HOCHSCHULE OSNABRÜCK
UNIVERSITY OF APPLIED SCIENCES

Institut für Duale Studiengänge

Prüfungsleistung im Modul Betriebssysteme
des Studiengangs Wirtschaftsinformatik

Memory Management

am Beispiel XINU

Eingereicht von:	Matthias Fischer(700643) Jonathan Hermsen(723517)
Studiengruppe:	15DWF1
Betreuer:	Prof. Dr.-Ing. Ralf Westerbusch
Modul:	Betriebssysteme
Abgabedatum:	19.06.2017

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

Abkürzungsverzeichnis

BIOS	Basic-Input-Ouput-System
CPU	Central Processing Unit
DRAM	Dynamic RAM
FIFO	First-In-First-Out
LRU	Least-Recently-Used
MMU	Memory Management Unit
NRU	Not-Recently-Used
RAM	Random Access Memory
ROM	Read-Only-Memory
SRAM	Static RAM
TLB	Translation Lookside Buffer

1 Einleitung

Um die Geschwindigkeit und Sicherheit des Betriebssystem zu gewährleisten benötigt das Betriebssystem die sogenannte Speicherverwaltung. Diese hat das Ziel den vorliegenden Speicher im System möglichst Effizient auszunutzen, ohne dabei ungewollt Daten oder Befehle zu modifizieren. Des Weiteren wird eine Speicherverwaltung benötigt, wenn das System mehrere Prozesse gleichzeitig ausführen soll. Die folgende Ausarbeitung widmet sich dem Thema der Speicherverwaltung von Betriebssystemen und nimmt dabei Bezug auf das Betriebssystem Xinu. Im Verlauf der Ausarbeitung wird zunächst das benötigte Vorwissen vermittelt. Anschließend folgt eine Einführung in das Thema Memory Management. Danach sollen verschiedene Modelle und Konzepte des Memory Managements gezeigt und erläutert werden. Diese werden jeweils durch Codebeispiele aus Xinu unterstützt.

2 Theoretischen Grundlagen

Ein typischer Rechner nach der Von-Neumann-Architektur besteht aus einer Zentraleinheit und Peripheriegeräten.[1] Die Zentraleinheit umfasst die wesentlichen Komponenten der Hauptplatine. Eine essentielle Komponente der Hauptplatine ist die Central Processing Unit (CPU), da sie für die Steuerung und Verwaltung der Hardware zuständig ist.[2] Der Random Access Memory (RAM)-Speicher ist ein flüchtiger Speicher, der zur Laufzeit die Programmbefehle sowie deren Daten enthält.[3] Damit die Hardware-Komponenten nach dem Start des Rechners überprüft werden können und anschließend das Betriebssystem gestartet werden kann, befindet sich im Read-Only-Memory (ROM)-Speicher das Basic-Input-Ouput-System (BIOS).[4] Auf der Hauptplatine befindet sich außerdem ein Bus-System und Schnittstellen für Peripheriegeräte, welche die Kommunikation zwischen den Komponenten ermöglichen.[5] Die Peripheriegeräte bieten dem Anwender die Möglichkeit mit dem Rechner zu interagieren. Diese Art von Geräten lassen sich teilweise in die drei Gruppen Eingabe, Ausgabe und Massenspeicher unterteilen.[6] Typische Eingabegeräte sind beispielsweise die Maus und die Tastatur. Typische Ausgabegeräte ist der Drucker, die Grafikkarte und der Bildschirm. Jedoch gibt es auch Peripheriegeräte die sich nicht eindeutig einteilen lassen, wie die Festplatte und der USB-Stick, die jeweils als Ein- und Ausgabegerät, sowie auch als Massenspeicher fungieren. Der Fokus dieser Ausarbeitung liegt auf der Hauptspeicherverwaltung. Der Hauptspeicher, auch Arbeitsspeicher genannt, wird benötigt da die Register der CPU stark begrenzt sind.[7] Register sind prozessorinterne Speicherplätze und dienen der CPU zur Ausführung von Operationen.[8] Die Befehle und Daten der Anwendungen finden demnach

[1] [Herold et al., 2012, S. 92]
[2] [Herold et al., 2012, S. 95]
[3] [Herold et al., 2012, S. 96]
[4] [Herold et al., 2012, S. 96]
[5] [Herold et al., 2012, S. 96]
[6] [Herold et al., 2012, S. 118]
[7] [Herold et al., 2012, S. 107]
[8] [Herold et al., 2012, S. 98]

nicht vollständig in den Registern der CPU platz und müssen in einem größeren Speicher, dem Arbeitsspeicher, ausgelagert werden.[9] Heutige Hauptspeicher bestehen aus Halbleiterspeichern die als RAM bezeichnet werden.[10] Alle RAM-Bausteine sind flüchtige Speicher mit der Eigenschaft, dass deren Inhalte byteweise gelesen, sowie beschrieben werden können.[11] Des Weiteren wird noch zwischen den RAM-Varianten Static RAM (SRAM) und Dynamic RAM (DRAM) unterschieden.[12] Der statische RAM-Speicher hält mit Hilfe eines klassischen Flip-Flop-Gatter aus Transistoren die Informationen bis die Betriebsspannung am Speicher abfällt.[13] Der Vorteil der SRAM-Speicher liegt in der geringen Zugriffszeit, welche jedoch durch den hohen Stromverbrauch und der hohen Herstellungskosten relativiert wird.[14] Aufgrund der Nachteile werden SRAM-Speicher nur für Prozessorregister oder schnelle Cache-Speicher verwendet.[15] DRAM-Speicher verlieren aufgrund der Nutzung von Kondensatoren schon nach wenigen Millisekunden ihre Informationen.[16] Aus diesem Grund muss der Speicherinhalt periodisch aufgefrischt werden.[17] Mit der Folge, dass eine höhere Zugriffszeit benötigt wird. Aufgrund der billigeren Herstellung und des geringeren Stromverbrauches wird der Arbeitsspeicher meistens als DRAM realisiert.[18]

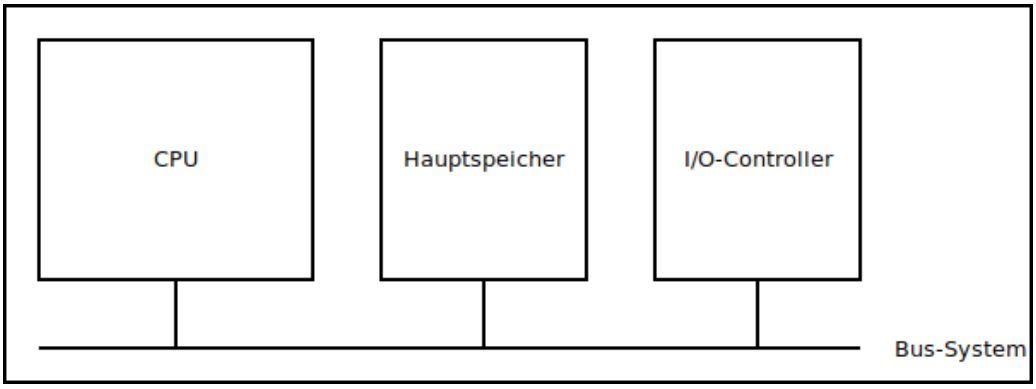

Abbildung 1: Rechneraufbau nach der Von-Neumann-Architektur

In dem folgenden Kapitel wird erläutert wie das Betriebssystem die Verwaltung des Hauptspeicher bewältigt, dabei mögliche Fehler abfängt und behandelt.

3 Memory Management

Damit Anwendungen von der CPU ausgeführt werden können, muss das Betriebssystem der Anwendung einen Bereich des Hauptspeichers zuteilen. In diesem Speicherbereich befinden sich bei einem

[9] [Herold et al., 2012, S. 107]
[10] [Herold et al., 2012, S. 108]
[11] [Herold et al., 2012, S. 108]
[12] [Häberlein, 2011, S. 182]
[13] [Häberlein, 2011, S. 182]
[14] [Herold et al., 2012, S. 108]
[15] [Herold et al., 2012, S. 108]
[16] [Häberlein, 2011, S. 183]
[17] [Häberlein, 2011, S. 183]
[18] [Herold et al., 2012, S. 109]

Rechner mit der Von-Neumann-Architektur einerseits die Befehle, sowie die Daten der Anwendung. Diese Zuteilung und Verwaltung des Speichers sind Aufgaben des Memory Managements und werden von dem Memory Manager des Betriebssystems durchgeführt. Der Begriff Speicherverwaltung umfasst im Prinzip auch die Verwaltung der Cache-Speicher in der CPU, da diese jedoch häufig durch die Hardware direkt verwaltet werden, liegt die Fokus dieser Ausarbeitung auf der Verwaltung des Hauptspeichers.

3.1 Ohne Speicherabstraktion

Das einfachste Modell der Speicherverwaltung besitzt keine Speicherabstraktion. Das heißt, dass Prozesse direkt auf die Speicheradressen des Hauptspeichers zugreifen.[19] Der maximale Speicherverbrauch eines Prozesses ist in diesem Modell durch den physikalischen Hauptspeicher begrenzt, da ohne Speicherverwaltung ausschließlich der gesamte Prozess als ein Ganzes in den Hauptspeicher geladen werden kann.[20] Angewendet wird dieses Modell beispielsweise in Embedded Systems, da die Komplexität der Implementierung sehr gering ist.[21] Jedoch birgt der unkontrollierte Zugriff der Anwendungen auf den physikalischen Speicher auch ein Risiko, da die Anwendung die Daten und Befehle des Betriebssystem überschreiben kann.[22] Außerdem ist zu erwähnen, dass ohne Swapping des Hauptspeichers eine simultane Ausführung von mehreren Anwendungen nicht möglich ist.[23] Swapping bezeichnet hierbei die Auslagerung des gesamten Speichers einer Anwendung aus dem Hauptspeicher auf ein anderes Medium wie beispielsweise einer Festplatte.[24] Das Thema Swapping wird im Folge der Ausarbeitung noch weiter erläutert.

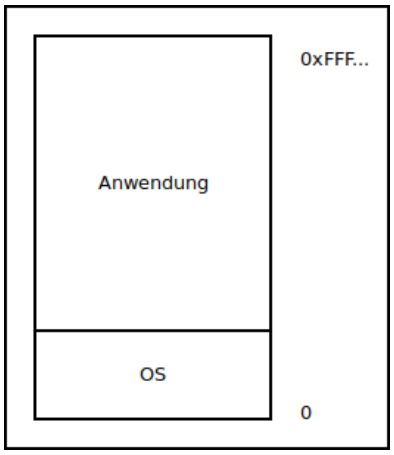

Abbildung 2: Hauptspeicher ohne Speicherabstraktion

[19] [Tanenbaum and Bos, 2016, S. 239]
[20] [Tanenbaum and Bos, 2016, S. 239]
[21] [Tanenbaum and Bos, 2016, S. 242]
[22] [Tanenbaum and Bos, 2016, S. 241]
[23] [Tanenbaum and Bos, 2016, S. 241]
[24] [Tanenbaum and Bos, 2016, S. 241]

3.2 Adressräume

Ein Modell das die simultane Ausführung von Prozessen ermöglicht ist die Speicherabstraktion Adressräume. Die Idee dieses Modells ist die Allokation eines Speicherbereiches zu einem Prozess und der dynamischen Relokation der Speicherbereiche.[25] Einem Prozess wird beispielsweise der Adressbereich 128 bis 192 zugewiesen und einem anderen Prozess der Bereich 192 bis 224. Nun besteht das Problem, dass der Prozess wissen müsste an welche Stelle im Hauptspeicher geladen wird, damit Zugriffe und Sprünge der Anwendung auf die richtigen Speicheradressen verweisen.[26] Das Problem löst das Adressraum Modell mit Basis- und Limitregistern.[27] Die Entwickler benutzten bei der Programmierung einer Anwendung einen Adressraum von 0 bis zur maximale Speichergröße.[28] Sobald nun der Prozess ausgeführt wird, wird die Startadresse des Prozesses in den Basisregister und die Größe des Prozesses in das Limitregister der CPU geschrieben.[29] Bei der Ausführung einer Operation wird nun zu jeder Adresse der Basisregister addiert.[30] Das Ergebnis dieser Addition bildet die physikalische Adresse im Hauptspeicher ab.[31] Um Beispielsweise die physikalische Adresse des Befehls der Anwendung 1 aus der folgenden Abbildung zu ermitteln, muss die Startadresse (in der CPU befindet sich die Adresse im Basisregister) der Anwendung mit der Adresse aus dem Adressraum der Anwendung addiert werden. Falls die angegebene Adresse größer ist als der Wert des Limitregisters, erfolgt ein Betriebssystem Interrupt und das Betriebssystem beendet den Prozess.[32] Jedoch besteht auch bei der Implementierung von Adressräumen mithilfe von Basis- und Limitregistern immer noch das Problem, dass der maximale Speicherbedarf eines Prozesses von dem physikalischen Speicher begrenzt wird.[33] Eine Ausführung einer Anwendung oder mehrere simultan laufende Anwendungen die in Summe einen höheren Hauptspeicherverbrauch haben als der physikalische Speicher es zulässt, ist nicht möglich.[34]

3.3 Swapping

Eine Strategie die es erlaubt mehrere Prozesse trotz fehlendem Hauptspeicherplatzes auszuführen ist das Swapping.[35] Es bezeichnet die Auslagerung von Prozessen von dem Hauptspeicher auf die Festplatte, um Hauptspeicherplatz zu gewinnen.[36] Beispielsweise soll in der folgenden Abbildung der Prozess 4 ausgeführt werden. Derzeit ist der Hauptspeicher mit den Daten und Befehlen des Betriebssystem sowie der Prozesse 1, 2 und 3 vollständig gefüllt. Demnach kann der Prozess 4 nicht ohne eine Auslagerung eines oder mehrerer Prozesse geladen werden. Deswegen werden zunächst die Prozesse 2 und 3 auf die Festplatte ausgelagert, anschließend der Prozess in den freien Speicherbereich kopiert und die

[25] [Tanenbaum and Bos, 2016, S. 244]
[26] [Tanenbaum and Bos, 2016, S. 244]
[27] [Tanenbaum and Bos, 2016, S. 244]
[28] [Tanenbaum and Bos, 2016, S. 244]
[29] [Tanenbaum and Bos, 2016, S. 244]
[30] [Tanenbaum and Bos, 2016, S. 244]
[31] [Tanenbaum and Bos, 2016, S. 244]
[32] [Tanenbaum and Bos, 2016, S. 244]
[33] [Tanenbaum and Bos, 2016, S. 244]
[34] [Tanenbaum and Bos, 2016, S. 244]
[35] [Tanenbaum and Bos, 2016, S. 245]
[36] [Tanenbaum and Bos, 2016, S. 245]

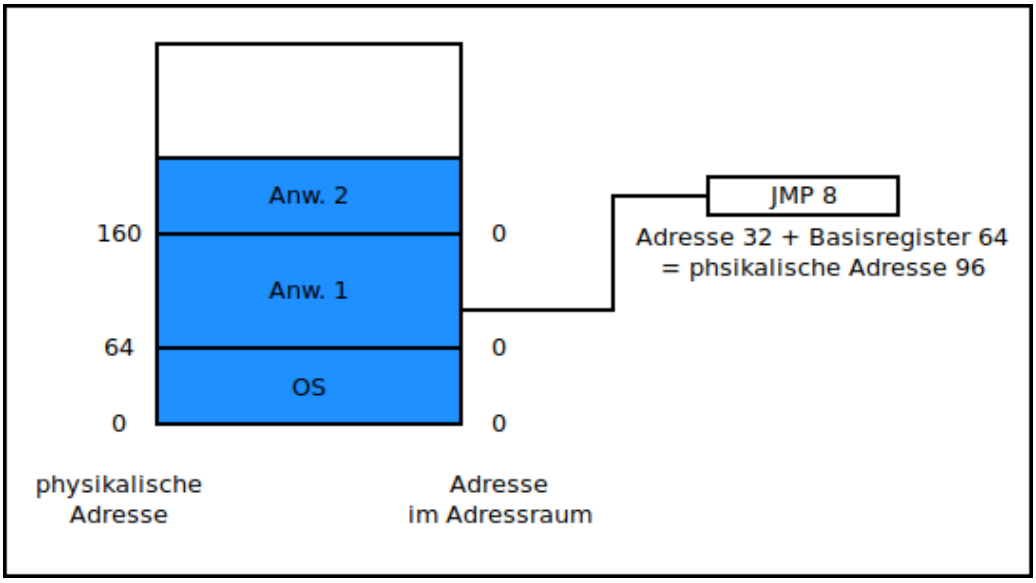

Abbildung 3: Hauptspeicher mit Adressräumen implementiert mittels Basis- und Limitregistern

Befehle des Prozesses 4 ausgeführt. Durch die Ein- und Auslagerung von Prozessen können Lücken im Speicher entstehen, welche mit der Technik Speicherverdichtung zu großen Lücken zusammengefasst werden können, in dem alle Prozesse so weit wie möglich nach unten geschoben werden.[37] Da Prozesse während der Laufzeit wachsen können kann es vorkommen, dass Prozesse in andere größere Speicherlücken umgelagert werden müssen.[38] Falls keine passende Lücke vorhanden ist, kommt es notfalls zur Auslagerung anderer Prozesse.[39] Diese Relokationen können mit hilfe eines Puffers in einigen Fällen verhindert werden. Der Wachstum eines Prozesses kann einerseits Aufgrund wachsender Daten und andererseits aufgrund wachsender Anzahl von Rücksprungadressen entstehen.[40] Falls ausschließlich dynamische Daten vorhanden sind, sollten der Puffer oberhalb angesetzt werden, sodass die Daten nach oben hin wachsen können.[41] Falls beide Wachstums-Varianten beachtet werden müssen, sollte ein Stack mit lokalen Variablen und Rücksprungadressen im oberen Teil des Speicherbereiches und die dynamischen Daten im unteren Teil alloziert werden.[42] Der freien Speicher befindet sich in dieser Variante mittig.

Die Kombination aus Adressräumen und Swapping erlaubt es mehrere Prozesse, die gemeinsam nicht in den Hauptspeicher passen, parallel auszuführen. Jedoch kann das Betriebssystem auch mit dieser Abstraktion keine Prozesse ausführen, deren Speicherverbrauch größer als der physikalische Speicher ist.

[37] [Tanenbaum and Bos, 2016, S. 246]
[38] [Tanenbaum and Bos, 2016, S. 247]
[39] [Tanenbaum and Bos, 2016, S. 247]
[40] [Tanenbaum and Bos, 2016, S. 247]
[41] [Tanenbaum and Bos, 2016, S. 247]
[42] [Tanenbaum and Bos, 2016, S. 247]

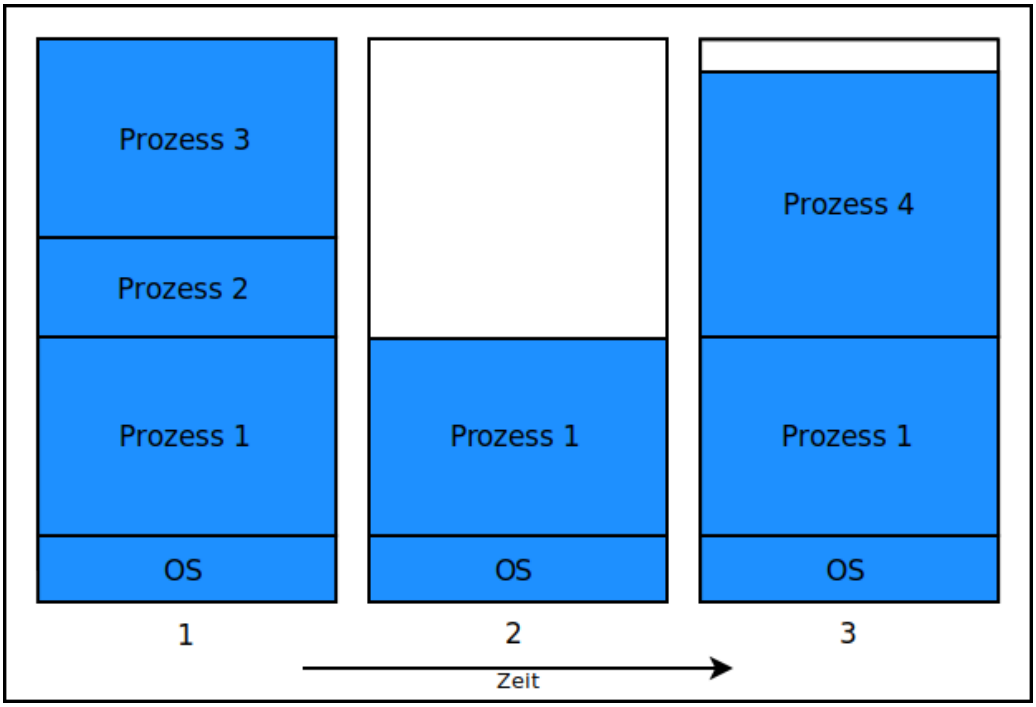

Abbildung 4: Darstellung des Swapping von Prozessen

3.4 Verwaltung von freiem Speicher

Um den freien physikalischen Speicher möglichst effizient auszunutzen, gibt es verschiedene Arten der Speicherverwaltung. Diese Zielen darauf ab, möglichst wenige und kleine Lücken im Speicher zu lassen, um den Anteil an ungenutztem Speicher so klein zu halten wie es möglich ist.

Vier der wichtigsten Optionen sind die Anordnung als Stack, eine verkettete Liste, eine Bitmap und das Buddy System. In Xinu werden von diesen der Stack standardmäßig und die verkettete Liste auf Anfrage genutzt.[43]

3.4.1 Stack

Bei der Nutzung von Stacks werden die möglichen Speicheradressen als Stapel gespeichert. Sollte eine Adresse benötigt werden. So wird die oberste Adresse vom Stapel gezogen und verwendet. Umgekehrt wird eine Adresse, die frei geworden ist, wieder oben auf den Stapel gelegt.[44]

[43] [Comer, 2015, S. 157]
[44] [Comer, 2015, S. 165]

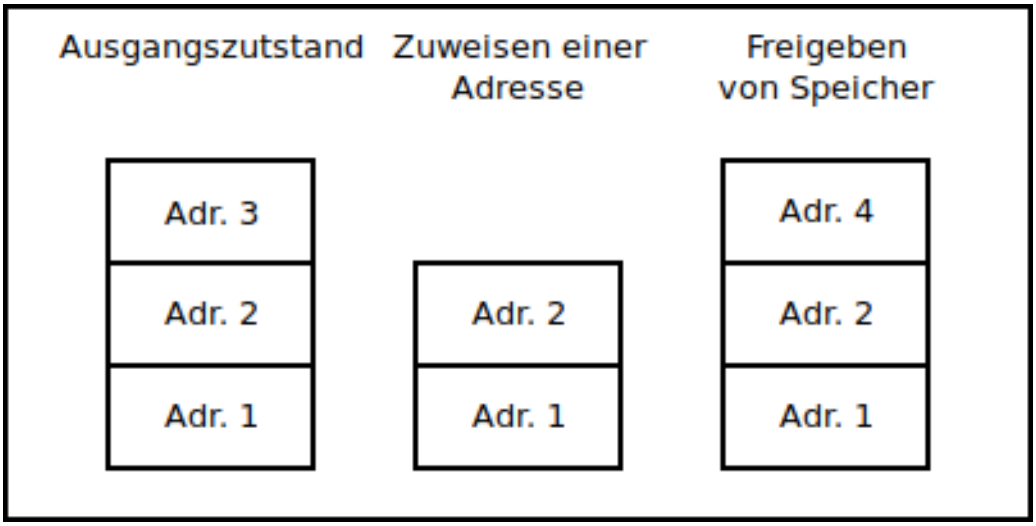

Abbildung 5: Darstellung eines Stacks für die Verwaltung von freiem Speicher

3.4.2 Verkettete Liste

Wenn der Speicher mit Hilfe einer verketteten Liste organisiert werden soll, so werden in dieser Liste jeweils ein Zeiger auf den Startpunkt und die Länge des Blocks gepflegt. Diese Liste kann nun auf der Suche nach einem passenden Speicherplatz durchlaufen werden.

Dabei gibt es mehrere Möglichkeiten:[45]

- First fit: Die Liste wird von Anfang an durchlaufen und der erste Block, der groß genug ist, wird genutzt.
- Next Fit: Die Liste hat einen Iterator. Die Suche nach passenden Speicherblöcken wird nicht immer vom Anfang der Liste aus durchgeführt sondern vom letzten Ort, an dem sich der Iterator befand.
- Best Fit: Die Liste wird komplett durchlaufen. Danach wird der Block genutzt, dessen größe der des zu speichernden Elementes am nächsten ist, ohne diese zu unterschreiten. Dadurch bleibt der geringstmögliche Speicher ungenutzt.
- Worst Fit: Es wird die Liste komplett durchlaufen. Nun wird jedoch der größte Speicherblock gesucht, damit die nach dem Speichern übrige Speicherkapazität des Blocks groß genug ist, um noch einen nutzbaren Block zu bilden.
- Quick Fit: Lücken ähnlicher Größen werden in verschiedenen Listen gespeichert. Ein Speicherbereich wird dann über diese Liste zugewiesen.

[45] [Tanenbaum and Bos, 2016, S. 250–251]

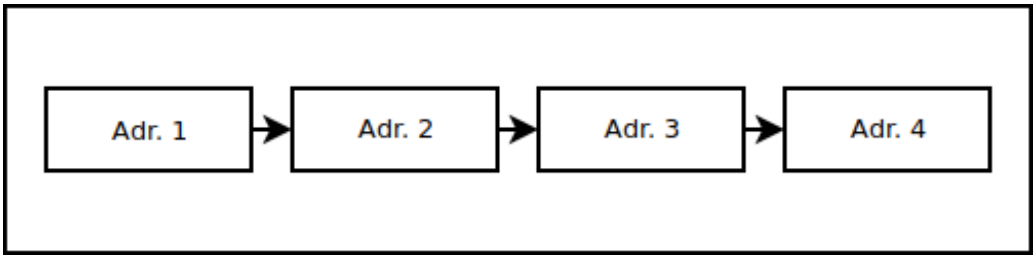

Abbildung 6: Darstellung einer verketteten Liste für die Verwaltung von freiem Speicher

3.4.3 Bitmap

Bei der Speicherverwaltung über Bitmaps wird jedem Speicherblock in einer Bitmap ein Bit zugeordnet. Dieses zeigt dann an, ob der betreffende Speicher im Moment belegt ist, oder nicht.[46]

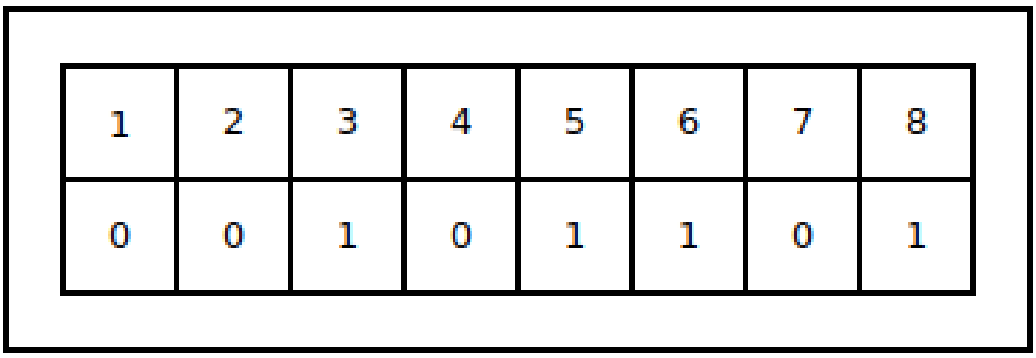

Abbildung 7: Darstellung einer Bitmap für die Verwaltung von freiem Speicher

3.4.4 Buddy-System

Das Buddy-System teilt den Speicher in verschiedene Blöcke von mehreren festgelegten Größen auf, meist als Zweierpotenzen. Sollte dann ein Element abgespeichert werden, so wird ein Block der bestmöglichen vorhandenen Größe genutzt, um möglichst wenig Speicher ungenutzt gelassen.

Wenn nun allerdings sämtliche Blöcke einer Speichergröße belegt sind, diese Größe aber benötigt wird, so ist es möglich, einen Block der nächsthöheren Größe zu halbieren. Die beiden dadurch entstandenen kleineren Speicherblöcke werden Buddies genannt.[47]

3.5 Virtueller Speicher

Das Problem des begrenzten Speicherplatzes wird durch den sogenannten virtuellen Speicher gelöst. Dessen Idee ist es den Adressraum eines Prozesses in sogenannte Seiten(Pages) aufzuteilen.[48] Falls

[46] [Tanenbaum and Bos, 2016, S. 248–249]
[47] [Brause, 2017, S. 126]
[48] [Tanenbaum and Bos, 2016, S. 252]

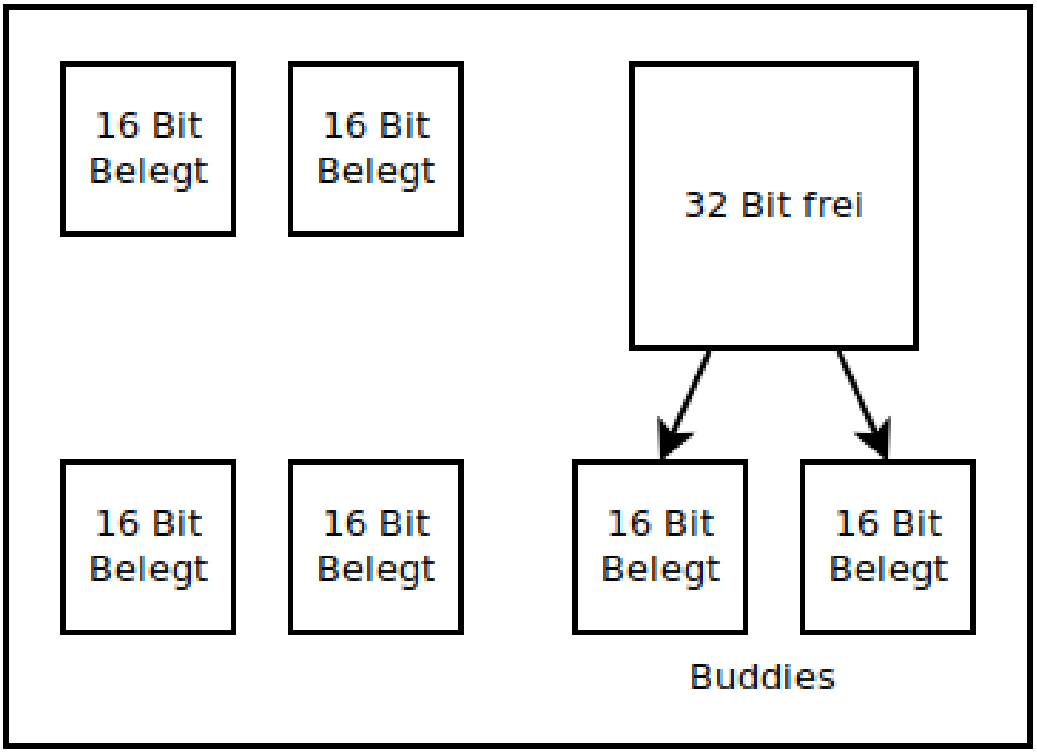

Abbildung 8: Darstellung eines Buddy-Systems für die Verwaltung von freiem Speicher

der Prozess einen Befehl einer nicht im Speicher vorhandenen Seite ausführen möchte, wird ein Betriebssystem Interrupt ausgelöst.[49] Daraufhin lädt das Betriebssystem die benötigte Page nach und führt den fehlgeschlagenen Befehl erneut aus.[50] Demnach muss nicht der gesamte Adressraum eines Prozesses im Hauptspeicher vorhanden sein und gleichzeitig können auch Anwendungen ausgeführt werden, die größer sind als der physikalische Hauptspeicher.

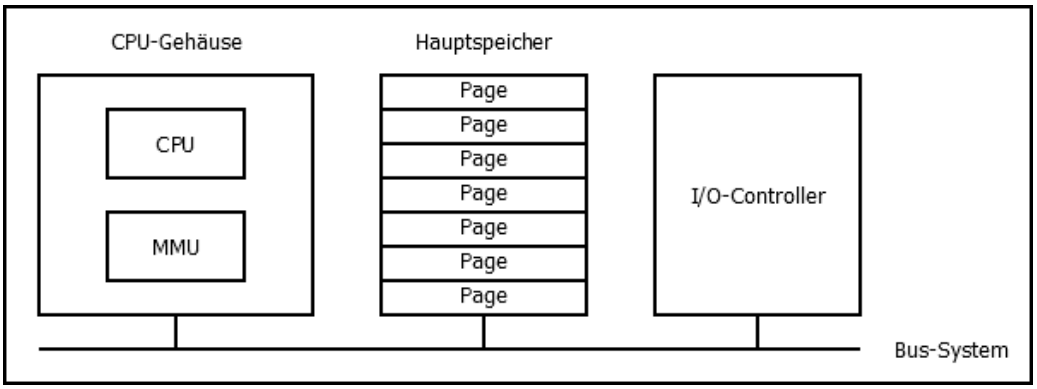

Abbildung 9: Abstraktionsmodell virtueller Speicher mit Pages

[49] [Tanenbaum and Bos, 2016, S. 252]
[50] [Tanenbaum and Bos, 2016, S. 252]

3.5.1 Paging

Seiten haben eine feste Speichergröße die oft die Größe des Seitenrahmens entspricht.[51] Seitenrahmen(page frame) sind die entsprechenden physikalischen Einheiten des RAMs.[52] Die Adressen der Anwendung werden auch als virtuelle Adressen bezeichnet und bilden in Summe den virtuellen Adressraum des Prozesses.[53] In einem System ohne Speicherabstraktion würden diese Adressen direkt angesprochen werden. Virtuelle Adressen werden von der CPU zunächst an die Memory Management Unit (MMU) geschickt, die diese Adresse dann mit Hilfe einer Seitentabelle zu der entsprechenden physikalischen Adresse umwandelt.[54] Da die Seiten eine einheitliche Größe haben, kann mit Hilfe der virtuellen Adresse erkannt werden, welche Seite für die Operation benötigt wird.[55] Ist angeforderte Seite nicht im Speicher vorhanden, wirft die MMU ein Interrupt.[56] Das Betriebssystem kopiert daraufhin die gewünschte Seite in den Hauptspeicher, aktualisiert die Seitentabelle und führt den fehlgeschlagenen Befehl erneut aus.[57]

3.5.2 Paging Table

Die virtuelle Adresse besteht aus einer virtuellen Seitennummer und einem Offset für die Adresse in einer Seite.[58] Als Index der Tabelle wird die Seitennummer genutzt zu der dann die entsprechende Seitenrahmennummer, das Present-/Absent-Bit und weitere Steuerungs-Flags gespeichert werden.[59] Das Present-/Absent-Bit gibt an ob die Seite sich aktuell im Speicher befindet oder nicht.[60] Die weiteren Flags können beispielsweise Zugriffsrechte auf die Seitenrahmen repräsentieren und somit Zugriffe auf gesperrte Seitenrahmen verhindern.[61] Die Seitennummer der virtuellen Adresse wird mit der Seitenrahmennummer ersetzt und ergibt damit die physikalische Adresse.[62] Da es bei der Übersetzung einer virtuellen Adresse durch die MMU zu vielen Zwischenschritten kommt und somit die Performance beeinträchtigt wird, wurde der Translation Lookside Buffer (TLB) entwickelt.[63] Der TLB ist häufig ein Teil der MMU und ermöglicht es Übersetzungen ohne Zugriff auf die Seitentabelle durchzuführen.[64] Der Puffer enthält Einträge über die Seiten, deren Informationen enthalten grundsätzlich die gleichen Informationen wie die der Seitentabelle.[65] Zusätzlich enthalten die Einträge noch die virtuelle Seite und ein Gültigkeits-Flag, welches die Aktivität der Seite repräsentiert.[66] Wird nun eine virtuelle

[51] [Tanenbaum and Bos, 2016, S. 253]
[52] [Tanenbaum and Bos, 2016, S. 253]
[53] [Tanenbaum and Bos, 2016, S. 253]
[54] [Tanenbaum and Bos, 2016, S. 253]
[55] [Tanenbaum and Bos, 2016, S. 253]
[56] [Tanenbaum and Bos, 2016, S. 254–255]
[57] [Tanenbaum and Bos, 2016, S. 254–255]
[58] [Tanenbaum and Bos, 2016, S. 257]
[59] [Tanenbaum and Bos, 2016, S. 257]
[60] [Tanenbaum and Bos, 2016, S. 255]
[61] [Tanenbaum and Bos, 2016, S. 258]
[62] [Tanenbaum and Bos, 2016, S. 257]
[63] [Tanenbaum and Bos, 2016, S. 260–261]
[64] [Tanenbaum and Bos, 2016, S. 260–261]
[65] [Tanenbaum and Bos, 2016, S. 260–261]
[66] [Tanenbaum and Bos, 2016, S. 260–261]

Adresse in die MMU zur Übersetzung eingegeben, vergleicht die MMU parallel alle Einträge mit der virtuelle Adresse.[67] Wird ein Eintrag im TLB gefunden, kann die virtuelle Adresse ohne Zugriff auf die Seitentabelle übersetzt werden.[68] Andernfalls lädt die MMU den entsprechenden Eintrag aus der Seitentabelle und ersetzt diesen mit einen älteren Eintrag aus dem TLB.[69] Der nächste Zugriff auf die Seite ist danach wieder ohne einen Selektion der Seitentabelle möglich.[70]

3.5.3 Seitenersetzungsalgorithmen

Bei jedem Seitenfehler muss das Betriebssystem, falls kein Platz im Hauptspeicher vorhanden ist, eine Seite auslagern um Platz für die gewünschte Seite zu machen.[71] Hierbei gibt es mehrere Algorithmen die entscheiden welche Seite ausgelagert werden soll.[72] Der optimale Algorithmus wäre einer der überprüft von welchen der derzeit eingelagerten Seiten die wenigsten noch auszuführenden Befehle enthält beziehungsweise die längste Wartezeit bis zur nächsten Ausführung eines Befehls besitzt.[73] Dieser Algorithmus ist jedoch in der Realität nicht realisierbar, da das Betriebssystem nicht weiß auf welche Seite demnächst zugegriffen wird.[74] Der Not-Recently-Used (NRU)-Algorithmus nutzt die zwei Statusbits R und M.[75] Das R-Bit wird gesetzt sobald ein Schreib- oder Lesezugriff auf eine Seite stattfindet.[76] Wiederum das M-Bit gesetzt wird falls Daten in der Seite modifiziert wurden.[77] Mithilfe eines Timer-Interrupts setzt das Betriebssystem zyklisch die R-Bits zurück.[78] Falls nun ein Seitenfehler auftritt und eine Seite ausgelagert werden muss klassifiziert das Betriebssystem alle vorhandenen Seiten in vier Klassen.[79] Klasse 0 sind Seiten die nicht referenziert und nicht modifiziert wurden. Klasse 1 sind Seiten die nicht referenziert und modifiziert wurden. Klasse 2 sind Seiten die referenziert und nicht modifiziert wurden. Klasse 4 sind Seiten die referenziert und modifiziert wurden.[80] Das Betriebssystem entfernt mit dem NRU-Algorithmus nun eine zufällige Seite aus der niedrigsten nicht leeren Klasse.[81] Bei dem First-In-First-Out (FIFO)-Algorithmus verwaltet das Betriebssystem die im Speicher vorhandenen Seiten als verkettete Liste. Die erste Seite der Liste wird entfernt und die neue Seite wird am Ende der Liste angefügt.[82] Der Second-Chance-Algorithmus funktioniert auch nach dem FIFO Prinzip mit dem Unterschied, dass bei der Auslagerung einer Seite überprüft wird ob die Seite das R-Bit gesetzt hat.[83] Wurde die Seite referenziert, wird das R-Bit zurückgesetzt, die Seite wie eine neue Seite an das Ende der Kette verschoben und anschließend nach einer neuen Seite zum auslagern

[67] [Tanenbaum and Bos, 2016, S. 260–261]
[68] [Tanenbaum and Bos, 2016, S. 260–261]
[69] [Tanenbaum and Bos, 2016, S. 260–261]
[70] [Tanenbaum and Bos, 2016, S. 260–261]
[71] [Tanenbaum and Bos, 2016, S. 267]
[72] [Tanenbaum and Bos, 2016, S. 267]
[73] [Tanenbaum and Bos, 2016, S. 268]
[74] [Tanenbaum and Bos, 2016, S. 268]
[75] [Tanenbaum and Bos, 2016, S. 269]
[76] [Tanenbaum and Bos, 2016, S. 269]
[77] [Tanenbaum and Bos, 2016, S. 269]
[78] [Tanenbaum and Bos, 2016, S. 269]
[79] [Tanenbaum and Bos, 2016, S. 270]
[80] [Tanenbaum and Bos, 2016, S. 270]
[81] [Tanenbaum and Bos, 2016, S. 270]
[82] [Tanenbaum and Bos, 2016, S. 270]
[83] [Tanenbaum and Bos, 2016, S. 271]

gesucht.[84] Der Least-Recently-Used (LRU)-Algorithmus folgt der Annahme, dass eine Seite die in den letzten Operationen häufig gebraucht wurde, sehr wahrscheinlich in den folgenden Operationen ebenso benötigt wird.[85] Daraus lässt sich schließen, dass lang ungenutzte Seiten auch länger benötigt bleiben und demnach ausgelagert werden können.[86] Die Realisierung des LRU-Algorithmus ist mit Hilfe von verketteter Listen möglich.[87] Jedoch werden wenige Rechner nach diesem Schema realisiert, da die Aktualisierung der Liste einen hohen Rechenaufwand benötigt.[88] Eine Alternative zu diesem ist der durch Software implementierte Not-Frequently-Used-Algorithmus, bei dem bei jedem Timerinterrupt ein Softwarezähler pro Seite mitzählt, ob das R-Bit gesetzt wurde oder nicht.[89]

3.6 Segmentierung

Das Paging erlaubt es mehrere Anwendungen auszuführen welche einzeln oder in Summe größer sind als der physikalische Speicher des Systems. Jedoch ermöglichen viele Programmiersprachen eine dynamische Allokation von Speicher. In dem normalen Paging-Modell könnte der Speicherbereich für beispielsweise einer Tabelle volllaufen und der Entwickler der Software müsste ein solches Problem abfangen.[90] Die Speicherabstaktion Segmentierung löst dieses Problem in dem es den Anwendungsspeicher in Einheiten mit freier Größe, den sogenannten Segmenten, unterteilt.[91] Jedes Segment besitzt seinen eigenen Adressraum mit Zugriffsrechten, welcher wachsen und schrumpfen kann.[92] Eine virtuelle Adresse besteht hierbei aus der Kombination der Segmentnummer und einer Adresse innerhalb des Segmentes.[93] Die Adressübersetzung der Segmentierung funktioniert analog zur der Adressübersetzung des Paging. Jedoch kann es bei dieser Variante zu einer großen Anzahl von kleinen Lücken im Speicher kommen, die für benötigte Segmente zu klein sind.[94] Diese Verschwendung des Speicherplatzes wird als externe Fragmentierung bezeichnet und kann durch die Speicherverdichtung verhindert werden.[95]

In der Praxis gibt es kaum Systeme die Segmentierung als Speicherabstraktion nutzen. Häufiger ist die Kombination aus Segmentierung und Paging, jedoch sind trotzdem die meisten Systeme mit der Speicherabstraktion Paging implementiert.

[84] [Tanenbaum and Bos, 2016, S. 271]
[85] [Tanenbaum and Bos, 2016, S. 272]
[86] [Tanenbaum and Bos, 2016, S. 272]
[87] [Tanenbaum and Bos, 2016, S. 272]
[88] [Tanenbaum and Bos, 2016, S. 272]
[89] [Tanenbaum and Bos, 2016, S. 273]
[90] [Tanenbaum and Bos, 2016, S. 302]
[91] [Tanenbaum and Bos, 2016, S. 303]
[92] [Tanenbaum and Bos, 2016, S. 303]
[93] [Tanenbaum and Bos, 2016, S. 304]
[94] [Tanenbaum and Bos, 2016, S. 306]
[95] [Tanenbaum and Bos, 2016, S. 306]

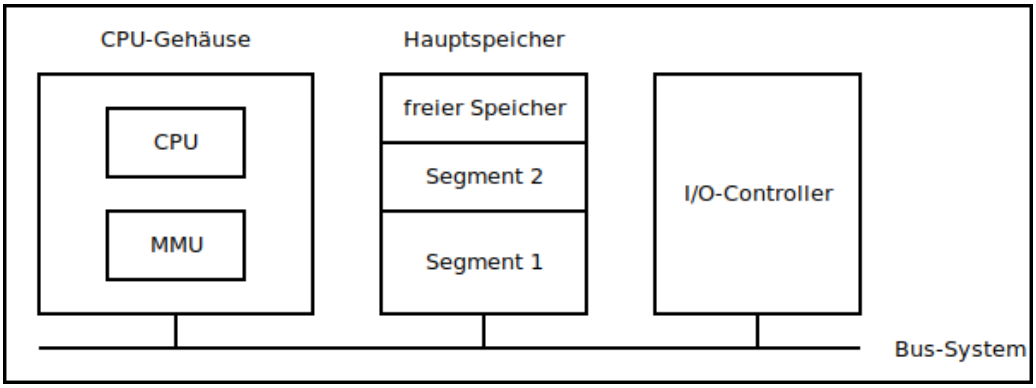

Abbildung 10: Abstraktionsmodell virtueller Speicher mit Segmenten

3.7 Buffer Pools

Das Betriebssystem Xinu nutzt keine virtuellen Adressräume, sondern Buffer Pools.[96] Hierbei wird der physikalische Speicher in Bereiche, den Buffer Pools, aufgeteilt.[97] Jeder Buffer Pool hat eine gleiche Anzahl von Speicherblocks mit derselben Größe.[98] Nach der Erstellung eines Buffer Pools wird der physikalische Speicher alloziert und kann nicht mehr vergrößert oder verkleinert werden.[99] Die Informationen des Buffer Pools werden in einer Systemtabelle *buftab* gespeichert.[100] Dessen Einträge bestehen aus der Buffer Pool-ID, der Buffer-Größe und einem Zeiger auf die Buffer-Liste.[101] Diese Tabelle wird während des Boot-Vorgangs des Betriebssystems erstellt.[102] Ein Prozess kann mit der Funktion *mkbuffpool* einen neuen Buffer Pool anlegen.[103] Hierbei fordert die Funktion die Anzahl der zu erstellenden Buffer und deren Größe.[104] Zunächst werden die Argumente der Funktion überprüft, falls die Werte nicht realisierbar sind folgt eine Fehlermeldung.[105] Andernfalls berechnet die Funktion die gesamte Größe des Buffer Pools und alloziert den Speicher.[106] Darauf erstellt die Funktion einen Eintrag in der Systemtabelle *buftab*.[107] Nachdem der Eintrag erstellt wurde, iteriert die Funktion über die Anzahl der Buffers und erstellt jeweils einen Eintrag in der Buffer-Liste des Buffer Pools.[108] Zuletzt gibt die Funktion die ID des Buffer Pools an den aufrufenden Prozess zurück.[109]

```
1 /**
2  * @file  bfpalloc.c
3  *
4  */
```

[96] [Comer, 2015, S. 176]
[97] [Comer, 2015, S. 176]
[98] [Comer, 2015, S. 176]
[99] [Comer, 2015, S. 177]
[100] [Comer, 2015, S. 177]
[101] [Comer, 2015, S. 177]
[102] [Comer, 2015, S. 183]
[103] [Comer, 2015, S. 181]
[104] [Comer, 2015, S. 181]
[105] [Comer, 2015, S. 181]
[106] [Comer, 2015, S. 181]
[107] [Comer, 2015, S. 181]
[108] [Comer, 2015, S. 181]
[109] [Comer, 2015, S. 181]

```
5  /* Embedded Xinu, Copyright (C) 2009. All rights reserved. */
6
7  #include <stddef.h>
8  #include <interrupt.h>
9  #include <memory.h>
10 #include <bufpool.h>
11
12 /**
13  * @ingroup memory_mgmt
14  *
15  * Acquire heap storage and subdivide into buffers.
16  *
17  * @param bufsize
18  *      Size of individual buffers, in bytes.
19  *
20  * @param nbuf
21  *      Number of buffers in the pool.
22  *
23  * @return
24  *      On success, returns an identifier for the buffer pool that can be passed
25  *      to bufget() or bfpfree(). On failure, returns ::SYSERR.
26  */
27 int bfpalloc(uint bufsize, uint nbuf)
28 {
29     struct bfpentry *bfpptr;
30     struct poolbuf *bufptr;
31     int id, buffer;
32     irqmask im;
33
34     bufsize = roundword(bufsize) + sizeof(struct poolbuf);
35
36     if (bufsize > POOL_MAX_BUFSIZE ||
37         bufsize < POOL_MIN_BUFSIZE || nbuf > POOL_MAX_NBUFS || nbuf < 1)
38     {
39         return SYSERR;
40     }
41
42     im = disable();
43     for (id = 0; id < NPOOL; id++)
44     {
45         bfpptr = &bfptab[id];
46         if (BFPFREE == bfpptr->state)
47         {
48             break;
49         }
50     }
51     if (NPOOL == id)
52     {
53         restore(im);
54         return SYSERR;
```

```
55      }
56      bfpptr->state = BFPUSED;
57      restore(im);
58
59      bfpptr->freebuf = semcreate(0);
60      if (SYSERR == (int)bfpptr->freebuf)
61      {
62          bfpptr->state = BFPFREE;
63          return SYSERR;
64      }
65
66      bfpptr->nbuf = nbuf;
67      bfpptr->bufsize = bufsize;
68      bufptr = (struct poolbuf *)memget(nbuf * bufsize);
69      if ((void *)SYSERR == bufptr)
70      {
71          semfree(bfpptr->freebuf);
72          bfpptr->state = BFPFREE;
73          return SYSERR;
74      }
75      bfpptr->next = bufptr;
76      bfpptr->head = bufptr;
77      for (buffer = 0; buffer < nbuf; buffer++)
78      {
79          bufptr->poolid = id;
80          bufptr->next = (struct poolbuf *)((ulong)bufptr + bufsize);
81          bufptr = bufptr->next;
82      }
83      signaln(bfpptr->freebuf, nbuf);
84
85      return id;
86  }
```

Listing 1: Funktion für die Buffer Pool Erstellung

Nachdem der Prozess nun einen Buffer Pool erstellt hat, kann dieser mit der Buffer Pool ID einen Buffer anfordern.[110] Dies geschieht über die Funktion *getbuf* mit der Buffer Pool ID als Argument.[111] *getbuf* funktioniert Synchron, dass heißt falls kein freier Buffer im Pool vorhanden ist, wartet das System solange bis ein Buffer freigegeben wird, gibt die Adresse des Buffers an den aufrufenden Prozess zurück und entfernt den Buffer aus der verketteten Liste des Buffer Pools.[112] Die Funktion liefert jedoch nicht direkt die erste Adresse des Buffers, sondern speichert in dieser die Pool ID und gibt anschließend die darauf folgende Adresse zurück.[113] Dieses Verfahren dient zur Identifizierung des Pools.[114]

```
1  /**
2   * @file bufget.c
```

[110] [Comer, 2015, S. 178]
[111] [Comer, 2015, S. 178]
[112] [Comer, 2015, S. 178]
[113] [Comer, 2015, S. 179]
[114] [Comer, 2015, S. 179]

```
3  *
4  */
5  /* Embedded Xinu, Copyright (C) 2009. All rights reserved. */
6
7  #include <stddef.h>
8  #include <semaphore.h>
9  #include <interrupt.h>
10 #include <bufpool.h>
11
12 /**
13  * @ingroup memory_mgmt
14  *
15  * Allocate a buffer from a buffer pool. If no buffers are currently available,
16  * this function wait until one is, usually rescheduling the thread. The
17  * returned buffer must be freed with buffree() when the calling code is
18  * finished with it.
19  *
20  * @param poolid
21  *        Identifier of the buffer pool, as returned by bfpalloc().
22  *
23  * @return
24  *        If @p poolid does not specify a valid buffer pool, returns ::SYSERR;
25  *        otherwise returns a pointer to the resulting buffer.
26  */
27 void *bufget(int poolid)
28 {
29     struct bfpentry *bfpptr;
30     struct poolbuf *bufptr;
31     irqmask im;
32
33     if (isbadpool(poolid))
34     {
35         return (void *)SYSERR;
36     }
37
38     bfpptr = &bfptab[poolid];
39
40     im = disable();
41     wait(bfpptr->freebuf);
42     bufptr = bfpptr->next;
43     bfpptr->next = bufptr->next;
44     restore(im);
45
46     bufptr->next = bufptr;
47     return (void *)(bufptr + 1);          /* +1 to skip past accounting structure */
48 }
```

Listing 2: Funktion für die Anforderung eines Buffers

Nachdem der Prozess seine Aufgaben bewältigt hat, muss dieser die allozierten Buffer wieder an den Pool zurückgegeben.[115] Dies ist mit der Funktion *freebuf* möglich.[116] Als Argument verlangt die Funktion den Zeiger auf den Buffer.[117] Mit diesem Zeiger liest die Funktion zunächst die Pool ID aus, um anschließend den Buffer wieder in der verketteten Liste des Buffer Pools einzufügen.[118] Zuletzt wird noch dem System mitgeteilt, dass wieder ein freier Buffer im Pool vorhanden ist.[119]

```
1  /**
2   * @file  buffree.c
3   *
4   */
5  /* Embedded Xinu, Copyright (C) 2009. All rights reserved. */
6
7  #include <stddef.h>
8  #include <semaphore.h>
9  #include <interrupt.h>
10 #include <bufpool.h>
11
12 /**
13  * @ingroup memory_mgmt
14  *
15  * Return a buffer to its buffer pool.
16  *
17  * @param buffer
18  *        Address of buffer to free, as returned by bufget().
19  *
20  * @return
21  *        ::OK if buffer was successfully freed; otherwise ::SYSERR. ::SYSERR can
22  *        only be returned as a result of memory corruption or passing an invalid
23  *        @p buffer argument.
24  */
25 syscall buffree(void *buffer)
26 {
27     struct bfpentry *bfpptr;
28     struct poolbuf *bufptr;
29     irqmask im;
30
31     bufptr = ((struct poolbuf *)buffer) - 1;
32
33     if (isbadpool(bufptr->poolid))
34     {
35         return SYSERR;
36     }
37
38     if (bufptr->next != bufptr)
39     {
```

[115] [Comer, 2015, S. 179]
[116] [Comer, 2015, S. 179]
[117] [Comer, 2015, S. 179]
[118] [Comer, 2015, S. 180]
[119] [Comer, 2015, S. 180]

```
40          return SYSERR;
41      }
42
43      bfpptr = &bfptab[bufptr->poolid];
44
45      im = disable();
46      bufptr->next = bfpptr->next;
47      bfpptr->next = bufptr;
48      restore(im);
49      signaln(bfpptr->freebuf, 1);
50
51      return OK;
52  }
```

Listing 3: Funktion für die Freigabe eines Buffers

Xinu verwendet ein rudimentäres Modell zur Verwaltung des Hauptspeichers. Die Implementierung besitzt keine Schutzmechanismen auf fremde Speicheradressen und bietet auch keine Möglichkeit Anwendungen die mehr Speicherplatz benötigen als der physikalische Speicher es zulässt auszuführen. Jedoch muss beachtet werden, dass das Betriebssystem für Lernzwecke entwickelt wurde und den lernenden auf einfache Art und Weise die Funktionsweise der Betriebssysteme erläutern soll.

4 Kritische Reflexion

Die Methode der Wahl war eine Literaturrecherche. Zunächst war angedacht, diese noch durch das implementieren von auf einem Raspberry Pi mit einem praktischen Bezug zu unterstützen. Dies erwies sich jedoch als schwierig, da die Portierung des Betriebssystem auf den Raspberry Pi noch in der Entwicklungsphase ist. Des Weiteren bietet eine Literaturrecherche nur eine Momentaufnahme der aktuellen Gegebenheiten. Es kann sich mit der Zeit noch viel im Feld der Speicherverwaltung tun, das von dieser Arbeit noch nicht abgedeckt werden kann. Außerdem war es nicht einfach, Literatur zu finden, welche das Memory Management speziell bei beschreibt. Aus diesem Grund musste an einigen Stellen auf Literatur zu Betriebssystemen im Allgemeinen zurückgegriffen werden.

5 Fazit

Es wird klar, dass das Memory Management in zeitgenössischen Betriebssystemen eine wichtige Rolle spielt. Das Memory Management bestimmt ob eine Ausführung mehrerer Prozesse möglich ist und wirkt sich auch direkt auf die Geschwindigkeit des Rechners aus.

Das Betriebssystem Xinu verfolgt in der Speicherverwaltung einen spartanischen Ansatz, es wird dem Entwickler ein rudimentäres Werkzeug zur Verfügung gestellt mit dem die Implementierung einfacher Anwendungen möglich ist. Jedoch bergen diese Funktionen keinen Schutz vor Modifizierungen fremder

Adressen. Da das Betriebssystem jedoch nicht für den alltäglichen Gebrauch, sondern für die Vermittlung von Wissen über Betriebssysteme konzeptioniert wurde, sind diese Schwächen nicht sonderlich relevant. Im Gegenteil das Betriebssystem erfüllt seinen Zweck vollständig.

Literaturverzeichnis

Rüdiger Brause. *Betriebssysteme: Grundlagen und Konzepte*. Springer Vieweg, Berlin, 4., erweiterte auflage edition, 2017. ISBN 978-3-662-54099-2.

Douglas Comer. *Operating system design: The Xinu approach*. Chapman & Hall/CRC, London, 2. ed. edition, 2015. ISBN 978-1-4987-1243-9.

Tobias Häberlein, editor. *Technische Informatik*. Vieweg+Teubner, Wiesbaden, 2011. ISBN 978-3-8348-1372-5.

Helmut Herold, Bruno Lurz, and Jürgen Wohlrab. *Grundlagen der Informatik*. Always learning. Pearson, München, 2., aktualisierte aufl. edition, 2012. ISBN 978-3-86894-111-1.

Andrew S. Tanenbaum and Herbert Bos. *Moderne Betriebssysteme*. Always learning. Pearson, Hallbergmoos/Germany, 4., aktualisierte auflage edition, 2016. ISBN 978-3868942705.

Eidesstattliche Erklärung

Wir erklären an Eides statt, dass wir unsere Hausarbeit

Memory Management

selbstständig und ohne fremde Hilfe angefertigt haben und dass wir alle von anderen Autoren wörtlich übernommenen Stellen wie auch die sich an die Gedankengänge anderer Autoren eng anlehnenden Ausführungen unserer Arbeit besonders gekennzeichnet und die Quellen zitiert haben.

Ort, Datum	Unterschrift
Ort, Datum	Unterschrift