Implementierung eines ELF-Programm-Loaders für das Betriebssystem ULIX

(unter Verwendung von Literate Programming)

Bachelorarbeit

in der Fakultät Informatik

vorgelegt von:	Frank Kohlmann
Fakultät:	Informatik
Matrikelnummer:	2036325
Erstprüfer:	Prof. Dr. Ralf U. Kern
Zweitprüfer:	Dipl.-Math. Dipl.-Inform. Hans-Georg Eßer

1. März 2013, 07:29

Prüfungsrechtliche Erklärung

Ich, Frank Kohlmann, Matrikel-Nr. 2036325, versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

_____

Unterschrift

Zusammenfassung

In dieser Bachelorarbeit geht es um die Implementierung eines Programm-Loaders für das Betriebssystem Ulix-i386. Programm-Loader sind eine zentrale Komponente von Betriebssystemen, da erst durch sie Programme durch das Betriebssystem ausgeführt werden können. In dieser Arbeit wurde eine Implementierung umgesetzt, die das Executable and Linking Format verwendet. ELF ist das Standardformat für Programme auf Unix-Systemen. Für die Entwicklung des ELF-Programm-Loaders wurde die von Donald E. Knuth erfundene Programmiertechnik Literate Programming eingesetzt. Diese Programmiertechnik erlaub es gut dokumentierten Programmcode zu erstellen, da der Schwerpunkt der Entwicklungsarbeit in erster Linie auf der Dokumentation des Programmcodes liegt.

Inhaltsverzeichnis

Einleitung

In diesem Dokument wird ein Programm-Loader für das Lehrbetriebssystem Ulix-i386 implementiert und dokumentiert. Für die Implementierung musste zunächst eine intensive Einarbeitung in den bestehenden Source-Code von Ulix-i386 erfolgen. Bei der Umsetzung kommt eine spezielle Programmiertechnik namens *Literate Programming* zum Einsatz. Der Softwareentwicklungsprozess unter Verwendung dieser Programmiertechnik unterscheidet sich dabei in einigen Punkten von klassischen Herangehensweisen der Softwareentwicklung. Bei der klassischen Softwareentwicklung wird im allgemeinen Sourcecode produziert, der während bzw. nach der Entwicklung des Sourcecodes kommentiert und dokumentiert wird. *Literate Programming* vertauscht diese beiden Entwicklungsschritte. Bei der Verwendung von *Literate Programming* wird die Dokumentation des Sourcecodes erstellt, bevor der eigentliche Sourcecode geschrieben wurde. Der Fokus des Softwareentwicklungsprozesses wird damit stärker auf die Dokumentation der entwickelten Software gerichtet. Auf diese Weise wird dem Leser ermöglicht, die Gedankengänge zur Lösung eines Problems direkt nachzuvollziehen und besser zu verstehen. Die Dokumentation und der Sourcecode werden dabei in das selbe Dokument geschrieben. Aus dem so entstandenen Quelldokument können sowohl der Programm-Sourcecode als auch der Dokumentations-Sourcecode erzeugt werden. Um die Dokumentation in druckbarer zu Erzeugen kommt z.B. ein Textsatzsystem wie (La)TeX zum Einsatz. Es gibt mehrere *Literate Programming* Systeme wie z.B. WEB, NoWEB, CWEB, funnelweb, fweb oder LEO [Wik13c]. In dieser Ausarbeitung wird NoWEB in Kombination mit der Programmiersprache C und dem Textsatzsystem LaTeX verwendet. Die Funktionsweise von NoWEB wird in Kapitel 1.2 beschrieben.

1. Grundlagen

In diesem Kapitel werden grundlegende Informationen zu den Themen Ulix-i386, NoWEB und ELF vorgestellt. Dabei geht es hauptsächlich darum, die zum Verständnis der Implementierung nötigen Grundlagen zu schaffen.

1.1. Ulix-i386

Ulix-i386 ist ein Unix-ähnliches Betriebssystem, welches grundlegende Konzepte von Betriebssystemen im Rahmen von Lehrveranstaltungen demonstrieren soll. Entwickelt wird Ulix-i386 von Felix Freiling und Hans-Georg Eßer. Ulix-i386 befindet sich zum gegenwärtigen Zeitpunkt noch in der Entwicklungsphase. Die Implementierung von Ulix-i386 erfolgt ebenso wie diese Bachelorarbeit unter Verwendung von *Literate Programming*, wodurch gut strukturierter, dokumentierter und verständlicher Sourcecode eines Betriebssystems entsteht. Eines der erklärten Ziele von Ulix-i386 ist es, Konzepte von realen Betriebssystemen möglichst einfach umzusetzen und so einen „leichteren“ Zugang zu diesem Themenbereich zu ermöglichen. In den nachfolgenden Kapiteln werden grundlegende Konzepte von Ulix-i386, die mit dem Laden von ELF-Programmen in Zusammenhang stehen thematisiert. Dabei werden die angesprochenen Bereiche von Ulix-i386 nicht im Detail erklärt, oder gar auf Implementierungsdetails eingegangen. Es geht in den folgenden Kapiteln lediglich darum, einen Einblick in Ulix-i386-Bereiche zu gewähren, die in Zusammenhang mit dem Laden von Programmen stehen.

1.1.1. Speicherverwaltung

Ulix-i386 verwendet eine virtuelle Speicherverwaltung. Dabei werden virtuelle Adressbereiche auf den tatsächlich vorhandenen physikalischen Speicher abgebildet. Die Adressen der virtuellen Adressbereiche bilden dabei einen zusammenhängenden Bereich. Die auf dem System ausgeführten Programme arbeiten somit immer mit virtuellen Adressen. Jedem Programm steht zu diesem Zweck ein eigener *Adressraum* zur Verfügung [FE]. Der *Adressraum* eines Prozesses wird nachfolgend auch als *Prozessimage* bezeichent. Eine schmatische Darstellung eines solchen Prozessimages ist in Abbildung 1.1 dargestellt.

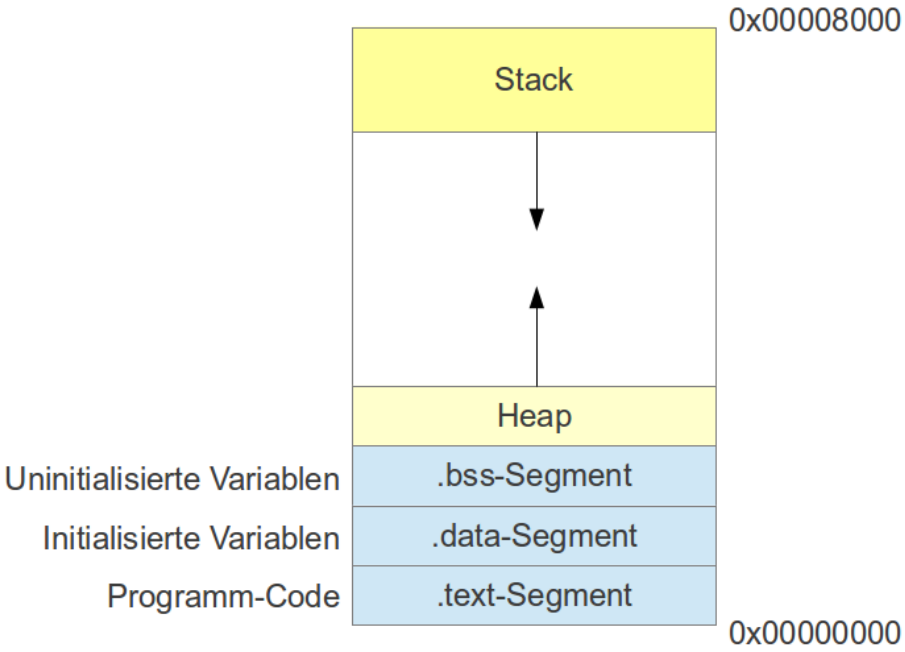

Abbildung 1.1.: Prozessimage eines Programms

Die Darstellung in Abbildung 1.1 zeigt, dass jedes *Prozessimage* an der virtuellen Adresse `0x00000000` beginnt. Dort befinden sich die jeweiligen *Segmente* eines Programms. Am oberen Ende des Prozessimages befindet sich der Stack eines Programms, der von oben nach unten wächst. Die Größe eines Prozessimages ist von ULIX-i386 mit 32×1024 Byte festgelegt und wird durch die symbolische Konstante `PROCESS_IMAGE_SIZE` definiert. Da es zum aktuellen Entwicklungsstand von ULIX-i386 noch keine Möglichkeit gibt um Arbeitsspeicher dynamisch zu allozieren, wird das gesamte Bereich des Prozessimages durch ULIX-i386 beim Erzeugen des Adressraums alloziert. Es kann deshalb auf jede virtuelle Speicheradresse im *Prozessimage* zugegriffen werden. Weiter sei noch darauf hingewiesen, dass sich im letzten GiByte – also der Adressbereich von `0xc0000000` bis `0xffffffff` (siehe [FE]) – des virtuellen Adressbereichs, der Programmcode des Kernels befindet. Prozesse dürfen somit ausschließlich virtuelle Adressen unterhalb der Adressgrenze `0xc00000000` verwenden.

1.1.2. Prozessverwaltung

Obwohl das Laden und Ausführen von Programmen in direktem Zusammenhang mit Prozessen steht – da für die Ausführung jedes Programms ein Prozess benötigt wird – gibt es im ELF-Programm-Loader nur wenige Berührungsstellen mit der *Prozessverwaltung* von ULIX-i386. Dennoch werden in diesem Kapitel einige wissenswerte Punkte zum Thema *Prozessverwaltung* in ULIX-i386 angesprochen.

Ein ULIX-i386-Prozess besteht im Wesentlichen aus einem *Prozessimage* und einer Prozessverwaltungsstruktur. Diese Verwaltungsstrukturen werden vom Kernel verwaltet. Außerdem verfügt ULIX-i386 über einen Prozess-Scheduler. Der Prozess-Scheduler hat die Möglichkeit einen laufenden Prozess zu unterbrechen, um einen anderen Prozess zur Ausführung zu bringen. Nach welcher Scheduling-Strategie der Prozess-Scheduler arbeitet ist für den Programm-Loader nicht von Bedeutung. Um überhaupt einen Prozess zu Erzeugen stellt die ULIX-i386-Systembibliothek *ulixlib* die Funktion `fork` zur Verfügung. Somit ist jeder Prozess in der Lage weitere Prozesse zu Erzeugen. Der Ausgangsprozess ist dabei der Prozess *init*, welcher die ULIX-i386-Shell startet. Für den ULIX-i386-Programm-Loader besteht keine Notwendigkeit neue Prozesse zu erzeugen, da er immer aus einem bereits existenten Prozess heraus aufgerufen wird.

1.1.3. Systemaufrufe

Systemaufrufe werden von Betriebssystemen eingesetzt, um Programmen, die im Usermode laufen die Möglichkeit zu geben Operationen auszuführen, die dem Betriebssystem vorbehalten sind. Dies ist z.B. beim Zugriff auf Hardwarekomponenten oder anderen Ressourcen der Fall. Der Aufruf eines Syscalls folgt dabei einem festen Schema. Das Usermode-Programm muss dazu die Nummer des jeweiligen Syscalls in das CPU-Register *eax* laden. Außerdem können dem *Systemaufruf* Argumente übergeben werden, indem sie in die Register *ebx*, *ecx*, etc. geladen werden. Danach löst das Programm einen *Interrupt* aus, der für den Aufruf von Syscalls zuständig ist. Bei ULIX-i386 ist dieser *Interrupt* `int 0x80`. Danach übernimmt das Betriebssystem die Kontrolle und führt den entsprechenden Syscall aus. Was nachfolgen im Betriebssystem passiert, wird in Kapitel 3.2.1 genauer erläutert. Damit ein Usermode-Programm ohne die Verwendung von Assembler-Code einen Syscall aufrufen kann stellt die Systembibliothek *ulixlib* entsprechende Funktionen für die Systemaufrufe zur Verfügung.

1.1.4. Dateisystem-Zugriff

ULIX-i386 stellt in der aktuellen Version ein einfaches Dateisystem mit dem Namen „simplefs“ zur Verfügung. Das simplefs-Dateisystem erlaubt es mittels Funktionen wie `simplefs_open`, `simplefs_read`, `simplefs_close`, etc. auf Dateien zuzugreifen. Die simplefs-Funktionen werden benötigt, um die gewünschte ELF-Datei zu lesen und auszuführen. Alle simplefs-Funktionen sind in der Systembibliothek *ulixlib* enthalten und können von Usermode-Programmen verwendet werden.

1.2. NoWEB: Überblick und Konzepte

NoWEB ist eine Neuentwicklung des von Donald E. Knuth entwickelten *Literate Programming* Systems WEB. Es wurde in den Jahren 1989 bis 1999 von Norman Ramsey entwickelt. NoWEB erfüllt im Prinzip zwei grundsätzliche Aufgabe von Literate Programming. Diese Aufgaben sind das Extrahieren von Programm-Sourcecode und das Extrahieren, von Dokumentations-Sourcecode. Zu diesem Zweck stellt NoWEB zwei Tools zur Verfügung. Die Tools sind `notangle` und `noweave`. `notangle` extrahiert den Programmcode aus der NoWEB Ursprungsdatei. `noweave` erzeugt die Dokumentation, welche den beschreibenden Text und den Sourcecode in druckbarer Form enthält. Abbildung 1.2 zeigt, wie die genannten Tools zusammenarbeiten.

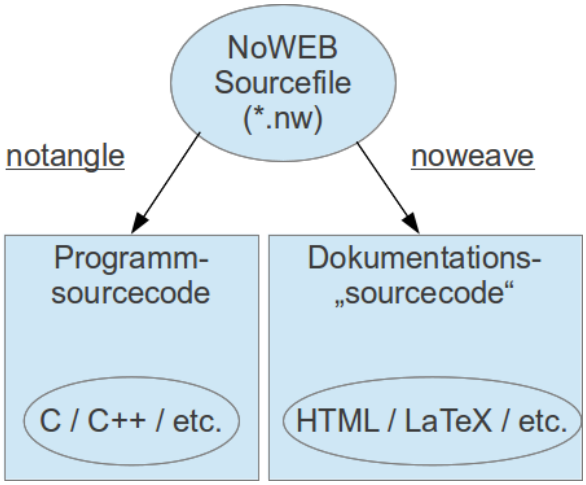

Abbildung 1.2.: NoWEB workflow

Der Vorteil von NoWEB gegenüber anderen *Literate Programming* Systemen ist, dass es bei der Wahl der Programmiersprachen und des Dokumentationssystems flexibel ist. NoWEB kann z.B. mit Programmiersprachen wie C, C++, PASCAL, etc. eingesetzt werden. Grundsätzlich ist allerdings der Einsatz jede Programmiersprache möglich. Auch bei die Wahl des Dokumentationssystems gibt es mehrere Möglichkeiten wie z.B. TeX, LaTeX oder HTML.

Der Programmcode und die Dokumentation werden bei NoWEB in die selbe Datei geschrieben. Diese Ursprungsdatei erhält für gewöhnlich die Endung `*.nw`. Die Ursprungsdatei dient dann den beiden Tools `noweave` und `notangle` als Eingabedatei. Das Tool `notangle` kann dabei durchaus mehrfach auf die selbe Ursprungsdatei angewendet werden. Der Grund dafür ist, dass `notangle` bei jedem Durchlauf eine Ausgabedatei erzeugt. Wenn also ein Softwareprojekt aus mehreren Sourcecode Dateien besteht ist es nötig, dass `notangle` mehrfach

ausgeführt wird. Welchen Sourcecode aus der Ursprungsdatei diese Ausgabedateien dann enthalten kann über die Angabe des *Code-Chunks* als Aufrufparameter gesteuert werden. Die folgende Abbildung für einen Aufruf von `notangle` sollte diesen Zusammenhang verdeutlichen.

```
$ notangle -L -Rmodule.c BeispielProjekt.nw > BeispielProjekt.c
```

Listings 1.1: Programmcode extrahieren

Mit dem Parameter „-L“ wird angegeben, dass die Ausgabedatei die Zeilennummern enthalten soll, aus welchen die Codedefinitionen in der Ursprungsdatei stammen. Dadurch hat der Compiler beim Übersetzen des C-Source-Code die Möglichkeit, bei Fehlern die Zeilennummern der NoWEB-Datei anzugeben. Der *Code-Chunk*-Name nach dem Parameter „-R“ gibt an, welcher *Code-Chunk* extrahiert werden soll. Der Befehl in Listing 1.1 erzeugt die Datei `BeispielProjekt.c`. In dieser Datei ist der Sourcecode enthalten, der in der Ursprungsdatei im *Code-Chunk* `module.c` definiert wurde. Die C-Sourcecode-Datei `BeispielProjekt.c` kann nachfolgend mit gewöhnlichen Compiler- und Linkeraufrufen in Maschinencode übersetzt werden. Ab diesem Zeitpunkt besteht praktisch kein Unterschied mehr zu einer „normalen“ C-Sourcecode-Datei. Um die Dokumentation des Programms zu erzeugen wird der Aufruf in Listing 1.2 verwendet.

```
$ noweave [options] BeispielProjekt.nw > BeispielProjekt.tex
```

Listings 1.2: Dokumentation extrahieren

Das `noweave` Kommando erzeugt in diesem Fall aus der Urspungsdatei `BeispielProjekt.nw` eine LaTeX-Datei mit dem Namen `BeispielProjekt.tex`. Diese kann dann mit den üblichen LaTeX-Werkzeugen weiterverarbeitet werden und ist somit die Grundlage für die Dokumentation des Programms.

1.3. ELF-Überblick

Die Abkürzung ELF steht für Executable and Linking Format. Dabei handelt es sich um ein binäres Dateiformat, welches zur Ablage von Computerprogrammen in einem definierten Format verwendet wird. Dateien im *Executable and Linking Format* können sowohl ausführbare Programme als auch Bibliotheken beinhalten. Es kommt heute in mehreren Unix-artigen Betriebssystemen wie z.B. BSD, Linux, Minix zum Einsatz. Grundsätzlich ist ELF (Betriebssystem-) plattformunabhängig und kann deshalb in verschiedenen Betriebssystemen eingesetzt

werden. Um ebenfalls eine gewisse (Hardware-) Architekturunabhängigkeit zu erreichen gibt es Erweiterungen, wie z.B. das *FatELF*, welches in der Lage ist, Maschinencode mehrerer Architekturen bereitzustellen.

ELF wurde ursprünglich von den UNIX System Laboratories (USL) als Teil des *Application Binary Interface* (ABI) entwickelt und veröffentlicht [Com95]. Seit 1993 ist ELF vom Tool Interface Standard Committee (TISC) als Standard für ausführbare Programme akzeptiert [Wik13a].

1.3.1. ELF-Aufbau

Die grundsätzliche Struktur einer Datei im ELF ist in Abbildung 1.3 dargestellt. Dabei ist zu beachten, dass das ELF zwei unterschiedliche Formatvarianten für eine ELF-Datei vorsieht. Die eingesetzte Formatvariante richtet sich dabei nach dem Verwendungszweck der erzeugten ELF-Datei. Nach der ELF Spezifikation können entweder ausführbare Programmdateien oder Bibliotheksdateien im ELF erzeugt werden.

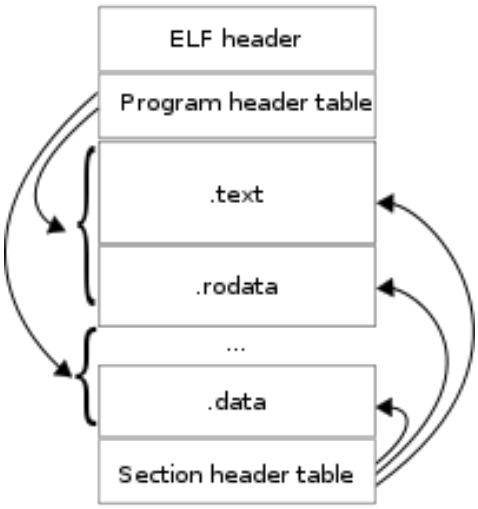

Abbildung 1.3.: Struktur von ELF-Dateien. Quelle: [Wik13b]

a) **Ausführungsvariante:** In der Ausführungsvariante ist der mittlere Teil der ELF-Datei, wie in Abbildung 1.3 dargestellt, in *Segmente* unterteilt. Dabei ist der *Programm-Header* zwingend erforderlich und der Sektions-Header optional. Der *Programm-Header* wird in dieser Variante verwendet, um die einzelnen *Segmente* zu adressieren. In den Segmenten

sind alle für die Ausführung des Programms benötigten Daten enthalten. Dies beinhaltet den eigentlichen Programmcode (*.text*-Segment), die vom Programm benötigten Daten (*.data*-Segment) sowie zusätzliche Informationen über die ELF-Datei.

b) **Linkervariante:** In der Linkervariante ist der mittlere Teil der ELF-Datei, wie in Abbildung 1.3 dargestellt, in *Sektionen* unterteilt. Dabei ist der Sektions-Header zwingend erforderlich und der *Programm-Header* optional. In dieser Variante werden der Sektions-Header verwendet, um die *Sektionen* der ELF-Datei zu adressieren. Die Linkervariante enthält zusätzliche Informationen, die es dem Betriebssystem ermöglichen eine dynamisch gelinkte ELF-Datei auszuführen und Programmteile dynamisch zur Ausführung zu bringen. Diese Variante des ELFs ist allerdings nicht Teil des hier implementierten Programm-Loaders.

Für Ulix-i386 wird vorerst nur die Ausführungsvariante relevant sein, da die „speziellen“ Ulix-i386-ELF-Dateien immer statisch gelinkt werden und deshalb zur Ausführungszeit keine dynamischen Bibliotheken nachgeladen werden müssen. Um die einzelnen Teile einer ELF-Datei anzusprechen, werden Strukturen verwendet. Diese Strukturen werden aus vier 32 Bit und einem 16 Bit breiten Datentypen gebildet. Die Strukturen des ELFs werden in Kapitel 1.3.2 behandelt.

1.3.2. Typdefinitionen für ELF-Dateien

Die Typdefinitionen für ELF wurden aus dem Dokument *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2* [Com95] übernommen.

Die fünf Datentypen `Elf32_Addr`, `Elf32_Half`, `Elf32_Off`, `Elf32_Sword` und `Elf32_Word` sind die grundlegenden Typen, aus denen alle weiteren Strukturen von ELF-Dateien aufgebaut werden. Es handeln sich dabei mit Ausnahme von `Elf32_Half` (16-Bit) um 32-Bit Datentypen.

15 ⟨*ELF-Typ-Definitionen* 15⟩≡ (31a) 16▷

```
typedef unsigned int        Elf32_Addr;     //  4 byte
typedef unsigned short int  Elf32_Half;     //  2 byte
typedef unsigned int        Elf32_Off;      //  4 byte
typedef signed int          Elf32_Sword;    //  4 byte
typedef unsigned int        Elf32_Word;     //  4 byte
```

Defines:
`Elf32_Addr`, used in chunks 16, 18, and 59a.
`Elf32_Half`, used in chunk 16.
`Elf32_Off`, used in chunks 16, 18, and 59a.
`Elf32_Sword`, never used.
`Elf32_Word`, used in chunks 16, 18, and 59a.
Uses `byte` 56b.

Für das Laden eines Programms sind vor allem die ELF-Header-Struktur und die ELF-Programm-Header-Struktur von Bedeutung.

Typdefinition des ELF-Header

16 ⟨*ELF-Typ-Definitionen* 15⟩+≡ (31a) ◁15 18▷

```
typedef struct {
  unsigned char   e_ident[EI_NIDENT]; // Magic Number: 0x7f 0x45(E) 0x4c(L) 0x46(F)
  Elf32_Half      e_type;             // ausführbare / relozierbare Datei
  Elf32_Half      e_machine;          // Hardware Architektur
  Elf32_Word      e_version;          // ELF Version
  Elf32_Addr      e_entry;            // Einsprungpunkt in den Programmcode
  Elf32_Off       e_phoff;            // Offset zur Programm-Header Tabelle
  Elf32_Off       e_shoff;            // Offset zur Sektions-Header Tabelle
  Elf32_Word      e_flags;            // Prozessorspezifische Flags
  Elf32_Half      e_ehsize;           // Groesse des ELF Headers
  Elf32_Half      e_phentsize;        // Groesse eines Programm-Header Eintrags
  Elf32_Half      e_phnum;            // Anzahl der Programm-Header
  Elf32_Half      e_shentsize;        // Groesse eines Sektions-Header Eintrags
  Elf32_Half      e_shnum;            // Anzahl der Sektions-Header
  Elf32_Half      e_shstrndx;         // Index des Sektions-Header,
} Elf32_Ehdr;                         // der StringTable
```

Defines:
`Elf32_Ehdr`, used in chunks 40a, 61, and 62.
Uses `Elf32_Addr` 15, `Elf32_Half` 15, `Elf32_Off` 15, and `Elf32_Word` 15.

Der oben definierte Typ `Elf32_Ehdr` ist die Header-Struktur einer ELF-Datei. Der Eintrag `e_entry` enthält dabei die Speicheradresse des Programms an der die Ausführung beim Programmstart beginnen soll. Diese Speicheradresse zeigt, wie in Kapitel 1.4 beschrieben immer auf die *main*-Funktion des Programms. Das Feld `e_phoff` enthält ausgehend vom Dateianfang den Offset zu der Speicheradresse, an welcher die *Programm-Header* stehen.

Das nachfolgend abgebildete Listing 1.3 wurde mit dem Unix-Tool `readelf` erzeugt. Das Programm `readelf` ist in der Lage ELF-Dateien zu analysieren und Informationen über die Datei auszugeben. In Listing 1.3 wurde `readelf` dazu verwendet, die ELF-Header-Informationen des Beispielprogramms anzuzeigen.

```
ELF-Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Klasse:                            ELF32
  Daten:                             2er-Komplement, Little-Endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI-Version:                       0
  Typ:                               EXEC (ausführbare Datei)
  Maschine:                          Intel 80386
  Version:                           0x1
  Einstiegspunktadresse:             0x8ef
  Beginn der Programm-Header:        52 (Bytes in Datei)
  Beginn der Sektions-header:        15476 (Bytes in Datei)
  Flags:                             0x0
  Größe dieses Headers:              52 (Byte)
  Größe der Programm-Header:         32 (Byte)
  Number of program headers:         5
  Größe der Sektions-Header:         40 (bytes)
  Anzahl der Sektions-Header:        20
  Sektions-Header Stringtabellen-Index: 17
```

Listings 1.3: Headerinformationen (elfbin)

In der mit „Magic:“ betitelten Zeile zeigt `readelf` den Inhalt des Feldes `e_ident` (siehe 1.3.2) an. Die ersten vier Werte des Feldes `e_ident` bilden zusammen die „Magic-Number“, durch die eine ELF-Dateie identifiziert werden kann. Im Einzelnen besteht die „Magic-Number“ aus dem Wert `0x7f` und den drei ASCII-Werten der Zeichen 'E', 'L' und 'F'. Der Wert `0x7f` ist dabei per Definition festgelegt und hat keine weitere Bedeutung. Die Werte an den Indexpositionen 4 bis 6 des Feldes `e_ident` bezeichnen die „Klasse“, „Byte-Reihenfolge“ und „Version“ der ELF-Datei. Anhand der Werte des `e_ident`-Feldes kann festgestellt werden, ob eine ELF-Datei vorliegt oder nicht. Der folgende *Code-Chunk* wird vom ELF-Programm-Loader verwendet, sicherzustellen, dass es sich um eine ELF-Datei handelt, die unter Ulix-i386 ausgeführt werden kann.

17 ⟨*ELF-Datei überprüfen* 17⟩≡ (35)

```
if ( ! (
          pElfHeader->e_ident[EI_MAG0]    == ELFMAG0      &&
          pElfHeader->e_ident[EI_MAG1]    == ELFMAG1      &&
          pElfHeader->e_ident[EI_MAG2]    == ELFMAG2      &&
          pElfHeader->e_ident[EI_MAG3]    == ELFMAG3      &&
          pElfHeader->e_ident[EI_CLASS]   == ELFCLASS32   &&
          pElfHeader->e_ident[EI_DATA]    == ELFDATA2LSB  &&
          pElfHeader->e_ident[EI_VERSION] == EV_CURRENT
       ) )
{
  // errno = ENOEXEC ;
  r->eax = -1;
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("Invalid file format.");
  #endif
  return;
}
```

Uses ELF_TEST_MSG 59b, ENOEXEC 59c, and TEST_ERROR_MSG 59b.

Der vorangehende Code prüft anhand der ersten vier Bytes, ob es sich um eine Datei im ELF handelt. Weiter wird mit `ELFCLASS32` getestet, ob die Objektdaten in der ELF-Datei 32-Bit-Speicheradressen verwenden. Das fünfte Byte (`EI_DATA`) gibt Auskunft darüber, wie Ganzzahlen in der ELF-Datei codiert sind. Da ULIX für die Intel-x86-Architektur entwickelt wird müssen die Daten der ELF-Datei in 2er-Kompliment-Darstellung und *little-endian*-Byte-Reihenfolge codiert sein. Somit wird im obigen Codeabschnitt geprüft, dass auch die Daten der ELF-Dateien in diesem Format vorliegen. Sollte eine der Bedingungen der `if`-Abfrage des *Code-Chunk* nicht erfüllt sein, so wird der Ladevorgang der ELF-Datei abgebrochen und der *Syscall-Handler* verlassen. Um den aufgetretenen Fehler im nachfolgend ausgeführten Programmcode identifizieren zu können, müsste eigentlich der Error-Code `ENOEXEC` gesetzt werden. Da es in der aktuellen Version von ULIX-i386 noch keine Möglichkeit gibt einen Error-Code zu setzen, wurde die betreffende Stelle nur als Kommentar eingefügt. Die Variable `errno` wäre dabei eine globale - in der Systembibliothek definierte - Variable. Durch das setzen des Wertes `-1` in der Strukturkomponenten `r->eax` wird angezeigt, dass ein Fehler aufgetreten ist.

Typdefinition des ELF-Programm-Header ELF-Programm-Header enthalten Informationen über die Programm-Segmente einer ELF-Datei. Für jedes Programm-Segment existiert ein *Programm-Header*, der benötigt wird, um auf das jeweilige Segment zuzugreifen. Die folgende Struktur beschreibt den Aufbau eines ELF-Programm-Headers.

18 ⟨*ELF-Typ-Definitionen* 15⟩+≡ (31a) ◁16 59a▷

```
// Elf32_Phdr     (ELF-Programm-header)
// ----------------------------------------------------------------------------
//
typedef struct {
  Elf32_Word      p_type;     // Beschreibt den Zweck des Programm-Header
  Elf32_Off       p_offset;   // Offset zum Segment
  Elf32_Addr      p_vaddr;    // Virtuelle Adresse des Segments im Prozessimage
  Elf32_Addr      p_paddr;    // Physikalische Adresse (wird nicht verwendet)
  Elf32_Word      p_filesz;   // Groesse des Segments in der ELF-Datei (in Byte)
  Elf32_Word      p_memsz;    // Groesse des Segments im Prozessimage (in Bytes)
  Elf32_Word      p_flags;    // Read / Write / Execute
  Elf32_Word      p_align;    // Memory alignment
} Elf32_Phdr;
```

Defines:
`Elf32_Phdr`, used in chunks 41, 42, 61a, and 62.
Uses `Elf32_Addr` 15, `Elf32_Off` 15, and `Elf32_Word` 15.

ELF-Programm-Header sind, wie schon in Kapitel 1.3.1 beschrieben, nur in ausführbaren ELF-Dateien vorhanden bzw. werden nur in solchen Dateien benötigt. Sie stehen in ausführbaren ELF-Dateien direkt im Anschluss an den ELF-Header. Genau genommen handelt es sich um eine Tabelle von Programm-Headern, wobei die Größe der Tabelle variabel ist. Die Anzahl der vorhandenen ELF-Programm-Header ist im ELF-Header enthalten. Dort wird

auch festgelegt, wie groß ein einzelner ELF-Programm-Header ist (siehe Listing 1.3). Jeder ELF-Programm-Header einer ELF-Datei repräsentiert dabei einen Bereich innerhalb der ELF-Datei. Diese Bereiche werden *Segmente* genannt. Die *Segmente* sind diejenigen Daten, die aus der ELF-Datei in das *Prozessimage* eines Prozesses kopiert werden müssen, damit das in der ELF-Datei definierte Programm ausgeführt werden kann. Die *Programm-Header* enthalten dabei alle nötigen Informationen um die *Segmente* in ein *Prozessimage* zu laden.

1.3.3. ELF-Beispielprogramm

Um die Ausführungen in dieser Arbeit anschaulicher zu machen, wird hier ein kurzes C-Programm definiert und nachfolgend zur Erklärung des ELF und der Funktionsweise des ELF-Programm-Loaders verwendet. Dieses C-Programm wird außerdem auch für den Test des ELF-Programm-Loaders verwendet.

19a ⟨*elfbin Beispielprogramm 19a*⟩≡ (32b) 19b ▷
```
#include "../Apps/C/ulixlib.h"
int x = 106;  // hex: 0x6A
```
Defines:
x, used in chunks 19c and 61c.

Der Wert von `x` wird sich im *.data*-Segment der ELF-Datei wiederfinden.

19b ⟨*elfbin Beispielprogramm 19a*⟩+≡ (32b) ◁19a 19c ▷
```
#define __NR_printchar      0x1001
#define __NR_printstack     0x1137
```
Defines:
__NR_printchar, never used.
__NR_printstack, used in chunk 33.

19c ⟨*elfbin Beispielprogramm 19a*⟩+≡ (32b) ◁19b
```
int main (int argc, char** argv)
{
  int y = 44015;  // hex: 0xabef
  int i;

  printf("### Teststring Ausgabe\n");
  printf("Wert von x: %5d 0x%x\n",x,x);
  printf("Wert von y: %5d 0x%x\n",y,y);
  printf("Anzahl der Argumente von elfbin (argc): %d\n",argc);
  printf("Argumente von elfbin (argv):\n");
  for ( i=0; i<argc; i++ )
    printf("\targv[%d]: %s\n",i,argv[i]);

  exit(argc);
}
```
Defines:
main, used in chunks 20 and 47b.
Uses x 19a.

Die *main*-Funktion des Beispielprogramms ist sehr einfach gehalten. Sie enthält lediglich eine Variablendeklaration, einige Textausgaben, die für den Test verwendet werden und einen Aufruf *exit*-Funktion, um das Programm wieder zu verlassen. Den Wert `0xabef` der Variablen `y` kann man später im *.text*-Segment der ELF-Datei wiederfinden. Der Aufruf der Funktion *exit* beendet das Programm. Da es sich bei *exit* um eine Funktion aus der Systembibliothek *ulixlib* handelt, ist es erforderlich, dass das Beispielprogramm mit *ulixlib* statisch zusammen gelinkt wird, damit in der daraus resultierenden ELF-Datei sowohl der Code des Beispielprogramms als auch der Code der *exit*-Funktion aus *ulixlib* enthalten sind.

Damit das Beispielprogramm `elfbin` und das Testprogramm `UlixTestRun` auf einfache Weise übersetzt werden können wird nachfolgend ein Makefile erstellt, welches die nötigen Compiler- und Linker-Anweisungen enhält, um diese beiden Programme zu erstellen.

20 ⟨*Makefile für Beispiel- und Test-Programm* 20⟩≡ (32c)

```
CC=gcc
RM=rm -f

LDOPTIONS=--entry=main,-Ttext=0

COPTIONS=-O0 -fstrength-reduce -finline-functions -nostdinc
COPTIONS+= -fno-builtin -nostdlib
COPTIONS+= -Wl,
CCOPTIONS=$(COPTIONS)$(LDOPTIONS)

INC    = ./../Apps/C/
LIBOBJ = ../Apps/C/ulixlib.o
EXE1   = elfbin
EXE2   = UlixTestRun

all: prog

prog: $(EXE1) $(EXE2)

$(EXE1):
        $(CC) -I$(INC) $(CCOPTIONS) $(LIBOBJ) $(EXE1).c -o $(EXE1)

$(EXE2):
        $(CC) -I$(INC) $(CCOPTIONS) $(LIBOBJ) $(EXE2).c -o $(EXE2)

clean:
        $(RM) *.o *.out $(EXE1) $(EXE2)

.PHONY: all prog clean
```

Uses `main` 19c.

1.4. Erstellung von Programmdateien für Ulix-i386

Da die Programme, die unter ULIX-i386 ausgeführt werden sollen in einer Linux-Umgebung (siehe Kapitel 2.2.1) erzeugt werden, müssen diese mit speziellen Compiler- und Linker-Flags übersetzt werden. Der Grund dafür ist, dass den ausführbaren Dateien beim Linken der Objekt-Dateien Funktionen aus der Linux-stdlib hinzugefügt werden, die nicht mit ULIX-i386 kompatibel sind. Um dies zu verhindern werden zur Übersetzung folgende Compliler-/ und Linker-Flags verwendet:

a) **Compiler-Flags:**
`-O0 -fstrength-reduce -finline-functions -nostdinc -fno-builtin -nostdlib`

Beschreibung der Compiler-Flags:

`[-O0:]` deaktiviert zunächst alle vom Compiler durchgeführten Optimierungen.

`[-fstrength-reduce]` aktiviert die Optimierung von Schleifen wieder.

`[-finline-functions]` erlaubt dem Compiler *inline-Funktionen* zu verwenden.

`[-nostdinc -nostdlib]` verbieten dem Compiler, in den Standard-Include-/Library-Verzeichnissen des Systems nach Header-/ bzw. Bibliotheksdateien zu suchen.

`[-fno-builtin]` verbietet dem Compiler eine Sonderbehandlung von bestimmten builtin-Funktionen.

b) **Linker-Flags:**
`--entry=main,-Ttext=0`

Beschreibung der Linker-Flags:

`[--entry=main]` sorgt dafür, dass der *Einsprungpunkt* für das Programm auf die Adresse der *main*-Funktion gesetzt wird. Das bedeutet, der ELF-Header der Ausgabedatei enthält als *Einsprungpunkt* die Adresse in der Binärdatei, an der die *main*-Funktion steht.

`[-Ttext=0]` bewirkt, dass die virtuelle Zieladresse für das *.text*-Segment auf den Wert `0x00000000` gesetzt wird.

Durch die genannten Compiler- und Linker-Optionen wird eine Binärdatei erzeugt, die im Wesentlichen nur noch aus Programmcode und Programmdaten besteht. Das Ziel dabei ist es, eine möglichst einfache Binärdatei im ELF ohne besondere Optimierungen oder Informationen, die nicht direkt für die Ausführung des Programms benötigt werden, zu erzeugen.

2. Entwicklungsumgebung

2.1. Aufbau und Komponenten

Eine Übersicht der Entwicklungs-/Testumgebung ist in Abbildung 2.1 dargestellt.

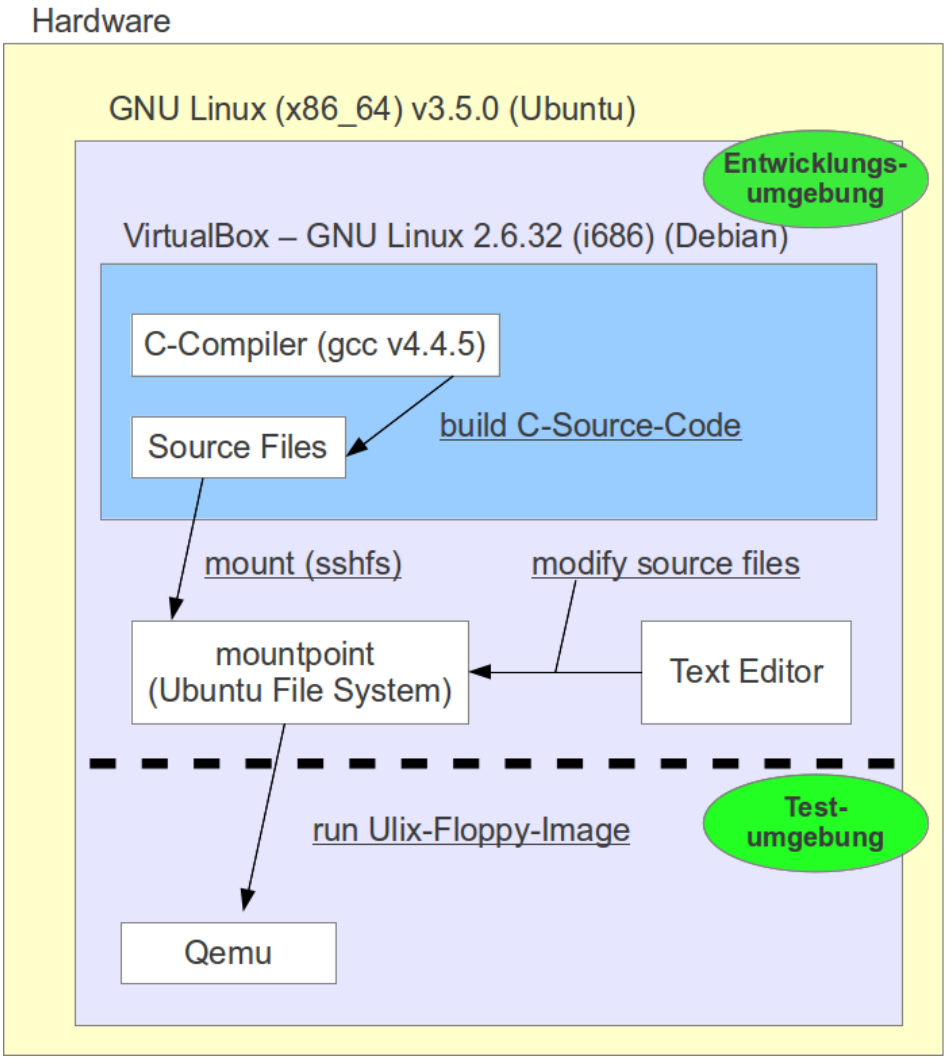

Abbildung 2.1.: Ulix-i386 Development/Test Environment

Die Entwicklungsumgebung, in der der Ulix-i386 ELF-Programm-Loader entwickelt wird, lässt sich in drei Bereiche unterteilen (siehe Abbildung 2.1). Auf der Hardware des Entwicklungs-

PCs ist eine 64-Bit Version von Ubuntu-Linux installiert. Für ULIX-i386 muss allerdings 32-Bit Programmcode erzeugt werden. Aus diesem Grund wird für die Erzeugung des ULIX-i386 Programmcodes ein kleines GNU-Debian-Linux 32-Bit System innerhalb einer virtuellen Maschine (*VirtualBox* [Ora]) installiert. In diesem Debian System werden alle für die Übersetzung des Programmcodes benötigten Tools wie C-Compiler, Linker, Assembler etc. installiert. Das GNU-Debian-System wird also ausschließlich zur Übersetzung des ULIX-i386 Programmcodes verwendet. Auf diese Weise wird die Verwendung von Cross-Compilern vermieden. Die eigentliche Entwicklung des ULIX-i386 Programmcodes findet in dem Ubuntu-Linux System statt. Dafür werden unter Linux verfügbare Texteditoren wie z.B. `gedit` oder `kate` verwendet. Das Verzeichnis, in dem die ULIX-i386 Sourcecode-Dateien abgelegt sind, befindet sich im Dateisystem des Debian-Systems. Damit das Ubuntu-Linux System auf die Sourcecode-Dateien von ULIX-i386 zugreifen kann, wird das Sourcecode Verzeichnis mit dem Linux-Tool `sshfs` in das Dateisystem des Ubuntu-Linux Systems eingebunden. Somit können beide Systeme so auf die Sourcecode-Dateien von ULIX-i386 zugreifen als würden sich die Sourcecode-Dateien im lokalen Dateisystem befinden.

2.2. Hostsystem

Bei dem Hostsystem der Entwicklungsumgebung handelt es sich, wie schon im vorherigen Abschnitt erwähnt, um ein Ubuntu-Linux (x86_64). Theoretisch könnte aber auch ein anderes Betriebssystem eingesetzt werden, unter der Voraussetzung, dass es den Betrieb einer virtuellen Maschine mit *VirtualBox* und den Hardwareemulator *qemu* [Bel] unterstützt. Da hier die genannte Linux-Distribution Ubuntu zum Einsatz kommt, werden nachfolgend die verwendeten Komponenten und deren Einsatz erläutert.

Weil das Hostsystem nicht an dem Übersetzungsvorgang von ULIX-i386 beteiligt ist, werden auf dem Hostsystem auch keine Compiler, Linker, Bibliotheken oder sonstige Komponenten, die Einfluss auf den Übersetzungsvorgang haben benötigt. Das Hostsystem dient lediglich als Grundlage für die eingesetzten virtuellen Maschinen und um den eigentlich Sourcecode des ULIX-i386-Moduls und der Dokumentation zu erstellen. Damit die Dateien des Sourcecodes aus dem Hostsystem heraus bearbeitet werden können, benötigt das Hostsystem einen virtuellen Netzwerkadapter, der sich im selben Netzwerk befindet wie der Netzwerkadapter der virtuellen Maschine. Einen solchen virtuellen Netzwerkadapter stellt *VirtualBox* zur Verfügung. Nachdem sich nun das Hostsystem und das Betriebssystem der virtuellen Maschine in einem gemeinsamen Netzwerk befinden kommt das Linux-Tool `sshfs` zum Einsatz. `sshfs` erlaubt es über ein Netzwerk ein entferntes Verzeichnis in das Dateisystem des lokalen Betriebssystems einzubinden. Um eine Verbindung zu der virtuellen Maschine aufzubauen, muss auf dem Gastsystem ein aktiver ssh-Server zur Verfügung stehen. Für die Anmeldung an

dem entfernten System ist ein Benutzeraccount des Gastsystems erforderlich. Im konkreten Fall wird das entsprechende Verzeichnis der virtuellen Maschine in das lokale Verzeichnis `~/ulixdev/` eingebunden. Für die Erstellung des ULIX-i386-Sourcecodes kann jetzt ein beliebiger Texteditor, der auf dem Hostsystem zur Verfügung steht, verwendet werden. Hier wurde für diesen Zweck `gedit` verwendet.

2.2.1. Gastsystem (VM / Compilerumgebung)

Das Gastsystem der virtuellen Maschine ist ein GNU Debian Linux (x86). Dieses System dient in erster Linie dazu, den C-Sourcecode von ULIX-i386 in x86 Maschinencode zu übersetzen. Für den Übersetzungsvorgang kommen die Unix-Tools `make` und `gcc` zum Einsatz. Außerdem werden auch die beiden im Kapitel 1.2 erläuterten NoWEB-Tools `notangle` und `noweave` verwendet. Da auf das System der virtuellen Maschine über das interne Netzwerk zugegriffen werden soll, muss auch ein ssh-Server auf dem Gastsystem der virtuellen Maschine installiert werden. Die benötigten Linux-Tools sind alle in der Paketverwaltung des Gastsystems vorhanden und können unter geringem Aufwand mit `APT` installiert werden. Außer den genannten Linux-Tools besteht das Gastsystem aus einer minimalen Standardinstallation von GNU-Debian-Linux. Es enthält also auch keine grafische Oberfläche oder sonstige Komponenten, die unnötig Speicherplatz verbrauchen würden.

2.2.2. Laufzeit-/ Testumgebung

Die Laufzeit- bzw. Testumgebung ist gleichzeitig die Entwicklungsumgebung des Hostsystems. Auf diesem wird ULIX-i386 mittels des Hardwareemulators `qemu` ausgeführt. Um ULIX-i386 in `qemu` zu starten, werden zwei Disketten-Images verwendet. Diese Images sind in den Dateien `ulixboot.img` und `ulixdata.img` enthalten. Die Datei `ulixboot.img` enthält den eigentlichen Maschinencode des ULIX-i386-Betriebssystems und wird beim Start von `qemu` ausgeführt. Das Disketten-Image `ulixdata.img` enthält die Dateien `elfbin`, `UlixTestRun` und einige modifizierte Versionen von `elfbin`, die für die Tests des Programm-Loaders verwendet werden. Dieses Image bildet das Dateisystem von ULIX-i386, auf welches der ELF-Programm-Loader zugreift, um Programme im ELF zu laden. Der Aufruf um ULIX-i386 mittels `qemu` zu starten ist in Listing 2.1 dargestellt.

```
$ qemu -m 64 -fda ulixboot.img -fdb ulixdata.img -d cpu_reset \
     -s -serial mon:stdio -serial tcp::4444,server \
     -hda ulixshell.img | tee ulix.output
```

Listings 2.1: Testumgebung – QEMU Start

Bei genauerer Betrachtung des Aufrufs von `qemu` fällt auf, dass dabei eine zusätzliche Datei mit dem Namen `ulixshell.img` beteiligt ist. Diese Datei enthält die in Ulix-i386 verwendete Shell, welche nach dem Systemstart aufgerufen wird. Die Notwendigkeit warum die Ulix-Shell durch diesen zusätzlichen Mechanismus an `qemu` übergeben werden muss, liegt in der Entwicklungshistorie von Ulix-i386 begründet. Es handelt sich dabei lediglich um eine weitere Möglichkeit, um aus dem Ulix-i386-Kernel heraus auf eine externe Datei zuzugreifen. In zukünftigen Versionen von Ulix-i386 wird diese Methode nicht mehr verwendet. Stattdessen wird die Ulix-i386-Shell ebenfalls in `ulixdata.img` gespeichert. Aus diesem Grund wird auf diese Methode des externen Dateizugriffs nicht weiter eingegangen. Nachdem Ulix-i386 gestartet wurde, wird der Benutzer von der Ulix-i386-Shell zur Eingabe von Befehlen aufgefordert. Die Situation direkt nach dem Ulix-i386-Systemstart zeigt Listing 2.2.

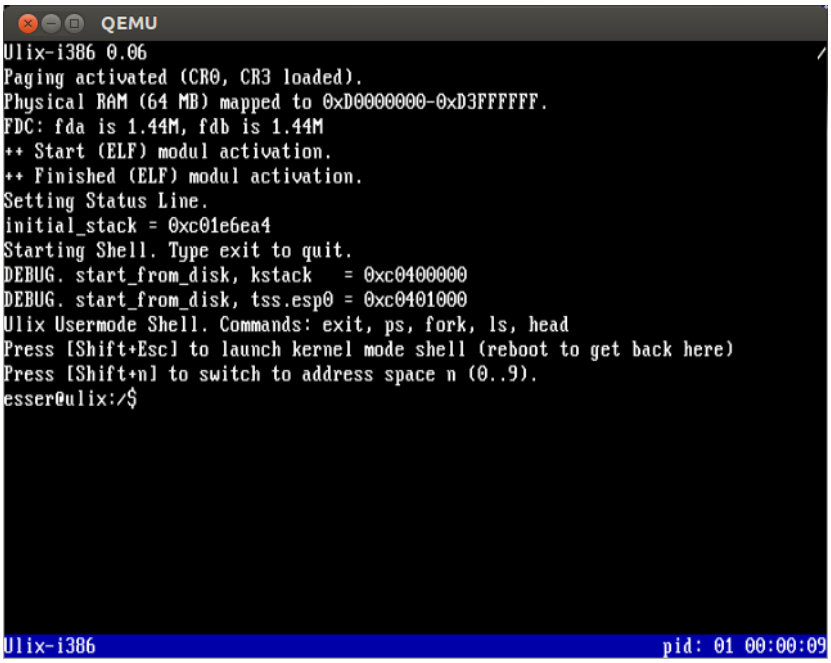

Abbildung 2.2.: Testumgebung – QEMU Ulix-i386-Shell

Die Ulix-i386-Shell unterstützt in der aktuellen Version die von Unix-artigen Systemen bekannten Kommandos `exit`, `ps`, `ls` und `head`. Zusätzlich ist das Testkommando `fork` verfügbar, um den *Systemaufruf* `fork` zu testen. Diese Kommandos sind in der Shell integriert (builtin). Das Kommando `ls` listet die in dem Floppyimage `ulixdata.img` enthaltenen Dateien auf. Die in dem Floppyimage enthaltenen ELF-Programm-Dateien können über ihren Dateinamen in der Shell aufgerufen und ausgeführt werden. Das Laden und Ausführen der ELF-Programme übernimmt dabei der in dieser Arbeit implementierte ELF-Programm-Loader.

2.3. Übersetzung der Source-Dateien

2.3.1. NoWEB-Source-Verarbeitung

Die von NoWEB verarbeitete Source-Datei enthält den gesamten LaTeX-Sourcecode der Dokumentation und den C-Sourcecode des Ulix-i386-ELF-Programm-Loaders. Um die Dokumentation und den Ulix-i386-Maschinencode erzeugen zu können, muss der jeweilige Sourcecode aus der NoWEB-Source-Datei extrahiert werden. Dies geschieht mittels der in Listing 2.2 und Listing 2.3 dargestellten Kommandos. An dieser Stelle wird darauf hingewiesen, dass die abgebildeten Kommandos nicht ganz den tatsächlichen Aufrufen entsprechen. Der Grund dafür ist, dass die Kommandos in Makefiles integriert sind und deshalb für Datei- und Kommandonamen teilweise Makefile-Variablen zum Einsatz kommen. Um die dargestellten Abbildungen übersichtlicher zu gestalten, wurden diese Variablen durch die tatsächlichen Namen der Kommandos und Dateien ersetzt.

```
$ notangle -L -Relfbin.c bachelorarbeit.nw > elfbin.c
$ notangle -L -UlixTestRun.c bachelorarbeit.nw > UlixTestRun.c
$ notangle -L -RMakefile.elf bachelorarbeit.nw > Makefile
```

Listings 2.2: Extrahieren des C-Sourcecode

Die drei Aufrufe von `notangle` extrahieren aus der Datei `bachelorarbeit.nw` die beiden C-Sourcecode-Dateien `elfbin.c`, `UlixTestRun.c` und das Makefile der beiden C-Programme. Die verwendeten Aufrufparameter von `notangle` wurden bereits im Kapitel 1.2 erläutert. Wie die beiden Programme in Maschinencode übersetzt werden, wird in Kapitel 2.3.2 beschrieben.

```
$ noweave -autodefs c -index -delay bachelorarbeit.nw > bachelorarbeit.tex
```

Listings 2.3: Extrahieren der Dokumentation

Der Aufruf von `noweave` erzeugt aus der Ursprungs-Datei `bachelorarbeit.nw` die LaTeX-Datei `bachelorarbeit.tex`. In dieser `.tex`-Datei sind die LaTeX-Anweisungen der Dokumentation inklusive des C-Sourcecodes in druckbarer Form enthalten. Sie dient im Kapitel 2.3.3 zur Erzeugung der Dokumentation als PDF-Dokument.

2.3.2. Erzeugung des Ulix-i386-OS-Code

Nachdem im Kapitel 2.3.1 der Sourcecode mithilfe von NoWEB in Dokumentations- und Programmcode aufgeteilt wurde, kann nun der Programmcode für den Ulix-i386-Programm-Loader übersetzt werden. Der Programmcode des Ulix-i386-Moduls besteht aus drei C-Sourcecode-Dateien. Diese Dateien sind:

- **elf-types.h** – Typdefinitionen für die Verarbeitung von ELF-Dateien
- **module.h** – Headerdatei des ELF-Moduls
- **module.c** – Sourcecode des ELF-Moduls

Um diese Sourcecode-Dateien zu übersetzen und ein lauffähigen ULIX-i386-Image zu erzeugen, muss der C-Sourcecode des ELF-Loader-Moduls in Maschinencode übersetzt werden. Dies erfolgt mit dem in Listing 2.4 dargestellten Aufruf des `gcc`-Compilers.

```
$ gcc -D ULIX -std=c99 -g -O -fstrength-reduce -fomit-frame-pointer \
     -finline-functions -nostdlib -nostdinc -fno-builtin -I./include \
     -c -o module.o module.c
```

Listings 2.4: Kompilieren des Programm-Loader Sourcecode

Der Compileraufruf erzeugt die Objekt-Datei `module.o`. Diese Datei enthält den Objekt-Code des ELF-Programm-Loaders und muss nachfolgend zu dem Objekt-Code von ULIX-i386 gelinkt werden. Der Objekt-Code von ULIX-i386 ist in der Datei `ulix.o` enthalten. Durch den in Listing 2.5 dargestellten Linder-Aufruf wird der Objekt-Code der beiden genannten Dateien zu einer ausführbaren Binärdatei zusammengefasst.

```
$ ld -T ulix.ld -o ulix.bin *.o \
     ATA-mindrvr/mindrvr/mindrvr.o | tee ld.log
```

Listings 2.5: Linken des ULIX-i386 Objekt-Code

Auch im Fall des hier abgebildeten Linker-Aufrufs wird darauf hingewiesen, dass der Aufruf tatsächlich in einem Makefile stattfindet. Wie schon in Kapitel 2.3.1 erwähnt wurden deshalb Makefile-Variablen durch die tatsächlichen Kommandos ersetzt. Aus dem dargestellten Linker-Aufruf ist außerdem ersichtlich, dass noch weitere Objekt-Code-Dateien am Linkvorgang beteiligt sind. Auf diese Objekt-Dateien wird in diesem Dokument nicht weiter eingegangen, da sie für das ELF-Modul von ULIX-i386 nicht relevant sind. Der vorhergehende Linker-Aufruf erzeugt eine Datei mit dem Namen `ulix.bin`. Diese Datei wird verwendet, um ein bootfähiges Floppyimage zu erstellen, welches mithilfe von `qemu` gestartet werden kann.

2.3.3. Erzeugung der Dokumentation

In Listing 2.3 wurde mithilfe des Kommandos `noweave` aus einer NoWEB-Datei eine LaTeX-Datei extrahiert. Diese LaTeX-Datei ist die Grundlage für die Erzeugung der PDF-Dokumentation. Die Schritte, um aus der LaTeX-Datei eine PDF-Datei zu generieren, unterscheide

sich durch die Beteiligung von NoWEB nicht von der Verarbeitung einer gewöhnlichen LaTeX-Datei. Allerdings wird das LaTeX-Paket `noweb.sty` benötigt, um die Code-Chunks des *Literate Program* zu verarbeiten. Diese Paket-Datei muss im Aufrufverzeichnis oder im Modulsuchpfad von LaTeX enthalten sein. Die Aufrufe, um aus dem LaTeX-„Sourcecode“ eine PDF-Datei zu erzeugen, werden in Listing 2.6 gezeigt.

```
$ pdflatex bachelorarbeit.tex
$ bibtex bachelorarbeit
$ makeindex bachelorarbeit
$ pdflatex bachelorarbeit.tex
$ pdflatex bachelorarbeit.tex
```

Listings 2.6: Erzeugen der PDF-Dokumentation (LaTeX)

Dass hier das Kommando `pdflatex` mehrfach aufgerufen werden muss, liegt in der Arbeitsweise von LaTeX begründet. Deshalb wird an dieser Stelle nur auf die LaTeX-Dokumentation verwiesen [lat]. Als Ergebnis der Kommando-Aufrufe in Listing 2.6 wird eine PDF-Datei erzeugt, welche die Dokumentation des ELF-Programm-Loaders inklusive dessen Programmcodes enthält.

3. Implementation

Der Sourcecode des ULIX-i386-ELF-Programm-Loaders besteht aus insgesamt drei Sourcecode-Dateien. Diese sind `elf-types.h`, `module.h` und `module.c`. Die Header-Datei `elf-types.h` enthält Definitionen und Typdeklarationen, die für ELF notwendig sind. Der eigentliche Code des ELF-Programm-Loaders befindet sich in den Dateien `module.h` und `module.c`. Die Datei `module.h` beinhaltet hauptsächlich Definitionen, Typdeklarationen und Vorausdeklarationen von Funktionen, die aus ULIX-i386 stammen und vom Programm-Loader verwendet werden. In der Datei `module.c` befindet sich der eigentlich Quellcode des Programm-Loaders, der für den Zugriff und die Verarbeitung von ELF-Dateien verantwortlich ist. Damit diese drei Dateien erzeugt werden, muss hier für jede dieser Dateien ein eigener *Code-Chunk* angelegt werden.

31a ⟨*elf-types.h* 31a⟩≡

```
#ifndef _ELF_TYPES_H_
#define _ELF_TYPES_H_
#define USE_HELPER_FUNCTIONS
⟨ELF-Definitionen 58⟩
⟨ELF-Typ-Definitionen 15⟩
#endif // _ELF_TYPES_H_
```

Defines:
  `_ELF_TYPES_H_`, never used.
  `USE_HELPER_FUNCTIONS`, used in chunks 31–33.

31b ⟨*module.h* 31b⟩≡

```
/* module.h */
/* Header Datei ELF Program Loader */

⟨Externe Definitionen (ULIX) 56a⟩
⟨Externe Typ-Definitionen (ULIX) 56b⟩
⟨Externe Deklarationen (ULIX) 55⟩
⟨ELF-Programm-Loader Definitionen 59b⟩
#ifdef USE_HELPER_FUNCTIONS
⟨Hilfsfunktionen (Deklaration) 61a⟩
#endif
```

Uses `USE_HELPER_FUNCTIONS` 31a.

32a ⟨*module.c* 32a⟩≡
```
/* Code */
#include "elf-types.h"
#include "module.h"

⟨initialize module 33⟩
⟨syscall handler 35⟩
⟨elf functions (never defined)⟩
#ifdef USE_HELPER_FUNCTIONS
⟨Hilfsfunktionen (Implementation) 61b⟩
#endif
```
Uses USE_HELPER_FUNCTIONS 31a.

Die *Code-Chunks* der einzelnen Quelldateien enthalten selbst auch wieder *Code-Chunks*, welche die Gesamtaufgabe des Programm-Loaders in kleinere Teile aufteilen und an anderen Stellen im Dokument definiert werden. An dieser Stelle des Dokumentes befindet sich gewissermaßen die „Sammelstelle“, an der alle *Code-Chunks* des Dokumentes zusammengeführt werden.

Neben den bisher erläuterten Quelldateien werden für das Beispielprogramm und das Testprogramm eigene Quellcode-Dateien erzeugt. Dies sind die beiden Dateien `elfbin.c` und `UlixTestRun.c`. Um die Regeln, nach denen die beiden Programme übersetzt werden müssen, ebenfalls in diesem Dokument festlegen zu können, wird außerdem ein Makefile erzeugt, welches die *Compiler*- und *Linker*-Anweisungen für beide Beispielprogramme enthält.

32b ⟨*elfbin.c* 32b⟩≡
```
⟨elfbin Beispielprogramm 19a⟩
```

32c ⟨*Makefile.elf* 32c⟩≡
```
⟨Makefile für Beispiel- und Test-Programm 20⟩
```

3.1. Initialisierung des Programm-Loaders

Damit Ulix-i386 den in diesem Dokument beschriebenen und implementierten Programm-Loader verwenden kann, muss er beim Systemstart von Ulix-i386 initialisiert werden. Um dies zu ermöglichen, muss ein Ulix-i386-Modul eine Funktion mit dem Namen `initialize_module` definieren. Diese Funktion wird beim Ulix-i386-Systemstart für jedes Modul aufgerufen. In der Funktion `initialize_module` kann jedes Modul dann selbst definieren, was für seine Initialisierung nötig ist Im Fall des ELF-Programm-Loaders bedeutet das nur die Registrierung eines Systemaufrufs. Ulix-i386 stellt dafür die Funktion `insert_syscall` zur Verfügung. Auf diese Weise können Systemaufrufe und deren Behandlung dynamisch registriert und definiert werden und können so aus dem Hauptdokument von Ulix-i386 ausgelagert werden.

Im folgenden *Code-Chunk* wird dem System in der Funktion `initialize_module` mitgeteilt, dass es für den *Systemaufruf exec* einen *Syscall-Handler* `syscall_exec` gibt. Dieser *Syscall-Handler* wird in diesem Modul/Dokument definiert und wird ausgeführt, wenn der *Systemaufruf exec* von einem Programm aufgerufen wird. Desweiteren wird zu Diagnosezwecken ein *Syscall-Handler* `syscall_printstack` registriert. Dieser *Syscall-Handler* wird der Nummer `__NR_printstack` zugewiesen. Er dient dazu, den Usermode-Stack des laufenden Programms auszugeben. Der Syscall `__NR_printstack` wird in der finalen Version von Ulix-i386 nicht benötigt und sollte entfernt werden.

33 ⟨*initialize module* 33⟩≡ (32a)

```
void initialize_module () {
  #ifdef ELF_DEBUG_MSG
  printf ("++ Start (ELF) modul activation.\n");
  #endif

  insert_syscall(__NR_exec, syscall_exec);
  #ifdef USE_HELPER_FUNCTIONS
  insert_syscall(__NR_printstack, syscall_printstack);
  #endif
  #ifdef ELF_DEBUG_MSG
  printf ("++ Finished (ELF) modul activation.\n");
  #endif
}
```

Defines:
`initialize_module`, used in chunk 59b.
Uses `__NR_exec` 59b, `__NR_printstack` 19b 59b, `syscall_exec` 35, `syscall_printstack` 61b, and `USE_HELPER_FUNCTIONS` 31a.

Damit ist das ELF-Programm-Loader-Modul in Ulix-i386 registriert. Für die Initialisierung des Moduls sind keine weiteren Schritte nötig. Nachfolgend muss definiert werden, was geschehen soll, wenn ein *exec* Syscall aufgerufen wird.

3.2. Programmcode laden

Das Laden eines ELF-Programms erfolgt durch den *Systemaufruf exec*. Bevor das eigentliche Laden und Starten eines ELF-Programms erläutert wird, muss zunächst auf das Verhalten von Ulix-i386 beim Aufruf eines Systemcalls eingegangen werden.

3.2.1. Systemaufruf Interface in Ulix-i386

Das *Syscall-Interface* von Ulix-i386 sieht vor, dass Aufrufparameter in Registern an den *Syscall-Handler* übergeben werden. Der Syscall *exec* erwartet einen Aufrufparameter, der ein Zeiger auf den Dateinamen des auszuführenden ELF-Programms enthält, und einen Zeiger,

der auf ein Array von Zeigern, die auf die Aufruf-Argumente des Programms zeigen, zeigt. Soll ein Usermode-Programm einen *exec*-Systemaufruf auslösen, so muss es zunächst die Nummer des Systemaufrufs in das Register *eax* laden. Im Fall des *exec*-Systemaufrufs ist dies die Systemaufrufnummer `__NR_exec`. Weiter lädt das Usermode-Programm einen Zeiger auf den Dateinamen in das Register *ebx* und den Zeiger auf die Argumente in das Register *ecx*. Im Anschluss kann das Usermode-Programm den *Interrupt* `0x80` auslösen. Dieser sorgt dafür, dass die *Interrupt Service Routine* (*ISR*) für den *Interrupt* „0x80“ ausgeführt wird. Diese *ISR* schaltet die CPU in den *Privileged Mode* (Ring 0). Außerdem erzeugt die ISR_128 (dezimal: 128 = 0x80) eine Struktur, die den CPU-Kontext zum Zeitpunkt des Systemaufrufs enthält. Diese Struktur wird von der *Interrupt Service Routine* mittels eines Zeigers an einen generischen *Syscall-Handler* übergeben. Der generische *Syscall-Handler* ermittelt anhand der Systemaufrufnummer den speziellen *Syscall-Handler* für den jeweiligen *Systemaufruf*. Bei dem speziellen *Syscall-Handler* handelt es sich im Fall des Systemaufrufs *exec* um die hier definierte Funktion `syscall_exec`. Da der generische *Syscall-Handler* die von der ISR erhaltene Struktur auch an den speziellen *Syscall-Handler* weitergibt, können in der hier definierten Funktion `syscall_exec` die im Usermode-Programm übergebenen Aufrufparameter ermittelt werden. Der Zeiger auf den Programm-Dateinamen befindet sich in der Struktur-Komponente `r->ebx`. In der Struktur-Komponente `r->ecx` befindet sich ein Zeiger auf das Zeigerfeld, welches Zeiger auf die Aufruf-Argumente enthält. Die Struktur mit den CPU-Registern wird im unten definierten *Syscall-Handler* über die Zeigervariable `r` angesprochen und hat den Typ `struct regs*`. Die Definition von `struct regs` befindet sich im Source-Code von ULIX-i386, muss aber in diesem Dokument wiederholt werden, um dem Compiler zu ermöglichen, die Größe einer solchen Struktur zu berechnen. Um einen Wert aus einem *Syscall-Handler* zurückzugeben wird der gewünschte Rückgabewert in die Strukturvariable `r->eax`, der vom generischen *Syscall-Handler* übergebenen Struktur geschrieben und der *Syscall-Handler* mit *return* verlassen.

34 ⟨*Struktur für Zugriff auf Kernel-Stack* 34⟩≡ (56b)

```
struct regs {
  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;
};
```

Uses `uint` 56b 56b.

3.2.2. Syscall-Handler (exec)

Der folgende Programmcode definiert den *Syscall-Handler* des *exec* Systemaufrufs. Die Aufgabe des Syscall-Handlers wird in mehrere Teilbereiche zerlegt. Diese Teilaufgaben werden durch die unten definierten *Code-Chunks* repräsentiert. Die einzelnen *Code-Chunks* werden

im weiteren Verlauf dieses Kapitels implementiert und erläutert.

35 ⟨*syscall handler* 35⟩≡ (32a)

```
void syscall_exec (struct regs *r) {
  ⟨Aufrufargumente prüfen 36a⟩
  ⟨Aufrufargumente sichern 36c⟩
  ⟨ELF-Datei in Speicher kopieren 37b⟩
  ⟨ELF-Datei überprüfen 17⟩
  ⟨Dynamisch allozierten Speicher freigeben 40b⟩
  ⟨Prozessimage ersetzen 41⟩
  ⟨Aufrufargumente in Heap-Speicher kopieren 45a⟩
  ⟨Usermode-Stack initialisieren 45b⟩
  return;
};
```

Defines:
`syscall_exec`, used in chunks 33 and 59b.

Der *Syscall-Handler* bekommt über den Strukturzeiger `struct regs *r` zwei wichtige Informationen über das auszuführende Programm. Diese Informationen sind der Name des Programms und ein Array von Zeigern auf die Aufruf-Argumente des Programms. Damit der Stack im späteren Verlauf so vorbereitet werden kann, dass der *main*-Funktion ihre jeweiligen Argumente zur Verfügung stehen, müssen diese gesichert werden, bevor das *Prozessimage* durch das des neuen Programms ersetzt wird.

Einer *main*-Funktion werden immer zwei Argumente übergeben. Das erste Argument ist `int argc`. Das zweite Argument ist `char* argv[]`. Diese beiden Argumente müssen an der richtigen Position auf den Stack gelegt werden, damit die *main*-Funktion während ihrer Ausführung darauf zugreifen kann. Die *exec*-Man-Page gibt zu `char* argv[]` folgende Informationen.

> *The execv(), execvp(), and execvpe() functions provide an array of pointers to null-terminated strings that represent the argument list available to the new program. The first argument, by convention, should point to the filename associated with the file being executed. The array of pointers must be terminated by a NULL pointer.* [mana]

Da nicht nur die Zeichenketten der Argumente selbst NULL-terminiert sein müssen, sondern auch das Zeigerarray – welches auf die Argumente zeigt – durch eine NULL-Terminierung abgeschlossen wird, kann durch eine Iteration über alle Argumente – bis zur NULL-Terminierung – die Anzahl (`argc`) der Argumente ermittelt werden. Der Zeiger auf das Zeigerarray der Argumente befindet sich dabei in `r->ecx`. Bevor der Programm-Loader auf die Argumente zugreift, wird der Zeiger auf den Dateinamen geprüft. Der *Adressraum* eines Prozessimages

beträgt 32 KiByte (siehe Kapitel 1.1.1). Hier wird nun überprüft, ob der Zeiger auf eine Adresse innerhalb dieses Speicherbereiches zeigt.

36a ⟨*Aufrufargumente prüfen* 36a⟩≡ (35) 36b▷

```
char** uArgv = (char**)r->ecx;
if ( uArgv[0] < 0 || uArgv[0] > PROCESS_IMAGE_SIZE )
{
  // errno = EFAULT;
  r->eax = -1;
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("[EFAULT] filename points outside accessible address space.");
  #endif
  return;
}
```

Uses EFAULT 59c, ELF_TEST_MSG 59b, PROCESS_IMAGE_SIZE 59b, and TEST_ERROR_MSG 59b.

Nachdem überprüft wurde, dass Zeiger auf den Dateinamen in einen gültigen Speicherbereich zeigt, muss die Länge des Dateinamens getestet werden. Ein Dateiname darf im simplefs-Dateisystem maximal MAX_FILE_LENGTH Zeichen lang sein.

36b ⟨*Aufrufargumente prüfen* 36a⟩+≡ (35) ◁36a

```
if ( strlen(uArgv[0]) > MAX_FILE_LENGTH )
{
  // errno = ENAMETOOLONG;
  r->eax = -1;
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("[ENAMETOOLONG] filename is too long.");
  #endif
  return;
}
```

Uses ELF_TEST_MSG 59b, ENAMETOOLONG 59c, MAX_FILE_LENGTH 59b, and TEST_ERROR_MSG 59b.

Jetzt können die Aufruf-Argumente in den Kernel-Speicher gesichert werden.

36c ⟨*Aufrufargumente sichern* 36c⟩≡ (35) 37a▷

```
int kArgc = 0;

while ( uArgv[kArgc] != 0 )
  kArgc++;
```

Da nun die Anzahl der Argumente bekannt ist, kann Kernel-Speicher alloziert werden, um die Argumente temporär zu sichern. Die Aufruf-Argumente müssen gesichert werden, da nicht bekannt ist, wo sich die Zeichenketten der Argumente im *Prozessimage* des alten Programms befinden. Deshalb kann auch nicht sichergestellt werden, dass diese während der Ersetzung des Prozessimages nicht überschrieben werden, bevor sie an die korrekte Position auf dem neuen Stack kopiert wurden.

37a ⟨*Aufrufargumente sichern* 36c⟩+≡ (35) ◁36c

```
char** kArgv = kmalloc(kArgc* sizeof(char*));
if (kArgv == 0)
{

}
int i;
char* tmpPara;
size_t argLen;
for (i = 0; i<kArgc;i++)
{
  argLen = strlen(uArgv[i])+1;
  kArgv[i] =  kmalloc(argLen * sizeof(char));
  memcpy ( kArgv[i], uArgv[i], argLen );
}
```

Uses `size_t` 56b.

Nachdem die Aufruf-Argumente der *main*-Funktion gesichert wurden, kann mit dem Laden des neuen Programms begonnen werden. Der erste Schritt zum Laden eines Programms ist, die ELF-Datei zu öffnen. Um zu überprüfen, ob die ELF-Datei überhaupt im Dateisystem existiert, ist keine besondere Dateisystemfunktion nötig. Es wird stattdessen versucht, die gewünschte Datei zu öffnen, und daraufhin der *Returncode* der dafür verwendeten Funktion `simplefs_open` ausgewertet. Die Funktion `simplefs_open` gibt im Erfolgsfall einen Filedeskriptor zurück. Sollte ein Fehler aufgetreten sein, so wird von `simplefs_open` der Wert `-1` zurückgegeben.

In diesem Fall müsste der Error-Code `EACCES` gesetzt werden, welcher dem aufrufenden Programm anzeigt, dass die Datei nicht geöffnet werden konnte. Da dies aktuell noch nicht möglich ist, wird die betreffende Code-Zeile auskommentiert. Darüber hinaus muss der *Syscall-Handler* den Rückgabewert `-1` an das aufrufende Programm zurückgeben. Das aufrufende Programm kann so den Error-Code auswerten und entsprechend darauf reagieren.

37b ⟨*ELF-Datei in Speicher kopieren* 37b⟩≡ (35) 38b▷

```
char* filename = (char*)r->ebx;
int fd = simplefs_open ( filename );
if (fd<0)
{
  // errno = EACCES;
  r->eax = -1;
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("Cannot access file.");
  #endif
  ⟨Kernel-Speicher freigeben (Argumente) 38a⟩
  return;
}
```

Uses `ELF_TEST_MSG` 59b and `TEST_ERROR_MSG` 59b.

Da zu diesem Zeitpunkt bereits Kernel-Speicher für die Argumente reserviert wurde, muss dieser Speicher im Fehlerfall wieder freigegeben werden. Der folgende Code erledigt diese Aufgabe.

38a ⟨*Kernel-Speicher freigeben (Argumente)* 38a⟩≡ (37–40 46)

```
for (i=0;i<kArgc;i++)
  kfree(kArgv[i]);
kfree(kArgv);
```

Ulix-i386 stellt in der aktuellen Version ein einfaches Dateisystem zur Verfügung. Mit der Funktionsfamilie der `simplefs_*`-Funktionen können Dateien aus einem Floppyimage z.B. geöffnet, gelesen oder geschrieben werden. Allerdings implementiert dieses einfache Dateisystem noch keine Berechtigungen von Dateien. Deshalb kann an dieser Stelle des ELF-Programm-Loaders die Prüfung, ob das auszuführende ELF-Binary ausführbar ist, nicht final implementiert werden. Aus diesem Grund wird hier eine mögliche Implementierung im nachfolgenden *Code-Chunk* zur Verfügung gestellt. Der Code ist in einen C-Kommentar-Block gesetzt und kann angepasst werden, wenn die nötigen Dateisystemfunktionen verfügbar sind.

38b ⟨*ELF-Datei in Speicher kopieren* 37b⟩+≡ (35) ◁37b 38d▷
⟨*ELF-Datei Ausführrechte prüfen* 38c⟩

38c ⟨*ELF-Datei Ausführrechte prüfen* 38c⟩≡ (38b)

```
/*
struct stat buf;
int ret = fstat( fd, &buf );

if ( buf.st_mode & S_IXUSR )
  printf("Owner has execute permission.\n");
if ( buf.st_mode & S_IXGRP )
  printf("Group has execute permission.\n");
if ( buf.st_mode & S_IXOTH )
  printf("Others have execute permission.\n");

...
*/
```

Im nachfolgenden Codeabschnitt wird mittels der Funktion `simplefs_lseek` die Größe der ELF-Datei ermittelt. Zu diesem Zweck wird an das Dateiende gesprungen. Die Funktion `simplefs_lseek` gibt dabei die neue Schreib-/Leseposition zurück. Das entspricht der Größe der Datei in Bytes. Anschließend muss der Positionszeiger in der Datei wieder auf den Dateianfang gesetzt werden, damit die Datei von Beginn an gelesen werden kann.

38d ⟨*ELF-Datei in Speicher kopieren* 37b⟩+≡ (35) ◁38b 39▷

```
int length = simplefs_lseek(fd, 0, SEEK_END);

if ( simplefs_lseek(fd, 0, SEEK_SET) == -1 || length == -1 )
```

```
{
  // errno = EIO;
  r->eax = -1;
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("An I/O error occurred.");
  #endif
  ⟨Kernel-Speicher freigeben (Argumente) 38a⟩
  return;
}
```

Uses EIO 59c, ELF_TEST_MSG 59b, SEEK_END 56a, SEEK_SET 56a, and TEST_ERROR_MSG 59b.

Konnte die ELF-Datei ohne Fehler geöffnet werden, so kann das *Prozessimage* des neuen Programms erstellt werden. Dazu müssen die Text- und *Daten-Segmente* des neuen Programms in das *Prozessimage* des aktuellen Prozesses kopiert werden. Dabei wird das *Prozessimage* des alten Programms überschrieben und somit ungültig. Außerdem muss der Usermode-Stack, welcher am Ende des Prozessimages steht, für die Ausführung des neuen Programms initialisiert werden.

39 ⟨*ELF-Datei in Speicher kopieren* 37b⟩+≡ (35) ◁38d 40a▷

```
byte* buffer = (byte*) kmalloc(length);
if (buffer == 0)
{
  // errno = ENOMEM
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("Insufficient kernel memory available.");
  #endif
  r->eax = -1;
  ⟨Kernel-Speicher freigeben (Argumente) 38a⟩
  return;
}
memset( buffer, 0, length);

if ( simplefs_read(fd, buffer, length) != length )
{
  // errno = EIO;
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("An I/O error occurred.");
  #endif
  r->eax = -1;
  kfree(buffer);
  ⟨Kernel-Speicher freigeben (Argumente) 38a⟩
  return;
}
```

Uses byte 56b, EIO 59c, ELF_TEST_MSG 59b, ENOMEM 59c, and TEST_ERROR_MSG 59b.

Der vorhergehende Codeabschnitt sorgt dafür, dass temporär Kernel-Speicher alloziert wird, um die gesamte ELF-Datei einzulesen. Diese Methode ist in Hinblick auf geringen Verbrauch von zusätzlichem Speicher und aus Performancesicht nicht besonders effizient. Allerdings vereinfacht diese Vorgehensweise den Code zum Laden der ELF-Datei, da auf diese

Weise nur einmal Speicher alloziert und wieder freigegeben werden muss. Somit trägt diese Vorgehensweise zur Übersichtlichkeit des Programmcodes bei. Geringe Effizienz und Performance werden dabei zugunsten der Einfachheit in Kauf genommen.

Da sich nun die komplette ELF-Datei im Speicher des Kernels befindet, wird die Datei nicht mehr benötigt und kann somit geschlossen werden.

40a ⟨*ELF-Datei in Speicher kopieren* 37b⟩+≡ (35) ◁39

```
if ( simplefs_close(fd) == -1 )
{
  // errno = EACCESS;
  #ifdef ELF_TEST_MSG
  TEST_ERROR_MSG("Cannot access file.");
  #endif
  r->eax = -1;
  kfree(buffer);
  ⟨Kernel-Speicher freigeben (Argumente) 38a⟩
  return;
}

Elf32_Ehdr* pElfHeader  = (Elf32_Ehdr*) buffer;
```

Uses EACCESS 59c, Elf32_Ehdr 16, ELF_TEST_MSG 59b, and TEST_ERROR_MSG 59b.

Zu diesem Zeitpunkt muss auch sichergestellt werden, dass dynamisch allozierter Speicher wieder freigegeben wird, um *Memory-Leaks* zu vermeiden. Dynamischer Speicher könnte von dem alten Programm beispielsweise mit einem `malloc`-Aufruf alloziert worden sein. Die *ulixlib*-Bibliothek stellt allerdings im aktuellen Stand noch keine Möglichkeit zur Verfügung, um Speicher zu allozieren. Aus diesem Grund kann zu diesem Zeitpunkt auch noch keine Implementierung zur Freigabe dieses Speichers zur Verfügung gestellt werden. Es wird lediglich ein Platzhalter eingefügt, der eine zukünftige Implementierung zur Freigabe des dynamisch allozierten Speichers aufnehmen kann. Die Freigabe des Speichers darf erst ausgeführt werden, nachdem sichergestellt wurde, dass die ELF-Datei korrekt geladen werden kann und es sich um eine gültige ELF-Datei handelt.

40b ⟨*Dynamisch allozierten Speicher freigeben* 40b⟩≡ (35)

```
/* Hier muss dynamisch allozierter Usermode Speicher freigegeben werden. */

/* NOT IMPLEMENTED */
```

Als nächstes werden Zeiger auf die *Programm-Header* benötigt. In den Programm-Headern stehen die Offsets zu den jeweiligen Code-Segmenten und die virtuellen Adressen, an welche das jeweilige Code-Segment kopiert werden soll. Die *Programm-Header* haben eine feste Größe, welche im ELF-Header in der Variable `e_phentsize` eingetragen ist. Mithilfe eines Zeigers auf den ersten *Programm-Header* können durch Zeigerarithmetik auch alle folgenden *Programm-Header* angesprochen werden.

41 ⟨*Prozessimage ersetzen* 41⟩≡ (35) 42▷

```
// Zeiger auf den ersten Programm-Header
Elf32_Phdr* pFirstProgramHeader = (Elf32_Phdr*) (
    ((byte*)pElfHeader)
  + pElfHeader->e_phoff
);
```

Uses `byte` 56b and `Elf32_Phdr` 18.

In Listing 3.1 sind die *Programm-Header* des Beispielprogramms dargestellt. Die Ausgabe wurde mit dem Linux-Tool `readelf` erzeugt. An dem Typ *LOAD* kann man ablesen, welche Programmteile in den Speicher geladen werden sollen. Außerdem gibt die Ausgabe von `readelf` Auskunft darüber, was mit den jeweiligen Segmenten passieren soll. Die Felder eines Programm-Headers sind:

- **Offset:** Gibt den Offset des Segments innerhalb der ELF-Datei an.
- **VirtAdr:** Enthält die virtuelle Adresse, an die das Segment geladen werden soll.
- **PhysAdr:** Enthält die physikalische Adresse, an die das Segment geladen werden soll. Dieses Feld wird in Ulix-i386 nicht verwendet, das Ulix-i386 mit virtueller Speicherverwaltung arbeitet.
- **DateiGr:** Gibt die Größe des Segments in der ELF-Datei an.
- **SpeiGr:** Gibt die Größe des Segments im Speicher an.
- **Flg:** Enthält die Zugriffs-Flags, die für das Segment im Speicher gesetzt werden sollen (**R**ead, **W**rite, **E**xecute). Zugriffsflags können in der aktuellen Version von Ulix-i386 noch nicht gesetzt werden.
- **Ausr:** Gibt das Alignment im Arbeitsspeicher vor.

```
Programm-Header:
  Typ            Offset   VirtAdr    PhysAdr    DateiGr SpeiGr  Flg Ausr
  LOAD           0x001000 0x00000000 0x00000000 0x00968 0x00968 R E 0x1000
  LOAD           0x001968 0x00001968 0x00001968 0x00004 0x00004 RW  0x1000
  LOAD           0x0020d4 0x080480d4 0x080480d4 0x00024 0x00024 R E 0x1000
  NOTE           0x0020d4 0x080480d4 0x080480d4 0x00024 0x00024 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

Listings 3.1: Program-Header (elfbin)

Die *Programm-Header* stehen direkt im Anschluss an den ELF-Header der ELF-Datei. Dies gilt allerdings nur in der Ausführungsvariante. In der Linkervariante sind die *Programm-Header* optional. Deshalb enthält der ELF-Header auch den Offset zu den Programm-Headern in der Variablen `e_phoff`.

Für das Beispielprogramm `elfbin` kann man den Offset zu den Programm-Headern aus der Ausgabe von `readelf` der Zeile „Beginn der Programm-Header“ entnehmen (siehe Listing 1.3). Im nachfolgenden Codeabschnitt wird über alle *Programm-Header* iteriert und der Programmcode in das *Prozessimage* kopiert. Dafür wird zunächst ein Zeiger auf den *Programm-Header* der aktuellen Iteration berechnet. Dieser Zeiger wird in der Variablen `pProgramHeader` gespeichert.

42 ⟨*Prozessimage ersetzen* 41⟩+≡ (35) ◁41 43a▷

```
unsigned int idx;
Elf32_Phdr* pProgramHeader = pFirstProgramHeader;
uint* nextHeapAddress;
for( idx=0; idx<pElfHeader->e_phnum; idx++)
{
  /* Zeiger auf den aktuellen Programm-Header ermitteln */
  Elf32_Phdr* pProgramHeader = (Elf32_Phdr*) (
      ((byte*)pFirstProgramHeader) + idx * pElfHeader->e_phentsize
  );

  if ( pProgramHeader->p_type != PT_LOAD )
  {
    continue;
  }
```

Uses `byte` 56b, `Elf32_Phdr` 18, and `uint` 56b 56b.

Die Prüfung des Programm-Header-Typs (`p_type`) auf den Wert `PT_LOAD` sorgt dafür, dass nur Programm-Segmente in das *Prozessimage* geladen werden, die auch dafür vorgesehen sind. Ein Blick auf die Programm-Header-Tabelle in Listing 3.1 zeigt, dass es auch andere Typen von Programm-Headern gibt. Im Fall des Beispielprogramms ist das der Programm-Header-Typ *NOTE*. Er ist für die Ausführung des Programms nicht von Bedeutung. Der Inhalt eines *NOTE*-Segments ist nicht spezifiziert. Diese *Segmente* können verwendet werden, um ELF-Dateien zu markieren. Auf diese Weise können andere Programme die ELF-Dateien identifizieren und eventuelle Kompatibilitäts-Prüfungen etc. durchführen. Lauf ELF-Spezifikation darf der Inhalt solcher *Segmente* die Ausführung des Programms nicht beeinflussen.

> *Note information is optional. The presence of note information does not affect a program's TIS conformance, provided the information does not affect the program's execution behavior. Otherwise, the program does not conform to the TIS ELF specification and has undefined behavior.* [Com95]

Aus diesem Grund werden diese *Segmente* auch nicht in das *Prozessimage* kopiert.

Im folgenden Codeabschnitt wird ein Workaround verwendet, der verhindert, dass die genannten *Daten-Segmente* in das *Prozessimage* kopiert werden. Im konkreten Fall des Beispielprogramms handelt es sich um das dritte Programm-Segment vom Typ *LOAD* (siehe List-

ing 3.1). Der Grund warum dieses Segment nicht geladen werden kann ist die virtuelle Zieladresse, an die das Programm-Segment im *Prozessimage* geladen werden soll. Die virtuelle Zieladresse ist in diesem Fall größer als die größte Adresse des Prozessimages. Diese Zieladressen kommen zustande, da die Erzeugung der Programme - wie in Kapitel 1.4 beschrieben - noch nicht optimal an ULIX-i386 angepasst wurde. Bei genauerer Betrachtung fällt jedoch auf, dass es sich bei diesem Programm-Segment um den *NOTE* Bereich handelt. Dieser ist für die Ausführung nicht relevant und kann deshalb ignoriert werden.

43a ⟨*Prozessimage ersetzen* 41⟩+≡ (35) ◁42 43b▷

```
// WORKAROUND:
//
//       ELF enthält einen Bereich, der an eine
//       Adresse kopiert werden soll, die größer
//       als die größte Adresse im Prozessimage ist.
//
//       -> WENN
//                (Virtuelle Adresse + Segmentgröße) > (PROCESS_IMAGE_SIZE)
//          DANN
//                wird der Programmblock nicht kopiert
//
#ifdef DO_LOAD_WORKAROUND
if ( (pProgramHeader->p_vaddr + pProgramHeader->p_memsz) > PROCESS_IMAGE_SIZE )
{
  #ifdef ELF_DEBUG_MSG
  printf("[%s] Ignoring program block (%u)\n", _MODULENAME_, idx);
  #endif
  continue;
}
#endif
```

Uses _MODULENAME_ 59b, DO_LOAD_WORKAROUND 59b, and PROCESS_IMAGE_SIZE 59b.

Nachdem der aktuelle *Programm-Header* ermittelt wurde, kann jetzt das Codesegment, welches zu dem *Programm-Header* gehört, in das *Prozessimage* kopiert werden. Dafür wird als erstes ein Zeiger auf das zugehörige Code-Segment berechnet.

43b ⟨*Prozessimage ersetzen* 41⟩+≡ (35) ◁43a 44a▷

```
// Zeiger auf das aktuelle Code Segment ermitteln
byte* pCodeSegment = (byte*) (
    ((byte*)pElfHeader) + pProgramHeader->p_offset
);
```

Uses byte 56b.

Die Zeigervariable `pCodeSegment` enthält nun eine Referenz auf das Code-Segment, welches in das *Prozessimage* kopiert werden soll. Die virtuelle Zieladresse, an die das Code-Segment kopiert werden soll, wird dabei in der Membervariablen `p_vaddr` des Programm-Headers bereitgestellt. Der *Programm-Header* stellt ebenfalls die Größe des jeweiligen Code-Segments

zur Verfügung. Sie ist in der Membervariablen `p_memsz` gespeichert. Nun kann mit einem `memcpy`-Aufruf das gewünschte Code-Segment in das *Prozessimage* kopiert werden.

44a ⟨*Prozessimage ersetzen* 41⟩+≡ (35) ◁43b 44b▷

```
    // Zeiger auf die virtuelle Adresse an der das Codesegment
    // im Prozessimage geladen werden soll
    byte* dest = (byte*)pProgramHeader->p_vaddr;

    // Programmcode in das Prozessimage kopieren
    memcpy( dest, pCodeSegment, pProgramHeader->p_memsz );
    nextHeapAddress = (uint*) dest + pProgramHeader->p_memsz;
  }
```

Uses `byte` 56b and `uint` 56b 56b.

Nachdem alle *Programm-Header* durchlaufen und die damit verknüpften *Daten-Segmente* in das *Prozessimage* kopiert wurden, wird der `buffer`, der für die temporäre Kopie der ELF-Datei angelegt wurde, nicht mehr benötigt. Der betroffene Speicherbereich kann demnach wieder freigegeben werden. Vorher muss allerdings noch der Einsprungpunkt, welcher der Adresse der *main*-Funktion des Programms entspricht (siehe Kapitel 1.4), gesichert werden. Diese Adresse wird später für den eigentlichen Aufruf des Programms benötigt. Die Variable `pElfHeader->e_entry` enthält den Einsprungpunkt-Adresse in das neue Programm. Diese Adresse wird in die Variable `r->eip` geschrieben. Dies sorgt dafür, dass nach dem Verlassen des Syscall-Handlers die Ausführung an dieser Adresse fortgesetzt und somit das Programm gestartet wird.

44b ⟨*Prozessimage ersetzen* 41⟩+≡ (35) ◁44a

```
  r->eip = pElfHeader->e_entry;
  kfree(buffer);
```

Der nächste Schritt, ist die zuvor gesicherten Aufruf-Argumente des Programms in den Heap-Speicher des neuen Prozessimages zu kopieren. Im Normalfall müsste zu diesem Zweck Speicher alloziert werden. Da dies allerdings – wie in Kapitel 1.1.1 beschrieben – nicht möglich ist, wurde während des Kopierens der Programm-Segmente die nächste freie Heap-Adresse in der Variablen `nextHeapAddress` gespeichert. An dieser Adresse wird Platz für ein Zeigerarray auf die Zeichenketten der Argumente freigehalten. In den danach angrenzenden Speicher werden die eigentlichen Zeichenketten der Argumente kopiert. Während des Kopierens werden die Adressen auf die Zeichenketten in das Zeigerarray geschrieben. Der resultierende Aufbau des Prozessimages wird in Abbildung 3.1 dargestellt.

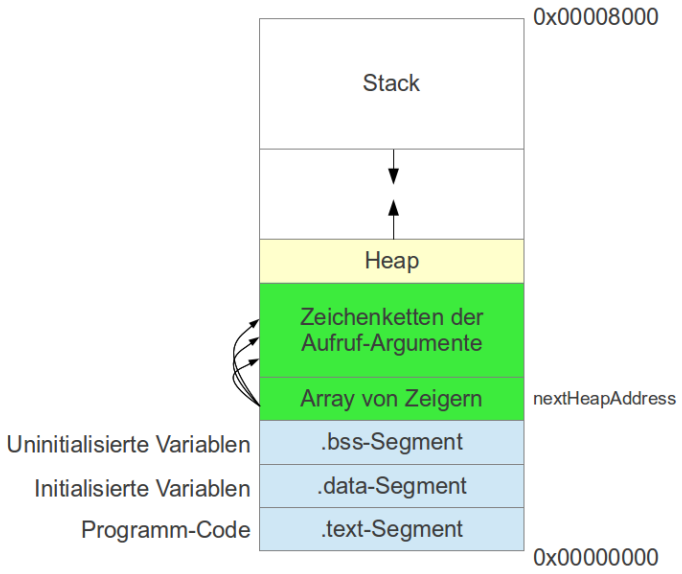

Abbildung 3.1.: Prozessimage mit Aufruf-Argumenten

45a ⟨*Aufrufargumente in Heap-Speicher kopieren* 45a⟩≡ (35)

```
uint* adest = nextHeapAddress;
uint* adestrun = adest;
adestrun += kArgc;

for (i=0;i<kArgc;i++)
{
  argLen = strlen(kArgv[i])+1;
  memcpy (adestrun, kArgv[i], argLen);
  *(adest+i) = adestrun;
  adestrun+=argLen;
}
```

Uses `uint` 56b 56b.

Nachfolgend wird der Usermode-Stack des aufrufenden Programms mit Nullen überschrieben, damit das nachfolgend ausgeführte Programm keine Daten des aufrufenden Programms mehr auslesen kann.

45b ⟨*Usermode-Stack initialisieren* 45b⟩≡ (35) 46▷

```
memset ( (uint*)r->useresp, 0, PROCESS_IMAGE_SIZE - r->useresp);
```

Uses `PROCESS_IMAGE_SIZE` 59b and `uint` 56b 56b.

Nachdem der alte Usermode-Stack überschrieben wurde, können die Aufruf-Argumente für das neue Programm auf den Stack gelegt werden. Der Stack hat danach die in Abbildung 3.2 dargestellte Form.

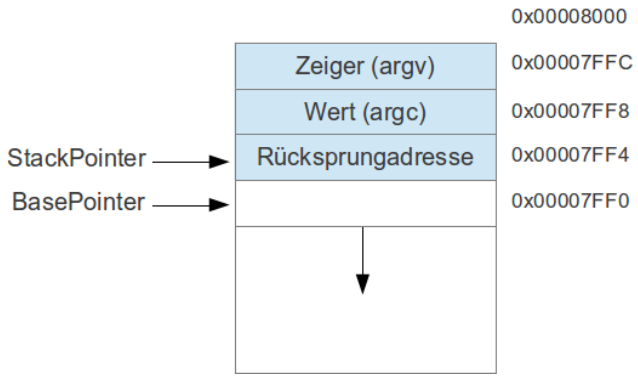

Abbildung 3.2.: Initialer Stack

46 ⟨*Usermode-Stack initialisieren* 45b⟩+≡ (35) ◁45b

```
uint* stk = PROCESS_IMAGE_SIZE;
*(--stk) = adest;
*(--stk) = kArgc;

⟨Kernel-Speicher freigeben (Argumente) 38a⟩

*(--stk) = 0x00000000;  // dummy address value
r->useresp = stk;
r->ebp = stk-1;
```

Uses `PROCESS_IMAGE_SIZE` 59b and `uint` 56b 56b.

Hinweis:

Am Ende des Code-Chunks zur Initialisierung des Usermode-Stack wurde der Wert `0x00000000` auf den Stack gelegt. Dies hat folgenden Grund: Die *main*-Funktion erwartet an dieser Stelle die Rücksprungadresse der Funktion, aus der sie aufgerufen wurde. Eine solche Funktion existiert in dieser Implementierung des ELF-Programm-Loaders nicht. Stattdessen wird – wie schon erwähnt – nach dem Verlassen des Syscall-Handlers direkt mit der Ausführung der *main*-Funktion fortgefahren. Diese Einschränkung führt auch dazu, dass die *main*-Funktion nicht mit *return* verlassen werden kann. Dies würde dazu führen, dass die Ausführung an der Adresse `0x00000000` fortgesetzt würde, was einen Page-Fault zur Folge hat. Aus diesem Grund müssen Programme, die mit dem ULIX-i386-ELF-Programm-Loader geladen werden, immer mit einem Aufruf der *exit*-Funktion aus der ULIX-i386-Standardbibliothek beendet werden.

An dieser Stelle ist das *Prozessimage* des neuen Programms für dessen Ausführung vorbereitet. Es wurden Code- und Datensegmente ausgetauscht und der Stack für das neue Programm initialisiert.

4. Tests

In diesem Kapitel wird die Vorgehensweise, um die Funktionalität des Ulix-i386-Programm-Loaders zu testen, erläutert.

4.1. Testkonzept

Um die korrkete Funktionalität des Ulix-i386-Programm-Loaders zu testen, werden Test-Cases definiert, die das Verhalten des Programm-Loaders in verschiedenen Fehler- und Erfolgssituationen verifizieren. Diese Test-Cases werden von dem Programm `UlixTestRun` nacheinander aufgerufen und die Testergebnisse auf der Konsole ausgegeben. Die Durchführung der Tests findet innerhalb der Laufzeitumgebung von Ulix-i386 statt. Das Test-Programm wird selbst auch vom Ulix-i386-Programm-Loader gestartet und setzt so die korrekte Funktionalität des Programm-Loaders vorraus. Die erfolgreiche Durchführung eines Testlaufs kann somit als weiterer impliziter Test des Gesamtsystems betrachtet werden.

4.2. Test-Programm (`UlixTestRun`)

Es folgt nun die Implementierung der *main*-Funktion für das Test-Programmes `UlixTestRun`.

47a ⟨*UlixTestRun.c* 47a⟩≡
```
⟨EXEC ERROR Definitionen 59c⟩
⟨RunTest Funktion 48a⟩
⟨UlixTestRun Main 47b⟩
```

47b ⟨*UlixTestRun Main* 47b⟩≡ (47a)
```
int main(int argc, char* argv[])
{
  /* Erfolgreiche Ausführung erwartet */
  ⟨TC01 elfbin drei Argumente success 48b⟩
  ⟨TC02 elfbin ein Argument success 49a⟩

  /* Fehler beim Laden erwartet */
  ⟨TC03 EFAULT 49b⟩
  ⟨TC04 EACCES 50a⟩
  ⟨TC05 EIO 50b⟩
  ⟨TC06 ENAMETOOLONG 51a⟩
  ⟨TC07 ENOEXEC 51b⟩
```

```
    ⟨TC08 ENOMEM 52a⟩
    ⟨TC09 ETXTBSY 52b⟩

    exit(0);
  }
```

Uses main 19c.

Die Funktion RunTest führt den eigentlichen Aufruf von execvp durch. RunTest wird dabei von allen Test-Cases mindestens einmal aufgerufen. Bei jedem Aufruf von RunTest – und damit auch von execvp – wird versucht ein ELF-Programm zu starten.

48a ⟨RunTest Funktion 48a⟩≡ (47a)

```
int RunTest(char* argv[])
{
  int pid = fork ();
  int status=-1;
  int i;

  if (pid == 0) {
    // Kind
    int ret_code = execvp(argv[0], argv);
    if ( ret_code == -1 )
    {
      printf("[ERROR] execvp\n");
    }
    exit(-1);
  } else {
    // Vater
    while (waitpid (pid, &status, 0) == 0); // Das setzen des Status funktioniert
                                            // in der aktuellen Ulix-Version nicht.
    return 0;
  }
  return 0;
}
```

4.3. Definition der Test-Cases

In diesem Kapitel werden die Test-Cases, die zur Verifizierung des ELF-Programm-Loaders dienen, definiert. Jeder Test-Case ruft dabei die RunTest-Funktion auf, um ein ELF-Binary zu starten. Da die Test-Cases immer wieder die selben Variablennamen verwenden, wird jeder Test-Case in einem eigenen Programm-Block definiert, wodurch sich die Test-Cases in voneinander unabhängigen Scopes befinden. Die beiden Test-Cases TC01 und TC02 testen das erfolgreiche Laden eines ELF-Programms.

48b ⟨TC01 elfbin drei Argumente success 48b⟩≡ (47b)

```
  {
    int TNR = 1;
```

```
  printf("**********************\nRunning TC%d:\n**********************\n",TNR);
  char* binary = "elfbin";
  char* args[4] = {0};
  args[0] = binary;
  args[1] = "hallo";
  args[2] = "welt";
  RunTest(args);
  printf("Test %d successful? [ ]\n**********************\n\n",TNR);
}
```

49a ⟨*TC02 elfbin ein Argument success* 49a⟩≡ (47b)

```
{
  int TNR = 2;
  printf("**********************\nRunning TC%d:\n**********************\n",TNR);
  char* binary = "elfbin";
  char* args[2] = {0};
  args[0] = binary;
  RunTest(args);
  printf("Test %d successful? [ ]\n**********************\n\n",TNR);
}
```

Die folgenden Error-Codes stammen aus der `execve`-man-page: [manb]

Nachfolgend werden für alle hier aufgelisteten Error-Codes - die von dem ELF-Programm-Loader gesetzt werden können - Test-Cases definiert. Nicht alle Test-Cases sind zum gegenwärtigen Zeitpunkt aktiviert, da für ihre Ausführung noch ULIX-i386-Funktionalitäten fehlen.

1. **EFAULT** Zeiger auf Dateiname referenziert einen nicht zugreifbaren Speicherbereich
2. **EACCES** Dateiname ist keine gültige Datei
3. **EIO** Ein-/Ausgabefehler
4. **ENAMETOOLONG** Dateiname ist zu lang (Dateisystemabhängig)
5. **ENOEXEC** Dateiname ist keine gültige ELF-Datei
6. **ENOMEM** Nicht ausreichend Kernel-Speicher allozierbar
7. **ETXTBSY** ELF-Datei ist von einem anderen Prozess (schreibend) geöffnet

TC03: EFAULT In diesem Test-Case wird das Verhalten des Programm-Loaders getestet, wenn der Zeiger auf den Dateinamen auf eine Adresse außerhalb des zugreifbaren Adressraums zeigt. Die Adresse `0x11000400` ist dabei willkürlich gewählt worden.

49b ⟨*TC03 EFAULT* 49b⟩≡ (47b)

```
{
  int TNR = 3;
  printf("**********************\nRunning TC%d:\n**********************\n",TNR);
  char* binary;
```

```
    binary = 0x11000400; // Willkürlicher Wert
    char* args[2] = {0};
    args[0] = binary;
    RunTest(args);
    printf("Test %d successful? [ ]\n**********\n\n",TNR);
  }
```

TC04: EACCESS In diesem Test-Case wird getestet, ob die auszuführende Datei im Dateisystem existiert. Um den Test durchzuführen, wird der Name einer Datei übergeben, die nicht existiert.

50a ⟨*TC04 EACCES* 50a⟩≡ (47b)

```
  {
    int TNR = 4;
    printf("**********\nRunning TC%d:\n**********\n",TNR);
    char* binary = "notexistent";
    char* args[2] = {0};
    args[0] = binary;
    RunTest(args);
    printf("Test %d successful? [ ]\n**********\n\n",TNR);
  }
```

TC05: EIO In diesem Test-Case wird getestet, ob auf die auszuführende Datei zugegriffen werden kann. Dieser Fehler kann in der aktuellen Version von Ulix-i386 nicht provoziert werden. Damit dieser Fehler auftritt, wird ein Dateisystem benötigt, welches Zugriffsrechte von Dateien implementiert. Damit könnte versucht werden, eine Datei auszuführen, für die keine Leserechte vorhanden sind. Da `simplefs` derzeit keine Dateisystemberechtigungen unterstützt, wurde dieser Test-Case deaktiviert.

50b ⟨*TC05 EIO* 50b⟩≡ (47b)

```
  /*{
    int TNR = 5;
    printf("**********\nRunning TC%d:\n**********\n",TNR);
    char* binary = "notreadable";
    char* args[2] = {0};
    args[0] = binary;
    RunTest(args);
    printf("Test %d successful? [ ]\n**********\n\n",TNR);
  }*/
```

TC06: ENAMETOOLONG Das simplefs-Dateisystem erlaubt eine maximale Länge von `MAX_FILE_LENGTH` für Dateinamen. Wird der Programm-Loader mit einem Dateinamen aufgerufen, der größer ist, so muss das Laden des Programms mit dem Error-Code `ENAMETOOLONG` abgebrochen werden.

51a ⟨*TC06 ENAMETOOLONG* 51a⟩≡ (47b)

```
{
  int TNR = 6;
  printf("**********************\nRunning TC%d:\n**********************\n",TNR);
  char* binary = "tooLooooooong"; // Dieser Dateiname ist 12 Zeichen lang.
  char* args[2] = {0};           // Maximal erlaubt sind 11 Zeichen.
  args[0] = binary;
  RunTest(args);
printf("Test %d successful? [ ]\n**********************\n\n",TNR);
}
```

TC07: ENOEXEC Der nächste Test-Case überprüft, ob der Programm-Loader ein ungültige ELF-Datei erkennen kann. Eine ungültige ELF-Datei liegt vor, wenn mindestens eines der ELF-Identifikations-Felder im ELF-Header nicht korrekt ist. Welche Felder dies genau sind, wurde in Kapitel 1.3.2 erläutert. Um diesen Test durchzuführen, wird die gültige ELF-Datei `elfbin` kopiert und unter einem anderen Dateinamen abgelegt. Danach werden in den einzelnen Kopien von `elfbin` die entsprechenden Identifikations-Felder manipuliert. Der Test-Case muss für alle Aufrufe dieser `elfbin`-Kopien den Error-Code `ENOEXEC` ausgeben.

51b ⟨*TC07 ENOEXEC* 51b⟩≡ (47b)

```
{
  int TNR = 7;
  printf("**********************\nRunning TC%d:\n**********************\n",TNR);
  char* binary = "elfbin0";
  binary = "elfbin1";
  char* args[2] = {0};
  args[0] = binary;
  RunTest(args);

  binary = "elfbin2";
  args[0] = binary;
  RunTest(args);

  binary = "elfbin3";
  args[0] = binary;
  RunTest(args);

  binary = "elfbin4";
  args[0] = binary;
  RunTest(args);

  binary = "elfbin5";
  args[0] = binary;
  RunTest(args);

  binary = "elfbin6";
  args[0] = binary;
  RunTest(args);
  printf("Test %d successful? [ ]\n**********************\n\n",TNR);
}
```

TC08: ENOMEM Dieser Test-Case soll einen Out-of-memory-Fehler erzeugen. Der Test-Case ist aktuell deaktiviert, da in der momentanen Ulix-i386-Version dieser Fehler nicht provoziert werden kann.

52a ⟨*TC08 ENOMEM* 52a⟩≡ (47b)

```
/*{
  int TNR = 8;
  printf("**********************\nRunning TC%d:\n**********************\n",TNR);
  char* binary = "elfbinBIG";
  char* args[2] = {0};
  args[0] = binary;
  RunTest(args);
  printf("Test %d successful? [ ]\n**********************\n\n",TNR);
}*/
```

TC09: ETXTBSY In diesem Test-Case wird die Datei `elfbin` von `UlixTestRun` geöffnet. Anschließend ruft `UlixTestRun` - wie in allen anderen Test-Cases - die Funktion `RunTest` auf. In diesem Fall muss der ELF-Programm-Loader einen Fehler ausgeben, da ein auszuführendes Programm nicht gleichzeitig von einem anderen Prozess geöffnet sein darf. Da `simplefs` es noch nicht erlaubt eine Datei explizit zum Schreiben zu öffnen wurde dieser Test-Case deaktiviert.

52b ⟨*TC09 ETXTBSY* 52b⟩≡ (47b)

```
/*{
  int TNR = 9;
  printf("**********************\nRunning TC%d:\n**********************\n",TNR);
  char* binary = "elfbin";
  char* args[2] = {0};
  args[0] = binary;

  int fd = open( binary );  //
  RunTest(args);
  close(fd);
  printf("Test %d successful? [ ]\n**********************\n\n",TNR);
}*/
```

4.4. Testdurchführung

Zunächst muss Ulix-i386 mithilfe von `qemu` gestartet werden. Dem Benutzer steht nach dem Start des Systems die Ulix-i386-Shell zur Verfügung. Durch Eingabe des Kommandos `UlixTestRun` werden alle Tests des ELF-Programm-Loaders durchgeführt. `UlixTestRun` gibt dabei die Ergebnisse jedes einzelnen Test-Cases auf der Standardausgabe aus. Nachdem alle Tests abgeschlossen sind müssen die Ausgaben der einzelnen Test manuell in das Testprotokoll kopiert werden.

4.5. Testergebnisse

Dieses Testprotokoll enthält die Ausgaben des Test-Programmes nach einem vollständigen Testlauf. Die Ausgaben der jeweligen Tests wurden händisch überprüft und für korrekt befunden. Die Zeilen „Test $ successful? [ ]“ wurden nach der manuellen Überprüfung der Ausgaben ausgefüllt.

```
**********************
Running TC1:
**********************
### Teststring Ausgabe
Wert von x:   106 0x6a
Wert von y: 44015 0xabef
Anzahl der Argumente von elfbin (argc): 3
Argumente von elfbin (argv):
        argv[0]: elfbin
        argv[1]: hallo
        argv[2]: welt
Test 1 successful? [X]
**********************

**********************
Running TC2:
**********************
### Teststring Ausgabe
Wert von x:   106 0x6a
Wert von y: 44015 0xabef
Anzahl der Argumente von elfbin (argc): 1
Argumente von elfbin (argv):
        argv[0]: elfbin
Test 2 successful? [X]
**********************

**********************
Running TC3:
**********************
[LOADER-TEST] ERROR: [EFAULT] filename points outside accessible address space.
[ERROR] execvp
Test 3 successful? [X]
**********************

**********************
Running TC4:
**********************
[LOADER-TEST] ERROR: Cannot access file.
[ERROR] execvp
Test 4 successful? [X]
**********************

**********************
Running TC6:
**********************
[LOADER-TEST] ERROR: [ENAMETOOLONG] filename is too long.
[ERROR] execvp
Test 6 successful? [X]
**********************

**********************
Running TC7:
**********************
[LOADER-TEST] ERROR: Invalid file format.
```

```
[ERROR] execvp
[LOADER-TEST] ERROR: Invalid file format.
[ERROR] execvp
[LOADER-TEST] ERROR: Invalid file format.
[ERROR] execvp
[LOADER-TEST] ERROR: Invalid file format.
[ERROR] execvp
[LOADER-TEST] ERROR: Invalid file format.
[ERROR] execvp
[LOADER-TEST] ERROR: Invalid file format.
[ERROR] execvp
Test 7 successful? [X]
**********************
```

Listings 4.1: Test Protokoll

A. Definitionen

Dieses Kapitel enthält Definitionen, die vom Ulix-i386-Programm-Loader benötigt werden. Die Funktion dieser Definition wird an den Stelle erläutert, an denen sie verwendet werden.

A.1. Ulix-i386 Definitionen

Die nachfolgenden Definitionen stammen aus Ulix-i386 und müssen in diesem Dokument als `extern` deklariert werden, damit der Sourcecode des ELF-Programm-Loaders ohne den Ulix-Sourcecode übersetzt werden kann.

55 ⟨*Externe Deklarationen (ULIX)* 55⟩≡ (31b)

```
extern size_t strlen(const char *str);
extern int current_as;
extern int ulix_fork (struct regs_syscall *r);
extern void syscall_waitpid (struct regs_syscall *r);
extern void syscall_exit (struct regs_syscall *r);
extern void insert_syscall (int syscallno, void* syscall_handler);
extern int simplefs_open (char* filename);
extern int simplefs_close (int fd);
extern int simplefs_read (int fd, char* buf, int nbytes);
extern int simplefs_write (int fd, char* buf, int nbytes);
extern int simplefs_lseek(int fd, int offset, int whence);
extern int printf(const char *format, ...);

extern volatile int current_task;
extern char testprocess2[];
extern char testprocess[];
extern tss_entry;
extern int register_new_tcb (int as_id);
extern void start_program_from_disk(int secno);
extern void print_process_list();
extern void activate_address_space(int id);
extern void cpu_usermode(unsigned int address, unsigned int stack);
extern void print_page_table();
extern int kernel_read_sector (int secno, char* buf);
extern int create_new_address_space(int initial_ram);
extern uint* request_new_page (int need_more_pages);
extern void *memcpy(void *dest, const void *src, size_t count);
extern void switch_to_user_mode();
extern void putch(unsigned char c);
extern void* kmalloc (uint size);
extern void kfree (void* pointer);
extern void printhex (uint i);
```

```
extern void *memset(void *dest, char val, size_t count);
```

Uses size_t 56b and uint 56b 56b.

56a ⟨*Externe Definitionen (ULIX)* 56a⟩≡ (31b)

```
#define boolean unsigned int
#define true 1
#define false 0
#define null 0
#define PAGE_SIZE 4096
#define KERNEL_STACK_SIZE PAGE_SIZE
#define MAX_THREADS 1024
#define GPR_REGISTERS 6
#define EAX 0
#define EBX 1
#define ECX 2
#define EDX 3
#define ESI 4
#define EDI 5
#define asm __asm__
#define SEEK_SET        0
#define SEEK_CUR        1
#define SEEK_END        2
#define CMDLINE_LENGTH 50
```

Defines:
boolean, used in chunk 56b.
CMDLINE_LENGTH, used in chunk 56b.
EAX, never used.
EBX, never used.
ECX, never used.
EDI, never used.
EDX, never used.
ESI, never used.
false, never used.
GPR_REGISTERS, used in chunk 56b.
KERNEL_STACK_SIZE, never used.
MAX_THREADS, used in chunk 56b.
null, never used.
PAGE_SIZE, never used.
SEEK_CUR, never used.
SEEK_END, used in chunk 38d.
SEEK_SET, used in chunk 38d.
true, never used.

56b ⟨*Externe Typ-Definitionen (ULIX)* 56b⟩≡ (31b)

```
typedef unsigned char byte;
typedef int size_t;
typedef unsigned int uint;
typedef unsigned short u16int;
typedef unsigned int thread_id;
typedef uint addr_space_id;

⟨Struktur für Zugriff auf Kernel-Stack 34⟩
struct regs_syscall {
  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;
};

typedef struct {
  thread_id tid;
  thread_id ppid;   // parent process
  unsigned int context[GPR_REGISTERS]; // general purpose registers
  unsigned int esp; // user stack pointer
  unsigned int esp0; // kernel stack pointer
  unsigned int eip; // program counter
  unsigned int ebp; // base pointer
  char* kstack;
  addr_space_id addr_space;
  int state;
  struct regs regs;
  char cmdline[CMDLINE_LENGTH];
  thread_id next; // id of the next thread
  thread_id prev; // id of the previous thread
  boolean used;
  short int fixstack;
  int exitcode;
  int waitfor;    // in case of waitpid(): the pid of the child we wait for
} TCB;

extern TCB thread_table[MAX_THREADS];
```

Defines:
byte, used in chunks 15, 39, 41–44, and 62.
size_t, used in chunks 37a and 55.
TCB, never used.
thread_id, never used.
u16int, never used.
uint, used in chunks 34, 42, 44–46, 55, and 61b.
Uses boolean 56a, CMDLINE_LENGTH 56a, GPR_REGISTERS 56a, and MAX_THREADS 56a.

A.2. ELF-Definitionen

Die hier aufgeführten Definitionen werden für ELF benötigt. Alle vom ULIX-i386-Programm-Loader verwendeten Definition werden in den Kapiteln 1 und 3 erläutert.

58 ⟨*ELF-Definitionen* 58⟩≡ (31a)

```
// ELF header e_ident[] (Identification Indexes)
#define EI_MAG0     0         //  File identification
#define EI_MAG1     1         //  File identification
#define EI_MAG2     2         //  File identification
#define EI_MAG3     3         //  File identification
#define EI_CLASS    4         //  File class
#define EI_DATA     5         //  Data encoding
#define EI_VERSION  6         //  File version
#define EI_PAD      7         //  Start of padding bytes
#define EI_NIDENT   16        //  Size of e_ident[]

// ELF header EI_MAG[0-3]
#define ELFMAG0     0x7f      //  e_ident[EI_MAG0]
#define ELFMAG1     'E'       //  e_ident[EI_MAG1]
#define ELFMAG2     'L'       //  e_ident[EI_MAG2]
#define ELFMAG3     'F'       //  e_ident[EI_MAG3]

// ELF header EI_CLASS
#define ELFCLASS32    1       //  32-bit objects

// ELF header EI_DATA
#define ELFDATA2LSB   1       //  2er-Komplement und little-endian

// ELF header e_type
#define ET_EXEC 2             //  Exeutable file

// ELF header e_machine
#define EM_386  3             //  Intel Architecture

// ELF header e_version
#define EV_NONE 0       //  Invalid version
#define EV_CURRENT 1    //  Current version

// Segment Types
#define PT_NULL         0
#define PT_LOAD         1
#define PT_DYNAMIC      2
#define PT_INTERP       3
#define PT_NOTE         4
#define PT_SHLIB        5
#define PT_PHDR         6
#define PT_LOPROC       0x70000000
#define PT_HIPROC       0x7fffffff
```

59a ⟨*ELF-Typ-Definitionen* 15⟩+≡ (31a) ◁18

```
// Elf32_Shdr     (ELF-Section-header)
// ------------------------------------------------------------------------
//
typedef struct {
  Elf32_Word      sh_name;
  Elf32_Word      sh_type;
  Elf32_Word      sh_flags;
  Elf32_Addr      sh_addr;
  Elf32_Off       sh_offset;
  Elf32_Word      sh_size;
  Elf32_Word      sh_link;
  Elf32_Word      sh_info;
  Elf32_Word      sh_addralign;
  Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

Uses Elf32_Addr 15, Elf32_Off 15, and Elf32_Word 15.

59b ⟨*ELF-Programm-Loader Definitionen* 59b⟩≡ (31b)

```
#define __NR_waitpid              7       // Von Ulix vorgegeben
#define __NR_exec                 11      // Von Ulix vorgegeben
#define __NR_printstack           0x1137  // Beliebiger Wert, der noch
                                          // nicht von Ulix verwendet wird
#define DO_LOAD_WORKAROUND                // temp definition
//#define ELF_DEBUG_MSG                     // temp definition

#define _MODULENAME_              "ELF-Loader"
#define PROCESS_IMAGE_SIZE        32*1024
#define STACK_WIDTH               sizeof(unsigned int*)

#define ELF_TEST_MSG
#define TEST_ERROR_MSG(msg)       printf("[LOADER-TEST] ERROR: %s\n",msg);
#define MAX_FILE_LENGTH           11
⟨EXEC ERROR Definitionen 59c⟩

void syscall_exec (struct regs *r);
void initialize_module();
```

Defines:
__NR_exec, used in chunk 33.
__NR_printstack, used in chunk 33.
__NR_waitpid, never used.
MODULENAME, used in chunk 43a.
DO_LOAD_WORKAROUND, used in chunk 43a.
ELF_TEST_MSG, used in chunks 17 and 36–40.
MAX_FILE_LENGTH, used in chunk 36b.
PROCESS_IMAGE_SIZE, used in chunks 36a, 43a, 45b, 46, and 61b.
STACK_WIDTH, used in chunk 61b.
TEST_ERROR_MSG, used in chunks 17 and 36–40.
Uses initialize_module 33 and syscall_exec 35.

59c ⟨*EXEC ERROR Definitionen* 59c⟩≡ (47a 59b)

```
#define EIO               5
#define ENOEXEC           8
#define ENOMEM           12
```

```
#define EACCESS          13
#define EFAULT           14
#define ETXTBSY          26
#define ENAMETOOLONG     36
```

Defines:
EACCESS, used in chunk 40a.
EFAULT, used in chunk 36a.
EIO, used in chunks 38d and 39.
ENAMETOOLONG, used in chunk 36b.
ENOEXEC, used in chunk 17.
ENOMEM, used in chunk 39.
ETXTBSY, never used.

B. Hilfsfunktionen

Die in diesem Kapitel definierten Funktionen werden nicht mehr vom ELF-Programm-Loader verwendet. Sie wurden während der Entwicklung zu Diagnosezwecken eingesetzt. Damit diese Funktionen übersetzt werden, muss die symbolische Konstante `USE_HELPER_FUNCTIONS` definiert werden.

61a ⟨*Hilfsfunktionen (Deklaration)* 61a⟩≡ (31b)

```
void syscall_printstack(struct regs *r);
void printProgramHeader( Elf32_Ehdr* pElfHeader, Elf32_Phdr* pProgramHeader );
void printSectionHeader(Elf32_Shdr* pCurrentSectionHeader, const char* pShStrTab);
void printElfHeader( Elf32_Ehdr* header );
```

Uses `Elf32_Ehdr` 16, `Elf32_Phdr` 18, `printElfHeader` 61c, `printProgramHeader` 62, `printSectionHeader` 63, and `syscall_printstack` 61b.

Der *Syscall-Handler* `syscall_printstack` gibt den Usermode-Stack des aufrufenden Programms auf der Standardausgabe aus.

61b ⟨*Hilfsfunktionen (Implementation)* 61b⟩≡ (32a) 61c ▷

```
void syscall_printstack(struct regs *r)
{
  printf("PRINTSTACK START SP: 0x%X\n",r->useresp);
  uint t = 1;
  while(r->useresp+STACK_WIDTH*t < PROCESS_IMAGE_SIZE)
  {
    printf("tid: %d stack: 0x%8X  %8X\n",
            current_task,
            r->useresp+STACK_WIDTH*t,
            *((uint*)(r->useresp+STACK_WIDTH*t)));
    t += 1;
  }
  printf("PRINTSTACK END\n");
}
```

Defines:
`syscall_printstack`, used in chunks 33 and 61a.
Uses `PROCESS_IMAGE_SIZE` 59b, `STACK_WIDTH` 59b, and `uint` 56b 56b.

Die Funktion `printElfHeader` gibt Informationen über den ELF-Header auf der Standardausgabe aus.

61c ⟨*Hilfsfunktionen (Implementation)* 61b⟩+≡ (32a) ◁61b 62 ▷

```
void printElfHeader( Elf32_Ehdr* header )
{
```

```
  int idx;
  printf("**********************\n");
  printf("***** ELF-HEADER *****\n");
  printf("**********************\n");
  printf("e_ident:     ");
  for (idx=0;idx<EI_NIDENT;idx++)
    printf(" %02x",header->e_ident[idx]);
  printf("\n");
  printf("e_ident:      %c%c%c\n",
         header->e_ident[1],header->e_ident[2],header->e_ident[3]);
  printf("e_version:    %u\n",header->e_version);
  printf("e_entry:      0x%x\n",header->e_entry);
  printf("Start of program headers: %u (offset - bytes into file)\n",
         header->e_phoff);
  printf("Start of section headers: %u (offset - bytes into file)\n",
         header->e_shoff);
  printf("e_flags:      0x%x\n",header->e_flags);
  printf("e_ehsize:     %u\n",header->e_ehsize);
  printf("e_phentsize:  %u\n",header->e_phentsize);
  printf("e_phnum:      %u\n",header->e_phnum);
  printf("e_shentsize:  %u\n",header->e_shentsize);
  printf("e_shnum:      %u\n",header->e_shnum);
  printf("e_shstrndx:   %u\n",header->e_shstrndx);
}
```

Defines:
printElfHeader, used in chunk 61a.
Uses Elf32_Ehdr 16 and x 19a.

Die Funktionen printProgramHeader gibt Informationen aller *Programm-Header* auf der Konsole aus.

62 ⟨*Hilfsfunktionen (Implementation)* 61b⟩+≡ (32a) ◁61c 63▷

```
void printProgramHeader( Elf32_Ehdr* pElfHeader, Elf32_Phdr* pProgramHeader )
{
  int idx;
  /* Zeiger auf das aktuelle Code Segment ermitteln */
  byte* pCodeSegment = (byte*) (
      ((byte*)pElfHeader)
    + pProgramHeader->p_offset
  );

  printf( "\nSegment ");

  // Segment-Code in hex ausgeben
  for ( idx=1; idx-1<pProgramHeader->p_memsz;idx++)
  {
    if (idx%16 == 1)
      printf("\n0x%08X ",idx-1+pProgramHeader->p_offset);
    else if (idx%8  == 1)
      printf(" ");

    printf(" %0x02x",pCodeSegment+idx-1);
```

```
    }
    printf("\n");
    return;
  }
```

Defines:
  `printProgramHeader`, used in chunk 61a.
Uses `byte` 56b, `Elf32_Ehdr` 16, and `Elf32_Phdr` 18.

Die Funktionen `printSectionHeader` gibt die Informationen eines Sektionsheaders aus.

63 ⟨*Hilfsfunktionen (Implementation)* 61b⟩+≡ (32a) ◁62

```
  void printSectionHeader(Elf32_Shdr* pCurrentSectionHeader, const char* shstrtab)
  {
    printf("sh_name:      %s\n", shstrtab+pCurrentSectionHeader->sh_name);
    printf("sh_type:      %u\n", pCurrentSectionHeader->sh_type);
    printf("sh_flags:     %u\n", pCurrentSectionHeader->sh_flags);
    printf("sh_addr:      0x%X\n", pCurrentSectionHeader->sh_addr);
    printf("sh_offset:    %u\n", pCurrentSectionHeader->sh_offset);
    printf("sh_size:      %u\n", pCurrentSectionHeader->sh_size);
    printf("sh_link:      %u\n", pCurrentSectionHeader->sh_link);
    printf("sh_info:      %u\n", pCurrentSectionHeader->sh_info);
    printf("sh_addralign  %u\n", pCurrentSectionHeader->sh_addralign);
  }
```

Defines:
  `printSectionHeader`, used in chunk 61a.

Chunk Index

⟨ *UlixTestRun.c* 47a⟩
⟨ *Usermode-Stack initialisieren* 45b⟩

Identifier Index

Index

Abbildungsverzeichnis

Listings

Quellenverzeichnis

[Bel] Fabrice Bellard. Qemu. http://wiki.qemu.org/Main_Page, accessed online: 2013/28/02.

[Com95] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. *TIS Committee*, (May), 1995.

[FE] Felix Freiling and Hans-Georg Eßer. *Design and Implementation of the Ulix-i368 Operating System.* dissertation, Friedrich-Alexander-Universität Erlangen-Nürnberg.

[lat] Latex documentation. http://latex-project.org/guides/, accessed online: 2013/27/02.

[mana] Exec - linux programmer's manual. http://www.kernel.org/doc/man-pages/, accessed online: 2013/27/02.

[manb] Execve - linux programmer's manual. http://www.kernel.org/doc/man-pages/, accessed online: 2013/28/02.

[Mau04] Wolfgang Mauerer. *Linux Kernelarchitektur.* Carl Hanser Verlag München Wien, 2004.

[Ora] Oracle. Virtual box. https://www.virtualbox.org/, accessed online: 2013/28/02.

[Tan87] Andrew S. Tanenbaum. *Operating Systems - Design and Implementation.* Prentice Hall Inc., 1987.

[Wik13a] Wikipedia. Executable and linking format, 2013. http://de.wikipedia.org/wiki/Executable_and_Linking_Format, accessed online: 2013/12/02.

[Wik13b] Wikipedia. Executable and linking format, 2013. http://en.wikipedia.org/wiki/Executable_and_Linkable_Format, accessed online: 2013/12/02.

[Wik13c] Wikipedia. Literate programming, 2013. http://de.wikipedia.org/wiki/Literate_programming, accessed online: 2013/20/02.