Technische Universität Dresden
Institut für Systemarchitektur
Professur Betriebssysteme

Großer Beleg

Kleine Adressräume für FIASCO

Sarah Hoffmann

9. August 2002

Inhaltsverzeichnis

1. Einleitung **4**

2. Stand der Technik **6**
2.1. IA32-Speichermanagement 6
2.1.1. Segmentierung 6
2.1.2. Virtueller Speicher 7
2.1.3. Zugriffsschutz 8
2.2. FIASCO 8
2.2.1. Speichermanagement 8
2.2.2. Mapping 9
2.2.3. Interprozess-Kommunikation (IPC) 9
2.3. Andere Arbeiten 9

3. Entwurf **10**
3.1. Konzept und Begriffe 10
3.2. Ziele 10
3.3. Design 12
3.3.1. Die Schnittstelle 12
3.3.2. Segmente 13
3.3.3. Paging 14
3.3.4. Daten aus dem Nutzeradressraum: IPC 15

4. Realisierung in FIASCO **17**
4.1. Änderungen und Ergänzungen im Code 17
4.2. Spezielle Aspekte 19
4.2.1. Taskwechsel 19
4.2.2. Paging 20

5. Leistungsbewertung **22**
5.1. IPC-Roundtrip-Zeiten 22
5.2. L^4Linux Benchmarks 23

6. Zusammenfassung und Ausblick **26**

A. Systemruf thread_schedule **27**

1. Einleitung

Als die ersten Betriebssysteme entwickelt wurden, geschah dies mit dem Gedanken, eine abstrakte Schicht zwischen Anwendung und Hardware einzuführen. Naheliegenderweise bedeutete das, alle Treiber für Hardware im Kern des Betriebssystemes selbst laufen zu lassen. Dieser Ansatz eines monolitischen Betriebssystems hat mit den Jahren jedoch einige Schwächen gezeigt. Die Systeme sind zu beträchtlicher Größe angewachsen und relativ unflexibel geworden, besonders dann, wenn das gleiche Betriebssystem mit einer möglichst reichhaltigen Auswahl an Hardware zusammenarbeiten soll. Außerdem zieht es einige Sicherheits- und Stabilitätsprobleme nach sich: Da ein so großes System unmöglich zental zu warten ist, werden Treiber von den verschiedensten Anbietern geschrieben; Treiber, die Zugriff auf sicherheitskritische Funktionen des Betriebssystems benötigen und bei Fehlfunktion den Kern selbst in Mitleidenschaft ziehen.

Als Lösung wurde vorgeschlagen, dass das Betriebssystem nur noch grundlegende Funktionen anbietet und alles Weitere Servern überlassen wird, die als Nutzerprozesse laufen. Mikrokerne der ersten Generation, wie etwa Mach [RJO+89], besaßen noch etwa 150 Funktionen, die der zweiten Generation wollen mit weniger als 10 auskommen. Damit teilt sich der Koloss Betriebssystem in einen Mikrokern und viele kleine Einheiten auf. Alle arbeiten unabhängig voneinander und können sich gegenseitig nicht mehr so einfach Schaden zufügen oder gar das ganze System zum Absturz bringen.

Das trotz alledem die bekanntesten und am häufigsten eingesetzten Betriebssysteme monolithischer Bauweise sind, liegt – wenn man einmal von marktwirtschaftlichen Faktoren absieht – vor allem daran, dass sie noch immer einen entscheidenden Geschwindigkeitsvorteil gegenüber Mikrokern-Betriebssystemen besitzen. Jede Funktion, die vom Betriebssystem selbst angeboten wird, kann durch einen einzigen Kerneintritt ausgeführt werden. Wenn in Mikrokernen nun diese Funktionen an Nutzertasks delegiert werden, muss der Aufruf durch Interprozess-Kommunikation (IPC) erledigt werden. Das bedeutet: Eintritt in den Kern, Zustellen der Nachricht, dann ein Taskwechsel zum Server, der die Funktion ausführt. Bei Bedarf schickt der Server eine Antwort zurück, wofür wiederum einen Kerneintritt und ein Taskwechsel benötigt wird.

Um die Geschwindigkeit eines monolithischen Betriebssystems zu erreichen, muss es also Ziel eines jeden Mikrokerns sein, IPC zu optimieren. Dazu gibt es verschiedene Vorschläge. Einer von ihnen stammt von Jochen Liedtke und beschäftigt sich speziell mit der Optimierung der Prozessumschaltung für L4 auf Rechnern der IA32-Architektur: *Kleine Adressräume*[Lie95].

L4 [Lie96b] ist eine Mikrokern-API der zweiten Generation. Sie stellt im Wesentlichen virtuelle Adressräume bereit, in denen ein oder mehrere Threads ausgeführt werden können. Das Betriebssystem schützt die Adressräume voreinander und bietet IPC an. Mit einer speziellen Form des IPC können außerdem Zuordnungen zwischen virtuellem und physischem Speicher getroffen werden. Für alle weiteren Aufgaben sind Nutzerprozesse verantwortlich.

Die Prozessumschaltung in L4 auf IA32 besteht hauptsächlich aus dem Retten des Prozessorstatus und dem Auswechseln der Seitentabelle. Bei letzterem wird zwar nur das Register neu geladen, das auf die Seitentabelle zeigt, jedoch zieht dies einige versteckte Kosten nach sich.

Die mit Hilfe der Seitentabelle errechneten Umsetzungen von virtuellen in physische Adressen werden in einem Cache, dem Translation Lookaside Buffer(TLB), gespeichert. Weil in ihm nicht vermerkt ist, zu welcher Seitentabelle ein Eintrag gehört, wird er beim Austausch der Seitentabelle gelöscht. Jeder Taskwechsel hat also zur Folge, dass dieser Cache wieder neu gefüllt werden muss.

Das Löschen des TLB lässt sich nur verhindern, wenn die Seitentabelle nicht ausgewechselt wird, selbst dann nicht, wenn zu einem anderen Adressraum umgeschaltet wird. Das bedeutet, dass mehrere Prozesse sich einen virtuellen Adressraum teilen müssen. In der Tat bietet die IA32-Architektur die Möglichkeit, den virtuellen Speicher in mehrere voreinander geschützte Teile zu gliedern, sogenannte Segmente.

Neben den „normalen" Anwendungen gibt es in Mikrokernsystemen wesentlich mehr „Serverprozesse". Das sind eben jene Prozesse, die Aufgaben des monolithischen Betriebssystems übernommen haben, also zum Beispiel Server, die Netzwerkverbindungen verwalten oder Swapper, die für die Verwaltung des Speichers und insbesondere dessen Auslagerung auf Festplatte verantwortlich sind. Solche Prozesse benötigen selbst meist wenig Speicher, kommunizieren aber besonders häufig mit den verschiedensten Prozessen im System. Idee der kleinen Adressräume ist es nun, diesen Serverprozessen nicht mehr einen vollständig eigenen virtuellen Adressraum zur Verfügung zu stellen, sondern sie in einem eigenen Segment in einen ungenutzen Teil eines jeden anderen Adressraumes einzublenden. Das Umschalten zu einer solchen Servertask erfordert dann nur eine Anpassung der Segmente, jedoch kein Auswechseln des Seitenverzeichnisses mehr. Die Segmente stellen sicher, dass die Adressräume dennoch voneinander getrennt und geschützt sind.

FIASCO [Hoh98] ist ein Mikrokern der zweiten Generation, welcher an der TU Dresden entwickelt wurde. Er implementiert die L4-API. Besonderes Ziel von FIASCO ist es, Echtzeitanwendungen neben gewöhnlichen Anwendungen laufen lassen zu können.

Kleine Adressräume wurden bereits im Original L4-Kern L4/x86 von Jochen Liedtke und in L4/Hazelnut der Universität Karlsruhe erfolgreich implementiert. Dort brachten sie erhebliche Geschwindigkeitsvorteile. Ziel dieser Arbeit ist es, solche kleinen Adressräume für FIASCO bereitzustellen.

Dieser Beleg gliedert sich wie folgt: Kapitel 2 gibt einen kurzen Überblick über die Speicherverwaltung der IA32-Prozessorfamilie und über die für die folgenden Ausführungen wichtigen Funktionen der L4-Schnittstelle sowie deren Implementierung in FIASCO. Die beiden folgenden Kapitel beschäftigen sich mit dem Entwurf und der Implementierung kleiner Adressräume selbst. In Kapitel 5 wird zeigt, dass mit der Implementierung Leistungsteigerung erreicht werden konnten. Zum Abschluss folgen noch einige zusammenfassende Betrachtungen sowie ein Ausblick auf künftige Arbeiten.

2. Stand der Technik

An dieser Stelle kann nur ein kurzer, stark vereinfachter Einblick in die für die Entwicklung kleiner Adressräume wichtigen Hard- und Softwaremechanismen gegeben werden. Ausführlichere Informationen über die Pentium-Architektur finden sich in den entsprechenden Handbüchern [Int99]. Die Schnittstelle des L4 Mikrokerns ist in [Lie96b] beschrieben. Zum FIASCO-Kern schließlich sind eine Reihe von Dokumenten auf den Seiten des Instituts Betriebssysteme der TU Dresden [TUD] verfügbar.

2.1. IA32-Speichermanagement

Aus historischen Gründen bietet die Speicherverwaltung der IA32-Archtiektur zwei voneinander unabhängige Mechanismen, den Speicher einzuteilen. Neben der Verwendung von virtuellem Speicher, mit dem unabhängig vom vorhanden Hauptspeicher der volle Adressraum verfügbar ist, besteht die Möglichkeit, diesen Adressraum in kleinere „Unteradressräume“, sogenannte *Segmente*, zu teilen.

2.1.1. Segmentierung

Die Idee der Segmentierung stammt aus Zeiten der Intel-8086-CPU, als der Adressbus plötzlich breiter war als die CPU-Register und deshalb der Speicher nicht mehr vollständig über Registerinhalte addressiert werden konnte. Um dennoch auf den gesamten Adressraum zugreifen zu können, wurde er in Segmente zerlegt und Adressen immer innerhalb dieser Segmente interpretiert. Heute sind zwar die Register wieder breit genug, um vollständige Adressen aufzunehmen, die Segmentierung bleibt aber in ähnlicher Form bestehen und bietet die Möglichkeit, den Speicher in voreinander geschützte Bereiche unterschiedlicher Funktion zu unterteilen.

Die Beschreibung der Segmente ist in zwei Deskriptortabellen, einer obligatorischen globalen (GDT, Global Descriptor Table) und einer optionalen lokalen (LDT, Local Descriptor Table), abgelegt. Neben Basisadresse und Größe finden sich hier noch einige Angaben zum Typ und zum Zugriffsschutz. Eine Speicheradresse wird immer im Kontext eines Segmentes interpretiert. Die angeforderte logische Adresse wird zuerst einmal daraufhin überprüft, ob sie sich innerhalb ihres Segmentes befindet. Dann wird die Basisadresse des Segmentes zu ihr hinzuaddiert, wodurch eine lineare Adresse entsteht.

Welches Segment verwendet wird, legen die Segmentregister fest. Jedes Register besitzt einen transparenten Cache, der die Segmenteigenschaften zwischenspeichert, damit bei der Umrechnung nicht jedes Mal auf die im Hauptspeicher liegenden Deskriptortabellen zugegriffen werden muss. Es gibt insgesamt sechs Segmentregister. Beim Zugriff auf Code wird implizit das Register CS verwendet, beim Lesen und Schreiben des Stacks das Register SS. Für alle anderen Speicherzugriffe kann zwischen den Registern DS bis GS gewählt werden, wobei DS genutzt wird, solange das Segmentregister bei der Adressierung nicht explizit angegeben wird.

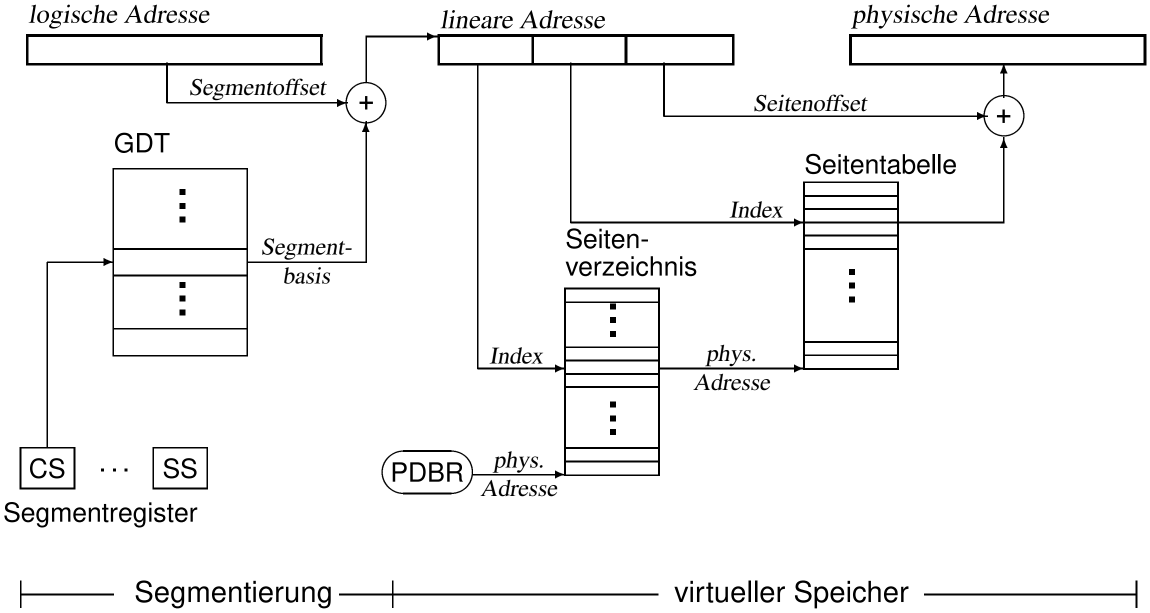

Abbildung 2.1.: Adressumsetzung in der IA32-Architektur

Segmentierung ist nicht abschaltbar. Segmentregister werden immer ausgewertet, auch dann, wenn sie den vollständigen virtuellen Adressraum umspannen. Der Prozessor erwartet, dass für jede verwendete Privilegstufe zwei Typen von Segmenten definiert sind: ein Code- und ein Datensegment, wobei letzteres aufgrund seines Typs auch für das Stacksegmentregister benutzt wird.

2.1.2. Virtueller Speicher

Bedeutender als die Segmentierung ist die Unterstützung von virtuellem Speicher. Hierzu wird die lineare Adresse, die nach Auflösung der Segmente entstanden ist, über eine zweistufige Seitentabelle in eine physische Adresse umgesetzt. Die erste Stufe, das *Seitenverzeichnis*, enthält eine Reihe von Einträgen, die jeweils auf Tabellen der zweiten Stufe, die *Seitentabellen*[1], verweisen. In diesen finden sich dann physische Adressen, die auf Rahmen im Hauptspeicher zeigen. Beide Arten von Tabellen enthalten außerdem noch Informationen zum Zugriffsschutz und für die Verwaltung.

Von der 32 Bit breiten Adresse werden 10 Bit für die Indizierung des Seitenverzeichnisses genutzt, weitere 10 Bit für die Seitentabelle. Die übrigen 12 Bit dienen als Offset in die Seite. Auf dem Pentium und seinen Nachfolgern ist es möglich, die zweite Stufe auszulassen. Dann werden die unteren 22 Bit vollständig als Offset verwendet. Somit lassen sich Seiten von 4 kB oder 4 MB Größe bilden.

Seitenverzeichnisse und -tabellen sind im Hauptspeicher abgelegt. Es kann mehrere Seitenverzeichnisse geben, jedoch bestimmt nur eines die aktuelle Adressumsetzung. Dessen physische

[1] Derjenige Teil der Leserschaft, der die Ähnlichkeit der Begriffe verwirrend findet, sei dahingehend beruhigt, dass Seitentabellen für diese Arbeit praktisch kaum eine Rolle spielen und man sich daher ganz auf Seitenverzeichnisse konzentrieren kann.

Abbildung 2.2.: Aufteilung des virtuellen Speichers in FIASCO

Adresse ist im *Page Directory Base Register* (PDBR)[2] gespeichert und kann durch Neuladen jederzeit geändert werden.

Da die Umsetzung von linearen in physische Adressen relativ aufwändig ist, werden alle Abbildungen in einem Cache, dem *Translation Lookaside Buffer (TLB)* zwischengespeichert. Ihm fehlt jedoch eine spezielle Kennzeichnung, zu welchem Seitenverzeichnis ein Eintrag gehört. Daher muss beim Neuladen des PDBR der TLB immer vollständig geleert (*TLB-Flush*) und dann nach und nach wieder neu gefüllt werden.

Eine Optimierung bieten die Prozessoren ab Pentium Pro, indem sie erlauben, Einträge, die von allen Seitentabellen gemeinsame genutzt werden, als global zu kennzeichnen und so vom Löschen aus dem TLB auszuschließen.

2.1.3. Zugriffsschutz

Die IA32-Architektur kennt vier Privilegstufen. Segmenten können alle vier, Seiten nur zwei verschiedene Privilegstufen zugeordnet werden. Der Prozessor muss auf gleicher oder niedrigerer Stufe laufen, um Zugriff auf sie zu erlauben. Zusätzlich kann noch die Art des Zugriffes – lesend oder lesend/schreibend – festgelegt werden.

Zugriffsrechte auf ein Segment werden beim Laden des Segmentregisters überprüft. Das Laden selbst ist kein privilegierter Befehl, kann also von jedem beliebigen Prozess ausgeführt werden, solange er die entsprechenden Privilegien für das zu ladende Segment besitzt.

Zugriffsrechte auf Seiten des virtuellen Speichers werden bei jeder Adressauflösung kontrolliert. Seitenverzeichnisse können nur im Kernmodus ausgetauscht werden.

2.2. FIASCO

FIASCO ist ein an der Technischen Universität Dresden entwickelter Mikrokern. Er beruht auf der L4-API. Geschrieben wurde FIASCO komplett in C++, vorrangiges Ziel war ein voll unterbrechbarer Kern, der auch für Echtzeitanwendungen verwendbar ist.

2.2.1. Speichermanagement

Wie die meisten anderen aktuell verfügbaren Betriebssysteme benutzt FIASCO ein lineares Speichermodell. Die obligatorischen Kern- und Nutzersegmente umspannen den kompletten virtuellen Adressraum und nutzen auf diese Weise die Segmentierung nicht. Als Datensegment dient immer das Nutzerdatensegment. Nur für das Stacksegmentregister wird das Kerndatensegment verwendet, da es vom Prozessor beim Kerneintritt automatisch geladen wird.

Abbilung 2.2 zeigt die Aufteilung des virtuellen Adressraumes. Er ist in 3 GB Nutzeradressraum und 1 GB Kernadressraum geteilt. Der Kernadressraum wird über die Zugriffsrechte in den

[2] Das PDBR ist auch bekannt unter seinem ursprünglichen Namen: CR3, Control Register 3, was allerdings nicht besonders aussagekräftig ist.

Seitentabellen vor Nutzertasks gesichert. Der Kern darf auf den ganzen Adressraum zugreifen. Der Kernbereich wird zwischen allen Adressräumen geteilt.

Jede Task besitzt ihr eigenes Seitenverzeichnis, welches eingeblendet wird, sobald zu einem ihrer Threads umgeschaltet wird. So werden die Nutzeradressräume voreinander geschützt.

2.2.2. Mapping

Dem Minimalitätsprinzip der Mikrokern-Philosophie folgend, wird die Aufteilung des verfügbaren physichen Speichers in L4 ebenfalls von Nutzertasks geregelt, um entsprechende Zuordnungsstrategien frei wählen zu können. L4 bietet nur den Mechanismus, um die gewählte Zuordnung auch durchzusetzen. Jede Task kann physische Seiten, die sie in ihren Adressraum eingeblendet hat, an anderes Tasks weitergeben, wenn die empfangende Task zustimmt. Dazu gibt es eine spezielle Form des IPC, mit der physische Seiten von einem Task an einen anderen geliehen (*mapping*), verschenkt (*granting*) oder auch wieder entzogen (*unmapping*) werden können. Das Betriebssystem verwaltet dafür eine Mappingdatenbank, in der vermerkt ist, wer an wen welche Seiten vergeben hat, und kümmert sich darum, dass sich das Mapping in den Seitenverzeichnissen und -tabellen widerspiegelt.

2.2.3. Interprozess-Kommunikation (IPC)

L4 spezifiziert zwei Arten von IPC. Beim *Short IPC* werden alle Werte über Register ausgetauscht, während beim *Long IPC* ein ganzer Speicherbereich übertragen wird. In FIASCO wird für *Long IPC* ein sogenanntes IPC-Fenster im Kernbereich benutzt. Dazu wird der Bereich des Empfänger-Adressraumes, in den kopiert werden soll, in dieses Fenster eingeblendet und der sendende Thread kopiert im Kernmodus direkt aus dem Nutzeradressraum in dieses Fenster. Da pro Task nur ein solches Fenster zur Verfügung steht, muss die Einblendung bei jedem Threadwechsel gelöscht und beim Zurückschalten zum sendenden Thread neu vorgenommen werden. Das ist insofern aufwändig, als das es immer einen TLB-Flush nach sich zieht.

2.3. Andere Arbeiten

Kleine Adressräume sind Teil der L4-Spezifikation für Pentium und bereits in L4/x86 sowie L4/Hazelnut erfolgreich implementiert worden. Beide Betriebssysteme wurden in erster Linie mit Hinblick auf Performanz entwickelt und sind daher im Kern nur bedingt unterbrechbar. Mit FIASCO muss sich zeigen, ob die Geschwindigkeitsvorteile, die in diesen Systemen realisiert wurden, auch dann noch zu erreichen sind, wenn Mehraufwand betrieben werden muss, um echtzeitfähig zu bleiben.

3. Entwurf

In diesem Kapitel soll es nur um allgemeine Konzepte gehen. Es werden alle wichtigen Designentscheidungen vorgestellt, ohne dabei auf spezielle Interna von FIASCO einzugehen. Wie und wo sie implementiert wurden, erklärt dann das nächste Kapitel.

3.1. Konzept und Begriffe

Vorrangiger Verwendungszweck kleiner Adressräume sollen Servertasks sein, die relativ wenig Speicherplatz benötigen, aber häufig von vielen verschiedenen Threads in Anspruch genommen werden. Neben den eingangs erwähnten Serverprozessen, ist zum Beispiel auch der L^{4}Linux-Server [Hoh96], der die Ausführung von Linux-Programmen auf L4 ermöglicht, eine interessante Anwendung.

Ziel ist es, dass diese Prozesse keinen eigenen Adressraum mehr benötigen. Stattdessen nisten sie sich parasitär bei denjenigen Tasks ein, die noch einen eigenen Adressraum besitzen, und werden dort ausgeführt. Das Prinzip ist das gleiche, wie es beim Betriebssystemkern angewendet wird. Im Kernbereich des Adressraumes, wo Nutzer bisher nicht zugreifen durften, wird für solche Tasks etwas Speicher, ein *Task-Fenster*, reserviert. Dieses Fenster ist global, es wird zwischen allen Adressräumen geteilt. Sollen mehr als eine Task in einem kleinen Adressraum laufen, kann das Task-Fenster noch segmentiert werden, wobei jeder Teil als Segment für genau eine Task fungiert.

Der virtuelle Speicher teilt sich somit in ein großes und mehrere kleine Segmente. Das große Segment erstreckt sich über die unteren 3 GB des Adressraums und wird von der Task benutzt, der das Seitenverzeichnis gehört. Die kleinen Segmente müssen im Kernbereich oberhalb der 3-GB-Grenze Platz finden. Ihre Größe ist frei wählbar. Während das großen Segment von verschiedenen Tasks verwendet wird, bleiben die Tasks in den kleinen Segmenten immer die gleichen.

Anders gesagt ändert sich aus Sicht der Tasks nichts, solange sie im großen Segment residieren. Wie bisher wird, sobald zu ihnen umgeschaltet wird, ihr Seitenverzeichnis vom Kern eingeblendet und sie arbeiten damit. Kleine Tasks hingegen benutzen das zur Zeit eingeblendete Seitenverzeichnis als *Wirtsadressraum*, kopieren Teile ihres Seitenverzeichnisses hinein und arbeiten dort.

Abbildung 3.1 zeigt exemplarisch, wie sich der Speicher in FIASCO mit kleinen Adressräumen aufteilt.

3.2. Ziele

Vom Konzept her kann jeder beliebige Prozess in einem kleinen Adressraum ausgeführt werden. In Hinblick darauf, dass der Taskwechsel verbessert werden soll, eignen sich jedoch Servertasks am besten. Bei dieser Art von Prozessen kann man außerdem davon ausgehen, dass sie sehr

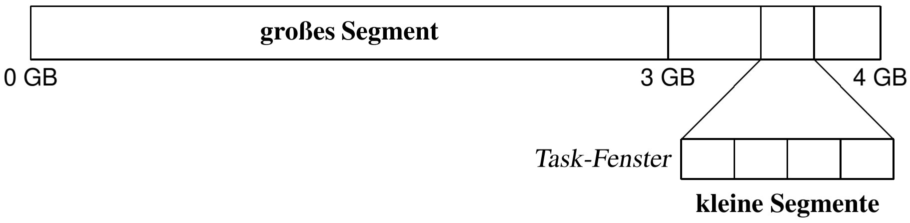

Abbildung 3.1.: Speicheraufteilung mit kleinen Adressräumen

langlebig sind und Zuordnungen zwischen virtuellem und physischem Speicher relativ stabil sind, sich das Seitenverzeichnis also selten ändert. Zu optimieren ist daher der „Normalbetrieb“, nicht so sehr seltene Operationen, wie das Verschieben von Tasks zwischen Segementen oder Mapping und Unmapping von Speicher.

Weitere Ziele bei der Verwirklichung kleiner Adressräume waren folgende:

Geringer Einfluss auf große Tasks Durch die Verwendung von kleinen Adressräumen sollte ein möglichst geringer Overhead für Tasks in normalen Adressräumen entstehen. Wo immer möglich, sollte zusätzlicher Verwaltungsaufwand zu Lasten der kleinen Tasks gehen. Dazu gehört auch, dass bei Nichtverwendung der kleinen Adressräume möglichst geringe Leistungseinbußen gegenüber der jetzigen Version ohne diese entstehen.

Unterbrechbarkeit Die Echtzeitfähigkeit von FIASCO darf durch die kleinen Adressräume nicht eingeschränkt werden, weshalb aufwändige Operationen unterbrechbar bleiben müssen.

Dynamisches Ein- und Auslagern Tasks müssen zu jedem beliebigen Zeitpunkt in kleine Adressräume verlegt oder daraus entfernt werden können. Ebenso sollte die Aufteilung des Task-Fensters zur Laufzeit einstellbar sein und nicht etwa schon während der Kompilierung des Kerns festgelegt werden müssen.

Transparenz Programme sollten in kleinen Adressräumen lauffähig sein, ohne neu kompiliert werden zu müssen. Weder Tasks im großen noch im kleinen Segment dürfen etwas voneinander wissen, was insbesondere einschließt, dass ihre Adressräume voreinander geschützt bleiben. Kleine Tasks dürfen nichts davon erfahren, dass ihnen weniger virtueller Speicher zur Verfügung steht. Greift solch eine Task auf Speicher außerhalb eines ihr zugeteilten Segmentes zu, muss sie vom Betriebssystem zurück in ein großes Segment verschoben und dort weiter ausgeführt werden.

Erstrebenswert wäre natürlich vollständige Transparenz gegenüber den Nutzertasks. Das ist leider aufgrund der traditionellen Speicheraufteilung von Programmen nicht möglich. Im Wesentlichen geht ein Programm davon aus, dass sich sein Stack am oberen Ende seines verfügbaren Speichers befindet und nach unten wächst, während dynamisch allokierter Speicher (Heap) im unteren Teil des Adressraumes beginnt und nach oben wächst. Theoretisch müsste deshalb eine Task, die wegen Speichermangels in einen großen Adressraum wechseln möchte, in der Mitte aufgebrochen werden, der Stack an das obere Ende verschoben werden und der Rest des Speichers an das untere Ende. Da aber alle Speicherobjekte im Code über lineare Adressen

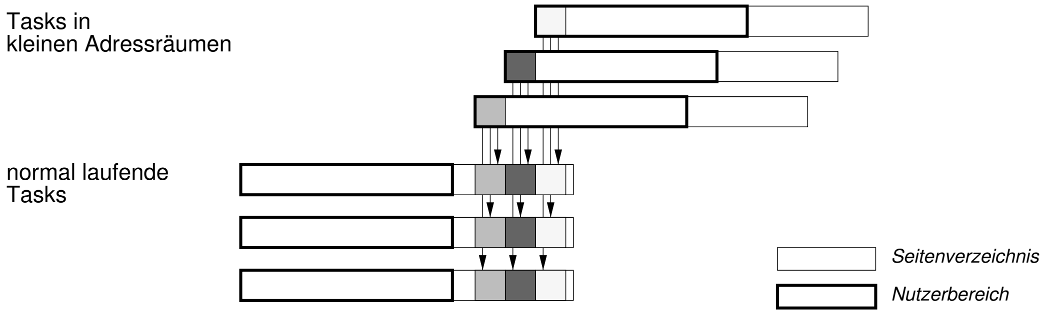

Abbildung 3.2.: Kleine Tasks kopieren einen Teil ihres Seitenverzeichnisses in andere hinein

addressiert werden, ist das nicht ohne weiteres möglich. Eine Lösung für dieses Problem zu finden, sei zukünftigen Arbeiten überlassen. Hier sei davon ausgegangen, dass Tasks, die in kleinen Adressräumen arbeiten wollen, sich selbst entsprechende Speicherbeschränkungen auferlegen. Dazu sollte jeder moderne Compiler in der Lage sein. Es kann allerdings erfordern, dass das Programm neu gelinkt werden muss.

3.3. Design

Um den Aufwand für die Verwaltung der kleinen Adressräume gering zu halten, genügt im Folgenden ein einfacheres Speichermodell. Es interessiert nur noch der virtuelle Speicher, welchen wir aus Sicht der Seitenverzeichnisse betrachten. Der Speicher unterteilt sich in 4 MB große Seiten, die jeweils durch Einträge im Verzeichnis charakterisiert sind. Ob sich dahinter eine Seitentabelle mit einer Reihe von 4-KB-Rahmen oder ein zusammenhängender 4-MB-Rahmen verbirgt, spielt keine Rolle. Des Weiteren sollten Tasks unabhängig davon, ob sie sich in großen oder kleinen Segmenten befinden, ihr eigenes Seitenverzeichnis behalten. Das „Einblenden" kleiner Adressräume beschränkt sich in diesem Modell darauf, die ersten Einträge aus dem Seitenverzeichnis der kleinen Task in das Wirtsseitenverzeichnisses zu kopieren, und zwar an die Stelle, an der sich das Segment dieser Task befindet.

Es sind nun drei große Aufgaben zu erledigen. Zuerst einmal muss eine Schnittstelle zur Nutzung der kleinen Adressräume erstellt werden. Damit mehrere Tasks in einem virtuellen Adressraum laufen können, muss FIASCO als Nächstes das lineare Speichermodell aufgeben und es muss ein segmentorientiertes Modell implementiert werden. Zum Schluss ist schließlich noch die Speicherverwaltung an die veränderten Bedingungen anzupassen.

3.3.1. Die Schnittstelle

Zu entscheiden welche Task geeignet ist, in welcher Art von Adressraum zu laufen, ist nicht Sache eines Mikrokerns. Daher genügt es, einen Systemruf bereitzustellen, der eine Task in ein bestimmtes Segment verschiebt. Im L4-Systemruf `thread_schedule` ist dafür bereits ein entsprechender Parameter vorgesehen. In ihm ist sowohl die Größe kleiner Segment codiert, als auch die Information, in welchen der kleinen Segmente die Task verschoben werden soll. Die gewählte Größe gilt für alle kleinen Segmente. Das Task-Fenster wird also in gleiche Teile geteilt. Weist ein Thread während eines Systemrufes eine neue Größe zu, werden alle sich noch in kleinen Segmenten befindliche Tasks daraus entfernt und das Layout des Task-Fensters angepasst. Eine Fehlerrückgabe an den Aufrufer gibt es nicht. Ist also der angeforderte Adressraum nicht vorhanden oder schon zugewiesen, schlägt die Verschiebung ohne Fehlermeldung an den Aufrufer fehl. Die vollständige Syntax von `thread_schedule` findet sich in Anhang A.

Durch den Systemruf kann eine Task also direkt durch Positionsangabe oder indirekt durch Änderung der Größe in ein anderes Segment gelangen. Eine weitere implizite Verlagerung ist möglich, wenn ein Thread im kleinen Segment versucht, auf Speicher außerhalb dieses Segmentes zuzugreifen. Dann muss die ganze Task wieder einen vollständigen eigenen Adressraum bekommen. Während die ersten beiden Operationen relativ unkritisch sind – der systemrufende Thread kann erwarten, kurz blockiert zu werden – kann in letzterem Fall der Thread erst weiterlaufen, wenn sich seine Task wieder im großen Segment befindet. Daher sollte zumindest eine solche Verlagerung immer erfolgreich abzuschließen sein. Um das zu erreichen, dürfen im Code keine Annahmen über die Position einer Task gemacht werden.

3.3.2. Segmente

FIASCO benutzte bisher ein flaches Speichermodell und versuchte so das Setzen der Segmentregister weitestgehend zu vermeiden. Einzig beim Betreten und Verlassen des Kern-Modus wurden Code- und Stack-Segmentregister neu gesetzt. Für die kleinen Adressräume benötigt nun die Nutzertask ihre eigenen eingeschränkten Segmente. Der Kern hingegen muss weiterhin auf den kompletten Speicher zugreifen können, da er ein festes Speicherlayout erwartet. So wird zum Beispiel in FIASCO der physische Speicher am oberen Ende des virtuellen Speichers eingeblendet. Auf Objekte, von denen nur die physische Adresse bekannt ist, wird dort zugegriffen.

Um die verschiedenen Segmentbereiche für die großen und kleinen Adressräume zu implementieren, ist eine offensichtliche Herangehensweise, für jeden von ihnen einen Segmentdeskriptor in der GDT anzulegen. Dann sollte sich das Umschalten von und nach kleinen Adressräumen darauf beschränken, die Segmentregister entsprechend neu zu setzen. Bei der Realisierung dieses Ansatz stößt man bei allerdings auf eine Reihe von Schwierigkeiten.

Da der Kern seine eigenen Segmente besitzt, können die Segmentregister erst gesetzt werden, wenn der Kern-Modus verlassen wird. Das bedeutet, dass die mehr oder weniger aufwändige Suche nach den richtigen Deskriptor nicht nur beim Umschalten von Nöten ist, sondern bei jedem Kernaustritt.

Eine weitere Frage, die sich hier stellt, ist die nach dem Schutz der Adressräume großer und kleiner Tasks untereinander. Der Schutz über die Seitentabellen kommt nicht mehr in Frage, da alle Nutzertasks auf der gleichen Privilegstufe arbeiten und gegenseitig auf ihre Daten zugreifen können, solange sie sichtbar sind. Leider gilt das gleiche für den Schutz mittels Segmenten. Die GDT selbst ist Teil des Kernspeichers und kann von Tasks nicht verändert werden. Jedoch kann jedes Segment, dass in der GDT einmal definiert wurde, in jeder Privilegstufe auch benutzt werden, solange die Rechte mit denen des Segmentes übereinstimmen. Um zu verhindern, dass eine Task aus Versehen oder böswillig einen Eintrag der GDT benutzt, der einem anderen Task zugeordnet ist, müssen inaktive Segmente durch zusätzliche Mittel verborgen werden, etwa in dem sie als nicht vorhanden markiert werden.

In jedem Fall muss beim Umschalten die GDT manipuliert werden. Daher bietet es sich an, zum ursprünglichen Aufbau mit je zwei Einträgen für Kern und Nutzer zurückzukehren und beim Taskwechsel die Deskriptoren selbst zu manipulieren, indem Basis und Größe der aktuellen Task angepasst werden. Für die Task sind nun die anderen Segmente unsichtbar. Die Segmentregister enthalten wie bisher einen konstanten Wert. Eine umständliche Suche nach dem korrekten Segment entfällt. Es darf jedoch nicht vergessen werden, sie neu zu laden, wenn die Deskriptoren geändert werden, damit ihr dazugehöriger Cache aktualisiert wird.

3.3.3. Paging

Damit bleibt noch die Aufgabe, die Seitenverzeichnisse kleiner Tasks in den Bereich anderer Adressräume einzublenden, der ihrem Segmente entspricht. Wirtsadressraum ist dabei potentiell jede existierende Task. Eine kleine Task muss also ihr Seitenverzeichnis in jedes andere kopieren und auch Änderungen überall reflektieren.

Wird eine Seite neu eingelagert, ist es noch möglich, das Problem einfach durch den klassischen Seitenfehler-Mechanismus behandeln zu lassen. Die fehlende Seite löst einen Seitenfehler aus. Während dessen Behandlung stellt der Kern fest, dass es sich um eine Adresse im Task-Fenster handelt, ermittelt, zu welcher Task sie gehört, und übernimmt den entsprechenden Eintrag aus dem Seitenverzeichnis der kleinen Task. Ist der Eintrag auch dort nicht vorhanden, wird die bisher auch verwendete Methode zur Seitenfehlerbehandlung benutzt. Die geforderte Seite wird in das Seitenverzeichnis der Task eingetragen und von dort aus an die entsprechende Stelle im Wirtsadressraum kopiert.

Leider lassen sich bei dieser Vorgehensweise keine Änderungen an Seiteneinträgen entdecken, denen bereits eine physische Adresse zugeordnet war, so zum Beispiel beim Ändern der Zugriffsrechte oder beim Entfernen einer Seitenzuordnung aus dem Adressraum. Außerdem kommt es bei Systemen, die mit vielen kurzlebigen Tasks arbeiten[1], zu einer wahren Flut von Seitenfehlern, da jede Seitenfehlerbehandlung gleich darauf durch das Löschen der Task wieder zunichte gemacht wird.

Der zweistufige Aufbau der Seitentabellen ermöglicht einen Ansatz, bei dem Änderungen im Seitenverzeichnis vermieden werden können. Dazu lege man bereits bei Betriebssystemstart für den Bereich der kleinen Adressräume feste Seitentabellen an, die sogleich in ein Master-Seitenverzeichnis eingetragen werden. Von dort werden sie auch in das Seitenverzeichnis einer jeden neu angelegten Task übernommen. Müssen jetzt Änderungen an der Seitenzuordnung vorgenommen werden, so dürfen diese nur in die fixen Seitentabellen übertragen werden. Da sich die Zeiger auf diese Seitentabellen niemals ändern, hat jede Task garantiert die aktuelle Sicht auf die kleinen Adressräume.

Das Verfahren bringt zwei große Probleme mit sich. Beim Verlagern einer Task in den kleinen Adressraum müssen, anstatt nur die Einträge der Seitenverzeichnisse zu kopieren, vollständige Seitentabellen übernommen werden. Im schlimmsten Falle wird die 1024-fache Menge an Einträgen kopiert. Ein wenig relativert sich der Aufwand jedoch dadurch wieder, dass garantiert nur eine Kopie gemacht werden muss.

Weitaus schwerer wiegt die Tatsache, dass auch einstufige Seitentabellen unterstützt und von FIASCO genutzt werden. In diesem Fall geschieht die Adressauflösung und dementsprechend alle Änderungen direkt im Seitenverzeichnis. Will man für diese Adressbereiche nicht auf eine der anderen hier beschriebenen Strategien ausweichen, müssen diese Adresszuweisungen künstlich auf zwei Stufen erweitert werden. Abgesehen von dem hohen Aufwand dafür, entsprechende

[1] Also jedes System, welches der UNIX-Philosophie „Jeder Aufgabe sein Tool“ anhängt, im Allgemeinen und L^{4}Linux im Speziellen.

Seitentabellen anzulegen, bedeutet das eine wesentlich stärkere Belastung für den TLB. Auch hier müssen im schlimmsten Falle statt einem plötzlich 1024 Einträge gepuffert werden.

In Hinblick auf FIASCO sind für eine solche Variante umfangreiche und potentiell fehlerträchtige Änderungen des Mapping-Algorithmus nötig, was am Ende hauptsächlich gegen seine Verwirklichung sprach.

Somit bleibt nichts anderes übrig, als alle Änderungen an privaten Seitenverzeichnissen in alle Adressräume zu kopieren. Wollte man dies innerhalb der Mapping-Operationen tun, bedeutete das, durch alle Adressräume zu iterieren und die Änderungen dorthin zu übertragen. Weil das kostspielig ist, muss eine solche Operation unterbrechbar sein. Das allerdings hat zur Folge, dass sich die Menge der Adressräume während des Traversierens ändern kann. Bedenkt man nun, dass ein Teil der Seitenverzeichnisse die Änderung gar nicht benötigt, weil sie nie als Wirtsadressraum dienen, erscheint der dafür zusätzlich nötige Aufwand nicht gerechtfertigt.

Der einzige Adressraum, der Änderungen sofort sehen muss, ist der aktuell eingeblendete. Für alle anderen gilt, dass sie die neuen Seitenzuordnungen spätestens dann übernommen haben müssen, wenn eine Task in kleinem Adressraum aktiviert wird, während der große Adressraum der sichtbare ist. Das lässt sich relativ einfach durch Versionierung des Adressbereiches für die kleinen Adressräume erreichen.

Dazu bekommt jede private Seitentabelle eine Versionsnummer, die den Zeitpunkt der letzten Änderung im kleinen Speicherbereich wiedergibt. Da Änderungen in der Einprozessorumgebung immer sequentiell erfolgen, genügt ein einfacher Zähler. Des Weiteren wird eine Master-Kopie des Speicherbereiches für kleine Adressräume benötigt.

Wird das Seitenverzeichnis einer kleinen Task nun geändert, werden die modifizierten Einträge zuerst in die Master-Kopie übertragen und dort die Version inkrementiert. Außerdem muss, wie oben erwähnt, das aktuell eingeblendete Seitenverzeichnis aktualisiert werden. Damit sind alle Einträge sichtbar, die Änderung abgeschlossen.

Wenn das nächste Mal auf einen Task in einem kleinen Adressraum umgeschaltet wird, muss mittels der Version überprüft werden, ob dessen Seitenverzeichnis gegenüber dem Wirtsverzeichnis inzwischen geändert wurde. Ist das der Fall, wird der Wirt aktualisiert und dessen Version angepasst.

3.3.4. Daten aus dem Nutzeradressraum: IPC

Beim IPC werden Daten zwischen Threads potentiell unterschiedlicher Adressräume übertragen. Wie in Abschnitt 2.2.3 beschrieben geschieht das beim Short IPC nur mit Hilfe der CPU-Register. Deshalb wird er von kleinen Adressräumen nicht berührt. Anders sieht es beim Long IPC aus.

Um Daten in den Adressraum des empfangenden Threads zu kopieren, kann das IPC-Fenster zunächst weiterverwendet werden. Kleine Tasks können das Fenster des Wirtsadressraums mitbenutzen, da es bei jedem Threadwechsel gelöscht wird. Es besteht kein Unterschied zu dem Fall, dass zwei Threads der gleichen Task gleichzeitig Long IPC ausführen und sich das Fenster teilen müssen.

Beim Kopieren aus dem Adressraum des sendenden Threads werden jedoch vom Kern Speicherbereiche aus Nutzeradressraumes gelesen. Unglücklicherweise haben Kern und Nutzer jetzt zwei unterschiedliche Sichten auf den Adressraum. Für den Nutzer sind Segmente transparent. Mit dem IPC-Systemruf werden von ihm daher logische Adressen relativ zu seinem Segment geliefert. Der Kern jedoch arbeitet mit linearen Adressen und muss diese Zeiger deshalb anpassen.

Zum einen besteht die Möglichkeit, die linearen Adressen „per Hand“ zu berechnen. Die Dimensionierung des gerade verwendeten Segments ist dem Kern bekannt. Er kann dessen Basisadresse zum übergebenen Zeiger addieren und den neu entstandenen Zeiger wie den alten verwenden. Vorteil dieser Methode ist, dass sie sehr einfach ist und nur minimale Änderungen an vorhandendem Code erfordert. Der Nachteil liegt darin, dass zwischen Berechnung und Verwendung eines solchen Zeigers der Kern nicht unterbrochen werden darf. Während einer Unterbrechung kann jemand die Task in ein anderes Segment verlagern und den Zeiger dadurch unbrauchbar machen. Für einfache Zugriffe lohnt es sich, diese Möglichkeit im Kopf zu behalten, für Long IPC mit seinen potentiell sehr langwierigen Kopieroperationen kommt sie nicht in Frage.

Eine andere Möglichkeit ist, den Segmentierungsmechanismus der CPU zu benutzen. Dazu ist eines der ungenutzen Datensegmentregister mit dem Nutzerdatensegment zu füllen und dann beim Zugriff in den Nutzeradressraum zu verwenden. Da das Segmentregister bei jedem Taskwechsel ohnehin neu geladen werden muss, wird selbst bei längeren und potentiell unterbrochenen Kopieroperationen immer auf das richtige Segment zugegriffen. Ein positiver Seiteneffekt ist, dass die CPU auch über die Einhaltung der Segmentgrenzen wacht. Beim Umrechnen per Hand muss immer zuerst überprüft werden, ob die Adresse innerhalb des kleinen Segmentes gültig ist.

Nachteil ist der hohe Implementierungsaufwand. Der für diese Arbeit verwendetet Compiler gcc bietet keine Sprachkonstrukte, um für den Zugriff auf eine Variable andere Segemente als DS/ES zu verwenden. Der Rückgriff auf Inline-Assembler ist unumgänglich.

4. Realisierung in FIASCO

Der erste Teil dieses Kapitels gibt einen Überblick darüber, wo im Code Änderungen gemacht wurden und welche Klassen welche Aufgaben übernommen haben. Im zweiten Teil wird dann noch genauer auf einige interessante Aspekte eingegangen.

4.1. Änderungen und Ergänzungen im Code

Abbildung 4.1 gibt einen Überblick über die veränderten Klassen sowie deren Abhängigkeiten.

Speicherlayout

Bedingt dadurch, dass im Kernbereich für jeden Thread, der in FIASCO geschaffen werden kann, der Platz für den Threadkontrollblock freigehalten wird, ist der Kernbereich bereits zum großen Teil reserviert. Daneben ist hier der physische Speicher eingeblendet sowie das IPC-Fenster. Diejenigen Einträge in denen Informationen über die Task gespeichert sind, stehen ebenfalls nicht zur virtuellen Adressierung zur Verfügung. Für das Task-Fenster ließ sich nur noch ein 48 MB großer Bereich finden. Dies entspricht nicht der L4-Spezifikation, sollte jedoch für erste Anwendungen ausreichend sein. Die Implementierung ist so gehalten, dass das Task-Fenster beliebig verschoben und vergrößert werden kann, indem einfach das Kern-Speicherlayout geändert wird.

Die Klasse smas_t

Neu hinzugekommen ist die Klasse `smas_t`. Sie kümmert sich um die Aufteilung und Verwaltung des Task-Fensters. In direkter Realisierung des im letzen Kapitel benutzten Speichermodelles wird das Fenster als eine Menge von 4-MB-Seiten betrachtet. Für jede von ihnen ist in einer statischen Liste vermerkt, welcher Task sich darin befindet. Die wichtigsten Funktionen der Klasse sind die Einteilung des Task-Fensters in mehrere Segmente (`set_space_size()`), die Bewegung von Tasks (`move()`) sowie die Berechnung einer Task aus einer Adresse im Task-Fenster (`linear_to_small()`) für die Seitenfehlerbehandlung.

Die Klasse kmem

`kmem` stellt das Kern-Seitenverzeichnis dar. Für die kleinen Adressräume wird es als Master-Kopie des Task-Fensters verwendet. Dafür gibt es Funktionen zur Aktualisierung und Versionierung.

Zusätzlich verwaltet `kmem` auch die GDT, da sie ein Teil der Kerndaten ist. Um die Nutzersegmentdeskriptoren zu schreiben, ist die Funktionen `set_gdt_user()` hinzugekommen.

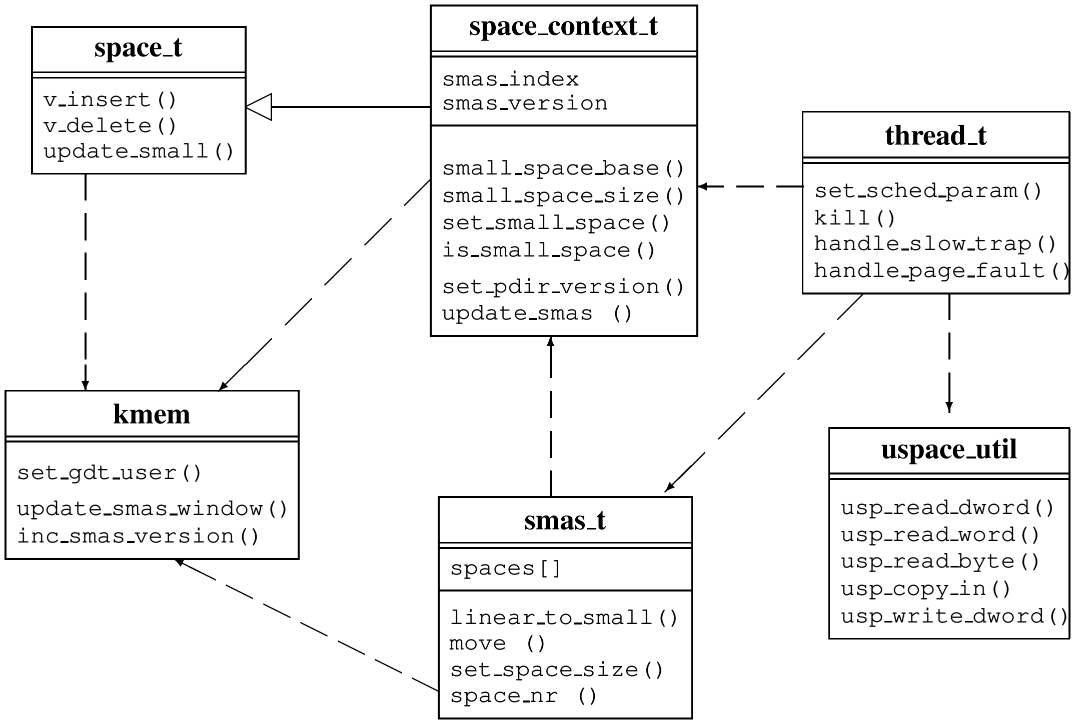

Abbildung 4.1.: Modifizierte Klassen und deren Abhängigkeiten

entry.S

Beim Kernein- und -austritt werden die Datensegmentregister neu gesetzt. GCC benutzt nur die Register DS und ES, weshalb es beim Eintritt ausreicht, diese beiden für den Kern zu laden. GS fungiert als User-Access-Segement. Sollte es nicht mehr auf das Nutzersegment zeigen, muss es neu geladen werden. Beim Austritt müssen alle vier Register korrekt gesetzt werden, wobei davon ausgegangen werden kann, dass FS und GS nicht verändert wurden. Sollte sich das Nutzersegment während eines Taskwechsels verschieben, werden beide Register sofort nach der Änderung der GDT neu geladen.

Die Klasse space_context_t

Diese Klasse stellt das Seitenverzeichnis einer Task dar und ist beim Taskwechsel dafür verantwortlich, sich selbst einzublenden. Daher muss ihr für die kleinen Adressräume bekannt sein, in welchem Segment sie läuft. Als Wirtsadressraum ist in ihr auch die Version des Task-Fensters zu merken. Beide Variablen sind in ungenutzen Einträgen des Seitenverzeichnisses gespeichert.

Die Klasse space_t

Die beiden Funktionen `v_insert()` und `v_delete()` sind die einzigen Operationen die Veränderungen an Seitenverzeichnissen vornehmen. Somit hat sich erfreulicherweise die Implementierung des Paging auf diese beschränkt. Noch mehr vereinfacht wurde es dadurch, dass niemals bestehende Zuordnungen geändert werden, sondern nur gelöscht und wieder eingefügt werden können. Dazu mehr im Abschnitt 4.2.2.

Die Klasse thread_t

Bedingt durch ihre Vielseitigkeit, waren die Änderungen an dieser Klasse recht umfangreich:

- Erweiterung des `thread_schedule`-Systemrufs
- Seitenfehlerbehandlung für Adressen im Task-Fenster
- Behandlung von allgemeinen Schutzverletzungen, die auf Überschreitung eines kleinen Segmentes zurückzuführen sind
- Anpassung der Nutzeradressraumzugriffe beim LongIPC sowie bei der Interruptbehandlung im `slow_trap_handler()`

Die Implementierung des `thread_schedule` Systemrufs folgt nicht ganz der L4-Spezifikation. Wird im Parameter für kleine Adressräume 0 übergeben, verbleibt der Task in ihrem Adressraum, anstatt, wie verlangt, in den großen zurückzukehren. Diese kleine Änderung war nötig, da mehrere Anwendungen in dieser Hinsicht fehlerhaft implementiert waren, so zum Beispiel der Ressourcemanager RMGR und L^4Linux.

Da im `slow_trap_handler()` Zeiger auf den Nutzeradressraum bis an den Kerndebugger durchgereicht werden, wurde hier auf die Methode der Adressumrechnung zurückgegriffen. Allerdings kann nicht garantiert werden, dass der Kern zwischen Berechnung und Zugriff nicht unterbrechbar ist. Hier muss in Zukunft auch der Kerndebugger noch angepasst werden.

Hilfsfunktionen in uspace_util.cpp

Inline-Assembler-Aufrufe für Zugriffe auf den Nutzeradressraum sind in dieser Datei gekapselt. Neben Funktionen zum Lesen von Bytes, Wörtern und Doppelwörtern, ist hier auch ein `memcpy` implementiert, welches das GS-Register nutzt. Letzteres versucht zwar zumindest doppelwortweise zu kopieren, ist aber ansonsten wenig optimiert. Hier besteht noch Potential.

4.2. Spezielle Aspekte

4.2.1. Taskwechsel

Nicht unbedingt überraschend, war `switchin_context()` der Klasse `space_context_t` eine der kritischsten Stellen der Implementierung bezüglich der Performanz. Es erwies sich als sehr lohnenswert, etwas mehr Zeit für die Optimierung dieser Funktion aufzuwenden. Bisher wurde in FIASCO nicht zwischen Thread- und Taskwechsel unterschieden. In jedem Fall wurde `switchin_context()` aufgerufen, dass dann das PDBR nur neu lud, wenn sich der neue vom alten Wert unterschied. Während das bisher keine Auswirkungen hatte, bedeutet es nun, dass auch beim Umschalten zwischen Threads gleicher Task unnötigerweise der hier vorgestellte erweiterte Taskwechsel vollständig zu durchlaufen ist.

Die Umschaltung beinhaltet jetzt zwei Teile: Setzen des Seitenverzeichnisses und Verändern der GDT-Einträge.

Kleine Tasks vergleichen die Version des Task-Fensters im zur Zeit eingeblendeten Seitenverzeichnisses mit dem des Kerns und kopieren, wenn nötig den Inhalt. Das PDBR wird von ihnen nicht geändert. Große Tasks wechseln, wie sie das bisher getan haben, bei Bedarf das PDBR aus, kümmern sich jedoch nicht um die Versionierung.

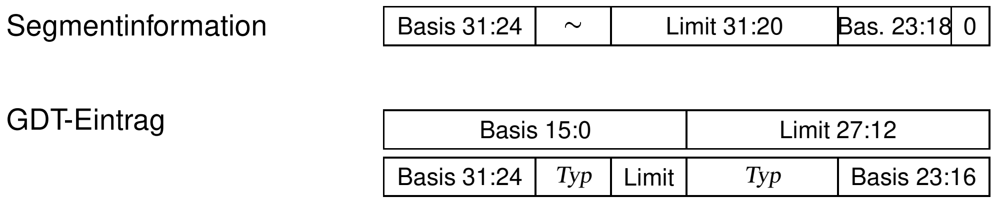

Abbildung 4.2.: Aufbau der Segmentinformation in Relation zum GDT-Eintrag

Danach werden für kleine und große Tasks in gleicher Weise die Segmentdeskriptoren der GDT neu geschrieben. An dieser Stelle erwies es sich als wichtig, dass sich der Aufbau der Segmentinformationen in der Klasse `space_t` an dem der GDT-Einträge orientiert. So kann die Segmentinformation, abgesehen von einigen Bitmanipulierungen, direkt in die GDT übertragen werden. Abbildung 4.2.1 verdeutlicht dies.

Neben den Speicherzugriffen, die durch die Manipulation der GDT entstehen, müssen außerdem noch die Segmentregister neu geladen werden, was ebenfalls aufwändig ist. Um beides beim Wechsel zwischen Threads der gleichen Tasks zu sparen, merkt sich `kmem` zusätzlich die Segmentbeschreibung der aktuellen Task und schreibt die GDT-Einträge nur neu, wenn sich deren Inhalt auch wirklich ändert.

4.2.2. Paging

Wie bereits erwähnt, werden zum Schreiben in den Nutzerbereich der Seitenverzeichnisse in FIASCO konsequent nur die beiden Funktionen `v_insert()` und `v_delete()` der Klasse `space_t` verwendet. Indem die Versionierung der kleinen Adressräume auf dieser relativ niedrigen Ebene ansetzt, war es möglich, dass die Mappingdatenbank, einer der komplexesten Teile des FIASCO-Kernes, nichts von kleinen Adressräumen zu wissen braucht. Mappings werden von ihr in das Seitenverzeichnis eingetragen, welches dann selbst dafür verantwortlich ist, die Änderung wenn nötig an die Wirtsverzeichnisse zu propagieren.

Das Kernseitenverzeichnis `kmem` dient als Master-Kopie, also als dasjenige Seitenverzeichnis, dass immer eine aktuelle Sicht des Task-Fensters enthält. Da beim Neuanlegen einer Task der Kernbereich, in dem sich auch das Task-Fenster befindet, immer daraus in das Seitenverzeichnis kopiert wird, erreicht man auf diese Weise, dass neue Tasks automatisch eine aktuelle Sicht auf die kleinen Adressräume erhalten.

Versioniert wird über das ganze Task-Fenster, da im Seitenverzeichnis nicht ausreichend Platz ist, um mehrere Versionsnummern zu speichern. Beim Umschalten zu einer kleinen Task wird die Versionsnummer geprüft und bei Bedarf das gesamte Task-Fenster kopiert. Das hat den Nachteil, dass es die Umschaltzeit zu kleinen Tasks, nachdem Mappings durchgeführt wurden, signifikant erhöht.

Damit das möglichst selten vorkommt, wurde keine reine Versionierung implementiert. Wie schon im Abschnitt 3.3.3 erläutert, können neu eingefügte Seiten auch bei der Seitenfehlerbehandlung erkannt werden. Daher werden solche Seiteneinträge in `v_insert` zwar in die Master-Kopie übernommen, jedoch wird die Version des Task-Fensters nicht erhöht.

Welche Stratgie man benutzt, hängt davon ab, wie aufwändig das Kopieren des Task-Fensters im Vergleich mit der Seitenfehlerbehandlung ist, wieviele Seitenverzeichnisse als Wirt dienen und wie oft sich Mappings ändern. Da Mapping-Operationen nicht zu den vorrangig zu optimierenden Funktionen gehören, steht eine genauere Betrachtung noch aus.

Da die Version nur in einer endlichen Variable gespeichert ist, wäre eigentlich noch der Sonderfall des Überlaufes zu betrachten. Geht man davon aus, dass Tasks in kleinen Adressräumen nicht sehr häufig wechseln, sollte auch das Seitenverzeichnis relativ stabil bleiben und eine 24-Bit-Variable ausreichen. Trotz alledem sollte der Fall behandelt werden, zum Beispiel indem durch alle Tasks iteriert und eine aktuelle Version des Task-Fensters übernommen wird.Diese Behandlung ist innerhalb diesr Arbeit noch nicht realisiert worden.

Eine Reihe von Änderungen waren nötig, weil im Code immer wieder Annahmen darüber gemacht wurden, dass ein bestimmtes Seitenverzeichnis eingeblendet ist bzw. nicht ist. So wurde zum Beispiel beim rekursiven Löschen von Mappings angenommen, dass unterliegende Adressräume nicht eingeblendet sein können und daher ein TLB-Flush unnötig sei. Diese Annahme stimmt so nicht mehr, wenn mehrere Tasks in einem Adressraum laufen. Daraus entstehende Fehler waren sehr schwer aufzuspüren, da sie nur sporadisch auftreten und oft Auswirkungen haben, die man nicht sofort auf einen Fehler im Betriebssystemkern zurückführt.

5. Leistungsbewertung

Da kleine Adressräume in erster Linie der Verbesserung der Geschwindigkeit dienen, war eine Optimierung des Codes neben der eigentlichen Implementierung ein wichtiges Ziel. Tatsächlich erwiesen sich erste Versionen als bedeutend langsamer als ein FIASCO ohne kleine Adressräume. Wo die wesentlichen Geschwindigkeitseinbußen liegen, sei hier kurz beschrieben.

Gegenüber dem Original-Kern fehlen der Implementierung mit kleinen Adressräumen (im Folgenden kurz: FIASCO-SMAS) noch einige Features, die speziell der Geschwindigkeitsoptimierung dienen. Daher macht es an dieser Stelle wenig Sinn, absolute Zahlen zu nennen. Stattdessen beschränkt sich diese Bewertung auf Vergleiche mit dem gleichartigen Original-Kern ohne die genannten Features.

Der erste Teil beschäftigt sich mit den Zeiten, die für einfache IPC benötigt werden, im zweiten Abschnitt wird dann noch ein Benchmark des L4Linux-Servers als reale Anwendung vorgestellt.

5.1. IPC-Roundtrip-Zeiten

In Tabelle 5.1 sind die durchschnittlich nötigen Takte für die verschiedenen Arten von Short und Long IPC aufgeführt. Die Messungen erfolgten auf einem 133-MHz-Pentium sowie einem P4 mit 1,6 GHz.

Zuerst einmal ist natürlich der Vergleich zwischen gleichartiger IPC im Original-Kern und in FIASCO-SMAS interessant. Hier wird ersichtlich, dass die kleinen Adressräume nicht ganz ohne Einfluss bleiben können. Messungen haben ergeben, dass der erste große Störfaktor das Neuladen aller Segmentregister beim Wechsel zwischen Kern- und Nutzermodus ist. Zweite

	Pentium			P4		
Takte	Short IPC		Long IPC	Short IPC		Long IPC
	Intra	Inter	(1000 Byte)	Intra	Inter	(1000 Byte)
FIASCO	733	1282	14353	3501	4871	15478
FIASCO-SMAS						
groß/groß	882	1497	14795	3713	5290	16072
klein/groß	-	945	14475	-	4394	15844
klein/klein	884	968	14590	3671	4376	15799
L4/Hazelnut						
groß				3226	4474	
klein				3226	4086	

Tabelle 5.1.: IPC-Roundtrip-Zeiten

unvermeidbare Stelle ist die erweiterte Semantik des Seitenaustauschens. Hier muss zusätzlich geprüft werden, ob es sich um eine kleine Task handelt und ob der GDT-Eintrag neu geladen werden muss.

Weitaus positiver fallen die Vergleiche zwischen IPC-Roundtrip-Zeiten aus, wenn ein kleiner Adressraum involviert ist. Wie zu erwarten, macht es beim Short IPC im gleichen Adressraum kaum einen Unterschied, ob es sich um einen großen oder kleinen handelt. FIASCO auf dem P4 macht sich die in Abschnitt 2.1.2 erwähnten globalen Seiten zu nutze und verhindert durch sie, dass bei einem TLB-Flush Kernseiten aus dem TLB entfernt werden. Daher sind die Einsparungen, die durch die kleinen Adressräume erzielt werden, nicht so groß wie noch beim Pentium.

Ebenfalls kaum Unterschiede gibt es beim Long IPC. Auch das war zu erwarten, da Long IPC das IPC-Fenster benutzt. Wenn es bei jedem Taskwechsel aus dem Adressraum entfernt wird, wird auch der TLB immer gelöscht und die Vorteile kleiner Adressräume werden hinfällig.

Vergleich mit L4/Hazelnut

Die Implementierung kleiner Adressräume in L4/Hazelnut folgt im Wesentlichen dem hier vorgestellten Entwurf. Jedoch wird beim Ausblenden von Seiten in kleinen Segmenten keine Versionierung genutzt und stattdessen durch alle Adressräume iteriert. Außerdem unterstützt es globale Seiten auch für die kleinen Adressräume.

Auch beim Vergleich mit Hazelnut sollte beachtet werden, dass es sich dabei nur um Vergleichswerte und nicht um absolute Zahlen handeln kann. Da das gleiche Testprogramm verwendet wurde, wird der Kern wie bei FIASCO-SMAS über den `int30` betreten, nicht über den speziell optimierten `sysenter/sysexit`-Pfad.

5.2. L^{4}Linux Benchmarks

L^{4}Linux ist ein speziell für L4 angepasster Linux-Kern, der als Server auf dem Mikrokern läuft und so die Ausführung von Linux-Binaries als Tasks in L4 erlaubt. Er bietet sich zum Leistungstest an, da er relativ komplex ist, eine ganze Reihe von Tasks beschäftigt, die auf ihn zugreifen und außerdem bereits Benchmark-Programme für Linux existieren. Für die Tests wurde der Mikrobenchmark LMBench und L^{4}Linux$\tilde{2}$.2.20 auf einem Pentium 133 verwendet. Der L^{4}Linux-Server blendet an den Anfang des virtuellen Adressraumes den physischen Speicher ein und allokiert darüber allen weiteren vom Kern benötigten Speicher. Damit er noch in einem kleinen Adressraum Platz findet, wurde FIASCO-SMAS so verändert, dass eine einzelne Task das vollständige Task-Fenster von 48 MB verwenden kann und der physische Speicher von L^{4}Linux wurde auf 32 MB begrenzt.

In den Ergebnissen spiegeln sich die bei den IPC-Rountrip-Zeiten gemachten Beobachtungen wieder. Operationen innerhalb eines Prozesses, wie etwa einfache Speicherzugriffe, werden von den kleinen Adressräumen nicht berührt. Deutliche Verbesserungen werden bei allen Aufgaben erreicht, die den L^{4}Linux-Kern benötigen. Wie beim einfachen IPC können beim einfachen Systemruf etwa 30% eingespart werden.

Die schlechten Zeiten für den Kontextwechsel liegen in der Art und Weise begründet, wie LMBench die Messungen vornimmt. In einem Pipe-Ring wird die Zeit ermittelt, die benötigt wird, ein Byte durch diesen Ring zu schicken und dann der Overhead subtrahiert, der durch die Verwendung der Pipe selbst entsteht. Dieser Overhead wird berechnet aus der Zeit, die zum

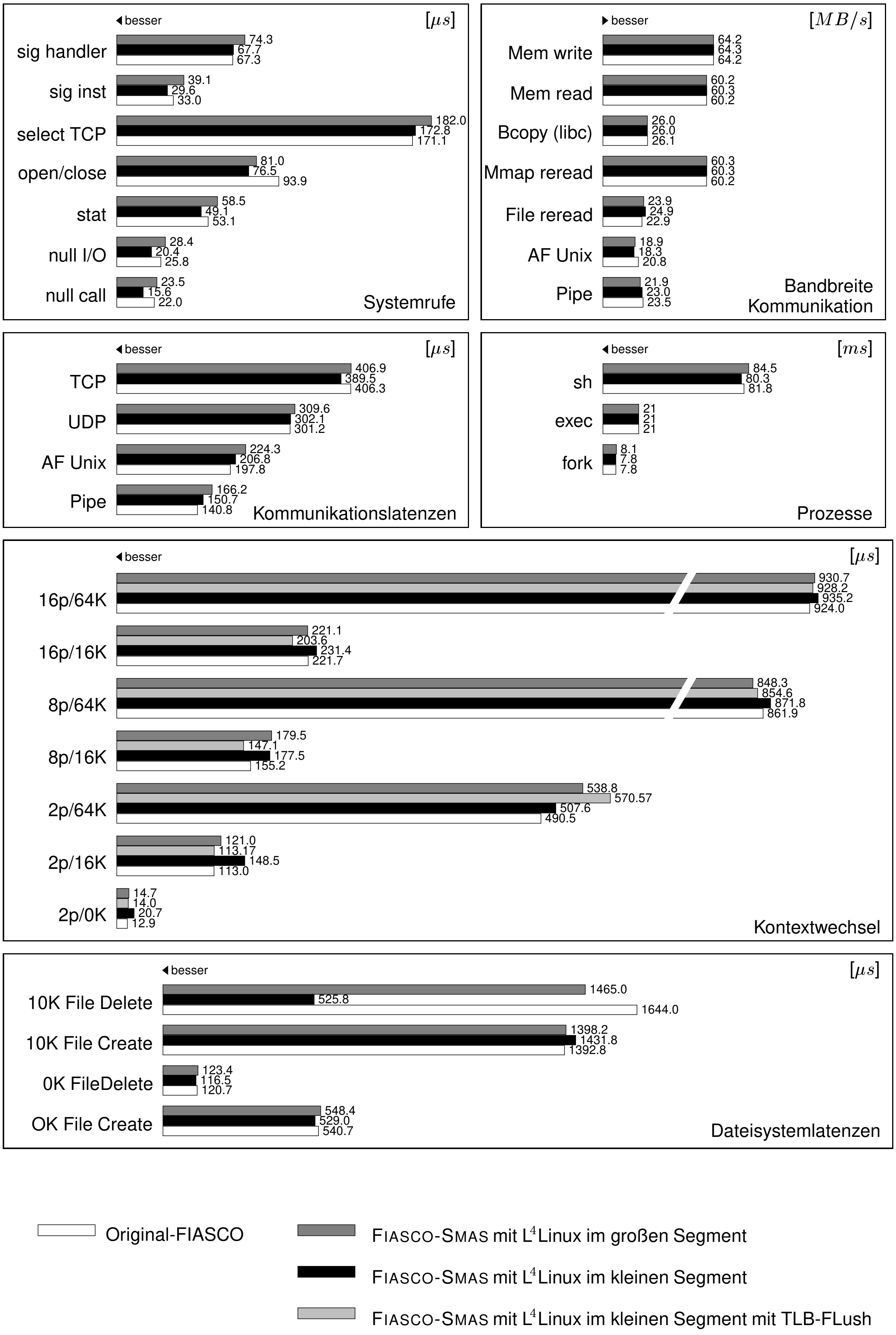

Abbildung 5.1.: Ergebnisse des Mikrobenchmarks LMBench

Schreiben und Lesen einer Pipe im gleichen Prozess benötigt wird. Im Standard-Linux ist hierzu kein Taskwechsel nötig, in L^{4}Linux muss jedoch, da beides Systemrufe sind, zur Linux-Server-Task umgeschaltet werden. Im Original-FIASCO wird dabei der TLB gelöscht, nicht aber in FIASCO-SMAS, wenn L^{4}Linux im kleinen Adressraum läuft. Bei einer Pipe zwischen zwei Prozessen erfolgt jedoch in jedem Fall ein TLB-Flush, da alle Linux-Prozesse in großen Adressräumen laufen. Das bedeutet, dass der Verlust, der durch das Neuladen des TLB-Cache entsteht, in einem L^{4}Linux im großen Adressraum als Teil des Overheads betrachtet wird, im kleinen Adressraum nicht. Wird FIASCO-SMAS so modifiziert, dass der TLB beim Umschalten in einen großen Adressraum immer gelöscht wird, werden für den Kontextwechsel etwas gleiche Zeiten wie für L^{4}Linux im großen Adressraum benötigt.

6. Zusammenfassung und Ausblick

Ziel dieses Beleges war es, kleine Adressräume in FIASCO zu implementieren und so zu ermöglichen, mehrere Tasks transparent im selben virtuellen Adressraum auszuführen. Dazu wurde in FIASCO zum ersten Mal die Segmentierung der IA32-Architektur verwendet und die Speicherverwaltung erweitert.

Bisher funktionieren kleine Adressräume nur in der Standard-Implementierung des FIASCO-Kerns. Im Rahmen des Leistungsanalyse-Projektes [Pet02] werden seit Neustem spezielle Geschwindigkeitsoptimierungen geboten. So wird `sysenter/sysexit`, ein besonderer Mechanismus zum Kerneintritt, unterstützt und ein spezieller in Assembler geschriebener IPC-Pfad geboten, der für häufigsten Fälle der Short IPC optimiert ist. Um FIASCO-SMAS praktisch nutzbar zu machen, ist eine Unterstützung dieser Features dringend notwendig. Gleiches gilt für I/O-Flexpages.

Wie schon im Kapitel 5 erwähnt, besteht bei Long IPC noch gewaltiges Verbesserungpotential, wenn beim Kopieren zwischen Adressräumen nicht mehr das IPC-Fenster verwendet wird sondern direkt die kleine Segmente. Das wird allerdings durch die Unterbrechbarkeit ein etwas schwierigeres Problem.

FIASCO für Multiprozessoren [Pet01] wird demnächst verfügbar sein. Ob dafür eine Unterstützung kleiner Adressräume überhaupt sinnvoll ist und wie sie aussehen kann, muss noch ausdiskutiert werden.

Mit diesen Ergänzungen kann der schon in dieser ersten Realisierung erreichte Geschwindigkeitsgewinn hoffentlich in Zukunft noch ausgebaut werden. Wünschenswert ist außerdem, dass durch weitere Verbesserungen der Implementierung selbst die Verluste, die gegenüber Standard-FIASCO entstehen noch verringert werden können.

Am Ende bleibt bei dieser Art von Optimierung immer der bittere Beigeschmack, dass sie eigentlich in erster Linie Fehler der Hardware-Architektur ausgleichen soll. Für kleine Adressräume ist das eben die fehlende Kennzeichnung von Einträgen im TLB.

Wird die Idee der geteilten Adressräume weiter getrieben, führt das in letzter Konsequenz zu Single-Space-Betriebssystemen wie Mungi [Mun], die virtuellen Speicher wieder vollständig in Software realisieren. Zu beurteilen inwieweit das ein Fort- oder Rückschritt ist, sei jedem selbst zu überlassen.

A. Systemruf thread_schedule

(Auszug aus „L4-Pentium Implementation“ [Lie96a])

param word	EAX		EAX	*old param word*
~	ECX		ECX	*time.low*
~	EDX		EDX	*time.high*
ext preempter.low	EBX	INT 0x34	EBX	*old preempter.low*
ext preemter.high	EBP		EBP	*old preempter.high*
dest id.low	ESI		ESI	*partner.low*
dest id.high	EDI		EDI	*partner.high*

Der Systemruf kann von Schedulern benutzt werden, um Priorität, Länge und externen Preempter anderer Threads festzulegen. Außerdem liefert er den Status des Threads zurück. Zu beachten ist, dass aus Sicherheitsgründen Statusinformationen des Threads nur vom zuständigen Scheduler erfragt werden können.

Der Systemruf hat nur eine Wirkung, wenn die Priorität des angegebenen Zieles kleiner oder gleich der *maximum controlled priority (mcp)* der aktuellen Task ist.

Parameter

param word

$mr_{(8)}$	$er_{(4)}$	$0_{(4)}$	$small_{(8)}$	$prio_{(8)}$

small — Kontrolliert die Zuordnung von kleinen Adressräumen zur gewünschten Task.

$= 0$ — Die Task wird aus einem vorher zugeordneten kleine Adressraum entfernt, das heißt, sie hat danach nur noch den gewöhnlichen großen Adressraum.
Anmerkung: In FIASCO-SMAS *wird die Zuordnung nicht verändert. Siehe Abschnitt 4.1.*

$= 2ks + s$ — Der Task wird der kleine Adressraum k zugeteilt. Die Größe des kleinen Adressraumes wird durch s festgelegt. Erlaubte Kombinationen von k und s sind:

s	Größe kleiner Adressräume	k
1	4 M	1…127
2	8 M	1…63
4	16 M	1…31
8	32 M	1…15
16	64 M	1…7
32	128 M	1…3
64	256 M	1

Zu beachten ist, dass alle gleichzeitig zugeordneten kleinen Adressräume die gleiche Größe haben müssen. Wird ein kleiner Adressraum mit einer anderen Größe zugewiesen, werden alle existierenden Zuordnungen implizit aufgelöst.

$= FF$ — Die zur Zeit gültige Zuweisung (oder die nicht vorhandene Zuweisung) der Task wird nicht verändert.

old param word

$mr_{(8)}$	$er_{(4)}$	$0_{(4)}$	$small_{(8)}$	$prio_{(8)}$

small — Gibt die aktuelle Zuordnung der gewünschten Task zu kleinen Adressräumen zurück.

$= 0$ — Zu keinem kleinen Adressraum zugeordnet.

$= 2ks + s$ — Kleiner Adressraum k mit der Größe $sx4$M zugeordnet. (Siehe obere Tabelle)

alle anderen Parameter — siehe L4 Reference Manual [Lie96b]

Literaturverzeichnis

[Hoh96] Michael Hohmuth. Linux-Emulation auf einem Mikrokern. Diplomarbeit, TU Dresden, 1996.

[Hoh98] Michael Hohmuth. The FIASCO Kernel: Requirements definition. Technical report, Dresden University of Technology, 1998.

[Int99] Intel Corp. *Intel Architecture Software Developers Manual, Volume 3: System Programming*, 1999.

[Lie95] Jochen Liedtke. Improved address-space switching on Pentium processors by transparently multiplexing user address spaces. Arbeitspapiere der GMD No. 933, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1995.

[Lie96a] Jochen Liedtke. *L4-Pentium Implementation*. IBM T.J. Watson Research Center, 1996.

[Lie96b] Jochen Liedtke. *L4 Reference Manual - 486, Pentium, Pentium Pro*. GMD – German Nation Research Center for Information Technology, 1996.

[Mun] Mungi – Single-Address-Space Operating System.
`URL: http://www.cse.unsw.edu.au/~disy/Mungi`.

[Pet01] Michael Peter. Portierng des FIASCO μ-Kernel auf SMP-Systeme. Großer Beleg, TU Dresden, 2001.

[Pet02] Michael Peter. Leistungsanalyse und -optimierung des L4Linux-Systems. Diplomarbeit, TU Dresden, 2002.

[RJO+89] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alesandro Forin, David Golub, and Michael B. Jones. Mach: a system software kernel. In *Proceedings of the 1989 IEEE International Conference, COMPCON*, pages 176–178, San Francisco, CA, USA, 1989. IEEE Comput. Soc. Press.

[TUD] FIASCO-Webseiten. Institut Betriebssysteme, Fakultät Informatik, TU Dresden, `http://os.inf.tu-dresden.de/fiasco`.