Diplomarbeit:
Linux-Emulation auf einem Mikrokern

Michael Hohmuth

29. August 1996
Inhaltsverzeichnis

1 Einleitung 5
  1.1 Über dieses Dokument ................................. 6
  1.2 Danksagung ............................................ 6

2 Stand der Technik 7
  2.1 Unix-Systeme auf Mikrokernen .......................... 7
  2.2 Der Mikrokern L4 ........................................ 8
  2.3 Das Betriebssystem Linux ................................. 9

3 Entwurf 11
  3.1 Entwurfsziele ........................................... 11
  3.2 Interruptverwaltung .................................... 11
  3.2.1 Annahmen der architekturunabhängigen Linux-Module 11
  3.2.2 Top Halves ........................................... 12
  3.2.3 Bottom Halves ....................................... 12
  3.2.4 Synchronisierung der Interrupt-Aktivitäten .......... 13
  3.2.5 Wechselseitiger Ausschluß durch c11() .............. 13
  3.3 Nutzer- und Kern-Aktivitäten; Scheduling .............. 14
    3.3.1 Annahmen der architekturunabhängigen Linux-Module 14
    3.3.2 Nutzeraktivitäten und Schutzmechanismus ........... 15
    3.3.3 Kern-Aktivitäten ................................ 15
    3.3.4 Scheduling ......................................... 15
  3.4 Speicherverwaltung und Copy-In/Out ........................ 16
    3.4.1 Annahmen der architekturunabhängigen Linux-Module 16
    3.4.2 Physischer Speicher ................................ 16
    3.4.3 Virtueller Speicher ................................ 16
    3.4.4 Copy-In/Out ....................................... 17
  3.5 Kerneintritt und Signalzustellung .......................... 20
    3.5.1 Annahmen der architekturunabhängigen Linux-Module 20
INHALTSVERZEICHNIS

3.5.2 Interrupts .............................................. 20
3.5.3 Kerneintritt aus Nutzer-Aktivitäten .................... 20
3.5.4 Kerneintritt aus Kern-Aktivitäten ........................ 21
3.5.5 Signalzustellung ......................................... 22
3.6 Zusammenfassung der Entwurfsentscheidungen ............ 23

4 Implementation ................................................. 25
  4.1 Maschinenabhängige Linux-Subsysteme ..................... 25
  4.2 Interrupts .................................................. 25
  4.3 Kern- und Nutzeraktivitäten ............................... 26
    4.3.1 Kerneintritt .......................................... 26
    4.3.2 Umschaltung zwischen Kernaktivitäten ................ 26
  4.4 Speicherverwaltung ..................................... 27
  4.5 Signalzustellung ......................................... 28
  4.6 Zusammenfassung ......................................... 28

5 Leistungsbewertung ........................................... 29
  5.1 Testumgebung ............................................. 29
  5.2 Meßergebnisse ............................................ 29
  5.3 Interpretation der Meßergebnisse .......................... 32

6 Zusammenfassung und Ausblick ............................... 33

A Glossar ......................................................... 35

Literatur .......................................................... 39
Kapitel 1

Einleitung


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:

  
  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.
  

- 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:

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.


1.2 Danksagung


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

\[
\text{MD5 (freunde)} = 36f085284392f640f67c190bc112ca8
\]
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].


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

2.2 Der Mikrokern L4


Implementiert sind Operationen für folgende Objekte:

- **Flexiblen großen 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 eingebunden (map) und weitergeleitet (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).


L4 ist ein sehr kleiner, außerordentlich schneller Mikrokern:

- Sein IPC-Mechanismus ist eine Größenordnung schneller als bei traditionellen Mikrokernen.

- L4's Paging-Mechanismus ist durch seine Leichtgewichtigkeit besonders effizient.


L4 enthält keinerlei Gerätetreiber, sondern unterstützt auf Nutzerebene implementierte Gerätetreiber durch folgende Mechanismen:
2.3. DAS BETRIEBSSYSTEM LINUX

- 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 Interruppts 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.

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

Durch die wohldefinierte maschinew- (oder architektur-) abhängige Schicht erreicht der Linux-Kern eine hohe Portabilität: Die restlichen Kernbestandteile sind dadurch maschinenumabhängig 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.

**Gerätedreiberunterstützung** 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 Gerätedreibern und dem architekturunabhängigen Linux-Code eine architekturübergreifende Schnittstelle zur Interrupt-Hardware zur Verfügung.

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


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.

---

1Aktivitäten 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


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-Implementierungen 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-Behandlungs Routinen 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.


- Kritische Abschnitte (sowohl in Interrupt-Behandlungs Routinen 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 Empfänger-Thread 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-Behandlungsroute 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-Behandlungs Routinen 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 Semaphore intr_count zur Verfügung, das bei jeder Interrupt-Behandlung durch Top Halves erhöht werden muß; Bottom Halves dürfen nur abgearbeitet werden, wenn das Semaphore 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.


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.
3.2. INTERRUPTVERWALTUNG


Ein potenzielles 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 Kernzugeführt (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.


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 wechselseitigen 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).


3.3 Nutzer- und Kern-Aktivitäten; Scheduling

3.3.1 Annahmen der architekturunabhängigen Linux-Module


- Neben den freiwiligen 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).


3.3. **Nutzer- und Kern-Aktivitäten; Scheduling**

### 3.3.2 Nutzeraktivitäten und Schutzmechanismus


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 eingebunden 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.


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 zuzuweisen, besitzt Linux einen Scheduler, der eine Linux-spezifische Scheduling-Strategie umsetzt.

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

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 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, size)`, 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


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),
3.4. SPEICHERVERWALTUNG UND COPY-IN/OUT

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


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 0xffffffff. Im Kern ist ab Adresse 0 der gesamte RAM und dahinter der kern-interne virtuelle Speicher eingebunden; dahinter könnte der Nutzeradressraum eingebunden werden, etwa zwischen 0x20000000 und 0xffffffff.

   Lösung mit einer gemeinsamen L4-Task für alle Linux-Kern-Aktivitäten.
   Der eingebundene 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.
2. Die Copy-In/Out-Operationen interpretieren die Seitentabellen selbst und kopieren direkt in/aus die Kachel.

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) eingeblendet- te Seiten Seitenfehler aus, die von Linux' normalen Behandlungsprozeduren gehandhabt werden. Auf einmal eingeblendetete 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 cl1() 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.
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 eingebundene Nutzeradreßraum ab und zu ausgetauscht werden, so daß schon einmal eingebundene Seiten unter Umständen nochmals eingebundet werden müssen.


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

\[\text{Zeiten für i486, 75 MHz}\]
3.5 Kerneintritt und Signalzustellung

3.5.1 Annahmen der architekturreunabhängigen Linux-Module


- 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.


(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-Behandlungs Routinen 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 Kontextzuge rufe 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.
3.5. KERNEINTRITT UND SIGNALZUSTELLUNG

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 Behandlungsroute muß den Prozessorzustand sichern, so daß der Kern ihm 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 Si-
gnal zuzustellen) wiederherstellen und zum unterbrochenen Nutzerkontext zurückkehren.
Da die Behandlungsroute im Nutzer-Thread ablaufen muß, wird für die Nutzer-Task 
eine Emulationsbibliothek benötigt, die der Kern beim Start des Prozesses in den Nut-
zerprozeß 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 wer-
den können: Die Nachricht wird an einen speziellen Thread in der Nutzer-Task zuge-
gestellt, 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 be-
schriebenen 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 auf-
gesetzten 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, bdfush 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 Prozeduraufk.


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.


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 virtuellen 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.


Voraussetzung für die Zustellung von Signalen ist im allgemeinen, daß CPU-Zustand (Register) und Stack des Prozesses manipulierbar sein müssen, damit gegebenenfalls Signal-Behandlungs Routinen 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 betrifft, so daß Signale zugestellt werden können.

Daher ist es notwendig, eine Möglichkeit zu schaffen, die Nutzer-Task durch eine extern gene-rierte 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
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-Systempagger 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, id1e) 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 Kernereignisse 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.

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 \texttt{14\_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 boshaf tes 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.
Kapitel 4

Implementation


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 Geräteunterstützung (Zugriff auf die Floppy-, DMA-, Interrupt-Controller- und Uhrenhardware),
- die Schnittstelle zum BIOS32,
- alle i386-Geräte-

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 eintrittsvariant ist.

4.3.1 Kerneintritt


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 schla- fenzulegen, 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.


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.
4.4. SPEICHERVERWALTUNG

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_pwait() 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:

```c
if (current->state == TASK_RUNNING)
{
    thread_yield(); /* kurz zu anderen bereiten
                      Aktivitäten umschalten */
}
else
{
    ...
    kernel_will_exit(); /* Kontext freigegeben (current usw.) */
    condition_pwait(&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


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_ppte()` 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_page()` für nicht vorhandene Seiten, `do_vp_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.


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

- Wegoptimierung von Signal-Nachrichten, wenn die Nutzer-Aktivität der Kern bereits betreht 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 betreht 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.


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 architekturenabhängigen 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


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.
### LMBENCH 1.0 S U M M A R Y

Processor, Processes - times in microseconds

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Mhz</th>
<th>Null Syscall</th>
<th>Null Process</th>
<th>Simple /bin/sh Process</th>
<th>Mmap 2-proc</th>
<th>8-proc ctxsw</th>
<th>8-proc ctxsw</th>
</tr>
</thead>
<tbody>
<tr>
<td>carola</td>
<td>Linux 2.0.0</td>
<td>100</td>
<td>3</td>
<td>2K</td>
<td>12K</td>
<td>73K</td>
<td>239</td>
<td>11</td>
</tr>
<tr>
<td>carola.1</td>
<td>Linux 2.0.0</td>
<td>100</td>
<td>3</td>
<td>2K</td>
<td>12K</td>
<td>73K</td>
<td>231</td>
<td>11</td>
</tr>
<tr>
<td>carola.2</td>
<td>Linux 2.0.0</td>
<td>100</td>
<td>3</td>
<td>2K</td>
<td>12K</td>
<td>72K</td>
<td>229</td>
<td>10</td>
</tr>
<tr>
<td>carola.3</td>
<td>Linux 1.3.94</td>
<td>105</td>
<td>56</td>
<td>16K</td>
<td>39K</td>
<td>143K</td>
<td>518</td>
<td>111</td>
</tr>
<tr>
<td>carola.4</td>
<td>Linux 1.3.94</td>
<td>105</td>
<td>48</td>
<td>15K</td>
<td>37K</td>
<td>131K</td>
<td>516</td>
<td>92</td>
</tr>
<tr>
<td>carola.5</td>
<td>Linux 1.3.94</td>
<td>100</td>
<td>51</td>
<td>14K</td>
<td>37K</td>
<td>129K</td>
<td>510</td>
<td>96</td>
</tr>
</tbody>
</table>

*Local* Communication latencies in microseconds

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

*Local* Communication bandwidths in megabytes/second

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Pipe TCP</th>
<th>File Mmap</th>
<th>Bcopy (lib)</th>
<th>Bcopy (hand)</th>
<th>Mem read</th>
<th>Mem write</th>
</tr>
</thead>
<tbody>
<tr>
<td>carola</td>
<td>Linux 2.0.0</td>
<td>22 9 22 51</td>
<td>23</td>
<td>22</td>
<td>62</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>carola.1</td>
<td>Linux 2.0.0</td>
<td>22 10 22 50</td>
<td>23</td>
<td>22</td>
<td>62</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>carola.2</td>
<td>Linux 2.0.0</td>
<td>22 10 22 51</td>
<td>23</td>
<td>22</td>
<td>62</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>carola.3</td>
<td>Linux 1.3.94</td>
<td>9 8 21 21</td>
<td>20</td>
<td>21</td>
<td>60</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>carola.4</td>
<td>Linux 1.3.94</td>
<td>10 8 17 23</td>
<td>22</td>
<td>21</td>
<td>61</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>carola.5</td>
<td>Linux 1.3.94</td>
<td>10 7 17 25</td>
<td>22</td>
<td>21</td>
<td>61</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

Memory latencies in nanoseconds

(WARNING - may not be correct, check graphs)

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Mhz</th>
<th>L1 $</th>
<th>L2 $</th>
<th>Main mem</th>
<th>Guesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>carola</td>
<td>Linux 2.0.0</td>
<td>99 10 190</td>
<td>316</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>carola.1</td>
<td>Linux 2.0.0</td>
<td>99 10 196</td>
<td>316</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>carola.2</td>
<td>Linux 2.0.0</td>
<td>99 10 223</td>
<td>316</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>carola.3</td>
<td>Linux 1.3.94</td>
<td>104</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bad mhz?</td>
</tr>
<tr>
<td>carola.4</td>
<td>Linux 1.3.94</td>
<td>104</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bad mhz?</td>
</tr>
<tr>
<td>carola.5</td>
<td>Linux 1.3.94</td>
<td>99 0 188</td>
<td>312</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### 5.2. MEßERGEBNISSE

#### LMBENCH 1.0 SUMMARY

Comparison to best of the breed

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

Processor, Processes - factor slower than the best

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Mhz</th>
<th>Null Syscall</th>
<th>Process</th>
<th>Null Process</th>
<th>/bin/sh Mmap 2-proc</th>
<th>8-proc ctxsw</th>
<th>ctxsw</th>
</tr>
</thead>
<tbody>
<tr>
<td>carola</td>
<td>Linux 2.0.0</td>
<td>100</td>
<td>*3</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>carola.1</td>
<td>Linux 2.0.0</td>
<td>100</td>
<td>*3</td>
<td>*1.8K</td>
<td>*12.0K</td>
<td>1.0</td>
<td>1.0</td>
<td>1.1</td>
</tr>
<tr>
<td>carola.2</td>
<td>Linux 2.0.0</td>
<td>100</td>
<td>*3</td>
<td>1.0</td>
<td>1.0</td>
<td>*70.6K</td>
<td>*229</td>
<td>*10</td>
</tr>
<tr>
<td>carola.3</td>
<td>Linux 1.394</td>
<td>105</td>
<td>19</td>
<td>8.7</td>
<td>3.2</td>
<td>2.0</td>
<td>2.3</td>
<td>11</td>
</tr>
<tr>
<td>carola.4</td>
<td>Linux 1.394</td>
<td>105</td>
<td>16</td>
<td>7.7</td>
<td>3.0</td>
<td>1.8</td>
<td>2.3</td>
<td>9.2</td>
</tr>
<tr>
<td>carola.5</td>
<td>Linux 1.394</td>
<td>100</td>
<td>17</td>
<td>7.4</td>
<td>3.0</td>
<td>1.8</td>
<td>2.2</td>
<td>9.6</td>
</tr>
</tbody>
</table>

*Local* Communication latencies - factor slower than the best

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Pipe</th>
<th>UDP</th>
<th>RPC/UDP</th>
<th>TCP</th>
<th>RPC/TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>carola</td>
<td>Linux 2.0.0</td>
<td>1.1</td>
<td>*213</td>
<td>1.0</td>
<td>*300</td>
<td>*669</td>
</tr>
<tr>
<td>carola.1</td>
<td>Linux 2.0.0</td>
<td>1.1</td>
<td>1.0</td>
<td>*489</td>
<td>1.1</td>
<td>1.1</td>
</tr>
<tr>
<td>carola.2</td>
<td>Linux 2.0.0</td>
<td>*45</td>
<td>*213</td>
<td>1.0</td>
<td>1.1</td>
<td>1.0</td>
</tr>
<tr>
<td>carola.3</td>
<td>Linux 1.394</td>
<td>9.8</td>
<td>4.2</td>
<td>3.0</td>
<td>3.7</td>
<td>2.9</td>
</tr>
<tr>
<td>carola.4</td>
<td>Linux 1.394</td>
<td>8.3</td>
<td>3.7</td>
<td>2.7</td>
<td>3.3</td>
<td>2.6</td>
</tr>
<tr>
<td>carola.5</td>
<td>Linux 1.394</td>
<td>8.6</td>
<td>3.6</td>
<td>2.7</td>
<td>3.2</td>
<td>2.6</td>
</tr>
</tbody>
</table>

*Local* Communication bandwidths - percentage of the best

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>TCP</th>
<th>File Mmap</th>
<th>Bcopy</th>
<th>Bcopy</th>
<th>Mem</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>carola</td>
<td>Linux 2.0.0</td>
<td>*22</td>
<td>96%</td>
<td>98%</td>
<td>99%</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>carola.1</td>
<td>Linux 2.0.0</td>
<td>99%</td>
<td>*9</td>
<td>99%</td>
<td>98%</td>
<td>99%</td>
<td>*21</td>
</tr>
<tr>
<td>carola.2</td>
<td>Linux 2.0.0</td>
<td>98%</td>
<td>*22</td>
<td>*51</td>
<td>*22</td>
<td>99%</td>
<td>99%</td>
</tr>
<tr>
<td>carola.3</td>
<td>Linux 1.394</td>
<td>38%</td>
<td>77%</td>
<td>92%</td>
<td>41%</td>
<td>86%</td>
<td>97%</td>
</tr>
<tr>
<td>carola.4</td>
<td>Linux 1.394</td>
<td>44%</td>
<td>79%</td>
<td>77%</td>
<td>44%</td>
<td>96%</td>
<td>97%</td>
</tr>
<tr>
<td>carola.5</td>
<td>Linux 1.394</td>
<td>44%</td>
<td>75%</td>
<td>77%</td>
<td>48%</td>
<td>98%</td>
<td>98%</td>
</tr>
</tbody>
</table>

Memory latencies in nanoseconds - factor slower than the best

(WARNING - may not be correct, check graphs)

<table>
<thead>
<tr>
<th>Host</th>
<th>OS</th>
<th>Mhz</th>
<th>L1 $</th>
<th>L2 $</th>
<th>Main mem</th>
<th>Guesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>carola</td>
<td>Linux 2.0.0</td>
<td>99</td>
<td>-5.0</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>carola.1</td>
<td>Linux 2.0.0</td>
<td>99</td>
<td>-5.0</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>carola.2</td>
<td>Linux 2.0.0</td>
<td>99</td>
<td>-5.0</td>
<td>1.2</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>carola.3</td>
<td>Linux 1.394</td>
<td>104</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bad mhz?</td>
</tr>
<tr>
<td>carola.4</td>
<td>Linux 1.394</td>
<td>104</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>Bad mhz?</td>
</tr>
<tr>
<td>carola.5</td>
<td>Linux 1.394</td>
<td>99</td>
<td>???</td>
<td>*188</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
5.3 Interpretation der Meßergebnisse

Auf den ersten Blick fällt auf, daß unter der Linux-Emulation auf L4 gegenüber Linux/i386 die Durchschnitte auf 44-99% sinken und die Latenzzahlen auf das 1,8-19-fache steigen. In absoluten Werten heißt es 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 14_myself() stattfinden (Erfahren der Thread-ID des ausführenden Threads) [25]. Setzt man die Dauer für einen L4-Systemruf bei etwa 1,5 μs an [22], so tragen alleine diese 8 Systemrufe für 12 μs der Systemaufrufzeit bei.

Die 14_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


Dies würde auch teilweise das schlechtere Abschneiden beim „Mmap rerede“-Benchmark (Einblenden bereits im Hauptspeicher zwischengespeicherter Seiten) und bei der Prozeßerverzweigung 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 Maßprogramm 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 Taktempulse 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 80h“ 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 Kernadresraum und umgekehrt.

CPU: („Central Processing Unit“) Mikroprozessor.

**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-Behandlungs Routinen 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 *Semaphore* realisiert. Kann man die *Semaphore* nicht sofort erhalten, muß man sich suspendieren und darauf warten, vom Besitzer der Semaphore 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 *schedule*en: 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.


**Seite**: Kleinte Einheit zur Manipulation eines *Adreßraums*.

**Semaphore**: 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 Semaphore, und derjenige, der den Zähler zuletzt inkrementiert hat, „besitzt“ das Semaphore.
Seitentabelle: Datenstruktur, die Adreßräume aufeinander abbildet. Wird von der Intel-CPU verwendet, um den virtuellen Adreßraum auf den physischen Speicher (RAM) abzubilden.


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


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


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


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


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