = 0
int i;
for (i = 0; i < 20; i++) {
    addto (&sum, &cnumbers[i]);
}
```

What’s happening here or, more specifically, what are the `*` and `&` operators doing? Let’s start with the `&` operator which is called the *address-of operator*: It “gets” the memory address of the variable, i.e., a numerical value that says where in memory the variable is stored. In the above loop this happens with both `sum` and `cnumbers[i]`. The two addresses are then provided to the `addto` function.

When you look at the function’s prototype, you see that `addto()` does not expect two complex *values* but something else, namely, two addresses of complex variables. These are just 32-bit integers (if you’re working on a 32-bit machine), but the function knows to
expect complex numbers (and not something else) at those addresses. Inside the function you cannot directly access the numbers by writing c1 or c2, because those two arguments are not complex numbers, but addresses. In order to access the contents, you have to dereference the address, and that is what the * operator is for. It is called the dereference operator.

c1 and c2 are what C calls pointers: internally they store the addresses of two complex variables, and practically that turns them into pointers to those variables. Prefixing them with the * operator turns them into the (wanted) complex variables. Thus, *c1 and *c2 are of type complex. You already know how to access the real and imaginary parts of a complex variable, so it should be obvious why *c1.re and *c2.re deliver the intended values.

Why can we use addto() to actually change the value of the first operand? That’s because the memory addresses of the real variables are provided. Using pointers as function arguments is a form of call-by-reference (as opposed to call-by-value where a function gets to work with a copy of the values).

Since expressions of the form *var.element are often needed, there is a different way to write them which is better readable and also makes it clearer that we deal with a pointer: that alternative is var->element. The -> combination looks like an arrow (it points). The prettier way to write the addto function is this:

```c
void addto(complex *c1, complex *c2) {
    c1->re += c2->re;
    c1->im += c2->im;
}
```

A.3 Strings? There is no String

One of the properties of C is that there is no built-in string type. But obviously, strings are much needed. What C does have, is an elementary character type (char) that simply is a signed byte, storing values between -128 and 127; the ASCII table holds only 128 values, so the positive numbers of this range are sufficient to store any ASCII character. You can also use the unsigned char type which ranges from 0 to 255.

The simplest way to introduce a string in your program is using a character array:

```c
char my_string[128];
```

(or with unsigned char instead of char).

This defines a character array of length 128. All functions in the standard C libraries use a null character (ASCII value 0) to mark the end of such a string, so the above definition actually provides a string that can hold no more than 127 characters: The last position cannot be used other than to store a 0 value (to indicate that this string consists of 127 characters). This is a popular source of programming errors, where a developer thinks “my password string can consist of 16 characters” and then proceeds to write char password[16] which lets him store only 15 characters.
An alternative way to think of a string is as just a consecutive chunk of memory (where a character is stored at each of its addresses). Then it is enough to simply store the starting address of the string. A pointer can do just that, and in case of strings, a pointer to char makes the most sense. So typing

\[
\text{char } \ast \text{my_string;}
\]  

does also declare a string. However, there’s an important difference: The first example (with the array) reserves a defined amount of memory where the string can be stored—the new version does not do so. It simply says: \text{my_string} points to a chunk of memory which is interpreted as a string. That address may or may not be initialized to 0. In any case, just declaring the string does nothing that helps us store a string.

How can we use such a pointer? First of all, if you already have a string (say, one declared as an array), you can assign its address to the pointer. Consider the following code:

\[
\begin{align*}
\text{char array_string[12] = "Hello World";} & \quad // \text{declares and assigns} \\
\text{char } \ast \text{pointer_string;} & \\
\text{pointer_string} = \text{array_string;}
\end{align*}
\]

The first line shows how strings can be filled with content at the time of declaration. The \textit{string literal} "Hello World" is just a shorthand for \{'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '\0'\} which is the standard way to initialize arrays during declaration. When you type a string in "..." signs, the string-terminating '\0' character is automatically added. The second line declares a pointer to char. The interesting instruction is in the last line: it assigns \text{array_string} to \text{pointer_string}. When you use just the name of an array, that is automatically interpreted as the (starting) address of the array. Thus, \text{array_string} is a pointer to char, and this address is assigned to \text{pointer_string} by the instruction. Afterwards, both \text{pointer_string} and \text{array_string} reference the \textit{same} memory address. You can check that by changing a single character in one string and then printing the other one:

\[
\begin{align*}
\text{array_string[5]} & = '_'; & // \text{replace blank with '}' \\
\text{printf ("array_string: %s\\n", array_string);} & // \text{print the first string} \\
\text{printf ("pointer_string: %s\\n", pointer_string);} & // \text{print the second string}
\end{align*}
\]

The output will be identical (\textit{Hello World} with an underscore). The assignment does not work vice versa, i.e., you cannot have an instruction \text{array_string} = \text{pointer_string} in the same situation: the compiler knows that the \text{array_string} variable points to a fixed memory address, whereas \text{pointer_string} is truly a variable. What is changeable in an array variable, are the elements, not the position and size of the array.

If you have no array-type strings, then how to get a free memory location? You can use the \texttt{malloc()} function (memory allocation) to request space. If you want to store a string with 100 characters (plus the terminating zero), you could type

\[
\text{pointer_string} = \text{(char*) malloc(101);}
\]
type cast

The `malloc(101);` call will reserve memory and return its start address which is then stored in the variable. The new memory may or may not be initialized. The extra `(char*)` part before `malloc` performs a type conversion: the return value of `malloc` is `(void*)`, a pointer to data of unknown type. Placing the desired type `(char*)` in round brackets in front of it changes the type to `char*`. This kind of conversion is called a type cast or simply a cast. You can always convert pointers to something into pointers to something else, though some conversions make no sense. A conversion in the C program does not translate into the execution of any code (in the generated assembler language): a pointer is an address, and all addresses have the same type when looked at at the machine level; on a 32-bit machine every address is a 32-bit integer. But the C compiler keeps track of how you define your pointers and issues warnings when you assign a pointer to a pointer of a different type without adding the explicit cast.

### A.4 String Operations

Once you have the memory that is necessary to store a string (either via a direct array declaration of via `malloc`), you want to work with the string. As C has no string type and no methods, you cannot write `s1 = s1 + s2;` or `s1.append(s2);` in order to append a string `s2` to another string `s1`. Instead you need to use a function that does just that. The same holds for coping: You cannot assign a string to another one (and expect that to result in a copy), so even `s1 = s2;` is illegal for character arrays. It is legal when `s1` is a pointer, but does not duplicate the string. It copies the address. The standard functions for copying a string and appending a string (to another one) are

```
char *strcpy (char *dest, char *src);
char *strcat (char *dest, char *src);
```

Instead of `s1 = s2;` you would write `strcpy (s1, s2);` and instead of `s1 = s1 + s2;` you need `strcat (s1, s2);`. However, these functions may not always have the intended effect, and that is the source of many security holes in applications.

You can find our implementation of `strcpy` in the book: ULiX provides its own version since it cannot use the functions which are available on the development (Linux) system.

The functions `strcpy` and `strcat` should be used very carefully because they can write beyond the end of the memory area that was reserved for the string. Typing

```
char s1[10];
char s2[25] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
strcpy (s1, s2); // copy long string s2 into s1
```

is perfectly legal C code, but `strcpy()` will not stop when it reaches the end of `s1`. Instead it will just fill the following bytes as well, and that may result in a number of things, for example `s2` being destroyed (if it is located directly behind `s1`) or an application crashing if the addresses behind `s1` are not available.
That is why it is best to replace all uses of `strcpy` and `strcat` with their safe variants which are called `strncpy` and `strncat`: they have a third argument via which you can limit the number of bytes that are actually written:

```
char *strncpy (char *dest, char *src, size_t n);
char *strncat (char *dest, char *src, size_t n);
```

Using them (and using the right value for \( n \)) the following code causes no problems (though it still cannot achieve what can’t be done, i.e., copying a large string into a small one):

```
char s1[10];
char s2[25] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
strncpy (s1, s2, 10); // copy up to 10 characters of s2 into s1
```

This code will not write beyond the end of \( s1 \), but it will also leave \( s1 \) in a bad state: the string will not be null-terminated, but instead contain the first ten characters of \( s1 \) without termination. The only way to avoid this (if the size limit cannot be helped) is this:

```
char s1[10];
char s2[25] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
strncpy (s1, s2, 9); // copy 9 characters...
s1[9] = '\0'; // and manually terminate the string
```

Of course there are other ways that let you avoid such situations. For example, when you want to copy a string you can first use the function

```
size_t strlen (const char *s);
```

to discover the length of a string, then reserve the appropriate amount of memory for a copy (using `malloc()`) and then create the copy.

For comparing two strings you need yet another function since a simple comparison like \( (s1==s2) \) will only compare the strings’ starting addresses.

```
int strcmp (const char *s1, const char *s2);
int strncmp (const char *s1, const char *s2, size_t n);
```

compare \( s1 \) and \( s2 \) character-by-character and return 0 if the strings are equal. If \( s1 < s2 \) (lexicographically), they return a negative number, and if \( s1 > s2 \), they return a positive number. So test for \( (strcat(s1,s2)==0) \) to check whether two strings are identical. For both functions the comparison ends when the null-termination occurs in one of the strings, the safe “\( n \)” version also stops after \( n \) characters have been read.
A.5 Pointer Arithmetic

Consider the following, naive implementation of a `strcpy()` function:

```c
char *strcpy (char *dest, char *src) {
    int i;
    for (i = 0; i < strlen(src); i++) {
        dest[i] = src[i]; // copy i'th character
        dest[i] = '\0'; // terminate dest
    }
    return dest;
}
```

It works because `dest` and `src` can be treated like arrays (though the parameters are declared as pointers). However it is neither necessary to use a counter variable nor the `strlen()` function. Instead the following version is preferred in the realm of C programmers:

```c
char *strcpy (char *dest, char *src) {
    char *tmp = dest;
    while ((*dest++ = *src++) != '\0') ;
    return tmp;
}
```

For most people who are new to C, this looks like garbage though it’s perfectly correct C (and does what you want). Let’s explain the code in detail.

- First of all `*dest++ = *src++` is C shorthand for the three commands `*dest = *src; dest++; src++;`, and the value of the whole expression is the value of `*dest` or `*src before the two increment commands. (`x++` is another shorthand, meaning `x = x+1`.)
- Adding 1 to a `char` pointer increases the memory address by 1 (as would be expected). At the beginning `dest` points to the first character of the destination string, `dest[0]`. After the increment it points to what was previously `dest[1]`, the second character.
- The `while` loop continues the byte-wise copying until a null character is found (and copied). The destination string is now complete (null-terminated), but `dest` no longer points to the beginning of the target string since multiple `++` operations have increased its value. That is why we kept the initial value in `tmp`.
- That saved value is then returned. (The `strcpy()` function must always return a pointer to the new string.)

Now why is this section titled Pointer Arithmetic? If we were to work with integer arrays (instead of character arrays), we might also be interested in a copy function, let’s call it `intarrcpy()` (integer array copy). Its implementation looks almost identical to the second `strcpy()` version, with all occurrences of “char” replaced with “int”:

```c
char *intarrcpy (char *dest, char *src) {
    int i;
    for (i = 0; i < strlen(src); i++) {
        dest[i] = src[i]; // copy i'th character
        dest[i] = '\0'; // terminate dest
    }
    return dest;
}
```
int *intarrcpy (int *dest, int *src) {
    int *tmp = dest;
    while ( (*dest++ = *src++) != 0 )
        ;
    return tmp;
}

But let’s look at the ++ operator again: If it also increased the values of dest and src by 1, the function would have to fail since a 32-bit integer needs four bytes for storage, not just one. So what we want is to add 4 to the addresses in every step of the loop. The surprising news is: This is exactly what ++ does, and that is why it is called pointer arithmetic. The ++ operator (as well as -- and the regular addition and subtraction) consider the size of the base type that the pointer points to. For an int pointer ptr, the ptr++ commands actually increases ptr’s address by sizeof(int) (which is 4).

Pointer arithmetic is not restricted to base types. If you define a struct something structure that contains a lot of data, totaling in 2604 bytes of data for each such variable, and declare a pointer of that type (via struct something *ptr;), then each ptr++; command will modify the address by adding 2604. This makes it easy to walk through array-like structures even when they were never declared as arrays.

There is one problem with pointer arithmetic that sometimes leads to wrong code. We already mentioned that you can cast pointer types to different pointer types. Look at the following example to see what can go wrong:

char s[10] = "ABCDEFGHI";
char *charptr;
int *intptr;

charptr = s; // points to the 'A' in s
intptr = (int*) charptr; // same address
intptr++; // pointer arithmetic!
charptr = (char*) intptr; // cast/copy it back

If you expect charptr to point to the 'B' in s, then you’ve made the mistake that we want to explain. It actually points to the 'E' because the pointer arithmetic was performed on an int pointer, so addresses are always modified in multiples of 4.

A.6 C Pre-Processor

The C compiler runs a pre-processor before actually compiling the code. That pre-processor looks for commands that begin with # and acts on them. These can be used for including other source files, for conditional compilation and for macro definitions. Since we use some of these features, we give a short explanation of each of these three possibilities but only discuss the details which are relevant for reading and/or modifying the Unix code.
• Including files: Using the command
   
   `#include "path/to/file.h"`

   you can include other files in the source file. Typically those are header files (ending in `.h`), but you can include any file you want. If the path name starts with a slash, it is treated as an absolute path, otherwise as a relative one. So you can include the file `xyz.h` from the upper directory by writing
   
   `#include "../xyz.h"`

   —regardless of where the files are placed absolutely.

• Macro definition: The `#define` command declares a macro. In its simplest version that leads to a simple search-and-replace. For example,

   `#define BLOCK_SIZE 1024`

   lets the compiler search the source file for the string `BLOCK_SIZE` and replace every occurrence with 1024.

   A more advanced version of macros uses parameters, so macros provide an alternative method to writing (simple) functions. A typical example is finding the smaller of two values:

   `#define MIN(x,y) ((x<y) ? x : y)`

   The expression `(x<y) ? x : y` evaluates to `x` if `x<y` is true and to `y` otherwise. Using `MIN(x,y)` in the code makes it more readable. However, this must be used with care, as the following example shows which attempts to increase two variables while picking their minimum: `MIN(v1++,v2++)` does not do what is expected because the macro is expanded to `((v1++ < v2++) ? v1++ : v2++)`. Here, `v1` and `v2` are compared. Let’s assume that `v1` is the smaller one. Then both variables are incremented (as expected). However, in the next step `v1` is increased again: The condition is true, so `v1++` is evaluated. The total result is that the value of `MIN(v1++,v2++)` is the old value of `v1 +1`, and `v1` gets incremented twice (while `v2` is incremented only once).

   When you work with macros, use them only with constant arguments or arguments which have no side effects. (In the example, `MIN(f(x1),f(x2))` with some function `f()` would also call `f(x1)` twice, not once, if `(f(x1)<f(x2))` evaluates as true.)

• Conditional Compilation: You can create simple if-then-else constructions which can remove parts of the code before compilation. This is often used for inserting debug code during the development which is then removed for the final release of the software. As an example consider the following code block:

   `#define DEBUG`

   `int somefunc (int x) {
      int res;
      #ifdef DEBUG
      printf ("DEBUG: somefunc() called with argument %d\n", x);
      res = x / 3;
      printf ("DEBUG: somefunc() going to return %d\n", res);
   }`

```c

else
    res = x / 3;
#endif
return res;
}

When you compile this code it will contain the debug output because the DEBUG macro is defined. (Note that it has no specific value, it is just defined, so #ifdef will evaluate it as true.) Simply remove that #define DEBUG line and recompile to get the version which contains only the "else" case: the lines between #else and #endif.

Of course, the same effect could be achieved without a macro (by using a DEBUG variable and a regular if expression), but then all of the code would be compiled. With the macro, the pre-processor removes the unwanted lines from the source code before the compiler runs.

The pre-processor does not touch occurrences of a macro name inside a string literal: the arguments of printf() in the example contain DEBUG in the string argument, and they remain intact.

You can check the effect of pre-processor commands by calling the gcc compiler with the -E option: gcc -E file.c -o file.i creates a new file file.i where all pre-processor commands have been executed. That file is now free of pre-processor commands. An interesting alternative to the -E option is the -save-temps option which performs all compilation steps, but keeps the intermediate files which are normally deleted. When you run the command

```
cgcc -save-temps testprog.c -o testprog
```

you can find four new files: testprog.i is the pre-processor-modified version of the source file, testprog.s is the compiled assembler version (with readable assembler source code), testprog.o is the assembled object file, and finally testprog is the binary executable file which contains library code or links to dynamically loaded libraries. Figure A.1 shows an example of the created files.

### A.7 Further Reading

Since pointers seem to be a crucial topic for most students who are new to C programming, we suggest reading a whole book about pointers: “Pointers on C” [Ree98] by Kenneth Reek is an excellent read and comes with many exercises, both practical and theoretical. The author uses diagrams to show what points where and what content is stored in which memory locations.

For those who truly want to delve into the language, the description of the C compiler in David Hanson and Christopher Fraser’s LCC book [HF95] offers deep insight. If you understand the workings of a C compiler, you’ve also mastered the language. Plus: the book is another literate programming example which in itself makes it worth reading—provided that you like this style.
Another interesting book about C is Peter van der Linden’s “Expert C Programming: Deep C Secrets” [vdL94]. The author looks at some obscure details and explains unexpected phenomena with direct quotations from the ANSI C standard definition [Ame89]. There are many comparisons of language features in C and C++ which are especially helpful if you’re used to writing C++ code. The publisher’s website has a 60-page sample.

Many good C books are rather old as the language was created in the eighties. Some of them are still available in print. If you want to have some fun with C programs, look at the International Obfuscated C Code Contest website, http://www.ioccc.org/.
Most of the Unix code is written in C, but some small parts had to be done in Assembler since C cannot directly access the CPU’s registers or execute specific CPU instructions such as the ones that enable or disable interrupts. In many cases it is sufficient to use gcc’s inline assembler feature that lets you drop a few lines of assembler in the middle of a C function (see Section B.4), but we also needed a separate assembler source file for the early steps in the system initialization. In this chapter we give a very short introduction to some of the features available on Intel i386 and higher CPUs and the syntax of the commands.

When you use assembly language, you are somewhat limited in the way you can structure the code. For example, where C has several types of loops (for, while, do) and nestable if-then-else expressions, assembler does not. Instead, you can make comparisons and jump elsewhere in the code (depending on the result of that comparison). That is closer to early Basic dialects where branching worked via “IF condition GOTO line number” statements. However, there’s no need to learn the assembler ways of expressing for and while loops, because we don’t want to write all our code in assembler.

## B.1 CPU Registers

Just like a C program that accesses a variable (which is stored somewhere in RAM), assembler code often works with memory, too. For example, the CPU provides instructions that inspect the contents of two memory cells, add or subtract them and store the result in the first of the two cells. That is basically what the C compiler creates when it compiles
a C command like \( \text{var1 += var2;} \) or \( \text{var1 -= var2;} \). But memory access is expensive: it takes some time to translate a virtual address into a physical address and fetch the memory contents via the memory bus. It is too slow to perform all operations that way. Every CPU has a set of faster memory cells: the CPU-internal registers. They are even faster than the first level cache which is embedded in the same chip as the CPU’s core: they provide instant access.

Intel’s CPUs (like many other processors) have a set of general purpose registers which can be used to hold arbitrary data and perform calculations on them, and then there’s also a set of special purpose registers which is what we’re really interested in, because we need to read or write some of those registers to influence and control paging, interrupt handling and other critical tasks.

There are eight general registers [Int86, p. 29]: \( \text{EAX, EBX, ECX, EDX, EBP, ESP, ESI, and EDI} \). You can use them to hold values and perform calculations and comparisons. Each of these registers is 32 bits wide, and you can also access the lower 16 bits of them by using a different name (\( \text{AX, BX, CX, DX, BP, SP, SI, and DI} \); all without the leading “E”). The first four registers can be separated even further into higher and lower halves which are only eight bits wide (and hold a byte): \( \text{AH, BH, CH and DH} \) are the higher halves, \( \text{AL, BL, CL} \) and \( \text{DL} \) are the lower halves (see Figure B.1) which are sometimes needed for I/O when data is read from or written to a port that grants the CPU access to the internal 8-bit register of some chip, e.g. a disk controller.

For example, in order to add the contents of \( \text{EBX} \) to \( \text{EAX} \) you could use the assembler instruction \( \text{add eax, ebx} \). The \( \text{mov} \) instruction copies (not: moves) a value from one register to another, so \( \text{mov eax, ebx} \) performs an \( \text{eax := ebx} \) action. You can also access memory locations with these commands: \( \text{mov eax, [ebx]} \) will load a 32-bit value from the memory addresses pointed to by \( \text{EBX} \) and copy it to \( \text{EAX} \)—the C equivalent would be \( \text{eax := *ebx} \) (with \( \text{ebx} \) interpreted as an \text{int*} pointer).

The registers \( \text{ESP} \) (extended stack pointer) and \( \text{EBP} \) (extended base pointer) are used for working with stacks. \( \text{ESP} \) points to the top of the stack and is changed whenever a subroutine is called or it exits and when a value is pushed onto or popped from the stack. The base pointer helps with staying oriented on the stack: While a subroutine executes, it may often modify the stack (thus changing \( \text{ESP} \)). But the stack also holds the arguments which were provided to the subroutine as well as its local variables, and by letting \( \text{EBP} \) point to the address between arguments and local variables, it is possible to access them.
without a need to consider changes to \textit{ESP}. For example, \texttt{EBP+8}, \texttt{EBP+12}, \texttt{EBP+16} store the first, second and third argument, and \texttt{EBP–4}, \texttt{EBP–8} and \texttt{EBP–12} store the first, second and third local variable (assuming that all those values are 32-bit integers). Between those two areas there is still some room: address \texttt{EBP+4} holds the return address, and at the address that \texttt{EBP} points directly to you find the “old value” of \texttt{EBP} that was used for the previous subroutine call (in a setting where subroutines call other subroutines).

\section*{B.2 A Few Standard Commands}

We will not give a detailed introduction to the instructions that the Intel x86 CPU provides, but here is a short overview of some of the most important ones. For the Intel processor platform, two “dialects” exist, the Intel and the AT&T one. The GNU C compiler supports both but defaults to the AT&T variant. We have decided to use the Intel syntax, because it is closer to C’s syntax: For example, you can load the \texttt{EAX} register with the value 0 via the command \texttt{mov eax, 0} (in Intel syntax). So the target of the \texttt{mov} command comes first which resembles the C command \texttt{eax = 0}. In AT&T syntax, the operands are reversed, with the target coming last and extra syntactical elements being needed (\texttt{mov $0, %eax}). In the following examples we will show the Intel syntax on the left hand side, and the AT&T syntax on the right.

\subsection*{B.2.1 Moving Data Around}

The simplest way of filling a register is loading an \textit{immediate} value. The \texttt{mov} instruction does that. In Intel syntax, the register name comes first, followed by the number expressed as in the following examples. In the AT&T syntax the order of arguments is reversed, and immediate values are prefixed with a dollar sign, whereas register names have a percent sign as prefix. Also, AT&T syntax appends a size identifier to the \texttt{mov} instruction, so instead of \texttt{mov} it is called \texttt{movb} (byte, 8 bits), \texttt{movw} (word, 16 bits) or \texttt{movl} (long, 32 bits).

If you want to copy the contents of a memory location, you can load the address in one of the registers and tell \texttt{mov} to look at the memory cell(s) that it points to. The last two examples in the following table show how this is done, once without an offset and once with an offset.

<table>
<thead>
<tr>
<th>Intel Syntax</th>
<th>AT&amp;T Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mov eax, 0xABCD}</td>
<td>\texttt{movl $0xABCD, %eax}</td>
<td>direct load, hexadecimal</td>
</tr>
<tr>
<td>\texttt{mov ebx, 1}</td>
<td>\texttt{movl $1, %ebx}</td>
<td>direct load, decimal</td>
</tr>
<tr>
<td>\texttt{mov eax, [ebx]}</td>
<td>\texttt{movl (%ebx), %eax}</td>
<td>copy memory at \texttt{EBX} to \texttt{EAX}</td>
</tr>
<tr>
<td>\texttt{mov eax, [ebx+0xF0]}</td>
<td>\texttt{movl 0xF0(%ebx), %eax}</td>
<td>with offset 0xF0</td>
</tr>
</tbody>
</table>

In those last two lines, if \texttt{EBX} holds the value \texttt{0xABCD0000}, then the first line will read the 32-bit integer stored at address \texttt{0xABCD0000}, whereas the second one reads address \texttt{0xABCD00F0}. In both cases the found value is written to the \texttt{EAX} register.

The original Intel syntax for hexadecimal numbers is \texttt{0xABCDh} with a \texttt{h} suffix (and a required \texttt{0} prefix if the number starts with a letter digit), but \texttt{nasm} supports both variants in
Intel mode, so we have chosen to use the 0xABC0 notation that is also C’s way of expressing hexadecimal numbers.

### B.2.2 Different Integer Sizes

In the register overview you have already seen that some registers can be accessed in ways which only use the lowest eight or 16 bits. For using these smaller versions, use the alternative names (e.g., ax for the 16-bit version and al for the 8-bit version). In the AT&T version you need to use a suffix again for expressing that you want to move a byte, word or long.

<table>
<thead>
<tr>
<th>Intel Syntax</th>
<th>AT&amp;T Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mov al, bl</td>
<td>movb %bl, %al</td>
<td>move a byte</td>
</tr>
<tr>
<td>mov ax, bx</td>
<td>movw %bx, %ax</td>
<td>move a word</td>
</tr>
<tr>
<td>mov eax, ebx</td>
<td>movl %ebx, %eax</td>
<td>move a long (32 bits)</td>
</tr>
<tr>
<td>mov al, byte ptr [ebx]</td>
<td>movb (%ebx), %al</td>
<td>move byte from memory</td>
</tr>
<tr>
<td>mov ax, word ptr [ebx]</td>
<td>movw (%ebx), %ax</td>
<td>move word from memory</td>
</tr>
<tr>
<td>mov eax, dword ptr [ebx]</td>
<td>movl (%ebx), %eax</td>
<td>move long from memory</td>
</tr>
</tbody>
</table>

Making the value size explicit makes even more sense when you access memory: Copying a byte is not the same as copying a long integer. In the Intel syntax explicit byte ptr, word ptr and long ptr keywords are used to state that a byte, word or long integer shall be read from memory, as shown in the last three lines. The equivalent AT&T commands don’t need this since the mov suffix already makes the length explicit.

### B.2.3 Arithmetic Operations

When talking about arithmetic operations, we only consider integer operations. The Intel CPUs also provides floating point operations, but we do not need them for Ulix.

For adding and subtracting you can use the add and sub instructions which take two arguments and add the source to the target; multiplication and integer division are handled by mul and div, but they don’t take two arguments but only one and use the EAX register as accumulator (i.e., EAX is implicitly both one of the source operands and the target):

<table>
<thead>
<tr>
<th>Intel Syntax</th>
<th>AT&amp;T Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>add eax, ebx</td>
<td>addl %ebx, eax</td>
<td>add EAX to EAX, eax += ebx</td>
</tr>
<tr>
<td>add eax, [ebx]</td>
<td>addl (%ebx), eax</td>
<td>add (long) memory contents at EBX to EAX</td>
</tr>
<tr>
<td>sub eax, ebx</td>
<td>subl %ebx, eax</td>
<td>subtract EBX from EAX, eax -= ebx</td>
</tr>
<tr>
<td>sub eax, [ebx]</td>
<td>subl (%ebx), eax</td>
<td>subtract (long) memory contents at EBX from EAX</td>
</tr>
<tr>
<td>mul ebx</td>
<td>mull %ebx</td>
<td>multiply EAX with EBX, result in EAX, eax *= ebx</td>
</tr>
<tr>
<td>div ebx</td>
<td>divl %ebx</td>
<td>divide EAX by EBX, result in EAX, eax /= ebx</td>
</tr>
</tbody>
</table>
B.2 A Few Standard Commands

B.2.4 Jumps, Calls, Comparisons and Conditional Jumps

Sometimes you want to jump to a specific program address in order to continue execution elsewhere. For those cases the jmp instruction can be used. When creating an assembler source file (for use with nasm) you will normally assign a label to an instruction that you want to jump to and then use it in the jmp instruction, like this:

```
infinite_loop: mov al, byte ptr [eax]
call printchar
add eax, 1
jmp infinite_loop
```

This example shows a further instruction for jumping to a new address: call also jumps to the supplied address, but before that it pushes the address of the following instruction (in the example: of add eax, 1) onto the stack. The assembler code at printchar can then execute the ret instruction which will pop that address from the stack and continue execution inside the above loop. To summarize the difference: When you jmp, there’s no easy way to get back; when you call, you can return with ret.

Simply jumping (or calling) unconditionally does not allow for any case distinctions: Code that only uses jmp and call will always execute the same commands in the same sequence. For a distinction of cases we need comparison operations and based on the result we want to decide whether we jump elsewhere or not.

The cmp instruction takes two arguments and compares them. They are either identical or one value is bigger than the other. The j* instructions in the following table jump if a certain condition (such as: first value is smaller than the second one) is met.

<table>
<thead>
<tr>
<th>Intel Syntax</th>
<th>AT&amp;T Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmp eax, ebx</td>
<td>cmpl %ebx, eax</td>
<td>compare EAX and EBX, then:</td>
</tr>
<tr>
<td>jl label</td>
<td>jl label</td>
<td>jump to label if EAX &lt; EBX</td>
</tr>
<tr>
<td>jle label</td>
<td>jle label</td>
<td>jump to label if EAX ≤ EBX</td>
</tr>
<tr>
<td>je label</td>
<td>je label</td>
<td>jump to label if EAX = EBX</td>
</tr>
<tr>
<td>jge label</td>
<td>jge label</td>
<td>jump to label if EAX ≥ EBX</td>
</tr>
<tr>
<td>jg label</td>
<td>jg label</td>
<td>jump to label if EAX &gt; EBX</td>
</tr>
<tr>
<td>jne label</td>
<td>jne label</td>
<td>jump to label if EAX ≠ EBX</td>
</tr>
<tr>
<td>sub eax, ebx</td>
<td>subl %ebx, eax</td>
<td>subtract EBX from EAX, then:</td>
</tr>
<tr>
<td>jz label</td>
<td>jz label</td>
<td>jump to label if result of last arith. operation = 0</td>
</tr>
<tr>
<td>jnz label</td>
<td>jnz label</td>
<td>jump to label if result of last arith. operation ≠ 0</td>
</tr>
</tbody>
</table>

The last three commands show that there are also conditional jumps that depend on the last arithmetic operation (instead of the last comparison). jz jumps if the zero flag is set which is the case if the last arithmetic operation resulted in a zero value. Similarly you can jump if the overflow flag is set which happens when the last operation caused an overflow, e.g., after adding 100 to 0xFFFFFFFF0.
B.2.5 Pushing and Popping With the Stack

We already mentioned the stack which is used by the call instruction for storing the return address. Using push and pop you can also push data on the stack and pop it back. If you want to call an assembler function with arguments, first push the values and then call the function. The function can then either pop all values from the stack into registers (though it has to remember the return address and restore the stack later) or it can access the stack contents directly via [ebp+8], [ebp+12] etc. if it saves the original ESP in the base pointer EBP. push and pop also accept memory locations, so you can push the value stored at the memory address that EAX points to by executing push [EAX].

In AT&T syntax, push and pop need a suffix to indicate how large the data are which must be pushed or popped, so you get pushw, pushl, popw and popl. (You cannot push or pop a single byte, see [Hyd10, p. 137].) Some assemblers support a pushb or push byte instruction, but that will actually turn a byte into a word (by filling it with zero bits) and then push that.

B.3 Special Commands

There are three kinds of special commands that we’ll introduce in this section: You can define constants (which are similar to C-defined macro constants), you can use macros (similar to C’s macros that look like functions), and you can store data bytes for creating data structures.

All of these require using the nasm assembler. If you want to work with another assembler, it is likely that the same features are available, but they might use a different syntax.

B.3.1 Data Storage (db, dw, dd)

Assembler code contains instructions and data. Since you cannot define data types (like in C), your only option is to know what a data structure should look like and then directly encode data in the binary (and later refer to the address where the data are located).

There are three commands which can store data:

- db is used to store individual bytes (or sequences of bytes). For example, dd 0x32, 0x38, 0x3b will store the three bytes 0x32, 0x38 and 0x3b (in this order). If you place a label before that command you can later reference the address where those three bytes can be found. Instead of hexadecimal (or regular) numbers, you can only provide a single character or a string: db 'xyz' will store the three bytes whose character representations are x, y and z. Note that such a string will not be null-terminated.
- dw does the same as db, but stores a double byte (a word), and
- dd stores a double word (or quad byte, consisting of four bytes). You need to use dd to store 32-bit addresses.
B.3 Special Commands

... 

\texttt{dw} and \texttt{dd} get the order of the bytes right so that they are stored in memory according to the \textit{endianess} of your platform, for example, \texttt{dd 0x12345678} will write bytes $0x78$, $0x56$, $0x34$ and $0x12$ because Intel x86 machines are of the \textit{little-endian} type where the smaller bit blocks come first.

There are also commands for storing floating-point numbers (\texttt{dq}, \texttt{dt}), but we do not need them for ULIx which uses only integers.

B.3.2 Constants (\texttt{equ})

With \texttt{equ} statements you can define identifiers which you can use in later code lines instead of the constants or expressions that you provided in the identifier definition. The syntax is always of the form

\begin{verbatim}
IDENTIFIER equ EXPRESSION
\end{verbatim}

For example, the ULIx assembler file \texttt{start.asm} contains the following lines:

\begin{verbatim}
MB_HEADER_MAGIC equ 0x1BADB002
MB_HEADER_FLAGS equ 11b
MB_CHECKSUM equ (MB_HEADER_MAGIC + MB_HEADER_FLAGS)
\end{verbatim}

They define three local identifiers \texttt{MB_HEADER_MAGIC}, \texttt{MB_HEADER_FLAGS} and \texttt{MB_CHECKSUM} that are used to create the multiboot header:

\begin{verbatim}
; GRUB Multiboot header, boot signature
dd MB_HEADER_MAGIC ; 00..03: magic string
dd MB_HEADER_FLAGS ; 04..07: flags
dd MB_CHECKSUM ; 08..11: checksum
\end{verbatim}

B.3.3 Macros (\texttt{%macro})

If your code contains repetitive sequences that you would turn into a function (when programming in C), you can use a \texttt{macro} to keep the level of repetitions down. We show you the macro that we used in \texttt{start.asm} to write the assembler functions \texttt{irq0}, \texttt{irq1}, ...

The macro definition looked like this:

\begin{verbatim}
%macro irq_macro 1
  push byte 0 ; error code (none)
  push byte %1 ; interrupt number
  jmp irq_common_stub ; rest is identical for all handlers
%endmacro
\end{verbatim}

and we called the macro this way:

\begin{verbatim}
... irq12: irq_macro 44
irq13: irq_macro 45
irq14: irq_macro 46
... 
\end{verbatim}
which the assembler expands to the following lines:

```assembly
irq12: push byte 0 ; error code (none)
push byte 44 ; interrupt number
jmp irq_common_stub ; rest is identical for all handlers

irq13: push byte 0 ; error code (none)
push byte 45 ; interrupt number
jmp irq_common_stub ; rest is identical for all handlers

irq14: push byte 0 ; error code (none)
push byte 46 ; interrupt number
jmp irq_common_stub ; rest is identical for all handlers
%endmacro
```

In the macro’s definition, the parameter `1` in the first line states that the macro can be used with one argument. That argument can be referred to via `%1`. So, while expanding the macro, every occurrence of `%1` is replaced with the argument. In the above examples we called the macro with arguments 44, 45 and 46.

If you need more than one argument (say: three), you set a different number, e.g. with `%macro name 3`. Then you access those arguments via `%1`, `%2` and `%3`.

### B.4 gcc Inline Assembler

The GNU C compiler gcc lets developers write inline assembler statements in the middle of C code. That is very helpful if only a few lines of assembler code are required—storing them in a separate assembler source file and making sure that both the C and the assembler code can see the variables which are needed in both places is laborious, and it also costs (a little) CPU time because an external assembler routine must be called via the function call mechanism whereas inline assembler code is just inserted between the translations of the preceding and successive C code.

You can find the best example for a quick line of assembler code in the book: Whenever we need to disable or enable the interrupts, we do this via the ⟨disable interrupts⟩ and ⟨enable interrupts⟩ code chunks which use the inline assembler instructions

```assembly
asm("cli"); // disable interrupts (clear interrupt flag)
or
asm("sti"); // enable interrupts (set interrupt flag)
```

to execute the `cli` and `sti` instructions.

However, this is the simple case. It is more typical that values (which are stored in C variables) have to be used as a parameter of the assembler instruction. In a pure assembler file you can write

```assembly
mov eax, some_label
```
to load the address of some_label in the EAX register, but you cannot similarly write an
inline assembler statement like

```c
asm ("mov eax, &some_C_variable")
```

to do the same for the address of some_C_variable. Instead, values or addresses must be
passed by preparing registers before the assembler instructions are executed (and fetching
possible result values via registers as well). gcc defines a special syntax to provide input
operands and output operands; the general instruction format is this:

```c
asm ("assembler instructions",
    : // optional output operands
    : // optional input operands
    : // optional "clobber" list
)
```

Operands are always written in the "reg" (variable) form; if there is more than one regis-
ter that we want to use, we use several such expressions and put commas between them.
The following registers can be used:

<table>
<thead>
<tr>
<th>Register</th>
<th>Reading</th>
<th>Writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAX</td>
<td>&quot;a&quot;</td>
<td>&quot;=a&quot;</td>
</tr>
<tr>
<td>EBX</td>
<td>&quot;b&quot;</td>
<td>&quot;=b&quot;</td>
</tr>
<tr>
<td>ECX</td>
<td>&quot;c&quot;</td>
<td>&quot;=c&quot;</td>
</tr>
<tr>
<td>EDX</td>
<td>&quot;d&quot;</td>
<td>&quot;=d&quot;</td>
</tr>
<tr>
<td>ESI</td>
<td>&quot;S&quot;</td>
<td>&quot;=S&quot;</td>
</tr>
<tr>
<td>EDI</td>
<td>&quot;D&quot;</td>
<td>&quot;=D&quot;</td>
</tr>
<tr>
<td>any</td>
<td>&quot;r&quot;</td>
<td>&quot;=r&quot;</td>
</tr>
</tbody>
</table>

We use that syntax in the syscall* functions of the user mode library: as an example,
here is the code for syscall3, which takes three parameters, stores them in EAX, EBX and
ECX, raises the interrupt 0x80 and returns a result from EAX:

```c
inline int syscall3 (int eax, int ebx, int ecx) {
    int result;
    asm ("int $0x80" : "=a" (result) : "a" (eax), "b" (ebx), "c" (ecx) );
    // |- instruction -| |- output regs -| |- input registers -|
    return result;
}
```

The instruction explicitly copies the variables eax, ebx and ecx into the corresponding reg-
isters and then, after int 0x80, explicitly copies the contents of EAX into the result variable.

"r" (or "=r" for output) can be used for an arbitrary register, i.e., the compiler will choose
a register as it sees fit. But then we don’t know which register will hold the value. We
can reference the variables by observing the order in which they appear in the output and
input operand lists, and then address them as %0, %1, ... in the assembler code.

For example, in order to add two variables var1 and var2 and store the sum in sum, we
could write
asm ("movl %1, %%eax \n"
     "addl %2, %%eax \n"
     "movl %%%eax, %0"
     : "=r" (sum) // output
     : "r" (var1), "r" (var2) // input
     : "%eax" // "clobber" list
);

We don’t know what registers will hold the values of var1 and var2, but we simply move or add them to EAX and finally write the sum (which is in EAX after the addl instruction) back to register 0. That is then copied to sum.

We do not want the compiler to use EAX for one of the variable values, and that is where the clobber list comes in: With it we can tell the compiler which registers it must not use, so in the case of the above addition we put EAX on that list. Note that the syntax for the register name is different in the instructions (%eax) and in the clobber list (%eax)! In the clobber list you could also write eax instead of %eax, but not %eax.

## B.5 Further Reading

A freely downloadable introductory text is Paul Carter’s “PC Assembly Language Book” [Car06]. It is available in English and a few other languages via the author’s website, http://www.drpaulcarter.com/pcasm/.

If you’re more interested in concepts of assembler programming than in the actual syntax, “The Art of Assembly Language” [Hyd10] is a good alternative introduction: The author, Randall Hyde, has created his own dialect of Intel 32-bit assembler called HLA (High Level Assembler) which looks more natural since it has commands for conditional loops and standard input/output. HLA programs have more similarity with C programs than normal assembler code, but writing programs in HLA still teaches all the basic principles of assembler.

“Linux Assembly Language Programming” [Nev00] by Bob Neveln focuses on Linux; for example there is a description of the ELF binary format used by Linux (and Ulx).

For gcc inline assembler the “GCC Inline Assembly How-to” [S03] is available online at http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html.
Other Educational Operating Systems

As mentioned in the introduction chapter, we have assembled a list of other educational operating systems. We already discussed Minix and Xinu (see page 38) and suggested having a look at these two systems. But there are many more:

• One of the oldest instructional texts in this area is the Lions’ commentary [Lio96], originally written by John Lions in 1976 but only published 20 years later. Its roughly 100 pages of code documentation are intended to be read side-by-side with a specially enumerated printout of the source code of Unix Version 6. Each section of the commentary explains a specific code section that is identified by its first line number.

• Xv6 is a simple Unix-like teaching OS [CKM12] originally developed in the summer of 2006 for MIT’s OS course, and its documentation mimics the Lions’ commentary in that the source code is available in a document with full line numbering (throughout all source files) and a descriptive document refers to those line numbers.

Our Ulix implementation has borrowed some code from Xv6, for example for dealing with the hard disk controller and with serial ports.

• Topsy (Teachable Operating System) was developed at ETH Zurich [FCZP95], the original version runs on the MIPS architecture. Later it was ported to Intel i386 [Ruf98] and to the Pentium 4 [Ryf07] by students of the same university.

• Thix is a Unix-like operating system developed by Tudor Hulubei and documented in a technical report [Hul95]. While the author’s goal was not to create an educational resource but “to learn about operating systems design and architecture, kernel algorithms, resource allocation, process scheduling, memory management policies, etc.”,
the resulting code and report make an interesting read.

The Ulix implementation uses parts of the floppy driver code from Thix.

- Nachos (Not another completely heuristic operating system) [CPA93, And30] was originally developed in Berkeley and is currently supported at University of Washington. It was written in C++ and the sources were well-commented, though classically (i.e., directly in the source files). The system has to be run on a specific MIPS hardware emulator (also provided by the authors). Development of the original code was stopped, but there is a Java version [HC30] now used in Berkeley. The concept of the authors is to provide a system with only the most basic properties which is then extended by students during a course. Another successor to Nachos is Pintos.

- The goal of the Pintos [Pfa10, Pin09] developers was to replace Nachos. It differs from it in several aspects: it runs on 32-bit Intel x86 hardware (the authors suggest to execute it in the Bochs [LDA+14] or qemu [B+14] PC emulators), and it uses the C programming language instead of C++. Its feature set is similar to the one of Ulix. Pintos is also similar to Unix systems, but does not provide a fork function. It is currently used for teaching at Stanford and other universities.

- OS/161 is a kernel that was written in C for classes at Harvard University. It runs on an emulator for a machine called System/161 and is available for download at http://www.eecs.harvard.edu/~syrah/os161/. The authors have described their experiences with using OS/161 in class in a short conference paper [HLS02].

- iPozix (spelled iPosix) is a small Unix-like kernel for 32-bit Intel machines that was written in C++ by two students of Oldenburg University [MT09]. Later a practical course with exercises based on iPozix was designed by another student [Phl10].

- L4 is a family of micro-kernels that is sometimes used for educational purposes, for example at Technical University of Dresden (in the “Building Microkernel-Based Operating Systems” course). Reference manuals for the x86 [Lie96] and MIPS architectures [EHL97] describe the system.

- FreeDOS is a clone of Microsoft’s MS-DOS, and its author has documented the development in a book [Vil96]. When he created FreeDOS, sources of MS-DOS were not available, so at that time studying FreeDOS was an alternative to reverse engineering the MS-DOS binaries. (In 2014 Microsoft published the source code of MS-DOS [Lev14, Shu14].)

- MicroC-OS II by Jean J. Labrosse is a real-time kernel that is described in a freely available book [Lab02]. The author states that the intended audience of the book includes “students interested in real-time operating systems”. There is also a newer version (MicroC-OS III) that runs on several platforms.

- OOSTuBS and MPStuBS are object-oriented operating systems for the Intel x86 platform which are used for classes at Friedrich Alexander University Erlangen-Nuremberg [Loh13]. MPStuBS is a version of OOSTuBS that supports multiprocessor systems. Students taking the Operating Systems course can decide which system they want to work with throughout the semester.
The following entries do not refer to educational operating systems, but to books which describe "real world" systems. However, they do it so thoroughly that these texts can also be used as learning materials.

- Marshall Kirk McKusick and George V. Neville-Neil give a very detailed description of the FreeBSD kernel in their book “The Design and Implementation of the FreeBSD Operating System” [MNN05]. There is a lot of code mixed with explanations and figures, but the code is only pseudo code that leaves out the details.

- The original Unix system is described in Bach’s book “The Design of the Unix Operating System” [Bac86]. It is based on Unix System V, Release 2 (from 1984) and describes the internal data structures and algorithms in a similar way as the FreeBSD book does: Real code is replaced with more accessible descriptions of what needs to be done.

- Lixiang et al. have written “The Art of Linux Kernel Design” [LWD+14]. Even though it was published in 2014, it is based on Linux version 0.11 which was released in December 1991. The old code is not as complex as that of current versions, and the authors use it for in-depth descriptions of the internal data structures and algorithms.
(example elf program test.asm 225) .................. 225
(function for adding, with pointer syntax 638a) ... 638a
(example for recursive u_open calls 413c) ....... 413c
(function implementations 100b) ............... 44a, 100b, 100c,
(example for signal handlers 568c) .......... 568c
(103a, 103b, 109c, 113b, 114a, 116e, 118b, 119b, 120a,
(example for syntax highlighting 29) ........... 29
(120b, 122d, 123d, 133b, 138c, 139e, 140a, 140b, 146a,
(example for system calls in linux 199) ....... 199
(146c, 151c, 162d, 163c, 165b, 166c, 168d, 170c, 170e,
(example function 635) ...................... 635
(171a, 171c, 172a, 173a, 183c, 184b, 184e, 185b, 185c,
(example minix super block 442c) .......... 442c
(186a, 186b, 187a, 188d, 189, 197a, 201b, 201d, 202b,
(example: mutual exclusion using global lock bit and
206b, 209c, 217a, 228b, 255a, 260a, 276d, 289a, 293d,
interrupt masking 353) ................... 353
(294, 296, 297, 319d, 323b, 324a, 324b, 327a, 328a, 328c,
(example: mutual exclusion using interrupt masking 352)
329b, 330a, 332b, 334a, 334b, 335b, 336b, 337b, 338c,
352)
(340e, 341, 342b, 344c, 345c, 361c, 362, 364b, 364c, 366a,
(example: nested critical sections 356) ....... 356
(366b, 366c, 367b, 368, 369c, 371a, 373c, 406, 408a, 409b,
(example structure 1 636a) .................. 636a
(409c, 412b, 412c, 414b, 415a, 418a, 418b, 419a, 419b,
(example structure 2 636b) .................. 636b
(420a, 420b, 421b, 421d, 422a, 422c, 424d, 424e, 425c,
(example: synchronization using Lock 355) .... 355
(432e, 440b, 495b, 496b, 496d, 497, 498a, 499a, 499d, 500,
(example: synchronization using Test-and-Set 354b) 354b
(505b, 506b, 507b, 507c, 509d, 510b, 512b, 512c, 516d,
(example: write() implementation 204b) .... 204b
(517b, 517c, 518d, 519d, 520c, 522e, 528c, 528e, 530c,
(example: write() prototype 204a) ......... 204a
(530d, 531b, 532d, 533b, 534b, 536b, 537a, 538b, 539c,
(external minix2 inode 442a) ................. 442a, 442b, 459a
(function implementation inactive) ......... 391a, 392a, 392b
(fault handler: display status information 152a) ... 151c,
391c, 152a, 290a
(152b) 151c, 152b, 290a, 291
(fault handler: terminate process 152b) ....... 151c, 152b, 290a,
1(fdc dma init 543a) .......................... 540c, 543a
(find a free frame and reserve it 118c) ... 118b, 118c, 119a
(find contiguous virtual memory range 120c) .... 120b, 120c
(find device and local path 408c) ......... 408a, 408c, 408d
(find free TCB entry 188a) ................... 188a, 188b, 188d
(first attempt for locking the keyboard buffer (in keyboard
(first attempt for locking the keyboard buffer (in
handler implementation) 381b) )381b
(syscall_readchar) 381a) .................. 381a
(format the disk image with minix fs 436b) .. 436b
(get frame address from page descriptor's integer repre-
(get password entry: fill target buffers 586a) ... 585a, 586a
(resentation 99) ................................ 99
(get password entry: read password file into passwords
and parse it 585b) ............... 585a, 585b
(free this inode 481a) ...................... 480c, 481a
(free this inode: (1) direct blocks 481b) ..... 481a, 481b
(free this inode: (1) direct blocks 481b) ..... 481a, 481b
(free this inode: (1) single indirect blocks 482a) ... 481a,
482a
(free this inode: (1) single indirect blocks 482a) ... 481a,
482a
(free this inode: (4) inode itself 483) .......... 481a, 483
(free this inode (3) double indirect blocks 482b) ... 481a,
482b
(free this inode (3) double indirect blocks 482b) ... 481a,
482b
(fseek on an empty minix filesystem 436d) ....... 436d
(function for adding) .................... 637c
lib-build/Makefile
interrupt descriptor table
install GDTs for User Mode
initialize system
initialize syscalls
initialize kernel global variables
initialize kernel global variables
initialize swap
initialize syscalls
install flat gdt
install GDTS for User Mode
install the fault handlers
install the interrupt descriptor table
install the interrupt handlers
install the timer
irq15 example
kernel main
keyboard handler: find active process, target_pid
keyboard handler implementation
keyboard handler: wake sleeping process
(lib-build/init.c) 191
(lib-build/Makefile) 622

(global variables) 92
(id: put process to sleep) 531
(id: read data from the controller) 530
(id: read sector on device) 527
(id: read/write sector on device) 526
(id: write data to the controller) 530
(id: write sector on device) 527
(identity map page i in kernel.pt) 106
(impossible way to build the sum)
)init to idle transformation
initialize filesystem
initialize kernel global variables
initialize system
install to idle transformation
impossible way to build the sum
old minix filesystem implementation
offset-test.c
mx_write
mx_open
mx_ftruncate
module.nw
lib-build/tools/tp.c
lib-build/tools/swapper.c
lib-build/tools/su.c
lib-build/process.ld
Linux mtab entry
Linux system calls
look at the image with hexdump
macro definitions
main prototype
Makefile
map page starting at 0x00000000 + PAGE_SIZE+fid to frame fid
MENULIST 86
minix filesystem implementation
minix open 420c
minix open 440b, 430b, 440b, 440a, 445b, 446, 447c, 448, 451a, 451b, 452a, 453b, 456, 457b, 461d, 462d, 463a, 464b, 467b, 468b, 468c, 469c, 470b, 473a, 474c, 476b, 478b, 479b, 480a, 480c, 484b, 484c, 484e, 487a, 488a, 489a, 490a, 490d, 491c, 492
modify process to execute signal handler
module.nw
more address_space entries
more ksl semaphore entries
more TCB entries
mx_ftruncate: free double indirect block
mx_ftruncate: free single indirect block
mx_open 464c
mx_open 464b, 464c, 464d, 465, 466c
mx_open 464c, 464d, 464d, 465, 466c
mx_open 464c
mx_open 464b, 464b, 464c, 464d, 465, 466b
mx_open 465b, 466b, 466c, 467b, 467c, 467d, 467f, 468b, 468c, 469c, 470b, 473a, 474c, 476b, 477c, 479b, 480a, 480c, 484b, 484c, 484e, 487a, 488a, 489a, 490a, 490d, 491c, 492
nestable begin critical section
nestable begin critical section (first version)
nestable end critical section
nestable end critical section (first version)
nestable end critical section (first version)
nestable end critical section (first version)
no string buffer overflow
offset-test.c
old minix filesystem implementation
page fault handler: check if page was paged out
page fault handler: check if page was paged out
page fault handler: check if page was paged out
null
activate_address_space: 170c, 190a, 605c
add: 186b
address_space: 161, 162b, 173a, 304a
address_space: 122c, 152a, 162b, 162d, 162e, 163c, 165b, 166c, 167a, 167b, 167c, 169a, 170c, 170e, 171a, 171c, 173a, 210a, 211a, 211c, 232c, 233c, 255b, 257b, 260a, 261, 279c, 289c, 291, 296, 297, 307a, 308c
addr_space_id: 122c, 158b, 158c, 162d, 163c, 166c, 168a, 168c, 168d, 169a, 170a, 170b, 170c, 170e, 171a, 171c, 172a, 188c, 188d, 190a, 210a, 308c
addto:
add_to_blocked_queue: 185c, 187a
add_to_front:
add_to_kstack_delete_list: 166c, 168d
add_to_ready_queue: 184b, 186b, 192d, 212, 257c, 364c, 563b
array_string:
AS_DELETE: 162a, 166c
AS_FREE: 162a, 162d, 169a, 171a, 307a
as_map_page_to_frame: 164c, 164d, 165a, 165b, 173a, 211a, 257c, 291
AS_USED: 122c, 162a, 162e, 163c, 308c
ata_init: 45c, 534b
atoi8: 595
atoi: 582a, 585a, 586a, 595
basename: 455a, 455b, 577b
BINARY_LOAD_ADDRESS: 159a, 163c, 190c, 192d
bindump:
binwrite:
block: 187a
BLOCKADDRESSES_PER_BLOCK: 472, 473a, 474a, 476b, 477c
blocked_queue: 183a, 183c, 185b, 185c, 186a, 186b, 187a, 218b, 323d, 360a, 362, 365a, 366c, 368, 391a, 522a, 529a, 544d, 606
BLOCKSIZE: 440a, 451a, 453b, 471a, 471c, 472, 473a, 475b, 475c, 476b, 480c, 484e, 496d, 497, 508c, 509d, 510b, 515a, 518b, 582a
boolean: 45e
buffer_cache: 509a, 509b, 509d, 510b, 511a, 511b, 512b, 512c
BUFFER_CACHED_SIZE: 508b, 509a, 509b, 509d, 510b, 511a, 511b, 512b, 512c
BUFFER_CLEAN: 506b, 507b, 510a
buffer_contain: 507b, 512c
BUFFER_DIRTY: 507c, 510a
buffer_entry: 508c, 509a
buffer_lock: 509a, 509b, 509d, 510b, 512b, 606
buffer_read: 506b, 506d
buffer_sync: 511b, 512a, 512b, 513a
buffer_write: 506b, 507b, 507c, 509c, 510b
BUF_READ: 515b, 517c, 518d, 520c, 522e
BUF_STAT_FINISHED: 515b, 518a, 518b, 521a
BUF_STAT_NEW: 515b, 516d
BUF_STAT_TRANSFER: 515b
BUF_WRITE: 515b, 517c, 518d, 520c, 522e
BUILDDATE: 35a, 337c, 605a
busy: 371b, 390a, 390b, 533b, 537b
byte: 45e
charptr:
chdir: 434c
check_access: 576d, 577c
CHECKGROUP: 576d, 577b, 579b
check_perms: 576d, 579a, 579c
CHECKUSER: 576d, 577b, 579b
CHKWORLD: 576d, 577b, 579b
chmod: 591b
chown: 591a, 591b
clear_frame: 113b, 119b
close: 429b, 467b, 585b
closefile:
clrscr: 331d
CMD_GET: 518b, 519a, 521a
CMDDLINE_LENGTH: 224c, 234b, 235a, 235b
CMD_NUMSEC: 519a
CMD_PUT: 518a, 519a
CMD_STAT: 519a
CMD_TERM: 519a
cnumbers:
complex: 636a, 636b, 636c, 636d, 637a, 637b, 637c, 637d, 638a
Compute:
context_t: 142a, 142b, 146a, 146b, 146c, 151c, 174b, 175, 201d, 206d, 209c, 213d, 216b, 219c, 221a, 222b, 223e, 224c, 234b, 255c, 258b, 260a, 276d, 282c, 289a, 299a, 310a, 319d, 328c, 331a, 332d, 342b, 370d, 372a, 372d, 416a, 416b, 426b, 433b, 493b, 513a, 519d, 532d, 546b, 565c, 566d, 583a, 587c, 587d, 590b, 610c, 610d
ESRC: 561c, 562b, 565c
euid: 487a, 565c, 573a, 573b, 576c, 576d, 577b, 577c, 578, 580c, 581, 582a, 583a, 587d, 588b
example_function:
exception_messages: 151a, 152a
execv: 191a, 235d, 235e
exit: 214, 217d, 218a, 513e
EXIT_MUTEX: 350, 353, 354b, 355, 390a, 390b
f: 623b
false: 46a
fault0: 147b, 148a, 149b
fault10: 147b, 148a, 149b
fault11: 147b, 148a, 149b
fault12: 147b, 148a, 149b
fault13: 147b, 148a, 149b
fault14: 147b, 148a, 149b
fault15: 147b, 148a, 149b
fault16: 147b, 148a, 149b
fault17: 147b, 148a, 149b
fault18: 147b, 148a, 149b
fault19: 147b, 148a, 149b
fault20: 147b, 148a, 149b
fault21: 147b, 148a, 149b
fault22: 147b, 148a, 149b
fault23: 147b, 148a, 149b
fault24: 147b, 148a, 149b
fault25: 147b, 148a, 149b
fault26: 147b, 148a, 149b
fault27: 147b, 148a, 149b
fault28: 147b, 148a, 149b
fault29: 147b, 148a, 149b
fault30: 147b, 148a, 149b
fault31: 147b, 148a, 149b
fault3: 147b, 148a, 149b
fault4: 147b, 148a, 149b
fault5: 147b, 148a, 149b
fault6: 147b, 148a, 149b
fault7: 147b, 148a, 149b
fault8: 147b, 148a, 149b
fault9: 147b, 148a, 148c, 149b
fault_common_stub: 149a, 150b
fault_handler: 150a, 150b, 151b, 151c
fault_macro_0: 149a, 149b
fault_macro_no0: 149a, 149b

faults: 148a, 148b, 607c
fdbuf: 538c, 543a, 549c, 550b
fdbuf_command: 538d, 539c, 549c, 550b
fdbuf_drive: 539b, 539c, 540c, 544a, 544b, 547d, 548b, 551b
fdbuf_results: 537a, 540d, 548b, 551a, 551b
fdbuf_head: 539b, 539c, 540c, 540d, 548b
fdbuf_init: 45c, 552b, 552c
fdbuf_lock: 547a, 547b, 549d, 549e, 552c
fdbuf_map_type: 552c, 553a, 553b
fdbuf_mode: 540c, 542b, 542c
fdbuf_need_reset: 536b, 536d, 537a, 539c, 540c, 540d, 546b, 547d, 548b, 551a, 551b
fdbuf_out: 536a, 536b, 540c, 540d, 542c, 548b, 551a, 551b
fdbuf_read_sector: 549c, 550d
fdbuf_recalibrate: 548b, 551b
fdbuf_reset: 539c, 550e, 551a
data_results: 537a, 537d, 540d, 548b, 551b
data_seek: 539c, 548a, 548b
data_sleep: 545a, 545b, 547d
data_ticks: 546c, 547a, 547d
data_ticks_till_motor_stops: 539c, 546c, 547a
data_timeout: 546b, 546c, 547a, 547d
data_timer: 546d, 546e, 547a
data_track: 539b, 539c, 540c, 540d, 548b
data_waits_interrupt: 546b, 546c, 547a, 547d
data_wakeup: 545d, 546a, 546b, 547a
data_write_sector: 549b, 550b, 550d
data: 499a, 541c, 544a, 544b, 547a, 549d, 551a, 552c
fdd_drive_name: 541c, 552c
fdds_in_use: 541c
fdd_type: 499a, 541c, 549d
FD_SECSIZE: 549c, 549e, 550a, 550b, 550d
fileblocktozone: 471c, 473a, 475c, 484e
fileexists: 576a, 576b
fill_gdt_entry: 109c, 110a, 194a, 197a
fill_idt_entry: 138b, 138c, 139b, 148b, 202e
fill_page_desc: 100c, 101a, 111b, 123b
fill_page_table_desc: 103b, 103c, 105b, 108, 116b
findZeroBitAndSet: 447b, 447c, 448
first_page: 120c, 121a, 121c
FLOPPY_CHANNEL: 543a, 543b
FLOPPY_CONTROLLER_BUSY: 537b
FLOPPY_CONTROLLER_ENABLE: 544a, 544b, 544c, 551a
FLOPPY_DIRECTION: 536b, 536c, 537b
FLOPPY_DMAINT_ENABLE: 544a, 544b, 544c, 551a
FLOPPY_DTL: 540c, 541a
flooply_handler: 546b, 552c
FLOPPY_MASTER: 536b, 536c, 537a, 537b
FLOPPY_NEW_BYTE: 537a, 537b
flooply_queue: 544d, 544e, 545b, 546a, 564c, 606
FLOPPY_READ: 539a, 543a, 549c
FLOPPY_RECALIBRATE: 551b, 552a
FLOPPY_SEEK: 548b, 549a
FLOPPY_SENSE: 540d, 548b, 549a, 551a, 551b
FLOPPY_SPEC2: 542c, 542d
FLOPPY_SPECIFY: 542c, 542d
FLOPPY_WRITE: 539a, 540d, 550b
FOLLOW_LINK: 411c, 420a
fork: 213f, 213g, 214
frameid: 118c, 119a
framenos:
free_a_frame: 310e, 311b, 513e
free_frames: 112b, 112e, 119a, 119b, 123c, 310a, 311b, 342b, 513e, 604b, 613c
front_of_blocked_queue: 185b, 364c
FS_DEV: 405b, 410a, 412c, 414b, 415a, 418a, 418b, 420b, 421d, 422a, 422c, 425c, 588b, 589a
FS_ERROR: 410a, 412c, 414b, 415a, 418a, 418b, 420b, 421d, 422a, 422c, 425c, 588b, 589a
FS_FAT: 410a, 412c, 414b, 415a, 418a, 418b, 420b, 421d, 422a, 422c, 425c, 588b, 589a
FS_MINIX: 405b, 410a, 412c, 414b, 415a, 418a, 418b, 419a, 419b, 420b, 421d, 422a, 425c, 588b, 589a
fs_names: 406, 410b, 492
ftable: 112c, 112d, 112e, 113b, 114a, 603
fttruncate: 429b
g: 236, 356, 405a, 408c, 409c, 607c, 162
gdt: 92b, 109c, 110a, 196a
gdt_entry: 91, 110a
gdt_flush: 110a, 110b, 111b, 116b
gdt_ptr: 92a, 92b
getcwd: 434b, 434c
gindent: 429b
get_dev_and_path: 408b, 408c, 411e, 419a, 419b, 588b, 589a
get_eip: 212, 213b
get_errno: 206b, 206d
get_free_address_space: 162c, 162d, 163c
get_free_frames: 310d, 310e, 311b, 513e
get_irqmask: 139f, 140a, 140b
get_new_lock: 306c, 310g, 367b, 369c, 509b, 516c, 530b, 552c
get_new_semaphore: 364b
getpid: 214, 223b, 311b, 513e, 568b
getppid: 214, 223b
getpwnam_r: 585a
getwuid_r: 585a
gets: 432a
gettid: 214, 223a, 223b
get_xy: 331d
gfd2pf: 424e, 426b
gid: 478b, 573a, 573b, 580c, 581, 582a, 583a, 584c, 587d
gp: 92b, 110a, 110b, 608a, 608b
GUI:
handler1:
handler2:
harddisk_queue: 529a, 529b, 531a, 532d, 564c, 606
hash: 304a, 304b, 306e, 306f, 307b, 308c
hd_buf: 530a, 530c, 530d, 532b, 532d
hd_direction: 530a, 530c, 530d, 532d
hd_lock: 530a, 530b, 530c, 530d
HD_OP_NONE: 529d, 530c, 530d, 532d
HD_OP_READ: 529d, 530c, 532d
HD_OP_WRITE: 529d, 530d, 532d
HD_SECSIZE: 529d, 530a, 530c, 530d, 532b, 532d
hd_size: 499a, 534a, 534b
hexdump: 290a, 436c, 437, 603, 605c, 607a, 608a, 608b, 611b, 612b, 612c
higherhalf: 94
hour: 339d, 340b, 340c, 340e, 341, 343b, 605a
HZ: 539c, 540a, 547a
IDE_BSY: 525a, 533b
IDE_CMD_IDENT: 525a, 534b
IDE_CMD_READ: 525a, 527b
IDE_CMD_WRITE: 525a, 527c
IDE_DF: 525a, 533b
IDE_DROPO: 525a, 533b
IDE_ERR: 525a, 533b
ide_handler: 532c, 532d, 534b
iedevid: 526, 532a, 533a, 533b
ide: 282e, 282f, 282g
idt: 138a, 138c, 146d
idt_entry: 137a, 138a, 146d
idt_load: 146d, 146f, 147a
idtp: 138a, 146d, 147a
idt_ptr: 137b, 138a
if_nested_level: 357a, 357b, 357c, 357d, 357e
if_state: 357d, 357e, 383b, 384a, 384b
iii:
inb_delay: 536b, 537a, 538b, 552c
INDEX_FROM_BT: 113a, 113b, 114a
initialize_blocked_queue: 183b, 183c, 218c, 323e, 363d, 364b, 367b, 522b, 529b, 544e
initialize_module: 44b, 45a

importb: 133b, 140a, 320b, 336b, 339d, 344c, 345c, 519d, 532a, 532b, 533b, 534b
install: 133b
install_interrupt_handler: 146b, 146c, 323b, 339a, 520a, 534b, 552c
install_syscall_handler: 173d, 201b, 201c, 206f, 213e, 217c, 220b, 221c, 222e, 224a, 224e, 235c, 259a, 260c, 282d, 299b, 310c, 328e, 331b, 333a, 370e, 372b, 373a, 416c, 428a, 434a, 493d, 513b, 565a, 567a, 583b, 587e, 590c, 611a
intarrcp: 643a
interrupt_handlers: 145b, 146a, 146c
interrupt: 145b
int_minix2_inode: 459a, 459c, 460a, 464d, 467b, 468b, 468c, 469c, 470c, 471b, 473a, 475a, 476b, 484e, 607b
intptr: 643b
INVALID.Track: 541c, 542a, 551b
I_CLOCK_CHANNEL: 338c, 338d
I_CLOCK_COMMAND: 338c, 338d
I CMOS_CMD: 339b, 339d, 552c
I CMOS_DATA: 339b, 339d, 552c
I COM1: 336b, 344a, 344c
I COM2: 344a, 344c, 345c, 519d
I DMA_ADDR_2: 542e, 543a
I DMA_COUNT_2: 542e, 543a
I DMA_PORTFP: 542e, 543a
I DMA_INIT: 542e, 543a
I DMA_MODE: 542e, 543a
I DMA_PAGE_2: 542e, 543a
I FLOPPY_COMMAND: 535, 536b, 537a
I FLOPPY_OUTPUT: 535, 544a, 544b, 551a, 552c
I FLOPPY_RATE: 535, 542c, 552c
I FLOPPY_STATUS: 535, 536b, 537a
I IDE_COMMAND: 525b, 527b, 527c, 534b
I IDE_DATA: 525b, 532b, 532d, 534b
I IDE_DEVCTRL: 525b, 526
I IDE_DISKSEL: 525b, 526, 534b
I IDE_SEC_COUNT: 525b, 526
I IDE_SECTOR: 525b, 527a
syscall_link
syscall_get_xy
syscall_gettid
syscall_getcwd
syscall_clrscr
syscall_chdir
syscall4
syscall2
symlink

syscall_symlink
syscall_setpsname
syscall_readlink
syscall_query_ids
syscall_pthread_mutex_trylock
syscall_pthread_mutex_init
syscall_login
syscall_truncate
syscall_unlink
syscall_waitpid
syscall_write
syscall_write_screen
syscall_write sector

swapper_lock: 310a, 310f, 310g, 311a
syscall: 429b
sync: 513c, 513d
__sync_add_and_fetch: 391a, 391b
__sync_lock_test_and_set: 354a
__sync_sub_and_fetch: 391a, 391b
syscall1: 203c, 207b, 213g, 221f, 223b, 260e, 282f, 310e, 331d, 513d
syscall2: 174d, 203c, 218a, 224f, 259c, 299d, 328g, 333c, 373d, 429b, 434c, 493f, 584c, 587b
syscall3: 203c, 224f, 235e, 331d, 373e, 429b, 434c, 568b, 584c, 591b
syscall4: 203a, 203b, 220d, 429b, 591b
syscall_chdir: 433b, 434a
syscall_chmod: 509b, 590c
syscall_chown: 509a, 590b, 590c
syscall_close: 426b, 428a
syscall_clscr: 331a, 331b
syscall_diskfree: 493a, 493b, 493d
syscall_execve: 234a, 234b, 235c
syscall_exit: 152b, 166c, 216a, 216b, 217c, 260a
syscall_fork: 213c, 213d, 213e
syscall_free_a_frame: 310a, 310c
syscall_ftruncate: 426b, 428a
syscall_getcwd: 433a, 433b, 434a
syscall_getdents: 426b, 428a
syscall_get_errno: 206d, 206f
syscall_get_free_frames: 309, 310a, 310c
syscall_getgid: 222b, 222e
syscall_getpid: 222b, 222e
syscall_getent: 223d, 223e, 224a
syscall_gettid: 222a, 222b, 222e
syscall_getxy: 331a, 331b
syscall: 202c, 202d, 202e
syscall_handler: 201a, 201b, 201d, 202c
syscall_idle: 282a, 282c, 282d
syscall_isatty: 426b, 428a
syscall_kill: 565a, 565b, 565c
syscall_link: 426b, 428a
syscall_login: 583a, 583b
syscall_lseek: 426b, 428a
syscall_mkdir: 426b, 428a
syscall_open: 426b, 428a
syscall_page out: 298c, 299a, 299b
syscall_print inode: 610c, 610d, 611a
syscall_pthread_create: 258a, 258b, 259a
syscall_pthread_exit: 259d, 260a, 260c
syscall_pthread_mutex_destroy: 372c, 372d, 373a
syscall_pthread_mutex_init: 370c, 370d, 370e
syscall_pthread_mutex_lock: 372a, 372b
syscall_pthread_mutex_trylock: 372a, 372b
syscall_pthread_mutex_unlock: 371d, 372a, 372b
syscall_query ids: 587c, 587d, 587e
syscall_read: 426b, 428a
syscall_readchar: 416a, 416b, 416c
syscall_readlink: 426b, 428a
syscall_read screen: 332d, 333a
syscall_read sector: 426b
syscall_resign: 219c, 220e, 221a, 221c
syscall_rmdir: 426b, 428a
syscall_sbrk: 173b, 173d, 174b
syscall_setegid: 583a, 583b
syscall_set_errno: 206c, 206d, 206f
syscall_setuid: 583a, 583b
syscall_setgid: 583a, 583b
syscall_setpsname: 224b, 224c, 224e
syscall_setterm: 328c, 328e
syscall_setuid: 582b, 583a, 583b
syscall_setxy: 330b, 331a, 331b
syscall_signal: 566c, 566d, 567a
syscall_stat: 426b, 428a
syscall_systime: 426b, 428a
syscall_sync: 512d, 513a, 513b
syscall_table: 200b, 201b, 201d
syscall_truncate: 426b, 428a
syscall_unlink: 426b, 428a
syscall_waitpid: 219b, 219c, 220a, 220b
syscall_write: 426a, 426b, 428a
call_write: 426a, 426b, 428a
call_write_screen: 332d, 333a
call_write sector: 426b
_sys_stack: 94, 95a
system_kbd: 318d
SYSTEM_KBD_BUFFLEN: 318a, 318b, 321b, 324a, 416b
system_kbd_count: 318d, 318e, 610a
system_kbd_lastread: 318d, 318e, 610a
system_kbd_pos: 318d, 318e, 610a
system_start_time: 339c, 340b, 342c
system_ticks: 306d, 311a, 338a, 342c, 342d, 343b
system_time: 338a, 342c, 343b, 475c, 478b, 605a
target_pid: 321a, 322b, 565c
TCB: 175, 176b, 188b, 188d, 190a, 210a, 210b, 223e, 224f, 255c, 260a, 276c, 280a, 291, 562b, 580c, 581, 582a
term_buffer: 325b, 326a, 334b, 335b
TERMINALS: 318a, 318c, 324a
terminals: 318c, 318d, 318e, 321b, 324a, 326c, 381b, 416b
terminal_t: 318b, 318c, 321b, 324a, 416b
TEST BITS: 540d, 548b, 548c, 551b
test_frame: 114a, 114b, 118c, 119b, 613c
testvar: 118a
textmemptr: 116c, 329b, 335b, 609
thenin: 305b, 308c
thread_id: 175, 178a, 178c, 181, 183a, 184a, 184b, 184c, 185a, 185b, 185c, 186a, 186b, 187c, 188a, 190a, 192c, 209c, 210a, 216b, 255a, 255b, 278a, 281, 322a, 362, 364c, 366c, 368, 391a, 424d, 424e, 426b, 546a
thread_table: 152b, 176b, 184b, 184c, 184d, 185c, 186a, 186b,
TSTATE_WAITKEY: 180a, 188a, 188d, 190a, 206b, 210b, 216b, 217a, 217b, 219c, 220a, 222c, 223e, 224c, 234b, 255c, 260a, 277b, 278a, 278b, 281, 322a, 322b, 324a, 328c, 329b, 330a, 332b, 334b, 335b, 36b, 369c, 371a, 381a, 383a, 412b, 416b, 424d, 424e, 426b, 432e, 478b, 487a, 562b, 564b, 565c, 566b, 573b, 577c, 580c, 581, 582a, 587d, 588b, 605d, 606

TSTATE_WAITFOR: 338b, 338c, 339a

tmppuf:
t_new: 210b, 212, 255c, 257c, 276c, 277b, 278b, 279c, 280a, 280b, 425a, 567b, 567c
t_old: 210b, 255c, 257c, 276c, 277b, 278b, 279c, 425a

TOP_OF_KERNEL_MODE_STACK: 159c, 192b, 196a, 211b, 257b, 261, 280a

TOP_OF_THREAD_KSTACK: 255d, 257c, 280a

TOP_OF_USER_MODE_STACK: 152a, 159b, 165a, 167b, 192d, 231, 289c, 291

totalfree: 613c

TRANSFER_OK: 540d, 541a

TRICKGD: 93, 94

TRUE: 29

true: 46a

truncate: 429b

tss_entry: 195, 197a

tss_entry_struct: 194b, 195

tss_flush: 110a, 197c, 197d, 280a

TSTATE_EXIT: 180a, 216b, 217a, 260a, 281, 564a

TSTATE_FORK: 180a, 210b, 255c

TSTATE_LOCKED: 180a, 361c, 366a, 391a, 392a, 392b

TSTATE_READY: 180a, 184b, 186b, 278a, 278b, 563b, 564c

TSTATE_STOPPED: 180a, 563a, 563b

TSTATE_WAITFLP: 180a, 545b, 564c

TSTATE_WAITFOR: 180a, 217a, 219c, 281, 564c

TSTATE_WAITHD: 180a, 531a, 564c

TSTATE_WAITKEY: 180a, 416b, 564c

TSTATE_WAITSC: 521b, 521c, 546c

TSTATE_ZOMBIE: 152b, 180a, 217a, 281

uart: 336b, 344b, 344c, 345c, 519d

uartinit: 344c, 345a, 345d

uart2putc: 345b, 345c, 517b, 518a, 518b, 521a

uartputc: 335b, 336a, 336b, 598a

uchdir: 432d, 432e, 433b, 488a, 582a

uchmod: 589a, 590b

uchown: 588a, 588b, 590b

uclose: 190c, 216b, 229a, 233b, 418a, 420a, 420c, 426b, 488a, 582a

uexec: 228a, 228b, 234b

ufork: 188d, 209a, 209c, 213d

utftruncate: 420a, 420b, 426b

ugetcwd: 432d, 432e, 433b, 488a

ugetdents: 422b, 422c, 426b

uid: 478b, 573a, 573b, 580b, 580c, 581, 582a, 583a, 584b, 584c, 584e, 585a, 587d

uint: 46b

uint6_t: 46b

uint32_t: 46b

uint64_t: 46b

ukill: 321a, 561b, 562b, 565c

u_link: 419a, 426b

uxlib_printchar: 598b, 598c

u_login: 582a, 583a

ulong: 46b, 109b, 109c, 340e, 341

ulonglong: 46b, 534a, 534b

u_lseek: 233b, 293b, 293d, 294, 418a, 426b

umain: 191a

u_mapped: 101a, 165b

u_mappd: 103c, 166a

u_mkdir: 422a, 426b

uname: 35a, 337c, 605a, 609, 610a

uinittime: 340b, 340d, 340e

unlink: 429b

u_open: 190c, 229a, 293b, 411b, 411c, 412c, 420a, 420c, 426b, 488a, 582a

UPDATE_BUF: 507a, 507b

u_pthread_create: 254b, 255a, 258b

u_pthread_mutex_destroy: 372d, 373b, 373c

u_pthread_mutex_init: 369b, 369c, 370d

u_pthread_mutex_lock: 371a, 372a

u_pthread_mutex_trylock: 371a, 372a

u_pthread_mutex_unlock: 370f, 371a, 372a

u_read: 190c, 229a, 233b, 294, 413c, 414b, 420c, 426b, 582a

ureadchar: 432b

ureadline: 214, 430, 431, 432a, 586b

u_readlink: 420c, 426b

u_reopen: 425a, 425b, 425c

u_rmmdir: 421e, 422a, 426b

u_sbrk: 172b, 173a, 173c, 174b, 233c, 257c

u_seteuid: 581, 583a

u_setuid: 581, 583a

u_setgid: 580c, 583a

u_setgid: 580c, 583a

u_signal: 566a, 566b, 566d

ustack: 257c

u_stat: 413b, 420c, 421c, 421d, 426b, 432e, 576b, 576d, 577b, 577c

u_symlink: 419b, 426b

u_truncate: 420a, 426b

u_unlink: 418b, 426b, 488a

u_write: 293d, 414a, 415a, 426b

video_pt: 111a, 111b

VIDEORAM: 116b, 116c, 327a, 327b, 332b, 334b, 337b, 342d

virt: 95b

vt: 326a, 326c, 327a, 328c, 328f, 328g, 329b, 330a, 332b, 334b, 335b

vt_activate: 321a, 326e, 327a

VT_BLUE_BACKGROUND: 326b, 329b, 609

vt_escr: 329b, 331a, 337c, 608b

vt_get_xy: 330a, 331a

VT_HEIGHT: 325a, 334a, 334b

vt_move_cursor: 321a, 327a, 327d, 328a, 329b, 330a, 335b

VT_NORMAL_BACKGROUND: 326b, 326c, 329b, 333e, 334a, 335b
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_RED_BACKGROUND</td>
<td>326b, 609</td>
</tr>
<tr>
<td>vt_scroll</td>
<td>334b, 335b</td>
</tr>
<tr>
<td>vt_scroll_mem</td>
<td>334a, 334b</td>
</tr>
<tr>
<td>vt_set_xy</td>
<td>329a, 329b, 330a, 331a</td>
</tr>
<tr>
<td>VT_SIZE</td>
<td>325a, 325b, 326c, 327a, 329b, 332b, 337b</td>
</tr>
<tr>
<td>VT_WIDTH</td>
<td>325a, 329b, 334a</td>
</tr>
<tr>
<td>wait_fdc_interrupt</td>
<td>540d, 547c, 547d, 548b, 551a, 551b</td>
</tr>
<tr>
<td>waitpid</td>
<td>180a, 220c, 220d, 606</td>
</tr>
<tr>
<td>waitpid_queue</td>
<td>217a, 218b, 218c, 219c, 281, 564c, 606</td>
</tr>
<tr>
<td>wait_semaphore</td>
<td>361b, 361c, 391a</td>
</tr>
<tr>
<td>wake_waiting_parent_process</td>
<td>216b, 216c, 217a, 564a</td>
</tr>
<tr>
<td>waste_time</td>
<td>568c</td>
</tr>
<tr>
<td>word</td>
<td>45e</td>
</tr>
<tr>
<td>write</td>
<td>35b, 123b, 170c, 199, 204a, 204b, 213b, 429a, 429b, 431, 460b, 475a, 525a, 528a, 539c, 543a, 575, 598c, 624</td>
</tr>
<tr>
<td>writeblock</td>
<td>445b, 448, 451a, 453b, 454a, 475c, 477b, 477c, 497, 504, 507c</td>
</tr>
<tr>
<td>writeblock_fd</td>
<td>507b, 550c, 550d</td>
</tr>
<tr>
<td>writeblock_hd</td>
<td>507b, 529c, 531b</td>
</tr>
<tr>
<td>writeblock_nb_serial</td>
<td>518c, 518d</td>
</tr>
<tr>
<td>writeblock_raw</td>
<td>507b, 512b</td>
</tr>
<tr>
<td>writeblock_serial</td>
<td>507b, 522d, 522e</td>
</tr>
<tr>
<td>WRITE_CR3</td>
<td>279a, 279b, 279c</td>
</tr>
<tr>
<td>WRITE_PROTECTED</td>
<td>540d, 541a</td>
</tr>
<tr>
<td>write_screen</td>
<td>332a, 332b, 333b, 333c, 333e</td>
</tr>
<tr>
<td>writesector</td>
<td>523</td>
</tr>
<tr>
<td>writesector_hd</td>
<td>530d, 531b</td>
</tr>
<tr>
<td>write_swap_page</td>
<td>293c, 293d, 296</td>
</tr>
<tr>
<td>write_tss</td>
<td>196a, 196b, 197a, 197b, 280a</td>
</tr>
<tr>
<td>yearlength</td>
<td>341</td>
</tr>
</tbody>
</table>
Index

Symbols

+ ≡ (continuation of chunk) ........................................ 20
⟨ (begin chunk name; output version) ............................. 19
⟩ (end chunk name; output version) .............................. 19
@ (end of code chunk) ............................................. 20
>> (end chunk name; input version) .............................. 20
<< (begin chunk name; input version) ........................... 20
Øb prefix ............................................................ 25
Øx prefix ............................................................ 24

A

abort ................................................................. 556
access control ..................................................... 50
access permissions (files) ................................. 457, 571, 574
address decoder logic ........................................... 64
address relocation ............................................... 62
address space ...................................................... 63
creation ............................................................. 163
destruction ......................................................... 166
enlargement ........................................................ 172
extra entries for multiple kernel stacks .............. 257
for forked process ................................................. 210
getting rid of thread stacks ................................. 261
implementation .................................................. 158
organization ......................................................... 66
shared by threads of the same process ............... 254
switch between .................................................... 169
Ulix ................................................................. 84
address space descriptor ................................. 158
address space table ...................................... 158, 161
address translation ............................................ 63, 64
multi-level address translation .......................... 77
Amstrad CPC home computer ............................. 117
application .......................................................... 31, 628
Assembler language .... 31, 36, 43, 46, 86, 110, 124, 202
arithmetic operations ......................................... 650
AT&T syntax ....................................................... 31, 47

BareMetal operating system ............................ 31
cli and sti instructions ........................................ 47
db, dw, dd (data storage) .................................... 652
disassembler ....................................................... 240
equ (constants) ................................................. 653
generated code (from C) .................................... 629
get EIP register .................................................... 213
hlt instruction ..................................................... 281
inline assembler ............................................... 133, 654
int 0x80 instruction ................................. 173, 203
Intel syntax ......................................................... 31, 47
interrupt handler ................................................. 138
introduction ....................................................... 647
iret instruction .................................................... 288
jumps and calls .................................................... 651
load the GDT ..................................................... 94, 110
load the IDT ....................................................... 146
load the GDT ..................................................... 197
macros .............................................................. 653
mov instruction .................................................... 649
nasm assembler ............................................... 86, 616
ports .............................................................. 133, 528
pre-processor ..................................................... 46, 624
registers ............................................................ 647
stack usage ........................................................ 652
associative memory ........................................... 80
AT&T syntax (Assembler) .................................. 31, 47
atomic ............................................................... 178, 348, 351–354, 392
average interaction time ...................................... 267

B

bad TSS fault .................................................... 150
BareMetal operating system ............................ 31
base address ....................................................... 58
basename (of a path) ............................................ 454
Basic programming language ............................. 117
BCD (binary-coded decimal) ................................ 339
best-fit allocation .................................. 55
binary numbers .................................. 25
bindump ........................................... 630
BIOS ............................................. 86
bitmap ............................................. 54
   inode bitmap .................................. 442, 444
   zone bitmap .................................. 442, 444
block device .................................... 503
block read/write
   buffer cache .................................. 509
   floppy disk .................................. 550
   generic, with buffer cache ................. 506
   generic, without buffer cache ............. 507
   hard disk .................................... 529
   serial disk .................................. 522
   synchronize buffer cache .................... 512
blocked (thread state) .......................... 177, 180
blocked queue ................................. 177, 179, 181, 360
   floppy disk .................................. 544
   for kernel locks .............................. 365
   for kernel semaphores ....................... 360
   hard disk .................................... 529
   implementation .............................. 185
   keyboard .................................... 323
   process waits for child termination ......... 218
   serial disk .................................. 521
blocked signal .................................. 561
Bochs PC emulator ............................... 616
boot loader ..................................... 41
boot process ................................... 85
bottom half ..................................... 386
brk system call ................................ 172
Buddy system .................................... 55
buffer cache .................................... 508
   block read/write functions ................. 509
   kernel lock .................................. 509
   synchronization ............................. 512
build process .................................. 615
busy waiting .................................... 355, 358

C
C function: prototype vs. implementation .......... 28
C programming language ......................... 23, 30
   array ........................................ 636
   buffer overflow ............................. 640
   clear C ...................................... 31
   comments .................................... 29
   conditional compilation ..................... 644
   data type .................................... 636
   header files ................................ 31
   implementation ............................. 28, 43
   inline assembler ............................ 133
   introduction ................................ 635
   macro ........................................ 644
   pointer ...................................... 638
   pointer arithmetic .......................... 642
   pre-processor ................................ 643
   prototype .................................... 28, 43
   string ........................................ 638
cache ........................................... 72
   cache coherence problem ...................... 72, 271
   cache miss .................................. 73
   card reader .................................. 394
cascading PIC .................................. 131
CD-ROM filesystem ISO-9660 ..................... 395
character device ................................ 503
chdir system call ................................ 433
child process .................................. 208
chip select ..................................... 64
chmod system call ................................ 590
chown system call ................................ 590
c li ............................................. 47, 129
clock algorithm (page replacement strategy) .... 303
clock chip ..................................... 338
ctone function (Linux) .......................... 254
close system call ................................ 426
closing a file .................................. 418, 467, 496
cluster ......................................... 398, 399
cluster run (NTFS) ................................ 399
CMOS chip ...................................... 339
code chunk ..................................... 19, 26
   chunk name .................................. 26
   chunk number ................................ 27
   continuation ................................. 20
   defined identifiers ......................... 28
   next/previous continuation ................. 27
   root chunk .................................. 27
   used by ....................................... 28
code region .................................... 65, 159
code segment (kernel mode) ...................... 110
code segment (user mode) ....................... 194
coding standard ................................ 28
Commodore C64 home computer ................... 117
compaction ..................................... 60
comparison operators ............................ 29
competitive interaction pattern .................. 347
concurrent programming ......................... 246
context ......................................... 141, 174
context switch ........................................... 169, 249, 252, 272
  losing values on the stack .......................... 276
  store new page directory address in CR3 ........ 278
  update TSS ............................................ 280
contiguous allocation .................................. 51, 120
contiguous filesystem .................................. 395
continuation (of a code chunk) ...................... 20
cooperative interaction model ....................... 347
coprocessor not available fault .................... 150
CP/M operating system ............................... 190, 397, 400
  filename convention ................................ 398
CPU affinity ............................................ 271
CPU burst ................................................ 250
CPU usage .............................................. 264
CR0 register ............................................ 109
CR3 register ............................................ 97, 109, 278
critical section ........................................ 183, 348, 349, 374, 377
critical section
  nested .................................................. 356
CTWILL ................................................... 23
CWEB .................................................... 23

D
data block ............................................... 442
data region ............................................. 65, 159
data segment (kernel mode) ......................... 110
data segment (user mode) ............................ 194
data stream (NTFS) .................................... 400
date (CMOS chip) ....................................... 339
Debian Linux (Ulix development) ................... 615
debugging ............................................... 603
defragmentation ........................................ 60
deleting a file .......................................... 480
descriptor privilege level ............................. 91, 136, 193
device filesystem ....................................... 494
df program ............................................. 491
Dijkstra, Edsger W. .................................... 358
direct memory access (DMA) .......................... see DMA
directory ............................................... 395, 421, 452, 486, 494
get entries ............................................. 422, 490
make ...................................................... 422, 486
remove ................................................... 422, 487
directory name (of a path) ............................ 454
dirty bit (D bit), page descriptor ................. 72, 100, 305
dirty page .............................................. 305
disable interrupts ..................................... 352
disassembler .......................................... 240
diskfree system call ................................... 493
dispatcher .............................................. 178, 186, 360
distributed programming .............................. 247
divide by zero fault ................................... 150
DMA (direct memory access) ......................... 504, 514, 525, 527, 535
  address of DMA buffer .............................. 538
  DMA controller ....................................... 542
  set up the transfer .................................. 542
double fault ............................................ 150
double indirection ..................................... 400, 473, 477, 482
DPL (descriptor privilege level) .................... 91, 136, 193
drive letter ............................................ 400
dynamic data ........................................... 65
dynamic partitioning .................................. 53

E
eax_return macro ...................................... 174
effective group ID ..................................... 573
effective user ID ....................................... 573
EFLAGS register ....................................... 152
EIP register ............................................ 198, 212, 273, 567
ELF (executable and linking format) .............. 86, 225
  execv function ..................................... 228
  file format type declarations .................... 227
  linker configuration file ......................... 236
  loader ............................................... 225
  program header ................................... 232
ELFAGS register ....................................... 142
enable an interrupt ................................... 140
dead of the list ....................................... 183
defat protocol ........................................ 350
descriptor privilege level ............................ 148, 360
epoch (Unix time) ..................................... 340
error code ............................................. 198, 212, 273, 567
executable and linking format (ELF) ........................... see ELF
execv system call ................................... 234
exit protocol .......................................... 350
exit system call ...................................... 216
Ext3 filesystem ....................................... 413
external fragmentation ................................ 60

F
FAT filesystem ........................................... 398
fault ..................................................... 42, 147
  error code .......................................... 148
fault handler .......................................... 174
  implementation ..................................... 151
  page fault handler ................................. see page fault handler
signal .................................................. see signal
terminate process ..................................... 152
FIFO page replacement strategy .................... 302
<table>
<thead>
<tr>
<th>File</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>change owner, group or permissions</td>
<td>587</td>
</tr>
<tr>
<td>close</td>
<td>418, 467, 496</td>
</tr>
<tr>
<td>delete</td>
<td>see file → unlink</td>
</tr>
<tr>
<td>hard link</td>
<td>419, 479</td>
</tr>
<tr>
<td>open</td>
<td>411, 464, 495</td>
</tr>
<tr>
<td>read</td>
<td>414, 470, 496</td>
</tr>
<tr>
<td>seek</td>
<td>418, 468, 498</td>
</tr>
<tr>
<td>soft link</td>
<td>419, 483</td>
</tr>
<tr>
<td>sparse</td>
<td>501</td>
</tr>
<tr>
<td>status</td>
<td>460, 489, 494, 499</td>
</tr>
<tr>
<td>status list</td>
<td>461</td>
</tr>
<tr>
<td>truncate</td>
<td>420, 484</td>
</tr>
<tr>
<td>unlink</td>
<td>480</td>
</tr>
<tr>
<td>write</td>
<td>415, 474, 497</td>
</tr>
<tr>
<td>File Allocation Table</td>
<td>398</td>
</tr>
<tr>
<td>File descriptor</td>
<td>conversion process ↔ global descriptors</td>
</tr>
<tr>
<td>global</td>
<td>410</td>
</tr>
<tr>
<td>inside a process</td>
<td>410, 424</td>
</tr>
<tr>
<td>local</td>
<td>410, 459</td>
</tr>
<tr>
<td>Filesystem</td>
<td>contents of the Ulix root disk</td>
</tr>
<tr>
<td>contiguous</td>
<td>395</td>
</tr>
<tr>
<td>CP/M</td>
<td>397</td>
</tr>
<tr>
<td>device filesystem</td>
<td>494</td>
</tr>
<tr>
<td>Ext3</td>
<td>413</td>
</tr>
<tr>
<td>FAT</td>
<td>398</td>
</tr>
<tr>
<td>HPFS</td>
<td>399</td>
</tr>
<tr>
<td>Implementation in Ulix</td>
<td>403</td>
</tr>
<tr>
<td>ISO-9660, CD-ROM</td>
<td>395</td>
</tr>
<tr>
<td>Layered Design</td>
<td>403</td>
</tr>
<tr>
<td>Library Functions</td>
<td>429</td>
</tr>
<tr>
<td>Logical Filesystem</td>
<td>403</td>
</tr>
<tr>
<td>Minix</td>
<td>403, 435</td>
</tr>
<tr>
<td>implementation in Ulix</td>
<td>439</td>
</tr>
<tr>
<td>Non-contiguous</td>
<td>396</td>
</tr>
<tr>
<td>NTFS</td>
<td>399</td>
</tr>
<tr>
<td>System Calls</td>
<td>423</td>
</tr>
<tr>
<td>Unix</td>
<td>400</td>
</tr>
<tr>
<td>Virtual Filesystem</td>
<td>403</td>
</tr>
<tr>
<td>Working Directory</td>
<td>432</td>
</tr>
<tr>
<td>First-come first-served scheduler</td>
<td>265</td>
</tr>
<tr>
<td>First-fit allocation</td>
<td>54</td>
</tr>
<tr>
<td>Fixed equal size partitioning</td>
<td>51</td>
</tr>
<tr>
<td>Fixed variable size partitioning</td>
<td>53</td>
</tr>
<tr>
<td>Floating point fault</td>
<td>150</td>
</tr>
<tr>
<td>Floppy controller</td>
<td>131</td>
</tr>
<tr>
<td>Floppy Disk Interrupt Handler</td>
<td>130, 546</td>
</tr>
</tbody>
</table>

**f**

- fork operation | 207
- duplicate file descriptors | 425
- fork system call | 213
- fragmentation | 60
- compaction | 60
- defragmentation | 60
- external | 60
- internal | 60
- frame | 69
- allocation | 118
- freeing a frame | 308
- list of free frames | 111
- frame table | 111
- framebuffer (VGA) | 111
- free frame list | 53
- FreeBSD operating system | 17
- kernel | 659
- signal numbers | 559
- Ulx development | 615
- FreeBSD operating system | 658
- frequency (timer chip) | 338
- f*ck_minix | 435
- ftruncate system call | 426

**G**

- gang scheduling | 271
- GByte | 25
- GDT (global descriptor table) | 90, 193
  - code segment (kernel mode) entry | 110
  - code segment (user mode) entry | 194
  - data segment (kernel mode) entry | 110
  - data segment (user mode) entry | 194
- l*get | 94, 110
- overview of entries | 197
- TSS (task state segment) entry | 196
- general protection fault | 150
- get_erro system call | 206
- get_free_frames system call | 309
- getcwd system call | 433
- getdent system call | 426
- getpid system call | 222
- getppid system call | 222
- getpsinfo system call | 223
- gettid system call | 222
- g*ibyte | 25
- gigabyte | 25
global descriptor table .................................. see GDT
global file descriptor .................................. 410
GNU C compiler ........................................... 615
GPL (GNU General Public License) ..................... 24

H
hard disk interrupt handler ............................. 130, 532
hard link ................................................. 419, 479
hardware ................................................. 313
  CMOS chip ............................................. 339
  floppy disk ............................................ 535
  hard disk ............................................. 395, 525
  IDE (hard disk) controller .......................... 525
  keyboard .............................................. 313
  magnetic tape drive .................................. 394
  serial console ........................................ 336
  serial hard disk ..................................... 514
  serial port ......................................... 343
  terminal .............................................. 325
  timer ................................................. 338
  VGA graphics adapter ............................... 325
  virtual console ...................................... 317
hash ....................................................... 304, 306
heap ....................................................... 65, 159
Heisenbug ................................................ 379
hexadecimal numbers ................................... 24
hierarchic page tables ................................ 75
higher half trick ....................................... 93, 108
HPFS filesystem ........................................ 399
Hulubei, Tudor .......................................... 553

I
I/O burst .................................................. 250
I/O port .................................................... 132
IDE controller .......................................... 131
  LBA28 .................................................. 527
identity mapping ........................................ 97, 104
idle process ............................................ 282
idle system call ........................................ 282
IDT (interrupt descriptor table) ..................... 136
  entering interrupt handlers ...................... 148
  entering interrupt handlers ...................... 139
  lidt ..................................................... 136, 146
  structure declaration .............................. 137
IDTR register .......................................... 146
  implementation (C function) ....................... 28, 43
  in ....................................................... 132
indirection ............................................. 400, 472, 476, 481
inherent concurrency ................................... 247
init process ............................................ 189
  becomes the idle process .......................... 282
inline assembler ........................................ 654
inode ...................................................... 400
  internal .............................................. 459
  remove if link count is 0 ......................... 480
  synchronize internal and on-disk ............... 468
inode bitmap ........................................... 442, 444
inode table ............................................. 442
instantaneousness (difficult word) .................. 178
instruction pointer (EIP) ............................ see EIP register
int 0x88 ............................................... 173, 203
Intel 8259 PIC ........................................... 130, 387
Intel syntax (Assembler) ............................. 31, 47
Intel x86 architecture ................................ 30
  accessing ports ..................................... 133
  CR3 register ......................................... 97
  enable paging, CR0 register .................... 109
  global descriptor table ......................... 90
  interrupt .......................................... 130
  interrupt descriptor table ...................... 136
  page descriptor .................................... 99
  page directory entry ............................. 101
  protected mode .................................... 59
  real mode .......................................... 59
  segment descriptor ................................ 90
  TSS (task state segment) ......................... 194
inter-process communication (IPC) .................... 556
interaction patterns ................................... 347
interaction time ........................................ 267
internal fragmentation ............................... 60
internal inode .......................................... 459
  synchronize with on-disk version ............... 468
interrupt ................................................. 42, 129
  disabling for synchronization .................. 352
  enable a specific interrupt ..................... 140
  lost wakeup ........................................ 352, 387
  spurious .......................................... 387
interrupt descriptor table ............................ see IDT
interrupt descriptor table register ............... 146
interrupt flag ......................................... 47, 129
nice values ........................................ 270
POSIX mutexes .................................. 368
readelf package ................................ 227
setpriority function .............................. 286
setreuid man page ................................ 573
signal numbers .................................. 559
source code ...................................... 633
stat structure .................................... 489
swap file .......................................... 288
system call numbers .............................. 204
tasklet ............................................ 386
Ulix development ................................ 615
Lions, John ........................................ 657
list directory entries .............................. 422, 490
literate programming.............................. 19
example .......................................... 19
prototype vs. implementation ................... 28
local file descriptor .............................. 410, 459
locality principle ................................ 80
lock (hardware instruction, for synchronization) 354
lock (kernel lock) ................................. see kernel lock
logical address .................................. 89
logical block addressing (LBA) .................. 527
logical filesystem ................................. 403
login .............................................. 580
login system call .................................. 582
lost wakeup ....................................... 352, 387
lseek system call ................................... 426
ltr .................................................. 194

M

macro (Assembler) ................................. 653
macro (C) .......................................... 644
magnetic tape drive ............................... 394
major device number .............................. 504
margin note ........................................ 26
master file table (MFT) ........................... 399
master PIC ......................................... 131
maximum number of address spaces ............. 158
maximum number of kernel locks ............... 365
maximum number of semaphores ................ 363
maximum number of system calls ................ 200
maximum number of threads ..................... 176
maximum number of virtual terminals .......... 325
maximum physical address ....................... 111
MByte .............................................. 25
mebibyte .......................................... 25
megabyte .......................................... 25
memory functions (standard library) ............. 596
memory management .............................. 49
access control ..................................... 50, 67
address relocation ............................... 62
address space ..................................... 63
address translation ............................... 63
best-fit allocation ................................ 55
bitmap ............................................. 54
Buddy system ..................................... 55
clear memory for newly loaded program ....... 232
contiguous allocation ............................. 51
copying physical memory ....................... 209
dirty page ......................................... 305
dynamic partitioning .............................. 53
first-fit allocation ................................ 54
fixed equal size partitioning ..................... 51
fixed variable size partitioning .................. 53
fragmentation ..................................... 60
frame allocation .................................. 118
free space management .......................... 50
implementation in Ulix ........................... 83
memory-mapped I/O ............................... 64
next-fit allocation ................................ 55
non-contiguous allocation ....................... 61
page allocation ................................... 119
page-based virtual memory ..................... 69
protection ......................................... 67
quick-fit allocation .............................. 55
segmentation ...................................... 58
swapping ......................................... 62
Ulix memory layout ............................... 87
virtual memory .................................... 63
worst-fit allocation .............................. 55
memory management unit ....................... 63, 70, 160, 287
emulation in Ulix ................................. 439
operating system ................................. 18, 38
minor device number ............................. 504
mkdir system call .................................. 426
mkfs.minix ........................................ 435, 437, 439
MMU .............................................. see memory management unit
modifier key ....................................... 313
mount .............................................. 401
mount option ....................................... 401
mount point ................................------- 401
mount table ........................................ 405
MPStuBS operating system .................. 658
MS-DOS operating system .................. 64, 190, 399, 400, 406, 658
FAT filesystem .................................. 398
filename convention .......................... 398
ntools software ................................ 616
multi-level address translation ............ 77
multi-level scheduler .......................... 270
multiboot header ............................... 86
multiprocessor scheduler .................... 271
multiprocessor synchronization ............ 355
mutex ............................................. 359, see kernel lock
mutual exclusion ................................ 350, 359, 377
k-mutual ......................................... 359

N
Nachos operating system ..................... 658
named semaphore ............................. 377
nasm assembler ................................ 86, 616
nested critical sections ..................... 356
new technology filesystem .................. 399
next-fit allocation ............................. 55
non-contiguous allocation ................... 61
non-contiguous filesystem ................... 396
non-interruptible kernel ..................... 379
non-maskable interrupt fault ................ 150
non-preemptive scheduling .................. 264
notangle .......................................... 23, 27
noweave .......................................... 23
noweb ............................................. 23, 618
NTFS filesystem ................................ 399
null page descriptor .......................... 73, 79
numbers .......................................... 70
binary ........................................... 25
hexadecimal .................................... 24
octal ............................................. 24

O
octal numbers .................................. 24
offset ............................................. 70
one-sided synchronization .................. 382
OOSTuBS operating system ................. 658
open system call .............................. 426
opening a file ................................ 411, 464, 495
OS/161 operating system .................. 658
OS/2 operating system ..................... 399
out ................................................ 132
out of bounds fault ........................... 150

P
page ............................................... 69
allocation ........................................ 119
offset ............................................ 70
page number .................................... 70
page descriptor ................................ 71, 76
32-bit integer interpretation .............. 99
sticky bit ........................................ 307
structure declaration ........................ 100
page descriptor trees ....................... 75
page directory ................................ 75
address space ................................ 160
page directory entry ......................... 98, 101
page fault ....................................... 150, 287
page fault handler ............................ 287
enlarge user mode stack ..................... 291
implementation in Ulix ..................... 288
paging in a paged-out page .................. 298
page frame ..................................... see frame
page locking .................................... 301, 307
page number ................................... 70
page replacement strategy .................. 300
clock ............................................. 303
dirty page ....................................... 305
FIFO ............................................ 302
freeing a frame ................................ 308
implementation in Ulix ..................... 304
least-frequently used ....................... 303
page locking ................................... 301
second chance ................................ 302
page table ...................................... 71
creation ......................................... 122
page table descriptor ....................... 76
structure declaration ........................ 102
page table entry .............................. 98
page table register ........................... 71
page-based virtual memory ................. 69
page_out system call ........................ 298
paging ............................................ 69
dirty page ....................................... 305
enable via CR0 register ..................... 109
multi-level address translation ............ 77
page replacement strategy ................. see page replacement strategy
paging out and in (swap file) ............ 295
swap file (paging file) ....................... see swap file
paging file ..................................... see swap file
parallel programming ...................... 247
parent process ................................ 208
password file ........................................ 581
PCB (process control block) ....................... 174
PD (page descriptor) ................................ 76
PDBR register ....................................... 99, 109
pending signal ...................................... 561
physical memory ................................... 63
PID (process identifier) ............................ 175
Pintos operating system ............................. 658
PIO (programmed input/output), cf. DMA ...... 525
pipe .................................................. 415
pointer arithmetic .................................. 642
polling .............................................. 153
port ................................................... 132
  Assembler language ............................... 133, 528
  CMOS chip ....................................... 339, 552
  DMA controller .................................. 542
  floppy controller ................................ 535
  IDE (hard disk) controller ..................... 525
  keyboard controller ............................. 153, 319
  programmable interrupt controller (PIC) .. 134
  serial port ..................................... 344
  timer chip ...................................... 338
  VGA graphics adapter ........................... 327
POSIX pthread mutex ................................ 368
POSIX thread ...................................... 254
  synchronization ................................ 368
PPID (parent process ID) ........................... 208
preemptive scheduling ............................. 264
presence bit (P bit), page descriptor .......... 72, 73, 77
printf (standard library) ......................... 597
priority-based scheduler ........................ 269, 286
privilege level ................................... 89
  process .......................................... 245
  aborting ........................................ 556
  child process .................................. 208
  clone function (Linux) .......................... 254
  ELF loader ...................................... 225
  exiting .......................................... 215
  file descriptors ................................ 424
  fork operation .................................. 207
  implementation ................................ 157
  loading a program file .......................... 228
  memory layout .................................. 159
  parent process ................................ 208
  process vs. thread .............................. 245
  termination by fault handler ................... 152
  wait for child .................................. 218
  wakes on key-press event ....................... 321
  working directory ............................... 432
zombie ............................................ 217
  removal in the scheduler ....................... 281
process control block ............................. 174
process descriptor base register (PDBR) ...... 99, 109
process file descriptor ........................... 410, 424
process identifier ................................ 175
process table ..................................... 176
processor hierarchy ............................... 252
program .......................................... 246
program code ..................................... 65
programmable interrupt controller ............ 130
  initialize ....................................... 134
protected mode ................................... 59, 88
  privilege level ................................ 89
  protection bits ................................ 72
  protection of code, data and stack .......... 67
  prototype (C function) ......................... 28, 43
PTD (page table descriptor) ...................... 76
pthread_create system call ....................... 258
pthread_exit system call ........................ 259
pthread_mutex_destroy system call ............. 372
pthread_mutex_init system call ................. 370
pthread_mutex_lock system call ................. 371
pthread_mutex_trylock system call .............. 371
pthread_mutex_unlock system call ............... 371
PTR ................................................ 71
Q
qemu PC emulator .................................. 616
query_ids system call .............................. 587
quick-fit allocation ................................ 55
quota system ..................................... 50
R
RAM disk ......................................... 407
Ramsey, Norman .................................. 23
random access (hard disk) ....................... 395
read system call .................................. 426
readchar system call ................................ 416
reading from a file ................................ 414, 470, 496
readlink system call ................................ 426
ready (thread state) .............................. 176
ready queue ....................................... 177, 179, 181, 182, 348, 360
  implementation ................................ 184
real group ID ..................................... 573
real mode ......................................... 59, 88
real time ......................................... 264
real user ID ....................................... 573
<table>
<thead>
<tr>
<th>Term</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>install signal handler (signal() function)</td>
<td>566</td>
</tr>
<tr>
<td>kill() function</td>
<td>562</td>
</tr>
<tr>
<td>pending</td>
<td>561</td>
</tr>
<tr>
<td>sent by the fault handler</td>
<td>556</td>
</tr>
<tr>
<td>SIG (forbidden address)</td>
<td>558</td>
</tr>
<tr>
<td>SIGABRT (abort)</td>
<td>557</td>
</tr>
<tr>
<td>SIGALRM (alarm clock)</td>
<td>557</td>
</tr>
<tr>
<td>SIGBUS (invalid address)</td>
<td>557</td>
</tr>
<tr>
<td>SIGCHLD (child terminates, stops, continues)</td>
<td>557</td>
</tr>
<tr>
<td>SIGCONT (continue)</td>
<td>557</td>
</tr>
<tr>
<td>SIGFPE (floating point exception)</td>
<td>557</td>
</tr>
<tr>
<td>SIGHUP (hang-up)</td>
<td>557</td>
</tr>
<tr>
<td>SIGILL (illegal instruction)</td>
<td>557</td>
</tr>
<tr>
<td>SIGINT (Ctrl-C pressed)</td>
<td>557</td>
</tr>
<tr>
<td>SIGKILL (kill at once)</td>
<td>558</td>
</tr>
<tr>
<td>signal handler</td>
<td>558</td>
</tr>
<tr>
<td>signal mask</td>
<td>559</td>
</tr>
<tr>
<td>SIGPIPEP (writing to pipe with no reader)</td>
<td>558</td>
</tr>
<tr>
<td>SIGPOLL (new data)</td>
<td>558</td>
</tr>
<tr>
<td>SIGPROF (alarm)</td>
<td>558</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>558</td>
</tr>
<tr>
<td>SIGSTOP (stop)</td>
<td>558</td>
</tr>
<tr>
<td>SIGSYS (unknown system call)</td>
<td>558</td>
</tr>
<tr>
<td>SIGTERM (terminate)</td>
<td>558</td>
</tr>
<tr>
<td>SIGTRAP (debugger breakpoint)</td>
<td>558</td>
</tr>
<tr>
<td>SIGTSTP (stop)</td>
<td>558</td>
</tr>
<tr>
<td>SIGTSTTIN (no terminal)</td>
<td>558</td>
</tr>
<tr>
<td>SIGTSTTOU (no terminal)</td>
<td>558</td>
</tr>
<tr>
<td>SIGURG (new data)</td>
<td>558</td>
</tr>
<tr>
<td>SIGUSR1 (user-defined)</td>
<td>558</td>
</tr>
<tr>
<td>SIGUSR2 (user-defined)</td>
<td>558</td>
</tr>
<tr>
<td>SIGVTALRM (alarm)</td>
<td>559</td>
</tr>
<tr>
<td>SIGXCPU (CPU usage exceeded)</td>
<td>559</td>
</tr>
<tr>
<td>SIGXFSZ (disk usage exceeded)</td>
<td>559</td>
</tr>
<tr>
<td>special treatment for KILL, STOP, CONT</td>
<td>563</td>
</tr>
<tr>
<td>signal (semaphore)</td>
<td>391</td>
</tr>
<tr>
<td>signal masking</td>
<td>376</td>
</tr>
<tr>
<td>signal system call</td>
<td>566</td>
</tr>
<tr>
<td>single indirection</td>
<td>400, 473, 477, 482</td>
</tr>
<tr>
<td>single mutual exclusion</td>
<td>131</td>
</tr>
<tr>
<td>slave PIC</td>
<td>131</td>
</tr>
<tr>
<td>soft link</td>
<td>419, 483</td>
</tr>
<tr>
<td>sparse file</td>
<td>501</td>
</tr>
<tr>
<td>spin lock</td>
<td>354, 355, 376</td>
</tr>
<tr>
<td>spurious interrupt</td>
<td>387</td>
</tr>
<tr>
<td>stack address space</td>
<td>165</td>
</tr>
<tr>
<td>Assembler language</td>
<td>652</td>
</tr>
<tr>
<td>enlarged by page fault handler</td>
<td>291</td>
</tr>
<tr>
<td>kernel mode</td>
<td>159</td>
</tr>
<tr>
<td>kernel stack delete list</td>
<td>168</td>
</tr>
<tr>
<td>layout when entering interrupt handler</td>
<td>141</td>
</tr>
<tr>
<td>required layout for starting the first process</td>
<td>193</td>
</tr>
<tr>
<td>user mode</td>
<td>159</td>
</tr>
<tr>
<td>stack fault</td>
<td>150</td>
</tr>
<tr>
<td>stack region</td>
<td>65</td>
</tr>
<tr>
<td>standard error</td>
<td>415</td>
</tr>
<tr>
<td>standard input</td>
<td>415, 430</td>
</tr>
<tr>
<td>standard output</td>
<td>415, 598</td>
</tr>
<tr>
<td>stat system call</td>
<td>426</td>
</tr>
<tr>
<td>state</td>
<td>176</td>
</tr>
<tr>
<td>static data</td>
<td>65</td>
</tr>
<tr>
<td>sti</td>
<td>47, 129</td>
</tr>
<tr>
<td>sticky bit (file)</td>
<td>572</td>
</tr>
<tr>
<td>sticky bit (page descriptor)</td>
<td>307</td>
</tr>
<tr>
<td>stream (NTFS)</td>
<td>400</td>
</tr>
<tr>
<td>string functions (standard library)</td>
<td>593</td>
</tr>
<tr>
<td>su program</td>
<td>586</td>
</tr>
<tr>
<td>SUID</td>
<td>458, 572</td>
</tr>
<tr>
<td>superblock</td>
<td>440</td>
</tr>
<tr>
<td>swap file</td>
<td>292</td>
</tr>
<tr>
<td>implementation in Ulix</td>
<td>292</td>
</tr>
<tr>
<td>paging out and in</td>
<td>295</td>
</tr>
<tr>
<td>writing and reading pages to/from disk</td>
<td>293</td>
</tr>
<tr>
<td>swapped out (thread state)</td>
<td>178, 180</td>
</tr>
<tr>
<td>swapped out queue</td>
<td>179</td>
</tr>
<tr>
<td>swapping</td>
<td>see also paging, 62, 292</td>
</tr>
<tr>
<td>switch off interrupts</td>
<td>352</td>
</tr>
<tr>
<td>symbolic link</td>
<td>see soft link</td>
</tr>
<tr>
<td>symlink system call</td>
<td>426</td>
</tr>
<tr>
<td>sync system call</td>
<td>512</td>
</tr>
<tr>
<td>synchronization</td>
<td>347</td>
</tr>
<tr>
<td>critical section</td>
<td>183, 348, 349</td>
</tr>
<tr>
<td>disabling interrupts</td>
<td>352, 374</td>
</tr>
<tr>
<td>k-mutual exclusion</td>
<td>359</td>
</tr>
<tr>
<td>lock (hardware instruction)</td>
<td>354</td>
</tr>
<tr>
<td>mutual exclusion</td>
<td>350</td>
</tr>
<tr>
<td>one-sided</td>
<td>382</td>
</tr>
<tr>
<td>semaphore</td>
<td>358</td>
</tr>
<tr>
<td>spin lock</td>
<td>354</td>
</tr>
<tr>
<td>test and set</td>
<td>353</td>
</tr>
<tr>
<td>threads</td>
<td>368</td>
</tr>
<tr>
<td>two-sided</td>
<td>382</td>
</tr>
<tr>
<td>user-level thread</td>
<td>375</td>
</tr>
<tr>
<td>via CPU instructions</td>
<td>353</td>
</tr>
<tr>
<td>system administrator</td>
<td>573</td>
</tr>
<tr>
<td>system call</td>
<td>42, 199</td>
</tr>
<tr>
<td>brk</td>
<td>172</td>
</tr>
<tr>
<td>Function</td>
<td>Page</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>chdir</td>
<td>433</td>
</tr>
<tr>
<td>chmod</td>
<td>590</td>
</tr>
<tr>
<td>chown</td>
<td>590</td>
</tr>
<tr>
<td>close</td>
<td>426</td>
</tr>
<tr>
<td>diskfree</td>
<td>493</td>
</tr>
<tr>
<td>error handling</td>
<td>205</td>
</tr>
<tr>
<td>example on Linux</td>
<td>199</td>
</tr>
<tr>
<td>execv</td>
<td>234</td>
</tr>
<tr>
<td>exit</td>
<td>216</td>
</tr>
<tr>
<td>filesystem access</td>
<td>423</td>
</tr>
<tr>
<td>fork</td>
<td>213</td>
</tr>
<tr>
<td>free_a_frame</td>
<td>309</td>
</tr>
<tr>
<td>ftruncate</td>
<td>426</td>
</tr>
<tr>
<td>get_errno</td>
<td>206</td>
</tr>
<tr>
<td>get_free_frames</td>
<td>309</td>
</tr>
<tr>
<td>getcwd</td>
<td>433</td>
</tr>
<tr>
<td>getdent</td>
<td>426</td>
</tr>
<tr>
<td>getpid</td>
<td>222</td>
</tr>
<tr>
<td>getppid</td>
<td>222</td>
</tr>
<tr>
<td>getpsinfo</td>
<td>223</td>
</tr>
<tr>
<td>gettid</td>
<td>222</td>
</tr>
<tr>
<td>idle</td>
<td>282</td>
</tr>
<tr>
<td>implementation in Ulix</td>
<td>200</td>
</tr>
<tr>
<td>installing a new handler</td>
<td>201</td>
</tr>
<tr>
<td>int 0x80</td>
<td>203</td>
</tr>
<tr>
<td>isatty</td>
<td>426</td>
</tr>
<tr>
<td>kill</td>
<td>565</td>
</tr>
<tr>
<td>link</td>
<td>426</td>
</tr>
<tr>
<td>login</td>
<td>582</td>
</tr>
<tr>
<td>lseek</td>
<td>426</td>
</tr>
<tr>
<td>mkdir</td>
<td>426</td>
</tr>
<tr>
<td>numbers taken from Linux</td>
<td>204</td>
</tr>
<tr>
<td>open</td>
<td>426</td>
</tr>
<tr>
<td>page_out</td>
<td>298</td>
</tr>
<tr>
<td>passing the return value via EAX</td>
<td>174</td>
</tr>
<tr>
<td>pthread_create</td>
<td>258</td>
</tr>
<tr>
<td>pthread_exit</td>
<td>259</td>
</tr>
<tr>
<td>pthread_mutex_destroy</td>
<td>372</td>
</tr>
<tr>
<td>pthread_mutex_init</td>
<td>370</td>
</tr>
<tr>
<td>pthread_mutex_lock</td>
<td>371</td>
</tr>
<tr>
<td>pthread_mutex_trylock</td>
<td>371</td>
</tr>
<tr>
<td>pthread_mutex_unlock</td>
<td>371</td>
</tr>
<tr>
<td>query_ids</td>
<td>587</td>
</tr>
<tr>
<td>read</td>
<td>426</td>
</tr>
<tr>
<td>readchare</td>
<td>416</td>
</tr>
<tr>
<td>readlink</td>
<td>426</td>
</tr>
<tr>
<td>resign</td>
<td>220</td>
</tr>
<tr>
<td>rmdir</td>
<td>426</td>
</tr>
<tr>
<td>set_errno</td>
<td>206</td>
</tr>
<tr>
<td>seteuid</td>
<td>582</td>
</tr>
<tr>
<td>setgid</td>
<td>582</td>
</tr>
<tr>
<td>setpsname</td>
<td>224</td>
</tr>
<tr>
<td>setterm</td>
<td>328</td>
</tr>
<tr>
<td>setuid</td>
<td>582</td>
</tr>
<tr>
<td>signal</td>
<td>566</td>
</tr>
<tr>
<td>SIGSYS signal</td>
<td>558</td>
</tr>
<tr>
<td>stat</td>
<td>426</td>
</tr>
<tr>
<td>symlink</td>
<td>426</td>
</tr>
<tr>
<td>sync</td>
<td>512</td>
</tr>
<tr>
<td>truncate</td>
<td>426</td>
</tr>
<tr>
<td>unlink</td>
<td>426</td>
</tr>
<tr>
<td>waitpid</td>
<td>219</td>
</tr>
<tr>
<td>write</td>
<td>426</td>
</tr>
</tbody>
</table>

T

tape drive .......................................................... 394

task ............................................................. see thread, process

task for the timer ............................................... 342

task state segment ........................................... 194

ltr ................................................................. 194

TSS descriptor .................................................. 196

update during context switch .................................. 280

task switch ........................................................ see context switch
tasklet ........................................................... 386

TCB ................................................................. see thread control block
team of threads .................................................. 246

terminal ............................................................ 325, 420
test and set (synchronization) ................................ 353

TEX ................................................................. 19, 181
text ............................................................... 65

THE (operating system) ............................................. 358

Thix operating system .......................................... 553, 657

thread .............................................................. 245

clone function (Linux) ............................................ 254

creation ........................................................... 254

exiting ............................................................. 259

fork operation (process) ....................................... 207

implementation in Ulix .......................................... 254

inherent concurrency ............................................ 247

POSIX ............................................................... 254

process vs. thread .............................................. 245

synchronization .................................................. see synchronization

system call for thread creation ................................ 258

team ................................................................. 246

thread control block .............................................. 174, 181, 182

creation ........................................................... 187

exit code .......................................................... 219
Index

<table>
<thead>
<tr>
<th>File descriptors</th>
<th>424</th>
</tr>
</thead>
<tbody>
<tr>
<td>for forked process</td>
<td>210</td>
</tr>
<tr>
<td>for new thread (in the same process)</td>
<td>255</td>
</tr>
<tr>
<td>for the first process</td>
<td>189</td>
</tr>
<tr>
<td>working directory</td>
<td>432</td>
</tr>
<tr>
<td>thread identifier</td>
<td>175, 182</td>
</tr>
<tr>
<td>thread state</td>
<td>176</td>
</tr>
<tr>
<td>thread table</td>
<td>176, 182</td>
</tr>
<tr>
<td>keeping zombie processes</td>
<td>217</td>
</tr>
<tr>
<td>throughput</td>
<td>264</td>
</tr>
<tr>
<td>TID (thread identifier)</td>
<td>175, 182</td>
</tr>
<tr>
<td>time (CMOS chip)</td>
<td>339</td>
</tr>
<tr>
<td>time division multiplex</td>
<td>249</td>
</tr>
<tr>
<td>timer</td>
<td>338</td>
</tr>
<tr>
<td>timer interrupt handler</td>
<td>130, 342</td>
</tr>
<tr>
<td>timer tasks</td>
<td>342</td>
</tr>
<tr>
<td>floppy driver</td>
<td>547</td>
</tr>
<tr>
<td>TLB</td>
<td>80, 253</td>
</tr>
<tr>
<td>invalidation</td>
<td>81</td>
</tr>
<tr>
<td>top half</td>
<td>386</td>
</tr>
<tr>
<td>Topsy (Teachable operating system)</td>
<td>657</td>
</tr>
<tr>
<td>TR register</td>
<td>194, 197</td>
</tr>
<tr>
<td>translation look-aside buffer</td>
<td>80</td>
</tr>
<tr>
<td>trap gate</td>
<td>137, 203</td>
</tr>
<tr>
<td>triple indirection</td>
<td>400</td>
</tr>
<tr>
<td>truncate system call</td>
<td>426</td>
</tr>
<tr>
<td>truncating a file</td>
<td>420, 484</td>
</tr>
<tr>
<td>TSS (task state segment)</td>
<td>194</td>
</tr>
<tr>
<td>ltr</td>
<td>194</td>
</tr>
<tr>
<td>TSS descriptor</td>
<td>196</td>
</tr>
<tr>
<td>update during context switch</td>
<td>280</td>
</tr>
<tr>
<td>turnaround time</td>
<td>264</td>
</tr>
<tr>
<td>two-sided synchronization</td>
<td>382</td>
</tr>
<tr>
<td>U</td>
<td></td>
</tr>
<tr>
<td>Ulix</td>
<td>1, 21</td>
</tr>
<tr>
<td>bootstrapping</td>
<td>618</td>
</tr>
<tr>
<td>build process</td>
<td>615</td>
</tr>
<tr>
<td>contents of the root disk</td>
<td>500</td>
</tr>
<tr>
<td>copyright</td>
<td>24</td>
</tr>
<tr>
<td>debugging help</td>
<td>603</td>
</tr>
<tr>
<td>design principles</td>
<td>21</td>
</tr>
<tr>
<td>development system</td>
<td>37</td>
</tr>
<tr>
<td>disk images</td>
<td>629</td>
</tr>
<tr>
<td>download the development environment</td>
<td>618</td>
</tr>
<tr>
<td>features</td>
<td>21</td>
</tr>
<tr>
<td>implementation of the Minix filesystem</td>
<td>439</td>
</tr>
<tr>
<td>implementation of the virtual filesystem</td>
<td>403</td>
</tr>
<tr>
<td>kernel mode shell</td>
<td>603</td>
</tr>
<tr>
<td>kernel semaphores</td>
<td>360</td>
</tr>
<tr>
<td>layout of the kernel code</td>
<td>41</td>
</tr>
<tr>
<td>memory layout</td>
<td>87</td>
</tr>
<tr>
<td>module</td>
<td>626</td>
</tr>
<tr>
<td>process implementation</td>
<td>157</td>
</tr>
<tr>
<td>signal implementation</td>
<td>559</td>
</tr>
<tr>
<td>system initialization</td>
<td>45</td>
</tr>
<tr>
<td>user mode library</td>
<td>47</td>
</tr>
<tr>
<td>users and groups</td>
<td>573</td>
</tr>
<tr>
<td>website</td>
<td>36</td>
</tr>
<tr>
<td>ulix.c</td>
<td>44</td>
</tr>
<tr>
<td>ulixlib.c</td>
<td>47</td>
</tr>
<tr>
<td>ulixlib.h</td>
<td>47</td>
</tr>
<tr>
<td>Unix</td>
<td>21, 659</td>
</tr>
<tr>
<td>porting to Ulix</td>
<td>31</td>
</tr>
<tr>
<td>signals</td>
<td>557</td>
</tr>
<tr>
<td>Unix process</td>
<td>245</td>
</tr>
<tr>
<td>Unix time (epoch)</td>
<td>340</td>
</tr>
<tr>
<td>unlink</td>
<td>480</td>
</tr>
<tr>
<td>unlink system call</td>
<td>426</td>
</tr>
<tr>
<td>unmount</td>
<td>401</td>
</tr>
<tr>
<td>user</td>
<td>571</td>
</tr>
<tr>
<td>effective user ID</td>
<td>573</td>
</tr>
<tr>
<td>implementation in Ulix</td>
<td>573</td>
</tr>
<tr>
<td>password file</td>
<td>581</td>
</tr>
<tr>
<td>real user ID</td>
<td>573</td>
</tr>
<tr>
<td>root user (UID 0)</td>
<td>573</td>
</tr>
<tr>
<td>user ID</td>
<td>573</td>
</tr>
<tr>
<td>user group</td>
<td>see group</td>
</tr>
<tr>
<td>user level semaphores</td>
<td>360</td>
</tr>
<tr>
<td>user level thread</td>
<td>252</td>
</tr>
<tr>
<td>user mode</td>
<td>161</td>
</tr>
<tr>
<td>GDT entries</td>
<td>194</td>
</tr>
<tr>
<td>switching to</td>
<td>193</td>
</tr>
<tr>
<td>user mode library</td>
<td>47</td>
</tr>
<tr>
<td>filesystem functions</td>
<td>429</td>
</tr>
<tr>
<td>user mode stack</td>
<td>159</td>
</tr>
<tr>
<td>enlarged by page fault handler</td>
<td>291</td>
</tr>
<tr>
<td>for a new thread</td>
<td>257</td>
</tr>
<tr>
<td>for forked process</td>
<td>211</td>
</tr>
<tr>
<td>prepare for loading a program</td>
<td>229</td>
</tr>
<tr>
<td>user-level interrupt</td>
<td>376</td>
</tr>
<tr>
<td>user-level semaphore</td>
<td>362, 374, 376</td>
</tr>
<tr>
<td>user-level thread</td>
<td>253, 362, 375</td>
</tr>
<tr>
<td>user-level thread library</td>
<td>362</td>
</tr>
<tr>
<td>user-level threads library</td>
<td>252</td>
</tr>
</tbody>
</table>
**Index**

**V**
- VFS (virtual filesystem) .................. 43, 403
- VGA graphics adapter ..................... 325
- video RAM .................................. 111, 327
- virtual address ............................. 64
  - structure .................................. 77
- virtual console .............................. 317
- virtual filesystem ........................... 43, 403
- virtual memory .............................. 41, 63, 65
  - accessing physical memory ................. 115
  - address space .............................. 63
  - address translation ......................... 63
  - clear memory for newly loaded program .... 232
  - copying physical memory ................... 209
  - CR3 register ............................... 97
  - identity mapping ......................... 97, 104
  - implementation in Ulix .................... 83
  - page allocation ............................ 119
  - page descriptor ............................ 71
  - page table ................................. 71
  - Ulix memory layout ....................... 87
- virtual monoprocessor ..................... 375, 376
- virtual multiprocessor .................... 249, 252, 362, 375, 376
- virtual processor .......................... 249, 252, 362
- virtual round robin scheduler .............. 268
- virtual terminal ............................ 325
- von Neumann, John ......................... 63
- von-Neumann architecture ................. 63

**W**
- wait (semaphore) .......................... 391
- waiting time ............................... 264
- waitpid system call ......................... 219
- Windows operating system .................. 30, 52, 57, 225, 399, 400, 406, 633
  - FAT filesystem ............................ 398
  - NTFS filesystem ........................... 399
  - Windows research kernel (WRK) ............ 633
- working directory .......................... 432
- worst-fit allocation ....................... 55
- write system call ........................... 426
- writing to a file ........................... 415, 474, 497
- written bit ................................. 72

**X**
- Xinu operating system ..................... 38
- xv6 operating system ...................... 344, 525, 657

**Z**
- Z80 processor .............................. 129
- Zilog Z80 processor ....................... 129
- zombie process ............................ 217
  - removal in the scheduler ................. 281
- zone ....................................... 441
- zone bitmap ............................... 442, 444
Bibliography


### Image Credits


Punch card image (Figure 12.1): Wikimedia Commons, http://commons.wikimedia.org/wiki/File:Blue-punch-card-front-horiz.png


Other hard disk icon (Figures 12.5, 12.6): Wikimedia Commons, http://commons.wikimedia.org/wiki/File:Harddisk.svg

The font used for Figure 10.1 and Figure 10.2 is “Aa Qwertz-Tasten” and was designed by Anke Arnold, http://www.anke-art.de/
GNU General Public License

Version 3, 29 June 2007


Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers’ and authors’ sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users’ freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

0. Definitions.

“This License” refers to version 3 of the GNU General Public License.

“Copyright” also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

“The Program” refers to any copyrightable work licensed under this License. Each licensee is addressed as “you”. “Licensees” and “recipients” may be individuals or organizations.

To “modify” a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work.
A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except that warranties might be provided to you by the licensee in violation of this License). If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.


The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.


You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is
released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to “keep intact all notices”.

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation’s users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the...
entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or
d) Limiting the use for publicity purposes of names of licensors or authors of the material; or
e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or
f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.


You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party’s predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor’s "contributor version".

A contributor’s “essential patent claims” are all patent claims
GNU General Public License

own  controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some
manner, permitted by this License, of making, using, or selling its
contributor version, but do not include claims that would be
infringed only as a consequence of further modification of the
contributor version. For purposes of this definition, "control"
includes the right to grant patent sublicenses in a manner
consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide,
royalty-free patent license under the contributor’s essential
patent claims, to make, use, sell, offer for sale, import and
otherwise run, modify and propagate the contents of its
ccontributor version.

In the following three paragraphs, a "patent license" is any
express agreement or commitment, however denominated, not to
enforce a patent (such as an express permission to practice a
patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an
agreement or commitment not to enforce a patent against the
party.

If you convey a covered work, knowingly relying on a patent
license, and the Corresponding Source of the work is not
available for anyone to copy, free of charge and under the terms
of this License, through a publicly available network server or
other readily accessible means, then you must either (1) cause the
Corresponding Source to be so available, or (2) arrange to deprive
yourself of the benefit of the patent license for this particular
work, or (3) arrange, in a manner consistent with the
requirements of this License, to extend the patent license to
downstream recipients. "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying
the covered work in a country, or your recipient’s use of the
covered work in a country, would infringe one or more
identifiable patents in that country that you have reason to
believe are valid.

If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance
of, a covered work, and grant a patent license to some of the
parties receiving the covered work authorizing them to use,
propagate, modify or convey a specific copy of the covered work,
then the patent license you grant is automatically extended to all
recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that
are specifically granted under this License. You may not convey a
covered work if you are a party to an arrangement with a third
party that is in the business of distributing software, under which
you make payment to the third party based on the extent of your
activity of conveying the work, and under which the third party
grants, to any of the parties who would receive the covered work
from you, a discriminatory patent license (a) in connection with
copies of the covered work conveyed by you (or copies made
from those copies), or (b) primarily for and in connection with
specific products or compilations that contain the covered work,
unless you entered into that arrangement, or that patent license
was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or
limiting any implied license or other defenses to infringement
that may otherwise be available to you under applicable patent
law.


If conditions are imposed on you (whether by court order,
agreement or otherwise) that contradict the conditions of this
License, they do not excuse you from the conditions of this
License. If you cannot convey a covered work so as to satisfy
simultaneously your obligations under this License and any other
pertinent obligations, then as a consequence you may not convey
it at all. For example, if you agree to terms that obligate you to
collect a royalty for further conveying from those to whom you
convey the Program, the only way you could satisfy both those
terms and this License would be to refrain entirely from
conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work
licensed under version 3 of the GNU Affero General Public
License into a single combined work, and to convey the resulting
work. The terms of this License will continue to apply to the part
which is the covered work, but the special requirements of the
GNU Affero General Public License, section 13, concerning
interaction through a network will apply to the combination as
such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new
versions of the GNU General Public License from time to time.
Such new versions will be similar in spirit to the present version,
but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the
Program specifies that a certain numbered version of the GNU
General Public License "or any later version" applies to it, you
have the option of following the terms and conditions either of
that numbered version or of any later version published by the
Free Software Foundation. If the Program does not specify a
version number of the GNU General Public License, you may
choose any version ever published by the Free Software
Foundation.

If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that
proxy’s public statement of acceptance of a version permanently
authorizes you to choose that version for the Program.

Later license versions may give you additional or different
permissions. However, no additional obligations are imposed on
any author or copyright holder as a result of your choosing to
follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE
EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN
OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE
PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF
MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND
PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD
THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST
OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.
IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the program's name and a brief idea of what it does.>

Copyright (C) <textyear> <name of author>

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

<program> Copyright (C) <year> <name of author>
This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
This is free software, and you are welcome to redistribute it under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see http://www.gnu.org/licenses/.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read http://www.gnu.org/philosophy/why-not-lgpl.html.