# Großer Beleg

## Fiasco
## Eine Neuimplementierung von L4 in C++

## Entwurf und Implementierung der Mapping-Datenbank

Lukas Grützmacher[1]
Technische Universität Dresden
Fakultät Informatik, Professur für Betriebssysteme

21. Oktober 1998

[1]eMail: `lukas.gruetzmacher@iname.com`

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Kapitel 1

# Einleitung

An der Professur Betriebssysteme der Technischen Universität Dresden wird an dem Betriebssystem-Projekt Drops gearbeitet. Das System setzt auf einen Mikrokern auf. Die zentrale Anforderung an das System ist Unterstützung von Anwendungen, die Quality-of-Service-Anforderungen stellen. Dazu gehören unter anderem Multimedia-Anwendungen.

Eine gute Implementierung eines Mikrokerns existiert mit L4. Doch L4 ist für die Mitarbeiter und Studenten hier an der Universität nicht wartbar und auf Grund der Lizenzbestimmungen nicht in dem Maße erweiterbar, wie es für die Weiterentwicklung von Drops notwendig ist. Außerdem sind die Echtzeitfähigkeiten nicht in allen Punkten zufriedenstellend. Deshalb entschied sich die Arbeitsgruppe der Professur, eine eigene Implementierung eines Mikrokernes nach dem Vorbild von L4 zu erstellen. Dabei wurde Wert darauf gelegt, das entstehende System, das den Namen Fiasco bekommen hat, in der Lehre der Studenten einsetzen zu können.

Das Konzept von L4 sieht vor, den Prozessen die Möglichkeit zu bieten, sich gegenseitig Speicherseiten zur Verfügung zu stellen. Für die Verwaltung der dadurch entstehenden Datenstruktur ist die Mapping-Datenbank verantwortlich, welche ich im Rahmen meines Großen Beleges entwickelt habe.

Die folgenden Kapitel beschreiben neben den Grundlagen meiner Arbeit (Kapitel 2) den Entwurf (Kapitel 3) und die Implementierung (Kapitel 4) der Mapping-Datenbank. Eine Analyse der Leistungsfähigkeit wird in Kapitel 5 geführt. Abschließend werden in Kapitel 6 und 7 die gewonnenen Ergebnisse bewertet und zusammengefaßt.

# Kapitel 2

# Grundlagen

## 2.1 Grundlagen Mikrokern

Die Idee eines Mikrokerns verbirgt sich bereits in diesem Namen. Es geht darum, den Kern des Betriebssystems auf ein absolutes Minimum zu reduzieren. Übrig bleiben dabei nur die Kommunikation zwischen den Prozessen (IPC[1]) sowie rudimentäre Speicher- und Prozeßverwaltung.

Vorteile dieser Implementierung sind die Überschaubarkeit und die Modularität. Auf Grund des relativ geringen Umfanges des Kerns[2] ist es leichter möglich, Fehler bei der Implementierung zu vermeiden. Außerdem kann auf jedem System, das auf einem Mikrokern basiert, die Wunschkonfiguration zusammengestellt werden, da die üblichen Funktionen eines Betriebssystems durch einen eigenen Server bereitgestellt werden können.

Somit eignet sich ein Mikrokern sowohl für den Betrieb eines Videoservers als auch als Basis von sogenannten *Embedded Systems*. Das sind Geräte, die durch Mikroprozessoren gesteuert werden, wie z.B. Waschmaschinen, moderne Fernsehgeräte.

## 2.2 Echtzeitsystem

Ein Echtzeitsystem hat die Besonderheit, neben der Zusicherung, eine Aufgabe erwartungsgemäß zu erfüllen, eine Zeitspanne zu garantieren, in der diese Aufgabe mit Sicherheit ausgeführt wird. Mit diesen Zusagen ist es dann möglich, Anwendungen, die periodisch eine gewisse Rechenzeit brauchen, in einen Task-Wechsel-Zyklus einzuplanen. Dazu übergibt das Programm ein Anforderungsprofil. Das Echtzeitsystem bestimmt nun die Rechenzeit für die einzelnen Befehlsfolgen und prüft, ob diese zu den gewünschten Zeitpunkten ausgeführt werden können, ohne mit den Rechenzeiten anderer Anwendungen zu kollidieren. Für diese Berechnung ist es nun notwendig, genau sagen zu können, wie lange ein Aufruf im System verweilt.

Für die Angabe der Zeiten unterscheidet man verschiedene Typen:

**Best Case:** Im *besten Fall* braucht das System die minimal mögliche Zeit, da der kürzeste Pfad durch den Algorithmus gegangen wird.

**Worst Case:** Im *schlechtesten Fall* wird der längste mögliche Weg (inklusive des mehrfachen Weges durch Schleifen) durchschritten und somit die maximale Zeitspanne benötigt.

Für die Berechnung der Zeit, in der die Anfrage an das System garantiert beantwortet wird, muß mit der Zeit des *Worst Case* gerechnet werden, denn es kann von außen kaum gesagt werden, welcher der möglichen Fälle eintritt.

[1] *Inter Process Communication*

[2] Der dem Projekt zugrundeliegende Mikrokern L4 hat eine Codegröße von 12KB.

Für Funktionen, die z.B. lediglich eine private Variable für die Außenwelt anbieten, ist die Berechnung des *Worst Case* einfach, da eben nur die Zeit für diese eine Aufgabe bestimmt werden muß.

Enthält eine Funktion jedoch Schleifen, wird die Berechnung schwieriger. Es ist notwendig, alle möglichen Bedingungen, die das Verweilen in der Schleife verlängern könnten, herauszufinden und deren Maximum zu bestimmen. Ist dies nicht möglich, kann eine Zusage über die Ausführungsdauer nicht gemacht werden, und das Echtzeitprinzip ist verletzt.

Für die Entwicklung meiner Datenbank bedeutet dies, immer zu garantieren, daß eine feste obere Schranke für die Bearbeitungsschritte feststellbar ist. Das heißt beispielsweise, beim Suchen eines Eintrags im Speicher muß jede mögliche Konstellation bedacht und ggf. mit einer Abbruchbedingung versehen werden.

## 2.3 Zweck der Datenbank

Der dem System zur Verfügung stehende Speicher wird beim Systemstart eines auf L4 basierenden Systems an einen Speicherverwalter (*Pager*) namens Sigma0 übergeben. Dieser blendet nun den gesamten Speicher in seinem eigenen virtuellen Adreßraum ein. Dabei entspricht die virtuelle der physischen Adresse. Je Kachel existiert nun ein Sigma0-Mapping (in der Abbildung 2.1 sind dies die Einträge S1, S2 bzw. S3).

Benötigt eine Task Speicher, so wird ihr von ihrem Pager[3] eine Seite zur Verfügung gestellt. Diesen Vorgang nennen wir Mappen. Dabei ist es möglich, diese Seite an einer beliebigen virtuellen Adresse dieser Task einzublenden. (Einblenden der Seite S2 in Task A an Adresse A2)

Jede Task kann nun die ihr eingeblendeten Seiten anderen Tasks zur Verfügung stellen. Somit haben alle beteiligten Tasks Zugriff auf den gleichen physischen Speicherbereich. Dies ermöglicht ihnen den Datenaustausch. (Mapping der Seite A2 in die Task C an Adresse C2 und Task D an Adresse D1 — gemeinsamer Zugriff auf die phys. Seite S2)

Anschaulich kann man sich für diese Verteilungsstruktur einen Baum vorstellen (siehe Abbildung 2.1). Die Wurzel eines Baumes bildet dabei das Sigma0-Mapping.

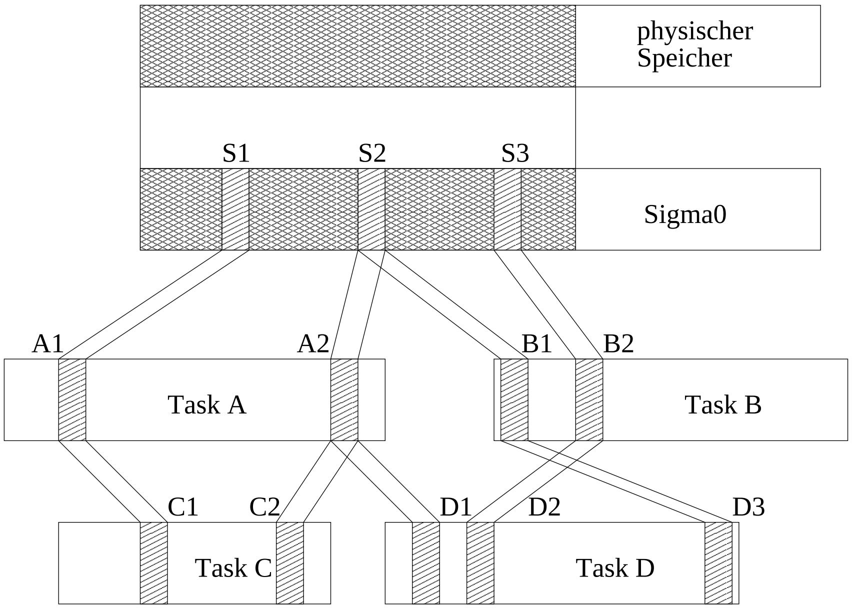


Abbildung 2.1: Veranschaulichung einer Mapping-Struktur

Wird einer Task T eine bestimmte Seite entzogen, so muß sichergestellt sein, daß auch

[3] Dies muß nicht Sigma0 sein.

allen anderen Tasks, die von T diese Seite bekommen haben, selbige auch wieder entzogen wird. Im Beispiel der Abbildung würde dies bedeuten, daß im Falle des Entzugs der Seite A2 auch die Seiten C2 und D1 entzogen werden müßten. Dagegen erzwingt der alleinige Entzug von C2 nicht den von D1.

Dieser Mechanismus tritt ebenso in Kraft, wenn eine der bis dato aktiven Tasks beendet und gelöscht werden soll. In dieser Situation müssen alle ihre Seiten und die davon ausgehenden Mappings den jeweiligen Tasks entzogen werden.

Um diese beiden Aufgaben effektiv realisieren zu können, ist eine Mapping-Datenbank notwendig, die genau diese Informationen der Verzweigungen speichert. Es wäre theoretisch auch denkbar, alle Seitentabellen zu durchsuchen, um herauszufinden, in welchen Adreßräumen eine bestimmte Kachel eingeblendet ist. Dies ist aber bei der Vielzahl von Seitentabellen äußerst zeitaufwendig und deshalb in einem Echtzeitsystem nicht brauchbar.

## 2.4 Überblick über andere Komponenten

Der folgende Abschnitt soll die Teile von Fiasco beschreiben, die ich nicht selbst entwickelt habe, aber für die Funktion der Datenbank benötige.

### 2.4.1 RMGR — Ressourcen-Manager

Der Ressourcen-Manager ist die erste Task, die beim Systemanlauf gestartet wird. Er ist für das Laden und Starten der ersten System-Prozesse zuständig. Zu diesen gehört der Fiasco-Betriebssystemkern, Sigma0 und die ersten auf den Kern aufsetzenden Server.

### 2.4.2 Sigma0

Sigma0[4], der erste und globale Speicherverwalter, ist eine der ersten Tasks, die vom RMGR gestartet werden. Vom RMGR bekommt Sigam0 auch den gesamten physischen Speicher zugeteilt und verwaltet ihn. Von Sigma0 bekommen die meisten Tasks nun ihren eigenen Speicher eingeblendet. Zu diesen gehören unter anderem "private" Pager, die den ihnen gegebenen Speicher anderen Prozessen zur Verfügung stellen können.

Nicht zuletzt bekommt auch der Kern seinen eigenen Speicherbereich zugeteilt. Die Aufgabe der Verwaltung übernimmt der *Kernel Memory Allocator* (KMem).

### 2.4.3 KMem — Kern-Seiten-Allokator

Damit der Kern selbst seine verschiedenen Aufgaben wahrnehmen kann, benötigt er eigenen Speicher. KMem sorgt nun dafür, daß dieser an die verschiedenen Komponenten des Kerns verteilt wird. So wendet sich auch die von mir zu entwickelnde Datenbank an diesen Allokator, um Speicher für die Verwaltung der Mapping-Einträge zu erhalten.

## 2.5 Schnittstellen

Von Michael Hohmuth[5], der das gesamte Projekt Fiasco leitet, wurden die Schnittstellen definiert, die ich realisieren sollte.

Die Datenbank besteht aus zwei C++-Klassen: `mapping_t` und `mapdb_t`. Erstere enthält die Daten eines Mapping-Eintrags (siehe 3.2). Die Schnittstellen werden in 2.5.2 vorgestellt.

Eine Instanz der Klasse `mapping_t` kann nur von der eigentlichen Datenbank in der Klasse `mapdb_t` erzeugt werden. Die Schnittstellen dieser Klasse werden in 2.5.1 beschrieben.

[4]sprich: Sigma Null

[5]eMail: `hohmuth@innocent.com`

### 2.5.1 Mapping-Datenbank

#### 2.5.1.1 `sigma_insert`

Parameter: page_id, type

Diese Methode erzeugt das erste Mapping der Kachel *page_id* vom physischen Speicher in den Sigma0-Pager. Als Mapping-Typ[6] wird *type* eingetragen.

#### 2.5.1.2 `insert`

Parameter: Vater, Virtuelle Adresse, TaskID, Mapping-Typ

Ausgehend von einem *Vater*-Eintrag wird ein neues Mapping als Kind eingetragen. Als charakterisierende Werte werden die *virtuelle Adresse* in einer bestimmten *Task* und der *Mapping-Typ* eingetragen. Alle bisher erzeugten Kinder vom *Vater* sind gleichwertig einzustufen, wie das neue Mapping.

#### 2.5.1.3 `grant`

Parameter: Virtuelle Adresse, TaskID, Mapping-Typ

Ein Task kann eine Seite, die in ihren Adreßraum eingeblendet ist, einer anderen zur Verfügung stellen und dabei auf die weitere Nutzung verzichten. Diesen Vorgang nennt man *grant*. Die charakteristischen Werte des neuen Mapping-Eintrags werden als Parameter übergeben.

#### 2.5.1.4 `lookup`

Parameter: Virtuelle Adresse, TaskID, Mapping-Typ

Diese Schnittstelle ermöglicht das Suchen nach einem Mapping, welches eindeutig durch die angegebenen Parameter beschrieben wird. Wenn ein Eintrag gefunden wurde, wird der Baum, in dem dieser Eintrag steht, für weitere Zugriffe gesperrt.

#### 2.5.1.5 `free`

Parameter: Mapping

Um die Sperre eines Baumes wieder freizugeben, muß diese Methode mit einem *Mapping* dieses Baumes als Parameter aufgerufen werden. Dies sollte jenes sein, welches man in der vorangegangenen Such- bzw. Einfügeoperation als Rückgabe erhalten hatte.

#### 2.5.1.6 `split`

Parameter: Mapping

Wenn einzelne kleine Seiten einer großen Seite an andere Tasks abgegeben werden sollen, muß diese in kleine Seiten aufgeteilt werden. Dazu wird als Parameter das große *Mapping* übergeben.

Diese Methode wird erst in einer späteren Version implementiert.

#### 2.5.1.7 `flush`

Parameter: Mapping, me_too

Soll ein Teilbaum einer Kachel gelöscht werden, so ruft man diese Methode mit dem *Mapping* auf, das die Wurzel dieses Teilbaumes ist. Der Parameter *me_too* gibt an, ob das angegebene Mapping selbst ebenfalls gelöscht werden soll oder nur dessen Kinder.

[6] ein Teil der zu speichernden Datenstruktur; siehe Abschnitt 3.2

### 2.5.2 Mapping-Einträge

#### 2.5.2.1 `space`

Parameter: keine
liefert den Index der Task zurück, für die dieses Mapping eingetragen wurde

#### 2.5.2.2 `vaddr`

Parameter: keine
liefert die virtuelle Adresse zurück, an die die Seite eingesetzt wurde

#### 2.5.2.3 `size`

Parameter: keine
liefert die Größe zurück, für die dieses Mapping steht

#### 2.5.2.4 `type`

Parameter: keine
liefert den Typ dieses Mappings zurück

#### 2.5.2.5 `parent`

Parameter: keine
sucht das Mapping, von dem aus das aktuelle erzeugt wurde — den Vater

#### 2.5.2.6 `next_iter`

Parameter: keine
Um über alle Mappings eines Baumes iterieren zu können, liefert diese Methode das nächste Mapping ausgehend von dem adressierten. Hat dieses Mapping keine Kinder, wird in den darüber liegenden Ebenen des Baumes nach weiteren Einträgen gesucht. (In Abb. 2.1 liefert `next_iter`(D1) den die Adresse des Eintrags von B1.)

#### 2.5.2.7 `next_child`

Parameter: Vater
Unter Angabe eines Vaters liefert diese Funktion das nächste Kind zurück. Sind keine weiteren Kinder des Vaters mehr vorhanden, sondern nur noch Geschwister, so ist dies am Rückgabewert erkennbar. (In Abb. 2.1 liefert `next_child`(D1) einen NULL-Zeiger.)

## 2.6 Stand der Technik

Im folgenden werde ich kurz zwei Implementierungen einer Mapping-Datenbank vorstellen: die Version des Original-L4 von Jochen Liedtke und die der Alpha-Version von Volkmar Uhlig [Uhli98].

### 2.6.1 Original-L4

Die Datenstruktur wird hier als binärer Baum abgespeichert. Jeder einzelne Eintrag erhält einen Zeiger auf den Seitentabellen-Eintrag, auf das erste Kind-Mapping sowie auf Vorgänger und Nachfolger unter seinen Geschwistern. Somit kommt man auf eine Strukturgröße von 16 Byte.

Zum Systemstart wird ein Teil des Hauptspeichers als Bereich für die Datenbank belegt. Der Anteil kann nur zur Compile-Zeit und nicht im laufenden Betrieb verändert werden. Dieser Speicherpool wird in kleine Zellen zerlegt, welche die einzelnen Einträge repräsentieren. Ein Teil dieser wird den Wurzel-Einträgen zugeordnet.

Während der Aktualisierung an Einträgen werden alle Unterbrechungen gesperrt, um gleichzeitigen Zugriff verschiedener Prozesse auf den gleichen Eintrag zu verhindern.

Zu kritisieren ist hier, daß dieses Konzept sehr unflexibel bei der Hauptspeichernutzung vorgeht. Es ist anzunehmen, daß der belegte Bereich für alle getesteten Anwendungen ausreicht. Dies bedeutet aber auch, daß im Regelfall ein großer Teil des Speichers ungenutzt bleibt. Ebenso ungünstig ist die Sperrung des gesamten Systems, während die Datenbank arbeitet.

### 2.6.2 Alpha-Version

Auch in dieser Implementation ist die Struktur im logischen Sinne als Baum abgespeichert. Dabei sind auch hier alle direkt verwandten Einträge in einer doppelt verketteten Liste verknüpft. Ein einzelner Eintrag hat eine Größe von 32 Byte und enthält neben den Zeigern auf die Nachbarn und den charakteristischen Daten des Mappings auch eine Sperrstruktur. Diese ermöglicht das Sperren von Teilbäumen, und beschränkt so den Zugriffsschutz auf einen minimalen Bereich der Datenbank.

Im Gegensatz zum Original-L4 wird kein fester Speicherbereich belegt. Es wird einfach an einer festgelegten Adresse begonnen, die einzelnen Einträge in den Speicher zu schreiben. Wenn hinter dieser virtuellen Adresse noch keine physische Seite eingeblendet ist, wird die Operation nur durch einen Page Fault und die entsprechende Behandlung verzögert.

## 2.7 Speicher-Allokatoren

Um die Verwaltung der Daten der Mapping-Datenbank zu vereinfachen und die Speicherausnutzung zu optimieren, soll ein Speicher-Allokator eingesetzt werden. [Vaha96] stellt verschiedene Strategien vor. In den folgenden Abschnitten habe ich diese Informationen zusammengefaßt.

### 2.7.1 Resource Map Allocator

Dieser Mechanismus ist der einfachste von den beschriebenen. Er geht von einem festen Speicherbereich aus, der über eine Freispeicherliste verwaltet wird. Es werden zwei Schnittstellen angeboten:

**offset_t rmalloc(size)** führt eine Anfrage für einen *size* Byte großen Speicherbereich durch.

**void rmfree(base, size)** gibt *size* Bytes ab Adresse *base* wieder frei.

Für die Belegung der Speicherzellen gibt es verschiedene Strategien:

**First Fit** belegt die erste Speicherzelle, die groß genug ist, die Anforderung zu erfüllen.

**Best Fit** belegt die kleinste aller Speicherzellen, welche die Anforderungen erfüllen können.

**Worst Fit** belegt einen Bereich der größten freien Zellen. Damit steigt die Wahrscheinlichkeit, daß der bleibende Rest auch noch eine weitere Anfrage erfüllen kann.

Da der Resource Map Allokator (RMA) die Möglichkeit bietet, festzulegen, wieviel Speicher eines Bereiches freigegeben werden soll, ist es dem Nutzer erlaubt, nur einen Teil seines zuvor belegten Speicherplatzes wieder freizugeben. (Das verhindert die Gefahr, daß nach der Freigabe des gesamten Bereiches die anschließende Neubeschaffung eines kleineren Bereiches nicht erfüllt werden kann, da inzwischen ein anderer Prozeß diesen Bereich zugeteilt bekommen hatte.) Dies ist einer neben den folgenden Vorteilen:

Der Algorithmus ist einfach zu implementieren.

Es können immer genau so viele Bytes angefordert werden, wie gebraucht werden, solange dies nicht die Gesamtgröße des Puffers übersteigt. Somit können Anfragen nach großen und kleinen Bereichen gleich gut erfüllt werden.

Nebeneinanderliegende freie Bereiche werden zu einem großen zusammengefügt.

Dennoch gibt es einige Nachteile, die den RMA nur für wenige Zwecke anwendbar machen:

Nach einer längeren Laufzeit mit unterschiedlich großen Anfragen und ebenso verschiedenen Nutzungszeiten ist der Puffer stark fragmentiert. Die entstehenden Lücken können natürlich für Anfragen nach kleinen Speicherplätzen genutzt werden. Für größere Speicherzellen muß jedoch ein neuer Puffer bereitgestellt werden.

Mit starker Fragmentation wächst auch die Frei-Liste. Ist sie ein statisch belegtes Feld, besteht die Gefahr eines Überlaufs bzw. erfordert eine Sonderbehandlung. Wird sie dagegen dynamisch verwaltet, braucht sie einen eigenen Allokator, und wir stehen immer wieder vor dem gleichen Problem.

Um zusammenhängende Speicherplätze zusammenzuführen, muß die Liste nach der Basisadresse sortiert sein. Sortieren ist aber bekanntermaßen eine zeitaufwendige Operation und für ein Echtzeitsystem (siehe Abschnitt 2.2) ungeeignet.

Zum Auffinden einer passenden Speicherzelle ist ein Durchsuchen der Frei-Liste erforderlich — bei optimierten Strategien wie *Best Fit* und *Worst Fit* sogar der ganzen Liste. Dies sind Aufgaben, die im Laufe der Zeit durch die steigende Fragmentierung immer aufwendiger werden; das System wird also langsamer.

### 2.7.2 Simple Power-of-Two Free Lists

Der Algorithmus "Simple Power-of-Two Free Lists" (SPTL) verwendet ein Feld von Frei-Listen. Dabei verwaltet jede Liste einen Pool von Puffern mit jeweils konstanter Größe. Der Name weist nun auf deren spezielle Größe hin: Exponenten von Zwei. Dies hat den Vorteil, eine physische Seite (oft 4096 Byte) ohne ungenutzte Reste in kleinere Puffer aufteilen zu können.

Jeder von den Puffern beginnt mit einem Header. Dessen Größe richtet sich nach der Adressbreite des Prozessors. (Da die meisten der aktuellen Prozessoren 32-Bit-Adressen verwenden, gehe ich auch im folgenden von 4 Byte Headergröße aus.) Dieser Header wird zur Verkettung der Puffer innerhalb der Listen verwendet. Ist der Puffer belegt, zeigt der Header auf die Liste, zu der dieser Puffer gehört.

Das Verfahren des Belegens und Freigebens ist erwartungsgemäß recht einfach: Beim Aufruf von `malloc` mit der gewünschten Speichergröße als Parameter wird der kleinste Puffer belegt, der die Anfrage gerade erfüllen kann. Dabei ist zu bedenken, daß bereits 4 Byte (oder mehr) vom Header belegt sind und von einem 32-Byte-Puffer nur 28 Byte übrig bleiben.

Bei der Freigabe braucht der Nutzer nur eine Funktion `free` mit der Basisadresse des Puffers aufrufen. Anhand des Header weiß der Allokator den Puffer wieder richtig einzureihen.

Für den Fall, daß die Anzahl der Puffer einer Größe erschöpft ist, gibt es viele Möglichkeiten, dieses Problem zu beheben. Mit einfacher Verweigerung der Anfrage und Belegung eines noch größeren Puffers seien hier nur zwei Beispiele genannt.

Als Vorteile, die SPTL gegenüber RMA bietet, sind die Beseitigung der Fragmentierung und des Problems des langwierigen Suchens nach einem passenden Puffer zu nennen. Dagegen ist die teilweise Freigabe eines Puffers nicht möglich.

Als wesentlicher Nachteil ist die Unflexibilität in Bezug auf die verwaltbaren Puffergrößen zu nennen. Wenn diese nicht genau $2^x - 4$ Byte sein sollen, bleibt immer ungenutzter Speicher übrig.

### 2.7.3 McKusick-Karels Allocator

Dieser Allokator ist eine Weiterentwicklung des SPTL. Er beseitigt das Problem, daß Anforderungen nach Speicherzellen, die genau die Größe von Exponenten von Zwei haben, mit einem doppelt so großen Puffer erfüllt werden muß. Dazu nutzt er mehrere zusammenhängende Seiten als Speicherpool. Der Zustand der einzelnen Seiten wird in einem Feld festgehalten. Als Zustände existieren die folgenden:

**frei** Das entsprechende Feldelement zeigt auf das nächste Element, daß eine leere Seite repräsentiert.

**unterteilt** Das Element enthält die Größe der Speicherzellen dieser Seite.

**Teil eines Puffers über mehrere Seiten** Das Feldelement zeigt auf die erste der Seiten, die diesen Puffer enthalten. Dort ist auch die Größe des Puffers abgespeichert.

Je Seite werden nur Puffer gleicher Größe gespeichert. So kann man auf die Verzeigerung der Puffer und somit auf den Header verzichten, da über die Basisadresse die Speicher-Seite und somit die zugehörige Frei-Liste identifiziert werden kann.

Doch es bleibt das Problem der Speicherverschwendung, wenn die gewünschte Größe nicht genau in eine Zelle paßt.

### 2.7.4 Buddy System

Der Buddy-System-Algorithmus vereint Ideen der Frei-Listen und dem Power-of-Two-System: ein großer Puffer wird solange halbiert, bis ein Puffer entsteht, der die Anfrage gerade erfüllen kann[7]. Geht man von einer physischen Seite mit 4096 Byte aus, erhält man z.B. 1024, 256 oder 32 Byte große Puffer.

Über eine Bitmap wird der Zustand der einzelnen Zellen verwaltet. Dabei entspricht jedes Bit einem Puffer der kleinsten gewünschten Größe. Man muß sich also bei der Implementierung entscheiden, welche Mindestgröße gebraucht wird. Klar ist, daß bei einer sehr feinen Granularität der notwendige Verwaltungsspeicher stark anwächst: Für eine 4KByte-Seite mit 4 Byte-Puffern ist eine 1024 Bit, also 128 Byte große Bitmap notwendig.

Dieser Algorithmus bringt den Vorteil, daß nebeneinanderliegende freie Speicherzellen anhand der Frei-Bits in der Bitmap leicht erkannt und wieder zu einer großen zusammengefügt werden können. Doch dieser Mechanismus birgt auch gleich wieder einen Nachteil in sich:

> Bei der Freigabe einer Zelle wird sofort geprüft, ob sie wieder mit ihrem Nachbarn zu einer größeren Zelle zusammengefügt werden kann. Dieser Vorgang wird solange fortgesetzt, bis kein Zusammenfügen mehr möglich ist. Wird gleich darauf eine neue Zelle der vorher freigegebenen Größe angefordert, muß die gerade entstandene große wieder mehrfach zerlegt werden. Ist die Lebensdauer einer Zelle nicht sehr lang, ist die Zeit, die das System mit der Verwaltung beschäftigt ist, möglicherweise größer als die Nutzzeit.

### 2.7.5 Lazy Buddy Algorithm

Um das Performanceproblem des Buddy Algorithmus zu lösen, wurde dieser zum Lazy Buddy Algorithm (LBA) verfeinert. Die Änderung tritt im Falle der Freigabe eines Puffers zu tage:

> Anhand der Formel[8] $slack = N - 2L - G$, die den Zustand des Pufferpools wiederspiegelt, wird entschieden, was zu tun ist. Es gibt folgende Zustände:

[7] Der Name "buddy" wurde vergeben, weil jede der entstehenden Hälften "buddy" genannt werden kann.

[8] N steht für die Gesamtzahl der zu diesem Zeitpunkt vorhandenen Puffer, L für die Puffer, die lokal aber nicht in der Belegungskarte als frei gekennzeichnet sind und G für die restlichen der freien Zellen. Mit A als Anzahl der belegten Puffer ergibt sich die Formel $N = A + L + G$, die zu jedem konsistenten Zeitpunkt gültig ist.

**lazy** $slack \geq 2$. Der Pool ist in einem guten Zustand; es sind genügend Puffer frei — das Zusammenfügen von Puffern ist nicht notwendig.

**reclaiming** $slack = 1$. Grenzfall — Wenn der gerade freigegebene Puffer mit seinem Nachbarn zusammengefaßt werden kann, wird dies getan, sonst folgt keine weitere Aktion.

**accelerated** $slack = 0$. der Zustand des Pools ist ungünstig — Es müssen Puffer gesucht werden, die zusammengefügt werden können.

(Der Algorithmus verhindert, daß *slack* einen Wert kleiner Null annimmt.)

Dieses Vorgehen verbessert zwar die Performance im allgemeinen. Es läßt sich von außen jedoch nie sagen, wie lange es dauern wird, bis die gewünschte Aktion ausgeführt ist. Der *Worst Case* — das Teilen des gesamten Puffers in die kleinste mögliche Einheit — ist zwar selten, muß aber bei der Echtzeitanalyse (siehe 2.2) berücksichtigt werden.

Außerdem ist das Problem der schlechten Speicherausnutzung durch die festgelegte Größe auf Exponenten von Zwei auch hier vorhanden.

### 2.7.6 Mach-OSF/1 Zone Allokator

Der Zone Allokator ist interessant, da er sich selbst zur Verwaltung seiner Daten verwendet. Er ist dazu geeignet, beliebig große Puffer bereitzustellen. Dabei wird für jede Objektart ein eigener Pool von Puffern — *Zone* genannt — erstellt, auch wenn zwei die gleiche Größe haben.

Die *Zone of Zones* ist die Zentralverwaltung mit einer Liste von Objekten, in denen die charakteristischen Daten einer jeden Zone gespeichert sind: *size*, *max* und *alloc*. (*size* steht für die Größe eines einzelnen Objekts, *alloc* für den Wert, um den eine *Zone* vergrößert werden soll, wenn der Pufferpool erschöpft wird. Dabei kann eine *Zone* nicht größer als *max* werden.)

Dieses Vorgehen verlangt die Anwendung einer "Garbage Collection", um den Anteil ungenutzten Speichers so gering wie möglich zu halten. Dazu muß im System eine weitere Task laufen, die CPU-Restzeiten nutzt, um alle *Zones* auf ihren Belegungszustand zu untersuchen und gegebenenfalls alle Objekte einer vollkommen freien Seite aus der Frei-Liste zu entfernen und die Seite an das Systems zurückzugeben.

Da die Objekte in der Frei-Liste nie effizient einzelnen Seiten zugeordnet werden können, ist es so schwierig, zuerst Objekte einer Seite zu belegen. Dazu wäre beispielsweise ein Sortieren der Frei-Liste notwendig, was, wie auch bei RMA (siehe 2.7.1), für Echtzeitsysteme nicht günstig ist.

### 2.7.7 Solaris 2.4 Slab Allocator

Der Slab Allokator verfolgt einen objektorientierten Ansatz. Je Datengröße und -typ wird ein Lager[9] geschaffen, das Speicherzellen dieser Größe bereitstellt, und diese nur über wohldefinierte Schnittstellen zuteilt.

Ein Lager kann aus einem oder mehreren Slabs bestehen. Für Dateneinheiten bis zur Größe einer Seite besteht ein Slab auch nur aus einer Seite. Ansonsten werden mehrere zusammenhängende Seiten (nach [Vaha96] 10) als Slab betrachtet und gemeinsam verwaltet.

Ein gewisse Anzahl von Bytes, die frei gewählt werden kann, wird am Anfang jedes Slab freigelassen. Dies bietet die Möglichkeit, die verschiedenen *Cache Lines* des Prozessors besser zu nutzen.

Ein weiterer Parameter bei der Erzeugung eines Lagers ist das *Alignment*. Erfordert es der Prozessor, daß auf die Datenobjekte immer 4-Byte-weise zugegriffen wird, so erzwingt dieser Parameter, die Speicherzellen entsprechend anzuordnen.

[9] In [Vaha96], ein Buch in englischer Sprache, wird die Bezeichnung Cache genutzt. Sie läßt sich mit "Lager" passend ins Deutsche übersetzten. Um Verwechslungen mit dem "Prozessorcache" zu verhindern, werde ich im folgenden den deutschen Begriff verwenden.

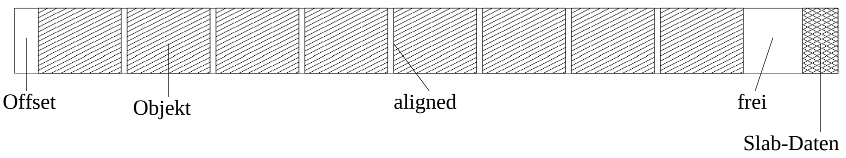


Abbildung 2.2: Seite, die als Slab eingerichtet ist

Die letzten 8 Byte eines Slabs werden allein für die Verwaltung desselben benötigt. Hier wird neben der Anzahl der noch freien Zellen und einem Zeiger auf die erste noch freie Zelle auch ein Zeiger auf einen weiteren Slab für dieses Lager gespeichert. Dies wird notwendig, wenn die Zahl der freien Zellen erschöpft ist.

| **11** | **21** | **32** |
|---|---|---|
| free entries | first free | next |

Abbildung 2.3: Datenstruktur zur Verwaltung eines Slab

Unter der Adresse der ersten freien Zelle ist die Adresse der nächsten gespeichert. Wird also die erste dem Klienten zur Verfügung gestellt, so wird vorher die Adresse der nächsten Zelle in den Verwaltungsbereich der Slab-Seite geschrieben.

Im Falle der Freigabe einer Zelle, wird an ihrem Beginn die Adresse der Zelle eingetragen, die im Verwaltungsbereich als erstes freies Element steht. Dort wiederum wird nun die Adresse der frei gewordenen Zelle registriert. Die Zuordnung eines Puffers zu einer bestimmten Seite erfolgt über ein Ausmaskieren der niederwertigen Bits der Adresse.

Hier zeigt sich, daß sich der Aufwand, den Überblick über freie und belegte Zellen zu behalten, über alle Anfragen verteilt. Die "Garbage Collection" ist somit als großer Vorteil des Slab Allokators zu werten, da die Ausführungszeiten der einzelnen Operationen des Allokators bestimmbar sind.

Dieses Konzept ist sehr flexibel einsetzbar. Große und kleine Objekte können nahezu gleich gut verwaltet werden. Einziger Nachteil ist der hohe Managementaufwand für Objekttypen, für die wenige Puffer benötigt werden. Es wäre denkbar, für diese Typen einen anderen Allokator einzusetzten, so z.B. den SPTL (siehe 2.7.2). In diesem Fall fällt dessen ungünstige Speicherausnutzung nur wenig ins Gewicht.

# Kapitel 3

# Entwurf

## 3.1 Anforderungen

Ziel meiner Arbeit ist es, eine möglichst kompakte und dennoch effiziente Datenstruktur zur Verwaltung der Mapping-Daten (siehe Abschnitt 2.3) zu entwickeln.

Bei der Ausarbeitung der Algorithmen für die Datenverwaltung durch die Mapping-Datenbank ist auf Echtzeitfähigkeit (siehe Abschnitt 2.2) aller Operationen zu achten.

Da Fiasco für die Lehre der Studenten eingesetzt werden soll, ist es wichtig, einen verständlichen Quelltext zu schreiben.

## 3.2 Datenstruktur

Ein Mapping-Eintrag ist durch folgende Daten gekennzeichnet:

- Virtuelle Adresse (VA), an die diese Seite gemappt werden soll
- zugehöriger Adreßraum (einer je Task)
- Größe (z.B. 4KB oder 4MB)
- Typ (Speichermapping, I/O-Mapping)
- Quell-Mapping

### 3.2.1 Der Eintrag

Da der Speicher immer seitenweise vergeben wird, sind die unteren 12 Bit einer VA immer Null und müssen somit nicht abgespeichert werden. Daraus ergibt sich für einen Zeiger auf die Seite ein Speicherbedarf von 20 Bit.

In L4 [Lied96] sind maximal 1024 gleichzeitig existierende Adreßräume vorgesehen. Deren Identifikator läßt sich mit 10 Bit kodieren.

Da es nur zwei Typen von Größen bzw. Typen an Mappings gibt, reicht jeweils 1 Bit zur Speicherung dieser Information.

Wie in Abschnitt 2.3 erklärt, ist die logische Struktur der Datenbank ein Wald von Bäumen (je Kachel ein Baum). Die übliche Speicherung solcher sogenannter *binärer* Bäume erfordert eine mindestens zweifache Verzeigerung: Nachfolger und Nachbar. In der Welt des 32-bitigen x86 benötigen diese beiden Zeiger zusammen 8 Byte. Die für das Mapping charakteristischen Daten benötigen aber nur 4 Byte. Zwei Drittel der ohnehin recht komplexen Datenbasis würden somit also nur für die Verknüpfung der Daten verbraucht. Da dies aber den Anforderungen an die Implementierung (siehe 3.1) widersprach, mußte auf eine komplette Verzeigerung der Mappingeinträge verzichtet werden. Es durfte dabei aber nicht die Zugehörigkeit der einzelnen Mapping-Einträge zueinander verloren gehen.

Es wurden nun verschiedene Möglichkeiten diskutiert.

**Kleine Zeiger** Als erstes wäre es denkbar, sich auf einen kleineren Zeiger zu beschränken. Dieser würde dann beispielsweise immer von der Basisadresse `0xD000 0000` gerechnet werden. Mit einer Reduktion auf 3 Byte für einen Zeiger könnte man einen Speicherbereich von 16 MByte adressieren. Dies ist zwar eine Größe, die für die Mapping-Struktur ausreichen würde, doch die Ersparnis beträgt nur 2 Byte.

Eine weitere Reduktion auf 2-Byte-Zeiger läßt aber nur einen adressierbaren Bereich von 64 KByte zu, was bei weitem nicht für die gesamte Datenbank ausreicht. Selbst bei einer relativen Verzeigerung wird eine aufwendige Sonderbehandlung notwendig, wenn im Umfeld des Vater-Mappings kein Platz mehr für das einzufügende Mapping existiert.

**Bäume im Kasten** Die nächste Variante ist, über die Tiefe des Mappings[1] die Zugehörigkeiten zu rekonstruieren. Die Mapping-Einträge für einen Mapping-Baum werden dabei kompakt (als Feld) in den Speicher geschrieben — beginnend beim Wurzel-Mapping, so daß der Nachfolger eines Mappings eine größere Tiefen-Zahl hat als das Mapping selbst. Nachbarn — Mappings mit gleicher Quelle — haben die gleiche Tiefen-Zahl.

Anhand der Abbildung 2.1 (auf Seite 7) möchte ich exemplarisch das Ergebnis dieses Vorgangs beschreiben. Zuerst wird als Wurzel das Sigma0-Mapping eingetragen (S2). Anschließend wird sein erster Nachfolger (A2) an die Liste angehängt. Bevor der nächste Nachfolger (B1) des Sigma0-Mappings angefügt wird, beginnt man, rekursiv alle Nachfolger von A2 an die Liste zu hängen. Dieses Vorgehen nennt man *pre-order traversal* und erhält so für diesen Baum die folgende Liste: A2 C2 D1 B1 D3 (siehe auch Abb. 3.1).

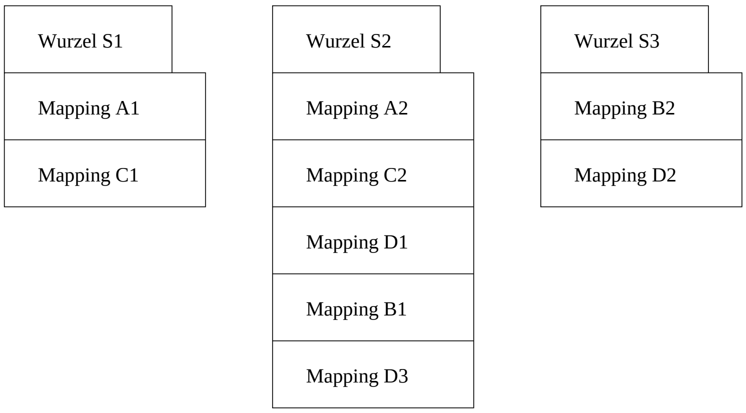


Abbildung 3.1: Mapping-Bäume von Abb. 2.1 als Liste gespeichert

Da eine hohe Schachtelungstiefe wenig sinnvoll ist[2], reicht ein Speicherplatz von 8 Bit vollkommen aus. Es ist auch möglich, einige wenige Bits anderen Daten (z.B. Sperrstruktur für Teilbäume) zur Verfügung zu stellen. Um die ohnehin nun notwendigen 5 Byte (4 Byte virtuelle Adresse und 1 Byte Tiefe) aber voll auszunutzen, wurden 8 Bit festgelegt.

Im Falle des Einfügens eines neuen Mappings in den Baum wird es in der Regel notwendig, Platz zu schaffen. Dies bedeutet im Mittel ein Verschieben der Hälfte der Mapping-Einträge. Daraus ergibt sich eine Komplexität von $O(n)$. Es läßt sich also nicht genau sagen, wie lange dieser Vorgang dauern wird.

**Indizes** Um das Manko des Verschiebens zu beseitigen, wäre eine Alternative denkbar, bei der jedes Mapping mit einem Index innerhalb des Baumes versehen wird. Dies bringt den Vorteil, jedes neue Mapping ohne etwaige Verschiebeoperationen hinter das letzte Mapping oder in einen zuvor frei gewordenen Bereich schreiben zu können und die Reihenfolge über

[1] Schachtelungstiefe im Baum bezüglich der Baumwurzel

[2] In der Praxis würde dies bedeuten, daß sehr viele Tasks eine einzige Seite unter sich verteilen, wobei jede einzelne Task sie nur an einen Nachbarn weitergibt. Sinnvoll ist eher, das eine Task eine Seite an viele Tasks weitergibt. So z.B. im Fall der *Shared Libraries*.

die Indizes zu erhalten. Man muß sich jedoch für eine gewisse Größe der Index-Variable entscheiden und steht so vor dem gleichen Problem wie bei der freien Verzeigerung. Hier würde zwar eine Größe von z.B. 64 KByte für einen Baum ausreichen, ist jedoch äußerst unflexibel und im Fall von kleinen Bäumen verschwenderisch. Außerdem folgt daraus unmittelbar eine notwendige Sonderbehandlung in dem Fall, daß der Baum größer als der adressierbare Datenbereich wird. Desweiteren ist zu bedenken, daß zum Auffinden jeweils zusammengehörender Mapping-Einträge der gesamte Baum durchsucht werden muß. Damit hätten wir auch hier eine Komplexität von $O(n)$.

Entschieden haben wir uns am Ende für die Variante "Bäume im Kasten". Sie erzeugt die kleinste Datenstruktur und ist nach unserer Ansicht am flexibelsten. Somit sind für die Speicherung eines Mappings nun 5 Byte notwendig. In der Abbildung 3.2 ist der Aufbau dargestellt.

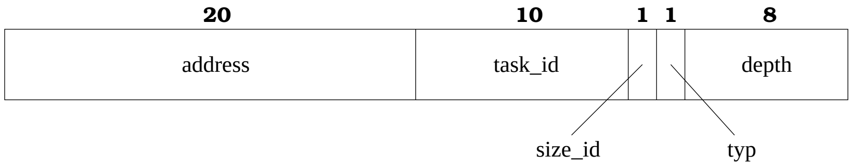


Abbildung 3.2: Aufbau der Mapping-Datenstruktur

### 3.2.2 Die Verwaltung

Die Verwaltung der Mapping-Einträge erfordert eine weitere Datenstruktur: die Wurzel des Baumes (Abbildung 3.3). Als erstes erfordert die Verhinderung gleichzeitigen Zugriffs auf einen Baum die Einführung eines Lock-Bits. Um nicht bei jedem Löschen eines einzelnen Mappings die Lücke schließen zu müssen, ist die Zählung der so frei gewordenen Plätze notwendig.

In einer laufenden L4-Umgebung gibt es Mapping-Bäume, die sehr groß werden (Shared Libraries) und andere, die nur sehr klein sind (private Seiten einer Task). Es wäre somit Verschwendung, für alle Bäume die gleiche Speichermenge zu reservieren. Außerdem ist dies unflexibel, da Sonderbehandlung notwendig wird, wenn die Anzahl der Einträge den geplanten Rahmen sprengt. Deshalb haben wir entschieden, diese Speicherverwaltung dynamisch zu gestalten. Die speziellen Entscheidungen hierzu werden in Abschnitt 3.3 erläutert. Hier ist nur entscheidend, daß die Variablen `mapping_count` und `size_id` notwendig werden.

Die einzelnen Daten ergeben sich wie folgt:

**mapping_count** verwaltet die Anzahl der Mapping-Einträge, die zu diesem Baum gehören. Mit 16 Bit ist die Anzahl der Mappings auf 65536 beschränkt, was bei einer maximalen Task-Anzahl von 1024 nicht bedenklich sein sollte.

**size_id** symbolisiert die Anzahl der Speicherplätze für Mapping-Einträge. Dabei wird diese Zahl nicht direkt eingetragen, sondern nach der folgenden Vorschrift berechnet:

$$Anzahl = K * 2^{SizeId}$$

$K$ entspricht der Anzahl der Einträge in der kleinsten Speicherzelle. In der aktuellen Implementierung ist $K = 4$, woraus sich die Zuordnungen in Tabelle 3.1 ergeben.

| size_id | 0 | 1 | 2 | 3 | 4 | 5 | 16 |
|---|---|---|---|---|---|---|---|
| Anzahl der Speicherplätze | 4 | 8 | 16 | 32 | 64 | 128 | 262144 |

Tabelle 3.1: Zuordnung des Größen-Index zur Größe der Speicherzelle

**lock** Dieses Bit wird gesetzt, wenn dieser Baum vor dem gleichzeitigen Zugriff mehrerer Tasks geschützt werden soll.

**empty_count** zählt die Anzahl der unbelegten Plätze zwischen dem Sigma0- und dem letzten Mapping. Diese entstehen, wenn beim Löschen eines einzelnen Mappings nicht so viel Platz frei wird, daß sich ein Umlagern lohnt. (Es werden hier also nicht die freien Plätze am Ende dieses Speicherbereiches verzeichnet.) Mit den per 7 Bit kodierten 127 leeren Einträgen sollte das System zurechtkommen, denn mit dieser Anzahl ungenutzter Plätze lohnt sich ein Umlagern.

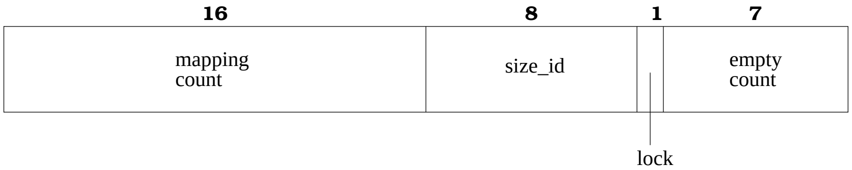


Abbildung 3.3: Aufbau der Mappingbaum-Wurzel

## 3.3 Speicherung der Mapping-Bäume

Die Verwaltung dieser Datenstruktur muß auf die Größe abgestimmt sein. Das heißt, es soll nach Möglichkeit kleine wie große Speicherbereiche genauso gut verwalten können. In [Vaha96] werden einige Strategien vorgestellt. Ein paar von diesen habe ich bereits in Abschnitt 2.7 zusammengefaßt. Hier folgen nun Diskussion und Entscheidung.

Auf Grund der Unflexibilität in Bezug auf die Objektgröße fallen *Simple Power-of-Two Free Lists*, *McKusick-Karels Allocator* und *Buddy System* zuerst heraus, auch wenn sie einige Vorteile, wie geringe Fragmentierung, haben. Doch der Anspruch, eine möglichst kompakte Datenstruktur für die Mapping-Einträge zu erarbeiten, verlangt auch eine effiziente Speichernutzung.

Der *Resource Map Allocator* ist zwar sehr einfach zu implementieren, kommt aber wegen seiner starken Fragmentierung und insbesondere der notwendigen Sortierung der Frei-Liste nicht in Frage, da dies sehr zeitaufwendig ist und wegen der Komplexität schwer echtzeitfähig zu implementieren ist.

Der *Zone Allocator* ist zwar sehr flexibel und effizient in der Speichernutzung, benötigt aber eine aufwenige Garbage Collection.

Nur der Slab Allokator erfüllt alle unsere Forderungen und wird auch in [Vaha96] als der am besten entworfene Allokator beschrieben. Nach einer Testserie, die von [Bown94] zitiert wird, ergibt sich eine durchschnittliche Fragmentation von 14% im Vergleich zu 45% für den *McKusick-Karels Allocator* und 46% für *Lazy Buddy Algorithm.*

# Kapitel 4

# Implementierung

## 4.1 Slab-Allokator

Da der Slab-Allokator (siehe Abschnitt 2.7.7) selbst den objektorientierten Ansatz verfolgt, liegt es nahe, ihn als Objekt zu implementieren, womit er sich auch aus dieser Sicht ausgezeichnet für das Projekt eignet. Aus der Funktionsweise ergeben sich die folgenden Schnittstellen:

**constructor** Der Konstruktor, der die Parameter zur Erzeugung eines Slab entgegennimmt, auf Zulässigkeit prüft und zur späteren Nutzung in die privaten Variablen überträgt. Als Parameter sind dabei die folgenden vorgesehen:

- Größe einer Speicherzelle
- Alignment jeder einzelnen Zelle
- ungenutzter Offset am Anfang einer Slab-Seite

**void *alloc()** sucht eine freie Speicherzelle in dem Slab, über den die Schnittstelle aufgerufen wurde. Ist dies der erste Aufruf oder ist der Vorrat an Zellen erschöpft, wird eine Kern-Speicherseite vom Kern-Seiten-Allokator angefordert und als Slab-Seite eingerichtet (siehe Methode `ctor(void *page)`).

**void free(void *object)** Die angegebene Speicherzelle wird wieder in die Liste der freien Zellen des Slabs eingeordnet.

**void *please_page()** Diese Methode ist speziell zum Aufruf durch den Kern-Seiten-Allokator gedacht. Er kann hiermit anfragen, ob einer der Slabs dieses Lagers völlig frei ist. Wenn ja, wird dieser aus der Liste der Slabs ausgeklinkt und somit für anderweitige Nutzung freigegeben.

**void destroy()** Da der hier implementierte Slab-Allokator ein generischer ist, müßten für jede daraus abgeleitete Klasse eigene Destruktoren geschrieben werden, die aber alle das gleiche machen würden: das Freigeben aller benutzten Seiten. Deshalb wurde diese Methode eingeführt, die diese Funktionalität für alle Klassen implementiert.

**void ctor(void *page)** Mit dem Aufruf dieser Methode — direkt nach der Belegung der neuen Seite durch den Benutzer (hier die Mapping-Datenbank) — werden die Verwaltungs-Daten am Ende der Seite *page* und die Verzeigerung der Zellen geschrieben.

## 4.2 L4-Slab-Allokator

Der Slab-Allokator sollte so konstruiert werden, daß er auch in späteren Projekten Anwendung finden kann. Deshalb wurden alle systemspezifischen Werte und Implementierungen in eine vom generischen Slab-Allokator abgeleiteten Klasse gekapselt, für die Mapping-Datenbank also einen L4-Slab-Allokator für einen Pentium, der von einer Seitengröße von 4KB und dem Kern-Seiten-Allokator "KMem" als Quelle für physische Seiten ausgeht.

Daraus ergeben sich die beiden folgenden als virtuell deklarierten Methoden:

**void *add_page()** Diese Methode fragt den Kern-Seiten-Allokator nach einer oder mehreren[1] Seiten und läßt sie sich an bestimmte Adressen in einem reservierten Speicherbereich einblenden. Die genaue Adresse legt der Memory-Manager fest. (siehe Abschnitt 4.3)

**void rm_page(void *page)** Ist eine Seite eines Lagers nicht mehr notwendig, so wird sie mit dieser Methode wieder aus dem Speicherbereich der Mapping-Datenbank entfernt, um sie der weiteren Nutzung durch den Kern-Seiten-Allokator zuzuführen.

## 4.3 Memory-Manager

Um der Mapping-Datenbank die Möglichkeit zu geben, mehrere Seiten zusammenhängend als Speicherbereich zu bekommen, muß sie die freien Adressen im reservierten Speicherbereich[2] entsprechend verwalten. Dafür ist der Memory-Manager zuständig.

### 4.3.1 Schnittstellen

**unsigned16_t reserv_page(unsigned8_t count)** fordert *count* hintereinanderliegende Seiten an. Zurückgegeben wird die Nummer der ersten Seite, von der an nun *count* Seiten vom Kern-Seiten-Allokator angefordert werden können.

**void return_page(unsigned16_t start, unsigned8_t count)** Werden eine oder mehrere Seiten aus dem Pool der Mapping-Datenbank nicht mehr benötigt und sind dem Kern-Seiten-Allokator zurückgegeben worden, müssen auch ihre Adressen im virtuellen Speicherbereich des Kernes wieder freigegeben werden. Mit der Methode `return_page()` werden von der Seite *start* an *count* Seiten als frei gekennzeichnet.

Bei der Seiten-Nummer wird von der Basisadresse `0xd000 0000` ausgegangen.

### 4.3.2 Algorithmus

Für alle Seiten innerhalb des virtuellen Speicherbereiches für die Mapping-Datenbank wird eine Belegungskarte in Form einer Bitmap geführt. Jedes Bit repräsentiert dabei eine Seite.

Die Wertigkeit der einzelnen Bits eines jeden Bytes werden so interpretiert, daß das kleinste Bit links steht. Wenn man nun alle Bytes der Karte von links nach rechts aufsteigend aneinander setzt, entsteht so ein Repräsentant des virtuellen Adressbereiches, den man sich ebenso angeordnet vorstellen soll. Damit kann man über die Bytegrenzen der Karte hinweg zusammenhängend leere Speicherbereiche finden.

Dazu wird eine Variable `muster` des Typs `unsigned32_t` (unsigned mit 4 Byte Größe) mit $xx$ Bits besetzt — so viele, wie Seiten angefordert wurden. Anschließend wird ein Zeiger des Typs `unsigned32_t` auf das erste Byte der Karte gesetzt und auf Übereinstimmung mit `0xffff ffff` untersucht. Ist dies zutreffend, sind die nächsten 4 Byte vollständig besetzt und der Zeiger wird um 4 Byte erhöht. Wenn nicht, wird eine Kopie von `muster` nacheinander über alle 32 Bit des untersuchten Bereiches gelegt und mit bitweisem UND verknüpft (siehe

[1] bei Erzeugung eines großen Lagers (mit einer Zellengröße von über 4kB)
[2] Für die Mapping-Datenbank wurde der Bereich von `0xd000 0000` bis `0xd9ff ffff` reserviert.

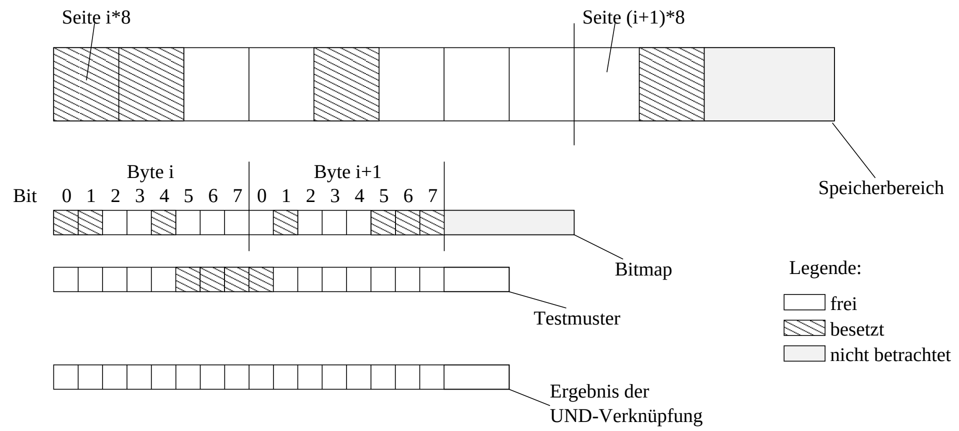


Abbildung 4.1: Memory-Manager in einem möglichen Zustand

Abb. 4.1). (Es werden dabei genau $32 - xx$ Vergleiche durchgeführt.) Ist dieses Ergebnis 0, können die gewünschten Seiten aus dem dazugehörigen Bereich belegt werden. Wird bei keinem dieser Vergleiche das Ergebnis 0 ermittelt, wird der Zeiger nur um ein Byte weitergestellt und die Suche von vorn begonnen.

Die zuvor beschriebene Vorgehensweise ist nur effizient, wenn mehrere Seiten angefordert werden. Deshalb habe ich einen Pfad für den häufigsten Fall der Anfrage nach einer einzelnen Seite hinzugefügt. Dieser nutzt zum Vergleich Variablen des Typs `unsigned8_t` (unsigned mit 1 Byte Größe), geht aber ansonsten äquivalent vor.

Viel einfacher ist die Freigabe der Seiten. Hier wird mit dem Bit Nr. *start* an die nächsten *count* Bits auf Null gesetzt. Die reale Adresse des ersten Bits läßt sich einfach ausrechnen: $freebyte = start\,DIV\,8$ bzw. $freebit = start\,MOD\,8$. Beim Durchlauf für die restlichen Bits wird nur *freebit* jeweils um 1 erhöht. Bei $freebit = 8$ wird *freebit* wieder auf Null gesetzt und *freebyte* inkrementiert.

## 4.4 Mapping-Datenbank

Die Mapping-Datenbank ist für die gesamte Verwaltung der Mapping-Struktur verantwortlich. Daraus ergibt sich, daß sie auch für die Initialisierung aller notwendigen Daten beim Systemstart zuständig ist. Dies ist die Aufgabe des:

**constructor** Er bereitet den Memory-Manager auf seine zukünftige Aufgabe vor. Dazu erhält dieser zwei physische Seiten, in denen jedes Bit eine virtuelle Seite im Mapping-Speicherbereich repräsentiert (siehe Abschnitt 4.3).
Danach wird die Größe des physischen Speichers ermittelt und ein von dieser Größe abhängender Speicherbereich erzeugt, in dem für jede physische Seite ab jetzt ein globaler Zeiger auf den dazugehörigen Mapping-Baum gespeichert wird.
Anschließend werden mehrere Slab-Allokatoren instanziiert — für jede Mapping-Baum-Größe, die erwartet wird, einer. In der aktuellen Implementierung werden statisch 10 Instanzen erzeugt[3]. Der größte kann dabei Bäume mit $4 * 2^9 = 2048$ Mappings enthalten. Sollte sich dies im Dauerbetrieb als zu wenig herausstellen ist einfach der Wert des Symbols L4_SLAB_NUMBER entsprechend zu erhöhen.
Danach wird für jede physische Seite ein Mapping-Baum mit einem entsprechenden Eintrag für den Sigma0-Pager erzeugt.

Ein Destruktor ist für ein laufendes System nicht sinnvoll: die Mapping-Datenbank ist solange notwendig, wie der Kern läuft. Deshalb ist er nicht implementiert worden.

[3] In der Zukunft wird dies dynamisch erfolgen.

Im folgenden werde ich nun die Vorgehensweisen der einzelnen Methoden der Mapping-Datenbank erläutern.

### 4.4.1 Einfügen von Mapping-Einträgen

Ausgehend von einem Quell-Mapping (Vater) können Kind-Mappings erzeugt werden. Dies wird notwendig, wenn eine Task eine ihrer Seiten an eine andere Task weitergeben will.

Der Methode `insert()` wird als ein Parameter das Quell-Mapping übergeben. Über die Werte in dem als Vater angegebenen Mapping kann in der Seitentabelle die physische Adresse der entsprechenden Seite ausgelesen werden. So läßt sich schnell der zum Mapping-Eintrag gehörige Baum finden.

Ist die maximale Anzahl von Mapping-Einträgen für die Baumgröße erreicht[4], muß der Baum in eine neue Speicherzelle umgelagert werden. Dazu muß vom L4-Slab-Allokator eine neue Speicherzelle angefordert werden (siehe Abschnitt 4.2). Im Zuge der Umlagerung werden alle nicht genutzten Einträge innerhalb des Baumes an das Ende gerückt.

Es mag verwirrend erscheinen, daß bei der Kontrolle der Größe auch leere Einträge mitgezählt werden und somit ungenutzt bleiben. Der Aufwand, zu untersuchen, wie einzelne Mappings auf freie Einträge umgelagert werden müßten, um für das neue Mapping an der richtigen Stelle Platz zu schaffen, ist meines Erachtens zu hoch. Man stelle sich dazu eine Zelle mit 4 Einträgen vor, von denen der erste, dritte und vierte belegt ist. Soll vom Mapping des dritten Eintrags aus nun ein neues Mapping eingefügt werden, wäre die Lösung, den dritten Eintrag in den zweiten zu kopieren und den frei gewordenen für das neue Mapping zu verwenden. Dies mag für Bäume dieser Größe machbar sein, ist für Bäume mit vielleicht 50 Einträgen jedoch sehr zeitaufwendig.

Es ist einfacher, eine größere Zelle zu belegen. In diesem Falle bleibt die Zellengröße eine Weile erhalten, auch wenn ein Mapping wieder gelöscht wird. Darauf gehe ich in 4.4.5 genauer ein.

Ist eine Verlegung des Baumes nicht notwendig, werden nur die hinter dem Vater-Mapping liegenden Einträge um eine Einheit (5 Byte) verschoben, um Platz für den neuen Eintrag zu schaffen. Die Zähler für die belegten Einträge werden inkrementiert.

Ist so der Platz für ein neues Mapping frei, werden noch die charakteristischen Daten eingetragen, ggf. der globale Zeiger zum Baum aktualisiert und die Adresse des Mappings zurückgegeben.

**Einschränkung der Spezifikation** Die Implementierung dieser Methode geht davon aus, daß zwischen dem Suchen eines Mapping-Eintrags und der Freigabe des Baumes nur ein einziges Mapping eingefügt wird. Zwar wird ggf., wie zuvor beschrieben, der Baum in eine größere Zelle verlegt, jedoch muß der Anwender dieser Methode nun damit rechnen, daß der Zeiger auf den Eintrag, den er durch die vorherige Suchoperation erhalten hatte, nun nicht mehr aktuell ist und in einen ungültigen oder bereits anderweitig vergebenen Speicherbereich zeigt.

Sollte es also notwendig werden, zwei oder mehr Mappings direkt nacheinander einzufügen, sollte man den Zeiger des Vaters durch einen Aufruf der Methode `parent()` über dem erhaltenen Mapping aktualisieren.

### 4.4.2 Erzeugen eines Sigma0-Mappings

Für die im Parameter als Kachel-Nummer übergebene Zahl wird ein Mapping-Baum erzeugt. Dazu wird als erstes von dem Slab-Allokator mit der kleinsten Baumgröße eine Speicherzelle angefordert. Anschließend werden Wurzel des Baumes und der Eintrag des Sigma0-Mappings in den Speicher geschrieben (siehe Abb. 3.3 und 3.2), der globale Zeiger für diesen Baum auf die Wurzel gesetzt und die Adresse des Sigma0-Mappings zurückgegeben.

[4] `header->count`+`header->empty_count`$= 3 * 2^{sizeid}$

### 4.4.3 Verschenken eines Mappings

Der *grant* genannte Vorgang ist leicht zu implementieren, da an der Adresse des übergebenen Mappings nur die charakteristischen Daten verändert werden müssen.

### 4.4.4 Suchen nach Einträgen

Um eine Änderung an einer Mappingstruktur vornehmen zu können, muß die Position des Mapping-Baumes bekannt sein. Die Methode `lookup()` bietet die Möglichkeit, mit bekannter virtueller Adresse und Task-Nummer eine Referenz zu dem entsprechenden Mapping zu suchen. Wird der Baum, der zu der entsprechenden physischen Seite gehört, gefunden, muß er zuerst für weitere Zugriffe gesperrt werden. Dafür wird in einer atomaren Aktion das Lock-Bit (siehe Abb. 3.3) auf seinen Zustand getestet und auf Eins gesetzt, wenn dies nicht schon geschehen war. In diesem Fall hat bereits ein anderer Prozeß den Zugriff reserviert, und die eigene Anfrage wird mit einem NULL-Zeiger als Antwort abgewiesen.

Erst jetzt kann wirklich nach dem gewünschten Mapping gesucht werden. Ist es wider Erwarten nicht zu finden, wird der Baum wieder freigegeben und NULL als Antwort gegeben, andernfalls die Adresse des Mappings.

### 4.4.5 Freigabe eines Baumes

Nach Abschluß aller Aktionen mit einem Mapping muß der Baum wieder freigegeben werden. Erst jetzt nimmt das System auch die Möglichkeit wahr, freie Einträge innerhalb des Baumes an das Ende zu verschieben. Schrumpft der Baum dadurch auf eine weit kleinere Größe, so daß er in einen kleineren Puffer verlagert werden kann, wird auch diese Aktion ausgeführt, bevor die Freigabe erfolgt.

Hierbei habe ich Grenzwerte festgelegt, die der Größenwert überschreiten muß, damit Umlagerungen ausgeführt werden. Für das Kompaktifizieren muß die Anzahl der freien Einträge 127 erreichen oder die Zahl der genutzten überschreiten. Das Belegen einer kleineren Zelle wird nur dann ausgeführt, wenn der Baum sogar in eine noch kleinere Zelle passen würde, also etwa nur die Hälfte der möglichen Einträge belegt bleibt.

Wenn kein Verkleinern des Baumes erfolgt, wird nun untersucht, ob noch mindestens ein freier Eintrag vorhanden ist. Ansonsten muß der Baum in eine größere Zelle verlegt und der globale Zeiger zum Baum aktualisiert werden.

Abschließend wird die Sperrung für den Baum aufgehoben.

### 4.4.6 Löschen von Einträgen

Ursprünglich war in dieser Methode die Reorganisation eines Mapping-Baumes implementiert. Im Test stellte sich jedoch heraus, daß eine Umlagerung von Mapping-Einträgen erst bei der Freigabe geschehen darf. Deshalb ist diese Methode deutlich kleiner geworden.

Nach dem Auffinden der Baum-Wurzel über den Eintrag in der Seitentabelle wird die Anzahl der Einträge gezählt, die gelöscht werden müssen. Dabei wird mit dem übergebenen Mapping (m1) beginnend solange ein Zeiger weitergestellt, bis ein so adressiertes Mapping (m2) einen Tiefen-Index aufweißt, der kleiner oder gleich dem von m1 ist. m2 ist dann ein Mapping, das nicht mehr zum Teilbaum mit m1 als Wurzel gehört. Leere Einträge werden mitgezählt. Die Suche bricht auch ab, wenn ein Eintrag mit der Baum-Ende-Markierung gefunden wird oder die Anzahl der maximal möglichen Einträge in dieser Zelle erreicht ist.

Nun wird der Speicherbereich aller zu löschenden Einträge überschrieben: entweder mit der Markierung für leere Einträge (wenn m2 auf ein gültiges Mapping zeigt) oder mit der Baum-Ende-Markierung (wenn m2 ebenso hinter das Baumende zeigt). Da bei der Suche nach freien Einträgen immer nur auf den Tiefen-Index geachtet wird, kann ohne Gefahr für Mißverständnisse der ganze Eintrag mit dem Tiefen-Index beschrieben werden.

Abschließend werden die Werte für freie und genutzte Einträge in der Baum-Wurzel aktualisiert.

## 4.5 Mappings

Um die Informationen über ein Mapping über die Grenzen der Mapping-Datenbank zu bringen, werden mehrere Methoden angeboten, die die gewünschten Werte zurückliefern. Nur drei Methoden der Klasse `mapping_t` haben einen etwas komplizierteren Aufbau. Diese sind für ein Durchsuchen eines Mapping-Baumes notwendig.

**next_iter** Ein Zeiger wird, ausgehend vom adressierten Mapping, um die Anzahl von Bytes weitergestellt, die der Größe eines Eintrags entsprechen, also 5 Byte. Befindet sich hier ein gültiges Mapping, wird dessen Adresse zurückgeliefert. Wird der Eintrag als frei erkannt, wird der Zeiger erneut um 5 Byte weitergestellt bzw. ein NULL-Pointer zurückgeliefert, wenn das Ende des Baumes erkannt wird.

**next_child** Äquivalent zu `next_iter` wird ein gültiges Mapping gesucht. Es wird jedoch nur als Ergebnis zurückgegeben, wenn anhand des Vergleiches des Tiefenindex erkannt werden kann, daß das gefundene Mapping ein Kind des als Parameter angegebenen ist.

**parent** Nachdem getestet wurde, ob das adressierte Mapping ein Sigma0-Mapping ist (da dies keinen Vorgänger hat), wird ähnlich den beiden zuvor beschriebenen Methoden vorgegangen, wobei der Zeiger nicht vor-, sondern zurückgestellt wird. Ist der Tiefenindex des gefundenen Mappings kleiner als der des adressierten, so wird es als Ergebnis zurückgeliefert. Sonst wird es wie leere Einträge ignoriert und der Zeiger weiter zurückgestellt.

## 4.6 Probleme bei der Implementierung

Im Laufe der Implementierungsarbeit zeigten sich einige Schwierigkeiten, die während des Entwurfes so nicht gesehen wurden. So ist es z.B. beim Durchlaufen eines komplett gefüllten Baumes nicht einfach, den letzten Eintrag eindeutig als solchen zu identifizieren, da direkt anschließend ein weiterer Baum folgen kann. Solange man durch eine vorherige Operation die Wurzel des Baumes gefunden hat, kann man die Adresse des letzten Eintrags ermitteln. Doch für Operationen wie `mapping_t->next_iter()` sind eigentlich keine Informationen aus der Wurzel notwendig.

Ein besonders großes Problem stellte für mich der Test meiner Datenbank dar. Die Datenbank ist für die Nutzung innerhalb von Fiasco gedacht, und wurde auch so entwickelt. Es ist möglich, eine laufende Version des Kerns von einem anderen Rechner zu debuggen. Doch da der größere Teil des Kerns nicht von mir stammt, habe ich kaum Kenntnis von den intern ablaufenden Vorgängen. Für einen solchen Test war ich also immer auf die direkte Zusammenarbeit mit Michael Hohmuth angewiesen. Aus zeitlichen Gründen mußte ich also eine andere Testmöglichkeit finden.

Dazu erarbeitete ich eine Linux-Umgebung für meine Datenbank. Für alle Anweisungen innerhalb meines Quelltextes, die direkt auf Teile des Kerns zugreifen, mußte ich ein Äquivalent für die Linux-Umgebung einbauen. Bei entdeckten Fehlern mußten diese dann immer an zwei Stellen gleichzeitig geändert werden. Dabei haben sich leicht Fehler eingeschlichen.

Die wenigen Versuche, die Mapping-Datenbank innerhalb von Fiasco ohne Michael Hohmuths Hilfe zu testen, wurde durch einen Stacküberlauf im OS-Kit[5] erschwert. Bei einigen Ausgaben auf dem Statusbildschirm blieb einfach das System stehen, ohne diese Ausgaben lief der Startvorgang wesentlich weiter.

[5]eine Sammlung von Bibliotheken, welche das Entwickeln von Betriebssystemen vereinfacht; siehe [FBB+97]

# Kapitel 5

# Leistungsbewertung

## 5.1 Echtzeitaspekte

Für die Berechnung der Systemruf-Zeiten ist es notwendig, die maximalen Laufzeiten in der Datenbank anzugeben. Nachfolgend sind diese Informationen für die einzelnen Methoden in Tabellen zusammengefaßt und mit Kommentaren versehen. Dabei will ich nicht einzelne Befehle angeben, sondern die Beschreibung des *Worst Case* und der entsprechenden Komplexität.

### 5.1.1 Mapping-Datenbank

Alle hier nicht genauer untersuchten Methoden enthalten nur sehr wenige Anweisungen ohne jede Möglichkeit von Wiederholungen. Dazu zählen die Methoden `space()`, `vaddr()`, `size()` und `type()` der Klasse `mapping_t`.

| Methode | Worst Case | Komplexität |
|---|---|---|
| Mapping einfügen (ohne Einschränkung) | Belegen einer neuen Seite für den Slab Allokator und einer größeren Zelle — Umlagern aller Einträge | linear wachsend mit der Größe der Speicherzelle |
| Mapping einfügen (mit Einschränkung aus Abschnitt 4.4.1) | Verschieben aller Einträge um einen Eintrag | linear wachsend mit der Größe der Speicherzelle |
| Mapping löschen | Markieren aller Einträge | linear wachsend mit der Größe des Baumes |
| Mapping suchen | Durchsuchen des ganzen Baumes | linear wachsend mit der Größe des Baumes |
| Baum freigeben | Belegen einer neuen Seite für den Slab-Allokator und einer kleineren Zelle — Umlagern aller Einträge | linear wachsend mit der Größe des Baumes |

Tabelle 5.1: Echtzeitbetrachtung der Methoden der Mapping-Datenbank

#### 5.1.1.1 Einfügen eines Eintrags

Falls wider Erwarten (Erklärung dazu in Abschnitt 4.4.5) eine neue Speicherzelle belegt werden muß, um einen neuen Eintrag einfügen zu können, wird die Bearbeitungszeit länger. Im *ungünstigsten Fall* muß der Slab-Allokator sich selbst eine neue Seite anfordern und einrichten.

Hält man sich an die Annahme, immer nur ein Mapping einzufügen (siehe Abschnitt 4.4.1), reduziert sich der *Worst Case* auf das Verschieben des gesamten Baumes um einen Eintrag für den Fall, daß das neue Mapping direkt nach dem Sigma0-Mapping eingefügt werden soll.

Da alle Einträge des Baumes hinter dem Vater-Mapping verschoben werden, also auch die noch nicht genutzten, ist die Zeit des Einfügens für alle die Baumgrößen konstant, die in die gleiche Speicherzelle passen.

#### 5.1.1.2 Löschen eines Eintrags

Da die Anzahl der zu löschenden Einträge von außen nicht bekannt ist, kann nur die maximale Anzahl von Einträgen in der Speicherzelle als eine obere Schranke für den Suchvorgang angesetzt werden.

#### 5.1.1.3 Suchen nach einem Eintrag

Nacheinander werden die Werte jedes gültigen Mappings im Baum mit denen des gesuchten verglichen. Maximal werden so viele Vergleiche durchgeführt, wie Einträge in diese Zelle passen.

#### 5.1.1.4 Freigeben eines Baumes

Wenn der Baum umgelagert werden soll (siehe dazu Abschnitt 4.4.5), muß eine neue Zelle belegt und alle nicht leeren Einträge, deren Anzahl eindeutig in der Baum-Wurzel steht, einzeln umgelagert werden. In diesem Fall muß man damit rechnen, daß der Slab-Allokator eine neue Seite anfordern und einrichten muß.

| Methode | Worst Case | Komplexität |
|---|---|---|
| Nächster Nachbar und Nächstes Kind | Überprüfung aller leeren Einträge | linear wachsend mit der Anzahl leerer Einträge |
| Vater suchen | Durchsuchen des ganzen Baumes | linear wachsend mit der Größe des Baumes |

Tabelle 5.2: Echtzeitbetrachtung der Methoden über die Mapping-Einträge

#### 5.1.1.5 Suche nach Mappings in der Umgebung eines bestimmten Mappings

Die beiden Methoden `next_iter()` und `next_child(parent)` der Klasse `mapping_t` bieten nahezu gleiche Funktionalität (siehe 4.5) und sind in der Echtzeitbewertung als äquivalent einzuschätzen. In den meisten Fällen sollte der erste gefundene Eintrag das gesuchte Mapping enthalten. Wenn sich jedoch leere Einträge im Baum befinden, kann die Suche entsprechend länger dauern. Im *schlechtesten Fall* müssen so viele Vergleiche vorgenommen werden, wie leere Einträge in der Wurzel des Baumes verzeichnet sind.

Die Methode `parent()` ist ähnlich zu bewerten. Jedoch ist hier die zu erwartende Anzahl von Vergleichen höher anzusetzen. Ist ein Mapping als letztes von vielen Kindern eines Vaters-Eintrags aufgelistet, müssen so viele Einträge verglichen werden, wie seine Geschwister Abkömmlinge haben. So genau läßt sich diese Zahl in der Praxis nur mit großem Zeitaufwand ermitteln, so daß als *Worst Case* mit der Gesamtzahl der Einträge des Baumes gerechnet werden muß.

### 5.1.2 L4-Slab-Allokator

#### 5.1.2.1 Neue Zelle belegen

Beim ersten Aufruf dieser Methode oder wenn die Anzahl der Zellen erschöpft ist, muß vor der Belegung einer Zelle eine neue Seite angefordert und eingerichtet werden (siehe

| Methode | Worst Case | Komplexität |
|---|---|---|
| Zelle belegen | Seite belegen und Einrichten — Zelle belegen | linear wachsend mit der Anzahl belegter Seiten in diesem Slab |
| Zelle freigeben | in Frei-Liste einketten | — |
| Seite belegen | Seite reservieren — Seite belegen (KMem) | siehe Memory-Manager |
| Seite einrichten | Schreiben der Verwaltungsdaten — verketten aller Zellen | linear wachsend mit der Anzahl der entstehenden Zellen |
| Seite freigeben | Reservierung aufheben — Seite zurückgeben | siehe Memory-Manager |
| freie Seite suchen | Durchsuchen aller Seiten des Slabs — Seite freigeben | linear wachsend mit der Anzahl der Seiten in diesem Slab |

Tabelle 5.3: Echtzeitbetrachtung der Methoden des Slab-Allokators

Abschnitte 5.1.2.3 und 5.1.2.4).

#### 5.1.2.2 Zelle freigeben

Hier werden nur die Zelle in die Kette der freien Zellen eingehängt und die Werte im Verwaltungsbereich der Seite aktualisiert. Die Zeit für diese Aktion ist konstant.

#### 5.1.2.3 Neue Seite belegen

Nach der Reservierung einer oder mehrerer Seiten beim Memory-Manager (siehe 5.1.3.1) werden diese über den Kern-Seiten-Allokator belegt und die Startadresse zurückgegeben.

#### 5.1.2.4 Seite einrichten

Am Beginn jeder Zelle wird die Adresse der nächsten Zelle eingetragen. Diese Aktion wird sooft wiederholt, wie Zellen in dieser Seite eingerichtet werden.

#### 5.1.2.5 Seite freigeben

Nach der Freigabe der Seite beim Kern-Seiten-Allokator wird die Seite auch in der eigenen Verwaltungsstruktur wieder freigegeben (siehe 5.1.3.2).

#### 5.1.2.6 Freie Seite suchen

Zum Auffinden einer Seite, die nicht unbedingt benötigt wird, müssen alle Seiten des Slab untersucht werden.

### 5.1.3 Memory-Manager

#### 5.1.3.1 Seiten reservieren

Es wird in der Belegungskarte ein Bereich gesucht, in dem die gewünschte Anzahl von Seiten reserviert werden kann. Die Maximalzahl an Suchvorgängen liegt dabei in der Größenordnung der Zahl von physischen Seiten, die der Mapping-Datenbank zur Verfügung stehen.

#### 5.1.3.2 Seite als frei markieren

Bei der Freigabe mehrerer Seiten in der Belegungskarte werden genau so viele Speicherzugriffe getätigt, wie Seiten als frei markiert werden sollen.

| Methode | Worst Case | Komplexität |
|---|---|---|
| Seite reservieren | Durchsuchen der gesamten Belegungskarte | linear wachsend mit der Größe des virtuellen Adressbereiches für die Mapping-Datenbank |
| Reservierung stornieren | Markieren der frei gewordenen Seiten (in der Karte) | linear wachsend mit der Anzahl der Seiten, die zurückgegeben werden |

Tabelle 5.4: Echtzeitbetrachtung der Methoden des Memory-Managers

## 5.2 Messungen

Da die Zusammenarbeit meiner Mapping-Datenbank mit dem restlichen Kern bisher noch nicht einwandfrei funktioniert, waren meine Testmöglichkeiten stark eingeschränkt. Es blieb mir keine andere Wahl, als in einer eigenen Umgebung auf Basis von Linux Geschwindigkeitsmessungen durchzuführen.

**Insert-Operation** In dieser Umgebung habe ich die einzelnen Zeiten gemessen, die für das Einfügen von 500 Mappings in einen Baum notwendig sind. Da mit wachsender Baum-Größe jeweils immer größere Speicherzellen zu belegen sind, ist zu erwarten, daß bei genau diesen Operationen mehr Zeit gebraucht wird, als wenn nur ein Teil der Einträge verschoben wird. Die Zeit, die für ein Verschieben der Einträge gebraucht wird, sollte mit der Speicherzellengröße wachsen.

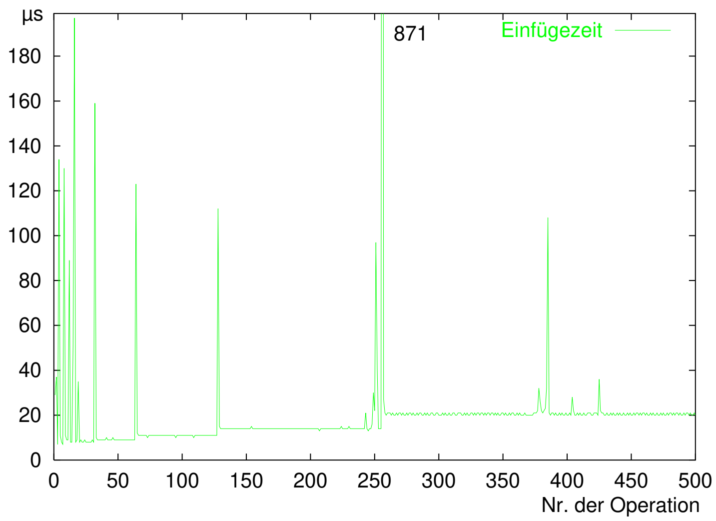


Abbildung 5.1: erster Durchgang — Einfügen von 500 Mappings

Um die Einflüsse der Linux-Umgebung erkennen zu können, habe ich die gesamte Operation mehrfach wiederholt. Deutliche Unterschiede sind nur im ersten Durchlauf (Abbildung 5.1) erkennbar. Dies wird durch die Tatsache verursacht, daß bei der Belegung einer größeren Speicherzelle für den Baum auch noch die Seite für den Slab-Allokator belegt und eingerichtet werden muß.

Die nachfolgenden Wiederholungen des Einfügens liefen dann nahezu gleich ab. Der Vergleich dieser Durchgänge (siehe Abb. 5.2) zeigt, daß vorhandene Spitzen in den Zeitkurven (z.B. um die Nummern 38 und 340) nicht in allen Durchgängen an genau der selben Stelle zu finden sind, lassen sich diese der Linux-Umgebung zuschreiben. Auch die Häufung im Abschnitt der ersten 50 Einfüge-Operationen (siehe Abb. 5.3) widerspricht nicht dieser Aussage.

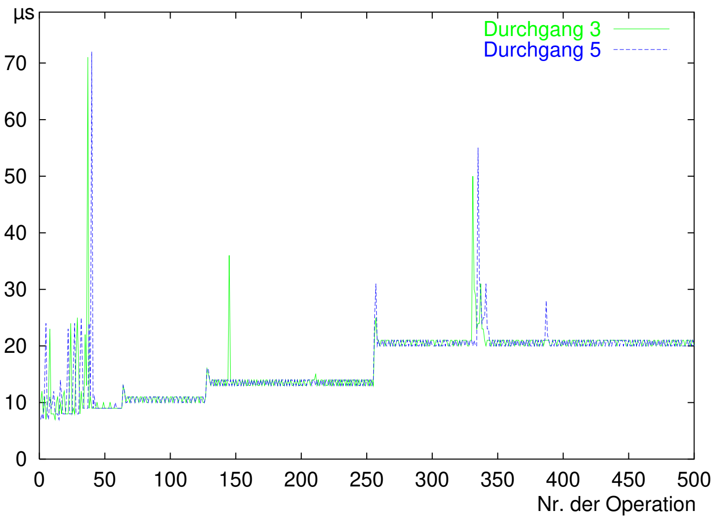


Abbildung 5.2: Vergleich der Durchgänge 3 und 5 — Einfügen von 500 Mappings

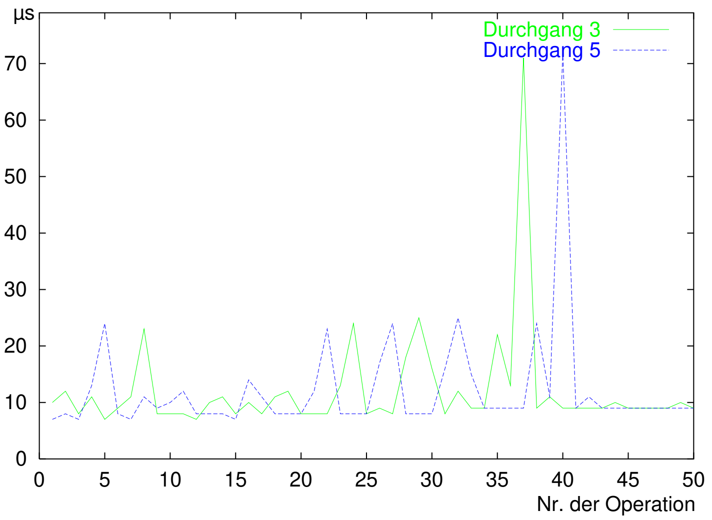


Abbildung 5.3: Vergleich der Durchgänge 3 und 5 — Abschnitt der ersten 50 Operationen

Wenn man von den Spitzen in den Zeitkurven absieht, da sie auf die Systemzeiten außerhalb der Datenbank zurückzuführen sind, werden die Erwartungen der Echtzeitanalyse erfüllt. Spitzen, die in allen Durchgängen zu finden sind (Einfüge-Operationen Nr. 8, 16, 32, 64, 128 und 256), treten genau an den Stellen zu Tage, wo sie erwartet wurden: bei Operationen, die ein Umlagern des Baumes erfordern. Deren Werte liegen jedoch bei fast allen Durchgängen unter 50 $\mu$s.

Die Unterschiede zwischen den Ausführungszeiten, bei steigender Nummer der Operation, entsprechen ebenso den Erwartungen: Nach einer Umlagerung des Baumes in eine neue Speicherzelle müssen immer mehr Einträge verschoben werden.

**Flush-Operation** Im Vergleich der Flush-Operationen[1] zeigen sich ebenso sehr ähnliche Zeitkurven. Die Spitzen (in der Abb. 5.4) sind genau zu den Zeitpunkten, an denen der Baum kompaktifiziert wird, was natürlich Zeit kostet. Die Tabelle 5.5 zeigt die Meßergebnisse der

[1] Die Freigabe des Baumes wurde gleich angeschlossen, da sonst keine Kompaktifizierung vorgenommen wird.

8 Durchläufe mit den Zeiten (in $\mu$s) für das Löschen des angegebenen Eintrags.

| Nr. der Operation | Testdurchlauf | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 127. Eintrag | 267 | 267 | 266 | 267 | 266 | 266 | 266 | 267 |
| 254. Eintrag | 180 | 178 | 178 | 178 | 178 | 179 | 179 | 179 |
| 374. Eintrag | 97 | 98 | 97 | 99 | 96 | 98 | 99 | 101 |
| 438. Eintrag | 52 | 51 | 51 | 50 | 50 | 51 | 51 | 50 |
| 470. Eintrag | 31 | 30 | 30 | 29 | 30 | 30 | 30 | 30 |

Tabelle 5.5: ausgewählte Meßergebnisse der Flush-Operation

Die Zeitspannen zwischen diesen Operationen mit Sonderbehandlung liegen zwischen 7 und 11 $\mu$s. Die 7 $\mu$s werden nur kurz nach dem Kompaktifizieren des Baumes erreicht. Der Wert steigt dann auf 11 $\mu$s an, weil jeweils die zuvor gelöschten Einträge genau in dem Bereich liegen, der von der Methode `flush()` untersucht wird, und somit immer mehr Zeit gebraucht wird.

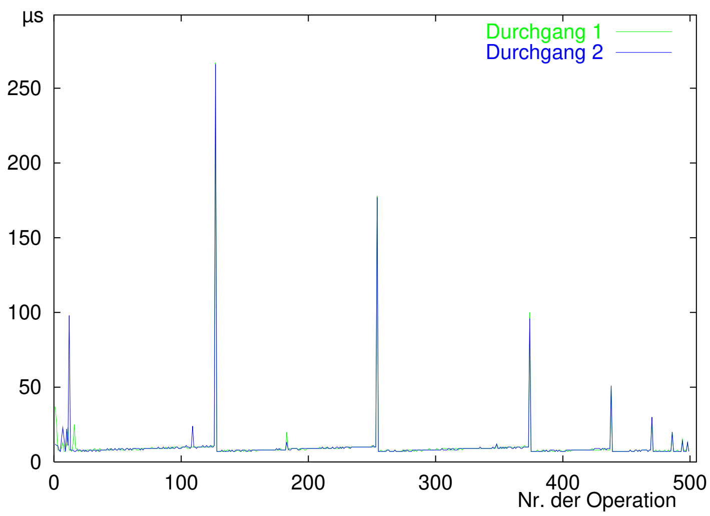


Abbildung 5.4: Vergleich zweier Flush-Folgen (1. und 2. Durchlauf)

Auch diese Werte belegen die Analyse in Bezug auf die Echtzeitfähigkeit der Datenbank: Zeiten für einen bestimmten Vorgang haben eine obere Schranke.

# Kapitel 6

# Schlußfolgerungen, Fragen und Ausblicke

Die vorliegende Version der Datenbank ist eine erste, relativ einfache Implementierung. Das primäre Ziel war die Erfüllung aller Aufgaben nach der Spezifikation [Hohm98]. So bleibt mein Betätigungsfeld auch nach Abschluß dieser Arbeit noch sehr weit: Es sind Algorithmenpfade zu extrahieren, die anschließend einer Optimierung unterzogen werden müssen. So ist das mittelfristige Ziel unserer Arbeit, die Performance des Kerns möglichst nahe an die der Originalversion des L4-Kerns von Jochen Liedtke heranzuführen. Dabei ist zu bedenken, daß der Code auch für Neueinsteiger[1] lesbar bleibt.

Zum jetzigen Zeitpunkt läßt sich aus meiner Arbeit ersehen, daß es möglich ist, mit einer sehr kompakten Datenstruktur (siehe Abschnitt 3.2) eine Mapping-Datenbank zu implementieren. Daneben wurde für die Verwaltung der Mapping-Daten ein Allokator entwickelt (siehe Anschnitt 3.3). Dieser existiert nun neben der kern-spezifischen auch in einer generischen Version (siehe Abschnitte 4.1 und 4.2), so daß er auch für anderweitige Aufgaben nutzbar ist.

Auch wenn die Zusammenarbeit mit den restlichen Teilen von Fiasco noch nicht zufriedenstellend ist, kann ich bereits erste Meßergebnisse (siehe Abschnitt 5.2) in einer Linux-Umgebung vorweisen, die auch die Echtzeitbetrachtungen (siehe Abschnitt 5.1) stärken.

Doch abgesehen von diesem Integrationsproblem, das ich in den nächsten Wochen bearbeiten werde, bleiben am Ende dieser Arbeit ungeklärte Fragen, die ich bemüht bin, in der nahen Zukunft so gut wie möglich zu beantworten.

So ist z.B. noch nicht untersucht, ob und wie es mit der entwickelten oder einer anderen Datenstruktur möglich ist, das Sperren von Einträgen auf einen Teil des Baumes zu beschränken und somit die Systemleistung zu verändern. Ebenso ist die Dimensionierung der einzelnen Teile der entwickelten Datenstrukturen in der Praxis zu testen, um die Speichernutzung weiter zu verbessern.

Desweiteren werde ich nach Möglichkeiten suchen, die Vorgänge, die sich bei den Messungen als besonders zeitaufwendig gezeigt haben, zu optimieren, um die Zeiten des *Worst Case* zu verringern.

Um die Echtzeitfähigkeiten des Kerns bestimmen zu können, müssen die Messungen aus Abschnitt 5.2 in der reinen Kernumgebung wiederholt werden, da die gewonnenen Werte nur die Begrenztheit der Ausführungszeiten aufzeigen.

Da der Slab-Allokator für die Speicherung der Mapping-Bäume die Möglichkeit bietet, die Cache-Nutzung des Prozessors zu verbessern, besteht auch hier ein Optimierungspotential.

[1] Hier ist insbesondere an zukünftige Studenten der TU Dresden gedacht.

# Kapitel 7

# Zusammenfassung

In der vorliegenden Arbeit werden Entwurf und Implementierung einer Datenbank für die Verwaltung der Mappingstruktur für FIASCO beschrieben. FIASCO ist eine Implementation eines Mikrokerns an der TU Dresden nach dem Vorbild von L4.

Kern der Arbeit war die Entwicklung der Datenbank unter Echtzeitaspekten. Bei der Analyse der implementierten Algorithmen wurde gezeigt, daß die Laufzeit aller Methoden im *Worst Case* ermittelbar ist.

Die ebenfalls neu entwickelte Datenstruktur für jeden Eintrag der Datenbank ist wesentlich kleiner als die Strukturen der bekannten Implementationen von L4.

Im Vergleich zum Original-L4 wurde auch der Sperrmechanismus verbessert. Unterbrechungen müssen nicht verhindert werden, und die Sperrung betrifft nur einen kleinen Teil der Datenbank.

## Danksagung

Ich möchte an dieser Stelle Michael Hohmuth für die Betreuung und den Mitarbeitern an der Professur Betriebssysteme für Ratschläge und Unterstützung danken.

# Anhang A

# Glossar

**Alignment** meint die Ausrichtung eines Speicherplatzes an einen bestimmten Byte-Rhythmus. Es gibt Prozessoren, die nicht oder nur schwer auf einzelne Bytes zugreifen können. Sie sind viel schneller im Zugriff auf ein Speicherblock von 4 bzw. 8 Byte (je nach Prozessortyp).

**Allokator** Bezeichnung für ein (Software-)Modul, daß Einheiten aus einem Pool verwaltet und an Klienten für die Benutzung zur Verfügung stellt.

**Cache Line** Da nicht für jede Speicherzelle des Hauptspeichers eine Zelle innerhalb des Prozessorcaches existiert bedarf es einer Zuordnungsvorschrift. Die *Cache Line* legt an Hand der Adresse im Hauptspeicher die Zelle des Cache-Speichers fest.

**Drops** *Dresden Real Time Operating System Project* — Betriebssystem-Projekt der Technischen Universität Dresden, Fakultät Informatik;
siehe `http://os.inf.tu-dresden.de/project/frontpage.html`

**OS-Kit** eine Sammlung von Bibliotheken, welche das Entwickeln von Betriebssystemen vereinfacht; siehe [FBB+97]

**Page Fault** Beim Zugriff auf einen virtuellen Speicherbereich, dem noch keine physische Seite zugeordnet ist, wird ein *Page Fault* (Seitenfehler) ausgelößt und durch den zuständigen Seitenverwalter behandelt.

**Pager** Der *Pager* verwaltet einen Pool von Speicherseiten. Anwendungen, die Speicher benötigen, wenden sich an “ihren” Pager, der ihnen Seiten zu Verfügung stellt.

**Shared Library** gemeinsame Bibliothek — Es ist üblich, sogenannte Code-Bibliotheken so zu erstellen, daß mehrere Prozesse gleichzeitig darauf zugreifen können. Sie werden einmal in den Speicher geladen und können dann von mehreren Prozessen benutzt werden.

# Literaturverzeichnis


[Vaha96] Uresh Vahalia (EMC Corporation Hopkinton, MA)
UNIX Internals — The New Frontiers
Prentice Hall, New Jersey 1996

[Lied96] Jochen Liedtke
L4 Reference Manual
GMD, Sankt Augustin

[Bown94] Bonwick, J.
The Slab Allocator: An Object-Caching Kernel Memory Allocator
Proceedings of the Summer 1994 USENIX Technical Conference
Juni 1994, Seiten 87-98

[Hohm98] Michael Hohmuth
The Fiasco Kernel: Requirements Definition
TU Dresden, 1998

[Uhli98] Volkmar Uhlig
Speicherverwaltung für den L4-Alpha Kern mit Echtzeitanforderungen
Großer Beleg, TU Dresden, Juni 1998

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, Olin Shivers
The Flux OSKit: A Substrate for OS and Language Research
In Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, October 1997

# Index