Diplomarbeit:
Linux-Emulation auf einem Mikrokern

Michael Hohmuth

29. August 1996

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die Forschung am Lehrstuhl Betriebssysteme der TU Dresden konzentriert sich auf echtzeitfähige Betriebssysteme, die auf den Mikrokernen L3 und L4 basieren. Ziel ist die Entwicklung einer Multi-Server-Architektur, die Anwendungen mit Quality-of-Service-Anforderungen unterstützt.

Um mit diesen Mikrokernen effektiv arbeiten und entwickeln zu können, steht momentan die Schaffung einer stabilen und benutzbaren Unix-Emulation auf dem Mikrokern L4 im Vordergrund.

Traditionelle Unix-Emulationen auf Mikrokernen, ob als Verlagerung eines monolithischen Kerns in einen einzigen Server auf Nutzer-Ebene oder realisiert durch mehrere kooperierende Server, erreichten im allgemeinen nicht die Leistung vergleichbarer monolithischer Systeme.

Um diese Leistungsverluste zu umgehen, wurden in der Vergangenheit zwei Ansätze verfolgt:

- Binden von Server-Code in den Adreßraum des Mikrokerns. Dadurch fallen viele Taskumschaltungen weg, und entfernte Prozeduraufrufe zwischen in den Kern gebundenen Servern werden zu lokalen Aufrufen.

  Der damit erzielte Leistungsgewinn geht natürlich zu Lasten der Systemsicherheit, weil der Mikrokern nun nicht mehr vor den Servern geschützt ist; daher kommt dieser Ansatz nur für vertrauenswürdige Server in Frage.

  Vertreter dieser Kategorie sind Mklinux [1] und SPIN [2].

- Erstellung „schlanker“ Mikrokerne und Server, die nur eine für eine bestimmte Anwendung benötigte Untermenge der Unix-Funktionalität implementieren (zum Beispiel keinen virtuellen Speicher).

  Der wichtigste Vertreter dieser Kategorie ist QNX [3].

Diese Arbeit geht davon aus, daß der Grund für die Leistungsverluste im wesentlichen in der Verwendung ineffizienter Mikrokerne besteht. Da nun mit L3 und L4 sehr effiziente Mikrokerne zur Verfügung stehen [4], soll hier ein neuer Weg versucht werden, um zu einer effizienten Unix-Emulation auf einem Mikrokern zu gelangen:

- Kombination eines schnellen Mikrokerns mit einem „normalen“ Unix-Server, ohne Änderung des Kerns (kein Hinzubinden von Server-Code) und ohne Verzicht auf Funktionalität.

Um diese Unix-Emulation so bald wie möglich nutzbar zu machen, bot sich die Portierung einer bereits verfügbaren Unix-Implementation an. Nachdem sich die Portierung eines Single-Servers vom Mikrokern Mach auf L3 als nicht sinnvoll erwies [5], erwogen wir die Portierung eines monolithischen Unix-Systems auf L4 (L4 ist inzwischen einsatzbereit).

Da kürzlich die Portierung des Linux-Kerns auf den OSF-Mach-Mikrokern durch die OSF bekannt wurde [1], entschlossen wir uns, ebenfalls den Linux-Kern als Basis für unsere Arbeit zu benutzen, um auf Erfahrungen der OSF-Gruppe zurückgreifen zu können.

1.1 Über dieses Dokument

Im nächsten Kapitel stellen wir zunächst Technologien zur Unix-Emulation auf Mikrokernen sowie die Betriebssysteme L4 und Linux vor.

Ausgehend von dieser Betrachtung entwerfen wir in Kapitel 3 einen Weg zur Portierung des Linux-Kerns auf den Mikrokern L4. Die folgenden Kapitel haben dann unsere Implementation und deren Leistung zum Thema. Abschließend fassen wir in Kapitel 6 das Erreichte zusammen.

1.2 Danksagung

Ich möchte mich an dieser Stelle herzlich bei der Gruppe Betriebssysteme der TU Dresden für die gewährte Unterstützung und Teilnahme an diesem Projekt bedanken, insbesondere bei Prof. Hermann Härtig, Robert Baumgartl, Martin Borriss, Sebastian Schönberg und Jean Wolter. Mein Dank gilt auch Dr. Jochen Liedtke von der GMD, dem ich viele fruchtbare Diskussionen verdanke.

Außerdem möchte ich mich bei meinen Freunden bedanken, die mir mein Studentenleben versüßten. Da es zu viele zum Aufzählen sind, muß an dieser Stelle ihr Hash-Value ausreichen:

`MD5 (freunde) = 36f085284392f640f67c190bc112caa8`

Kapitel 2

Stand der Technik

2.1 Unix-Systeme auf Mikrokernen

In der Vergangenheit waren Unix-Implementationen auf Mikrokernen schon oft ein Forschungsthema. Motivation dafür ist der Wunsch, die Vorteile eines Mikrokerns auszunutzen (Modularität, Flexibilität, Anpaßbarkeit) und dennoch nicht auf das Betriebssystem Unix als Werkzeug und Entwicklungsplattform verzichten zu müssen.

Um auf einem Mikrokern die Unix-Funktionalität zu erbringen, wurden verschiedene Ansätze entwickelt: [6]

Single-Server. In dieser Architektur erbringt eine einzelne auf dem Mikrokern im Nutzer-Modus laufende Task die Unix-Funktionalität. Beispiele sind der Mach-Single-Server der CMU [7], Lites [8], Mklinux [1], OSF/1 und Suns Spring Unix-Server [9].

Die vier erstgenannten Vertreter sind Portierungen von monolithischen Unix-Betriebssystemen auf den Mikrokern Mach [10]. Solche Portierungen sind ein geeignetes Mittel, um schnell zu einer Unix-Emulation auf einem Mikrokern zu gelangen, und bieten sich für Single-Server an, da diese prinzipiell ein in den Nutzer-Modus verlegtes monolithisches System darstellen.

Multi-Server. Implementiert man ein Unix-System als mehrere Server-Module, die jeweils einen Teil der Unix-Funktionalität erbringen, so erhält man einen Multi-Server. Beispiele für diesen Aufbau sind der Mach-Multi-Server der CMU [11] und GNU Hurd [12].

Der klare Vorteil dieser Architektur ist die verbesserte Modularität, die es bei geeignetem Design erlaubt, einzelne Teile der Unix-Emulation bei Bedarf auszutauschen oder zu erweitern. So wird es beispielsweise möglich, durch mehrere Emulationsmodule einem Rechner gleichzeitig mehrere Betriebssystem-„Persönlichkeiten" zu geben. Ein weiteres Beispiel ist Hurds Translator-Konzept: Es ermöglicht die Interpretation beliebiger Daten als Dateisystem, vorausgesetzt es existiert ein geeigneter Translator. So ist es beispielsweise möglich, Translatoren für tar-Archive oder ftp-Server zu bauen. [12]

Allen bisherigen Ansätzen gemein ist, daß im allgemeinen die Leistung der Unix-Emulationen schlechter ausfällt als die vergleichbarer monolithischer Systeme. Bisher wurde dafür oft der Mikrokern-Ansatz insgesamt verantwortlich gemacht; jedoch konnte in neuerer Forschung gezeigt werden, daß der Leistungsverlust im wesentlichen durch die Konstruktionsprinzipien älterer Mikrokerne bedingt war. Dies führte sowohl zur Entwicklung von Optimierungsverfahren für diese Kerne (z.B. Binden von Server-Code in den Adreßraum des Mikrokerns in Mach [13] und SPIN [2]) als auch zur Konstruktion neuer Mikrokerne einer zweiten Generation wie L3, L4 und QNX. [14] [4] [3]

2.2 Der Mikrokern L4

L4 ist ein Mikrokern der zweiten Generation für Intel-CPUs, der von Jochen Liedtke an der GMD entwickelt wurde. Er wurde nach dem Minimalitätsprinzip konstruiert und implementiert nur eine kleine Anzahl von Primitiven und Kernobjekten. Aufbauend auf diesen können flexibel Betriebssysteme konstruiert werden. [4]

Implementiert sind Operationen für folgende Objekte:

- Flexibel große Seiten, **Flexpages**, die den Zugriff auf bestimmte Adressen des Speicher- und des I/O-Adreßraums der CPU erlauben.
- **Virtuelle Adreßräume**, die aus Seiten aufgebaut sind. Flexpages (Mengen von Seiten) können zwischen Tasks (s.u.) durch IPC-Operationen eingeblendet (*map*) und weitergereicht (*grant*) werden, und es gibt einen Systemruf zum Ausblenden (*flush*) von Flexpages.
- **Threads** sind Aktivitäten, die in Adreßräumen ablaufen. Sie sind durch eindeutige IDs identifizierbar und können durch IPC-Operationen miteinander kommunizieren.
- L4 kennt zwei Sicherheits-Domänen:
  - **Tasks** bestehen aus einem Adreßraum und mindestens einem darin ablaufenden Thread. Der Adreßraum einer Task kann nur durch die zugeordneten Threads manipuliert werden (**Taskautonomie**).
  - **Clans** bestehen aus einer **Chief**-Task und den von ihr erzeugten Kind-Tasks (die ihrerseite wieder Chiefs ihres eigenen Clans sind). L4 leitet jede IPC-Nachricht, die Clan-Grenzen überschreitet, zunächst an den nächsten Chief weiter. Der Chief hat die Möglichkeit, die Nachricht unter Verwendung beliebiger Algorithmen zu verarbeiten oder zu überprüfen und dann auf dem IPC-Pfad weiterzuleiten. So wird es möglich, Sicherheits-Strategien zu implementieren, um zum Beispiel das System vor Tasks eines Clans zu schützen. [15] [16]

L4 ist ein sehr kleiner, außerordentlich schneller Mikrokern:

- Sein IPC-Mechanismus ist eine Größenordnung schneller als bei traditionellen Mikrokernen.
- L4s Paging-Mechanismus ist durch seine Leichtgewichtigkeit besonders effizient.
- Die Kontextwechsel- und Adreßraumumschaltzeiten konnten durch eine geeignete Konstruktion des Kerns minimiert werden, so daß es nicht nötig ist, Code in den Kern zu binden, um eine gute Leistung zu erzielen. [4]

L4 enthält keinerlei Gerätetreiber, sondern unterstützt auf Nutzerebene implementierte Gerätetreiber durch folgende Mechanismen:

- Der I/O-Adreßraum der Intel-CPU kann, wie der Speicheradreßraum auch, durch das Versenden von Flexpages in beliebige Tasks eingeblendet werden.
- Privilegierten Tasks ist es gestattet, das CPU-Statusregister zu manipulieren, so daß Interrupts gesperrt und wieder freigegeben werden können.
- Threads können sich an Interrupt-Quellen „anschließen". Hardware-Interrupts werden vom L4-Kern in Nachrichten umgewandelt und dem angeschlossenen Thread zugestellt. [15]

Ähnlich werden auch Speicher-Manager und Scheduler auf Nutzer-Ebene unterstützt:

- Seitenfehler werden vom L4-Kern in Nachrichten umgewandelt und an einen bei der Thread-Erzeugung zu definierenden Pager-Thread gesendet. Der unterbrochene Thread wird fortgesetzt, wenn der Pager eine Flexpage zurücksendet, die den Seitenfehler auflöst.
- Scheduling auf Nutzer-Ebene wird durch sogenannte „Preemption Handler" unterstützt: Läuft die aktuelle Zeitscheibe eines unter der Kontroll eines Preemption Handlers laufenden Threads ab, so wird dieser vom L4-Kern mit einer IPC-Nachricht unterrichtet. Der Thread wird erst nach einer Antwortnachricht des Preemption Handlers fortgesetzt. [15]

2.3 Das Betriebssystem Linux

Linux ist eine frei verfügbare Unix-Implementation für Rechner mit den CPUs Intel-x86, Motorola-68k, Digital-Alpha, PowerPC und SPARC (Portierungen auf weitere Maschinen sind in Arbeit). Es unterstützt eine Vielzahl von Anwendungen traditioneller Unix-Systeme wie das X-Window-System, TCP/IP-Netzwerksoftware, Emacs u.v.a.m. [17]

Der Linux-Kern wurde von Linus Torvalds aus Helsinki und vielen freiwilligen Helfern entwickelt. Es handelt sich um einen traditionellen monolithischen Kern, der alle Aufgaben klassischer Betriebssystem-Kerne übernimmt. Dazu enthält er folgende Bestandteile:

- eine maschinenabhängige Schicht, die für die anderen Kernbestandteile Funktionen und Abstraktionen bereitstellt, die die eigentliche Hardware kapseln;
- Gerätetreiber, davon einige maschinenabhängig;
- Protokolltreiber für Netzwerk-Protokolle;
- Dateisysteme; und
- Verwaltung aller Maschinen-Ressourcen wie zum Beispiel Rechenzeit und Speicher. [18]

Durch die wohldefinierte maschinen- (oder architektur-) abhängige Schicht erreicht der Linux-Kern eine hohe Portabilität: Die restlichen Kernbestandteile sind dadurch maschinenunabhängig und müssen bei einer Portierung des Linux-Kerns auf eine andere Architektur nicht modifiziert werden.

Da es in dieser Arbeit um die Portierung des Linux-Kerns auf den Mikrokern L4 geht, wollen wir auf diese Schicht noch etwas genauer eingehen.

Der architekturabhängige Teil muß Schnittstellen für folgende Aufgabenbereiche implementieren: [19]

ABI-Definition Dieser Teil definiert die Schnittstelle zwischen Applikationsprogrammen und dem Linux-Kern (ABI = *Application Binary Interface*), zum Beispiel Nummern von Systemrufen, Signalen usw. Wenn es für eine Architektur bereits eine andere Unix-Implementation gibt, so verwendet man oft deren ABI als Anhaltspunkt, um Binärkompatibilität zu erreichen.

Gerätetreiberunterstützung Dazu gehört die Kapselung von Hardwareschnittstellen, die es auf mehreren Architekturen gibt, die aber architekturabhängig angesteuert werden müssen, wie die Floppy- und DMA-Hardware und Zugriffe auf den I/O-Adreßraum der Maschine.

Interruptverwaltung Dieser Teil stellt den Gerätetreibern und dem architekturunabhängigen Linux-Code eine architekturübergreifende Schnittstelle zur Interrupt-Hardware zur Verfügung.

Aktivitäten sind eine kern-interne Abstraktion der Linux-Prozesse.[1] Sie bestehen aus einem Prozessorzustand (Thread) und einem Kontext.

In dieser Arbeit unterscheiden wir zwischen Nutzer- und Kern-Aktivitäten. Zum Kontext einer Nutzer-Aktivität gehört im wesentlichen ein Nutzer-Adreßraum (der Adreßraum des Linux-Prozesses). Jeder Nutzer-Aktivität ist eine Kern-Aktivität zugeordnet, zu deren Kontext neben dem aktuellen Zustand der Nutzer-Aktivität ein Kern-Stack und der (von allen Kern-Aktivitäten gemeinsam benutzte) Kern-Adreßraum gehören.

Der maschinenabhängige Teil muß Methoden zur Erzeugung und zum Löschen von Aktivitäten sowie zum Umschalten zwischen ihnen bereitstellen. Ferner gehören Operationen zur Manipulation des Nutzeradreßraums (**Copy-In/Out**) und zum Betreten und Verlassen des Kern-Modus dazu.

Weitere Schnittstellen unter anderem solche zum Booten und Rebooten und zur Unterstützung für das `proc`-Dateisystem und zum Schreiben einer `core`-Datei.

[1] Aktivitäten werden im Linux-Quelltext „Threads“ genannt. Wir vermeiden diesen Begriff hier, um Aktivitäten nicht mit den L4-Threads durcheinanderzubringen.

Kapitel 3

Entwurf

3.1 Entwurfsziele

Ziel dieser Arbeit war es, den Linux-Kern auf den Mikrokern L4 zu portieren, so daß er selbst als Anwendungsprogramm im Nutzer-Modus läuft. Dabei war eine 1:1-Binärkompatibilität mit der Linux-Originalimplementation (im folgenden Linux/i386 genannt) anzustreben.

Ein wichtiger Gesichtspunkt beim Entwurf war der Wunsch, möglichst viel Code von Linux/i386 zu übernehmen: Alle i386-Gerätetreiber sollten auch unter Linux/L4 funktionieren, und die Änderungen an den architekturunabhängigen Teilen des Linux-Kerns waren zu minimieren.

Die außerordentlich gute Leistung des Mikrokerns L4 (vgl. Abschnitt 2.2) erlaubt beim Entwurf eine „natürliche" Herangehensweise. Im Gegensatz zu anderen Arbeiten sind daher spezielle Einschränkungen oder ein Binden des Codes der Unix-Emulation in den Adreßraum des Mikrokerns nicht nötig. Stattdessen ist es möglich, von den von L4 gebotenen Mitteln schöpferisch Gebrauch zu machen:

- Sämtliche Linux-Dienste werden auf Nutzerebene erbracht, und die L4-Kernfunktionalität wird nicht erweitert, sondern strikt beibehalten.
- Da in L4 im Gegensatz zu vielen anderen Mikrokern-Implementationen die IPC-, Kontextwechsel- und Adreßraumumschaltzeiten sehr klein sind, ist es möglich, für die Emulation des Linux-Kerns mehrere Tasks zu verwenden.

3.2 Interruptverwaltung

Linux-Gerätetreiber erwarten vom Linux-Kern eine bestimmte Umgebung, zu der auch die von der Architekturanpassung bereitzustellende Interruptverwaltung gehört.

(Wenn wir im folgenden von „Interrupts" sprechen, so meinen wir von externen Geräten erzeugte Hardware-Interrupts (IRQs), und nicht Software-generierte Interrupts, die zum Kerneintritt führen; letztere werden im Abschnitt 3.5 besprochen.)

3.2.1 Annahmen der architekturunabhängigen Linux-Module

- Linux teilt die Interrupt-Behandlungsroutinen in Top Half und Bottom Half. [20]

Top Halves sind kleine Programm-Fragmente, die sofort nach Eintreffen des Interrupts abgearbeitet werden. Sie sind nur durch andere Top Halves unterbrechbar und sollten daher recht schnell abzuarbeiten sein. In der Regel bestätigen Top Halves den Interrupt einem externen Gerät und markieren dann eine oder mehrere Bottom Halves zur Ausführung.

Anschließend werden die markierten Bottom Halves abgearbeitet, wenn keine anderen Top Halves mehr ausgeführt werden. Sie führen normalerweise interrupt-ausgelöste, aber zeitunkritische Programm-Fragmente aus.

Die Einhaltung der folgenden Semantik wird vorausgesetzt: Top- und Bottom-Half können nur durch Top-Halves neu auftretender Interrupts unterbrochen werden, nie durch weitere Bottom-Halves oder „normalen" Kern-Code. (Dies bedeutet auch, daß während der Abarbeitung von Interrupt-Behandlungsroutinen kein Scheduling stattfinden kann.)

- Kritische Abschnitte (sowohl in Interrupt-Behandlungsroutinen als auch im „normalen" Linux-Code) werden häufig durch zeitweises Sperren aller Interrupts im CPU-Status-Register (Operation `cli()`) geschützt.[1]

3.2.2 Top Halves

L4 stellt Interrupts einem bestimmten Empfängerthread als Nachrichten zu. Um solche Interrupt-Nachrichten zu empfangen, muß sich der Empfänger zuvor über den L4-IPC-Mechanismus an die Interrupt-Quelle „anschließen". [15]

Da L4 jedem Thread nur den Anschluß an maximal eine Interrupt-Quelle gestattet, muß für jeden Hardware-Interrupt ein separater Thread gestartet werden.

Unmittelbar nach dem Empfang der Interrupt-Nachricht muß die Interrupt-Verwaltung die Top Half der Interrupt-Behandlungsroutine abarbeiten. Dies kann sofort im jeweiligen Interrupt-Thread erfolgen; eine Serialisierung der Abarbeitung der Top Halves verschiedener Interrupts ist nicht notwendig, da Linux die Möglichkeit der Unterbrechung von Interrupt-Behandungsroutinen durch neue Interrupts explizit vorsieht.

3.2.3 Bottom Halves

In Linux dürfen Top Halves nicht für die Abarbeitung von Bottom Halves unterbrochen werden; andersherum ist die Unterbrechung der Abarbeitung einer Bottom Half durch einen Interrupt mit zugehöriger Top Half sehr wohl möglich. Konzeptionell stellen daher die Bottom Halves eine von den Interrupts unabhängige Aktivität dar.

Um diese Semantik zu implementieren, stellt Linux das Semaphor `intr_count` zur Verfügung, das bei jeder Interrupt-Behandlung durch Top Halves erhöht werden muß; Bottom Halves dürfen nur abgearbeitet werden, wenn das Semaphor auf 0 steht.

Die Bottom Halves könnten innerhalb verschiedener Aktivitäten ausgeführt werden:

1. Die auf die Interrupt-Nachrichten wartenden Threads führen jeweils nach der Top Half die anfallenden Bottom Halves aus.

2. Die Bottom Halves werden von einem einzelnen dedizierten Thread abgearbeitet, der von den Interrupt-Threads nach Ausführung einer Top Half via IPC aktiviert wird. Da L4

[1] In mit `cli()` geschützten kritischen Abschnitten dürfen Seitenfehler auftreten; dies führt unter L4 zu besonderen Problemen, auf die in Abschnitt 3.2.5 näher eingegangen wird.

statische Prioritäten unterstützt, kann die Linux-Semantik durch Höherpriorisierung der Interrupt-Threads erreicht werden.

Die erste Lösung hat den Vorteil, daß sie die Linux-Semantik automatisch bereitstellt; außerdem führt sie zu einer einfacheren Implementation. Allerdings führt diese Lösung auch zu einer größeren Interrupt-Latenz, da unter Umständen noch mehrere Bottom Halves ausgeführt werden müssen, ehe der beteiligte Thread wieder Interrupt-Nachrichten empfangen kann.

Die zweite Lösung hat dieses Latenz-Problem nicht. Außerdem unterstützt sie auch Gerätetreiber, deren Bottom-Halves auf Interrupts (und Abarbeitung der zugehörigen Top Half) blockierend warten, wie beispielsweise der Tastatur-Treiber.

Ein potentielles Problem dieser Lösung ist jedoch, daß sie nicht auf Maschinen mit mehr als einem Prozessor funktioniert. Dafür müßte genaugenommen der Linux-Code reorganisiert werden, so daß die bisherige Linux-Semantik (Top Halves nicht unterbrechbar) nicht mehr garantiert werden muß. Die SMP-Variante von Linux/i386 umgeht das Problem bisher, indem der gesamte Kernzugriff (inklusive Interrupt-Behandlung) durch ein globales Lock synchronisiert wird; ein ähnlicher Workaround wäre auch für Linux/L4 denkbar.

Wir werten die Unterstützung von mehr Gerätetreibern und eine geringere Interrupt-Latenz höher als eine einfache Implementation und entscheiden uns daher für die zweite Variante.

3.2.4 Synchronisierung der Interrupt-Aktivitäten

Linux gewährleistet den Interrupt-Behandlungsroutinen (Top und Bottom Half), daß während ihrer Abarbeitung kein „normaler“ Linux-Kern-Code abläuft. Um diese Semantik unter L4 bereitzustellen, bieten sich folgende Möglichkeiten an:

1. Interrupt-Behandlungsroutinen müssen vor der Abarbeitung von Linux-Kern-Code stets zuerst ein globales Lock setzen.

2. Die Interrupt-Threads und der Bottom-Half-Thread erhalten höhere Prioritäten als die anderen Linux-Kern-Aktivitäten.

Die erste Variante entspricht der Implementation in Linux/i386-SMP und wurde im vorigen Abschnitt bereits kurz diskutiert. Sie hat den Vorteil, daß sie auch mit mehr als einer CPU funktioniert, allerdings auf Kosten der Interrupt-Latenz.

Variante 2 funktioniert auf Multiprozessormaschinen nicht ohne weiteres. Im Sinne einer besseren Interrupt-Latenz entschieden wir uns jedoch für diese Variante, zumal es noch keine L4-Implementation für Multiprozessormaschinen gibt.

3.2.5 Wechselseitiger Ausschluß durch `cli()`

In Linux/i386 wird wechselseitiger Ausschluß innerhalb kritischer Abschnitte im allgemeinen durch Löschen des Interrupt-Enable-Flags im CPU-Status-Register realisiert (implementiert durch die Operation `cli()`). Dies wäre prinzipiell auch in Linux/L4 möglich, da L4 privilegierten Tasks den Zugriff auf den entsprechenden Teil des CPU-Statusworts gestattet.

Allerdings gibt es unter L4 ein zusätzliches Problem, das diesen Ansatz unmöglich erscheinen läßt: Seitenfehler führen in L4 zur Umschaltung zu einem Pager-Thread und unter Umständen zum Aufruf des L4-Schedulers. Das kann dazu führen, daß Threads trotz `cli()` unterbrochen werden und kein wechselseitiger Ausschluß mehr stattfindet.

Um einen wechselseitiger Ausschluß mittels `cli()` dennoch zu realisieren, ergeben sich folgende Möglichkeiten:

1. Seitenfehler in kritischen Abschnitten sind zu verhindern.
2. Die Implementation der Operation `cli()` ist so zu verändern, daß kritische Abschnitte mit einem Semaphor („Interrupt-Lock") geschützt werden. [21]

Variante 1 ist leider nicht unproblematisch realisierbar; nähere Informationen dazu folgen im Abschnitt zur Speicherverwaltung (3.4).

Die zweite Variante hätte den Vorteil, daß sie unter Umständen sogar schneller als der Maschinenbefehl `cli` ist [21]. Ihre Realisierung ist jedoch etwas komplizierter. Problematisch ist sie in Fällen, in denen Gerätetreiber `cli()` nicht für einen wechselseitigen Ausschluß rufen, sondern um während der Programmierung des Interrupt-Controllers (PIC) keinen Interrupt zuzulassen.

Wegen der Realisierungsprobleme der ersten Variante entschieden wir uns gegen sie. Um Probleme mit der PIC-Programmierung zu umgehen, ruft unsere `cli()`-Realisierung zunächst außer dem Setzen des Interrupt-Locks auch noch den Maschinenbefehl „`cli`" auf. Später sollte der relevante Linux-Code so modifiziert werden, daß er an entsprechender Stelle selbst den „`cli`"-Maschinenbefehl ausführt.

3.3 Nutzer- und Kern-Aktivitäten; Scheduling

3.3.1 Annahmen der architekturunabhängigen Linux-Module

- Zwischen Nutzer- und Kern-Aktivitäten eines Linux-Prozesses wird synchron durch Betreten und Verlassen des Linux-Kerns hin- und hergeschaltet.
- Neben den freiwilligen Kerneintritten (Systemruf) gibt es erzwungene Kerneintritte (Ausnahmen, Hardware-Interrupts).
- Es ist immer nur ein (und genau ein) Prozeß aktiv, und demzufolge kann nur ein Prozeß Kern-Aktivitäten ausführen.
- Prozesse, die den Kern bereits betreten haben, können ihn noch einmal (rekursiv) betreten (z.B. durch Interrupt, Seitenfehler oder kern-internen Systemruf).
- Linux-Kern-Aktivitäten arbeiten stets im Kontext genau eines bestimmten Linux-Prozesses. Zu einem solchen Kern-Kontext gehört ein Registersatz, der Kern-Stack, die globale Variable `current` und der mit Copy-In/Out-Operationen manipulierte Nutzeradreßraum.
- Ein im Kern befindlicher Prozeß kann durch Aufruf der Funktion `schedule()` die Umschaltung (Aktivierung) zu einem anderen Prozeß gestatten. Dabei wird der aktuelle Zustand der Kern-Aktivität gesichert und nach der Rück-Umschaltung wiederhergestellt.

3.3.2 Nutzeraktivitäten und Schutzmechanismus

Linux-Prozesse stellen in einem Linux-System geschützte Entitäten dar. Sie sollen nur durch Systemrufe Zugriff auf Daten anderer Prozesse oder des Linux-Kerns erhalten können. Daher müssen sie in einer L4-Task gekapselt werden, da dies die einzige Möglichkeit in L4 ist, den Adreßraum einer Aktivität vor anderen Aktivitäten zu schützen.

Ein Linux-Prozeß besteht unter Linux/L4 also aus einer Nutzer-Aktivität in einer separaten Task und einer zugehörigen Kern-Aktivität (deren Modellierung im nächsten Abschnitt besprochen wird).

3.3.3 Kern-Aktivitäten

Wie oben erwähnt können sich Kern-Aktivitäten durch Aufruf der Funktion `schedule()` suspendieren, was zur Sicherung des Kern-Kontexts und zur Umschaltung zur Kern-Aktivität einer anderen Linux-Task führt.

Die Kern-Kontexte können auf verschiedene Weise implementiert werden:

1. als L4-Tasks mit je einem Thread
2. als L4-Threads in einer gemeinsamen L4-Task
3. als User-Level-Threads, die auf einem oder mehreren L4-Threads einer gemeinsamen L4-Task gemultiplext werden.

In der ersten Variante können Suspendierung und Aufwachen durch den L4-IPC-Systemdienst implementiert werden. Der Nutzeradreßraum kann permanent in den Adreßraum der Kern-Aktivität eingeblendet werden, und er wird bei einem Kontextwechsel automatisch mitumgeschaltet. Das führt zu einer besonders einfachen Copy-In/Out-Implementation: im wesentlichen eine Speicher-Kopieroperation.

Die zweite Variante hat den Vorteil eines schnelleren Kontextwechsels, da keine Adreßraumumschaltung erforderlich ist. Sie ist jedoch durch die L4-Implementation begrenzt, die nur 128 Threads pro Task erlaubt.

Dieses Problem hat die Variante 3 nicht. Außerdem ist ihr Kontextwechsel noch schneller, da dazu oft kein Eintritt in den L4-Kern nötig ist. Die Implementation des User-Level-Thread-Pakets könnte zunächst einfach aussehen, da Linux derzeit nur eine Kern-Aktivität zu einer Zeit unterstützt; somit würde nur ein L4-Thread benötigt, und die Synchronisation zwischen den User-Level-Threads gestaltete sich besonders einfach.

Die dritte Variante ist der zweiten also vorzuziehen. Die Entscheidung zwischen den Varianten 1 und 3 ist jedoch auch vom Design des Copy-In/Out-Mechanismus abhängig; wir verschieben sie daher bis Abschnitt 3.4.4.

3.3.4 Scheduling

Um Nutzer-Aktivitäten Prozessorzeit zuzuteilen, besitzt Linux einen Scheduler, der eine Linux-spezifische Scheduling-Strategie umsetzt.

Natürlich enthält auch der Mikrokern L4 einen Scheduler. Die Frage ist, ob und wie man die Linux-Strategie mit dem L4-Scheduler durchsetzen kann.

1. Durchsetzen der Linux-Strategie unter Verwendung von „Preemption Handlern", einem Mechanismus, den der L4-Kern zur Implementation von Scheduling-Strategien auf Nutzerebene anbietet. [15]

2. Übergang zu einem serverbasierten Modell: CPU-Scheduling erfolgt nicht mehr durch Linux, sondern nur durch den L4-Scheduler. Der Linux-Server bleibt inaktiv bis ein „Auftrag" (z.B. Systemruf) von einer Nutzertask eintrifft.

Variante 2 führt zu geringeren Kosten bei einer Präemption, da der Linux-Scheduler nicht aktiviert werden muß. Allerdings ist es mit dieser Variante nicht möglich, die Linux-Scheduling-Strategie direkt durchzusetzen, da L4 nur statische Prioritäten unterstützt; ein Nachbilden der Linux-Strategie durch beständiges Ändern der L4-Prioritäten ist nur mit großem Aufwand möglich.

Variante 1 erlaubt ein einfaches Durchsetzen der Linux-Scheduling-Policy, was unbedingt wünschenswert ist.

Leider waren zum Entwurfszeitpunkt Preemption Handler noch nicht einsatzbereit, so daß wir zunächst die serverbasierte Variante wählten.

3.4 Speicherverwaltung und Copy-In/Out

3.4.1 Annahmen der architekturunabhängigen Linux-Module

- Die Linux-Gerätetreiber gehen davon aus, daß von ihnen dereferenzierte virtuelle Speicheradressen physischen Speicher mit der selben Adresse ansprechen, d.h., daß der physische Speicher 1:1 in den virtuellen Kernadreßraum eingeblendet ist.

- Linux verwaltet den physischen Speicher selbst. Der architekturspezifische Teil muß ein vorgegebenes Seitentabellen-Interface bereitstellen, das Linux benutzt, um Seiten in Adreßräume einzublenden. [19]

- Linux benutzt die Prozeduren `memcpy_{from,to}_fs()`, um Daten in/aus den/m Adreßraum des Nutzerprozesses zu kopieren (Copy-In/Out). Diese Operationen müssen das Kopieren kleiner Datenmengen effizient unterstützen, da sie häufig für diesen Zweck benutzt werden.

3.4.2 Physischer Speicher

Damit die Gerätetreiber den physischen Speicher mit physischer = virtueller Adresse adressieren können, muß für alle Kern-Aktivitäten der RAM idempotent in den virtuellen Speicher eingeblendet werden. Die Nutzung der Segmentierungshardware der Intel-CPU-Architektur zu diesem Zweck ist nicht möglich, da L4 dies nicht unterstützt.

3.4.3 Virtueller Speicher

Speicherseiten, die von Linux' Speicherverwaltung mit der zu implementierenden Seitentabellen-Schnittstelle [19] in die Seitentabelle eingetragen werden, müssen in den jeweiligen virtuellen Adreßraum eingeblendet werden. Da eine L4-Task sich nicht selbst Seiten einblenden kann, ist dazu eine Pager-Task erforderlich. Diese erwartet Seitenfehler (oder entsprechende RPCs),

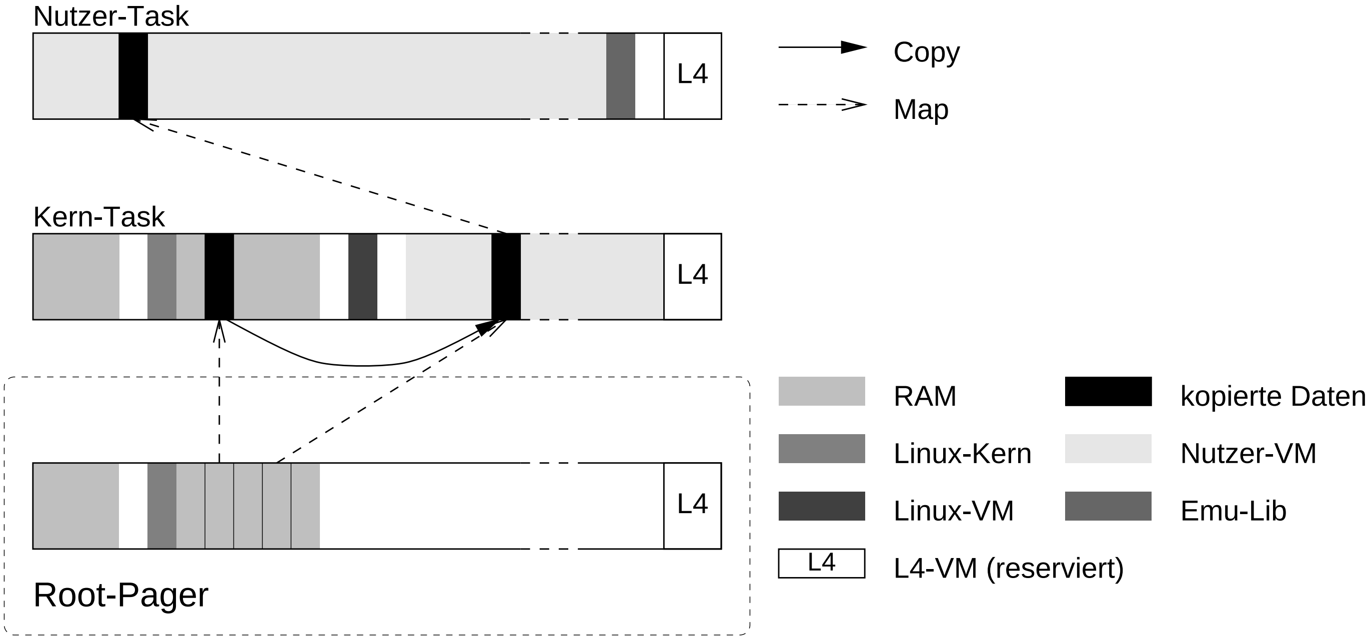

Abbildung 3.1: Copy-In/Out mit Einblenden des Nutzeradreßraums im Kern

schlägt die entsprechende Seite in der aktuellen Seitentabelle nach und sendet sie als Flexpage zurück. [15]

Pager für Nutzer-Tasks kann der Thread der entsprechenden Kern-Aktivität sein. Da aber auch der Linux-Kern intern virtuellen Speicher benutzt, ist eine weitere externe Pager-Task erforderlich (siehe Abbildung 3.3).

3.4.4 Copy-In/Out

Copy-In/Out ist der Linux-Mechanismus zum Kopieren von Daten in den Nutzeradreßraum hinein bzw. aus diesem heraus. Die Implementation dieses Mechanismus sollte besonders effizient ausfallen, da er häufig benutzt wird.

Folgende Möglichkeiten zur Realisierung dieses Mechanismus ergeben sich:

1. Einblenden des Nutzeradreßraums im Kern.

   Linux/i386 erlaubt einen Nutzer-Adreßraum mit gültigen Adressen zwischen `0` und `0xbfffffff`. Im Kern ist ab Adresse `0` der gesamte RAM und dahinter der kern-interne virtuelle Speicher eingeblendet; dahinter könnte der Nutzeradreßraums eingeblendet werden, etwa zwischen `0x20000000` und `0xdfffffff`.

   Lösung mit einer gemeinsamen L4-Task für alle Linux-Kern-Aktivitäten. Der eingeblendete Nutzeradreßraum muß bei jeder Taskumschaltung gewechselt werden (oder zumindest dann, wenn ein anderer Linux-Prozeß eine Copy-In/Out-Operation durchführt).

   Lösung mit einer L4-Task pro Linux-Kern-Aktivität. In diesem Fall übernimmt der L4-Kern die Adreßraumumschaltung.

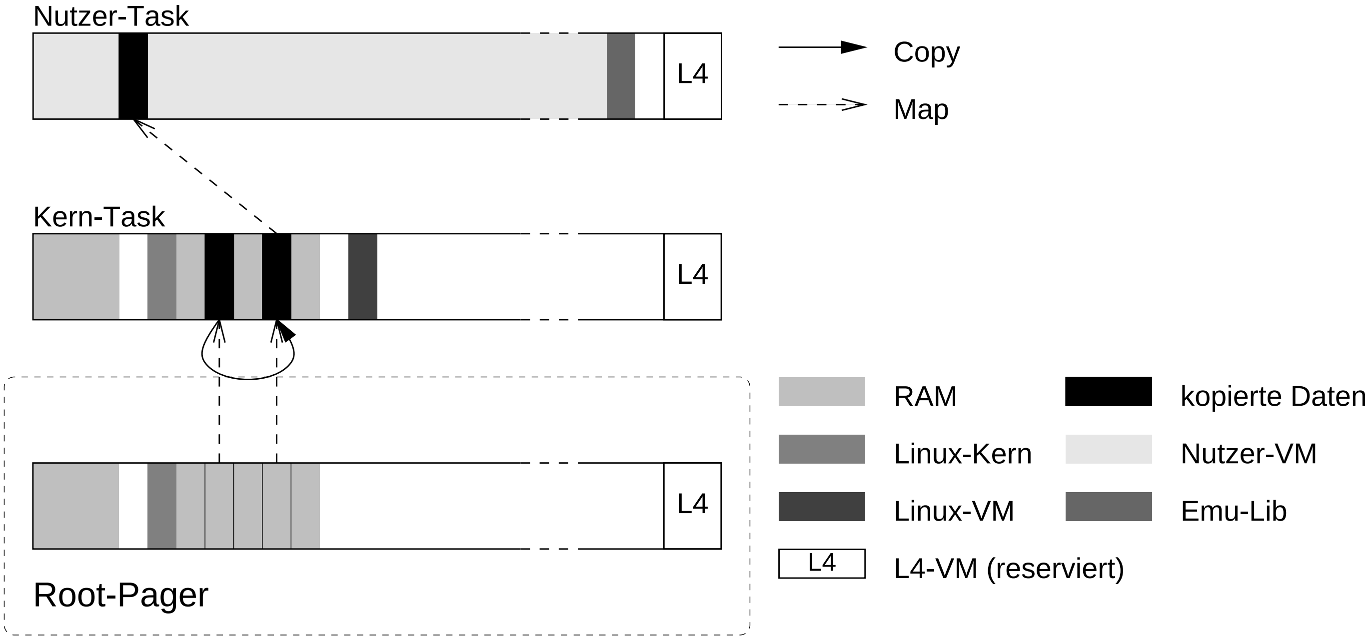

Abbildung 3.2: Copy-In/Out direkt in die Kachel

2. Die Copy-In/Out-Operationen interpretieren die Seitentabellen selbst und kopieren direkt in/aus den Kacheln.
3. Einblenden einzelner Speicherregionen der Prozesse, und Verwaltung dieser Regionen (analog OSFs Mklinux [1]).

Die erste Variante (siehe Abb. 3.1) ist einfach zu implementieren, denn die Linux/i386-Implementierung kann weitestgehend übernommen werden: Copy-In/Out sind im Grunde Speicherkopieroperationen und lösen für nicht (oder mit unpassendem Zugriffsattribut) eingeblendete Seiten Seitenfehler aus, die von Linux' normalen Behandlungsprozeduren gehandhabt werden.

Auf einmal eingeblendete Seiten kann bei weiteren Copy-In/Out-Operationen dann nahezu ohne weitere Kosten zugegriffen werden, bei der Lösung mit mehreren Tasks sogar noch nach mehreren Prozeßumschaltungen.

Variante 2 (Abb. 3.2) erfordert einen mittelgroßen Aufwand beim Parsen des Seitentabellen-Baums; die Ergebnisse dieser Operation können jedoch für spätere Zugriffe zwischengespeichert werden. Beim Kopieren kommen kleinere Kosten hinzu, da von Hand auf Seitengrenzen und nicht vorhandene Seiten geprüft werden muß.

Diese Lösung läßt sich so implementieren, daß keine Seitenfehler im Kern-Modus mehr auftreten, so daß das im Abschnitt 3.2.5 erläuterte Problem im Zusammenhang mit dem wechselseitigen Ausschluß durch `cli()` nicht auftritt.

Variante 3 erfordert einen recht hohen Aufwand zur Verwaltung der Speicherregionen und zum Aufsuchen einer Region; die Ergebnisse dieser Operation können jedoch ebenfalls für spätere Zugriffe zwischengespeichert werden. Dafür sind bei der eigentlichen Kopieroperation keine weiteren Tests nötig.

Wegen ihrer Vorteile wählen wir die erste Variante. Im folgenden diskutieren wir die relativen Vor- und Nachteile der Lösungen mit einer L4-Task für alle Linux-Prozesse bzw. für jeweils einen Prozess.

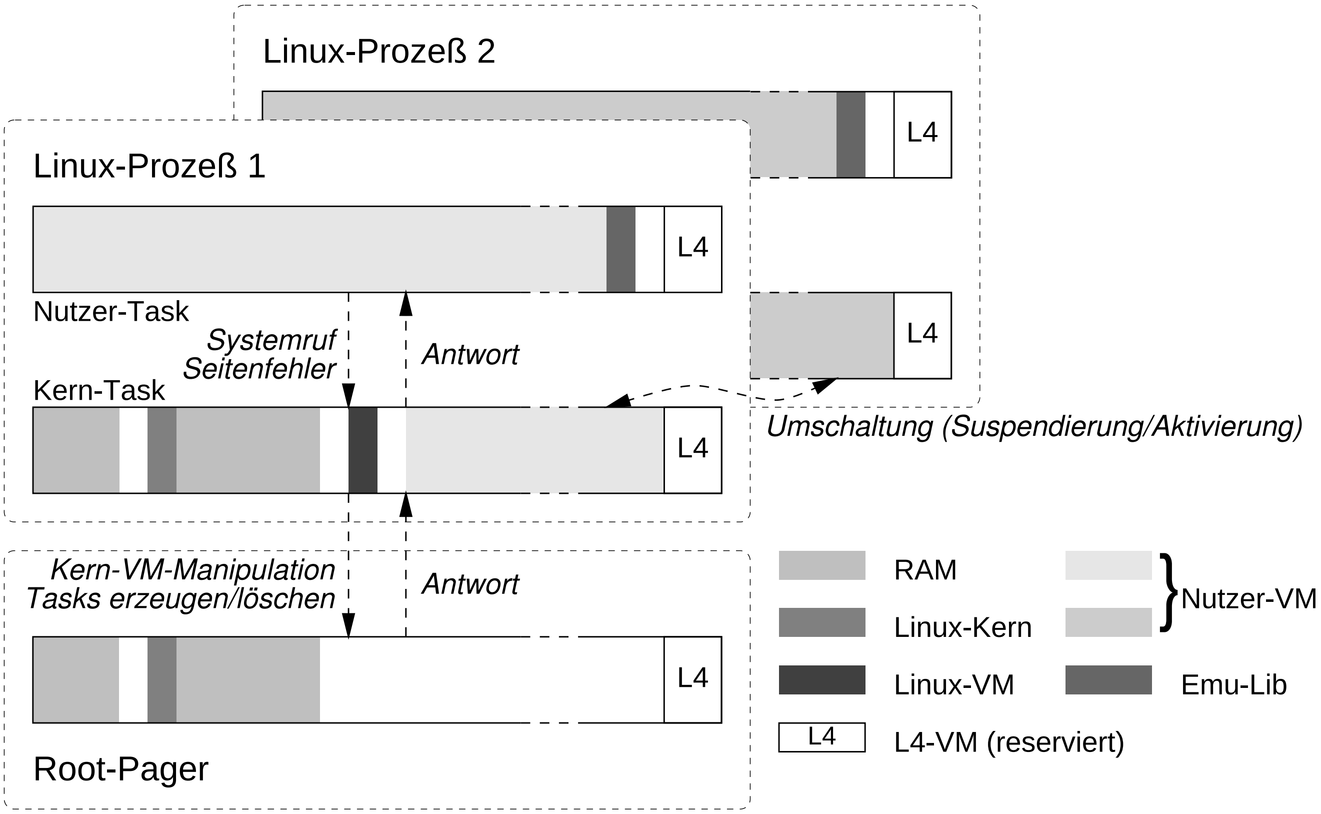

Abbildung 3.3: Aufbau der Adreßräume im Server-Tasksystem

Die Lösung mit nur einer Kern-Task erfordert ein User-Level-Thread-Paket (vgl. Abschnitt zur Kontextsicherung (3.3.3)). Die Umschaltung zwischen den Kern-Aktivitäten erfolgt schneller, aber dafür muß der eingeblendete Nutzeradreßraum ab und zu ausgetauscht werden, so daß schon einmal eingeblendete Seiten unter Umständen nochmals eingeblendet werden müssen.

Die Lösung mit N Kern-Tasks (für jeden Linux-Prozeß eine) hat diesen Nachteil nicht: Einmal etablierte Mappings bleiben über Kontextwechsel hinweg erhalten. Dafür dauern die Kontextwechsel etwas länger, und der Ressourcenverbrauch pro Linux-Prozeß ist höher: Benötigt werden jeweils eine Kern-Task mit zwei Threads (ein Pager-Thread und ein Service-Thread), und die Mapping-Datenbank des L4-Kerns wird stärker in Anspruch genommen. Ferner fällt ein einmaliger zusätzlicher Zeitaufwand beim Erzeugen einer neuen Task zum Einblenden des Codes und der Daten des Linux-Kerns an.

Der Zeitaufwand für oftmaliges Löschen des eingeblendeten Adreßraums und erneutes Einblenden von Seiten ($> 20\ \mu s$) ist wesentlich höher als der für eine Taskumschaltung (3,5 μs [22][2]). Daher entschieden wir und für die Lösung mit N Tasks (siehe Bild 3.3).

[2] Zeiten für i486, 75 MHz

3.5 Kerneintritt und Signalzustellung

3.5.1 Annahmen der architekturunabhängigen Linux-Module

- Ein Linux-Kerneintritt ist ein synchrones Umschalten eines Prozesses in den Kern-Modus durch Ausnahmen (Systemaufruf, Seitenfehler, ...) oder Interrupts. Die Umschaltung kann aus dem Nutzer-Modus oder (verschachtelt) aus dem Kern-Modus erfolgen. Die unterbrochene Aktivität wird erst beim (ebenfalls synchronen) Verlassen des Kern-Modus fortgesetzt.

- Unterbrochene Nutzeraktivitäten sind vollständig unter der Kontrolle des Linux-Kerns; das heißt, der Kern kann den Zustand der Nutzeraktivität (Register) und deren Adreßraum (z.B. Stack) beliebig manipulieren.

- Unmittelbar vor Verlassen des Linux-Kerns zum Nutzermodus müssen eventuell anliegende Signale zugestellt werden. Dies erfordert Zugriff auf Stack und Prozessorregister der Nutzer-Aktivität.

  (Kern-Aktivitäten können keine Signale erhalten.)

3.5.2 Interrupts

In Linux/i386 sind Interrupts lediglich ein spezieller Weg, den Kern synchron zu betreten. Interrupt-Behandlungsroutinen haben die Möglichkeit, auf den unterbrochenen Nutzer- oder Kern-Kontext zuzugreifen und ihn via `schedule()` umzuschalten (diese Methode zur Kontextumschaltung wird in Linux/i386 von Timer-Interrupt zur Präemption benutzt).

Diese Möglichkeit besteht unter L4 nicht mehr, da Interrupts in separaten Threads ablaufen (vgl. Abschnitt 3.2.2) und keinen einfachen Zugriff auf den unterbrochenen Kontext haben; die erforderliche Synchronität wird nicht durch Kerneintritt/ -rückkehr hergestellt, sondern durch Höherpriorisierung der Interrupt-Threads.

Daher müssen Kontextzugriffe aus dem Linux-Interrupt-Code entfernt werden. (Glücklicherweise werden solche Zugriffe nur für statistische und Debugging-Zwecke verwendet und nicht zur Erbringung kritischer Kern-Funktionalität.)

3.5.3 Kerneintritt aus Nutzer-Aktivitäten

Nutzer-Aktivitäten sollen den Linux-Kern bei Systemrufen und auftretenden Fehler- und Ausnahme-Bedingungen betreten. Systemrufe sind in Linux/i386 durch „`int 80`“ implementiert, sind also nur eine spezielle Ausnahme.

Um einen synchronen Kerneintritt gemäß der Linux-Semantik zu modellieren, müssen die Nutzer-Aktivitäten auf die Rückkehr der Kern-Aktivität warten. Dies läßt sich mit dem IPC-Mechanismus des L4-Kerns realisieren.

L4 behandelt Seitenfehler anders als alle anderen Ausnahmen: Seitenfehler werden via IPC einem Pager-Thread zur Bearbeitung überstellt, während andere Ausnahmen thread-lokal behandelt werden müssen [15]. Im folgenden gehen wir auf diese beiden Klassen des Kerneintritts näher ein.

Systemrufe und andere Ausnahmen (außer Seitenfehler). Ausnahmen werden dem unterbrochenen Thread vom L4-Kern als i386-Ausnahmen zugestellt, d.h. wie der Intel-Prozessor legt der L4-Kern spezielle Informationen über die Ausnahme auf den Stack und ruft eine Behandlungsprozedur entsprechend einer Tabelle (Interrupt Descriptor Table, IDT) auf, die vorher entsprechend initialisiert werden muß. [15]

Die Behandlungsroutine muß den Prozessorzustand sichern, so daß der Kern ihn lesen und manipulieren kann, dann den Kern via IPC aktivieren und nach dessen Rückkehr den Prozessorzustand (der eventuell vom Kern modifiziert wurde, beispielsweise um ein Signal zuzustellen) wiederherstellen und zum unterbrochenen Nutzerkontext zurückkehren. Da die Behandlungsroutine im Nutzer-Thread ablaufen muß, wird für die Nutzer-Task eine Emulationsbibliothek benötigt, die der Kern beim Start des Prozesses in den Nutzerprozeß einblendet und die die notwendigen Initialisierungen vornimmt, um auftretende Ausnahmen abzufangen.

Seitenfehler. Wie bereits erwähnt werden diese vom L4-Kern in Nachrichten umgewandelt und an einen Pager-Thread zugestellt. [15]

Diese Nachricht kann auf zwei Arten behandelt werden:

1. Umwandlung in eine Ausnahme, so daß Seitenfehler wie in Linux/i386 behandelt werden können: Die Nachricht wird an einen speziellen Thread in der Nutzer-Task zugestellt, der dem unterbrochenen Thread mit dem L4-Systemruf `l4_thread_ex_regs` eine Ausnahme zustellt.
2. Die Nachricht wird direkt der zugeordneten Linux-Kern-Task zustellt (d.h. ein Thread in der Kern-Task ist Pager für die Nutzer-Task).

Variante 1 hat den Vorteil, daß der unterbrochene Thread die Ausnahme wie im oben beschriebenen allgemeinen Fall behandeln kann. Insbesondere hat die Ausnahmebehandlung Zugriff auf den Registersatz des Linux-Prozesses, so daß Signale zugestellt werden können.

Die zweite Variante erlaubt keinen Zugriff auf den Prozessorzustand des unterbrochenen Threads, da der Thread bis zum Empfang der Antwortnachricht im vom L4-Kern aufgesetzten IPC schläft und danach unmittelbar an dem unterbrochenen Maschinenbefehl wiederaufsetzt. Dafür ist diese Variante wesentlich effizienter, da sie praktisch kosten-frei ist.

Da Linux Demand Paging aggressiv einsetzt, favorisieren wir die zweite Variante aufgrund der besseren Effizienz. Für die Signalzustellung nach Seitenfehlern benötigen wir daher eine spezielle Lösung, auf die wir im Abschnitt 3.5.5 eingehen werden.

3.5.4 Kerneintritt aus Kern-Aktivitäten

Systemrufe und Seitenfehler sind auch im Linux-Kernmodus erlaubt:

- Systemrufe werden von kern-internen Linux-Prozessen wie `init`, `bdflush` und `nfsiod` eingesetzt.
- Seitenfehler können bei Copy-In/Out auftreten.

Systemrufe. Anders als bei Systemrufen aus dem Nutzerkontext haben wir hier Einfluß auf die Implementation der Systemrufe. Linux/i386 benutzt auch für kern-interne Systemrufe die „`int 80`"-Schnittstelle. Für die L4-Implementation gibt es folgende Möglichkeiten:

1. „`int 80`“ + Ausnahmebehandlung, wie in Linux/i386.
2. Implementation durch direkten Prozeduraufruf.

Variante 1 bietet unter L4 keine Vorteile, da der Ausnahmebehandlungs-Code nicht für Nutzer- und Kern-Ebene gemeinsam benutzt werden kann, denn der auf Nutzer-Ebene nötige IPC zum Kern fällt weg.

Die zweite Variante ist schneller, da der Ausnahme-Mechanismus des L4-Kerns nicht benutzt wird. Problematisch ist diese Methode lediglich bei Systemrufen, die ein spezielles Stack-Layout erwarten, wie beispielsweise `clone()`. Dieses Problem läßt sich aber durch spezielle Wrapper in der Include-Datei `<asm/unistd.h>` lösen.

Aufgrund der höheren Effizienz fiel die Wahl auf Variante 2.

Seitenfehler Wie bereits diskutiert, wird für die Behandlung von Seitenfehlern ein Pager-Thread benötigt. Dieser Pager-Thread führt im wesentlichen architekturunabhängigen Linux-Code aus, der unter anderem Zugriff auf den virtuellen Kern-Speicher zugreifen kann.

Der Kern-Pager muß also in der Kern-Task selbst ablaufen. Um Seiten in den eigenen Adreßraum einblenden zu können, wird jedoch die Hilfe einer weiteren externen Task benötigt: Zu diesem Zweck führen wir einen Root-Pager ein, dessen Aufgabe es ist, auf Anforderung Seiten in den virtellen Adreßraum der Kern-Task einzublenden (siehe Abb. 3.3).

3.5.5 Signalzustellung

In diesem Abschnitt besprechen wir die Zustellung von Signalen zu einem Nutzer-Prozeß. Dies geschieht im allgemeinen kurz vor der Rückkehr aus dem Kernmodus zum Nutzer-Modus.

Die Problematik der Signalzustellung in einem System, das eine Task-Autonomie garantiert, wurde bereits in Jean Wolters Diplomarbeit [6] ausführlich behandelt. Daher soll der folgende kurze Abriß für unsere Zwecke genügen.

Voraussetzung für die Zustellung von Signalen ist im allgemeinen, daß CPU-Zustand (Register) und Stack des Prozesses manipulierbar sein müssen, damit gegebenenfalls Signal-Behandlungsroutinen aktiviert werden können, die der Prozeß installiert hat. Unter L4 ist diese Bedingung jedoch unter Umständen nicht erfüllt:

- Bei Seitenfehlern (vgl. Abschnitt 3.5.3): Der L4-Kern sendet für die Nutzer-Task in diesem Fall eine Nachricht an den Pager, ohne der Task Gelegenheit zu geben, vorher ihren Zustand zu sichern und ihn hinterher aus vorgegebenen Werten wiederherzustellen.

- Beim Server-Modell (vgl. Abschnitt 3.3.4) besteht keine Garantie, daß die Nutzer-Task jemals den Kern betritt, so daß Signale zugestellt werden können.

Daher ist es notwendig, eine Möglichkeit zu schaffen, die Nutzer-Task durch eine extern generierte Ausnahme zu zwingen, den Kern zu betreten. Dies ist jedoch aufgrund der vom L4-Kern garantierten Task-Autonomie nicht ohne weiteres möglich [15]; vielmehr wird dazu die Kooperation eines Threads innerhalb der Nutzertask benötigt.

Dies kann nicht der eigentliche Nutzer-Thread sein; es ist unmöglich, diesen zu zwingen, von Zeit zu Zeit beim Linux-Kern nachzufragen, ob inzwischen neue Signale anliegen. Vielmehr benötigen wir einen dedizierten Signal-Thread im Nutzeradreßraum, dessen Aufgabe es ist, auf Meldungen vom Linux-Server über neue Signale zu warten und dann dem Nutzer-Thread eine Ausnahme zuzustellen (L4-Systemruf `l4_thread_ex_regs()`), so daß dieser den Kern auf die gewünschte Weise betritt.

Ein Problem dieses Ansatzes ist, daß es unmöglich ist, die Nutzer-Task zur Kooperation zu zwingen: ein boshaftes Programm könnte den Signal-Thread beenden oder sonstwie manipulieren. Es ist jedoch die Semantik der Unix-Signale SIGKILL und und SIGSTOP, auch bei unkooperativen Prozessen zu funktionieren. Daher müssen für diese Signale spezielle Behandlungen eingeführt werden:

- SIGKILL wird der dem signalisierten Prozeß zugeordneten Kern-Task direkt zugestellt, so daß diese den Prozeß sofort beenden kann, ohne auf die Kooperation der Nutzertask angewiesen zu sein.
- SIGSTOP läßt sich mit dem Server-Modell nicht ohne weiteres erzwingen. Es wäre zwar eine Lösung mit einer Änderung der Priorität oder der Größe der Zeitscheibe des signalisierten Prozesses möglich, aber dieses Problem löst sich glücklicherweise auch durch die Benutzung von Preemption Handlern, sobald diese einsatzbereit sind (siehe Abschnitt 3.3.4), so daß es sich hier nicht lohnt, größeren Aufwand in die Lösung dieses Problems zu investieren.

3.6 Zusammenfassung der Entwurfsentscheidungen

In diesem Kapitel stellten wir ausgewählte Lösungen zu Problemen vor, die bei der Portierung des Kerns des Betriebssystems Linux auf den L4-Mikrokern auftraten. Wir beschäftigten uns mit den Subsystemen zur Interruptbehandlung, Zeitverwaltung, Ressourcenverwaltung (Speicher und Prozessor) und Signalzustellung.

Das Ergebnis ist ein aus mehreren Task bestehendes Server-System, das in separaten L4-Tasks laufenden Linux-Prozessen die Services eines Linux-Kerns bereitstellt. Dieses Server-System besteht aus folgenden Teilen (siehe Abbildung 3.3):

- Ein Root-Pager, in dessen Adreßraum der verfügbare physische Speicher eingeblendet ist (vom L4-Systempager Sigma-0 bereitgestellt) und der den Kern-Tasks anhand der von Linux aufgebauten Seitentabellen virtuellen Speicher bereitstellt.
- N Kern-Tasks — für jeden Linux-Prozeß eine:
  - In Kern-Task 0 (entspricht Linux-Prozeß 0, `idle`) laufen die Interrupt-Threads: für jeden Hardware-Interrupt ein L4-Thread, plus der Bottom-Half-Thread.
  - In allen anderen Kern-Tasks läuft ein Service-Thread, der Meldungen über Kerneintritte von der zugeordneten Nutzer-Task empfängt (Seitenfehler und Ausnahmen, wobei Systemrufe zu letzteren zählen), und ein Kern-Pager-Thread, der Seitenfehler des Service-Threads behandelt (solche treten zum Beispiel während der Initialisierung und bei Copy-In/Out-Operationen auf).

    Diese Kern-Tasks blenden in einen Teil ihres Adreßraums den Nutzeradreßraums ein, der für Copy-In/Out-Operationen verwendet wird.

- N Nutzer-Tasks — für jeden Linux-Prozeß eine. In diesen Tasks laufen ebenfalls zwei Threads: Ein Nutzer-Thread, der den Code der jeweiligen Linux-Anwendung ausführt, und ein Signal-Thread, der auf Meldungen eines Service-Threads wartet, um den Nutzer-Thread gegebenfalls zum Kern-Eintritt zu zwingen.

Kapitel 4

Implementation

Der im Kapitel 3 entworfene Linux-Server für den Mikrokern L4 wurde in einer ersten Version vollständig implementiert und im Internet unter der URL `ftp://ftp.inf.tu-dresden.de/pub/os/L4/devel/` bereitgestellt. Der Server implementiert vollständig die Funktionalität eines Linux-Systems[1]; wir hatten jedoch bisher wenig Gelegenheit, die Leistung des Servers genauer zu messen oder zu optimieren.

Die Anpassung an den L4-Mikrokern beläuft sich derzeit auf etwa 12000 Zeilen C-Quelltext (inklusive der von Linux/i386 übernommenen Teile, aber ohne bestimmte Header-Dateien, die von Linux/i386 mitbenutzt werden; zum Vergleich: die i386-Anpassung ist (ohne FPU-Emulation) etwa 18000 Zeilen groß; der Linux-Kern (ohne Architekturanpassung und Treiber) ist 167000 Zeilen groß, die mitgelieferten Treiber insgesamt 343600 Zeilen). Sie wurde in etwa viermonatiger Arbeit von vier Projekt-Beteiligten erstellt.

In diesem Kapitel gehen wir auf einzelne Aspekte der Implementation ein.

4.1 Maschinenabhängige Linux-Subsysteme

Maschinenabhängige Teile des Linux-Kerns, die vom L4-Kern nicht eingeschränkte Hardware-Schnittstellen benutzen oder beschreiben, konnten nahezu unverändert von Linux/i386 übernommen werden. Dies betrifft insbesondere folgenden Subsysteme:

- die ABI-Definition,
- die Gerätetreiberunterstützung (Zugriff auf die Floppy-, DMA-, Interrupt-Controller- und Uhrenhardware),
- die Schnittstelle zum BIOS32,
- alle i386-Gerätetreiber.

4.2 Interrupts

Die Interruptverwaltung wurde wie im Abschnitt 3.2 besprochen implementiert: Beim Systemstart wird der Bottom-Half-Thread und Threads für jeden Interrupt gestartet, wobei letztere

[1] Für zwei Besonderheiten der Linux/i386-Implementation wurde keine Unterstützung implementiert: Virtual-8086-Modus und ladbare Kernmodule für Linux/i386

sofort versuchen, sich mit dem entsprechenden L4-IPC-Mechanismus an die Interrupt-Quellen anzuschließen [15].

Eine Besonderheit stellt lediglich der Uhreninterrupt dar, der alle 10 ms auftreten muß: Die Interruptquellen, die normalerweise für diesen Zweck benutzt werden (IRQs 0 und 8), sind vom L4-Kern für interne Zwecke reserviert. Daher wird das Warten auf den Uhreninterrupt mit dem IPC-Mechanismus des L4-Kerns durch Timeouts emuliert.

Durch dieses Verfahren „geht" der Uhreninterrupt natürlich ungenau. Dies wird durch eine spezielle Synchronisation der Linux-Zeit (`jiffies`) mit der vom L4-Kern bereitgestellten Maschinenzeit ausgeglichen.

4.3 Kern- und Nutzeraktivitäten

Von besonderem Interesse sind hier der Kerneintritt und die Realisierung der synchronen Umschaltung zwischen den Kernaktivitäten, denn die L4-Anpassung muß gewährleisten, daß immer nur eine Instanz zu einer Zeit Linux-Code ausführt, da Linux nicht eintrittsinvariant ist.

4.3.1 Kerneintritt

Führt ein Linux-Programm einen Systemruf aus oder tritt eine Ausnahme oder ein Seitenfehler auf, so führt dies zu einer Nachricht an die zugeordnete Linux-Kern-Task, wie wir in Abschnitt 3.5.3 gesehen haben. Nachdem diese Nachricht vom Kern empfangen wurde, muß der nicht eintrittsinvariante Linux-Code ausgeführt werden. Um zu verhindern, daß zwei Instanzen gleichzeitig Linux-Code ausführen, müssen sich alle Kern-Aktivitäten synchronisieren.

Vorbild für den Synchronisationsmechanismus war die Linux-Portierung auf OSF Mach, die ebenfalls ein Server-Modell realisiert [1]. Dort wird der Kerneintritt mit einem *Mutex* (C-Typ `mutex_t`) synchronisiert, einem binären Semaphor mit Warteschlange [23].

Wir implementierten diese Funktionalität für L4 nach, wobei wir für die Suspendieren/Fortsetzen-Semantik den L4-IPC-Mechanismus benutzten: Um sich schlafenzulegen, hängt sich der aktuelle Thread in eine Warteschlange ein und wartet dann auf eine Aufweck-Nachricht von einem anderen Thread, der das Mutex gerade freigibt.

Nun ist es möglich, den gesamten Linux-Kern mit dem Mutex `KernelLock` zu schützen, welches vor der Abarbeitung von Linux-Code gesetzt und danach wieder freigegeben werden muß.

4.3.2 Umschaltung zwischen Kernaktivitäten

In Linux wird die Umschaltung zu anderen Kern-Aktivitäten durch Aufruf der Funktion `schedule()` ausgeführt. Diese Funktion entscheidet, welche Kern-Aktivität als nächste laufen soll und aktiviert diese.

Im Gegensatz zu Linux/i386 wird unter L4 diese Funktion nicht regelmäßig vom Uhreninterrupt aufgerufen, um Präemption von Nutzeraktivitäten zu erreichen, da Linux/L4 ein serverbasiertes System ist und die Umschaltung zwischen Nutzer-Tasks bereits vom L4-Scheduler durchgeführt wird (vgl. Abschnitt 3.3.4).

`schedule()` wird also nur explizit von Kernaktivitäten benutzt, um sich zu suspendieren. Es muß lediglich gewährleistet werden, daß keine Aktivität `schedule()` verläßt, bis ihr Linux-Prozeß-Zustand (wieder) auf `TASK_RUNNING` gesetzt ist.

Die Linux-Portierung auf OSF Mach benutzt zum Warten auf diese Bedingung eine *Condition Variable* (C-Typ `condition_t`) [23]. Auch in diesem Fall implementierten wir diese Funktionalität für L4 nach, und zwar die Libmach-Funktionen `condition_wait()` und `condition_signal()`: Erstere gibt ein angegebenes Mutex frei und wartet auf das Eintreten einer Bedingung, die durch eine Condition Variable repräsentiert wird, und letztere signalisiert ebendiese Bedingung.

Damit läßt sich die grundlegende Funktionalität von `schedule()` wie folgt erbringen:

```
if (current->state == TASK_RUNNING)
  {
    thread_yield();              /* kurz zu anderen bereiten
                                    Aktivitaeten umschalten */
  }
else
  do
    {
      ...
      kernel_will_exit();       /* Kontext freigeben (current usw.) */
      condition_wait(&(task->tss.condition), &KernelLock, task);
      kernel_has_entered(task); /* Kontext wiederherstellen */
      ...
    }
  while (current->state != TASK_RUNNING);
```

Außerdem muß in der Linux-Funktion `wake_up_process()`, die den Zustand einer Task auf `TASK_RUNNING` setzt, ein Aufruf zu `condition_signal()` eingefügt werden, damit suspendierte Tasks weiterlaufen können, wenn sie aus Sicht von Linux wieder rechenbereit werden.

4.4 Speicherverwaltung

Wie bereits erwähnt, manipuliert Linux den virtuellen Adreßraum seiner Aktivitäten mit einer vom architekturabhängigen Teil zu implementierenden Seitentabellen-Schnittstelle. Diese Schnittstelle geht von einer dreistufigen Seitentabellen-Hierarchie aus, die von der Architekturanpassung in das von der jeweils benutzten Maschine verwendete Format umgewandelt werden muß (z.B. Intel: zweistufige Hierarchie; PowerPC: invertierte Seitentabellen).

Für die Portierung auf L4 verwendeten wir die Architekturanpassung aus Linux/i386, so daß durch die Adreßraum-Manipulationen des Linux-Kerns letztendlich eine für den Intel-Prozessor geeignete zweistufige Seitentabellen-Hierarchie entsteht.

Anpassungen waren lediglich in den Funktionen `set_pte()` und `pte_clear` notwendig, die von Linux zum Manipulieren eines Seitentabelleneintrags verwendet werden: Um eine Seite auszublenden oder nicht-schreibbar einzublenden, wird der L4-Systemruf `l4_fpage_unmap()` aufgerufen.

Tritt nun ein Seitenfehler auf (d.h. der entsprechende Pager-Thread erhält eine Seitenfehler-Nachricht), so bildet unser Pager die Funktion der Intel-CPU nach: Er schlägt die entsprechende Seite in der Intel-Seitentabelle nach, prüft die Zugriffsattribute der Tabelleneinträge, ruft gegebenfalls die entsprechenden Linux-Seitenfehler-Prozeduren auf (`do_no_page()` für nicht vorhandene Seiten, `do_wp_page()` für schreibgeschützte Seiten) und sendet letztendlich die Seite als L4-Flexpage an den Thread zurück, in dem der Seitenfehler aufgetreten war.

4.5 Signalzustellung

Wie in Abschnitt 3.5.5 besprochen, wurde die Signalzustellung mit Hilfe eines im Nutzeradreßraum laufenden speziellen Threads realisiert, der Meldungen über neue Signale vom Linux-Server empfängt und dann dem Nutzer-Thread Ausnahmen zustellt.

Die Nachrichten an den Signal-Thread können von jeder Kern-Task generiert werden. Die Versendung erfolgt in der Linux-Prozedur `generate()`, die für das Setzen von Signal-Flags in einer Prozeß-Datenstruktur verantwortlich ist.

Interessant an der Implementation dieses Mechanismus waren im wesentlichen zwei Aspekte:

- Wegoptimierung von Signal-Nachrichten, wenn die Nutzer-Aktivität der Kern bereits betreten hat; in diesem Falle werden ihre Signale automatisch zugestellt, sobald sie den Kern verläßt.
- Verhinderung von Wettkampfbedingungen: Der Signal-Thread darf im Nutzer-Thread keine Ausnahme generieren, wenn dieser den Kern bereits betreten hat oder soeben dabei ist.

Das erste Problem wurde durch Einführung eines Locks `under_kernel_control` in der Prozeß-Datenstruktur gelöst, das unmittelbar beim Eintritt in den Linux-Kern gesetzt wird. Ist es für die zu signalisierende Task bereits gesetzt, kann die Signal-Nachricht wegoptimiert werden.

Das zweite Problem konnte durch ein Lock `emu_local_lock` gelöst werden, das die von der Ausnahmebehandlung im Nutzer-Adreßraum benutzte Datenstruktur schützt und das bei jeder Ausnahme sofort von der Ausnahmebehandlung gesetzt wird. Der Signal-Thread darf keine Signal-bedingte Ausnahme zustellen, solange er dieses Lock nicht erwerben kann.

Im Normalfall kann der Signal-Thread in diesem Fall die Signal-Nachricht sogar ignorieren, da der Nutzer-Thread ohnehin gerade den Kern betritt. Problematisch ist jedoch der Zustand, in dem der Prozeß zwar den Kern betreten, die Signalzustellung aber bereits durchgeführt hat: In diesem Fall darf der Signal-Thread die Nachricht *nicht* ignorieren, sondern muß die Ausnahme nach der Rückkehr des Nutzer-Threads aus dem Kern zustellen. Um letzteren Fall zu erkennen, wurde ein für den Signal-Thread sichtbares Flag `sigs_handled` eingeführt, das unmittelbar vor der Signalzustellung im Linux-Kern gesetzt wird.

4.6 Zusammenfassung

Wir implementierten ein auf dem Mikrokern L4 laufendes Linux-System, das zu Linux/i386 ABI-kompatibel ist. Das System kann in den Multi-User-Modus booten, und die wichtigste Applikationssoftware für Linux läuft (X Window System, Netzwerksoftware, Compiler usw.).

Bisher nicht realisiert wurden die Systemrufe zur Unterstützung des Virtual-8086-Modus der Intel-CPU und eine Schnittstelle für Kernmodule, die für Linux/i386 generiert wurden.

Das Ziel, möglichst keine Änderungen am architekturunabhängigen Teil des Linux-Kerns vorzunehmen, konnte weitestgehend eingehalten werden; es waren lediglich kleine Änderungen am Scheduler und am Signalzustellungsmechanismus und unbedeutende Änderungen an einigen Device-Treibern notwendig.

Kapitel 5

Leistungsbewertung

Um die Leistungsfähigkeit der ersten Version unserer Linux-Implementation auf dem Mikrokern L4 bewerten zu können, führten wir ein frei verfügbares Benchmark-Programm aus. Für eine genauere Analyse der Leistung der Linux-Portierung und eine Optimierung anhand der hier beschriebenen Resultate fanden wir jedoch bisher wenig Zeit.

5.1 Testumgebung

Wir verwendeten für unsere Tests einen Rechner mit einem Pentium-Prozessor (100 MHz), 32 MB Hauptspeicher, NE2000-Netzwerkkarte, BusLogic-SCSI-Controller und einer SCSI-Festplatte. Das Heimatverzeichnis war jeweils über NFS eingebunden.

Als Benchmarkprogramme benutzen wir das frei verfügbare Benchmark-Paket Lmbench [24].

Wir führten die Benchmarks auf der selben Maschine jeweils mehrmals unter Linux/i386 und Linux/L4 aus und verglichen dann die Resultate.

5.2 Meßergebnisse

Lmbench ist ein Benchmark-Paket, das von Larry McVoy entwickelt wurde. Es enthält im wesentlichen „Microbenchmarks“, die einzelne Subsysteme des Betriebssystems testen und messen. [24]

Wir führten jeweils drei Messungen mit Linux/i386 bzw. Linux/L4 aus. Auf den folgenden beiden Seiten sind die mit bei diesen Durchläufen erzielten Meßzeiten zu finden: die nächste Seite enthält die absoluten Meßzeiten, die darauffolgende Seite setzt die gemessenen Zeiten miteinander ins Verhältnis.

Die ersten drei Zeilen jeder Tabelle zeigen unter Linux/i386 erzielten Zeiten, die letzten drei die unter Linux/L4 erzielten.

L M B E N C H 1 . 0 S U M M A R Y

Processor, Processes - times in microseconds

Host	OS	Mhz	Null Syscall	Null Process	Simple Process	/bin/sh Process	Mmap lat	2-proc ctxsw	8-proc ctxsw
carola	Linux 2.0.0	100	3	2K	12K	73K	239	11	18
carola.1	Linux 2.0.0	100	3	2K	12K	73K	231	11	18
carola.2	Linux 2.0.0	100	3	2K	12K	72K	229	10	19
carola.3	Linux 1.3.94	105	56	16K	39K	143K	518	111	137
carola.4	Linux 1.3.94	105	48	15K	37K	131K	516	92	107
carola.5	Linux 1.3.94	100	51	14K	37K	129K	510	96	106

Local Communication latencies in microseconds

Host	OS	Pipe	UDP	RPC/ UDP	TCP	RPC/ TCP
carola	Linux 2.0.0	49	213	499	300	669
carola.1	Linux 2.0.0	48	216	489	322	716
carola.2	Linux 2.0.0	45	213	512	315	690
carola.3	Linux 1.3.94	442	891	1480	1104	1926
carola.4	Linux 1.3.94	375	792	1341	979	1735
carola.5	Linux 1.3.94	388	775	1309	962	1734

Local Communication bandwidths in megabytes/second

Host	OS	Pipe	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
carola	Linux 2.0.0	22	9	22	51	23	22	62	35
carola.1	Linux 2.0.0	22	10	22	50	23	22	62	35
carola.2	Linux 2.0.0	22	10	22	51	23	22	62	35
carola.3	Linux 1.3.94	9	8	21	21	20	21	60	35
carola.4	Linux 1.3.94	10	8	17	23	22	21	61	35
carola.5	Linux 1.3.94	10	7	17	25	22	21	61	35

Memory latencies in nanoseconds
(WARNING - may not be correct, check graphs)

Host	OS	Mhz	L1 $	L2 $	Main mem	Guesses
carola	Linux 2.0.0	99	10	190	316	
carola.1	Linux 2.0.0	99	10	196	316	
carola.2	Linux 2.0.0	99	10	223	316	
carola.3	Linux 1.3.94	104	-	-	-	Bad mhz?
carola.4	Linux 1.3.94	104	-	-	-	Bad mhz?
carola.5	Linux 1.3.94	99	0	188	312	

L M B E N C H 1 . 0 S U M M A R Y

Comparison to best of the breed

(Best numbers are starred, i.e., *123)

Processor, Processes - factor slower than the best

Host	OS	Mhz	Null Syscall	Null Process	Simple Process	/bin/sh Process	Mmap lat	2-proc ctxsw	8-proc ctxsw
carola	Linux 2.0.0	100	*3	1.0	1.0	1.0	1.0	1.1	*18
carola.1	Linux 2.0.0	100	*3	*1.8K	*12.0K	1.0	1.0	1.1	*18
carola.2	Linux 2.0.0	100	*3	1.0	1.0	*70.6K	*229	*10	1.1
carola.3	Linux 1.3.94	105	19	8.7	3.2	2.0	2.3	11	7.6
carola.4	Linux 1.3.94	105	16	7.7	3.0	1.8	2.3	9.2	5.9
carola.5	Linux 1.3.94	100	17	7.4	3.0	1.8	2.2	9.6	5.9

Local Communication latencies - factor slower than the best

Host	OS	Pipe	UDP	RPC/ UDP	TCP	RPC/ TCP
carola	Linux 2.0.0	1.1	*213	1.0	*300	*669
carola.1	Linux 2.0.0	1.1	1.0	*489	1.1	1.1
carola.2	Linux 2.0.0	*45	*213	1.0	1.1	1.0
carola.3	Linux 1.3.94	9.8	4.2	3.0	3.7	2.9
carola.4	Linux 1.3.94	8.3	3.7	2.7	3.3	2.6
carola.5	Linux 1.3.94	8.6	3.6	2.7	3.2	2.6

Local Communication bandwidths - percentage of the best

Host	OS	Pipe	TCP	File reread	Mmap reread	Bcopy (libc)	Bcopy (hand)	Mem read	Mem write
carola	Linux 2.0.0	*22	96%	98%	99%	99%	99%	99%	99%
carola.1	Linux 2.0.0	99%	*9	99%	98%	99%	*21	*61	*35
carola.2	Linux 2.0.0	98%	99%	*22	*51	*22	99%	99%	99%
carola.3	Linux 1.3.94	38%	77%	92%	41%	86%	97%	97%	98%
carola.4	Linux 1.3.94	44%	79%	77%	44%	98%	97%	98%	97%
carola.5	Linux 1.3.94	44%	75%	77%	48%	98%	98%	97%	98%

Memory latencies in nanoseconds - factor slower than the best
(WARNING - may not be correct, check graphs)

Host	OS	Mhz	L1 $	L2 $	Main mem	Guesses
carola	Linux 2.0.0	99	-5.0	1.0	1.0	
carola.1	Linux 2.0.0	99	-5.0	1.0	1.0	
carola.2	Linux 2.0.0	99	-5.0	1.2	1.0	
carola.3	Linux 1.3.94	104	-	-	-	Bad mhz?
carola.4	Linux 1.3.94	104	-	-	-	Bad mhz?
carola.5	Linux 1.3.94	99	???	*188	1.0	

5.3 Interpretation der Meßergebnisse

Auf den ersten Blick fällt auf, daß unter der Linux-Emulation auf L4 gegenüber Linux/i386 die Durchsätze auf 44–99% sinken und die Latenzzeiten auf das 1,8–19-fache steigen. In Absolutwerten heißt das etwa, daß Systemrufe ca. 50 μs länger dauern oder daß Pipes nur 10 MB/s Durchsatz erreichen statt 22 MB/s (um die beiden extremsten Beispiele herauszugreifen).

Diese Ergebnisse mögen etwas entmutigend erscheinen; wir glauben jedoch, daß diese Werte für eine erste, unoptimierte Version durchaus innerhalb der zu erwartenden Grenzen liegen. Es soll auch nicht übersehen werden, daß wichtige Meßergebnisse wie TCP- und Dateisystem-Durchsatz im brauchbaren Bereich liegen, was den subjektiven Eindruck bestätigt, daß Linux/L4 im praktischen Betrieb nicht spürbar langsamer ist als Linux/i386.

Im folgenden gehen wir auf einzelne Meßergebnisse ein und stellen Vermutungen an, wie sie zustande kamen. Leider hatten wir aus Zeitgründen noch keine Möglichkeit, diese Hypothesen zu überprüfen; dies muß jedoch im Anschluß an diese Arbeit geschehen.

Systemruf-Latenz: i386: 3 μs — unter L4: >17 μs.

Neben den stets auftretenden Extra-Kosten für das Abfangen der Ausnahme auf Nutzerseite sind überflüssige L4-Systemaufrufe eine mögliche Senke für die zusätzlich benötigte Zeit. Jean Wolter fand heraus, daß bei jedem Systemruf bestenfalls mindestens 8 Systemrufe `l4_myself()` stattfinden (Erfragen der Thread-ID des ausführenden Threads) [25]. Setzt man die Dauer für einen L4-Systemruf bei etwa 1,5 μs an [22], so tragen allein diese 8 Systemrufe für 12 μs der Systemaufrufzeit bei.

Die `l4_myself()`-Systemrufe lassen sich leicht einsparen — man kann die Thread-IDs der Threads auf ihrem privaten Stack ablegen und eine Inline-Funktion schreiben, die mit Hilfe des aktuellen Stack-Pointers die Thread-ID ermittelt.

`mmap()`-Latenz: i386: ca. 230 μs — unter L4: ca. 515 μs

Ein Problem beim Einblenden von Speicher in eine Applikation ist, daß momentan für jede `mmap()`-Operation die Kern-Task zweimal betreten werden muß: Einmal für den eigentlichen Systemruf, der die Seitentabellen entsprechend manipuliert, und ein zweites Mal, wenn der erste Seitenfehler in der manipulierten Region eintritt. Die Ursache ist, daß die Manipulation der Seitentabellen unter L4 nicht automatisch die Seiten in die Nutzer-Task einblendet.

Dies würde auch teilweise das schlechtere Abschneiden beim „Mmap reread"-Benchmark (Einblenden bereits im Haupspeicher zwischengespeicherter Seiten) und bei der Prozeßerzeugung erklären.

Eine mögliche Abhilfe wäre hier, bei Systemrufen, die Seiten in den Speicher einblenden, beim Senden der Systemruf-Antwort stets die eingeblendeten Seiten als Flexpages mitzusenden.

1st- und 2nd-Level-Cache-Zugriffszeiten Offenbar war es diesem Meßprogramm unter L4 nicht möglich, sinnvolle Meßwerte zu ermitteln. Eine mögliche Ursache ist, daß die zur Zeitabfrage benutzte L4-Systemuhr keine μs-Auflösung besitzt, sondern nur alle 1 ms aufgefrischt wird. Abhilfe wäre hier die Benutzung des Time-Stamp-Counters der Pentium-CPU, die die aufgetretenen Taktimpulse seit dem Systemstart mitzählt, zur Zeitabfrage.

Kapitel 6

Zusammenfassung und Ausblick

Ziel dieser Arbeit war die Portierung des Linux-Betriebssystems auf den Mikrokern L4. Verschiedene Lösungsmöglichkeiten wurden diskutiert, und eine Lösung schließlich implementiert.

Entstanden ist eine unter L4 laufende server-basierte Linux-Portierung, wobei nur minimale Änderungen am architekturunabhängigen Teil und an den Gerätetreibern des Linux-Kerns notwendig waren. Die Applikationsschnittstelle dieses Servers ist binärkompatibel zur Linux-Implementation für Intel-CPUs.

Die Leistungsdaten der ersten Implementationsversion lassen noch zu wünschen übrig, liegen jedoch im erwarteten Bereich. Es wurde bereits Raum für mögliche Optimierungen identifiziert.

Im Gegensatz zu den Optimierungsbestrebungen anderer Mikrokern-Projekte änderten wir den Mikrokern nicht ab (kein Hinzubinden von Server-Code in den Adreßraum des Mikrokerns) und beschränkten auch nicht die Funktionalität der Unix-Emulation.

In nächster Zukunft muß die Implementation in folgenden Bereichen verbessert werden:

Leistung Ziel ist es, so dicht wie möglich an die Leistungsdaten von Linux/i386 zu kommen.

Robustheit Momentan sind führen viele Fehlerbedingungen (zum Beispiel der Empfang einer unerwarteten Nachricht) noch zu fatalen Fehlern, die einen Prozeß oder sogar das gesamte System abstürzen lassen. Hier muß die Implementation gründlich aufgeräumt werden.

Alles in allem sind wir mit dem Ergebnis der Arbeit zufrieden. Neben der Implementation des Linux-Servers für L4 sind die Einblicke in die Funktionsweise moderner Betriebssysteme (L4, Linux) und die gesammelten Erfahrungen im Umgang mit großen Softwaresystemen weitere Erfolge.

Anhang A

Glossar

ABI: „Application Binary Interface". Beschreibt eine Binärschnittstelle, über die *Applikationen* Systemdienste in Anspruch nehmen können. Vgl. *API*.

Adreßraum: Menge gültiger Speicheradressen, auf die ein Programm zugreifen kann.

Aktivität: Im Kontext dieser Arbeit ist eine Aktivität ein *Thread*, der an einen bestimmten *Kontext* geknüpft ist.

API: „Application Programming Interface". Beschreibt eine Programmierschnittstelle (z.B. eine Menge von C-Funktionen), über die *Applikationen* Systemdienste in Anspruch nehmen können. Vgl. *ABI*.

Applikation: Anwendungsprogramm.

Ausnahme: (engl. „Exception") Eine Bedingung, die den Prozessor veranlaßt, den aktuellen *Kontext* zu verlassen und eine Ausnahmebehandlung auszuführen, wie zum Beispiel ein *Seitenfehler* oder eine Division durch Null.

In Linux werden Systemrufe durch den Befehl „`int 80`" ausgeführt, der ebenfalls eine Ausnahme auslöst.

BIOS32: Ein spezielles *ABI* zur Kommunikation mit der Firmware eines Gerätes.

Bottom Half: Siehe *Interrupt*.

Condition Variable: (auch „Bedingungsvariable") Eine Datenstruktur, die eine bestimmte Bedingung symbolisiert. Gültige Operationen über dieser Datenstruktur sind das Warten auf eine Bedingung und das Signalisieren, daß eine Bedingung eingetreten ist.

Copy-In/Out: Kopieren von Daten aus dem Adreßraum einer Nutzer-*Aktivität* in den Kern-Adreßraum und umgekehrt.

CPU: („Central Processing Unit") Mikroprozessor.

Demand Paging: Auf einem externen Speichermedium (z.B. Festplatte) abgelegte Code- und Datenseiten eines Programms werden beim Programmstart nicht sofort in den Hauptspeicher geladen, sondern erst bei auftretenden *Seitenfehlern* in den *Adreßraum* eingeblendet.

Flexpage: L4-Einheit zur Speichermanipulation. Eine Menge von *Seiten* läßt sich unter L4 zu Flexpages zusammenfassen; so ist es möglich, gleich mehrere *Seiten* in den *Adreßraum* eines *Threads* einzublenden, in dem ein *Seitenfehler* auftrat. [15]

Interrupt: (auch „Unterbrechung") Eine von einem externen Gerät angezeigte Bedingung, die zur Unterbrechung des aktuellen *Kontexts* und zur Abarbeitung einer Interrupt-Behandlungsroutine führt.

Interrupt-Behandlungsroutinen sind in Linux zweigeteilt: Die *Top Half* wird sofort nach Eintreffen des Interrupts ausgeführt; die *Bottom Half* wird erst ausgeführt, wenn keine weiteren Top Halves mehr auszuführen sind. (Vgl. Abschnitt 3.2.)

IPC: („Inter Process Communication"; auch „Interprozeßkommunikation") Ein L4-Mechanismus, der die Kommunikation zweier Threads durch Nachrichtenaustausch gestattet.

Kontext: Aktueller Zustand einer *Aktivität*. Dazu gehört der aktuelle Adreßraum, der Prozessorzustand (Register) und in Linux einige Kernvariablen wie `current`.

Lock: Allgemein eine Datenstruktur, mit der man einen wechselseitigen Ausschluß erreichen kann.

Mutex: (von „mutual exclusion") Eine spezielle Datenstruktur, die einen wechselseitigen Ausschluß mit einer binären *Semaphor* realisiert. Kann man die *Semaphor* nicht sofort erhalten, muß man sich suspendieren und darauf warten, vom Besitzer der Semaphor aufgeweckt zu werden.

Pager: Allgemein ein Mechanismus, der *Adreßräume* bereitstellt und verwaltet. In L4 wird diese Aufgabe von einem *Thread* wahrgenommen, der vom Mikrokern via *IPC* über *Seitenfehler* in zugeordneten *Threads* informiert wird und diese durch Antworten mit *Flexpages* auflösen kann.

PIC: („Programmable Interrupt Controller") Programmierbarer Interrupt-Baustein, der in jedem PC zu finden ist.

Präemption: Unterbrechung eines *Threads*, weil seine *Zeitscheibe* abgelaufen ist.

Preemption Handler: Ein L4-Mechanismus, der es gestattet, auf Nutzer-Ebene zu *schedulen*: Der Preemption Handler eines *Threads* wird via *IPC* über abgelaufene *Zeitscheiben* informiert und der *Thread* wird so lange blockiert, bis der Preemption Handler mit einer Nachricht antwortet.

Scheduling: Zuteilung der Ressource „Rechnerzeit" an Threads. S.a. *Preemption Handler*.

Seite: Kleinste Einheit zur Manipulation eines *Adreßraums*.

Semaphor: Eine Zähler, mit dem sich ein wechselseitiger Ausschluß realisieren läßt, der einen kritischen Abschnitt schützt. Solange man den Zähler noch inkrementieren kann, so daß er eine bestimmte Maximalzahl nicht überschreitet, darf man den kritischen Abschnitt betreten, ansonsten muß man warten.

Ist der Maximalwert = 1, so spricht man von einem binären Semaphor, und derjenige, der den Zähler zuletzt inkrementiert hat, „besitzt" das Semaphor.

Seitentabelle: Datenstruktur, die *Adreßräume* aufeinander abbildet. Wird von der Intel-*CPU* verwendet, um den virtuellen Adreßraum auf den physischen Speicher (RAM) abzubilden.

Seitenfehler: Unerlaubter Zugriff auf eine Speicheradresse: Entweder, an der Adresse ist keine Seite eingeblendet, oder auf eine nur lesbar eingeblendete Seite wurde schreibend zugegriffen. S.a. *Pager*.

SMP: „Symmetric Multi-Processing". Betriebssystem-Methode, um Maschinen mit mehr als einer *CPU* zu verwalten.

Systemruf: Mechanismus, um einen Dienst eines Betriebssystems aufzurufen. Sowohl L4 als auch Linux haben solche Mechanismen. S.a. *Ausnahme*.

Task: In L4 ein *Adreßraum*, in dem ein oder mehrere *Threads* laufen können.

Thread: In L4 ein Kontrollfluß in einer Task.

Top Half: Siehe *Interrupt*.

Zeitscheibe: Rechenzeit-Einheit, die einem *Thread* zugeteilt wurde.

Literaturverzeichnis

[1] *François Barbou des Places, Nick Stephen*: Linux on the OSF Mach3 microkernel; 1996 *FSF-sponsored Conference on Freely Distributable Software*

[2] *B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, S. Eggers*: Extensibility, Safety and Performance in the SPIN Operating System; *1995 SOSP*

[3] *D. Hildebrand*: An architectural overview of QNX; 1992 *1st Usenix Workshop on Microkernels and Other Kernel Architectures, Seattle, WA, pp. 113–126.*

[4] *Jochen Liedtke*: On μ-Kernel Construction; *1995 SOSP*

[5] *Michael Hohmuth, Sven Rudolph*: Steps Towards Porting a Unix Single Server to the L3 Microkernel; 1996 *Großer Beleg an der Fak. Informatik der TU Dresden*

[6] *Jean Wolter*: Emulation des UNIX-Prozeßkonzepts auf dem Mikrokern L3; 1995 *Master's Thesis, Dept. of Computer Science, TU Dresden; Technical report TUD/FI/95/12*

[7] *D. Golub, R. Dean, A. Forin, R. Rashid*: UNIX as an Application Program; 1990 *School of Computer Science, Carnegie Mellon University*

[8] *J. Helander*: Unix under Mach: The Lites Server; 1994 *Master Thesis, Dept. of Computer Science, Helsinki University of Technology*

[9] *G. Hammilton, P. Kougiouris*: The Spring Nucleus: A Microkernel for Objects; 1993 *Sun Microsystems Laboratories, Inc.*

[10] *K. Loepere (Editor)*: OSF Mach Kernel Principles, Revision 4; *OSF Mach series, Open Software Foundation and Carnegie Mellon University*

[11] *J. M. Stevenson, D. P. Julin*: Mach-US: UNIX On Generic OS Object Servers; 1995 *Usenix Technical Conference*

[12] *Michael I. Bushnell*: Towards a New Strategy of OS Design; *The January 1994 GNU's Bulletin*

[13] *M. Condict, D. Mitchell, F. Reynolds*: Optimizing Performance of Mach-based Systems By Server Co-Location: A Detailed Design; 1993 *OSF Research Institute*

[14] *J. Liedtke*: Improving IPC by kernel design; *1993 SOSP*

[15] *Jochen Liedtke*: L4 Reference Manual — 486, Pentium, Pentium Pro; 1996 *(available from ftp://borneo.gmd.de/pub/rs/L4/doc/)*

[16] *Jochen Liedtke*: Clans & Chiefs; 1992 *Architektur von Rechnersystemen, 12. GI/ITG Fachtagung, Kiel*

[17] *Greg Hankins (Editor)*: Linux Documentation Project; *(available from http://www.uni-paderborn.de/Linux/mdw/)*

[18] *Michael K. Johnson (Editor)*: Linux Kernel Hacker's Guide; *(available from http://www.redhat.com:8080/HyperNews/get/khg.html)*

[19] *Michael Hohmuth*: Linux Architecture-Specific Kernel Interfaces; 1996 *(available from http://www.inf.tu-dresden.de/~mh1/prj/linux-on-l4/)*

[20] *René Stange*: Systematische Übertragung von Gerätetreibern von einem monolithischen Betriebssystem auf eine mikrokernbasierte Architektur; 1996 *Master's Thesis, Dept. of Computer Science, TU Dresden*

[21] *Cristopher Small, Stephen Manley*: A Revisitation of Kernel Synchronization Schemes; 1996 *Harvard University*

[22] *Jochen Liedtke*: L4-Zeiten; *(private Email vom 29. August 1996)*

[23] *K. Loepere (Editor)*: OSF Mach Server Writer's Guide, Revision 4; *OSF Mach series, Open Software Foundation and Carnegie Mellon University*

[24] *Larry McVoy, Carl Staelin*: lmbench: Portable tools for performance analysis; *1995 USENIX Technical Conference*

[25] *Jean Wolter*: lmbench, null system call and why are the numbers so bad; *(Email-Nachricht an die L4-Linux-Mailingliste vom 8. August 1996)*