FOM Hochschule für Oekonomie & Management

Studienzentrum München

Bachelor-Thesis

zur Erlangung des Grades eines

Bachelor of Science (B.Sc.)

über das Thema

Implementation eines Paging-Subsystems für das Lehrbetriebssystem Ulix mit Literate Programming

von

Florian Knoll

Erstgutachter	Dipl.-Math., Dipl.-Inf. Hans-Georg Eßer
Matrikelnummer	256141
Abgabedatum	2014-05-28

Inhaltsverzeichnis

Abkürzungsverzeichnis

CPU Central Processing Unit

FIFO „First-In-First-Out“-Algorithmus

LRU „Least-Recently-Used“-Algorithmus

MMU Memory Management Unit

NRU „Not-Recently-Used“-Algorithmus

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Durch den stetigen Wandel und die Schnelllebigkeit in der Informationstechnologie wird für viele Softwaresysteme ein hoher Grad an Anpassungsfähigkeit und Wartbarkeit gefordert. Software soll sich möglichst schnell und mit akzeptablen Aufwand bzw. geringen Kosten an geänderte Ablaufprozesse oder neue Schnittstellen anpassen lassen. Dieser Umstand hat den Effekt, dass sich neben dem Entstehen einer möglichen Architekturerosion im Softwaresystem auch die Qualität der Programmdokumentation verschlechtert [Rie09, S. 339]. Die Ursachen hierfür finden sich zum einen im Zeit- und Kostendruck, dem Entwickler für erforderliche Anpassungen in der Software ausgesetzt sind. Zum anderen wird die Relevanz einer ausführlichen Dokumentation für die Wartbarkeit eines Softwaresystems oft unterschätzt [Leh93, S. 133]. Eine unvollständige oder obsolete Programmdokumentation erschwert jedoch die Einarbeitung in den Quellcode und damit die Möglichkeit des Entwicklers, die Software zu warten.

Sowohl die Forschung als auch die Lehre in den Informatik-Bereichen sind mit einer ähnlichen Problemstellung konfrontiert. Zum einen sollen sich Implementierungen (bspw. in Vorlesungs-Skripten) mit den dazugehörigen Ablaufbeschreibungen decken. Zum anderen sollen sich die Ablaufbeschreibungen möglichst verständlich und vermittelbar gestalten [LiT12b, S. 1723].

Einen möglichen Ansatz, den genannten Anforderungen gerecht zu werden, bietet das Paradigma des Literate Programming. Laut D. Knuth soll es durch eine engere Verzahnung von Dokumentation und Programmcode die Aktualisierung beider Elemente vereinfachen. Zudem ermöglicht es eine verständlichere Beschreibung des Quellcodes, die nicht an den Programmablauf des Algorithmus gebunden ist [Knu84, S. 97].

1.1 Stand der Forschung

Die Literatur-Recherche im Themenbereich Literate Programming zeigt, dass sich diverse Untersuchungen und wissenschaftliche Arbeiten mit diesem Themengebiet auseinandersetzen. Neben den Grundsatzfragen zu diesem alternativen Programmieransatz, mit denen sich u.a. die Ausarbeitungen von D. Knuth beschäftigen, lassen sich die gesichteten Arbeiten in folgende Bereiche untergliedern:

Ein Großteil der Ausarbeitungen fokussiert die Weiterentwicklung des ursprünglichen Systems WEB bzw. des damit verbundenen Programmieransatzes von D. Knuth. So erforschen bspw. Zeng [Zen91], Leisch [Lei02], Brown und Czejdo [Bro90] sowie Li-Thiao-Té [LiT12a] die Weiterentwicklung des Systems zur Verwendung weiterer Programmiersprachen, dynamischer Auswertungen und Datenbankabfragen.

Andere Arbeiten analysieren das Paradigma an sich und versuchen, dies mit anderen Konzepten zu kombinieren bzw. es auf andere Anwendungsbereiche zu adaptieren. Pepper [Pep91] vereint bspw. den Gedanken des Literate Programming mit dem

Konzept des Deductive Programming. Simons und Weber [Sim96] hingegen erörtern die Ausweitung des Literate-Programming-Ansatzes zur Beschreibung und Strukturierung formaler Entwicklungen. Ein weiteres Beispiel ist die Arbeit von Spotnitz [Spo98], welche die Anwendung von Literate Programming im Chemieingenieurwesen untersucht.

Eine ebenfalls große Teilmenge der Arbeiten beschreibt die Implementierung von Applikationen mithilfe des Literate Programming: Während Enquobahrie et al. [Enq07] z.B. eine Neuentwicklung im chirurgischen Bereich vorstellen, beschäftigen sich Ramsey und Marceau [Ram91] mit der Anwendbarkeit von Literate Programming anhand einer Implementierung im Rahmen eines Team-Projekts.

Einige weitere wissenschaftliche Ausarbeitungen erforschen u.a. die Anwendbarkeit von Literate Programming im Lehrbereich: So evaluiert z.B. Hurst [Hur96] mit einer semi-automatischen Korrekturmöglichkeit von Studienarbeiten die Optimierung der Arbeitsabläufe für Dozenten mittels Literate Programming. Shum und Cook [Shu95] hingegen erforschen, inwiefern sich mit Literate Programming gute Programmierpraktiken vermitteln lassen. Und Li-Thiao-Té [LiT12b] versucht mit Literate Programming via Lepton Diskrepanzen zwischen Informatik-Vorlesungsmaterial und Code-Beispielen, welche durch unabhängige Aktualisierungen entstehen können, entgegenzuwirken. Lepton ist hierbei ein Tool, das neben der Code-Extrahierung aus einem Literate Programming-Dokument auch die Kompilierung und Ausführung sowie die anschließende Integration der Programm-Ergebnisse in die Dokumentation bewerkstelligt.

1.2 Problemstellung

Die im Zuge der Dokumentenrecherche gesichteten Arbeiten untersuchen primär die Anwendbarkeit von Literate Programming zur Erstellung von Dokumentationen, deren Hauptadressaten Anwendungsentwickler sind. Im Bereich der Lehre beschäftigen sich zwar diverse Arbeiten mit der grundsätzlichen Anwendbarkeit von Literate Programming. Im Bezug auf die Recherchen im Rahmen dieser Arbeit kann jedoch davon ausgegangen werden, dass die Vermittlung der Funktionsweise komplexer Anwendungen an Studenten unter Verwendung von Literate Programming bisher nicht vollumfassend betrachtet wurde. Auch zur Entwicklung von Betriebssystemen (bzw. deren Komponenten) mittels Literate Programming wurden im Zuge der Dokumentenrecherche keine Arbeiten gefunden.

Es soll daher in einem Forschungsprojekt untersucht werden, wie praktikabel die Lehre einer mittels Literate Programming erstellten komplexen Anwendung im Rahmen einer Vorlesung ist. Das Projekt von Hans-Georg Eßer soll evaluieren, inwiefern ein mit Literate Programming erstelltes Lehrbuch, das den kompletten Quellcode des Betriebssystems Ulix vorstellt, bei Studenten das Verständnis der Betriebssystem-Theorie im Rahmen einer Vorlesung verbessert.

Als Teil dieses Projektes wird die Paging-Komponente des erwähnten Betriebssystems unter Verwendung von Literate Programming und der Programmiersprache C implementiert und deren Funktionsweise unter Einbezug der theoretischen Grundla-

gen erläutert. Der Fokus wird hierbei auf eine möglichst anschauliche und nachvollziehbare Kodierung gelegt, da später die Arbeitsweise dieses Subsystems Studenten in Vorlesungen nähergebracht werden soll.

1.3 Forschungsmethodik

Die systematische Erarbeitung des Themengebietes wurde unter Einbeziehung des Frameworks für wissenschaftliche Literaturrecherche nach H. Cooper durchgeführt [Coo88, S. 104 ff.]. Hierdurch konnte der Themenbereich um die Problemstellung herum entsprechend eingegrenzt und fokussiert werden. Abbildung 1.1 skizziert den schematischen Aufbau des Frameworks.

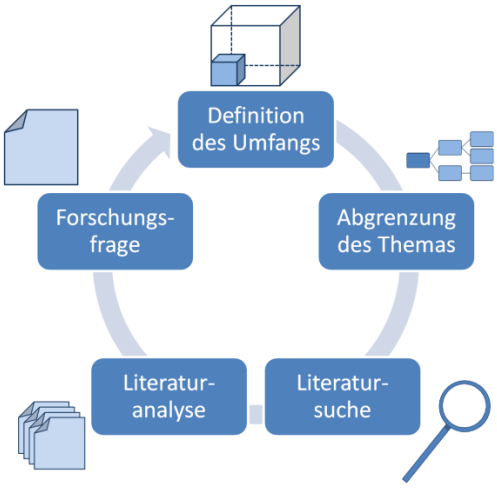

Abbildung 1.1: In Anlehnung an [Bro09, S. 7]: Literaturrecherche gemäß dem Framework nach H. Cooper

Das Framework beschreibt das Vorgehen bei Literaturrecherchen zur Erschließung und Bearbeitung von Forschungslücken. Im ersten Schritt „Definition des Umfangs“ soll der Blickwinkel des zu untersuchenden Objekts möglichst genau definiert werden, um die Herangehensweise der Untersuchung in Art und Umfang möglichst zielführend zu gestalten. Darauf aufbauend, kann im nächsten Schritt „Abgrenzung des Themas“ der Fokus auf den festgelegten Umfang gerichtet, und mit der Literaturrecherche begonnen werden. Nach den weiteren Schritten „Literatursuche“ und „Literaturanalyse“ lässt sich schließlich die Forschungsfrage präzisieren [Coo88, S. 108 ff.].

Wie Abbildung 1.1 andeutet, lassen sich durch iteratives Vorgehen anhand mehrerer Literaturrecherchen mit jeweils angepasstem Fokus Forschungslücken identifizieren und verifizieren. So wird im ersten Durchlauf die Forschungslücke eingegrenzt. In einem zweiten Durchlauf (mit Schwerpunkt auf das eingegrenzte Themengebiet) wird geprüft, ob es bereits Untersuchungen bzw. wissenschaftliche Arbeiten zu dem Thema gibt. Ein dritter Durchlauf der Literaturrecherche soll letztlich zur Schließung bzw. Verkleinerung der identifizierten Forschungslücke beitragen.

1.3.1 Dokumentenanalyse

Die in der Literaturrecherche gewonnenen Erkenntnisse basieren primär auf der Methodik der Dokumentenanalyse. Als Form der Sekundärerhebungsmethoden beschreibt sie die Erarbeitung, Verarbeitung und Interpretation von Informationen aus bestehenden Dokumenten. Diese können in Form von Texten, Zahlen, Grafiken, Bildern sowie Zeichnungen (in elektronischer oder gedruckter Form) vorliegen. Die Recherchen im Zuge dieser Arbeit wurden primär unter Verwendung von Monografien, Fachartikeln und Webseiten durchgeführt. Um die Informationsqualität der Nachweise zu gewährleisten, wurden Journals, die nicht VHB-gelistet sind, sowie Monografien mit weiteren gleichwertigen Quellen validiert [Hoe04, S. 24 f.].

1.3.2 Vorgehensmodell für die Implementierung

Im Rahmen der vorliegenden Arbeit wurde die Implementierung des Paging-Subsystems anhand des Wasserfallmodells durchgeführt. Dieses Vorgehensmodell eignet sich für derartige Projekte, da der Entwickler alle Prozessschritte selbst durchführt [Ruf08, S. 31 f.]. Abbildung 1.2 stellt die Einzelschritte des Wasserfallmodels kurz dar.

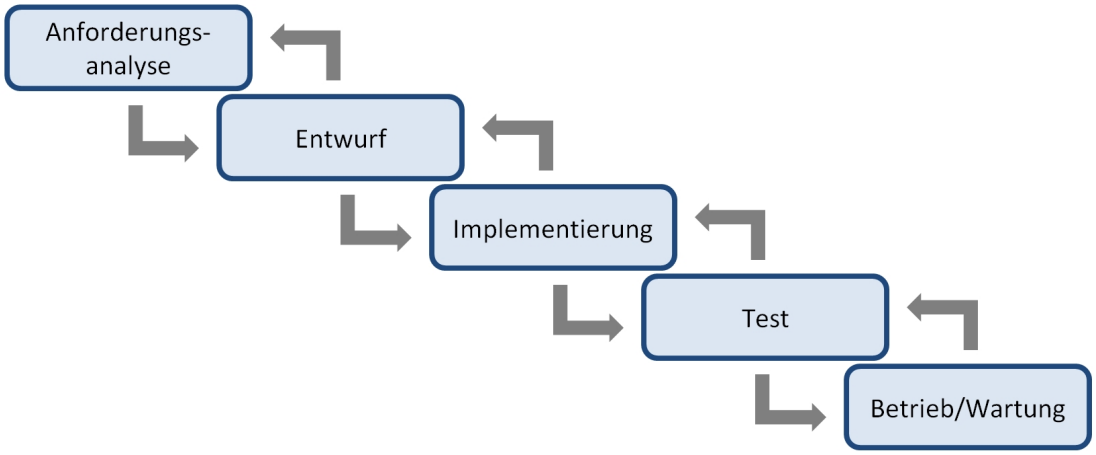

Abbildung 1.2: In Anlehnung an [Ber03, S. 4]: Wasserfallmodell

Die abgebildeten Projektphasen werden, wie dargestellt, sequenziell abgearbeitet. Mit der nächsten Stufe wird erst begonnen, wenn die aktuelle Phase abgeschlossen und validiert wurde. Stellt sich in einem Abschnitt heraus, dass die vorhergehende Stufe noch nicht abgeschlossen ist, so ist diese zuerst zu beenden, bevor mit dem aktuellen Abschnitt weiter verfahren wird [Ruf08, S. 31 f.].

Dem genannten Modell folgend, wurden für diese Arbeit in der Phase der Anforderungsanalyse die theoretischen Grundlagen zur Funktionsweise von Paging-Systemen erarbeitet. Anschließend wurde in Absprache mit H. Eßer der Funktionsumfang des Paging-Systems für Ulix eingegrenzt. Basierend auf diesen abgesprochenen Anforderungen erfolgte in Ableitung der erarbeiteten Theorie die Gestaltung der Detaillösungen für das Subsystem in der Phase „Entwurf“. Darauf aufbauend wurde

ebenfalls der thematische Aufbau dieser Arbeit entworfen, da in der anschließenden „Implementierung“ die Kodierung und die Dokumentation eng miteinander verzahnt sind. Nach abgeschlossenem Entwurf begann die Implementierung der Teilelemente, gefolgt von Funktionstests. Mit Bestehen der Funktionstests wurde das Subsystem in Ulix integriert.

Da der Fokus dieser Arbeit auf der Implementierung und Funktionsbeschreibung des Paging-Subsystems liegt, wird auf die Vorgehensweise während der Entwicklung nicht weiter eingegangen.

1.4 Rahmen der Arbeit

Nach einleitenden Worten zu Dokumentationsproblemen im Rahmen der Softwareentwicklung in Wirtschaft, Forschung und Lehre, wird im Bezug auf Literate Programming der aktuelle Forschungstand dargelegt. Des Weiteren folgen die thematische Abgrenzung der Problemstellung sowie die Beschreibung der verwendeten Forschungsmethoden. Mit einer Skizzierung des thematischen Aufbaus dieser Arbeit endet Kapitel 1.

Kapitel 2 geht auf die Entstehung des Literate-Programming-Paradigmas ein und beschreibt anschließend die Funktionsweise sowie die grundlegenden Syntaxregeln des Tools noweb.

Das Kapitel 3 befasst sich mit der Implementierung des Paging-Subsystems. Mit einem Überblick auf die erstellten Quelldateien wird zuerst ein Bild vom Aufbau des Subsystems vermittelt. Nach der grundlegenden Erklärung des Pagings und der virtuellen Speicherverwaltung in Kapitel 3.2, erfolgt in den Kapiteln 3.3 und 3.4 die Definition der für das Paging erforderlichen Datenstrukturen sowie des Zugriffs darauf. Kapitel 3.5 stellt den Aufbau der Auslagerungsdateien vor und erläutert zudem die Funktionen für Zugriff auf diese Dateien. Hierauf aufbauend folgt in Kapitel 3.6 die Umsetzung des Aus- und Einlagerungsmechanismus. Unter Verwendung der vorgestellten Mechanismen wird in Kapitel 3.7 der Page Fault Handler implementiert. Mit der Erstellung des Seitenersetzungsalgorithmus und der Speicherüberwachung in Kapitel 3.8 ist das Kapitel 3 und damit die Implementierung abgeschlossen.

Im Kapitel 4 wird die Funktionsfähigkeit der vorherigen Implementierung anhand der Testfälle für Einlagerung, Auslagerung, Seitenersetzung und Zufallsgenerator überprüft.

Abschließend folgt in Kapitel 5 ein Fazit zur Anwendbarkeit von Literate Programming mit einer kritischen Würdigung des methodischen Vorgehens und der Implementierung sowie ein Hinweis auf den weiteren Forschungsbedarf in diesem Themenbereich.

2 Literate Programming

Literate Programming ist eine alternative Herangehensweise bei der Entwicklung von Programmen. Der von D. Knuth entwickelte Programmieransatz rückt den Menschen als Hauptadressat des Quellcodes in den Mittelpunkt: Statt primär den Computer zu instruieren, soll das Hauptaugenmerk darauf liegen, anderen Menschen die Intention der jeweiligen Computer-Instruktionen näher zu bringen [Knu84, S. 97]. Knuth sieht den Programmierer eher in der Rolle des Autors, der ein Programm und seine Funktionen in einer für den Menschen sinnvollen, verständlichen Art und Reihenfolge (nicht zwangsläufig dem Programmablauf folgend) erläutert [Knu84, S. 97 f.]. Dabei sollen sich formale Elemente (Computerinstruktionen) und informale Elemente (Dokumentation der Instruktionen) gegenseitig ergänzen.

Um diesen Programmieransatz umsetzbar und praktikabel zu gestalten, entwickelte Knuth das System WEB, welches ursprünglich PASCAL, und in Spezialversionen andere Programmiersprachen wie bspw. COBOL, FORTRAN oder C (z.B. über CWEB) mit dem Textsatzsystem TEX kombinierbar machte [Knu84, S. 98]. In Anlehnung daran wurde von Norman Ramsey das Tool noweb entwickelt, welches den ursprünglichen Funktionsumfang von WEB auf grundsätzlich alle Programmiersprachen erweitert. Darüber hinaus werden die Textsatzsysteme TEX und LATEX sowie die Auszeichnungssprache HTML unterstützt [Ram94, S. 102, S. 105]. Aufgrund der einfacheren Handhabung von noweb im Vergleich zu WEB wird für die Implementierung des Paging-Subsystems in dieser Arbeit noweb herangezogen.

In Abgrenzung zum Literate Programming sind Konzepte wie bspw. Dokumentationsgeneratoren oder Programming by Intention zu nennen, die ein ähnliches Ziel (eine bestmögliche Dokumentation) mithilfe einer anderen Herangehensweise verfolgen. Da der Fokus dieser Arbeit auf der Methodik des Literate Programming liegt, werden keine weiteren Konzepte behandelt.

2.1 Funktionsweise

Eine noweb-Datei besteht aus einer beliebigen Aneinanderreihung von Blöcken (sog. Chunks), die entweder Quellcode oder Dokumentation beinhalten. Die Dokumentations-Chunks können u.a. im LATEX-, TEX- oder HTML-Format verarbeitet werden. Das Tool noweave wandelt den Inhalt der noweb-Datei in eine dem Textsatzsystem entsprechend formatierte Datei um [Ram94, S. 100, S. 105].

Die Code-Chunks beinhalten den Quellcode der Anwendung und ggf. Verweise auf weitere Code-Chunks. Sie können mit dem Tool notangle aus der noweb-Datei extrahiert und zu einer Quelldatei zusammengesetzt werden. Die sich daraus ergebende Datei lässt sich anschließend kompilieren und ausführen [Ram01, o.S.]. Ein weiteres Tool, das Programm nountangle, hat grundsätzlich die gleiche Funktionsweise wie notangle jedoch mit dem Unterschied, dass die Dokumentations-Chunks als Kommentare in die Quelldatei eingefügt werden [Ram01, o.S.].

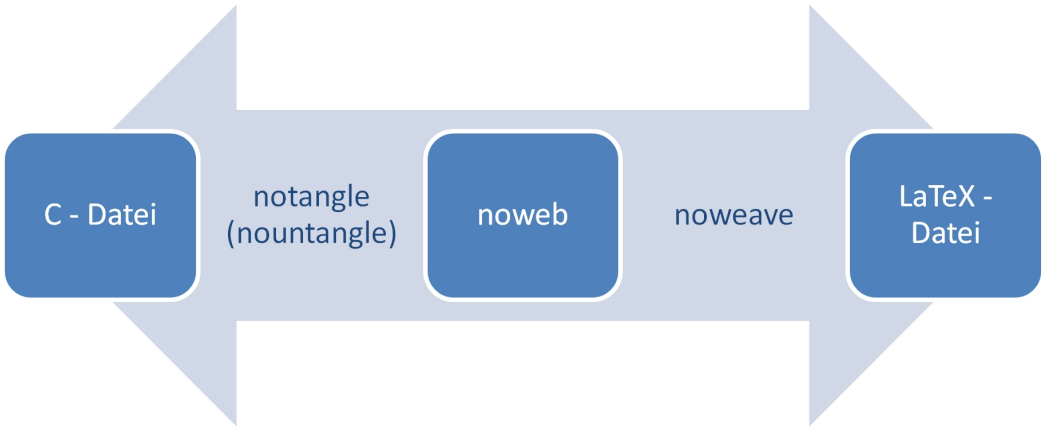

Abbildung 2.1: notangle und noweave

Am Beispiel dieser Arbeit veranschaulicht Abbildung 2.1 die Funktionsweise der oben beschriebenen Kommandozeilentools noweave und notangle. Im nachfolgenden Kapitel wird auf die grundlegenden Syntaxelemente von noweb-Dateien eingegangen.

2.2 Syntax

Die Chunks einer noweb-Datei werden nur an deren Anfang gekennzeichnet. Sie enden mit dem Beginn des nachfolgenden Chunks. Während Dokumentations-Chunks keinen Namen besitzen und mit `@` gekennzeichnet sind, werden Code-Chunks mit Namen versehen. Ein Code-Chunk beginnt mit:

```
<<Name des Code-Chunks>>=
```

Hierbei ist zu beachten, dass die Chunk-Definition am Zeilenanfang steht. Code-Chunks, die denselben Namen haben, werden von notangle anhand der Reihenfolge im noweb-Dokument zu einem Code-Chunk zusammengefasst. Innerhalb der Code-Chunks gelten die Syntaxvorschriften der jeweiligen Programmiersprache. Ausnahmen bilden die Verweise (in den Code-Chunks) auf andere Code-Chunks, die an beliebiger Stelle im noweb-Dokument liegen können. Dies ermöglicht es, Code-Elemente im noweb-Dokument in einer vom Programmablauf abweichenden Reihenfolge aufzuführen. Ein Verweis auf einen Code-Chunk ähnelt der Definition mit dem Unterschied, dass das Zeichen `=` weggelassen wird. Am folgenden Beispiel soll dies kurz verdeutlicht werden:

```
<<Variablendefinitionen>>=
  int a=1, b=1, c, i;

@
Dokumentationstext, der die Variablen und deren Initialwerte erklärt.
```

```
<<Fibonacci iterativ>>=
int fib(int n) {
  // Hier ein Verweis auf den vorherigen Code-Chunk:
  <<Variablendefinitionen>>

  if (n == 0) return 0;

  // Hier ein Verweis auf den nachfolgenden Code-Chunk:
  <<Schleife>>

  return b;
}

@
Dokumentationstext, der den grundsätzlichen Aufbau der iterativen
Fibonacci-Funktion beschreibt.

<<Schleife>>=
  for(i=3; i<=n; i++) {
    c = a + b;
    a = b;
    b = c;
  }

@
Dokumentationstext, der erläutert wie die Werte iterativ ermittelt
werden
```

Die Chunks `Variablendefinitionen`, `Schleife` und `Fibonacci iterativ` definieren neue Code-Chunks, die Codeelemente der Programmiersprache C beinhalten. Beendet werden diese durch Dokumentations-Chunks (`@`), die den Dokumentationstext enthalten. Der Chunk `Fibonacci iterativ` besteht aus der Funktion `fib(int n)`, die in Abhängigkeit vom übergebenen Parameter `n` die Fibonacci-Zahl berechnet und zurückgibt. Innerhalb des Chunks wird auf die Code-Chunks `Variablendefinition` und `Schleife` verwiesen. notangle setzt diese Chunks bei der Umwandlung wieder zu einer vollständigen Funktion zusammen.

Darüber hinaus existieren noch weitere Syntaxelemente, die jedoch für ein grundlegendes Verständnis nicht erforderlich sind. Daher werden diese Elemente im Rahmen der vorliegenden Arbeit nicht weiter erläutert.

3 Implementierung eines Paging-Subsystems für Ulix

Ulix ist ein sich in der Entwicklung befindendes Betriebssystem von Hans-Georg Eßer und Felix Freiling. Es wird unter Verwendung von Literate Programming entwickelt und soll dazu dienen, die Funktions- und Arbeitsweise UNIX/Linux-artiger Betriebssysteme an Informatik-Studenten zu vermitteln. Die hierzu erstellten Dokumentationen sollen die Arbeitsgrundlage für Vorlesungen im Fach „Betriebssysteme Theorie“ bilden. Die Entwicklung dieses Betriebssystems erfolgt im Rahmen des in Kapitel 1.2 beschriebenen Forschungsprojekts.

Im Zuge dieses Forschungsprojektes wird in den folgenden Kapiteln anhand des vorab beschriebenen Literate Programming ein für das Betriebssystem Ulix entworfenes Paging-Subsystem implementiert und erläutert. Im Nachfolgenden wird darauf verzichtet, auf weitere Varianten des Paging bzw. der Seitenverwaltung einzugehen, da im Fokus dieser Arbeit die Seitenverwaltung unter Ulix liegt. Zudem beschränken sich die Erläuterungen zu den Betriebssystemgrundlagen ausschließlich auf Funktionalitäten, die im direkten Zusammenhang mit dem Paging-Subsystem liegen.

Zum besseren Verständnis der Quelldateien wird deren Aufbau im nachfolgenden Abschnitt kurz skizziert. Die Implementierung der hierin genannten Teilelemente wird in den Folgekapiteln näher beschrieben.

3.1 Aufbau der Quelldateien

Die Datei `module.c` beinhaltet mit den Implementierungen der in den Folgekapiteln beschriebenen Funktionen den Hauptteil des Paging-Subsystems. Die Aufstellung im Chunk `module.c` zeigt die Verweise auf die erwähnten Funktionen, mit einem kurzen Hinweis auf ihre Aufgabenbereiche. Mittels `#include "module.h"` wird die dazugehörige Header-Datei `module.h` eingebunden, die im Anschluss erläutert wird.

9 ⟨*module.c* 9⟩≡

```
#include "module.h"

⟨initialize_module 24b⟩ // Systemstart

⟨page_fault 38b⟩ // Seitenfehler behandeln
⟨segfault 43c⟩ // Segmentation Fault ausgeben

⟨kswapd 52d⟩ // Speicherüberwachung

⟨not_recently_used 46a⟩ // Seitenersetzung mit NRU-Strategie
⟨rand 48a⟩ // Zufallszahlengenerator

⟨get_ptd 18a⟩ // Seitentabellen-Deskriptoren auslesen
⟨get_pdesc 18b⟩ // Seiten-Deskriptoren auslesen
⟨print_pdesc 19c⟩ // Seiten(tabellen)-Deskriptoren ausgeben
```

⟨*frame_2_file* 31b⟩ `// Seite auslagern`
⟨*file_2_frame* 35a⟩ `// Seite einlagern`

⟨*chkfile* 23c⟩ `// Auslagerungsdateien erstellen`

⟨*findblock* 27c⟩ `// Eintrag in Auslagerungsdatei suchen`

⟨*rblock* 25c⟩ `// Auslagerungsdateien auslesen`
⟨*wblock* 26a⟩ `// Auslagerungsdateien befüllen`

⟨*nextfreeblock* 28d⟩ `// Freie Auslagerungsdatei-Einträge finden`
⟨*nextfreeindexblock* 29⟩ `// in Indexdatei`
⟨*nextfreeoutpageblock* 30a⟩ `// in Blockdatei (über Bitmap)`

⟨*clearindex* 28b⟩ `// Datenblöcke freigeben`

⟨*get_usedblocklist_bit* 30c⟩ `// Bitmap auslesen`
⟨*set_usedblocklist_bit* 33c⟩ `// Bitmap editieren`

⟨*Testfall Random* 63d⟩ `// Test des Zufallszahlengenerators`
⟨*Testfall P1* 56e⟩ `// Test der Ein-/Auslagerung`

Uses `module.h` 10.

Die Header-Datei `module.h` beinhaltet die Definitionen von Konstanten, Makros, globalen Variablen und eigene Funktionsprototypen sowie die extern eingebundenen Variablen und Funktionsprototypen aus dem Betriebssystem Ulix. Die Teilelemente dieser Datei werden ebenfalls in den nachfolgenden Kapiteln an geeigneter Stelle erläutert.

10 ⟨*module.h* 10⟩≡
⟨*Konstanten* 13a⟩
⟨*Typdefinitionen* 13b⟩

⟨*externe Prototypen* 20b⟩
⟨*externe Variablen* 14b⟩

⟨*globale Variablen* 21a⟩
⟨*Prototypen* 19b⟩

Defines:
`module.h`, used in chunk 9.

Der Aufbau des Paging-Subsystems erstreckt sich somit primär auf die zwei genannten Quellcode-Dateien. Um das Subsystem in Ulix zu integrieren, sind vereinzelt Anpassungen im Quellcode von Ulix (in den Dateien `ulix.c` und `sh.c`) vorgenommen worden. Die betreffenden Code-Elemente werden ebenfalls an geeigneter Stelle aufgeführt und erläutert.

3.2 Paging

Das Paging ist eine Form der Hauptspeicher-Verwaltung in modernen Betriebssystemen. Die Problematik, dass der Hauptspeicher verhältnismäßig wenig Speicherplatz bietet, macht eine effiziente Verwaltung dieser Ressource erforderlich. Neben der Hauptaufgabe – der bloßen Bereitstellung von Arbeitsspeicher für Prozesse – sind hierbei noch weitere Anforderungen zu berücksichtigen. Ziele der Speicherverwaltung sind, insbesondere beim Parallelbetrieb von Prozessen, folgende Punkte [Ehs05, S. 287 f.]:

- Bereitstellen von Speicher: Die laufenden Prozesse sollen Speicher zur Verfügung haben.
- Speicherschutz bzgl. Kernel: Der Speicherbereich des Kernels muss vor dem Zugriff durch Prozesse geschützt sein.
- Speicherschutz bzgl. anderer Prozesse: Ein Prozess darf nur auf die ihm zugewiesenen Speicherbereiche zugreifen.
- Relokation: Die Ausführung einer Anwendung soll nicht an bestimmte Speicheradressen gebunden sein.

Um diese Ziele zu erreichen, findet die Abstraktion des physischen Hauptspeichers Anwendung. Hierdurch wird für jeden laufenden Prozess ein eigener virtueller Speicherbereich geschaffen, der den real verfügbaren physischen Speicher abstrahiert. Jedem Prozess wird hierbei ein logischer Speicherbereich zur Verfügung gestellt, der für alle Prozesse – unabhängig vom vorhandenen physischen Speicher – gleich groß und gleich adressierbar ist [Tan09, S. 243].

Der logische Speicherbereich wird gem. Paging in sog. Seiten untergliedert. Diese werden im Hauptspeicher in Speicherabschnitte (sog. Seitenrahmen) der gleichen Größe geschrieben. Diese Zerstückelung hat zur Folge, dass (temporär) nicht benötigte Teile auf die Festplatte verschoben werden können, ohne dass der gesamte Prozess auslagert werden muss. Zudem ist es für die Prozesse dadurch irrelevant, ob die Seiten physisch zusammenhängend oder in einer bestimmten Reihenfolge im Hauptspeicher abgelegt sind. Die virtuelle Adressierung bleibt für den Prozess somit immer gleich, egal an welcher physischen Stelle im Hauptspeicher oder (ausgelagert) auf der Festplatte sich die Seite befindet [Ehs05, S. 292] [Tan09, S. 244 ff.]. Abbildung 3.1 soll diese Abstraktion am Beispiel von zwei Prozessen veranschaulichen:

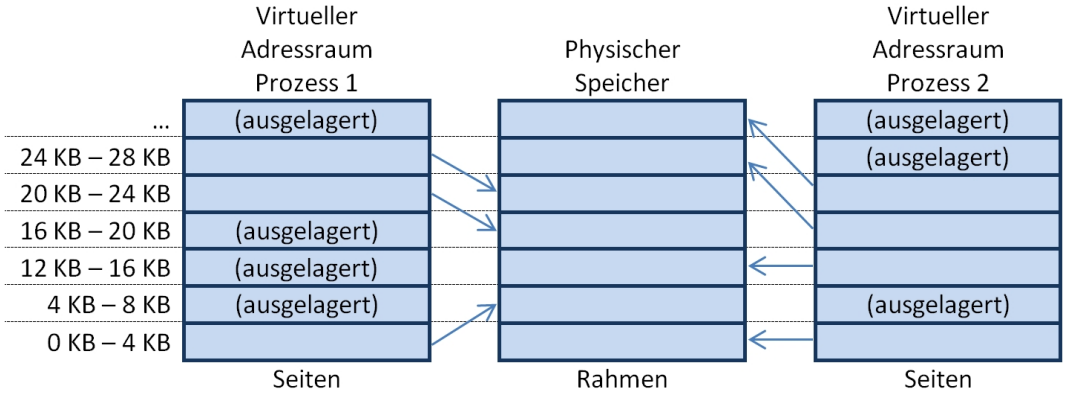

Abbildung 3.1: In Anlehnung an [Tan09, S. 243]: Virtueller Adressraum

Dies ermöglicht zum einen den Parallelbetrieb mehrerer Prozesse mit unterschiedlichen physischen Speicherbereichen im Hauptspeicher. Zum anderen müssen Prozesse während ihrer Ausführung nicht mehr komplett im Hauptspeicher liegen. Das Ziel der „Relokation“ lässt sich damit erfüllen.

Die partielle Auslagerung schafft darüber hinaus Platz für weitere Prozesse, denen dann (zumindest virtuell) der komplette Adressraum zur Verfügung steht. Das Ziel der „Speicherbereitstellung“ ist damit erfüllt. Des Weiteren sieht jeder Prozess durch die Zuweisung von logischen Adressräumen nur seinen eigenen Speicherbereich. Diese Abstraktion verhindert, dass auf Bereiche außerhalb des zugewiesenen Adressraums zugegriffen wird [Tan09, S. 232 ff.]. Durch diesen Umstand ist auch das Ziel des „Speicherschutzes bzgl. anderer Prozesse“ erfüllt.

Im Bezug auf den Schutz des Kernelspeicherbereichs ist die Differenzierung zwischen User-Mode und Kernel-Mode erforderlich. Um das Betriebssystem zu schützen, werden Prozesse des Endanwenders im sog. User-Mode ausgeführt. Prozesse in diesem Modus können nicht direkt auf die Hardware oder die Funktionen des Betriebssystems zugreifen. Das Betriebssystem hingegen agiert im sog. Kernel-Mode, der diesbzgl. privilegierter ist [Sta05, S. 56, S. 136]. Hinzu kommt, dass der Adressraum 0 des Kernels, wie jeder andere fremde Adressraum, für einen anderen (User-Mode-)Prozess nicht sichtbar ist. Somit lässt sich auch das Ziel „Speicherschutz bzgl. Kernel“ mit dem Konzept des Pagings realisieren.

Die Seiten unter Ulix haben (wie in Abbildung 3.1 dargestellt) eine Größe von 4 KByte, d.h. jede Seite hat exakt 4096 (bei 0 beginnend) Adressen. Möchte bspw. Prozess 1 aus der Abbildung 3.1 auf Adresse 1 der 1. Seite zugreifen, so ermittelt die Memory Management Unit (MMU) die korrekte physische Adresse 4097. Die virtuellen Adressen werden nicht – wie bei Rechnern ohne virtuelle Speicherverwaltung – direkt auf den Speicherbus gelegt. Der Zugriff erfolgt über die MMU, einer Hardwarekomponente im Rechner, die ausgehend von der virtuellen Adresse die physische Adresse ermittelt. [Tan09, S. 242 ff.]

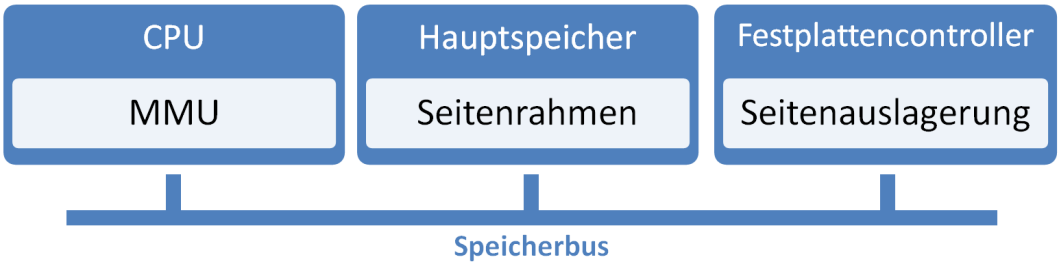

Abbildung 3.2: In Anlehnung an [Ehs05, S. 293] und [Tan09, S. 242]: Position der MMU

Während die MMU in älteren Rechnern als separierter Chip verbaut war, ist sie heute integrierter Bestandteil der Central Processing Unit (CPU) [Tan09, S. 242 ff.]. Zur Ermittlung einer physischen Adresse greift die MMU auf sog. Seitentabellen zu. Die Systematik und der Aufbau dieser Seitentabellen unter Ulix werden im folgenden Kapitel 3.3 dargestellt.

3.3 Seitentabellen

Die Seitentabellenstruktur in Ulix untergliedert sich in drei Ebenen. Es werden 1024 Adressräume verwaltet. Pro Adressraum (Typ `pf_address_space`) existiert ein Page Directory. Dieses Directory enthält bis zu 1024 Seitentabellen vom Typ `pf_page_table`, die wiederum jeweils bis zu 1024 Seiten-Deskriptoren (Typ `pf_page_desc`) enthalten.

13a ⟨*Konstanten* 13a⟩≡ (10) 16b▷

```
#define MAX_ADDR_SPACES    1024
#define MAX_TABLE_ENTRIES  1024
```

Defines:
`MAX_ADDR_SPACES`, used in chunk 14b.
`MAX_TABLE_ENTRIES`, used in chunks 15a, 16a, 19c, 32b, 39c, 50, and 51a.

Jeder Prozess erhält seinen eigenen Adressraum (`pf_address_space`). Ulix verwaltet alle Adressräume in einem Array, dessen erster Eintrag (Nr. 0) für den Kernel verwendet wird. Es können folglich maximal 1023 Prozesse parallel laufen. Pro Prozess stehen $1024 \times 1024 = 1048576$ Seiten – also bei einer Seitengröße von 4 KByte – 4 GByte virtueller Speicher zur Verfügung. Abzüglich des in alle Adressräume gemappten Kernel-Speicherbereichs (ab Adresse 0xBFC00000) bleiben jedem Prozess noch ca. 3 Gigabyte zur Verfügung. Der Zweck dieses Mappings wird in Kapitel 3.4 erläutert.

Um den Quellcode übersichtlicher gestalten zu können, wurden für die Bezeichner der elementaren Datentypen `unsigned int`, `unsigned short`, `unsigned char` kürzere Synonyme definiert:

13b ⟨*Typdefinitionen* 13b⟩≡ (10) 14a▷

```
typedef unsigned int    uint;
typedef unsigned short  ushort;
```

```
typedef unsigned char   uchar;
```

Defines:
uchar, used in chunks 21, 25–29, 33a, 36b, 38c, and 53a.
uint, used in chunks 14, 15b, 19–21, 30–35, 37a, 39a, 47, 48, and 51b.
ushort, used in chunks 21b and 33.

Der Adressraum (pf_address_space) eines Prozesses definiert sich durch die nachfolgend beschriebenen Eigenschaften. Ein Teilelement ist die Adresse des Page Directorys, welche beim Kontextwechsel durch den Scheduler in das Control Register 3 (CR3) geladen wird. Dadurch erhält die MMU, wenn es eine virtuelle Adresse in eine physische umzuwandeln gilt, Zugriff auf das Page Directory des derzeit aktiven Prozesses. Des Weiteren sind in dem Struct u.a. die Prozess-ID und ein Flag hinterlegt, das angibt, ob der Adressraum derzeit genutzt wird:

14a ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁13b 14c▷

```
typedef struct {
  void* pd;        // Virtuelle Adresse des page directory
  uint  physical;  // Physische Adresse des page directory
  int   pid;       // Prozess-ID
  uint  free;      // Flag, ob address space genutzt wird
  uint  memstart;  // Erste Adresse unter 0xC000.0000
  uint  memend;    // Letzte Adresse unter 0xC000.0000
}pf_address_space;
```

Defines:
pf_address_space, used in chunk 14b.
Uses uint 13b.

Das Array mit den Adressräumen ist als globale Variable in der Datei ulix.c definiert und daher vor der Verwendung in der Headerdatei als externe Variable einzubinden.

14b ⟨*externe Variablen* 14b⟩≡ (10) 36a▷

```
extern pf_address_space address_spaces[MAX_ADDR_SPACES];
```

Defines:
address_spaces, used in chunks 18, 46b, and 54.
Uses MAX_ADDR_SPACES 13a and pf_address_space 14a.

Die Page Directories, auf die vom Array address_spaces verwiesen wird, stellen ebenfalls Arrays dar. Die Einträge dieser Arrays sind die Seitentabellen-Deskriptoren vom Typ pf_page_table_desc. Wie nachfolgend aufgeführt, wird in diesen Deskriptoren neben diversen Informationen zum Status einer Seitentabelle mit frame_addr auch der Verweis zur Seitentabelle hinterlegt.

14c ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁14a 15a▷

```
typedef struct {              // Bit-Nr:
  uint present         :  1; //  0     --> im Hauptspeicher
  uint writeable       :  1; //  1     --> beschreibbar
  uint user_accessible :  1; //  2     --> im User-Mode erreichbar
  uint pwt             :  1; //  3     --> nicht verwendet
  uint pcd             :  1; //  4     --> nicht verwendet
  uint accessed        :  1; //  5     --> durch CPU gesetzt
```

```
    uint undocumented    :  1; //  6     --> nicht verwendet
    uint zeroes          :  2; //  8.. 7 --> nicht verwendet
    uint unused_bits     :  3; // 11.. 9 --> nicht verwendet
    uint frame_addr      : 20; // 31..12 --> Verweis auf Seitentabelle
  }pf_page_table_desc;
```

Defines:
pf_page_table_desc, used in chunks 15a, 18, 19, 39c, 47, and 54.
Uses uint 13b.

Das Array pf_page_directory, das diese Seitentabellen-Deskriptoren beinhaltet, definiert sich wie folgt:

15a ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁14c 15b▷

```
  typedef struct {
    pf_page_table_desc ptds[MAX_TABLE_ENTRIES];
  }pf_page_directory;
```

Defines:
pf_page_directory, used in chunk 18.
Uses MAX_TABLE_ENTRIES 13a and pf_page_table_desc 14c.

Der Seitentabellen-Deskriptor ist ein 32 Bit langes Struct, dessen Elemente in einzelne Bits untergliedert sind. Neben nicht verwendeten bzw. undokumentierten Bits sind Status-Bits enthalten, die angeben, ob die Seitentabelle derzeit ausgelagert, schreibgeschützt, durch User-Mode-Anwendungen erreichbar oder auslagerbar ist. Darüber hinaus markiert die MMU über ein accessed-Bit, ob auf diese Seitentabelle zugegriffen wurde [Int87, S. 100 ff.]. Der Zweck dieses Bits wird im Kapitel 3.8.1 näher beschrieben. Das Teilelement frame_addr dieses Structs enthält die ersten 20 Bit zur Startadresse der dazugehörigen Seitentabelle.

Die Seitentabelle ist, ähnlich wie address_spaces oder pf_page_directory, ein Array mit 1024 Einträgen. Die Einträge sind hier vom Typ pf_page_desc und stellen die Seiten-Deskriptoren dar, deren Aufbau stark dem der Seitentabellen-Deskriptoren ähnelt:

15b ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁15a 16a▷

```
  typedef struct{              // Bit-Nr:
    uint present         :  1; //  0     --> im Hauptspeicher
    uint writeable       :  1; //  1     --> beschreibbar
    uint user_accessible :  1; //  2     --> Im User-Mode erreichbar
    uint pwt             :  1; //  3     --> nicht verwendet
    uint pcd             :  1; //  4     --> nicht verwendet
    uint accessed        :  1; //  5     --> durch CPU gesetzt
    uint dirty           :  1; //  6     --> durch CPU gesetzt
    uint zeroes          :  2; //  8.. 7 --> sind immer null
    uint paged_out       :  1; //  9     --> NEU: Auslagerungszustand
    uint unused_bits     :  2; // 11..10 --> nicht verwendet
    uint frame_addr      : 20; // 31..12 --> Verweis auf Rahmen
  }pf_page_desc;
```

Defines:
pf_page_desc, used in chunks 16a, 18, 19, 33a, 36b, 39c, 47, and 54.
Uses uint 13b.

Damit der Page Fault Handler (siehe Kapitel 3.7) bei gesetztem `present`-Bit erkennen kann, ob eine Seite ausgelagert ist, wird eines der `unused_bits` zum Indikator `paged_out` umfunktioniert. Gem. [Int87, S. 100] können diese Bits vom Entwickler anderweitig verwendet werden. Ein weiterer Unterschied zum Seitentabellen-Deskriptor ist das `dirty`-Bit. In diesem Bit wird durch die CPU festgehalten, ob sich der Inhalt einer Seite verändert hat.

Das Array mit diesen Seiten-Deskriptoren definiert sich wie nachfolgend dargestellt.

16a ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁15b 21b▷

```
typedef struct {
  pf_page_desc pds[MAX_TABLE_ENTRIES];
}pf_page_table;
```

Defines:
`pf_page_table`, used in chunk 18b.
Uses `MAX_TABLE_ENTRIES` 13a and `pf_page_desc` 15b.

Zur Veranschaulichung skizziert Abbildung 3.3 den Aufbau der beiden Deskriptoren-Typen.

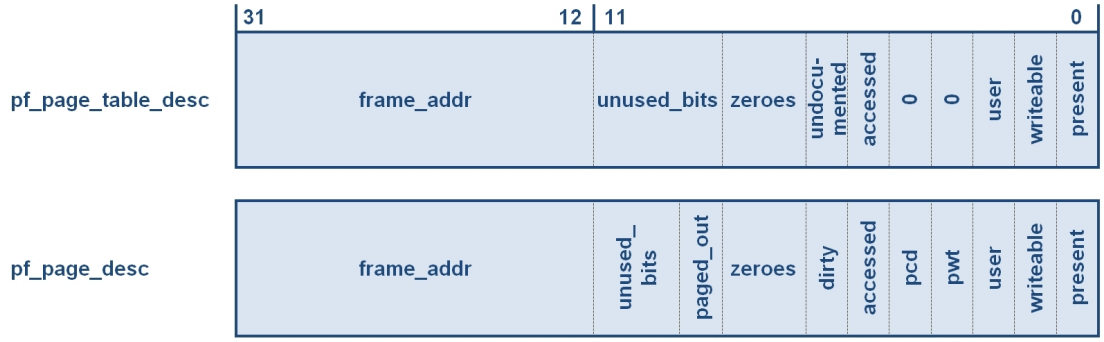

Abbildung 3.3: In Anlehnung an [Int87, S. 100]: Structs `pf_page_table_desc` und `pf_page_desc`.

Das nachfolgende Beispiel soll den Aufbau der Seitentabellenstruktur und die Übersetzung einer virtuellen Adresse in eine Physische Adresse erklären:

Gegeben ist die virtuelle Adresse 0x30D4400 (hex). Diese soll durch die MMU in eine physische Adresse übersetzt werden. Wie in Abbildung 3.4 dargestellt, wird die virtuelle Adresse in drei Bereiche untergliedert. Der linke Bereich (Bit 31 – 22) definiert den Index im Page Directory und der mittlere Bereich (Bit 21 – 12) gibt den Index in der Seitentabelle wieder. Der Offset (Bit 11 – 0) bildet die Adresse innerhalb der Seite. Er ist sowohl in der Seite, als auch im Seitenrahmen gleich und kann daher in der virtuellen und in der physischen Adresse verwendet werden.

Eine Seite hat in Ulix genau 4096 (hexadezimal: 0x1000) mögliche Adressen:

16b ⟨*Konstanten* 13a⟩+≡ (10) ◁13a 19a▷

```
#define PAGE_SIZE  4096
```

Defines:
`PAGE_SIZE`, used in chunks 19c, 21c, 32b, 33a, 36b, 39c, 50, 51a, and 59.

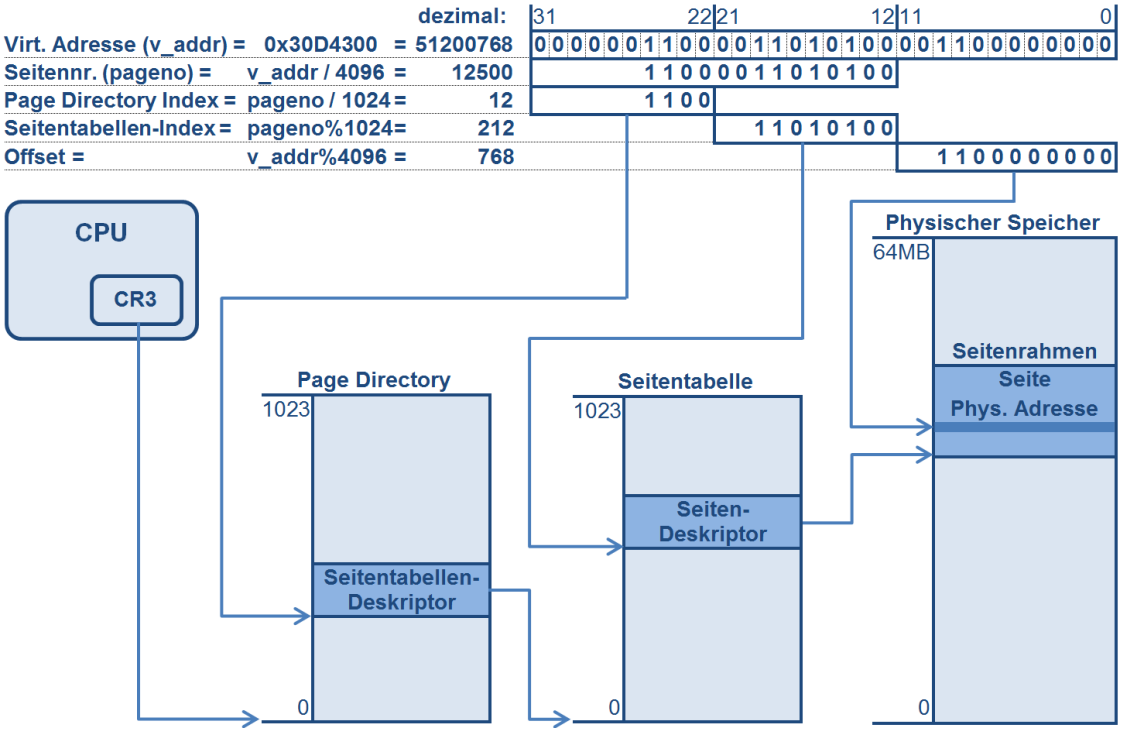

Abbildung 3.4: In Anlehnung an [Tan09, S. 243]: Virtueller Adressraum

Folglich lässt sich die Seitennummer durch Dividieren ermitteln:

Seitennummer = 0x30D4400 / 0x1000 = 0x30D4 bzw. 12500 Rest 768

Der Rest (0x30D4400 *mod* 0x1000) bildet den erwähnten Offset. Im Bezug auf die Seitentabellenstruktur mit max. 1024 Einträgen pro Tabelle lässt sich der Eintrag mit der betreffenden Seitentabelle im Page Directory berechnen, indem die ermittelte Seitennummer durch die maximale Anzahl der Einträge im Page Directory dividiert wird:

Index im Page Directory = 12500 / 1024 = 12 Rest 212

Über den Rest (12500 *mod* 1024) lässt sich der Index in der Seitentabelle errechnen.

Die Übersetzung der virtuellen Adresse in eine physische wird auf folgende Weise durchgeführt: Zuerst liest die MMU aus dem Control Register 3 der CPU die Page Directory-Adresse des aktuellen Prozesses und zieht den ermittelten Index im Directory hinzu, um auf den korrekten Eintrag im Page Directory zuzugreifen. Der Eintrag vom Typ `pf_page_table_desc` verweist in seinem Teilelement `frame_addr` auf die Startadresse der gesuchten Seitentabelle. Mit dem berechneten Seitentabellen-Index kann auch der betreffende Eintrag in der Seitentabelle vom Typ `pf_page_desc` ausgelesen werden. Dieser Seiten-Deskriptor beinhaltet in `frame_addr` die physische Adresse des gesuchten Seitenrahmens im Hauptspeicher. Zusammen mit dem Offset bildet sie schließlich die vollständige physische Adresse der Speicherstelle, die der

virtuellen Adresse zugeordnet ist. Ändert sich durch Aus- und Wiedereinlagern die physische Adresse, so wird sie im Seiten-Deskriptor eingetragen, und die virtuelle Adresse, die der Prozess sieht bleibt unverändert. Die im nachfolgenden Kapitel beschriebenen Funktionen setzen diese Systematik um.

3.4 Ausgabe und Bearbeitung der Seiten(tabellen)-Deskriptoren

Um die Seiten-Deskriptoren im Rahmen der Ein- und Auslagerung verändern zu können, werden Funktionen benötigt, die – ähnlich der MMU – den betreffenden Seitentabellen-Deskriptor bzw. Seiten-Deskriptor ermitteln. Diese Aufgabe übernehmen die Funktionen `get_ptd` und `get_pdesc`. Beiden Funktionen werden die berechneten Indizes von Page Directory und Seitentabelle sowie der betreffende Adressraum übergeben. Daraufhin werden Seitentabellen-Deskriptor bzw. Seiten-Deskriptor anhand der im Kapitel 3.3 beschriebenen Systematik ermittelt. Der Parameter Adressraum (`as`) wird später in der Speicherüberwachung (in Kapitel 3.8.2 beschrieben) und im „Not-Recently-Used“-Algorithmus (NRU) (in Kapitel 3.8.1 beschrieben) benötigt, da an diesen Stellen die Seitentabellenstrukturen aller Prozesse durchsucht werden.

18a ⟨*get_ptd* 18a⟩≡ (9)

```
pf_page_table_desc* get_ptd(int as, int pdindex) {
  pf_page_directory* pd;    //page_directory
  pf_page_table_desc* ptd;  //page_directory-Eintrag: page_table_desc

  pd = address_spaces[as].pd;
  ptd = (pf_page_table_desc*) &pd->ptds[pdindex];

  return ptd;
}
```

Uses `address_spaces` 14b, `get_ptd` 19b, `pf_page_directory` 15a, and `pf_page_table_desc` 14c.

18b ⟨*get_pdesc* 18b⟩≡ (9)

```
pf_page_desc* get_pdesc(int as, int pdindex, int ptindex) {
  pf_page_directory* pd;    //page_directory
  pf_page_table_desc* ptd;  //page_directory-Eintrag: page_table_desc
  pf_page_table* pt;        //page_table
  pf_page_desc* pdesc;      //page_table-Eintrag: page_desc

  pd = address_spaces[as].pd;
  ptd = (pf_page_table_desc*) &pd->ptds[pdindex];
  pt = (pf_page_table*) PHYSICAL(ptd->frame_addr << 12);
  pdesc = (pf_page_desc*) &pt->pds[ptindex];

  return pdesc;
}
```

Uses `address_spaces` 14b, `get_pdesc` 19b, `pf_page_desc` 15b, `pf_page_directory` 15a, `pf_page_table` 16a, `pf_page_table_desc` 14c, and `PHYSICAL` 19a.

Wie schon in Kapitel 3.3 erwähnt, sind im Seitentabellen-Deskriptor die ersten 20 Bit der 32 Bit langen Seitentabellen-Adresse im Element `frame_addr` hinterlegt. Um auf die Seitentabelle zugreifen zu können, ist der Rest mit Nullen zu füllen, weshalb ein Bitshift von 12 Stellen vorgenommen wird. Es besteht jedoch die Problematik, dass alle Adressangaben innerhalb der Seitentabellenstruktur auf physische Adressen zeigen. Da selbst der Kernel bei aktiviertem Paging nicht direkt auf den physischen Speicher zugreifen kann, wird in Ulix der gesamte physische Speicherbereich der Seitentabellenstruktur im Kernel in den virtuellen Adressraum (beginnend bei Adresse 0xD0000000) gemappt. Dies hat den Vorteil, dass durch Addition aus der physischen eine virtuelle Adresse wird, auf die dann zugegriffen werden kann; und genau diese Addition erledigt das Makro `PHYSICAL`:

19a ⟨*Konstanten* 13a⟩+≡ (10) ◁16b 21e▷

```
#define PHYSICAL(x) ((x)+0xD0000000)
```

Defines:
`PHYSICAL`, used in chunks 18b, 33a, and 36b.

Bevor die beiden Funktionen `get_ptd` und `get_pdesc` jedoch verwendet werden können, ist deren Deklaration in der Header-Datei `module.h` erforderlich:

19b ⟨*Prototypen* 19b⟩≡ (10) 20a▷

```
pf_page_table_desc* get_ptd(int as, int pdindex);
pf_page_desc* get_pdesc(int as, int pdindex, int ptindex);
```

Defines:
`get_pdesc`, used in chunks 18b, 19c, 33a, 36b, 39c, 46b, and 54.
`get_ptd`, used in chunks 18a, 19c, 39c, 46b, and 54.
Uses `pf_page_desc` 15b and `pf_page_table_desc` 14c.

Zu Testzwecken sowie zur Fehlersuche wird die Funktion `print_pdesc` implementiert. Die Funktion veranschaulicht die Verwendung der Funktionen `get_ptd` und `get_pdesc` zum Auslesen der in den Deskriptoren gespeicherten Werte.

19c ⟨*print_pdesc* 19c⟩≡ (9)

```
void print_pdesc(int as, uint v_address) {
  uint pageno = v_address/PAGE_SIZE;
  uint pdindex = pageno/MAX_TABLE_ENTRIES;
  uint ptindex = pageno%MAX_TABLE_ENTRIES;

  pf_page_table_desc* ptd = get_ptd(as, pdindex);
  pf_page_desc* pdesc = get_pdesc(as, pdindex, ptindex);

  printf("as: %d, pageno: %d, pdindex: %d, ptindex = %d\n",
    as, pageno, pdindex, ptindex);

  printf("ptd->frame_addr = %08x\n", ptd->frame_addr);

  printf("ptd  [%04d]: pres %d,wr %d,user_acc %d,"
    "pwt %d,pcd %d,acc %d\n",
    pdindex, ptd->present, ptd->writeable,
    ptd->user_accessible, ptd->pwt, ptd->pcd,ptd->accessed);
```

```
    printf("pdesc[%04d]: pres %d,wr %d,user_acc %d,"
      "pwt %d,pcd %d,acc %d,po %d,drty %d\n",
      ptindex, pdesc->present, pdesc->writeable,
      pdesc->user_accessible, pdesc->pwt, pdesc->pcd,
      pdesc->accessed, pdesc->paged_out, pdesc->dirty);

    return;
  }
```

Defines:
print_pdesc, used in chunks 40b and 56e.
Uses get_pdesc 19b, get_ptd 19b, MAX_TABLE_ENTRIES 13a, PAGE_SIZE 16b, pf_page_desc 15b, pf_page_table_desc 14c, printf 20b, and uint 13b.

Zu diesem Zweck werden zuerst die Seitennummer und die Tabellen-Indizes ermittelt und anschließend den genannten Funktionen übergeben. Danach erfolgt die Ausgabe der Werte durch die Funktion `printf`. Vor der ersten Verwendung dieser Funktion hat die Deklaration des Prototyps `print_pdesc` sowie die Einbindung von `printf` in der Header-Datei `module.h` zu erfolgen:

20a ⟨*Prototypen* 19b⟩+≡ (10) ◁19b 23b▷

```
    void print_pdesc(int as, uint v_address);
```

Defines:
print_pdesc, used in chunks 40b and 56e.
Uses uint 13b.

20b ⟨*externe Prototypen* 20b⟩≡ (10) 22▷

```
    // printf.c
    extern int printf(const char *format, ...);
```

Defines:
printf, used in chunks 19c, 24a, 40–43, 50, 51b, 53c, 56, 57, 61, 63d, and 64.

Nachdem der Zugriff auf die Seitentabellenstruktur mit den genannten Funktionen gewährleistet ist, wird in den nächsten Kapiteln auf die Implementierung der Auslagerungsdateien eingegangen.

3.5 Auslagerungsdateien

Um Seiten auslagern zu können, wird zuerst Platz auf der Festplatte benötigt. Grundsätzlich sind zum Speichern aktuell nicht benötigter Seiten neben Auslagerungspartitionen und separaten Festplatten u.a. Auslagerungsdateien denkbar. Partitionen und separate Festplatten haben den Vorteil, dass ein spezielles Dateisystem verwendet werden kann, welches nicht den Overhead eines konventionellen Dateisystems besitzt und dadurch effizienter genutzt werden kann (in UNIX-Systemen geläufig). Auslagerungsdateien hingegen werden zwar innerhalb konventioneller Dateisysteme verwendet, haben jedoch den Vorteil der variablen Dateigröße (in Windows-Systemen geläufig) [Tan09, S. 286 ff.]. Da die Implementierung eines speziellen Dateisystems für die auszulagernden Dateien über den Rahmen dieser Arbeit

hinausgehen würde, werden im Zuge der Ulix-Implementierung zwei Auslagerungsdateien verwendet:

21a ⟨*globale Variablen* 21a⟩≡ (10) 21d▷
```
char* indexfile = "/var/pindex";
char* blockfile = "/var/bfile";
```
Defines:
`blockfile`, used in chunks 24b and 26b.
`indexfile`, used in chunks 24b and 26b.

In der Indexdatei `indexfile` werden pro ausgelagerte Seite die Adressraum-Nummer des jeweiligen Prozesses, die Seitennummer innerhalb des Adressraums sowie der Index des in der Blockdatei `blockfile` abgelegten Seiteninhalts. Die Einträge in den Dateien haben somit folgendes Format:

21b ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁16a 21c▷
```
// Index-Eintrag
typedef struct{
  ushort as_no;
  uint   page_no;
  uint   block_no;
}index_entry;
```
Defines:
`index_entry`, used in chunks 27, 29, 33a, and 36b.
Uses `uint` 13b and `ushort` 13b.

Die Blockdatei `blockfile` hingegen besteht nur aus den Inhalten der ausgelagerten Seiten.

21c ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁21b 37a▷
```
// Block-Eintrag
typedef struct{
  uchar  load[PAGE_SIZE];
}out_page;
```
Defines:
`out_page`, used in chunks 27a, 33a, and 36b.
Uses `PAGE_SIZE` 16b and `uchar` 13b.

Über die Nummer des Adressraums und der betreffenden Seite in der Datei `indexfile` wird somit die Position der Seite in der Datei `blockfile` ermittelt. Darüber hinaus werden die Einträge in der Blockdatei über eine im Hauptspeicher liegende Bitmap abgebildet. In dieser Bitmap `usedblocklist` werden alle aktiven Einträge der Blockdatei als „belegt“ registriert.

21d ⟨*globale Variablen* 21a⟩+≡ (10) ◁21a 38c▷
```
uchar usedblocklist[UBL_SIZE];
```
Uses `uchar` 13b.

21e ⟨*Konstanten* 13a⟩+≡ (10) ◁19a 23a▷
```
#define UBL_SIZE     128
#define UBL_ENTRIES  UBL_SIZE * 8
```

Dies erleichtert und beschleunigt später die Suche nach freien Einträgen in der Blockdatei (siehe Seite 30), da hierfür die Einträge in der Auslagerungsdatei nicht gescannt werden müssen. Über die Konstante `UBL_SIZE` lässt sich die Größe der Bitmap festlegen. Derzeit stehen dem System – bei einem Character-Array mit 128 Einträgen – $8 * 128 = 1024$ Bits zur Verfügung (siehe `UBL_ENTRIES`). Es wird somit davon ausgegangen, dass die Auslagerungsdatei `blockfile` später nicht mehr als 1024 Seiten beinhaltet. Abbildung 3.5 soll den Aufbau der Auslagerungsdateien anhand eines Beispiels veranschaulichen.

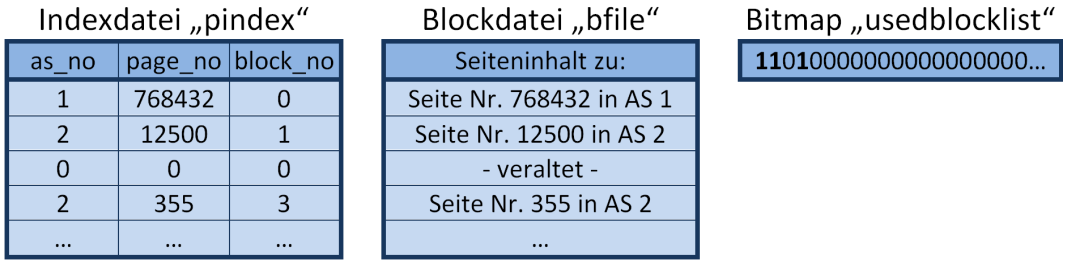

Abbildung 3.5: Aufbau der Auslagerungsdateien

Wie im Beispiel der Abbildung 3.5 dargestellt, enthält die Datei `indexfile` derzeit drei aktive Einträge zu Seiten der Adressräume 1 und 2. Der dritte Eintrag in dieser Datei wurde inzwischen gelöscht, also mit Nullen überschrieben. Passend dazu enthält die Blockdatei den Inhalt dieser ausgelagerten Seiten. Um später die Suche nach freien Plätzen in dieser Datei zu beschleunigen, wird die Belegung der Datei in der Bitmap `usedblocklist` festgehalten. Dies hat zudem den Vorteil, dass veraltete Einträge in der Blockdatei nicht mit Nullen überschrieben werden müssen.

Um die Auslagerungsdateien `indexfile` und `blockfile` erstellen, lesen und befüllen zu können, muss ein Zugriff auf das Dateisystem von Ulix erfolgen. In den folgenden Funktionen `chkfile`, `rblock` und `wblock` wird dies unter Verwendung der vom Dateisystem bereitgestellten Schnittstellen ermöglicht.

22 ⟨*externe Prototypen* 20b⟩+≡ (10) ◁20b 24a▷

```
// ulix.c
extern int mx_open(int device, const char *path, int oflag);
extern int mx_lseek(int device, int mfd, int offset, int whence);
extern int mx_write(int device, int mfd, void *buf, int nbyte);
extern int mx_read(int device, int mfd, void *buf, int nbyte);
extern int mx_unlink(int device, const char *path);
extern int mx_close(int device, int mfd);
```

Defines:
`mx_close`, used in chunks 23c, 25c, and 26a.
`mx_lseek`, used in chunks 25c and 26a.
`mx_open`, used in chunks 23c, 25c, and 26a.
`mx_read`, used in chunk 25c.
`mx_unlink`, used in chunk 24b.
`mx_write`, used in chunk 26a.

Die Ein- und Ausgabe-Funktionen mit dem Präfix `mx_` ähneln in ihrer Arbeitsweise den bekannten C-Funktionen `open`, `lseek` usw. jedoch mit dem Unterschied, dass

der erste Übergabeparameter das Laufwerk definiert. Bei ihrer Verwendung werden diverse Platzhalter und Bitmasken herangezogen, die nachfolgend deklariert sind:

23a ⟨*Konstanten* 13a⟩+≡ (10) ◁21e 27a▷

```
// aus ulix.c
#define DEV_HDA     0x300     // /dev/hda
#define O_RDONLY    0x0000    // nur lesend
#define O_WRONLY    0x0001    // nur schreibend
#define O_CREAT     0x0200    // Datei erstellen
#define SEEK_SET    0         // absoluter offset
```

Defines:
DEV_HDA, used in chunks 23–26.
O_CREAT, used in chunks 23c, 25c, and 26a.
O_RDONLY, used in chunk 25c.
O_WRONLY, used in chunks 23c and 26a.
SEEK_SET, used in chunks 25c and 26a.

Damit dem Paging-Mechanismus die Auslagerungsdateien zur Verfügung stehen, ist es nötig, eine Möglichkeit zur Erstellung dieser Dateien zu schaffen. Dies bewerkstelligt die Funktion `chkfile`.

23b ⟨*Prototypen* 19b⟩+≡ (10) ◁20a 24c▷

```
int chkfile(char* file);
```

Defines:
chkfile, used in chunk 24b.

Über den Parameter `file` erhält diese Funktion den Dateipfad der jeweiligen Auslagerungsdatei. Zunächst versucht `chkfile` die Datei unter dem angegebenen Pfad mit `mx_open` zu öffnen. Sollte die Datei unter diesem Pfad nicht vorhanden sein, legt die Funktion sie neu an. Realisiert wird dies durch den Übergabeparameter `O_CREAT`, der `mx_open` mitgegeben wird.

23c ⟨*chkfile* 23c⟩≡ (9)

```
int chkfile(char* file) {
  int fd;

  debug_printf ("chkfile(%s)\n", file);

  if((fd = mx_open(DEV_HDA, file, (O_WRONLY | O_CREAT))) == -1) {
    return 0;
  }else{
    mx_close(DEV_HDA, fd);
    return 1;
  }
}
```

Defines:
chkfile, used in chunk 24b.
Uses debug_printf 24a, DEV_HDA 23a, mx_close 22, mx_open 22, O_CREAT 23a, and O_WRONLY 23a.

Die externe Funktion `debug_printf` bewirkt eine Ausgabe durch den Emulator auf der Konsole und wird im Zuge dieser Arbeit zu Testzwecken bzw. zur Fehlersuche

angewandt. Damit die Funktion verwendet werden kann, ist sie in der Header-Datei `module.h` einzubinden.

24a ⟨*externe Prototypen* 20b⟩+≡ (10) ◁22 28c▷

```
// printf.c
extern int debug_printf(const char *format, ...);
```

Defines:
`debug_printf`, used in chunks 23c, 26–29, 33a, and 34c.
Uses `printf` 20b.

`chkfile` wird beim Ulix-Start über die Funktion `initialize_module` für die beiden Dateien `indexfile` und `blockfile` ausgeführt. So wird sichergestellt, dass die Auslagerungsdateien vorhanden sind, falls eine Seiten-Auslagerung erforderlich ist.

24b ⟨*initialize_module* 24b⟩≡ (9) 38d▷

```
void initialize_module() {

  mx_unlink(DEV_HDA, indexfile);
  mx_unlink(DEV_HDA, blockfile);

  chkfile(indexfile);
  chkfile(blockfile);
```

Defines:
`initialize_module`, used in chunks 24d and 25a.
Uses `blockfile` 21a, `chkfile` 23b 23c, `DEV_HDA` 23a, `indexfile` 21a, and `mx_unlink` 22.

Beim Paging kann es u.a. durch Systemabstürze vorkommen, dass ausgelagerte Seiten nicht mehr zurück in den Hauptspeicher geschrieben werden. Folglich sind die Auslagerungsdateien beim nächsten Systemstart evtl. noch mit alten und ungültigen Einträgen gefüllt, was zu Fehlfunktionen führen könnte. Deswegen werden bei jedem Systemstart die Auslagerungsdateien mit `mx_unlink` gelöscht und dann über `chkfile` neu angelegt. Theoretisch lässt sich vor dem Löschen prüfen, ob dies nötig ist. Jedoch funktioniert die gewöhnliche Größenbestimmung von Dateien in Ulix derzeit noch nicht. Man müsste somit die Datei öffnen und mittels `mx_lseek` ihre Größe auslesen. So zieht eine vorherige Prüfung jedoch mehr Zugriffsoperationen nach sich als das bloße Löschen und Anlegen ohne Prüfung.

Damit die Funktion `initialize_module` beim Systemstart von Ulix ausgeführt wird, ist zunächst ihre Deklaration in der Header-Datei `module.h` sowie deren Einbindung in `ulix.c` erforderlich:

24c ⟨*Prototypen* 19b⟩+≡ (10) ◁23b 25b▷

```
void initialize_module();
```

Defines:
`initialize_module`, used in chunks 24d and 25a.

24d ⟨*ulix.c - function prototypes - initialize_module* 24d⟩≡

```
extern void initialize_module();
```

Uses `initialize_module` 24b 24c.

Anschließend ist die `main`-Routine in `ulix.c` um folgenden Eintrag zu erweitern, damit die Funktion beim Systemstart aufgerufen wird:

25a ⟨*ulix.c - kernel main* 25a⟩≡
```
initialize_module();
```
Uses `initialize_module` 24b 24c.

Nachdem die Auslagerungsdateien erstellt werden können, benötigt der Paging-Mechanismus auch die Möglichkeit, die Dateien zu lesen und zu beschreiben. Dies bewerkstelligen die zwei Funktionen `rblock` und `wblock`.

25b ⟨*Prototypen* 19b⟩+≡ (10) ◁24c 27b▷
```
int rblock(int type, int indexno, uchar* block);
int wblock(int type, int indexno, uchar* block);
```
Defines:
`rblock`, used in chunks 26b, 27c, 29, and 36b.
`wblock`, used in chunks 26b, 28b, and 33a.
Uses `uchar` 13b.

Mithilfe von `rblock` erfolgt der Lesezugriff auf die Auslagerungsdateien. `rblock` erhält hierfür einen Zeiger auf einen Datenblock, den Typ des Datenblocks sowie dessen Index in der jeweiligen Auslagerungsdatei. In Abhängigkeit des übergebenen Typs wird entweder ein Datenblock aus der Indexdatei oder der Blockdatei gelesen und dem Zeiger zugewiesen.

25c ⟨*rblock* 25c⟩≡ (9)
```
int rblock(int type, int indexno, uchar* block) {
  int status, size, fd;
  char* file;

  ⟨Differenzierung des Auslagerungsdateityps 26b⟩

  fd = mx_open(DEV_HDA, file, O_RDONLY | O_CREAT);
  mx_lseek(DEV_HDA, fd, (indexno * size), SEEK_SET);
  status = mx_read(DEV_HDA, fd, block, size);
  mx_close(DEV_HDA, fd);
  return status;
}
```
Defines:
`rblock`, used in chunks 26b, 27c, 29, and 36b.
Uses `DEV_HDA` 23a, `mx_close` 22, `mx_lseek` 22, `mx_open` 22, `mx_read` 22, `O_CREAT` 23a, `O_RDONLY` 23a, `SEEK_SET` 23a, and `uchar` 13b.

Hierfür wird die Datei im read-only-Modus geöffnet und einem File-Deskriptor zugeordnet. Über diesen File-Deskriptor lässt sich mittels `mx_lseek` die Leseposition festlegen. Diese Leseposition ermittelt sich aus der Index-Nummer in der Datei multipliziert mit der Größe des Datenblocks. Die Rückgabewerte von `mx_read` und `mx_write` beinhalten die gelesene bzw. geschriebene Datenmenge oder aber einen Fehlerwert, falls der Lese-/Schreibvorgang nicht erfolgreich abgeschlossen wurde. Dieser Rückgabewert von `mx_read` wird der Variable `status` zugewiesen und

von `rblock` weitergereicht. Nach dem Lesevorgang wird die Datei bzw. der File-Deskriptor geschlossen.

Ähnlich verhält sich auch `wblock`. Hier wird, je nach übergebenem Typ, der entsprechende Block in die vorgegebene Datei geschrieben.

26a ⟨*wblock* 26a⟩≡ (9)

```
int wblock(int type, int indexno, uchar* block) {
  int status, size, fd;
  char* file;

  ⟨Differenzierung des Auslagerungsdateityps 26b⟩

  fd = mx_open(DEV_HDA, file, O_WRONLY | O_CREAT);
  mx_lseek(DEV_HDA, fd, (indexno * size), SEEK_SET);
  status = mx_write(DEV_HDA, fd, block, size);
  mx_close(DEV_HDA, fd);
  return status;
}
```

Defines:
`wblock`, used in chunks 26b, 28b, and 33a.
Uses `DEV_HDA` 23a, `mx_close` 22, `mx_lseek` 22, `mx_open` 22, `mx_write` 22, `O_CREAT` 23a, `O_WRONLY` 23a, `SEEK_SET` 23a, and `uchar` 13b.

Nachdem beide Funktionen sowohl für die Indexdatei als auch für die Blockdatei verwendet werden sollen, ist vor dem Lese-/Schreibzugriff zuerst festzulegen, auf welche Datei zugegriffen werden soll. Hierzu wird der Übergabeparameter `type` geprüft, damit der Name der Auslagerungsdatei und die Größe des zu lesenden bzw. zu schreibenden Datenblocks entsprechend gesetzt werden können.

26b ⟨*Differenzierung des Auslagerungsdateityps* 26b⟩≡ (25c 26a)

```
switch(type) {
  case ACC_INDEX:
    file = indexfile;
    size = INDEXSIZE;
    break;
  case ACC_OUT_PAGE:
    file = blockfile;
    size = OUT_PAGE_SIZE;
    break;
  default:
    debug_printf("wblock/rblock - Parameter type konnte "
      "nicht zugeordnet werden.\n");
    return 0;
    break;
}
```

Uses `ACC_INDEX` 27a, `ACC_OUT_PAGE` 27a, `blockfile` 21a, `debug_printf` 24a, `indexfile` 21a, `INDEXSIZE` 27a, `OUT_PAGE_SIZE` 27a, `rblock` 25b 25c, and `wblock` 25b 26a.

Um den Quellcode möglichst übersichtlich zu gestalten, werden Konstanten zur Differenzierung der Auslagerungsdateien und deren Einträge festgelegt. Die Ma-

kros `INDEXSIZE` und `OUT_PAGE_SIZE` definieren sich über die Größe der verwendeten Structs `index_entry` und `out_page`.

27a ⟨*Konstanten* 13a⟩+≡ (10) ◁23a 30b▷

```
#define ACC_INDEX      0
#define ACC_OUT_PAGE   1
#define INDEXSIZE      sizeof(index_entry)
#define OUT_PAGE_SIZE  sizeof(out_page)
```

Defines:
`ACC_INDEX`, used in chunks 26–29, 32c, 33a, and 36b.
`ACC_OUT_PAGE`, used in chunks 26b, 28d, 32c, 33a, and 36b.
`INDEXSIZE`, used in chunks 26b and 28b.
`OUT_PAGE_SIZE`, used in chunk 26b.
Uses `index_entry` 21b 21b and `out_page` 21c 21c.

Für die Verwirklichung eines Auslagerungsmechanismus müssen jedoch, neben dem Schreiben und Lesen von Einträgen in den genannten Auslagerungsdateien auch Einträge wiedergefunden werden. Hierzu wird, aufbauend auf `rblock`, die Funktion `findblock` erstellt. Übergabeparameter sind hierbei der Adressraum `as` und die Nummer `page_no` der zu suchenden Seite. Es wird die Variable `check` vom Typ eines Indexdatei-Eintrags angelegt, die iterativ mittels `rblock` befüllt wird. Somit können die Einträge in der Datei mit den Übergabeparametern verglichen werden.

27b ⟨*Prototypen* 19b⟩+≡ (10) ◁25b 28a▷

```
int findblock(int as, int page_no);
```

Defines:
`findblock`, used in chunk 36b.

27c ⟨*findblock* 27c⟩≡ (9)

```
int findblock(int as, int page_no) {
  int x = 0;
  index_entry check;

  while(rblock(ACC_INDEX, x, (uchar*)&check) > 0) {//EOF=0,ERR=-1
    debug_printf("findblock: Indexsuche nach as=%d, page_no=%d. "
      "Fund: as %d, page %d, blockindex %d\n",
      as, page_no, check.as_no, check.page_no, check.block_no);

    // Wenn der Eintrag gefunden wurde: Index zurückgeben.
    if ((check.as_no == as) && (check.page_no == page_no)) return x;
    x++;
  }

  // Eintrag nicht gefunden
  return -1;
}
```

Defines:
`findblock`, used in chunk 36b.
Uses `ACC_INDEX` 27a, `debug_printf` 24a, `index_entry` 21b 21b, `rblock` 25b 25c, and `uchar` 13b.

Als Abbruchparameter für die oben verwendete While-Schleife dient der Rückgabewert aus der Funktion `mx_read`, die von `rblock` weitergegeben wird. Ist

dieser Wert 0, so wurde das Ende der Datei erreicht. Bei einem Wert von -1 ist ein Fehler beim Lesen aufgetreten. Daher soll die Schleife so lange durchlaufen werden, bis der Rückgabewert ≤ 0 ist, oder bis der gesuchte Eintrag in der Datei gefunden ist.

Neben der Suche nach Einträgen ist es zudem erforderlich, neue Einträge am Ende der Datei bzw. am nächsten freien Platz eintragen zu können. Durch das Wiedereinlagern von Seiten werden in den Auslagerungsdateien Einträge frei. Diese müssen daher als leer bzw. freigegeben gekennzeichnet und später auch wiedergefunden werden können, damit die Auslagerungsdateien nicht unnötig groß werden und zerklüften. Für die Indexdatei gewährleistet dies die Funktion `clearindex`.

28a ⟨*Prototypen* 19b⟩+≡ (10) ◁27b 31a▷
```
int clearindex(int indexno);
```
Defines:
`clearindex`, used in chunk 36c.

28b ⟨*clearindex* 28b⟩≡ (9)
```
int clearindex(int indexno) {
  uchar ib[INDEXSIZE];
  memset(ib, 0, INDEXSIZE);

  return wblock(ACC_INDEX, indexno, (uchar*) &ib);
}
```
Defines:
`clearindex`, used in chunk 36c.
Uses `ACC_INDEX` 27a, `INDEXSIZE` 27a, `memset` 28c, `uchar` 13b, and `wblock` 25b 26a.

Um einen Eintrag als leer zu markieren, soll er mit Nullen gefüllt werden. Hierfür wird ein Character-Array in der Größe des jeweiligen Eintrags angelegt. Anschließend füllt `memset` das Array mit Nullen. `memset` ist eine externe Funktion, die von Ulix bereitgestellt wird und daher vor der Verwendung zu deklarieren.

28c ⟨*externe Prototypen* 20b⟩+≡ (10) ◁24a 32d▷
```
// ulix.c
extern void *memset(void *dest, char val, int count);
```
Defines:
`memset`, used in chunks 28b, 57a, and 59.

Soll eine neue Seite in die Auslagerungsdateien geschrieben werden, so lässt sich mittels der Funktion `nextfreeblock` der Index des nächsten mit `clearindex` freigegebenen Eintrags ermitteln. Gibt es keinen freien Platz, so wird der Index des nächsten neuen Eintrags am Dateiende ermittelt. Da nicht mehr benötigte Einträge in der Indexdatei komplett mit Nullen überschrieben werden und in der Blockdatei über die gesonderte Bitmap `usedblocklist` gekennzeichnet sind, ist die Suche des nächsten freien Eintrags in zwei Funktionen untergliedert.

28d ⟨*nextfreeblock* 28d⟩≡ (9)
```
int nextfreeblock(int type) {
  int x;
  if(type == ACC_INDEX) x = nextfreeindexblock();
```

```
  if(type == ACC_OUT_PAGE) x = nextfreeoutpageblock();
  debug_printf("Naechster Freier %d-Platz = %d "
    "(0=Index 1=Block)\n", type, x);
  return x;
}
```

Defines:
nextfreeblock, used in chunk 32c.
Uses ACC_INDEX 27a, ACC_OUT_PAGE 27a, debug_printf 24a, nextfreeindexblock 29 31a, and nextfreeoutpageblock 30a 31a.

In der Funktion nextfreeindexblock werden die Datensätze unter Verwendung von rblock in die vorher angelegte Variable check eingelesen. Anschließend wird geprüft, ob die Nummer des Adressraums as_no auf null gesetzt ist. Der Wert von as_no kann nur bei „gelöschten“ Einträgen null sein, da Seiten des Adressraums 0 (= Adressraum des Kernels) in Ulix nicht ausgelagert werden. Es werden alle Einträge iterativ geprüft, bis entweder ein freier Block gefunden oder das Ende der Indexdatei erreicht ist. Ist dies der Fall, wird der nächste Index am Ende der Datei zurückgegeben. Sollte die Funktion rblock einen Fehlerwert zurückgeben, so wird dieser Wert ebenfalls an die aufrufende Funktion zurückgegeben.

29 ⟨*nextfreeindexblock* 29⟩≡ (9)

```
int nextfreeindexblock() {
  int x = 0;
  int status;
  index_entry check;
  debug_printf("Ermittle nächsten freien Index-Block\n");

  while((status = rblock(ACC_INDEX, x, (uchar*) &check)) > 0) {
    if(check.as_no == 0) return x;
    x++;
  }

  //Dateiende (EOF = 0) erreicht
  if(status == 0) return x;

  // bei Fehler
  return status;
}
```

Defines:
nextfreeindexblock, used in chunk 28d.
Uses ACC_INDEX 27a, debug_printf 24a, index_entry 21b 21b, rblock 25b 25c, and uchar 13b.

Die Funktion nextfreeoutpageblock hingegen ist anders strukturiert. Eine Suche nach dem oben beschriebenen Schema wäre hier ineffizient, da das Auslesen und Prüfen aller Einträge der Blockdatei aufgrund des Festplatten-Zugriffs viel Zeit beansprucht. Daher werden die Einträge dieser Datei über eine Bitmap im Hauptspeicher verwaltet. In einer Iteration werden die einzelnen Bits auf FALSE geprüft. Ein Bit mit dem Wert FALSE stellt eine freie Stelle in der Blockdatei dar. Sobald

ein solches Bit gefunden wurde, gibt die Funktion die Stelle des Bits in der Bitmap zurück. Sind alle Bits der Bitmap belegt, wird der Fehlerwert -1 zurückgegeben.

30a ⟨*nextfreeoutpageblock* 30a⟩≡ (9)

```
int nextfreeoutpageblock() {
  int cnt;

  for(cnt = 0; cnt < UBL_ENTRIES; cnt++) {
    if(get_usedblocklist_bit(cnt) == FALSE) {
      return cnt;
    }
  }

  // usedblocklist voll
  return -1;
}
```

Defines:
`nextfreeoutpageblock`, used in chunk 28d.
Uses `FALSE` 30b and `get_usedblocklist_bit` 31a.

`TRUE` und `FALSE` sind hierbei Konstanten, die verwendet werden, um die Code-Elemente verständlicher zu gestalten. Sie sind in der Header-Datei `module.h` definiert:

30b ⟨*Konstanten* 13a⟩+≡ (10) ◁27a 53d▷

```
#define TRUE  1
#define FALSE 0
```

Defines:
`FALSE`, used in chunks 30a, 34d, 36, 38d, 41–43, and 53.
`TRUE`, used in chunks 31b, 33–36, 38b, 42a, and 53c.

Zum Auslesen der Bitmap bedient sich `nextfreeoutpageblock` der Funktion `get_usedblocklist_bit`. Da die Bitmap in einem Character-Array gespeichert ist, gestaltet sich der Zugriff auf ein einzelnes Bit etwas komplexer. Zuerst wird der Index des Bits durch 8 dividiert, um den Index des Bytes zu erhalten, in dem das Bit liegt. Anschließend wird der Modulo-Wert des Bit-Index verwendet, um die Position des Bits innerhalb des Bytes zu ermitteln. Der Wert des Bits lässt sich dann entweder wie hier durch `%2` oder durch `&1` auslesen.

30c ⟨*get_usedblocklist_bit* 30c⟩≡ (9)

```
uint get_usedblocklist_bit(uint no) {
  uint byte = usedblocklist[no/8];

  return (byte >> (no % 8)) % 2;
}
```

Uses `get_usedblocklist_bit` 31a and `uint` 13b.

Zur Verwendung der beschriebenen Funktionen ist zuvor deren Deklaration in der Header-Datei `module.h` erforderlich:

31a ⟨*Prototypen* 19b⟩+≡ (10) ◁28a 32a▷

```
int nextfreeblock(int type);
int nextfreeindexblock();
int nextfreeoutpageblock();
uint get_usedblocklist_bit(uint no);
```

Defines:
`get_usedblocklist_bit`, used in chunk 30.
`nextfreeblock`, used in chunk 32c.
`nextfreeindexblock`, used in chunk 28d.
`nextfreeoutpageblock`, used in chunk 28d.
Uses `uint` 13b.

Somit werden alle erforderlichen Dateizugriffe, die im Rahmen des Ein- und Auslagerns von Seiten benötigt werden, abgedeckt.

3.6 Aus- und Einlagerungsmechanismus

Die Inhalte der vorangegangenen Kapitel bilden den Grundstein für das Auslagern (also das Lesen der Seite im Hauptspeicher und das anschließende Öffnen und Befüllen der Auslagerungsdateien) sowie für das Einlagern (also das Öffnen und Auslesen der Auslagerungsdateien und das Zurückschreiben der Seite in einen neuen Seitenrahmen im Hauptspeicher). In den nachfolgenden Kapiteln werden die Funktionen `frame_2_file` und `file_2_frame` vorgestellt, die sich der Aus- bzw. Einlagerung annehmen.

3.6.1 Auslagerung

Die Funktion `frame_2_file` ist für das Auslagern einer Seite zuständig. Als Übergabeparameter werden zum einen der Adressraum `as` und zum anderen die virtuelle Adresse `v_address` benötigt, um die Startadresse der betreffenden Seite im betreffenden Adressraum zu ermitteln. Der Aufbau dieser Funktion ist wie folgt strukturiert:

31b ⟨*frame_2_file* 31b⟩≡ (9)

```
int frame_2_file(int as, uint v_address) {
  ⟨Ermittlung Indizes aus Seitentabellen-Struktur 32b⟩
  ⟨frame_2_file - Ermittlung Indexeinträge in Auslagerungsdateien 32c⟩
  ⟨frame_2_file - Seitenrahmen zu Auslagerungsdateien 33a⟩
  ⟨frame_2_file - Frame freigeben und Cache leeren 34c⟩
  ⟨frame_2_file - Seitendeskriptor aktualisieren 34d⟩
  return TRUE;
}
```

Defines:
`frame_2_file`, used in chunks 33a, 50, 51b, and 56e.
Uses `TRUE` 30b and `uint` 13b.

Vor ihrer Verwendung ist der Prototyp der Funktion in der Header-Datei `module.h` zu definieren.

32a ⟨*Prototypen* 19b⟩+≡ (10) ◁31a 33b▷

```
int frame_2_file(int as, uint v_address);
```

Defines:
`frame_2_file`, used in chunks 33a, 50, 51b, and 56e.
Uses `uint` 13b.

Nach der im Kapitel 3.3 erläuterten Struktur der Seitentabellen lassen sich anhand der virtuellen Adresse die Seitennummer `pageno`, der Index im Page Directory `pdindex` und der Index in der Seitentabelle `ptindex` wie folgt berechnen:

32b ⟨*Ermittlung Indizes aus Seitentabellen-Struktur* 32b⟩≡ (31b 35a)

```
uint v_frame_address = v_address & 0xFFFFF000;
int pageno = v_address / PAGE_SIZE;
int pdindex = pageno / MAX_TABLE_ENTRIES;
int ptindex = pageno % MAX_TABLE_ENTRIES;
```

Uses `MAX_TABLE_ENTRIES` 13a, `PAGE_SIZE` 16b, and `uint` 13b.

Da im Folgenden die Seiten-Startadresse der jeweils übergebenen virtuellen Adresse benötigt wird, enthält die Variable `v_frame_address` die UND-Verknüpfung der virtuellen Adresse mit einer Maske, um den Offset (also die Adresse innerhalb der Seite) auf Null zu setzen. Nachdem diese Werte berechnet sind, werden mittels `nextfreeblock` die nächsten freien Plätze in den Auslagerungsdateien ermittelt und in den Variablen `pindex_index` und `bfile_index` gespeichert.

32c ⟨*frame_2_file - Ermittlung Indexeinträge in Auslagerungsdateien* 32c⟩≡ (31b)

```
int pindex_index = nextfreeblock(ACC_INDEX);
int bfile_index = nextfreeblock(ACC_OUT_PAGE);
```

Uses `ACC_INDEX` 27a, `ACC_OUT_PAGE` 27a, and `nextfreeblock` 28d 31a.

Anschließend werden die Daten, die in die Auslagerungsdateien geschrieben werden, vorbereitet. Hierfür sind die Variablen `entry` und `bfile_page` deklariert. Zuerst werden die Teilelemente von `entry` mit dem betreffenden Adressraum `as`, der berechneten Seitennummer `pageno` und dem zuvor ermittelten Index des nächsten freien Platzes in der Datei `blockfile` befüllt. Danach wird der Inhalt des Seitenrahmens im Hauptspeicher mittels `memcpy` in den Struct kopiert. `memcpy` ist vor der ersten Verwendung in der Header-Datei `module.h` zu deklarieren:

32d ⟨*externe Prototypen* 20b⟩+≡ (10) ◁28c 34a▷

```
// ulix.c
extern void *memcpy(void *dest, const void *src, int count);
```

Defines:
`memcpy`, used in chunks 33a and 36b.

Sind die Variablen mit den entsprechenden Werten versehen, werden sie mittels `wblock` in die Auslagerungsdateien geschrieben.

33a ⟨*frame_2_file - Seitenrahmen zu Auslagerungsdateien* 33a⟩≡ (31b)

```
index_entry entry;
out_page bfile_page;
pf_page_desc* pdesc = get_pdesc(as, pdindex, ptindex);

// Indexeintrag vorbereiten
entry.as_no = as;
entry.page_no = pageno;
entry.block_no = bfile_index;
debug_printf("frame_2_file: Index anlegen: AS %d, Page %d, "
  "pindex-Eintrag: %d, Blockindex %d\n",
  as, pageno, pindex_index, bfile_index);

// Block vorbereiten/füllen
memcpy((uchar*) &bfile_page.load,
       (uchar*) PHYSICAL(pdesc->frame_addr<<12),
       PAGE_SIZE);

// Schreiben
debug_printf("frame_2_file: Schreibe Index: %d\n",
  wblock(ACC_INDEX, pindex_index, (uchar*) &entry));
debug_printf("frame_2_file: Schreibe Block: %d\n",
  wblock(ACC_OUT_PAGE, bfile_index, (uchar*) &bfile_page));

// Bitmap Aktualisieren
set_usedblocklist_bit(bfile_index, TRUE);
```

Uses ACC_INDEX 27a, ACC_OUT_PAGE 27a, debug_printf 24a, frame_2_file 31b 32a, get_pdesc 19b, index_entry 21b 21b, memcpy 32d, out_page 21c 21c, PAGE_SIZE 16b, pf_page_desc 15b, PHYSICAL 19a, set_usedblocklist_bit 33b 33c, TRUE 30b, uchar 13b, and wblock 25b 26a.

Darüber hinaus wird mittels `set_usedblocklist_bit` die Bitmap für die Blockdatei aktualisiert. Diese Funktion ist vorab in der Headerdatei `module.h` zu deklarieren.

33b ⟨*Prototypen* 19b⟩+≡ (10) ◁32a 35b▷

```
void set_usedblocklist_bit(uint no, ushort value);
```

Defines:
set_usedblocklist_bit, used in chunks 33a and 36c.
Uses uint 13b and ushort 13b.

Als Übergabeparameter erhält `set_usedblocklist_bit` die Stelle und den Wert des betreffenden Bits. Abhängig davon, ob es sich dabei um eine 1 oder eine 0 handelt, wird die Stelle entweder mit einem normalen oder exklusiven ODER verknüpft, wodurch sich der Wert des Bits ändert.

33c ⟨*set_usedblocklist_bit* 33c⟩≡ (9)

```
void set_usedblocklist_bit(uint no, ushort value) {
  uint byte = usedblocklist[no/8];

  if(value == 0) {
    usedblocklist[no/8] = (byte ^ 1 << (no % 8));  // XOR
  } else {
```

```
      usedblocklist[no/8] = (byte | 1 << (no % 8));  // OR
    }

    return;
  }
```

Defines:
  `set_usedblocklist_bit`, used in chunks 33a and 36c.
Uses `uint` 13b and `ushort` 13b.

Wurde der Inhalt des Seitenrahmens in die Auslagerungsdateien geschrieben, muss dem System mitgeteilt werden, dass der Rahmen aktuell nicht in Verwendung ist und somit einer neuen Seite zugewiesen werden kann. Dies erfolgt über die Funktionen `release_frame` und `mmu`, die bereits implementiert sind und im Kernel-Mode zur Verfügung stehen.

34a ⟨*externe Prototypen* 20b⟩+≡ (10) ◁32d 34b▷

```
    // ulix.c
    extern void release_frame(uint frameaddr);
```

Defines:
  `release_frame`, used in chunk 34c.
Uses `uint` 13b.

34b ⟨*externe Prototypen* 20b⟩+≡ (10) ◁34a 35c▷

```
    // ulix.c
    extern uint mmu(int id, uint vaddress);
```

Defines:
  `mmu`, used in chunk 34c.
Uses `uint` 13b.

34c ⟨*frame_2_file - Frame freigeben und Cache leeren* 34c⟩≡ (31b)

```
    debug_printf("%d free frames", free_frames);
    release_frame(mmu(as, v_frame_address));
    debug_printf("%d free frames", free_frames);

    __asm__ __volatile__("invlpg %0"::"m"(*(char*)(pageno<<12)));
```

Uses `debug_printf` 24a, `free_frames` 36a, `mmu` 34b, and `release_frame` 34a.

Das Invalidieren des Caches erfolgt mittels Inline-Assembler. Dies ist nötig, da der Cache noch Informationen über den Speicherort von Daten im Hauptspeicher beinhalten kann. Wird er nicht invalidiert, greift die MMU trotz ausgelagerter Seite auf den ungültigen Inhalt im Cache zu.

Darüber hinaus ist es auch erforderlich, die Informationen über den Verbleib der ausgelagerten Seite in den Seiten-Deskriptoren zu aktualisieren. Hierzu wird durch `get_pdesc` der betreffende Seiten-Deskriptor dem Zeiger `pdesc` zugewiesen. Ist der Seiten-Deskriptor dann im Zugriff, lässt sich das `present`-Bit sowie das `paged_out`-Bit anpassen. So kann der Page Fault Handler bei einem Zugriff auf den virtuellen Speicherbereich sofort feststellen, dass die Seite ausgelagert ist und die Wiedereinlagerung anstoßen.

34d ⟨*frame_2_file - Seitendeskriptor aktualisieren* 34d⟩≡ (31b)

```
    pdesc->present = FALSE;
    pdesc->paged_out = TRUE;
```

Uses `FALSE` 30b and `TRUE` 30b.

Mit der Aktualisierung des Seiten-Deskriptors ist der Auslagerungsvorgang abgeschlossen und der physische Speicherplatz im Hauptspeicher steht einer neuen Seite zur Verfügung.

3.6.2 Einlagerung

Die Vorgehensweise des Wiedereinlagerns ähnelt der des Auslagerns, jedoch in umgekehrter Reihenfolge. So wird zuerst ein neuer Rahmen im Hauptspeicher angefordert. Die physische Adresse des Rahmens wird dann im betreffenden Seiten-Deskriptor hinterlegt. Daraufhin werden die Auslagerungsdateien gelesen und die ausgelagerte Seite in den neuen Rahmen kopiert. Abschließend werden die Einträge in den Auslagerungsdateien freigegeben. Der grobe Aufbau der dafür zuständigen Funktion `file_2_frame` stellt sich demnach wie folgt dar:

35a ⟨*file_2_frame* 35a⟩≡ (9)

```
int file_2_frame(int as, uint v_address) {
  ⟨Ermittlung Indizes aus Seitentabellen-Struktur 32b⟩
  ⟨file_2_frame - Auslagerungsdateien zu Seitenrahmen 36b⟩
  ⟨file_2_frame - Einträge in Auslagerungsdateien freigeben 36c⟩
  return TRUE;
}
```

Defines:
`file_2_frame`, used in chunk 42a.
Uses `TRUE` 30b and `uint` 13b.

Vor ihrer ersten Verwendung ist die Funktion in der Headerdatei `module.h` zu deklarieren:

35b ⟨*Prototypen* 19b⟩+≡ (10) ◁33b 38a▷

```
int file_2_frame(int as, uint v_address);
```

Defines:
`file_2_frame`, used in chunk 42a.
Uses `uint` 13b.

Die Ermittlung der Indizes für Page Directory und Seitentabellen folgt dem gleichen Schema wie in den Kapiteln 3.3 und 3.6.1 erläutert. Mit den ermittelten Indizes kann der betreffende Seiten-Deskriptor `pdesc` bearbeitet werden. Zuerst wird `frame_addr` die physische Adresse des neu angeforderten Seitenrahmens zugewiesen. Zur Anforderung eines neuen Seitenrahmens wird die Ulix-Funktion `request_new_frame` verwendet, die diesen im Hauptspeicher registriert, die globale Variable `free_frames` um einen Frame reduziert und die physische Adresse des neuen Rahmens zurückgibt:

35c ⟨*externe Prototypen* 20b⟩+≡ (10) ◁34b 43e▷

```
// ulix.c
extern int request_new_frame();
```

Defines:
`request_new_frame`, used in chunk 36b.

36a ⟨*externe Variablen* 14b⟩+≡ (10) ◁14b 40a▷

```
// ulix.c
extern int free_frames;
```

Defines:
`free_frames`, used in chunks 34c and 53c.

Im nächsten Schritt werden die Auslagerungsdateien durchsucht. Zuerst erfolgt der Zugriff auf die Indexdatei. Hier wird der Eintrag zur Seite gesucht, der entsprechende Index in der Blockdatei ermittelt und der betreffende Datenblock daraus gelesen.

36b ⟨*file_2_frame - Auslagerungsdateien zu Seitenrahmen* 36b⟩≡ (35a)

```
pf_page_desc* pdesc = get_pdesc(as, pdindex, ptindex);
index_entry entry;
out_page bfile_page;

// Auslagerungsdateien auslesen
int index = findblock(as, pageno);
rblock(ACC_INDEX, index, (uchar*) &entry);
rblock(ACC_OUT_PAGE, entry.block_no, (uchar*) &bfile_page);

// Neuen, freien Rahmen anfordern
pdesc->frame_addr = request_new_frame();

// Frameinhalt aus Zwischenpuffer in Frame kopieren
memcpy((uchar*) PHYSICAL(pdesc->frame_addr<<12),
       (uchar*) &bfile_page.load,
       PAGE_SIZE);

// pdesc aktualisieren
pdesc->present    = TRUE;
pdesc->paged_out  = FALSE;
```

Uses `ACC_INDEX` 27a, `ACC_OUT_PAGE` 27a, `FALSE` 30b, `findblock` 27b 27c, `get_pdesc` 19b, `index_entry` 21b 21b, `memcpy` 32d, `out_page` 21c 21c, `PAGE_SIZE` 16b, `pf_page_desc` 15b, `PHYSICAL` 19a, `rblock` 25b 25c, `request_new_frame` 35c, `TRUE` 30b, and `uchar` 13b.

Anschließend wird die Seite vom Puffer `bfile_page` in den Seitenrahmen geschrieben. Der Zugriff auf den Seitenrahmen erfolgt unter Verwendung des in Kapitel 3.4 beschriebenen Makros `PHYSICAL` über den Page-Deskriptor, der inzwischen auf den neuen Rahmen verweist. Nach dem Kopieren des Seiteninhalts in den neuen Seitenrahmen mit `memcpy` werden die Statusbits im Seiten-Deskriptor angepasst. Es wird also über `present` und `paged_out` die Information hinterlegt, dass die Seite nicht mehr ausgelagert ist. Die Seitentabellen sind somit wieder aktuell und Zugriffe auf die virtuelle Adresse werden auf den neuen Seitenrahmen verwiesen. Abgeschlossen wird der Vorgang mit der Freigabe der alten Einträge in den Auslagerungsdateien.

36c ⟨*file_2_frame - Einträge in Auslagerungsdateien freigeben* 36c⟩≡ (35a)

```
clearindex(index);
set_usedblocklist_bit((entry.block_no), FALSE);
```

Uses `clearindex` 28a 28b, `FALSE` 30b, and `set_usedblocklist_bit` 33b 33c.

Damit sind die grundlegenden Ein- und Auslagerungsfunktionen des Paging-Subsystems implementiert. Auf diesen Mechanismen aufbauend, wird im nachfolgenden Kapitel auf den Page Fault Handler, dessen Zweck und Arbeitsweise eingegangen.

3.7 Page Fault Handler

Ein Page Fault stellt einen Zugriffsfehler dar, bei dem versucht wird auf eine nicht verfügbare Adresse zuzugreifen. Dies kann entweder aufgrund fehlender Berechtigungen bzw. falscher Adressangaben geschehen oder die angeforderte Seite befindet sich derzeit nicht im Hauptspeicher [Sta05, S. 343].

Sobald bspw. der Seitenersetzungsalgorithmus (beschrieben in Kapitel 3.8.1) die Auslagerung einer Seite anstößt, wird der Seiten-Deskriptor `pdesc` nach erfolgter Auslagerung dahingehend angepasst, dass die Seite als ausgelagert (`present = FALSE`, `paged_out = TRUE`) gekennzeichnet ist (siehe Kapitel 3.6.1). Dies hat zur Folge, dass die MMU beim Zugriff auf diese Seite einen Page Fault verursacht. Der Handler `fault_handler` in Ulix liest zuerst das Struct `regs` aus und erkennt, dass es sich bei der Nummer `int_no = 14` um einen Seitenfehler handelt.

37a ⟨*Typdefinitionen* 13b⟩+≡ (10) ◁21c

```
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;
};

typedef struct regs regs;
```

Defines:
`ebx`, used in chunks 43c and 56e.
`ecx`, used in chunk 43c.
`int_no`, used in chunk 37c.
`regs`, used in chunks 38–43 and 56e.
Uses `uint` 13b.

Daraufhin ruft er den im Rahmen dieser Arbeit implementierten Page Fault Handler `page_fault` auf. Hierfür ist jedoch vorher die Registrierung des Handlers erforderlich. Dazu wird in der Datei `ulix.c` die Funktion `page_fault` als externer Prototyp hinterlegt.

37b ⟨*ulix.c - function prototypes - page_fault* 37b⟩≡

```
extern void page_fault ();
```

Uses `page_fault` 38a 38b.

Darüber hinaus ist in der selben Datei die Funktion `fault_handler` so anzupassen, dass `int_no = 14` zum Aufruf des Page Fault Handlers führt.

37c ⟨*ulix.c - fault_handler* 37c⟩≡

```
if (r->int_no==14) page_fault (r);
```

Uses `int_no` 37a and `page_fault` 38a 38b.

Schließlich ist die Funktion auch in der Header-Datei `module.h` zu deklarieren, damit sie verwendet werden kann.

38a ⟨*Prototypen* 19b⟩+≡ (10) ◁35b 43b▷

```
void page_fault(struct regs *regs);
```

Defines:
`page_fault`, used in chunk 37.
Uses `regs` 37a.

Der Aufbau des implementierten Page Fault Handlers definiert sich durch das Einholen der benötigten Informationen und die anschließende Fehler-Fall-Unterscheidung und Bearbeitung.

38b ⟨*page_fault* 38b⟩≡ (9)

```
void page_fault(struct regs *regs) {
  if(pfhaktiv) return;
  pfhaktiv = TRUE;

  ⟨page_fault - Benötigte Informationen 39a⟩

  ⟨page_fault - FALL 1 Teil1 - Kernelbereich 40c⟩
  ⟨page_fault - FALL 2 - neue Seite 41b⟩
  ⟨page_fault - FALL 3 - ausgelagerte Seite 42a⟩
  ⟨page_fault - FALL 4 - Schreibschutz 42b⟩
  ⟨page_fault - FALL 1 Teil2 - ungültige Adresse 41a⟩
  ⟨page_fault - Wenn kein Fehlerfall zutrifft 43a⟩
}
```

Defines:
`page_fault`, used in chunk 37.
Uses `regs` 37a and `TRUE` 30b.

Das Statusflag `pfhaktiv` soll verhindern, dass die Funktion während einer Page-Fault-Bearbeitung durch einen weiteren Page Fault nochmal aufgerufen wird. Das Flag ist als globale Variable definiert und wird beim Systemstart auf `FALSE` gesetzt.

38c ⟨*globale Variablen* 21a⟩+≡ (10) ◁21d 48c▷

```
uchar pfhaktiv;
```

Uses `uchar` 13b.

38d ⟨*initialize_module* 24b⟩+≡ (9) ◁24b 53b▷

```
pfhaktiv = FALSE;
```

Uses `FALSE` 30b.

Im Page Fault Handler wird bei eintretendem Page Fault eine Fallunterscheidung durchgeführt. Um die Ursache für einen Page Fault zu ermitteln, werden das Struct `regs`, die Informationen aus den Seitentabellen sowie die Adressen aus den CPU Control Registers `CR2` und `CR3` benötigt. Das Auslesen dieser Register erfolgt in Anlehnung an [Mol08, o.S.].

39a ⟨*page_fault - Benötigte Informationen* 39a⟩≡ (38b) 39b▷

```
uint faulting_address;
uint cr3_address;
__asm__ __volatile__("mov %%cr2, %0" : "=r" (faulting_address));
__asm__ __volatile__("mov %%cr3, %0" : "=r" (cr3_address));
```

Uses uint 13b.

Das Register CR3 wird verwendet sobald für die CPU Paging aktiviert wurde. Hier wird die Page-Directory-Adresse des jeweiligen vom Scheduler aktivierten Prozesses abgelegt. Das ermöglicht der MMU den Zugriff auf die Seitentabellen des jeweiligen Prozesses. Das Register CR2, welches ebenfalls nur bei aktivem Paging verwendet wird, beinhaltet die Adresse, deren Aufruf den Page Fault verursachte [Int87, S. 88]. Beide CPU Register werden via Inline-Assembler ausgelesen. Anschließend werden die Informationen zum Page Fault aus dem Struct `regs` ausgelesen. Das Element `err_code` aus `regs` beinhaltet mehrere Informationen bzgl. der Fehlerumstände des Page Faults. Die nachfolgende Variablenzuweisung erfolgt in Anlehnung an [Mol08, o.S.] durch UND-Verknüpfungen.

39b ⟨*page_fault - Benötigte Informationen* 39a⟩+≡ (38b) ◁39a 39c▷

```
int err_present = !(regs->err_code &  0x1); // 00001
int err_rw =        regs->err_code &  0x2;  // 00010
int err_us =        regs->err_code &  0x4;  // 00100
int err_reserved =  regs->err_code &  0x8;  // 01000
int err_id =        regs->err_code & 0x10;  // 10000
```

Uses regs 37a.

Neben dem Auslagerungszustand (`err_present`) lassen sich aus dem Fehlercode `err_code` die Zugriffsart (`err_rw`) und die Privilegierung des verursachenden Prozesses (`err_us`) ermitteln. Des Weiteren lässt sich feststellen, ob der Seitentabelleneintrag beschädigt ist (`err_reserved`), oder ob der Page Fault während dem Laden von Befehlen (im Rahmen der Befehlsverarbeitung) entstand (`err_id`) [Mol08, o.S.]. Das Auslesen von `err_code` geschieht, wie in den obigen Quellcode-Kommentaren verdeutlicht, über die UND-Verknüpfung der Fehlercodes mit Bitmasken.

Im weiteren Verlauf werden die Informationen zur virtuellen Adresse, die den Seitenfehler verursachte, ermittelt:

39c ⟨*page_fault - Benötigte Informationen* 39a⟩+≡ (38b) ◁39b 40b▷

```
int pageno = faulting_address / PAGE_SIZE;
int pdindex = pageno / MAX_TABLE_ENTRIES;
int ptindex = pageno % MAX_TABLE_ENTRIES;
pf_page_table_desc* ptd = get_ptd(current_as, pdindex);
pf_page_desc* pdesc = get_pdesc(current_as, pdindex, ptindex);
```

Uses current_as 40a, get_pdesc 19b, get_ptd 19b, MAX_TABLE_ENTRIES 13a, PAGE_SIZE 16b, pf_page_desc 15b, and pf_page_table_desc 14c.

Der übergebene Parameter `current_as` ist eine globale Variable des Betriebssystems, welche die Adressraumnummer des aktuell vom Scheduler aktivierten Prozesses beinhaltet. Sie wird wie folgt in die Header-Datei `module.h` eingebunden.

40a ⟨*externe Variablen* 14b⟩+≡ (10) ◁36a 43d▷

```
// ulix.c
extern int current_as;
```

Defines:
`current_as`, used in chunks 39c, 40b, 42a, and 56e.

Liegen alle relevanten Informationen vor, können daraus Fehlerfälle abgeleitet werden. Damit die auftretenden Page Faults nachvollziehbar sind, werden diese Informationen (in Anlehnung an [Mol08, o.S.]) mit ausgegeben.

40b ⟨*page_fault - Benötigte Informationen* 39a⟩+≡ (38b) ◁39c

```
printf("\n**********************************"
  "**********************************\n");
printf("Page fault! ( ");
  if (err_present)  {printf("present ");}
  if (err_rw)       {printf("read-only ");}
  if (err_us)       {printf("user-mode ");}
  if (err_reserved) {printf("reserved ");}
  if (err_id)       {printf("instruction-fetch ");}
printf (") \n(Faulting address: %08x, PD (cr3): %08x)\n",
  faulting_address, cr3_address);
print_pdesc(current_as, faulting_address);
```

Uses `current_as` 40a, `print_pdesc` 19c 20a, and `printf` 20b.

Da Ulix in der derzeit verwendeten Version einige Funktionen noch nicht unterstützt, werden im Rahmen dieser Arbeit nur die folgenden Fehlerfälle unterschieden:

- Fall 1: Auf welchen Adressraum wird zugegriffen?
- Fall 2: Wird auf die Seite das erste Mal zugegriffen?
- Fall 3: Wird auf eine ausgelagerte Seite zugegriffen?
- Fall 4: Ist die Seite schreibgeschützt?

Fall 1 wird mittels Überprüfung der Page-Fault-verursachenden Adresse geklärt. Ist die Adresse im Bereich ab 0xBFC00000, wurde versucht auf den Kernel-Bereich zuzugreifen. Da Seiten dieses Bereichs nicht ausgelagert werden, darf dieser Fall gewöhnlich nicht eintreten. Daher wird über die Funktion `segfault` ein Segmentation Fault ausgelöst und der betreffende Prozess gestoppt. Der Aufbau dieser Funktion wird nach der Fehlerfall-Differenzierung näher beschrieben.

40c ⟨*page_fault - FALL 1 Teil1 - Kernelbereich* 40c⟩≡ (38b)

```
if(faulting_address >= 0xBFC00000) {

  if(err_us) {
    printf("FEHLERHAFTER ZUGRIFF AUF "
      "KERNELBEREICH IM USERMODE\n");
    segfault(regs);
    return;

  }else{
```

```
        printf("FEHLERHAFTER ZUGRIFF AUF "
          "KERNELBEREICH IM KERNELMODE\n");
        segfault(regs);
        return;
      }
    }

  // Zugriff auf User-Adressraum
    else if(faulting_address >= 0 && faulting_address < 0xBFC00000) {
```

Uses `printf` 20b, `regs` 37a, and `segfault` 43b 43c.

Befindet sich die Adresse im User-Bereich des Prozess-Adressraumes (0 bis 0xBFBFFFFF), so werden im Fehlerfall auch die Seiten-Deskriptoren geprüft. Andernfalls erfolgt ein Segmentation Fault, wenn auf eine ungültige Adresse zugegriffen wurde:

41a ⟨*page_fault - FALL 1 Teil2 - ungültige Adresse* 41a⟩≡ (38b)

```
    }else{
      printf("UNGUELTIGE ADRESSE! ABBRUCH.\n");
      segfault(regs);
      pfhaktiv = FALSE;
      return;
    }
```

Uses `FALSE` 30b, `printf` 20b, `regs` 37a, and `segfault` 43b 43c.

Fall 2 soll die Situation abbilden, dass die angeforderte Seite noch nicht angelegt ist. Dies ist der Fall, wenn im Seiten-Deskriptor `pdesc` die Bits `present` und `paged_out` jeweils auf `FALSE` gesetzt sind.

41b ⟨*page_fault - FALL 2 - neue Seite* 41b⟩≡ (38b)

```
  // FALL 2: Auf die Seite wird das erste Mal zugegriffen
    if(err_present &&
      pdesc->present   == FALSE &&
      pdesc->paged_out == FALSE) {

      printf("AUF DIESE SEITE WIRD DAS ERSTE MAL ZUGEGRIFFEN\n");
      segfault(regs);
      pfhaktiv = FALSE;
      return;
    }
```

Uses `FALSE` 30b, `printf` 20b, `regs` 37a, and `segfault` 43b 43c.

An dieser Stelle müsste die Seitentabellenstruktur entsprechend um einen Seiten-Deskriptor (ggf. auch um eine Seitentabelle) erweitert werden. Hinzu kommt, dass ein neuer Seitenrahmen angefordert werden und dessen Adresse im neu angelegten Seiten-Deskriptor hinterlegt werden muss. Nachdem Ulix noch nicht über die dafür erforderlichen Schnittstellen verfügt, wird auch hier ein Segmentation Fault ausgelöst und der betreffende Prozess gestoppt.

Fall 3 tritt ein, wenn die angesprochene Adresse auf eine ausgelagerte Seite verweist. Dies ist gegeben, wenn die Seite als „nicht vorhanden“ (`present = FALSE`) aber als „ausgelagert“ (`paged_out = TRUE`) gekennzeichnet ist.

42a ⟨*page_fault - FALL 3 - ausgelagerte Seite* 42a⟩≡ (38b)

```
else if(err_present &&
  pdesc->present   == FALSE &&
  pdesc->paged_out == TRUE) {

  printf("DIE SEITE IST AUSGELAGERT "
    "UND WIRD NUN ZURUECKGESCHRIEBEN\n");
  file_2_frame(current_as, faulting_address);
  pfhaktiv = FALSE;
  return;
}
```

Uses `current_as` 40a, `FALSE` 30b, `file_2_frame` 35a 35b, `printf` 20b, and `TRUE` 30b.

Hier werden sowohl der aktuelle Adressraum `current_as` als auch die fehlerverursachende Adresse `faulting_address` der Funktion `file_2_frame` übergeben und damit die Wiedereinlagerung (wie in Kapitel 3.6.2 beschrieben) angestoßen.

Im 4. Fehlerfall besteht die Situation, dass die virtuelle Adresse auf eine Seite verweist, die schreibgeschützt ist. Dies wäre bspw. beim Copy-on-write-Verfahren relevant, wenn ein Kindprozess auf die gemeinsamen Daten von Vater- und Kindprozess zugreifen möchte. Die betreffende Seite wird dann vor dem Schreibzugriff kopiert, damit der ursprüngliche Inhalt für den Vaterprozess unverändert bleibt. Da Ulix bei der Erstellung eines Kindprozesses den gesamten Adressraum kopiert, kann dieser Fehler im Normalfall nicht eintreten und wird daher mit einem Segmentation Fault und dem entsprechenden Hinweis beantwortet.

42b ⟨*page_fault - FALL 4 - Schreibschutz* 42b⟩≡ (38b)

```
else if(
   err_rw &&
  (pdesc->writeable == FALSE ||
   ptd->writeable   == FALSE   )) {

  // Copy-on-Write wird in Ulix nicht unterstützt!
  printf("DIE SEITE IST SCHREIBGESCHUETZT\n");
  segfault(regs);
  pfhaktiv = FALSE;
  return;
}
```

Uses `FALSE` 30b, `printf` 20b, `regs` 37a, and `segfault` 43b 43c.

Konnte der Page Fault nicht genau zugeordnet werden, so wird ebenfalls ein Segmentation Fault ausgelöst und der entsprechende Hinweis ausgegeben. Dies würde jedoch bedeuten, dass die obige Fallunterscheidung unvollständig ist und darf deswegen nicht vorkommen.

43a ⟨*page_fault - Wenn kein Fehlerfall zutrifft* 43a⟩≡ (38b)
```
printf("ERROR: PAGE FAULT KONNTE NICHT ZUGEORDNET WERDEN!!\n");
segfault(regs);
pfhaktiv = FALSE;
return;
```
Uses FALSE 30b, printf 20b, regs 37a, and segfault 43b 43c.

Mit den oben beschriebenen Segmentation Faults werden die fehlerverursachenden Prozesse abgebrochen. Hierdurch wird verhindert, dass diese Prozesse weitere Instruktionen ausführen und dadurch ggf. das System schädigen. Aufgrund der noch nicht vollständigen Unterstützung von Signalen wird statt dem SIGSEGV-Signal ein SIGKILL-Signal an den Prozess geschickt.

43b ⟨*Prototypen* 19b⟩+≡ (10) ◁38a 45▷
```
void segfault(struct regs *regs);
```
Defines:
segfault, used in chunks 40–43.
Uses regs 37a.

43c ⟨*segfault* 43c⟩≡ (9)
```
void segfault(struct regs *regs) {
  printf("\nSEGMENTATION FAULT!\n");
  regs->ecx = 9;
  regs->ebx = current_task;

  syscall_kill(regs);
}
```
Defines:
segfault, used in chunks 40–43.
Uses current_task 43d, ebx 37a, ecx 37a, printf 20b, regs 37a, and syscall_kill 43e.

Zu diesem Zweck wird das Struct regs vom jeweiligen Page Fault verwendet. In den dafür vorgesehenen Teilelementen des Structs (ecx und ebx) werden die Signalnummer und die Prozess-ID des aktuell laufenden Prozesses hinterlegt. Die dafür benötigte Prozessnummer ist in der globalen Variable current_task gespeichert, die in der Header-Datei module.h deklariert wird:

43d ⟨*externe Variablen* 14b⟩+≡ (10) ◁40a
```
// ulix.c
extern int current_task;
```
Defines:
current_task, used in chunk 43c.

Anschließend wird das Kill-Signal mittels syscall_kill an den betreffenden Prozess geschickt.

43e ⟨*externe Prototypen* 20b⟩+≡ (10) ◁35c 56b▷
```
// ulix.c
extern void syscall_kill(struct regs *r);
```
Defines:
syscall_kill, used in chunk 43c.
Uses regs 37a.

`syscall_kill` ist ein Syscall-Handler in Ulix, der das übergebene Signal prüft und anschließend an den ebenfalls im Struct `regs` hinterlegten Prozess weiterleitet.

Mit dem damit implementierten Page Fault Handler können ausgelagerte Seiten zurück in den Hauptspeicher geschrieben werden, wenn der laufende Prozess darauf zugreift. Im nachfolgenden Kapitel 3.8 wird darauf eingegangen, anhand welcher Kriterien die auszulagernden Seiten ausgewählt werden.

3.8 Seitenersetzung

In den vorherigen Kapiteln wurde beschrieben, wie bereits ausgelagerte Seiten wieder in den Hauptspeicher übertragen werden. Um jedoch eine Seitenersetzung zu realisieren, benötigt das System eine Möglichkeit auslagerbare Seiten zu erkennen. Die Schwierigkeit liegt darin Seiten auszuwählen, die nur selten verwendet werden, da jeder Ein- und Auslagerungsvorgang die Systemleistung negativ beeinflusst [Tan09, S. 255].

Zu den geläufigsten Seitenersetzungsalgorithmen gehört u.a. der „First-In-First-Out“-Algorithmus (FIFO). Der Algorithmus verwendet eine Queue, in der die Seiten dem Alter nach aufgelistet sind. Im Falle einer Auslagerung wird die älteste Seite (an erster Stelle in der Queue) ausgewählt. Nachdem die älteste Seite jedoch nichts über die Nutzungshäufigkeit aussagt, kann es bei diesem Algorithmus vorkommen, dass unter Umständen eine häufig genutzte Seite ausgelagert wird [Tan09, S. 258].

Eine Weiterentwicklung stellt der Algorithmus „Second-Chance“ dar. Dieser arbeitet wie der FIFO, überprüft jedoch zusätzlich das `accessed`-Bit des Seiten-Deskriptors. Sollte dieses Bit zur ersten Seite in der Queue gesetzt sein, wird die Seite nicht ausgelagert, sondern mit zurückgesetztem `accessed`-Bit an das hintere Ende der Queue gehängt. Ist das Bit nicht gesetzt, wird die Seite ausgelagert. In der Variante „Clock“ des „Second-Chance“ werden die Seiten in einer ringförmigen Liste verwaltet, was den Vorteil hat, dass die Seiten mit gesetztem `accessed`-Bit nicht verschoben werden müssen. Es wird lediglich der Zeiger um eine Position nach vorne verschoben [Tan09, S. 258 f.].

Ebenfalls geläufig ist der „Least-Recently-Used“-Algorithmus (LRU). Über eine Bit-Matrix wird hierbei die Seite ermittelt, die am wenigsten genutzt wird. Aufgrund seiner Komplexität ist der LRU zwar theoretisch realisierbar, erfordert hierfür jedoch Spezialhardware. Praktikable Annäherungen bilden der „Not-Frequently-Used“-Algorithmus und seine Variante „Aging“. Beide arbeiten mit Zählern für jede Seite, die einen Indikator für die Zugriffshäufigkeit bilden [Tan09, S. 260 ff.].

Neben weiteren Algorithmen, deren Beschreibung über den Rahmen dieser Arbeit hinausgehen würde, findet auch der „Not-Recently-Used“-Algorithmus häufig Anwendung. Im Zuge dieser Implementierung wird er für die Realisierung der Seitenersetzung in Ulix herangezogen, da er verständlich ist und akzeptable Ergebnisse beim Paging liefert [Tan09, S. 257 f.]. Seine Arbeitsweise wird im nachfolgenden Kapitel 3.8.1 erläutert.

Des Weiteren ist zu prüfen, mit welcher Paging-Strategie der Seitenersetzungsalgorithmus angewandt werden soll. Es wird grundsätzlich zwischen lokaler und globaler Seitenersetzung differenziert. Abbildung 3.6 soll den Unterschied verdeutlichen.

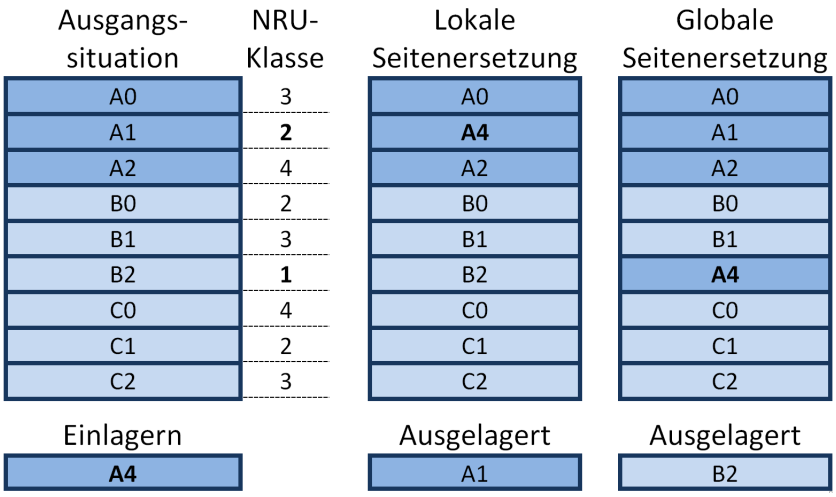

Abbildung 3.6: In Anlehnung an [Tan09, S. 271]: Differenzierung zwischen globalem und lokalem Paging

Wie in Abbildung 3.6 zu erkennen ist, ersetzen einzulagernde Seiten bei einer lokalen Paging-Strategie nur Seiten aus dem Adressraum, dem sie selbst zugeordnet sind. Dies führt zu einer festen, vordefinierten Anzahl an Seitenrahmen für alle Prozesse. Globale Paging-Strategien verteilen die Seitenrahmen hingegen dynamisch, da der Seitenersetzungsalgorithmus auf alle Speicherbereiche angewandt wird. Den Prozessen steht somit eine veränderliche Anzahl an Seitenrahmen zur Verfügung [Tan09, S. 270 f.].

In dieser Implementierung wird auf eine globale Paging-Strategie zurückgegriffen, da hierdurch das bei lokalen Strategien mögliche Thrashing (also die häufige Erzeugung von Seitenfehlern bei Erreichen der max. Seitenrahmenzahl im jeweiligen Prozess) verhindert wird [Tan09, S. 271].

3.8.1 Seitenersetzungsalgorithmus

Der Seitenersetzungsalgorithmus NRU ist in der Funktion `not_recently_used` implementiert. Für die spätere Verwendung dieser Funktion ist vorab der Prototyp in der Headerdatei `module.h` zu hinterlegen.

45 ⟨*Prototypen* 19b⟩+≡ (10) ◁43b 48b▷

```
int not_recently_used();
```

Defines:
`not_recently_used`, used in chunk 53c.

Gemäß dem NRU-Algorithmus sollen alle eingelagerten Seiten in Klassen unterteilt werden. Diese Untergliederung erfolgt in Ulix anhand der Statusbits in den Seiten-Deskriptoren, also den `accessed`- und `dirty`-Bits aller Seiten-Deskriptoren (`pdesc`). Die zwei Bits ergeben vier mögliche Konstellationen und somit vier Klassen:

Klasse 1	nicht referenziert nicht modifiziert	`accessed` = `FALSE` `dirty` = `FALSE`
Klasse 2	nicht referenziert modifiziert	`accessed` = `FALSE` `dirty` = `TRUE`
Klasse 3	referenziert nicht modifiziert	`accessed` = `TRUE` `dirty` = `FALSE`
Klasse 4	referenziert modifiziert	`accessed` = `TRUE` `dirty` = `TRUE`

Tabelle 3.1: In Anlehnung an [Tan09, S. 257]: Seitenkategorien im NRU-Algorithmus

Nach der erfolgten Kategorisierung soll gemäß NRU eine zufällige Seite aus der niedrigsten, nicht-leeren Klasse ausgelagert werden. So werden nur Seiten ausgelagert, von denen man aufgrund der zwei Statusbits ausgehen kann, dass sie im Moment weniger häufig benötigt werden, als andere eingelagerte Seiten [Tan09, S. 257]. Eine derartige Kategorisierung ließe sich auch über verkettete Listen realisieren. Da Ulix jedoch `malloc` noch nicht vollständig abbilden kann, wurde an dieser Stelle eine iterative Lösung ohne Listen gewählt:

46a ⟨*not_recently_used* 46a⟩≡ (9)

```
int not_recently_used() {
  ⟨not_recently_used - Variablendeklaration 47⟩
  ⟨not_recently_used - Schleife 46b⟩
    ⟨not_recently_used - Schleife - Klassifizierung 50⟩
  ⟨not_recently_used - Auslagerung 51b⟩
}
```

Defines:
`not_recently_used`, used in chunk 53c.

In dieser Variante wird für jede der vier Klassen nur eine Seite zwischengespeichert. Die Seitentabellenstruktur wird einmal durchlaufen, was durch drei ineinander geschachtelte For-Schleifen realisiert ist. Anhand der nachfolgenden Schleife ist ebenfalls zu erkennen, dass nicht nur die Seiten des derzeit aktiven Prozesses, sondern die Seiten aller Prozesse durchsucht werden und somit eine globale Paging-Strategie verfolgt wird (vgl. Kapitel 3.8).

46b ⟨*not_recently_used - Schleife* 46b⟩≡ (46a)

```
  // address spaces
  for(as=1;as<=1023;as++) {  // as 0 = Kernel

    // Nur belegte AS werden durchsucht.
    if(address_spaces[as].free) continue;

    // page directory (Elemente: page table desc)
    for(pdindex=0;pdindex<=766;pdindex++) {
      ptd = get_ptd(as, pdindex);

      // Nur vorhandene page tables werden durchsucht.
      if(!(ptd->present)) continue;
```

```
// page tables (Elemente: page desc)
for(ptindex=0;ptindex<=1023;ptindex++) {
  pdesc = get_pdesc(as, pdindex, ptindex);
```

Uses address_spaces 14b, get_pdesc 19b, and get_ptd 19b.

Auf oberster Ebene dieser verschachtelten Iteration werden die Adressräume (ab Adressraum 1, weil Adressraum 0 dem Kernel zugeordnet ist) durchlaufen. Währenddessen wird geprüft, ob der betrachtete Adressraum `address_spaces[as]` derzeit in Verwendung ist. Ungenutzte Adressräume werden an dieser Stelle übersprungen. Das gleiche gilt auch für das Page Directory (Elemente sind vom Typ `ptd`) und die Seiten-Deskriptoren. Die Iteration durch die Seitentabellenstruktur berücksichtigt zudem auch den in jeden Adressraum gemappten Kernelbereich. So werden nur die ersten 766 Seitentabellen im Page Directory geprüft, da der Kernelbereich bei Adresse 0xBFC00000 (Seitentabelle Nr. 767) beginnt. Nachfolgende Rechnung soll dies veranschaulichen:

Seitennummer = 0xBFC00000 / 0x1000 = 0xBFC00 bzw. 785408

Index im Page Directory = 785408 / 1024 = 767

Während die Seitentabellenstruktur durchlaufen wird, werden die jeweils aktuellen Seiten registriert und klassifiziert. Hierzu sind folgende Variablen erforderlich:

47 ⟨*not_recently_used - Variablendeklaration* 47⟩≡ (46a)

```
uint class1vaddr=0, class2vaddr=0,
     class3vaddr=0, class4vaddr=0;
int class1as=0, class2as=0, class3as=0, class4as=0;
int class1cnt=0, class2cnt=0, class3cnt=0, class4cnt=0;
int random_page = rand(4096)+1;

// Variablen zum Durchsuchen der Seitentabellen
int as,pdindex,ptindex;
pf_page_table_desc* ptd;
pf_page_desc* pdesc;
```

Uses pf_page_desc 15b, pf_page_table_desc 14c, rand 48b, and uint 13b.

Die Variablen `class1vaddr` – `class4vaddr` und `class1as` – `class4as` sollen während des Suchdurchlaufs die virtuelle Adresse sowie den dazugehörigen Adressraum der jeweils zuletzt gefundenen Seite speichern. Die Zähler `class1cnt` – `class4cnt` registrieren die Anzahl der Seitenfunde in ihrer jeweiligen Klasse und werden für den Abgleich mit einer vorgegebenen Zufallszahl `random_page` benötigt. Diese dient dazu, eine zufällige Seite aus der niedrigsten Klasse (die im Suchdurchlauf gefunden wurde) auszuwählen. Da der Hauptspeicher eine Größe von 64 MByte (also eine maximale Anzahl von 16384 Seiten mit einer Größe von jeweils 4 KByte) aufweist, wird bei vier möglichen Klassen und deren gleichmäßige Verteilung, mit einer maximalen Seitenanzahl von ca. 4096 Seiten pro Klasse gerechnet. Daher wird der Zufallsgenerator `rand` mit diesem Übergabeparameter aufgerufen.

Als Zufallsgenerator wird im Rahmen dieser Arbeit ein linearer Kongruenzgenerator herangezogen. Das bedeutet, dass die Zufallszahlengenerierung anhand einer linearen Funktion mit anschließender modularer Reduktion erfolgt. Jede ermittelte Zahl ist also maßgeblich für die Nachfolgende [Gen03, S. 11]. Die Menge der berechneten Werte hat somit nur den Anschein einer zufälligen Folge. Es wird daher von pseudozufälligen Zahlen gesprochen.

Es existieren diverse Varianten dieses Generators mit Zahlenfolgen unterschiedlicher Qualität. Sie unterscheiden sich dabei lediglich in der Belegung der Parameter `multiplikator`, `inkrement` und `modulus`. So werden bspw. Kongruenzgeneratoren mit den Parametern `multiplikator` = 16807, `inkrement` = 0 und `modulus` = $2^{31} - 1 = 2147483647$ aufgrund ihrer akzeptablen Zahlenfolgen als minimaler Standard bezeichnet. Ein weiteres Beispiel ist der „RANDU“-Generator der jedoch mit den Parametern `multiplikator` = 65539, `inkrement` = 0 und `modulus` = $2^{31} = 2147483648$ Zahlenreihen minderer Qualität liefert [Gen03, S. 20]. Damit später keine Prozesse benachteiligt werden und die Zahlenermittlung zudem möglichst schnell erfolgen kann, liegt der Fokus für die Zufallszahlengenerierung in dieser Arbeit primär auf einer einfachen Ermittlung und einer fairen Verteilung der Zufallszahlen. Daher wurden im Rahmen dieser Implementierung die Parameter des minimalen Standards gewählt:

48a ⟨*rand* 48a⟩≡ (9)

```
uint rand(int range) {

  uint zufallszahl;
  uint multiplikator = 16807;
  uint inkrement     = 0;
  uint modulus       = 2147483647;

  zufallszahl = (multiplikator * zufallszahl_alt + inkrement)% modulus;

  zufallszahl_alt = zufallszahl;
  return (zufallszahl % range);
}
```

Uses `rand` 48b, `uint` 13b, and `zufallszahl_alt` 48c.

Die Variable `zufallszahl_alt` beinhaltet den Startwert 1 und danach die jeweils vorherig ermittelte Zahl. Sie ist global gültig und wird zusammen mit dem Funktionsprototyp von `rand` in der Header-Datei `module.h` definiert:

48b ⟨*Prototypen* 19b⟩+≡ (10) ◁45 52c▷

```
uint rand(int range);
```

Defines:
`rand`, used in chunks 47, 48a, and 63.
Uses `uint` 13b.

48c ⟨*globale Variablen* 21a⟩+≡ (10) ◁38c 53a▷

```
int zufallszahl_alt = 1;
```

Defines:
`zufallszahl_alt`, used in chunk 48a.

Mit den erläuterten Variablen und dem Zufallsgenerator ist es möglich, die iterative Variante des NRU zu implementieren. Diese Variante sucht nicht aus einer zuvor klassifizierten Menge an Seiten eine zufällige Seite der niedrigsten, vorhandenen Klasse aus. Stattdessen speichert der Algorithmus den jeweils letzten Fund der zurzeit niedrigsten, gefundenen Klasse und erhöht den Zähler dieser Klasse. Erreicht dieser Zähler die zuvor ermittelte Zufallszahl, so wird der letzte Fund der niedrigsten, nicht-leeren Klasse ausgelagert. Das Programmablaufdiagramm in Abbildung 3.7 soll die Ablauflogik verdeutlichen.

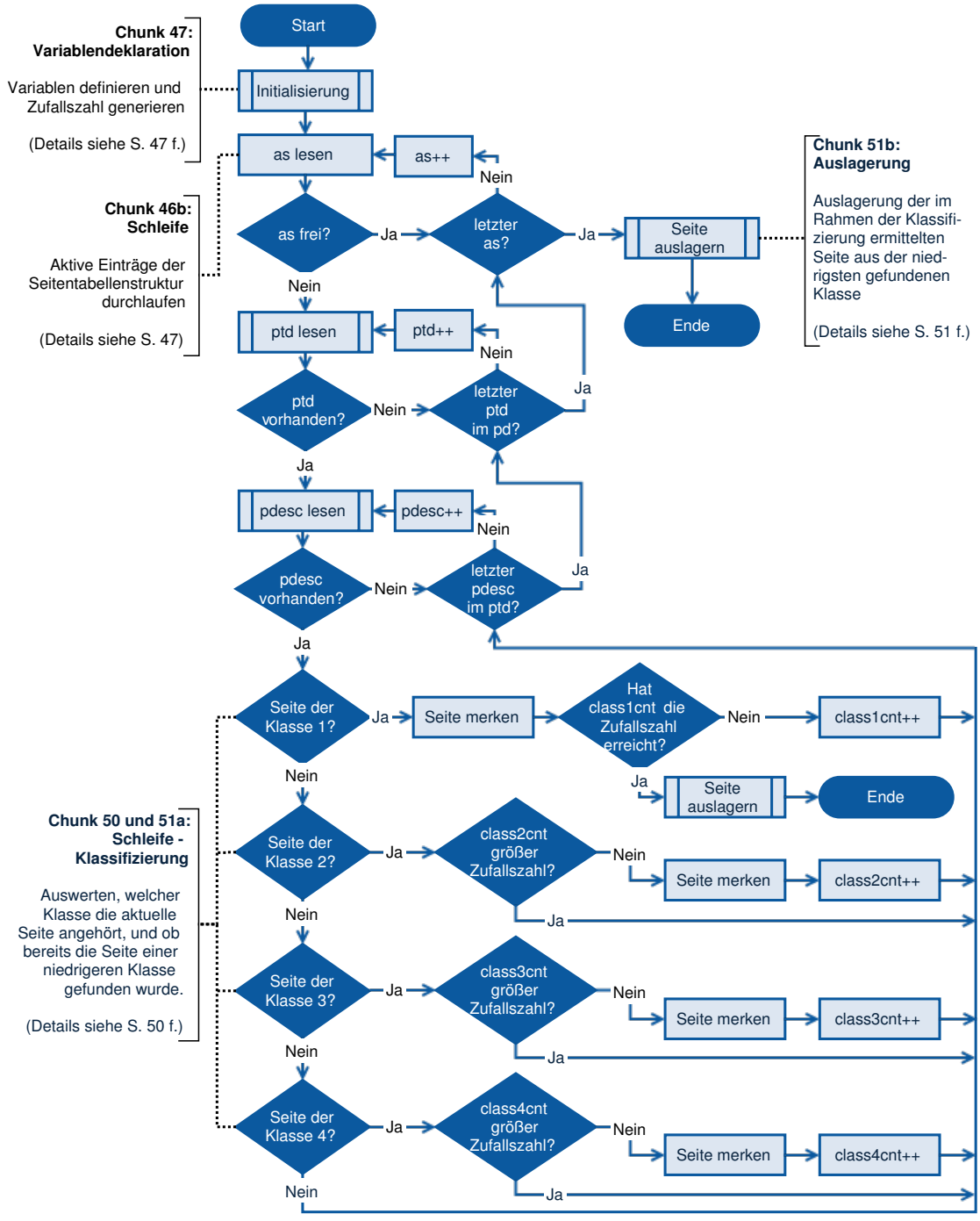

Abbildung 3.7: Iterative Variante des NRU-Algorithmus

Wie in Abbildung 3.7 zu erkennen ist, wird während der Iteration jede gefundene Seite anhand der Statusbits in `pdesc` geprüft. Da der NRU-Algorithmus vorschreibt, dass eine Seite aus der niedrigsten, nicht-leeren Klasse ausgelagert wird, werden Seiten nur registriert, wenn ihre Klasse der bisher niedrigsten Gefundenen entspricht. So werden bspw. Seiten der Klassen 3 und 4 ignoriert, wenn bereits eine Seite der Klasse 2 gefunden wurde. Nachfolgend wird diese Überprüfung anhand des Quellcodes näher erläutert.

Die Statusbits des aktuell betrachteten Seiten-Deskriptors `pdesc` werden gegen die Eigenschaften der Klasse 1 geprüft. Ist die Seite eingelagert (`present` = `TRUE`) aber nicht referenziert (`accessed` = `FALSE`) oder modifiziert (`dirty` = `FALSE`), so wird der Adressraum und die virtuelle Adresse in den Variablen `class1as` und `class1vaddr` abgelegt.

50 ⟨*not_recently_used - Schleife - Klassifizierung* 50⟩≡ (46a) 51a▷

```
if(pdesc->present && !(pdesc->accessed) && !(pdesc->dirty)) {
  class1as = as;
  class1vaddr = ((pdindex*MAX_TABLE_ENTRIES)+ptindex)*PAGE_SIZE;

  if(class1cnt == random_page) {
    printf("Funde: K1: %d, K2: %d, K3: %d, K4: %d, ZZ=%d\n",
      class1cnt, class2cnt, class3cnt,
      class4cnt, random_page);
    printf("Seite der Klasse 1 wird ausgelagert. vaddr=%08x,"
      " as=%d\n", class1vaddr, class1as);

    frame_2_file(class1as, class1vaddr);

    return 1;
  }
  class1cnt++;
}
```

Uses `frame_2_file` 31b 32a, `MAX_TABLE_ENTRIES` 13a, `PAGE_SIZE` 16b, and `printf` 20b.

Anschließend erfolgt die Prüfung, ob der Zähler `class1cnt` die Zufallszahl erreicht hat. Ist dies der Fall, wird die betreffende Seite ausgelagert und die Iteration abgebrochen. Als Rückgabewert wird die Klassen-Nummer verwendet.

Entspricht der im Zugriff befindliche Seiten-Deskriptor nicht den Eigenschaften der Klasse 1, so werden die Statusbits für die nachfolgenden Klassen geprüft. Dies erfolgt bei Klasse 2 jedoch nur, wenn bisher noch keine Seite der Klasse 1 registriert ist. Als Indikator hierfür wird die Variable, die den jeweiligen Adressraum beinhaltet (bspw. `class1as`), verwendet. Ebenso werden die höheren Klassen nur betrachtet, wenn noch keine Seite aus einer Niedrigeren registriert ist. Sobald also die erste Seite eingruppiert wurde, werden alle künftigen Seiten höherer Klassen übersprungen.

Bei Erreichen der Zufallszahl in den höheren Klassen wird jedoch nicht sofort ausgelagert oder die Iteration abgebrochen. Stattdessen wird der aktuelle Seiten-Deskriptor übersprungen und der Suchlauf fortgesetzt, da möglicherweise noch Seiten niedrigerer Klassen vorhanden sind.

51a ⟨*not_recently_used - Schleife - Klassifizierung* 50⟩+≡ (46a) ◁50

```
          if(!class1as &&
            pdesc->present && !(pdesc->accessed) && pdesc->dirty) {
            if(class2cnt++ >= random_page) continue;
            class2as = as;
            class2vaddr = ((pdindex*MAX_TABLE_ENTRIES)+ptindex)*PAGE_SIZE;
          }

          if(!class1as && !class2as &&
            pdesc->present && pdesc->accessed && !(pdesc->dirty)) {
            if(class3cnt++ >= random_page) continue;
            class3as = as;
            class3vaddr = ((pdindex*MAX_TABLE_ENTRIES)+ptindex)*PAGE_SIZE;
          }

          if(!class1as && !class2as && !class3as &&
            pdesc->present && pdesc->accessed && pdesc->dirty) {
            if(class4cnt++ >= random_page) continue;
            class4as = as;
            class4vaddr = ((pdindex*MAX_TABLE_ENTRIES)+ptindex)*PAGE_SIZE;
          }
        }
      }
    }
```

Uses MAX_TABLE_ENTRIES 13a and PAGE_SIZE 16b.

Wurde das Schleifenkonstrukt komplett durchlaufen (weil die Zufallszahl in Klasse 1 nicht erreicht wurde oder weil keine Seite der Klasse 1 gefunden wurde), wird die zuletzt gemerkte Seite der niedrigsten, nicht-leeren Klasse ausgelagert.

51b ⟨*not_recently_used - Auslagerung* 51b⟩≡ (46a)

```
    printf("Funde: K1: %d, K2: %d, K3: %d, K4: %d, ZZ=%d\n",
      class1cnt, class2cnt, class3cnt, class4cnt, random_page);

    uint vaddr;
    int class=0;

    if(class1as)           {as=class1as; vaddr=class1vaddr; class=1;}
    if(class2as && !class) {as=class2as; vaddr=class2vaddr; class=2;}
    if(class3as && !class) {as=class3as; vaddr=class3vaddr; class=3;}
    if(class4as && !class) {as=class4as; vaddr=class4vaddr; class=4;}
    if(!class)             {return -1;}

    printf("Seite der Klasse %d wird ausgelagert. vaddr=%08x, as=%d\n",
      class, vaddr, as);

    frame_2_file(as, vaddr);

    return class;
```

Uses frame_2_file 31b 32a, printf 20b, and uint 13b.

An dieser Stelle wird (bei Klasse 1 beginnend) aufsteigend geprüft, ob im Rahmen der Iteration eine Seite zwischengespeichert wurde. Ist dies der Fall, wird die betreffende Seite ausgelagert und die Funktion beendet.

Um im Seitenersetzungsalgorithmus über die Statusbits der Seiten Rückschlüsse auf deren Nutzungshäufigkeit zu ermöglichen, ist das `accessed`-Bit regelmäßig in allen Seiten-Deskriptoren zurückzusetzen. So lässt sich ermitteln, ob eine Seite seit dem letzten Zurücksetzen verwendet wurde. Des Weiteren gilt es auch zu definieren, wann der NRU-Algorithmus aufgerufen werden soll. Diese Punkte werden im nächsten Kapitel erörtert.

3.8.2 Speicherüberwachung

Die Speicherüberwachung ist normalerweise ein Daemon-Prozess, der in regelmäßigen Abständen die Speicherauslastung prüft und ggf. eine Seitenauslagerung anstößt. Ulix unterstützt momentan noch keine Daemon-Prozesse, deswegen wird eine Kernel-Funktion implementiert, die alle 100 Systemticks aufgerufen wird. Damit dies funktioniert, wird im Timer Handler von Ulix folgendes ergänzt:

52a ⟨*ulix.c - timer tasks* 52a⟩≡
```
if(system_ticks % 100 == 0) { kswapd(); }
```
Uses `kswapd` 52c 52d.

Um die Funktion `kswapd` später verwenden zu können, ist sie vorher sowohl in Ulix (in der Datei `ulix.c`) als auch in der Header-Datei `module.h` entsprechend zu deklarieren:

52b ⟨*ulix.c - function prototypes - kswapd* 52b⟩≡
```
extern void kswapd();
```
Uses `kswapd` 52c 52d.

52c ⟨*Prototypen* 19b⟩+≡ (10) ◁48b 56d▷
```
void kswapd();
```
Defines:
`kswapd`, used in chunk 52.

Nachdem der Daemon-Prozess in Linux `kswapd` heißt, wird die Funktion ebenfalls `kswapd` genannt [Tan09, S. 887]. Damit der im Kapitel 3.8.1 beschriebene Seitenersetzungsalgorithmus die Seiten im Bezug auf ihre Verwendung korrekt klassifizieren kann, werden (zusätzlich zur Prüfung der Speicherauslastung) die `accessed`-Bits in den Seiten-Deskriptoren im Sekundentakt zurückgesetzt. Wird auf eine Seite zugegriffen, setzt die CPU das Bit dieser Seite wieder auf `TRUE` [Int87, S. 101]. Der NRU-Algorithmus kann somit feststellen, ob seit dem letzten Zurücksetzen auf die jeweiligen Seite zugegriffen wurde. Die Struktur der Funktion `kswapd` stellt sich wie folgt dar:

52d ⟨*kswapd* 52d⟩≡ (9)
```
void kswapd() {
  if(nruaktiv) return;
```

```
    ⟨kswapd - Speicherüberwachung 53c⟩
    ⟨kswapd - Accessed-Bit zurücksetzen 54⟩

    return;
  }
```

Defines:
  kswapd, used in chunk 52.

Bevor die Funktion weitere Instruktionen ausführt wird die globale Variable `nruaktiv` geprüft. Sie soll verhindern, dass `kswapd` doppelt ausgeführt wird. Ähnlich der Variable `pfhaktiv` wird sie in der Header-Datei `module.h` deklariert und beim Systemstart in der Funktion `initialize_module` auf `FALSE` gesetzt.

53a ⟨*globale Variablen* 21a⟩+≡ (10) ◁48c

```
    uchar nruaktiv;
```

Uses uchar 13b.

53b ⟨*initialize_module* 24b⟩+≡ (9) ◁38d 56c▷

```
    nruaktiv = FALSE;
```

Uses FALSE 30b.

Enthält das NRU-Status-Flag den Wert `FALSE`, wird zuerst überprüft, ob die Anzahl der verwendeten Seitenrahmen im Hauptspeicher den Schwellwert `MIN_FREE_FRAMES` unterschritten hat. Ist dies der Fall, wird `nruaktiv` auf `TRUE` gesetzt und `not_recently_used` aufgerufen.

53c ⟨*kswapd - Speicherüberwachung* 53c⟩≡ (52d)

```
    if(free_frames < MIN_FREE_FRAMES) {
      nruaktiv = TRUE;
      printf("Dzt. %d freie Frames. Grenze von %d Frames ist "
        "unterschritten.\n--> NRU wird aufgerufen.\n"
        , free_frames, MIN_FREE_FRAMES);
      printf("Seite der Klasse %d ausgelagert.\n",not_recently_used());
      nruaktiv = FALSE;
      return;
    }
```

Uses FALSE 30b, free_frames 36a, MIN_FREE_FRAMES 53d, not_recently_used 45 46a, printf 20b, and TRUE 30b.

Die Konstante `MIN_FREE_FRAMES` ist in `module.h` festgelegt und definiert die globale Mindestanzahl der noch freien Seitenrahmen im Hauptspeicher.

53d ⟨*Konstanten* 13a⟩+≡ (10) ◁30b 56a▷

```
    #define MIN_FREE_FRAMES 100
```

Defines:
  MIN_FREE_FRAMES, used in chunk 53c.

Ist der Auslagerungsvorgang abgeschlossen, wird `nruaktiv` wieder auf `FALSE` gesetzt und die Funktion beendet. Die verwendete Schleifenstruktur zum Durchlaufen der

Seitentabellen ist mit der aus Kapitel 3.8.1 gleich. Es werden nur aktive bzw. belegte Elemente der Seitentabellenstruktur durchlaufen.

54 ⟨*kswapd - Accessed-Bit zurücksetzen* 54⟩≡ (52d)

```
int as,pdindex,ptindex;
pf_page_table_desc* ptd;
pf_page_desc* pdesc;

// address spaces
for(as=1;as<=1023;as++) {  // as 0 = Kernel

  // Nur belegte AS werden durchsucht.
  if(address_spaces[as].free) continue;

  // page directory (Elemente: page table desc)
  for(pdindex=0;pdindex<=766;pdindex++) {
    ptd = get_ptd(as, pdindex);

    // Nur vorhandene page tables werden durchsucht.
    if(!(ptd->present)) continue;

    // page tables (Elemente: page desc)
    for(ptindex=0;ptindex<=1023;ptindex++) {
      pdesc = get_pdesc(as, pdindex, ptindex);

      // Zurücksetzen des accessed-Bits
      if(pdesc->present && pdesc->accessed) pdesc->accessed = 0;
    }
  }
}
```

Uses address_spaces 14b, get_pdesc 19b, get_ptd 19b, pf_page_desc 15b, and pf_page_table_desc 14c.

Sobald eine Seite im Hauptspeicher präsent und deren `accessed`-Bit gesetzt ist, wird es zurückgesetzt. Somit kann im NRU-Algorithmus ermittelt werden, auf welche Seiten seit dem letzten Zurücksetzen zugegriffen wurde.

Damit ist das Paging-Subsystem für Ulix in seiner Grundfunktionalität komplett und einsatzbereit. Bei vollem Hauptspeicher werden prozessübergreifend wenig genutzte Seiten ausgelagert, was Platz für neue Prozesse schafft. Wird auf den Speicherbereich einer ausgelagerten Seite zugegriffen, kümmert sich der Page Fault Handler um die Wiedereinlagerung dieser Seite. Zudem werden falsche oder nicht berechtigte Zugriffe durch den Page Fault Handler unterbunden.

4 Test

Tests lassen sich grob in statische und dynamische Tests untergliedern. Statische Tests fokussieren sich primär auf den Quellcode und die damit verbundenen Teilaspekte wie die Syntax (Syntaxanalyse), den Datenfluss (Datenflussanalyse), den Kontrollfluss (Kontrollflussanalyse) oder den mathematisch-logischen Aufbau/Ablauf (formale Verifikation) [Rie97, S. 54]. Dynamische Tests differenzieren sich wiederum in unstrukturierte und formale Tests. Während unstrukturierte Tests sich auch auf den Quellcode beziehen, diesen aber anhand von Beispieldaten und Testfällen durchlaufen (Walkthrough-Tests), konzentrieren sich formale Tests einerseits auf das symbolische Ausführen bzw. auf das experimentelle Ausführen des Testobjekts in einer realen Umgebung (Test im engeren Sinn). Voraussetzung hierfür ist die Lauffähigkeit des Testobjekts [Rie97, S. 55 ff.].

Im Rahmen dieser Implementierung wurden statische Testverfahren zur Überprüfung der Funktionsfähigkeit der entworfenen Algorithmen sowie zur Entdeckung von nicht durchlaufenen Code-Elementen herangezogen. Darüber hinaus wurden dynamisch-formale Tests anhand von Testfällen vorgenommen, die die Funktionsfähigkeit des Paging-Subsystems prüfen sollen. Einige dieser Testfälle werden in den nachfolgenden Kapiteln exemplarisch beschrieben, um die Funktionsweise des Systems zu illustrieren.

4.1 Einlagerung (Page Fault Handler)

Die Anforderung an einen Page Fault Handler lässt sich anhand der im Kapitel 3.7 beschriebenen Funktionen festlegen. Demnach soll im Fall eines falschen oder nicht berechtigten Zugriffs (bspw. User-Mode-Anwendung auf Kernel-Adressraum) ein Segmentation Fault ausgelöst und der betreffende Prozess gestoppt werden. Findet ein Zugriff auf eine nicht-präsente Seite statt, so soll diese Seite vom Auslagerungsort zurück in den Hauptspeicher geladen werden, damit die CPU die verursachende Anweisung wiederholen kann.

Testfall	p1: Seite nach Page-Fault wiedereinlagern
Ziel	CPU kann nach Wiedereinlagerung der Seite die Page Fault-verursachende Anweisung ausführen
Vorbedingung	Betreffender Speicherbereich muss zuvor ausgelagert sein
Nachbedingung	Anwendung kann weiter ausgeführt werden

Tabelle 4.1: Spezifikation Testfall „p1“

Ausgehend von der Spezifikation in Tabelle 4.1 hat die Implementierung des Testfalls die Besonderheit, dass hier eine Interaktion zwischen User-Mode-Prozess und Kernel stattfinden muss. Der Prozess selbst ist nicht privilegiert, den Auslagerungsvorgang im Betriebssystem anzustoßen. Daher wird für diesen Testfall ein Syscall mit Signalnummer 99 eingeführt, über den der Auslagerungsvorgang gestartet werden

soll. Damit das Paging-Subsystem Syscalls verarbeiten kann, sind die Signalnummer und der dazugehörige Signal Handler zu registrieren. Zur besseren Lesbarkeit wird vorab `SYSCALL_P1` definiert:

56a ⟨*Konstanten* 13a⟩+≡ (10) ◁53d

```
#define SYSCALL_P1 99  // Syscall für Testfall "p1" in sh.c
```

Defines:
`SYSCALL_P1`, used in chunk 56c.

Anschließend wird die externe Funktion `insert_syscall` eingebunden, über die im Betriebssystem eine Registrierung der Signalnummer mit dazugehörigem Signal Handler erfolgen kann. Die Deklaration erfolgt in der Header-Datei `module.h`.

56b ⟨*externe Prototypen* 20b⟩+≡ (10) ◁43e

```
// ulix.c
extern void insert_syscall(int, void *);
```

Nun kann der Syscall beim Systemstart für den Testfall „p1“ mit seinem dazugehörigen Handler `testfall_p1` im System registriert werden.

56c ⟨*initialize_module* 24b⟩+≡ (9) ◁53b

```
    insert_syscall(SYSCALL_P1, testfall_p1);
    return;
}
```

Uses `SYSCALL_P1` 56a and `testfall_p1` 56d 56e.

Dadurch wird dem Usermode-Kindprozess des Testfalls ermöglicht, durch den Syscall `SYSCALL_P1` die Kernel-Funktion `testfall_p1` aufzurufen und die Adresse der angelegten Testvariable `testvar` durch das Struct `regs` zu übergeben. Vor der ersten Verwendung der Funktion `testfall_p1` ist deren Prototyp in der Header-Datei `module.h` zu definieren.

56d ⟨*Prototypen* 19b⟩+≡ (10) ◁52c 63c▷

```
void testfall_p1();
```

Defines:
`testfall_p1`, used in chunk 56c.

Die Funktion `testfall_p1` liest die übergebene Adresse aus und startet (neben der Testausgabe via `print_pdesc`) durch Aufruf von `frame_2_file` die Auslagerung. `frame_2_file` ermittelt dann, wie in Kapitel 3.6.1 ersichtlich, vor der Auslagerung die Adresse der Seite.

56e ⟨*Testfall P1* 56e⟩≡ (9)

```
void testfall_p1(struct regs *r) {
  printf("testfall_p1: Adresse der Test-Variable"
    " %d wurde uebergeben\n", r->ebx);
  print_pdesc(current_as, r->ebx);
  frame_2_file(current_as, r->ebx);

  return;
}
```

Defines:
  `testfall_p1`, used in chunk 56c.
Uses `current_as` 40a, `ebx` 37a, `frame_2_file` 31b 32a, `print_pdesc` 19c 20a, `printf` 20b, and `regs` 37a.

Damit ist der Syscall für User-Mode-Prozesse verfügbar. Das Testprogramm, welches den Syscall verwenden soll, wird in der Quelldatei `sh.c` zur User-Mode-Shell mit dem Kommando „p1“ definiert. Durch diesen Befehl wird ein Kindprozess gestartet, der im ersten Schritt eine Variable anlegt und deren Inhalt sowie Adresse ausgibt. Der Variableninhalt ist hier mit Character-Werten (66 = „B“, 65 = „A“) gestaltet, damit er im Hexdump der Auslagerungsdatei `blockfile` leichter zu erkennen ist.

57a ⟨*sh.c - Command p1* 57a⟩≡ 57b▷
```
else if ( strcmp ((char*)cmd, "p1") ) {
  int pid1;
  pid1 = fork ();

  if (pid1 == 0) {  // Kindprozess
    char testvar[256];
    memset(testvar,      66, 2);
    memset(testvar+2,    65, 252);
    memset(testvar+254,  66, 2);

    printf("Variable testvar = %s\n", testvar);
    printf("Adresse testvar = %08x // %d\n", &testvar, &testvar);
```
Uses `memset` 28c and `printf` 20b.

Anschließend wird mittels Syscall die Auslagerung der Seite (in der die Variable referenziert ist) angestoßen. Hierfür wird auf die Ulix-Funktion `syscall2` zurückgegriffen, da dieser neben der Signal-Nummer auch die Adresse der Variable `testvar` übergeben werden kann. So ist es möglich anhand der virtuellen Adresse von `testvar` die virtuelle Adresse der Seite zu ermitteln, in der die Variable abgelegt ist.

57b ⟨*sh.c - Command p1* 57a⟩+≡ ◁57a 57c▷
```
    syscall2(99, &testvar);
```

Nachdem die Seite ausgelagert und die Vorbedingung damit erfüllt ist, soll durch Zugriff auf die ausgelagerte Variable der Page Fault ausgelöst werden. Dies wird bspw. durch eine Bildschirmausgabe mittels `printf` erreicht.

57c ⟨*sh.c - Command p1* 57a⟩+≡ ◁57b
```
    printf("Variable testvar = %s\n", testvar);
    printf("Adresse testvar = %08x // %d\n", &testvar, &testvar);
    exit (0);  // Kindprozess beenden

  } else {  // Vaterprozess
    waitpid (pid1, &status, 0);
  }
}
```
Uses `printf` 20b.

Wenn der Page Fault Handler die hier getestete Funktionalität beherrscht, wird der Inhalt und die Adresse von `testvar` nach der Wiedereinlagerung korrekt ausgegeben.

Abbildung 4.1 zeigt die Bildschirmausgabe des Testprogramms. Im oberen Teil der Abbildung ist zu erkennen, wie Inhalt und Adresse der neu angelegten Variable `testvar`, nachdem sie mit `memset` befüllt wurde, ausgegeben werden. Anschließend erfolgt der Aufruf des Syscalls zur Auslagerung der Seite, in der die Variable `testvar` gespeichert ist. Der dazugehörige Handler gibt aus, dass er die Adresse der Variable erhalten hat. Vor der Auslagerung werden die Informationen zum Seitentabellen-Deskriptor `ptd` und zum Seiten-Deskriptor `pdesc` ausgegeben. Abschließend wird die Funktion `frame_2_file` gestartet, welche die Seite in die Auslagerungsdateien schreibt.

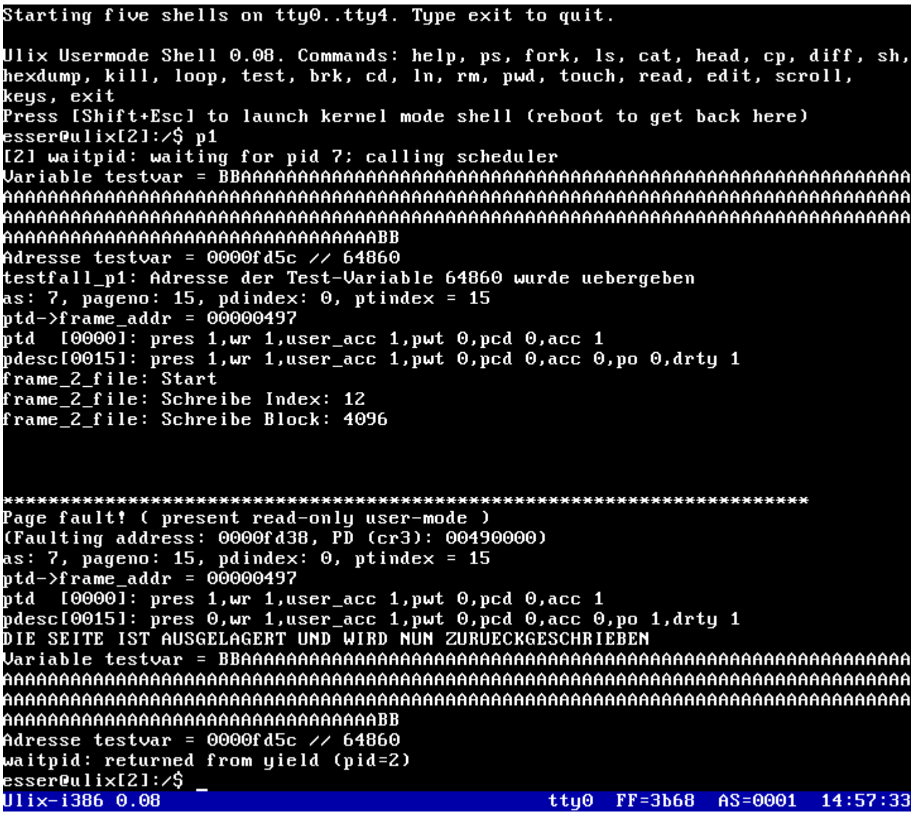

Abbildung 4.1: Testfall „p1“

Der untere Teil der Abbildung 4.1 zeigt die Ausgabe des Page Fault Handlers, der durch Zugriff auf den ausgelagerten Speicherbereich der Variable `testvar` ausgelöst wurde. Nach dem Zurückschreiben in den Hauptspeicher, belegt die wiederholte Ausgabe des Variableninhalts von `testvar`, dass die Daten richtig zurückgeschrieben wurden. Der Testfall wurde somit korrekt abgearbeitet.

4.2 Auslagerung

Die Sicherstellung der Funktionalität des Auslagerungsmechanismus wird zum einen an der Befüllung der Auslagerungsdateien und zum anderen an der Integrität der

ausgelagerten Daten festgemacht. Der dazugehörige Testfall „p2“ wird somit wie folgt spezifiziert:

Testfall	p2: Seite korrekt auslagern
Ziel	Seite ist korrekt indiziert und die Datenintegrität ist gewährleistet
Vorbedingung	Betreffende Seite ist im User-Mode-Speicherbereich
Nachbedingung	Daten sind nach Wiedereinlagerung weiter verwendbar

Tabelle 4.2: Spezifikation Testfall „p2“

Für diesen Testfall wird der in Kapitel 4.1 beschriebene Test nach dem Auslagerungsvorgang abgebrochen, da durch die anschließende Wiedereinlagerung die Datensätze in den Auslagerungsdateien gelöscht werden. Somit können nach der Ausführung des Testfalls die Auslagerungsdateien mittels Hexdump ausgelesen werden.

Wie im vorhergehenden Kapitel zu sehen ist, wird die Variable mittels `memset` mit Character-Werten (66 = „B“, 65 = „A“) befüllt. Durch diese Gestaltung soll der Wert und die Position der Variable in der Auslagerungsdatei besser hervorgehoben werden. Um prüfen zu können, ob wirklich die gesamte Seite korrekt ausgelagert wird, soll die Variable die gesamte Seite ausfüllen.

59 ⟨*sh.c - Command p1 - Testvariable* 59⟩≡

```
char testvar[PAGE_SIZE] __attribute__ ((aligned (PAGE_SIZE)));
memset(testvar,      66, 2);
memset(testvar+2,    65, 4092);
memset(testvar+4094, 66, 2);
```

Uses `memset` 28c and `PAGE_SIZE` 16b.

Durch den Schalter `__attribute__ ((aligned (PAGE_SIZE)))` wird dem Compiler die Anordnung der Variable `testvar` im Speicher vorgegeben. In diesem Fall wird `PAGE_SIZE` verwendet, damit die Variable bei einem Vielfachen von 4096 beginnt, also immer am Anfang einer Seite steht. Abbildung 4.2 zeigt den Inhalt der Auslagerungsdateien `indexfile` und `blockfile`.

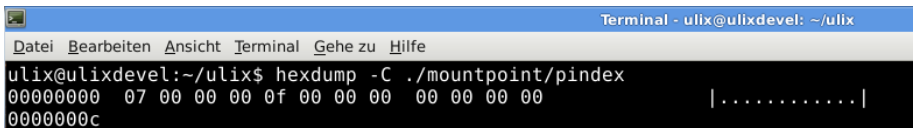

Abbildung 4.2: Testfall „p2“ - Hexdump der Indexdatei

Ein Vergleich des Hexdumps der Datei `indexfile` mit der Abbildung 4.1 zeigt, dass der Datensatz korrekt angelegt und richtig befüllt wurde: Es werden zum einen die korrekten Angaben zu Adressraum (`as` = 7) und Seitennummer (`pageno` = 15) der ausgelagerten Seiten gespeichert. Zum anderen erfolgt der korrekte Verweis auf den Eintrag in der Datei `blockfile` (`block_no` = 0):

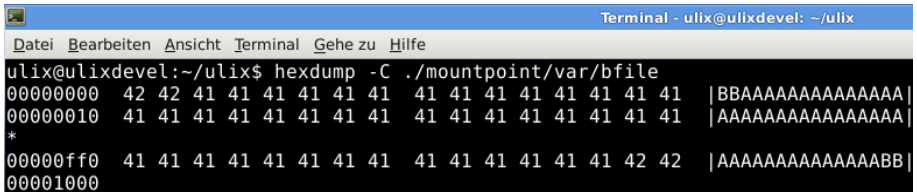

Abbildung 4.3: Testfall „p2“ - Hexdump der Blockdatei

Wie in Abbildung 4.3 zu erkennen ist, wurde der Wert der Variablen `testvar` unverändert und an die korrekte Position abgelegt. Dies belegen zum einen die ersten zwei und die letzten zwei „B“ der Zeichenkette, welche im Screenshot korrekt angezeigt werden. Zum anderen reicht der Variablenwert von Byte 0x0 bis Byte 0x1000, was mit 4096 Byte genau der Größe einer Seite (vgl. `PAGE_SIZE`) entspricht. Der Auslagerungsvorgang wurde folglich korrekt ausgeführt.

4.3 Seitenersetzung

Die Implementierung der Seitenersetzung ist, wie in den Kapiteln 3.8.1 und 3.8.2 ersichtlich, in zwei Teilbereiche untergliedert. In der Speicherüberwachung soll zum einen das `accessed`-Bit der derzeit verwendeten Seiten in regelmäßigen Zeitabständen zurückgesetzt werden, damit diese im Bezug auf ihre Nutzung klassifiziert werden können. Zum anderen ist die Anzahl der belegten Seitenrahmen im Hauptspeicher zu überwachen, damit rechtzeitig Platz für neue Seiten geschaffen werden kann. Darüber hinaus soll der NRU-Algorithmus, beim Unterschreiten der Mindestanzahl von freien Seitenrahmen, aktuell wenig genutzte Seiten auslagern, bis diese Schwelle nicht mehr unterschritten ist. Daher ist Testfall „p3“ wie folgt definiert:

Testfall	p3: NRU
Ziel	Es sollen wenig genutzte Seiten korrekt ausgelagert werden
Vorbedingung	- `accessed`-Bits wurden regelmäßig zurückgesetzt - `MIN_FREE_FRAMES` wurde unterschritten
Nachbedingung	Anzahl der belegten Seitenrahmen im Hauptspeicher unterschreitet nicht mehr `MIN_FREE_FRAMES`

Tabelle 4.3: Spezifikation Testfall „p3“

Zur Umsetzung der Testfall-Spezifikation wird in der Quelldatei zur User-Mode-Shell `sh.c` das Kommando „p3“ definiert. Dieser Befehl startet einen Kindprozess, der mithilfe der Ulix-Funktion `sbrk` 15096 Seiten anfordert. Dies hat zur Folge, dass dementsprechend neue Seitenrahmen im Hauptspeicher reserviert werden und dadurch die Anzahl der freien Seitenrahmen `free_frames` unter die Schwelle von `MIN_FREE_FRAMES` (siehe Kapitel 3.8.2) sinkt.

61 ⟨*sh.c - Command p3* 61⟩≡

```
else if ( strcmp ((char*)cmd, "p3") ) {
  int cnt, pages, status, pid1;
  pid1 = fork ();

  if (pid1 == 0) { // Kindprozess
    void *brk;
    void *last_brk;
    pages = 15096;

    do{
      if(cnt < pages){
        brk = sbrk (4096);
        if((int) brk == -1) printf("FEHLER!\n");

        last_brk = brk;
        cnt++;
      }
    }while(1);

  } else {
    // Vaterprozess
    waitpid (pid1, &status, 0);
  }
}
```

Uses `printf` 20b.

Über die Variable `pages` lässt sich die Anzahl der anzufordernden Seiten festlegen. Die hier definierte Seitenanzahl bewirkt (unter der Voraussetzung, dass der Testfall direkt nach dem Systemstart ausgeführt wird) eine Speicherbelegung, die genau 96 freie Seitenrahmen übrig lässt. Die Schwelle in `MIN_FREE_FRAMES` ist somit um vier Seitenrahmen unterschritten (siehe Kapitel 3.8.2).

Damit der Testfall funktionieren kann, ist es erforderlich, dass der Prozess aktiv bleibt. Ansonsten würde der Adressraum und damit der belegte Speicher freigegeben werden, bevor das System die Speicherbelegung regelt. Deswegen wurde die Iteration als Endlos-Schleife implementiert.

Der nachfolgende Screenshot in Abbildung 4.4 zeigt die Testausgaben aus der Speicherüberwachung und dem NRU-Algorithmus. Wie hier ersichtlich ist, unterschreitet die Anzahl der noch verfügbaren Seitenrahmen im Hauptspeicher den in `MIN_FREE_FRAMES` hinterlegten Testwert um vier Rahmen. Folglich sind vier Seiten auszulagern.

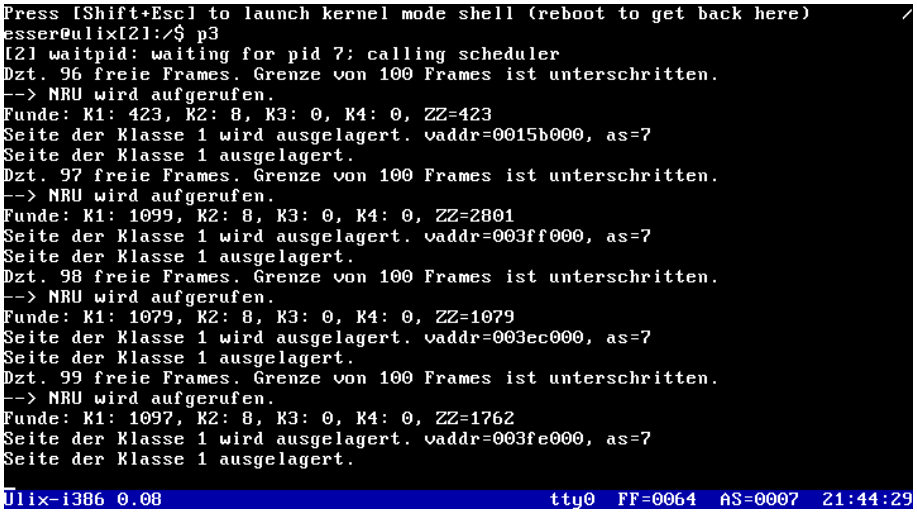

Abbildung 4.4: Testfall „p3“ - NRU-Algorithmus

Wie Abbildung 4.4 zeigt, erkennt die Speicherüberwachung das Unterschreiten des Schwellwertes von 100 freien Seitenrahmen und startet den NRU-Algorithmus. Im ersten Durchlauf wird die Zufallszahl auf 423 festgelegt. Die Seitentabellenstruktur wird also solange durchsucht, bis entweder die Zufallszahl durch den Zähler einer Seitenklasse oder das Ende der Seitentabellenstruktur erreicht ist. In dem Fall wurde die Zufallszahl durch den Zähler der Klasse 1 erreicht und der 423. Fund in dieser Klasse ausgelagert. Bei der nächsten Überwachung wird zwar festgestellt, dass sich die Anzahl der freien Seitenrahmen um einen Zähler erhöht hat, jedoch immer noch unter dem Schwellwert liegt. Deshalb wird der NRU noch weitere drei Mal aufgerufen. Am unteren Rand der Abbildung ist mit `FF=0064` (0x64 (hex) = 100 (dez)) die Anzahl der freien Seitenrahmen nach der erfolgten Regelung zu sehen.

Es lässt sich folglich festhalten, dass zum einen die Speicherüberwachung bei Unterschreiten des Schwellwertes den NRU-Algorithmus aufgerufen hat. Zum anderen wurden über den NRU-Algorithmus wenig benutzte Seiten der Klasse 1 zufällig ausgewählt und ausgelagert, bis der Schwellwert nicht mehr unterschritten war. Der Testfall wurde somit korrekt ausgeführt.

4.4 Zufallsgenerator

Der Zufallsgenerator für den NRU-Algorithmus bedient sich, wie in Kapitel 3.8.1 erläutert, der linearen Kongruenz zur Ermittlung von pseudozufälligen Zahlenfolgen [Gen03, S. 11 ff.]. Um sicherzustellen, dass keine Prozesse durch eine falsche Zufallszahlgenerierung ungewollt bevorzugt werden, soll der Testfall „rand“ die Verteilung der generierten Zahlen sowie deren Vorkommenshäufigkeit aufzeigen.

Testfall	rand: Faire Verteilung der Zufallszahlen
Ziel	generierte Zufallszahlen sollen in etwa gleich oft vorkommen
Vorbedingung	Die globale Variable `zufallszahl_alt` muss initialisiert sein
Nachbedingung	keine

Tabelle 4.4: Spezifikation Testfall „rand“

Damit der Zufallsgenerator dem Testfall entsprechend überprüft werden kann, wird die Funktion `testfall_rand` in der Datei `ulix.c` als externe Funktion mit eingebunden:

63a ⟨*ulix.c - function prototypes - testfall_rand* 63a⟩≡
```
extern void testfall_rand();
```
Uses `testfall_rand` 63c 63d.

Für den späteren Aufruf der Funktion ist ein neuer Befehl in der Shell erforderlich. Um nicht einen weiteren Syscall verwenden zu müssen, wird der Befehl diesmal in der Shell für den Kernel-Modus angelegt. Der User-Mode ist in diesem Fall nicht erforderlich, und ein weiterer Syscall würde den Quellcode nur unnötig komplex werden lassen.

63b ⟨*ulix.c - Command rand* 63b⟩≡
```
} else if ( strcmp (s, "rand") ) {
  testfall_rand();
```
Uses `rand` 48b and `testfall_rand` 63c 63d.

Des Weiteren ist die Deklaration der Funktion in der Header-Datei `module.h` erforderlich.

63c ⟨*Prototypen* 19b⟩+≡ (10) ◁56d
```
  void testfall_rand();
```
Defines:
`testfall_rand`, used in chunk 63.

Anschließend wird die Testfall-Funktion `testfall_rand` implementiert. Sie soll einerseits die Vorkommensreihenfolge und andererseits die Vorkommenshäufigkeit aufzeigen. Daher werden die mittels `rand` generierten Zufallszahlen zuerst über `printf` ausgegeben und deren Vorkommen im Array `zufall` gezählt. Um die spätere Ausgabe übersichtlich zu halten, werden nur Zahlen von 1 – 10 generiert.

63d ⟨*Testfall Random* 63d⟩≡ (9) 64▷
```
void testfall_rand() {
  int i, zz;
  int max = 500;
  int zufall[11] = {0};
```

```
for(i=1;i<=max;i++) {

  zz = rand(10)+1;

  printf("%02d ", zz);
  zufall[zz]++;

  if((i%26) == 0) printf("\n");
}
```

Defines:
  testfall_rand, used in chunk 63.
Uses printf 20b and rand 48b.

Für eine bessere Übersicht wird nach 26 Zahlen ein Zeilenumbruch ausgegeben. Wurde die Schleife komplett durchlaufen, zeigt eine Zusammenfassung die Häufigkeiten der einzelnen Zahlen an:

64 ⟨*Testfall Random* 63d⟩+≡ (9) ◁63d

```
printf("\nVerteilung der Zufallszahlen bei %d Durchlaeufen:\n"
  "1\t2\t3\t4\t5\t6\t7\t8\t9\t10\n"
  "%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t", max,
  zufall[1],zufall[2],zufall[3],zufall[4],zufall[5],
  zufall[6],zufall[7],zufall[8],zufall[9],zufall[10]);

return;
}
```

Uses printf 20b.

Abbildung 4.5 zeigt die Ausführung des Testfalls mit max = 500 generierten Zahlen. Es zeigt sich dass die Vorkommensreihenfolge zufälligen Charakter hat. Die Verteilung der Zufallszahlen (siehe unten in Abbildung 4.5) zeigt auch, dass jede Zahl der festgelegten Periode von 1 – 10 in etwa gleich oft generiert wird.

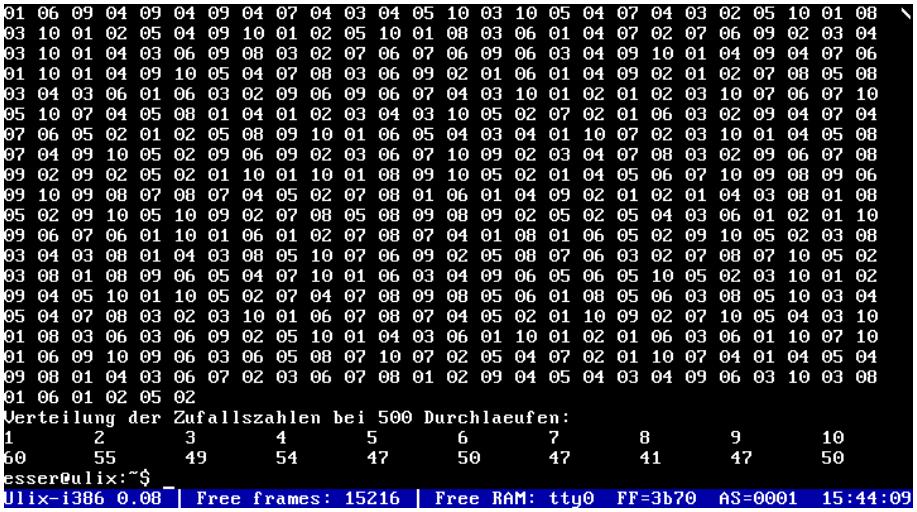

Abbildung 4.5: Testfall „rand“ - Vorkommensreihenfolge und Häufigkeit (500 Zahlen)

Der gleiche Testlauf mit einer größeren Zahlenmenge zeigt im Groben die gleiche Verteilung, wie sie in Abbildung 4.6 zu sehen ist.

Abbildung 4.6: Testfall „rand“ - Vorkommensreihenfolge und Häufigkeit (10000 Zahlen)

Damit gilt der Testfall als korrekt ausgeführt und die Funktionsfähigkeit des Zufallsgenerators für den in dieser Arbeit beschriebenen Anwendungsbereich als gegeben.

Das Bestehen der hier aufgeführten Testfälle zeigt die grundsätzliche Funktions- und Einsatzfähigkeit des implementierten Paging-Subsystems. Im Zuge der Entwicklung dieses Subsystems wurden noch diverse weitere Tests durchgeführt, deren Beschreibung über den Rahmen dieser Arbeit hinaus gehen würde.

5 Fazit

Wie sich im Laufe dieser Arbeit herausgestellt hat, birgt die Implementierung des Paging-Subsystems mit Literate Programming im Vergleich zu konventionellen Programmiermethoden einen gewissen Mehraufwand. Die stetige Abwechslung der formalen und informalen Elemente innerhalb der noweb-Dateien bewirkt jedoch eine detailliertere Beschreibung der betreffenden Funktionalitäten. Da der komplette Quellcode im Dokument enthalten und beschrieben ist, bildet auch ein übersichtlicher Aufbau des Quellcodes mit aussagekräftigen Funktions- und Variablennamen die Voraussetzung für eine verständliche Dokumentation. Darüber hinaus begünstigt der Aufbau die Wartbarkeit, da Änderungen des Quellcodes auch dazugehörige Aktualisierungen in der Dokumentation erleichtern.

5.1 Kritische Würdigung

Die Forschungslücke und die sich daraus ergebende Problemstellung für diese Arbeit wurden unter Verwendung des Rahmenwerks zur wissenschaftlichen Literaturrecherche nach H. Cooper ermittelt. Die zugrundeliegende Erhebungsmethodik beruht jedoch auf der Dokumentenanalyse, welche den Umstand der mangelnden Aktualität der verwendeten Quellen mit sich bringt. Des Weiteren besteht bei dieser Erhebungsmethodik die Problematik der Informationsqualität in den verwendeten Nachweisen. Dem wurde jedoch durch Nutzung von VHB-gerankten Quellen bzw. durch Validierung mit weiteren Nachweisen entgegengewirkt.

Die Implementierung des Paging-Subsystems erfolgte unter dem Gesichtspunkt der Übersichtlichkeit und einer möglichst geringen Komplexität. Es gibt daher diverse Ansätze, die Teilelemente des Subsystems effizienter zu gestalten. So ist bspw. die Verwaltung der Auslagerungsdateien nicht optimal. Mit einem anderen Aufbau der Dateien bzw. mit einem optimierten Zugriff würde sich die Anzahl der Schreib- und Leseoperationen eventuell noch reduzieren lassen. Darüber hinaus gilt es zu prüfen, ob ggf. die Verwendung einer Auslagerungspartition mit modifiziertem Dateisystem sinnvoll wäre. Der Page Fault Handler weist ebenso noch Defizite in der Funktionalität auf. So wird bspw. das `dirty`-Bit im Seiten-Deskriptor derzeit noch nicht berücksichtigt. Des Weiteren würden Seitenersetzungsalgorithmen wie bspw. der „Least Recently Used“ oder der „WSClock“ besser bzw. effizienter arbeiten, als der in dieser Implementierung verwendete NRU-Algorithmus [Tan09, S. 260, S. 267 ff.]. Der NRU wurde jedoch aufgrund seiner geringen Komplexität für diese Implementierung herangezogen. Für ein grundsätzliches Verständnis der Seitenersetzung in späteren Vorlesungen dürfte ein einfacher Algorithmus von Vorteil sein. Die iterative Umsetzung des Algorithmus ist auf den frühen Entwicklungsstand des Betriebssystems Ulix zurückzuführen. Zudem lässt sich noch anmerken, dass die Routinen zur Fehlerbehandlung innerhalb der Funktionen Verbesserungspotenzial enthalten. Andererseits würde dies den Quellcode unübersichtlicher gestalten und das Verständnis der Funktionsweise beeinträchtigen.

5.2 Weiterer Forschungsbedarf

Diese Arbeit hat mit der Paging-Komponente für Ulix einen Beitrag zum Forschungsprojekt von Hans-Georg Eßer geleistet. In Bezug auf die in den Kapiteln 1.2 und 1.3 erarbeitete Forschungslücke gilt es jedoch noch zu evaluieren, inwiefern sich Literate Programming zur Vermittlung der Funktionsweise von komplexeren Programmen (wie bspw. dem Betriebssystem Ulix) eignet. Die vollumfängliche Bearbeitung dieser Fragestellung würde über den Rahmen der vorliegenden Arbeit hinausgehen. Dieser Themenbereich wird im eingangs erwähnten Forschungsprojekt zur Dissertation von Hans-Georg Eßer „Design, Implementation, and Evaluation of the ULIX-i386 Teaching Operating System“ näher betrachtet. Im Zuge dessen wird auch das hier entwickelte Paging-Subsystem den Studenten in Vorlesungen näher gebracht. Inwiefern das Literate Programming die Lehrbarkeit dieses Subsystems beeinflusst, ließe sich dann bspw. anhand von Meinungsumfragen oder Lernfortschrittskontrollen evaluieren.

noweb - Programmdefinitionen

Literaturverzeichnis

[Ber03] Bergsmann, J. (2003): Vorgehensmodelle, in: Quality Newsletter 2003, Ausgabe 2003/2,

[Bro90] Brown, M., Czejdo, B. (1990): A hypertext for literate programming, in: Advances in Computing and Information - ICCI '90 Lecture Notes in Computer Science 1990, Volume 468, S. 250-259

[Bro09] Brocke, J., Simons, A., Niehaves, B., Riemer, K., Plattfaut, R., Cleven, A. (2009): Reconstructing the Giant: On the importance of rigour in documenting the literature search process, in: European Conference on Information Systems 2009, Jg. 17

[Coo88] Cooper, H. (1988): Organizing Knowledge Syntheses: A Taxonomy of Literature Reviews, in: Knowledge in Society 1988, Vol. 1, S. 104-126

[Ehs05] Ehses, E., Köhler, L., Riemer, P., Stenzel, H., Victor, F. (2005): Betriebssysteme - Ein Lehrbuch mit Übungen zur Systemprogrammierung in UNIX/Linux, München 2005

[Enq07] Enquobahrie, A., Cheng, P., Gary, K., Ibanez, L., Gobbi, D., Lindseth, F., Yaniv, Z., Aylward, S., Jomier, J., Cleary, K. (2007): The Image-Guided Surgery Toolkit IGSTK: An Open Source C++ Software Toolkit, in: Journal of Digital Imaging 2007, Vol. 20, Suppl. 1, S. 21-33

[Gen03] Gentle, J. (2003): Random Number Generation and Monte Carlo Methods, 2. Ausgabe, New York 2003

[Hoe04] Hömberg, K., Jodin, D., Leppin, M. (2004): Methoden der Informations- und Datenerhebung, Dortmund 2004

[Hur96] Hurst, A. (1996): Literate Programming as an aid to marking student assignments, in: ACSE '96 Proceedings of the 1st Australasian conference on Computer science education 1996, S. 280-286

[Int87] Intel Corporation (1987): Intel 80386 Programmer's Reference Manual 1986, Santa Clara 1987

[Knu84] Knuth, D. (1984): Literate Programming, in: The Computer Journal 1984, Vol. 27, Nr. 2, S. 97-111

[Leh93] Lehner, F. (1993): Quality control in software documentation - Measurement of text comprehensibility, in: Information & Management 1993, Nr. 25, S. 133-146

[Lei02] Leisch, F. (2002): Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis, in: Härdle, W., Rönz, B. (Hrsg), Compstat - Proceedings in Computational Statistics, Heidelberg 2002, S. 575-580

[LiT12a] Li-Thiao-Té, S. (2012): Literate Program Execution for Reproducible Research and Executable Papers, in: Procedia Computer Science 2012, Vol. 9, S. 439-448

[LiT12b] Li-Thiao-Té, S. (2012): Literate Program Execution for Teaching Computational Science in: Procedia Computer Science 2012, Vol. 9, S. 1723-1732

[Mol08] Molloy, J. (2008): JamesM's kernel development tutorials, URL: http://www.jamesmolloy.co.uk/tutorial_html/6.-Paging.html, Abruf am 13.05.2014

[Pep91] Pepper, P. (1991): Literate Program Derivation - A Case Study, in: Methods of Programming Lecture Notes in Computer Science 1991, Vol. 544, S. 101-124

[Ram91] Ramsey, N., Marceau, C. (1991): Literate Programming on a Team Project, in: Software: Practice and Experience 1991, Vol. 21, Issue 7, S. 677-683

[Ram94] Ramsey N. (1994): Literate Programming Simplified, in: IEEE Software 1994, Vol. 11, Nr. 5, S. 97-105

[Ram01] Ramsey N. (2001): notangle-Manpage, Teil des noweb-Pakets, Version 2.11b-3, URL: http://manpages.ubuntu.com/manpages/hardy/man1/notangle.1.html, Abruf am 24.05.2014

[Rie97] Riedemann, H. (1997): Testmethoden für sequentielle und nebenläufige Software-Systeme, Wiesbaden 1997

[Rie09] Riebisch, M., Bode, S. (2009): Software-Evolvability, in: Informatik Spektrum 2009, Vol. 32, Nr. 4, S. 339-343

[Ruf08] Ruf, W., Fittkau, T. (2008): Ganzheitliches IT-Projektmanagement, München 2008

[Shu95] Shum, S., Cook, C. (1995): Using Literate Programming to Teach Good Programming Practices, in: SIGSE Bulletin: The Papers of the Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education 1995, S. 66-70

[Sim96] Simons, M., Weber, M. (1996): An Approach to Literate and Structured Formal Developments, in: Formal Aspects of Computing 1996, Vol. 8, S. 86-107

[Spo98] Spotnitz, R. (1998): Literate Programming, in: Computers & Chemical Engineering, 1998, Vol. 22, Nr. 12, S. 1745-1747

[Sta05] Stallings, W. (2005): Operating Systems - Internals and Design Principles, 5. Ausgabe, New Jersey 2005

[Tan09] Tanenbaum, A. (2009): Moderne Betriebssysteme, 3. Auflage, München 2009

[Zen91] Zeng, Y. (1991): Literate Programming System CDS, in: Journal Of Computer Science & Technology 1991, Vol. 6, Nr. 3, S. 263-270

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde / Prüfungsstelle vorgelegen hat. Ich erkläre mich damit einverstanden, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hoch geladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

Lenggries, den 28.05.2014 Florian Knoll