Die L4-Mikrokern-Familie

Michael Steil

18. April 2002

Inhaltsverzeichnis

Diese Arbeit beschäftigt sich mit dem L4-Mikrokern und seinen Implementierungen. Dabei wird auf Mikrokerne im allgemeinen, den L4-Mikrokern als Vertreter der Mikrokerne der zweiten Generation im speziellen, sowie seine Implementierungen eigegangen. Schließlich wird die Linux-Portierung auf den L4-Kern beschrieben.

1 Mikrokerne

Bevor die Konzepte des L4-Mikrokerns behandelt werden können, muß zuerst geklärt werden, was ein Mikrokern überhaupt ist, worin seine Vorteile gegenüber konventionellen Kernen liegen, welche Nachteile dieses Design hat sowie welche in der Praxis häufig eingesetzten Betriebssysteme bereits auf einem Mikrokern beruhen.

1.1 Motivation

Um in die Vorteile eines Betriebssystems mit einem Mikrokern einzuführen, sollen im folgenden typische Nachteile zweier (noch) häufig anzutreffender nicht-Mikrokern-Betriebssysteme, Windows 95 und Linux, kurz darzugestellt werden.

Mit Windows 95 assoziiert man häufig eine geringe Stabilität. Diese ist jedoch nicht unbedingt in fehlerhaftem Code im Betriebssystem selbst begründet, sondern evtl. auch in fehlerhaften Treibern: Nur ein einziger Fehler in einem einzigen Treiber kann zum Absturz des Systems führen. Aber auch Fehler in Systemkomponenten wie Dateisystem und Netzwerkprotokollen können diesen Effekt haben. Auf Mikrokern-Betriebssystemen ist dies in der Regel kein Problem: Fehlfunktionen in Treibern, Dateisystem- oder Netzwerksoftware bringen nur diese einzelnen Komponenten zum Absturz. Wieso dem so ist und wieso Mikrokerne dieses Problem lösen, wird später beschrieben.

Denkt man an die grundsätzlichen Probleme beim Einsatz von Linux auf einem Arbeitsplatzrechner, fällt einem zunächst auf, daß die Verfügbarkeit von Treibern in keinem Vergleich zu der der Windows-Familie steht - und dies liegt nicht nur an der geringeren Verbreitung von Linux, sondern vielmehr daran, daß Hardwarehersteller für jede einzelne Version des Kerns eine eigene Version mitliefern müßten: Ein kompiliertes Kernelmodul für Version 2.4.0 läuft in der

Regel nicht auf 2.4.18, und um ein für die Serie 2.2 entwickeltes Modul im Kernel 2.4 zu verwenden, genügt noch nicht einmal ein erneutes Übersetzen - der Treiber muß an die neue Version angepaßt werden. Auch dieses Problem ist bei Mikokernen nicht vorhanden: Windows XP (ein als Mikrokern konzipiertes System) aus dem Jahr 2001 beispielsweise kann aufgrund seines Designs mit Treibern und Systemsoftware zusammenarbeiten, die für die im Jahr 1996 aktuelle Version 4.0 entwickelt wurden.

An einem weiteren Problem haben sowohl Windows 95 als auch Linux zu kämpfen: Die Weiterentwicklung eines großen Kerns wird immer schwieriger. Dies sieht man z.B. daran, daß die Fehlerbehebung bei Windows 95 sehr bald kaum mehr stattfand, grundsätzliche Fehler im Kern blieben von Windows 95 bis Windows ME unverändert. Selbst wenn man dies als politische Entscheidung von Microsoft sieht - auch die Weiterentwicklung von Linux ist seit der Version 2.4 ins Stocken gekommen: Neue, extern entwickelte Funktionen wie etwa das Dateisysteme XFS sind nach fast einem Jahr immer noch nicht in den Hauptkern integriert worden. Mikrokerne sind demgegenüber ungleich leichter wartbar, da sie per Design modularer sind. Neue Funktionen wie ein Dateisystem lassen sich in der Regel in ein System integrieren, ohne daß der Kern verändert werden müßte.

Diese drei Vorteile, Robustheit, Flexibilität und Modularität, sollen als Motivation für Mikrokerne genügen, da es wohl die entscheidendsten Punkte sind, die der Anwender selbst zu spüren bekommt.

Bevor nun über die Geschichte der Kernarchitektur von den monolithischen Kernen zu den Mikrokernen hingeführt wird, zunächst die Definition, worin sich denn nun ein Mikrokern von konventionellen Designs unterscheidet. Liedtke [4] definiert einen Mikrokern wie folgt: “Traditionell wird das Wort ’Kern’ zur Bezeichnung desjenigen Teils des Betriebssystems verwendet, den alles andere an Software verpflichtend gemeinsam hat. Die Grundsätzliche Idee des Mikrokern-Ansatzes ist es, diesen Teil zu minimieren, d.h. was immer möglich ist außerhalb des Kerns zu implementieren.” Wieso hierdurch die anfangs beschriebenen Probleme gelöst werden können, darauf wird in den nächsten Abschnitten eingegangen.

1.2 Geschichte der Kernarchitektur

Im folgenden sollen die Grundsätze und Eigenschaften von Mikrokernen im Detail behandelt werden, wobei geschichtlich vorgegangen werden soll: Die Grundlagen der Mikrokerne werden aus den Problemen der monolithischen Kerne erarbeitet, und die Probleme der Mikrokerne aus deren Grundlagen.

1.2.1 Monolithische Kerne und ihre Probleme

UNIX (in folgenden sei hiermit die Original-Implementation von AT&T gemeint) besitzt einen typischen monolithischen Kern als Konsequenz aus seiner Entwicklungsgeschichte. Die ersten Versionen von UNIX Anfang der 70er-Jahre liefen auf Maschinen mit einem Speicherausbau um 256 KB und das gesamte System bestand aus weniger als 10000 Zeilen Code. In der Praxis heißt das: Es gibt keine Struktur im Kern, keine Schichtung. Jede Funktion kann jede andere Funktion im Kern aufrufen. Befindet sich im Quelltext der Dateisystemkomponente eine Funktion, welche die Netzwerkkomponente verwenden kann, ist es möglich, daß der Netzwerk-Code diese aufruft. Somit ist in diesem Beispiel das Netzwerk vom Dateisystem abhängig. Letztendlich war also im UNIX-Quelltext so ziemlich alles von allem abhängig, eine Tatsache, welche die Fehlersuche enorm erschwert. A.S. Tanenbaum nennt diesen Ansatz “the big mess” [6]- die große Unordnung.

Neben der schlechten Wartbarkeit hat dieses Design grundsätzlich das Problem, daß es so gut wie unmöglich ist, eine Basiskomponente herauszunehmen oder auszutauschen. Die Funktionen zum (Low-Level-)Umgang mit dem Dateisystem war z.B. so sehr mit der Datenstruktur des Dateisystems auf dem Medium verwoben (“inode”), daß es bei AT&T UNIX quasi unmöglich war, grundsätzlich unterschiedliche Dateisysteme zu implementieren - vielen Schnittstellen fehlt eine gewisse Abstraktion.

Des weiteren hat das Design von AT&T UNIX den Nachteil, daß es aufgrund fehlender Schnittstellen im Kern unmöglich ist, Komponenten zur Laufzeit in den Kern zu laden oder aus ihm zu entfernen. Dies betrifft insbesondere Treiber: Ein Gerätetreiber war einfach ein Stück Code, der mit in den Kern hineinkompiliert wurde. Folglich war es also notwendig, bei der Installation des Systems die Komponenten- und Treiberkonfiguration festzulegen und den Kern selbst zu übersetzen - UNIX wurde unter anderem deshalb im Quelltext ausgeliefert. Auch Linux mußte in den ersten Versionen vom Anwender selbst übersetzt werden, wenn man mit den vorgefertigten Kernen nicht zufrieden war - ladbare Module gab es ursprünglich nicht.

Schließlich führt die Tatsache, daß ein recht großer Kern mit viel Code vollständig im priviligierten Modus der CPU läuft, zu einem Stabilitätsproblem. Will ein Anwendungsprogramm, z.B. die Shell, versehentlich oder absichtlich in den Speicher des Kerns schreiben, so wird dies vom Kern abgefangen und das Programm in der Regel abgebrochen. Die Dateisystemkomponente im Kern etwa darf hingegen überall hin schreiben, was bei einer Fehlfunktion im besten Fall zu unvorhersehbaren Fehlfunktionen, meist aber zu einem Systemabsturz führt.

Ein konventioneller, sogenannter “monolithischer” Kern wie der von AT&T UNIX hat also folgende grundsätzliche

Probleme:

- mangelnde Robustheit: Fehler in Dateisystem, Netzwerk und Treibern können nicht abgefangen werden.
- mangelnde Modularität: Der Code ist schlecht zu warten.
- mangelnde Flexibilität: Aufgrund mangelnder abstrakter Schnittstellen ist man bei Erweiterungen an Implementierungsdetails gebunden, Komponenten können nicht einfach ausgetauscht werden; es gibt keine zur Laufzeit ladbaren Treiber (Rekompilierung nötig)

Ohne das Konzept eines monolithischen Kerns aufzugeben, wurde mit der Zeit versucht, gegen diese Probleme vorzugehen. Auf Modularität kann bei der Neuimplementation eines Kerns z.B. von Anfang an Wert gelegt werden. Im Fall von Linux haben die Entwickler aber momentan dennoch Probleme mit dem Umfang des Systems [**?**]. Auch eine höhere Flexibilität kann durch klare abstrakte Schnittstellen in einem monolithischen Kern erreicht werden. Linux unsterstützt beispielsweise ab Version 2.0 ladbare Module. Selbst gegen die mangelnde Robustheit wurde etwas getan: Da Linux nicht nur von vielen Freiwilligen durchgesehen wird, sondern auch unzählige Wissenschaftler und Studenten auf Quelltextebene damit experimentieren, werden die allermeisten Fehler entdeckt - Linux gilt als einer der stabilsten Kerne. Die Robustheit wurde allerdings nicht durch das Design, sondern durch die im Vergleich zu anderen Kernen ungleich höheren Anzahl der investierten Arbeitsstunden am Quelltext gewonnen, die dem Open-Source-Modell zu verdanken sind.

Einige Probleme können also gemildert werden - ein besseres Design, das diese Probleme von vornherein ausräumt, wäre jedoch die bessere Lösung.

1.2.2 Mikrokerne

Die Mikrokern-Architektur basiert auf dem Grundsatz, den Kern zu minimieren, d.h. alles, was nicht unbedingt im Kernel-Mode laufen muß, in den User-Mode zu verbannen. Das 1985 begonnene Mach-Projekt wurde bekanntesten Vertreter der Mikrokerne der ersten Generation: Mach hat nur noch den Scheduler, die grundlegende Verwaltung des virtuellen Speichers sowie Interprozeßkommunikationsfunktionen im Kern. Das Ansteuern von Geräten, die Verwaltung des Dateisystems und des Netzwerks, sowie große Teile des Pagings sind etwa Aufgaben von User-Mode-Prozessen.

1.2.2.1 Eigenschaften Im folgenden sollen grundlegende Eigenschaften von Mikrokernen der ersten Generation am Beispiel von Mach beschrieben werden [5].

Scheduler Die Verwaltung der Prozessor-Ressourcen geschieht in Mach, indem höheren Schichten Threads und Tasks zur Verfügung gestellt werden. Ein Thread stellt eine Ausführungseinheit dar, während ein Task einen oder mehrere Threads zusammenfaßt, die gemeinsame Betriebsmittel wie Adreßraum und Ports haben.

Ports Abgesehen vom Adreßraum (über den eine Anwendung mit Erlaubnis des Kerns z.B. auf den Bildschirmspeicher zugreifen kann) sind Ports der globale Mechanismus, um auf Betriebsmittel zuzugreifen und mit anderen Tasks zu kommunizieren. Ein Port ist der Endpunkt einer einseitigen Verbindung, über die der Sender dem Empfänger Daten zukommen lassen kann. Über diesen Mechanismus sind Interprozeßkommunikation und Remote Procedure Calls (RPC; Aufruf einer Funktion in einem anderen Adreßraum) möglich. Aber auch Nachrichten von der Hardware, also Hardware-Interrupts, werden über Ports zugestellt.

Virtueller Speicher Der virtuelle Speicher wird zunächst von Mach selbst verwaltet. Es kann aber ein externer Pager eingesetzt werden, d.h. im Fall eines Seitenfehlers in einem Anwendungsprogramm kann dieser von Mach an einen User-Mode-Task weitergegeben werden, der diesen Fehler behandelt.

Server im User-Mode Alles weitere an Funktionalität muß im User-Mode implementiert werden. Dienste wie Dateisystem und Netzwerk sind einfach Server, deren Funktionen man über RPCs in Anspruch nehmen kann. Treiber greifen über Mach Ports auf Hardware zu und bekommen Informationen über Interrupts als Nachrichten über Ports zurück, sie können also problemlos im User-Mode implementiert werden.

1.2.2.2 Vorteile Der Mikrokernansatz löst offensichtlich die drei Hauptprobleme der monolithischen Kerne, mangelnde Robustheit, mangelnde Modularität und mangelnde Flexibilität:

Die höhere Robustheit von Mikrokernen liegt zunächst einmal darin begründet, daß viel weniger Code im Kernel-Mode und stattdessen im User-Mode läuft. Der Kern kann Fehlfunktionen in User-Mode-Komponenten erkennen, abfangen und die betroffene Komponente neu starten. Ein Fehler in einem Treiber hat somit also nicht mehr zwangsläuftig einen Neustart des gesamten Systems zur Folge.

Die hohe Modularität in einem Mikrokern wird vom Design erzwungen, da Komponenten nicht einfach “wild” Funktionen anderer Komponenten aufrufen können - Interaktionen zwischen Komponenten müssen über wohldefinierte Schnittstellen erfolgen, Dateisystem und Netzwerk

sind beispielsweise klar auseinanderzuhalten und eventuelle Interaktionen klar ersichtlich. Dies erhöht natürlich die Wartbarkeit des Codes.

Ein Mikrokern wie Mach erhöht auch die Flexibilität: Die abstrakten Schnittstellen und die hohe Modularität ermöglichen das einfache Austauschen von Komponenten und das Laden und Entladen von Komponenten zur Laufzeit des Systems. Selbst der Pager kann prinzipiell zur Laufzeit ausgetauscht werden.

Weitere Vorteile Abgesehen davon bietet dieses Design weitere Vorteile. Einer davon ist eine erhöhte Sicherheit. In monolithischen Systemen kann möglicherweise ein Treiber für ein Diskettenlaufwerk, da er im Kernel-Mode läuft, alle Informationen abhören, die durch das System laufen, und sie an einen externen Empfänger weiterleiten, oder sie sogar ändern. Ein nicht vertrauenswürdiger Grafiktreiber ist also ein potentielles Sicherheitsloch. In Mikrokernen haben Komponenten wie Treiber keine solchen Sonderrechte.

Des weiteren können Mikrokerne in der Regel mit weniger Speicher auskommen: Werden Treiber oder z.B. Netzwerkprotokolle längere Zeit nicht benötigt, können sie, da sie im User-Mode laufen, ebenso wie Anwendungen ausgelagert werden. In monolithischen Kernen kann diese Funktionalität nur mit zusätzlichem Aufwand erreicht werden.

Auch eine Verbesserung der Reaktionszeit auf Hardware-Interrupts ist möglich: Monolithische Kerne haben oft die Interrupts grundsätzlich deaktiviert, während sich die CPU im Kern befindet - in einem Mikrokern können allerdings Aktivitäten wie etwa die Verwaltung des Dateisystems von Interrupts unterbrochen werden.

Eng damit hängt die einfachere Verwaltung von SMP-Maschinen zusammen: Während es bei konventionellen Kernen einen Mehraufwand bei der Entwicklung darstellt, mehrere Prozessoren (für höhere Performance) gleichzeitig in den Kern zu lassen (Reentranz), können auf einem Mikrokern viele Komponenten ohne weiteres Zutun parallel ausgeführt werden.

Personalities Der bedeutendste zusätzliche Vorteil ist aber, daß ein Mikrokern keine spezielle Betriebssystemarchitektur erzwingt. Linux ist beispielsweise ein UNIX-Kern; es würde keinen Sinn machen, Linux als Kern für Windows zu verwenden, da man so eine zusätzliche Schicht dazwischenlegen müßte, die zwischen den verschiedenen Modellen (Prozesse, Dateisystem, ...) übersetzt. Ein Mikrokern bietet nur eine sehr allgemeine Abstraktion der wichtigsten Komponenten, implementiert aber keinerlei höhere Strategien. Eine Sammlung von Programmen, die auf einem Mikrokern ein bestimmtes Betriebssystem simulieren, nennt man eine Personality. Indem man die Personality austauscht, kann man ein vollständig unterschiedliches System erhalten. Es ist sogar möglich, mehrere Peronalities gleichzeitig laufen zu lassen: Ein Betriebssystem, bei dem eine Windows- und eine UNIX-Schicht, die sich auf derselben Ebene befinden, das Ausführen von Programmen beider Welten ermöglichen, wäre so prinzipiell möglich.

1.2.2.3 Probleme Allerdings haben diese Mikrokerne der ersten Generation einen gravierenden Nachteil: mangelnde Performance. Sehr schnell erkannte man die Interprozeßkommunikation als den entscheidenden Faktor: Was in konventionellen Systemen der Aufruf einer Funktion im Kern war (z.B. Zugriff auf eine Datei), was zwei Moduswechsel bedeutete (Einsprung in den Kernel-Mode und Rücksprung in den User-Mode), wurde in einem Mikrokernsystem zu einer Folge von RPCs: Das Programm ruft den Dateisystemtreiber auf, welches den Plattentreiber aufruft usw. Es sind also erheblich mehr Kernel-User-Moduswechsel erforderlich, da jeder RPC zwei IPC-Nachrichten verschickt (Parameter und Rückgabewert), und somit aus vier Einsprüngen in den Kern und Rücksprüngen zum Aufrufer besteht. Zudem sind pro RPC zwei Adreßraumwechsel (Kontextwechsel) nötig - der Aufruf einer Funktion in einem monolithischen Kern benötigt in der Regel gar keinen Kontextwechsel.

Deshalb versuchte man zunächst, die Interprozeßkommunikation durch optimierten Code zu beschleunigen, allerdings nur mit mäßigem Erfolg: Trotz einer Halbierung der Dauer eines IPC auf Mach war ein UNIX-Systemaufruf immer noch um einen Faktor 10 schneller als ein vergleichbarer Aufruf auf Mach. Im Großen war ein Mikrokern-System in der Ausführung typischer Anwendungen somit langsamer als ein System mit monolithischem Kern.

1.2.3 Erste Ansätze zur Geschwindigkeitssteigerung

Da man die Performancenachteile im Overhead beim Aufruf von Systemfunktionen mit zu vielen IPC-Nachrichten sah, die man nicht weiter beschleunigen konnte, entschloß man sich bei vielen Mikrokern-Projekten kurzerhand, zeitkritische Komponenten wie etwa das Dateisystem und einige Treiber wieder zurück in den Kern zu integrieren.

Diese Lösung ist offensichtlich recht inkonsequent, sie verträgt sich nicht mit der ursprünglichen Definition von Mikrokernen und verspielt einige der ursprünglichen Vorteile: Die Robustheit ist nur noch eingeschränkt erhalten, die Modularität teilweise nicht mehr notwendigerweise gegeben und auch die Flexibilität wird geschwächt. Auch Sicherheit, Speicherverbrauch und SMP-Unterstützung leiden unter diesem Kompromiß. Dennoch gelten die beschriebenen Vorteile immer noch für diejenigen Komponenten, die weiterhin im User-Mode laufen - solche Kompromiß-Mikrokerne sind in diesen Punkten den monolithischen Kernen weiterhin überlegen.

1.2.4 Mikrokerne der zweiten Generation

Einen Mikrokern der zweiten Generation wie L4 machen zwei Eigenschaften aus: die höhere Geschwindigkeit, die es durchaus mit monolithischen Betriebssystemen aufnehmen kann, sowie die höhere Flexibilität, die einen breiteren Einsatzbereich für Mikrokerne bedeutet.

Im Fall von L4 waren die Entwickler der Überzeugung, daß die Performance-Probleme an den jeweiligen Implementierungen und nicht etwa am Mikrokern-Konzept selbst zu suchen sind. Wie genau L4 diese höhere Geschwindigkeit bewerkstelligt, darauf wird im zweiten Teil detailliert eingegangen.

1.3 Überblick über bekannte Betriebssysteme

Bevor der allgemeine Teil über Mikrokerne abgeschlossen und auf L4 im speziellen übergegangen wird, soll ein kleiner Überblick über bekannte Betriebssysteme und ihre Kernarchitektur gegeben werden, so daß man die Möglichkeit hat, einen Eindruck zu gewinnen, wie verbreitet Mikrokerne im Alltag sind.

Im Moment sind (noch) viele monolithischer Kerne im Einsatz, sei es, weil sie sich aus klassischen Systemen entwickelt haben, oder schlicht und einfach aufgrund der entsprechenden anfänglichen Design-Entscheidung. So sind Linux, der Kern von BSD, sowie die Kerne vieler anderer UNIX-Systeme, die sich aus AT&T UNIX heraus entwickelt haben, wie etwa Solaris, monolithische Kerne. Auch Windows 95 arbeitet mit einem aus Windows 3 hervorgegangenen monolithischen Kern.

Andererseits gibt es aber auch UNIX-Systeme mit Mikrokernen: GNU/HURD ist etwa eine Sammlung von Servern, die eine UNIX-Personality auf Mach bereitstellt. ChorusOS ist ein von Sun Microsystems vertriebenes Mikrokern-UNIX, das gerne in eingebetteten Systemen (wie etwa dem Premiere World Empfänger “d-box 2”) eingesetzt wird.

Viele ursprüngliche Mikrokern-Systeme verfolgen aber die beschriebene Kompromißlösung, indem sie einige Komponenten wieder im Kernel-Mode laufen lassen. Windows NT, in der Version 3.1 und 3.5 ein System mit Mikrokern-Konzepten, intregrierte beispielweise ab Version 4.0 das Grafiksystem zurück in den Kern - DirectX-Spiele wären sonst in Windows XP auch kaum möglich. Mac OS X, das neue Betriebssystem für Apple Macintosh Rechner, basiert sogar auf Mach, dem wohl bekanntesten Mikrokern, in der Version 3.0. Aber auch hier wurden Treiber, Dateisystem, Netzwerk und sogar die UNIX-Schicht zurück in den Kern integriert.

Mikrokerne der zweiten Generation sieht man in der Praxis nur selten - als ein bekannter Vertreter ist neben L4 ist nur noch Neutrino zu nennen, der Kern des Echtzeit-UNIX-Systems QNX, welches sich aber bei eingebetteten System hoher Beliebtheit erfreut.

2 Der L4-Mikrokern

L4 ist ein Mikrokern der zweiten Generation, der nicht nur die Geschwindigkeitsprobleme der ersten Generation lösen, sondern auch eine höhere Flexibilität erreichen will, so daß auf ihm ein breites Spektrum an Personalities für verschiedenste Einsatzzwecke laufen kann. Er wurde Mitte der neunziger Jahre von Jochen Liedtke bei der GMD, bei IBM und an der Universität Karlsruhe zunächst für Intel x86-Prozessoren entwickelt.

2.1 Geschwindigkeit

Da das Hauptproblem der Mikrokerne der ersten Generation die Geschwindigkeit war, nahm sich Liedtke zunächst dieses Problems an. Er analysierte die Geschwindigkeit bestehender Kerne und schloß daraus auf Optimierungsmethoden [1].

2.1.1 Geschwindigkeitsanalyse

Anfang der neunziger Jahre gab es bereits eine Menge wissenschaftlicher Veröffentlichungen, die die Geschwindigkeit bestehender monolithischer Kerne und Mikrokerne maßen. Das Hauptaugenmerk richtete sich dabei auf die Interprozeßkommunikation, die aufgrund ihres häufigeren Einsatzes als Flaschenhals ausgemacht wurde. Liedtke faßte diese Messungen zusammen und bewertete die Ergebnisse.

2.1.1.1 Umschalten zwischen Kernel- und User-Mode Mach benötigte auf 486-Hardware in unabhängigen Messungen etwa 900 Takte für einen einfachen Systemaufruf. (Üblicherweise wird für solch einen Test “getpid” oder eine entsprechende Funktion verwendet, da diese effektiv nur eine Zahl ausliest und zurückliefert.) In diesem Fall muß also in den Kernel-Mode und danach wieder zurück in den User-Mode gewechselt werden. Oft wurden die hohen Kosten eines Moduswechsels auf Prozessorseite für die geringe Geschwindigkeit solcher Aufrufe verantwortlich gemacht: Ein 486 benötigt immerhin ca. 80 Takte, um in den Kernel-Mode und 20 Takte, um zurück in den User-Mode zu wechseln. Zum Vergleich: Eine Addition benötigt auf dieser Hardware nur einen Takt. Dennoch rechtfertigt dies nicht die hohen Kosten in Mach - es existieren 800 Takte Overhead, denn der Kern verrichtet bei einem solchen “leeren” Kernaufruf effektiv keine Arbeit. An dieser Stelle besteht also Optimierungsbedarf.

2.1.1.2 Umschalten zwischen Adreßräumen (Kontextwechsel) Eine weitere Operation, die als sehr teuer ange-

sehen wurde, ist das Umschalten zwischen Adreßräumen, also zwischen Tasks/Prozessen.

Schaltet man zwischen Tasks um, die ja im allgemeinen eine vollkommen unabhängige Speicherkonfiguration und somit eine unterschiedliche Seitentabelle haben, kann der TLB (Translation Lookaside Buffer), der die Umsetzung einer virtuellen in eine physische Adresse unterstützt, nicht länger verwendet werden. Bis der TLB wieder entsprechend gefüllt ist, müssen bei jedem Speicherzugriff zusätzliche Verzögerungen eingerechnet werden. Diese schlecht zu messenden Verzögerungen, die bei monolithischen Kernen wegen der selteneren Adreßraumumschaltung geringer ausfallen, sind aber rechnerisch typischerweise bei unter 50 Takten pro Kontextwechsel anzusiedeln. Performanceprobleme liegen auch hier bei den Implementierungen und nicht beim Konzept.

2.1.1.3 IPC Das Senden von Nachrichten über IPC-Mechanismen besteht nicht nur aus Modus- (Kernaufruf) und Adreßraum-Wechseln (Kern muß zum Kopieren den Adreßraum des Empfängers einblenden), sondern auch aus dem Übertragen der Nachricht. An dieser Stelle wurden Mikrokerne sowieso schon erheblich optimiert, mit 200 Mikrosekunden auf 486-Hardware für das Hin- und Zurückschicken einer Ein-Byte-Nachricht liegt Mach um einen Faktur 2 bis 4 unter diversen UNIX-Kernen. Da IPC im Mikrokern-Design sehr viel häufiger benötigt wird, genügt dies jedoch nicht. Da Kerne wie Spring, Exokernel und der von Liedtke entwickelte L4-Vorläufer L3 dieselbe Aufgabe auf derselben Hardware in etwa 10 Mikrosekunden erledigen können, wurde auch hier gezeigt, daß es sich um Implementierungsprobleme handelt.

2.1.1.4 Schlußfolgerung Diese Erkenntnisse leiteten Liedtke zu der Schlußfolgerung, daß nicht das Mikrokern-Konzeption, sondern nur die bestehenden Implementierung für die Geschwindigkeitsprobleme verantwortlich gemacht werden müssen. Viele Mikrokerne entstanden schrittweise als monolithischen Kernen und hatten immer noch mehrere hundert Systemaufrufe. Der L4-Kern zeigt, daß es auch schneller geht.

2.1.2 Ansätze zur Geschwindigkeitssteigerung (Auswahl)

2.1.2.1 Grundsätze Es gibt keinen einfachen Trick, der einen Mikrokern beschleuningt, sondern vielmehr eine große Anzahl kleiner Details, die, wendet man sie alle richtig an, zusammen für die Geschwindigkeitssteigerung sorgen. Dazu sind aber drei Grundsätze nötig:

- Der Kern Bottom-Up konstruiert.
- Er bietet dem User-Mode eine *minimale* Abstraktion der Maschine.
- Er ist notwendigerweise in Assembler geschrieben.

Beim Design des L4-Mikrokerns verfolgte Liedtke also einen anderen grundlegenden Ansatz: Es wurde nicht aus bestehenden Kernen alles weggenommen, was man auch im User-Mode ausführen konnte, sondern von unten her konstruiert: Welche Funktionen sind notwendig, um die gewünschte Flexibilität zu erreichen, und wie sieht das Modell mit bester Ausnutzung der Prozessorleistung aus?

Die minimale Abstraktion bedeutet in der Praxis, daß die Funktionen, die der Kern zur Verfügung stellt, sowie die Protokolle hierfür so primitiv wie möglich sein sollen. Mächtigere Funktionen können im User-Mode realisiert werden.

Alle Mikrokerne der ersten Generation waren in einer Hochsprache geschrieben, mit einem geringen architekturabhängigem Teil, um den Kern leichter portierbar zu machen. Liedtke fordert aber einen vollständig in Maschinencode implementierten Kern, da nur so die Eigenschaften einer Architektur optimal ausgenutzt werden können. Er schlägt sogar vor, für ansich kompatible Prozessoren wie dem 486 und den Pentium unterschiedliche Kerne zu implementieren, da aufgrund der internen Architektur auf den verschiedenen Prozessoren unterschiedliche Algorithmen schneller sind. Den Kern in Assembler zu schreiben sieht Liedtke sogar als Vorteil für die Portierbarkeit: Der Kern bietet eine Basis für Portierbarkeit: Alles an Code über dem recht kleinen Kern ist architekturunabhängig.

2.1.2.2 Methoden zur Geschwindigkeitssteigerung im Detail Im folgenden werden einige Methoden aufgezählt, die angewandt werden können, um ein System mit Mikrokern zu beschleunigen. Diese Methoden stammen aus den Bereichen Kern-Architektur, Algorithmen und Implementierung in Assembler.

Kombination mehrerer IPC-Aufrufe Ein RPC benötigt normalerweise zwei Systemaufrufe: einen zum Absetzen des Befehls und einen zum Empfangen der Nachricht. Der Server auf der anderen Seite ruft ebenso den Kern zweimal auf: zum Empfangen des Befehls und zum Zurücksenden des Rückgabewertes. Diese sehr häufigen Fälle können im Kern berücksichtigt werden, so daß es spezielle IPC-Aufrufe “call” und “reply & receive next” gibt, welche die Anzahl der Moduswechsel bei RPC halbieren.

Synchrone IPC Zudem gewinnt man mit einem synchronen Modell für die Kommunikation Geschwindigkeit: Mach etwa stellt gepufferte Ports zur Verfügung. Bei der synchronen Übertragung schickt der Sender seine Nachricht und blockiert, bis diese entgegengenommen wird. Auf den anderen Seite blockiert entsprechen der Empfänger, bis eine (gewünschte) Nachricht für ihn ankommt. Timeouts können verwendet werden, für den Fall, daß der Empfänger

keine Nachricht annimmt bzw. kein Sender eine Nachricht verschickt. Diese einfachere Form der IPC spart Takte bei einfacher Kommunikation mit Servern; komplexere Formen (Mach Ports, UNIX Pipes) können auf Basis dieses Modells im User-Mode implementiert werden.

Komplexere Nachrichten Kommunikation über Ports wie bei Mach ermöglicht wie bei UNIX-Pipes nur das Verschicken eines Stroms von Datenbytes - komplexere Datenstrukturen müssen zunächst serialisiert und bei der Gegenstelle wieder auseinandergenommen werden. Dies verleitet dazu, beispielsweise einen Befehlcode und seine Parameter als separate Nachrichten zu versenden. Ein komplexeres Nachrichtenformat mit beispielsweise einem String und beliebig vielen Zeigern auf weitere Daten kann die Anzahl der verschickten Nachrichten deutlich verringern.

Kopieren mit Temporärem Mapping Das Kopieren der IPC-Daten in den Adreßraum des Empfängers erfolgt meist in zwei Schritten: Zunächst werden die Daten in den Adreßraum des Kerns kopiert, und dann, nachdem der Adreßraum des Empfängers eingeschaltet wurde, schließlich an sein Ziel kopiert. Blendet der Kern jedoch mit Hilfe der Mechanismen des virtuellen Speichers den Zielbereich in den aktuellen Adreßraum ein, kommt er mit nur einer Kopie aus, was die Geschwindigkeit bei längeren Nachrichten effektiv verdoppelt.

Datenstrukturen im Kern Viele Kerne verwenden verkettete Listen für die Prozeß/Task/Thread-Kontrollblöcke. Arrays haben hier einen deutlichen Geschwindigkeitsvorteil. Die dadurch erzwungene Beschränkung der Anzahl an Ausführungseinheiten und der zusätzliche Speicherverbrauch sollten in der Regel praktisch nicht von Belang sein.

Zusätzliche Datenstrukturen etwa für die Liste der blockierten bzw. bereiten Threads oder für die zu einem Task gehörigen Threads können vermieden werden: Die bereiten Threads usw. kann man z.B. über eine doppelt verkettete Liste innerhalb des Kontrollblocks verwalten. Jeder Block hält einen Zeiger auf den nächsten bereiten Thread. Da beim Ende einer Zeitscheibe sowieso auf den aktuellen Block zugegriffen wird, ist es so besonders einfach, den nächsten auszuführenden Thread zu ermitteln. Die Zugehörigkeit von Threads zu Tasks kann in ihren Identifiern festgelegt werden. Wählt man den Wertebereich der Thread-IDs groß genug, so kann man sehr viele Informationen darin kodieren, auf die man mit einfachen AND- und Shift-Operationen zugreifen kann.

Parameter-Übergabe am Kernel-Interface über Register Bei Funktionsaufrufen in Hochsprachen werden die Parameter in der Regel über den Stack übergeben: Der Aufrufer “pusht” (beliebig viele) Parameter auf den Stack, der Aufgerufene lädt sie wieder. Effizienter ist die Übergabe von Parametern über die Prozessorregister. Insbesondere bei Kern-Aufrufen mit einer recht geringen Anzahl von Parametern bringt die Übergabe der Daten über die Register einen deutlichen Geschwindigkeitsvorteil. Das einfachere Interface hat zudem noch den Vorteil, daß der Compiler den Code für das Zusammenstellen der Parameter eher in den Code einfügt - bei komplizierterer Übergabe verwenden Compiler hierfür oftmals langsamere Unterroutinen.

Kurze Nachrichten über Register Da an vielen Stellen nur kurze Befehle, Bestätigungen und Statuscodes über IPC verschickt werden (insbesondere bei RPC), ist es sinnvoll, auch die Daten (oder zumindest Teile davon) beim Aufruf der IPC-Funktionalität des Kerns direkt in die Prozessorregister zu laden. Ansonsten würde der Aufrufer dem Kern einen Zeiger auf die Daten übergeben, der Kern kopiert die Daten in den Adreßraum des Empfängers, wo dieser sie wieder ausliest. So lädt nun der Sender z.B. einen Befehlscode einfach nur in ein Register, und der Empfäger kann ihn nach dem Empfang der Nachricht direkt auswerten. Bei längeren Nachrichten können die ersten Bytes der Nachricht in Registern übergeben werden.

Cache-Optimierung Auf unterster Ebene, nämlich auf der Ebene der Implementierung, sind noch viele wichtige Optimierungen möglich.

Obwohl man heute kaum mehr die Codegröße optimiert, ist das im Kern sehr sinnvoll. Der Bereich des Kerns, der für die Interprozeßkommunikation verantwortlich ist, sollte immer im Code-Cache des Prozessors gehalten werden können. Er muß somit sehr klein sein und sich möglichst nicht über eine Seitengrenze erstrecken. Ebenso können die Datenstrukturen des Kerns für den Cache optimiert werden, z.B. indem man auch die an Seitengrenzen ausrichtet und oft benutzte Daten kompakt zusammenfaßt.

Prozessor-Betriebsmittel-Sicherung bei Bedarf Um das Sichern spezieller Prozessorregister, die nur selten verwendet werden, wie etwa die Gleitkommaregister, zu beschleunigen, was bei jedem Thread-Wechsel mindestens einen Takt pro Register verbrauchen würde, kann man das Sichern einfach zunächst einmal unterlassen und den Prozessor eine Exception auslösen lassen, sobald sie von einem nachfolgenden Thread doch verwendet werden. Erst dann werden sie gesichert. Verwendet beispielsweise nur ein Thread die Gleitkommaregister, kommt man so vollständig ohne Sichern aus.

Weitere Optimierungen Die weiteren Optimierungsmethoden lesen sich wie das Pentium-

Optimierungshandbuch: Eine bedingte Verzweigung benötigt mehr Takte, wenn sie genommen wird, deshalb sollte man den wahrscheinlichsten Ausführungspfad so konstruieren, daß er ohne Verzweigungen auskommt. Dies ist eine Optimierung, die ein Compiler nicht selbständig machen könnte, da er nicht weiß, welche Ereignisse wahrscheinlicher sind als andere. Ebenso kann man weitere architekturspezifischen Merkmals ausnutzen, wie etwa die Tatsache, daß man bei x86-Prozessoren die unteren 8 bzw. 16 Bit der 32-Bit-Register besonders schnell ansprechen kann. Auch diese Optimierung wird von Compilern in der Regel nicht verwendet. Einsetzen kann man das z.B. bei Thread-IDs, in denen man den dazugehörigen Task in den unteren 16 Bit codieren könnte.

2.2 Architektur von L4

Nach dieser Hinführung über Geschwindigkeitsoptimierungen soll zur konkreten Architektur des L4-Mikrokerns übergegangen werden.

2.2.1 Grundlagen

Der L4-Kern bietet eine einfache Abstraktion der wenigen Komponenten, die zumindest teilweise im Kern realisiert werden müssen, da sie auf priviligierte Befehle angewiesen sind: Neben der Maschinenverwaltung (Abfangen und Weiterleiten von Interrupts und Exceptions) kümmert sich L4 um die grundlegende Speichervergabe, das Thread-Managemant und das Scheduling, sowie die Interprozeßkommunikation.

2.2.1.1 Speicher L4 stellt ein grundlegendes abstraktes Adreßraumkonzept zur Verfügung, überläßt die Implementierung eines Speicherverwalters und eines Pagers aber User-Mode-Servern.

Speicher kann in Form von Seiten flexibler Größe hierarchisch vergeben werden. Der Kern als grundlegender Speicherverwalter vergibt zunächst physische Seiten an Tasks. Empfänger dieser Seiten können sie (typischerweise im Rahmen einer mächtigeren API) an andere Task weitergeben. In der Praxis bedeutet das, daß die Seiten (zusätzlich oder ausschließlich) in den Adreßraum des Empfängers eingeblendet werden. Das Vergeben von Seiten an einen anderen Thread erfolgt über IPC: Mit jeder Nachricht können Speicherseiten übergeben werden; der Kern blendet jeweils den Speicher in den Adreßraum des Empfängers ein, wenn solch eine Nachricht verschickt wird.

Der Kern verfolgt bei der Vergabe seines Speichers das Prinzip, daß der erste Anforderer den Speicher bekommt; es existiert kein Rechtesystem.

2.2.1.2 Thread Der in L4 eingebaute einfache Scheduler kennt “Threads” als grundsätzliche Ausführungseinheit. Einer oder mehrere Threads werden in einem Task ausgeführt. Ein Task entspricht einem Adreßraum. Im Detail ist dies so gelöst: Ein Task besteht immer aus der maximalen Anzahl möglicher Threads (128 auf x86), wobei bei der Erstellung des Tasks nur einer aktiviert ist. Threads werden also nicht erstellt, sondern nur aktiviert. Entsprechend wird ein Thread beendet, indem man ihn ewig blockieren läßt. Auch Tasks werden nicht wirklich erstellt - will man einen neuen Task erzeugen, gibt man die Tasknummer mit an, d.h. man kann hierdurch einen bestehenden Task überschreiben und somit beenden. Ebenso ist es möglich, sich das Recht zu reservieren, später einen Task zu erstellen, indem man einen leeren Task (kein Adreßraum, kein aktiver Thread) erzeugt, den man später überschreibt. Auch hier gilt das Prinzip, wer zuerst anfordert, bekommt, soviel er will: Ein kritischer Systemprozeß kann so Tasks gleich bei der Initialisierung reservieren.

Die Beschränkung auf eine feste Anzahl an Threads pro Task stellt in der Praxis keine Flexibilitätseinschränkung dar: Benötigt man mehr Threads pro Task, so erstellt man einfach einen neuen Task mit demselben Adreßraum.

2.2.1.3 Kommunikation Die Kommunikation zwischen Threads erfolgt in L4 immer synchron und ungepuffert. Der Sender blockiert also, bis die Nachricht entgegengenommen wird (oder ein Zeitlimit abgelaufen ist), und der Empfänger blockiert entsprechend, bis ihm eine gewünschte Nachricht geschickt wird (oder Zeitlimit).

Ein IPC-Aufruf kann entweder nur eine Nachricht schicken, nur eine Nachricht empfangen, eine Nachricht schicken und gleich darauf vom Empfänger eine Antwort erwarten (für RPC), oder eine Nachricht schicken und die Nachricht eines beliebigen Senders empfangen (reply & receive).

Als Nachricht kann ein “direkter” String (die ersten paar Byte davon stehen in den Registern), eine beliebige Anzahl “indirekter” Strings (Aufrufer übergibt Zeiger) sowie Speicherseiten übertragen werden.

Clans & Chiefs Um die Nachrichtenübertragung überwachen und einschränken zu können, ist ein einfaches Sicherheitskonzept namens “Clans & Chiefs” im Kern realisiert. Ein Clan ist eine Menge von Tasks und/oder weiteren Clans (rekursiv). Jeder Clan hat einen zusätzlichen Task, seinen Chief. Bei der Interprozeßkommunikation wird eine Nachricht nach bestimmten Regeln möglicherweise zunächst zu einem Chief weitergeleitet. Dieser kann die Nachricht verwerfen, ändern oder weiterschicken. Auch hier greift der Kern evtl. wieder ein und leitet sie an einen weiteren Chief weiter. Die Regeln sind wie folgt:

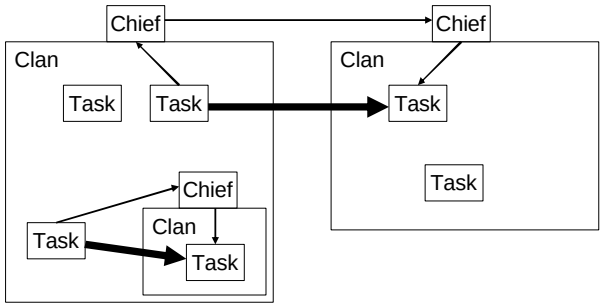
- Sind beide Parteien in demselben Clan, wird die Nachricht direkt geschickt.

- Ist der Empfänger außerhalb des aktuellen Clans, so wird sie an den Chief des Clans weitergeleitet.
- Ist der Empfänger innerhalb eines Clans im aktuellen Clan, so wird sie an den Chief des inneren Clans weitergeleitet.

Eine Nachricht kann so über beliebig viele Chiefs weitergeleitet werden, von jedem modifiziert oder abgefangen werden.

Eine UNIX OS Personality sollte z.B. die Prozesse (die auf L4 Tasks abgebildet werden) jedes Benutzers zu einem Clan zusammenfassen, so daß UNIX-Signale (die auf L4-IPC-Nachrichten abgebildet werden) wie SIGTERM oder SIGKILL, die an Prozesse eines anderen Benutzers geschickt werden, vom Chief des Empfänger-Clans angefangen werden. Ebenso kann man dieses Konzept verwenden, um gewisse Rechte einzuschränken, indem man z.B. eine Sandbox für nicht vertrauenswürdige Anwendungen erstellt (RPC an Dateisystem-Server usw. werden abgefangen). Auch inkompatible Server kann man so kompatibel zueinander machen: Ein Chief übersetzt einfach alle betreffenden IPC-Nachrichten zwischen den beiden. Für die Sicherheit eines Systems in Netz gegenüber Angriffen von außen kann man sorgen, indem man einen Chief pro Rechner die eintreffenden Nachrichten filtern läßt.

Die Abbildung zeigt ein einige teils verschachtelt konstruierte Clans. Die dicken Pfeile geben die beabsichtigte Kommunikationsrichtung an und die dünnen Pfeile den wirklichen Informationsfluß, sofern kein Chief die Nachricht abfängt.



2.2.2 Flexibiliät

Die größtmögliche Flexibiliät war bei L4 eines der Designziele. Möglichst unterschiedliche Konzepte sollten im User-Mode implementierbar sein, so daß unterschiedlichste Betriebssystem-Architekturen auf Basis von L4 implementiert werden können - L4 soll “policy-free” sein, Strategien sollen Aufgaben des User-Mode sein. Somit übertrifft L4 Mikrokerne der ersten Generation, wie etwa Mach, an Flexibilität:

Externer Scheduler Auf L4 kann ein User-Mode-Scheduler Zeitscheiben an Prozesse vergeben und einrichten, daß er selbst als nächster Thread wieder aufgerufen wird. So kann die volle Kontrolle beim User-Mode-Scheduler liegen; es ist aber auch möglich, daß ein externer Scheduler nur in die Strategie des in L4 eingebauten Schedulers eingreift, indem er Prioritäten von Threads zur Laufzeit ändert.

Externer Pager Seitenfehler werden von einem Thread im User-Mode behandelt (Mitteilung über IPC). Dieser muß sich darum kümmern, daß die Seite nachgeladen wird (RPC an Platten-Server) und übergibt die Daten dem Thread, der den Seitenfehler verursacht hat. Ein externer Pager kann gegebenenfalls für bestimmte zeitkritische Threads verhindern, daß sie ausgelagert werden.

Externer Speichermanager Wie bereits beschrieben, vergibt der Kern nur physische Speicherseiten, die komplette Speicherverwaltung liegt in Händen eines oder mehrerer Threads im User-Mode.

Externer Exception-Handler Über Exceptions in einem Thread wird per IPC ein weiterer Thread benachrichtigt. Dieser kümmert sich darum, z.B. den Task zu beenden. Auf diese Weise können auch Programme ausgeführt werden, die für ein anderes Betriebssystem geschrieben wurden - versucht eine Anwendung, eine Funktion “seines” Kerns aufzurufen, führt dies typischerweise zu einer Prozessor-Exception, die der Exception-Handler abfangen und das Verhalten dieser Operation simulieren kann.

Externer Interrupt-Handler Ebenso werden Hardware-Interrupts per IPC an Threads, also in der Regel Treiber übermitelt. Wegen der hohen Geschwindigkeit von IPC in L4 ist dies problemlos möglich.

2.2.3 Das L4-API

Auf die bewußt gering gehaltene Funktionalität von L4 kann man mit nur sieben verschiedenen Kern-Aufrufen zugreifen. Aufgrund dieses geringen Umfangs ist es möglich, das API hier zu im Detail zu beschreiben, um die Schnittstelle zwischen Threads und dem Kern ein wenig zu veranschaulichen [8].

2.2.3.1 ipc Der Aufruf von “ipc” erledigt beliebige Formen der Interprozeßkommunikation. Je nach Registerbelegung sind unterschiedliche Operationen möglich. Im einfachsten Fall werden nur Daten gesendet (“SEND”), Daten empfangen (“RECEIVE”) oder von einem bestimmten

Absender (“RECEIVE FROM”) oder von einer Interrupt-Quelle empfangen (“RECEIVE INTR”). Letztere Funktion muß mit “ASSOCIATE INTR” vorbereitet werden. Das bereits beschriebene Senden und darauffolgende Empfangen von derselben Quelle (“CALL”) und das Senden und darauffolgende Empfangen von einer beliebigen Quelle (“SEND/RECEIVE”) wird ebenfalls über diesen Kern-Aufruf bewerkstelligt.

Über einen Trick wurde in diese Schnittstelle Zugriff auf eine Zeitmessungsfunktion integriert: Gibt man an, daß auf eine Nachricht eines Senders mit einer ungültigen ID gewartet werden soll, und stellt man einen Timeout ein, kann so eine Verzögerung erreicht werden (“SLEEP”).

2.2.3.2 id_nearest “id_nearest” wird dazu verwendet, entweder die eigene Thread-IP oder den nächsten Chief zu ermitteln, der eine Nachricht empfangen würde, wenn man sie an einen gegebenen Thread schicken würde. Letztere Verwendung ist z.B. notwendig, um die Hierarchie an Clans & Chiefs herauszubekommen.

2.2.3.3 fpage_unmap Mit “fpage_unmap” wird der Kern angewiesen, eine Speicherseite, auf die man selbst Zugriff hat und die man weitergegeben hat, aus dem Adreßraum des Empfängers (und dem dessen Empfängers, falls er sie wieder weitergegeben hat usw.) zu entfernen. Ein Pager ruft diese Funktion auf, nachdem er eine Speicherseite ausgelagert hat. Weitere Zugriffe darauf erzeugen daraufhin Seitenfehler.

2.2.3.4 thread_switch Ein Thread kann mit “thread_switch” freiwillig den Prozessor abgeben und optional bestimmen, welcher Thread den Rest seiner Zeitscheibe haben soll. Ein externer Scheduler könnte sich (und nur sich) so unterschiedlich große Zeitscheiben zuweisen lassen und als erste Aktion die Zeitscheibe einem seiner verwalteten Threads übergeben.

2.2.3.5 thread_schedule “thread_schedule” ist ein weiterer Aufruf, der für einen externen Scheduler gedacht ist: Hiermit werden die Prioritäten von Threads gesetzt und ausgelesen.

2.2.3.6 lthread_ex_regs Mit “lthread_ex_regs” werden die Register eines Threads innerhalb des aktuellen Tasks ausgelesen, belegt oder mit denen des aktuellen Threads vertauscht. Durch das Vertauschen kann die Kontrolle ohne Einflußnahne des Schedulers sofort an einen anderen Thread innerhalb desselben Tasks übergeben werden, durch das Belegen startet oder stoppt man einen der immer existierenden Threads: Man setzt einfach den Befehlszähler des Threads auf neuen Code bzw. auf eine Anweisung, die ewig blockiert.

2.2.3.7 task_new “task_new” erstellt schließlich einen neuen Task; der erstellende Task wird sein Chief. Da man die Nummer des Tasks angibt, kann man so einen Task überschreiben oder nur löschen (Überschreiben mit leerem Task ohne Adreßraum und aktivierte Threads). Das Überschreiben funktioniert nur, wenn der aktuelle Task ein Chief ist, in dessen Clan (direkt oder indirekt) sich der zu löschende Task befindet.

2.2.4 Das L4-ABI auf x86

Damit man sich ein wenig vorstellen kann, wie ein Kern-Aufruf in der Praxis aussieht, soll an dieser Stelle das L4 Application Binary Interface (ABI) für x86-Prozessoren kurz vorgestellt werden. Während das API die Signaturen aller Aufrufe definiert, legt das ABI die Befehle zum Einsprung in den Kern sowie die genauen Registerbelegungen fest.

Der Einsprung in den Kern erfolgt wie etwa auch bei Linux mit dem Auslösen eines Softwareinterrupts über den “int”-Befehl. Entsprechend ihrer Reihenfolge in der vorherigen Beschreibung sind den 7 Systemaufrufen die Interrupts 0x30 bis 0x36 zugewiesen. Die Parameter werden, mit Ausnahme von Strings, allesamt über die Register übergeben. Es macht wenig Sinn, in diesem Rahmen die Belegungen aller 7 x86-Register für Aufruf- und Rückgabewerte aller 7 Systemaufrufe zu beschreiben, deshalb soll jetzt stellverstretend für alle nur “task_new” beschrieben werden.

Der Systemaufruf “task_new” erwartet in EDI:ESI den 64-Bit-Wert der Nummer des betreffenden Tasks, in EBP:EBX den 64-Bits-Wert der Nummer des Pager-Threads, der dem Task zugewiesen werden soll (oder 0 für einen inaktiven Task ohne Adreßraum), in ECX und EDX den Stackpointer bzw. den Befehlszähler für den ersten Thread. Wird ein aktiver Task erstellt, so steht in EAX die gewünschte Priorität, ansonsten die unteren 32 Bit des Chiefs, den der Task unterstellt sein soll. So ist es möglich, das Recht, einen Task zu erstellen, einem weiteren Chief zu übergeben. Als Rückgabewert steht in EDI:ESI die Nummer des erstellten Tasks bzw. 0, falls die Erstellung fehlschlug (Rechteproblem).

Task IDs und Thread IDs sind 64-Bit-Werte, in denen mehrere Informationen codiert sind. In einer Thread ID stehen z.B. 7 Bit für die Nummer des Threads innerhalb des Tasks, 11 Bit für den dazugehörigen Task, 11 Bit für den Chief, dem der Task unterstellt ist usw.

2.2.5 Wie funktioniert dann...?

Bei der Beschreibung der L4-Architektur sowie der Schnittstelle zum Kern wurde an vielen Stellen schon daraf eingegangen, wie dann typische Betriebssystemmechanismen realisiert werden. Im folgenden sollen einige dieser Mechanismen sowie einige weitere und einige typische Szenarien

noch genauer beschrieben werden.

2.2.5.1 Speicherverwaltung In der Praxis würde man beim Systemstart einen Speichermanager starten, der den gesamten physischen Speicher des Kerns erhält. Anwendungen können nun von diesem Server im Stil eines malloc()-Aufrufs Speicher anfordern. Der Speichermanager blendet dann einen entsprechend großen Bereich seines Speichers in den Adreßraum der Anwendung ein, indem er die Seiten über IPC an den Emfpänger schickt (um das Einblenden kümmert sich natürlich letzendlich der Kern, der diese IPC-Nachricht mitliest). War der zuletzt übergebene Bereich zu groß, kann sie für die neue Anforderung (mit-)verwendet werden, und es ist evtl. keine neue Seite nötig.

Ein Pager würde als Thread im Adreßraum des Speichermanagers realisiert, der bei Seitenfehlern Nachrichten vom Kern geschickt bekommt - die Anwendung, die den Seitenfehler ausgelöst hat, ist in einem Zustand, als hätte sie per IPC eine Seite angefordert: Sie ist blockiert und wird durch eine IPC-Nachricht mit der Speicherseite wieder aufgeweckt. Der Pager entscheidet nun also, welche Seite ausgelagert wird, schickt diese an den Platten-Server, entfernt die Seite aus dem Adreßraum der Anwendung (“fpage_unmap”), holt die gewünschte Seite vom Plattenserver und schickt die Seite der Anwendung, die den Seitenfehler verursacht hat. Das Scheduling des Kerns kümmert sich hierbei automatisch um die notwendige Effizienz: Nach einem Seitenfehler wird die betroffene Anwendung blockiert und der Pager wird ausgeführt. Wenn dieser dann auf den Plattenserver zugreift, wird seine restliche Zeitscheibe automatisch diesem zugeteilt. Der Plattenserver wird in der Regel die Daten nicht sofort von der Hardware geliefert bekommen und wird evtl. vom Scheduler unterbrochen bzw. er blockiert aktiv, da er auf einen Hardware-Interrupt wartet. Da die Anwendung, die den Seitenfehler ausgelöst hat, blockiert ist, ist sichergestellt, daß sie von Scheduler nicht berücksichtigt wird. Sobald der Platten-Server die Daten liefern kann, wird sofort wieder zum Pager umgeschaltet. Die eingelagerte Seite wird der Anwendung über IPC zurückgeliefert, wodurch sie wieder aufgeweckt wird und auch gleich die restliche Zeitscheibe des Pagers bekommt.

2.2.5.2 Thread-/Taskverwaltung, Scheduler Das Erstellen und Löschen von Tasks geschieht wie schon beschrieben über Funktion “task_new” des Kerns. In einer OS Personality würde ein Task-Manager beim Systemstart alle Tasks sich selbst zuweisen, so daß keiner seiner Tasks mehr einen Task über “task_new” erstellen, löschen oder überschreiben kann. Dies ist dann nur noch über einen RPC an den Task-Manager möglich, der sich um eine rechtebasierte Vergabe kümmern kann

Das Erstellen und Löschen von Threads geschieht wie beschrieben über lthread_ex_regs, indem der Befehlszähler einfach auf den neuen Code bzw. eine Blockier-Anweisung gesetzt wird. Diese Operation kann natürlich nur ein Thread innerhalb desselben Tasks ausführen.

Auch die Einflußname eines externen Schedulers wurde zum Teil schon beschrieben. Ein Scheduler kann die Priorität von Threads zur Laufzeit ändern, um so Einfluß auf deren Ausführng zu nehmen. Ebenso verfügt er über die Möglichkeit, immer nur sich selbst Zeitscheiben zuweisen zu lassen und diese dann anderen Threads zu geben.

2.2.5.3 Echtzeit Kombiniert man einige der beschriebenen Mechanismen, können auch Echtzeit-Betriebssysteme konstruiert werden bzw. bestimmte Anwendungen bevorzugt behandelt werden. Ein Scheduler kann z.B. verhinden, daß ein wichtiger Thread weniger als 50% der CPU bekommt. Auf einem großen Universitäts-Zentralrechner, der sowohl als wichtiger Server von Daten im Netz als auch als Arbeitsumgebung von Studenten verwendet wird, könnte man zwei Speichermanager ausführen: Vom einen fordert die Server-Software ihren Speicher an und vom anderen die restliche Software. Die Studenten werden es also nicht schaffen, durch extrem hohen Speicherverbrauch den Kern zum “Thrashing” der Speicherseiten der Serversoftware zu zwingen. Weist man dem Speichermanager des Servers besonders viel Speicher zu, deaktiviert man das Auslagern von Seiten bei diesem ganz, und implementiert man einen Plattentreiber mit einem Mindestdurchsatz für bestimmte Anwendungen, so kann Lösungen finden, so daß auch durch eine hohe Festplattenlast die Geschwindigkeit des Servers nicht unter ein bestimmtes Minimun fallen kann.

Auch bei Multimedia-Anwendungen sowie bei der Steuerung von Maschinen ist es wichtig, daß bestimmte Anwendungen ihr exaktes Timing aufrechterhalten. L4 kommt dem Konstrukteur eines Betriebssystems, das solche Probleme lösen kann, durch seine Flexibilität beim Scheduling sehr entgegen.

2.2.5.4 Treiber Treiber brauchen Zugriff auf zwei Arten von Betriebsmitteln: den in den Hauptspeicher eingeblendeten I/O-Bereich eines Geräts (Grafikspeicher oder Memory-Mapped-I/O) und die Hardware-Interrupts, die das Gerät auslösen kann (I/O-Ports, wie sie bei x86-Prozessoren verwendet werden, verwaltet L4 als spezielle Speicherseiten). Den Speicherberreich bzw. die Ports erhält der Treiber über den Speichermanager, der diesen Bereich gesondert verwalten kann, und die Hardware-Interrupts werden dem Treiber als IPC-Nachrichten zugestellt (“RECEIVE INTR”). Ein Platten-Treiber, der einen Sektor lesen will, schreibt den Befehl zum Lesen in den I/O-Port, den er beim Start vom Speichermanager zugewiesen bekommen hat und blockiert, indem er auf den Interrupt wartet. Durch diesen wird er schließlich auch wieder aufgeweckt (oder es tritt ein Timeout auf, den der Treiber dann behandelt), so daß die Daten aus den I/O-Ports gelesen werden können.

Hat das Gerät DMA-Zugriff, stehen die Daten womöglich bereits im Hauptspeicher.

An dieser Stelle stellt sich die Frage, ob denn ein Treiber, obwohl er im User-Mode läuft, den Speicherschutz umgehen kann, indem er seinem Gerät anweist, per DMA in den Speicher zu schreiben. Über DMA hat der Prozessor keine Kontrolle, deswegen kann man dieses Problem niemals vermeiden. Ein Treiber eines DMA-Gerätes kann also trotz aller Sicherheitsvorkehrungen dennoch das System zum Stillstand bringen, ein Plattentreiber kann sogar den kompletten Speicher auslesen (Speicher Sektor für Sektor auf Platte schreiben und gleich wieder auslesen). Deshalb müssen Treiber von DMA-Geräten zu den vertrauenswürdigen Komponenten gehören.

2.2.5.5 Systeminformation Schließlich stellt sich noch die Frage, wie eine Anwendung herausfinden kann, auf welcher Hardware sie läuft, wieviel Speicher installiert ist und welche Version des L4-Mikrokerns das System verwaltet. Ohne einen weiteren Aufruf des Kerns zu benötigen, erreicht man diese Informationen über eine spezielle Speicherseite, die man vom Kern anfordern kann. In ihr steht die L4-Versionsnummer, Daten über den Aufbau des Hauptspeichers, die Uhr, sowie beliebig viele weitere Informationen. Ein System-Task wird üblicherweise beim Systemstart diese Seite belegen und die Informationen darin per RPC bereitstellen.

3 Implementierungen

Bisher wurde immer vom “Mikrokern L4” gesprochen. Das ist soweit korrekt, Jochen Liedtke hat an der GMD seinen Mikrokern namens L4 entworfen und implementiert. Dieser Kern, auf den im folgenden unter dem Namen L4/x86 verwiesen wird, ist allerdings nicht die einzige Implementierung des L4-Kernel-Interfaces. Da L4/x86 vollständig in x86-Maschinensprache geschrieben war, und Mikrokerne nach Liedtke aus Prinzip nicht portabel sind, mußten L4-Kerne für andere Architekturen vollständig neu implementiert werden. Es gibt allerding auch mehrere Implementationen für x86-Prozessoren: Dies ist hauptsächlich durch das Copyright auf L4/x86 begründet.

3.1 L4/x86

Wie schon gesagt, wurde L4/x86 von Jochen Liedtke an der GMD vollständig in Assembler implementiert. Er ist vollständig und so weit fehlerfrei, daß er für weitere Forschungsprojekte die auf L4 basierten, z.B. an der TU Dresden, verwendet werden konnte. Von allen x86-Implementierungen ist diese die schnellste und mit nur 12 KB auch die kleinste.

Im Jahr 1996 wechselte Jochen Liedtke von der GMD in ein Forschungszentrum von IBM in New York, wo ein Derivat von L4/x86 mit dem Namen “Lava Nucleus” entstand. Im “SawMill”-Projekt will IBM eine Multi-Server-Linux-Peronality auf L4-Basis entwickeln. Das Ergebnis dieses Projekts soll die Forschung an Unix-Servern auf Basis von Mach ersetzen.

3.2 L4/Alpha, L4/MIPS

Wie eingangs erwähnt, konnte kein Code von L4/x86 für andere Prozessoren verwendet werden. Für Alpha und MIPS mußten, um die möglichen Geschwindigkeitsvorteile vollständig zu nutzen, die Schnittstelle zum Kern und die Datenstrukturen im Kern komplett neu entworfen und alles neu implementiert werden. An der TU Dresden und an der Univerity of New South Wales in Australien enstanden im Rahmen von Abschlußarbeiten so die Kerne L4/Alpha für den 64-Bit-Prozessor Alpha von Digital (verwendet in manchen UNIX-Rechnern) bzw. L4/MIPS für die R4000-Serie der 64-Bit-MIPS-Prozessoren (verwendet etwa in SGI-UNIX-Rechnern und im Nintendo 64). Beide Kerne unterliegen der GNU General Public License und werden von der UNSW weiter gepflegt. L4/Alpha ist übrigens die einzige L4-Implementation, die momentan SMP unterstützt, also auf Mehrprozessormaschinen läuft, und von L4/MIPS sagen die Entwickler, es sei der schnellste Kern, der für MIPS überhaupt verfügbar ist.

3.3 Fiasco

Die TU Dresden interessierte sich früh für L4/x86 und begann im Rahmen des DROPS-Projekts (Dresden Realtime OPerating System) weiterführende Forschungen von Echtzeitbetriebssystemen auf Basis des L4-Mikrokerns. Da L4/x86 als Assemblerprogramm jedoch schlecht zu warten war und zudem die GMD den Quelltext nur unter einer allzu restriktiven Lizenz anbot, entschloß man sich, L4 neu zu implementieren, wobei große Teile in C++ geschrieben wurden. Zwar widersprach die Implementation von Teilen in C++ einem der Grundgedanken von L4, dem Lehrstuhl für Betriebssysteme der TU Dresden kam es aber in seinen Forschungen nicht darauf an, zu beweisen, wie performant Mikrokerne sein können (dies hatte L4/x86 bereits gezeigt), sondern, ob es möglich ist, ein Echtzeitbetriebssystem auf einem solchen Kern aufzubauen.

3.4 Hazelnut

Die Universität Karlsruhe, an der Jochen Liedtke nach seiner Zeit bei IBM in New York eine Professur annahm, hat sich zum Ziel gesetzt, auf Basis des L4-Interfaces an Mikrokernen weiterzuforschen. Dazu wurde zunächst auch hier eine Neuimplementation nötig: Da der zu entwickelnde

Kern für Experimente insbesondere mit 64-Bit-Prozessoren verwendet werden sollte, war es nötig, ihn in fast vollständig in C++ zu schreiben. So entstand Hazelnut, eine sehr ausgereifte Version (aktuell ist der Release Candidate 2) eines L4/x86-kompatiblen Kerns. Hazelnut läuft momentan auf Pentium-kompatiblen, sowie auf ARM-CPUs (Prozessor für Embedded Systems).

3.5 Weitere Implementationen

Das L4-Interface wurde noch einige weitere Male für unterschiedliche Architekturen implementiert. So werden unter einer kommerziellen Lizenz Versionen für MIPS und Siroyan OneDSP angeboten. Eine Implementation für PowerPC, die unter der GPL veröffentlicht werden soll, wird gerade an der Univerity of York in Großbritannien entwickelt.

4 Anwendungen von L4

Ein Mikrokern allein ist natürlich für jemanden, der sich nicht für die konkrete Forschung interessiert, nicht sonderlich interessant. Allein hat er noch keinen Nutzen, man kann ihn nicht in der Praxis einsetzen, nicht als Entwicklungssystem für eigene Anwendungen, weil die Programmierung besonders umständlich ist, und erst recht nicht als Desktop-Betriebssystem, da so grundlegende Funktionen wie das Dateisystem oder eine Treiberarchitektur fehlen. Deswegen soll im folgenden auf die Anwendungen von L4 eingegangen werden, d.h. Betriebssystem-Projekte, die L4 als ihren Kern verwenden. Abgesehen von Bestrebungen, die noch keine Erfolge zeigen konnten, wie der Versuch, den Multi-Server HURD von Mach auf L4 zu portieren (GNU-Projekt), oder einen Multi-Server auf Linux-Basis für L4 zu entwickeln (SawMill-Projekt bei IBM), gibt es nur wenige Projekte, für die es Sinn machen würde, sie in diesem Rahmen zu besprechen. Mungi von der University of New South Wales etwa ist ein “Single-Address-Space”-Betriebssystem, d.h. alle Aktivitäten laufen in demselben Adreßraum. Dieses Projekt hat allerdings nicht allzuviel mit L4 direkt zu tun, es wurde im Grunde nur deshalb auf L4 als Basis zurückgegriffen, um sich die Entwicklung grundsätzlicher Kernfunktionen zu ersparen. Dieser Anschnitt konzentriert sich im weiteren nur auf das DROPS-Projekt der TU Dresden, und insbedondere auf L4Linux, einem Teilbereich von DROPS.

4.1 DROPS

Wie bereits erwähnt, versucht man an der TU Dresden, ein Echtzeitbetriebssystem auf Basis des L4-Mikrokerns zu konstruieren. Dabei stellte sich gleich zu Anfang ein Problem: Die Server für L4 mußten an einer separaten Maschine entwickelt werden (Cross-Development), es war nicht möglich, auf einem L4-System selbst zu arbeiten, da davon nur der Kern existierte. Man entschloß sich deshalb, eine OS Personality auf Basis von L4 zu entwickeln. Man hatte die Wahl, entweder ein UNIX-System als eine Menge von Servern neu zu implementieren (sehr viel Arbeit, Kompatibilitätsprobleme), ein Mikrokern-UNIX auf L4 zu portieren (wenige Gemeinsamkeiten, also ebenso viel Arbeit) oder einen monolithischen UNIX-Kern als einzelnen Server im User-Mode auf L4 laufen zulassen. Man entschied sich für letzteren Punk, und die Wahl des Kerns fiel auf Linux, zumal die Open Software Foundation in Zusammenarbeit mit Apple (die auf der Suche nach einem Nachfolger für ihr veraltetes Mac OS waren) Linux bereits erfolgreich auf den Mach-Mikrokern übertragen hatten (“MkLinux”).

Mit einem Linux-Server auf L4 versprach man sich ein Betriebssystem mit zwei Personalities: Linux und ein Echtzeitsystem, die gleichzeitig auf einer Maschine laufen konnten.

4.2 L4Linux

Im Rahmen einer Diplomarbeit entwickelte Michael Hohmuth zusammen mit Mitarbeitern an der TU Dresden schließlich 1996 eine Portierung von Linux 2.0 auf L4. In den folgenden Abschnitten soll auf die Grundsätze des Designs, die konkrete Umsetzung sowie die erzielte Leistung eingegangen werden [**?**][9].

4.2.1 Ziele

Die Entwickler legten von Anfang an folgende Ziele fest:

- Linux soll vollständig im User-Mode laufen, weder L4 noch der architekturneutrale Teil des Linux-Kerns (z.B. Dateisysteme) sollen angepaßt werden.
- Das so entstehende System soll binärkompatibel zu bestehenden Linux-Anwendungen sein, d.h. man kann L4 und L4Linux als Ersatz für einen entsprechenden Kern einer beliebigen Distribution verwenden.
- Es soll bestätigt werden, daß L4 flexibel und schnell genug ist, um eine effiziente UNIX Personality darauf zu implementieren.

Bei vielen Designentscheidungen spielte deswegen die erwartete Performance eine große Rolle.

4.2.2 Umsetzung

Um die Umsetzung im Detail und insbesondere die Entscheidungen für die eine oder andere Konzeption bei der Umsetzung zu verstehen, wären sehr gute Kenntnisse des Linux-Kerns nötig. Die Beschreibung im folgenden beschränkt sich daher auf die Grundlagen und erklärt gegebenenfalls einige Linux-Konzepte grob.

4.2.2.1 Allgemeines Ziel war es, Linux komplett um User Mode laufen zu lassen. Dabei sollte Linux nicht in einzelne Server zerlegt werden, wie man es bei der Konstruktion eines neuen Systems auf einem Mikrokern machen würde, da dies zum einen einen erheblich höheren Aufwand bedeutet hätte, und es zum anderen quasi unmöglich gemacht hätte, zukünftige Änderungen am Standard-Linux-Kern in L4Linux einfließen zu lassen. Der Linux-Kern wurde also zu einem einzelnen Task im User-Mode. Jeder Unix-Prozeß wird durch einen L4-Task realisiert. Ein weiterer Task kümmert sich um das Paging: Dieser "Root-Pager" behandelt Seitenfehler des Linux-Servers, während der Linux-Server jedoch selbst den virtuellen Speicher seiner Prozesse verwalten kann.

4.2.2.2 Interrupts Interrupts werden im Standard-Linux wie folgt behandelt: Im Kern befinden sich für jeden zu behandelnden Interrupt zwei Routinen, eine "Top Half" und eine "Bottom Half". Die Top Half wird aufgerufen, sobald der Interrupt eintritt, sie bestätigt in der Regel den Interrupt und markiert eine Bottom Half zur Ausführung, die sich um das weitere Vorgehen kümmert. Top Halves sind Aktivitäten im Kern, die eine höhere Priorität haben als Bottom Halves. Im Linux-Server von L4Linux ist diese Funktionsweise wie folgt realisiert: Es existiert ein separater Thread im Linux-Server für jeden Interrupt, auf den gewartet wird. Dieser markiert dann entsprechend die Bottom Halves zur Ausführung. Ein weiterer Thread im Linux-Server kümmert sich um die Ausführung aller markierter Bottom Halves.

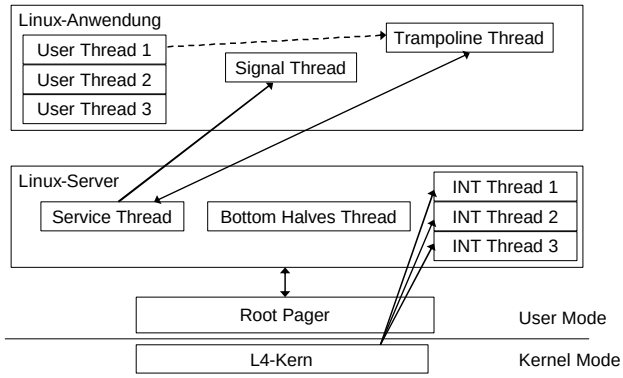
4.2.2.3 Systemaufrufe Ein dritter Thread im Linux-Server ist für die Systemaufrufe zuständig. Ein solcher Aufruf geschieht wie in L4 vorgegeben über RPC bzw. IPC. Da die Kompatibilität zu Standard-Linux bestehen bleiben mußte, war es nicht erlaubt, die C-Bibliothek so umzuschreiben, daß eine Anwendung per RPC mit dem Linux-Server kommuniziert; die "int 0x80"-Anweisungen, die eine Anwendung in normalem Linux für einen Systemaufruf verwendet, mußten auch unter L4Linux funktionieren. Dies ist aber kein Problem: Eine "int"-Anweisung in einer Anwendung führt zu einer Exception, die von einem separaten Thread pro Task behandelt werden kann. Dieser Thread überprüft dann im Code an der Stelle, an der die Exception aufgetreten ist, ob es ein "int 0x80" war, und wenn ja, kümmert er sich um den Aufruf des Linux-Servers per RPC und führt die Ausführung im Haupt-Thread seines Tasks fort. Diesen Trick nannten die Entwickler den "Trampolin-Mechanismus". Diese Lösung offensichtlich nicht optimal, ein direkter Aufruf des Linux-Servers über RPC wäre schneller. Die Entwickler realisierten auch diese Lösung durch eine Anpassung der C-Bibliothek: Dynamisch zur C-Bibliothek gelinkte Anwendungen konnten so beschleunigt werden, statisch gelinkte Anwendungen bzw. Anwendungen, die den "int 0x80"-Befehl direkt verwenden, funktionierten aber weiterhin.

4.2.2.4 Signale Das Zustellen von UNIX-Signalen an Linux-Prozesse geschieht ebenfalls über einen separaten Thread im Anwendungs-Task, dem vom Linux-Server Signale geschickt werden können. Dieser kann es dem Haupt-Thread seines Tasks dann entsprechend zustellen.

4.2.2.5 Scheduling Bei der Implementierung des Scheduling hatte man die Möglichkeit, entweder den Linux-Server das Scheduling übernehmen zu lassen, indem er die Strategie von L4 steuert ("thread_schedule"), oder einfach das Standard-Scheduling von L4 zu verwenden. Man entschied sich für die letztere Lösung, da zur Zeit der Entwicklung von L4Linux das externe Steuern des Scheduling noch nicht in L4/x86 implementiert war. Dies hat zur Folge, daß es auf L4Linux (noch) nicht möglich ist, Prioritäten für Prozesse zu vergeben, was aber in der Praxis kaum Auswirkungen hat.

4.2.2.6 Zusammenfassung Nach diesen Beschreibungen der einzelnen Mechanismen noch einmal eine Zusammenfassung des Konzepts: Es existieren zunächst immer zwei Tasks, der Root-Pager und der Linux-Server. Letzterer besteht immer aus dem "Service Thread", der Systemaufrufe entgegennimmt, einem Tread, der die Bottom Halves ausführt, sowie einen Thread für jeden Interrupt, den der Kern behandelt.

Für jeden Linux-Prozeß gibt es einen weiteren Task. Dieser besteht aus "User Threads", in denen der normale Programmcode ausgeführt wird, sowie aus einem "Signal Thread", der die Signale des Linux-Servers entgegennimmt und einem Thread für den Trampolin-Mechanismus. Die Abbildung stellt das Design von L4Linux noch einmal grafisch dar.



4.2.3 Leistung

Sobald L4Linux kompatibel genug war, um typische Anwendungen auszuführen, lag das Interesse der Entwickler daran, die Leistung von L4Linux (auf L4/x86) mit der von normalem Linux sowie von MkLinux, einem Linux-Server auf Mach, zu vergleichen. Dazu ist zunächst zu sagen, daß MkLinux in zwei Versionen existiert: MkLinux "user" realisiert die Linux-Funktionalität wie L4Linux mit einem Linux-Server im User-Mode. MkLinux "in-kernel" wurde auf Geschwindigkeit optimiert, indem der Linux-Server im Adreßraum von Mach lief - was die eingangs beschriebenen Nachteile mit sich bringt.

4.2.3.1 Microbenchmarks Im kleinen galt es zunächst, die benötigte Zeit für einen einfachen Linux-Systemaufruf ("getpid") zu messen: L4Linux brauchte unter Verwendung einer optimierten C-Bibliothek für diesen Systemaufruf ein wenig mehr als doppelt so lange als echtes Linux, mit der Trampolin-Methode etwas mehr als dreimal so lange. Im Vergleich zu MkLinux, welches 9mal (in-kernel) bzw. 66mal so lange brauchte, sind das jedoch äußerst gute Zahlen.

4.2.3.2 Macrobenchmarks Das einzelne Testen einiger typischer Operationen sah ähnlich schlecht für MkLinux und gut für L4Linux aus: Während MkLinux 120%-1200% (in-kernel) bzw. 120%-6450% (user) der Zeit brauchte, bewegte sich L4Linux im Bereich von 105%-250%.

Was im Endeffekt zählt, ist allerdings die letztendliche Geschwindigkeit bei echten Anwendungen. Das Kompilieren des Linux-Kerns benötigte auf L4Linux nur ca. 7% länger als auch echtem Linux, während MkLinux um 17% (in-kernel) bzw. 27% (user) langsamer war.

Weitere Tests bestätigten, daß die Verlangsamung durch L4Linux typischerweise zwischen 5% und 10% liegt. Die Entwickler bewerteten dies als eine Verlangsamung, mit der man durchaus leben kann; inbesondere auf Arbeitsplatz-Rechnern ist sie nicht zu bemerken.

4.3 Vorteil von DROPS + L4Linux

Natürlich stellt sich nun die Frage, welche Vorteile für den Endanwender es hätte, L4Linux anstelle von normalem Linux zu verwenden. Von den drei grundsätzlichen Vorteilen Stabilität, Modularität und Flexibilität profitiert man zunächst nicht; L4Linux verhält sich genau wie ein monolithisches Linux.

Es sind die neuen Möglichkeiten, die den Vorteil ausmachen: Es ist kein Problem, eine weitere Personality auf L4 zu implementieren, die dann parallel zu L4Linux läuft, wie etwa das Echtzeitsystem, das im Rahmen des DROPS-Projekts entwickelt wird. Auf eine solche Personality könnte auch von Linux aus zugegriffen werden, es wäre kein Problem, Echtzeit-Anwendungen zu realisieren, die als reguläres Linux-Programm auf der normalen Linux-Oberfläche (X-Window) laufen, aber die Echtzeitfähigkeiten der anderen Personality nutzen. Dies funktioniert an der TU Dresden bereits.

Etwas grundlegender wäre die Idee, den Linux-Kern nur als Kompatibilitätsebene zu verwenden und nach und nach Komponenten aus dem Linux-Server auszugliedern. Ein Hardware-Treiber, der als separater Task läuft, trägt natürlich zur Stabilität bei, ebenso kann man Netzwerkprotokolle oder Dateisysteme ausgliedern und zu separaten Servern machen. Verfolgt man diesen Ansatz konsequent, hat man früher oder später alle Vorteile eines Mikrokerns. Die TU Dresden geht sogar noch einen Schritt weiter: Dort wurde ein SCSI-Treiber implementiert, der als separater Thread auf der Echtzeit-Personality läuft und vom Linux-Server verwendet werden kann. Dieser Treiber kann eine gewisse Bandbreite an Daten garantieren (z.B. für Video-Streaming im Netz sehr sinnvoll), was ein Linux-Treiber grundsätzlich nicht kann. Wie man sieht, kann man auf allen Ebenen von den Vorteilen anderer Personalities profitieren.

Ein weiterer Vorteil hört sich zunächst ein wenig paradox an: L4Linux kann echtes Linux in Sachen Geschwindigkeit übertreffen. Die Entwickler von L4Linux haben dazu mit UNIX-Pipes experimentiert. Zunächst wurden die Verzägerung sowie der Durchsatz einer Pipe auf Linux und auf L4Linux gemessen. Dann wurde in einem separaten Server eine POSIX-konforme Pipe implementiert, welche die Funktionalität mit Hilfe von L4-Primitiven erreicht. Diese Methode unterbot die Verzögerung von echtem Linux knapp (22 μs statt 29 μs) und erhöhte die Bandbreite entscheidend (48-70 MB/s statt 41 MB/s). Insbesondere wenn man spezielle Formen der Kommunikation benötigt, kann L4 Dienste bieten, an deren Geschwindigkeit Linux nicht rankommt: Verwendet eine Anwendung etwa die synchrone IPC von L4 direkt, so kann die Verzögerung auf nur 5 μs gedrückt und die Bandbreite auf 65-105 MB/s gesteigert werden. Eine besonders hohe Bandbreite erreicht man, wenn ein Thread einem Thread eines anderen Tasks Zugriff auf seinen eigenen Speicher gibt, wie L4 das für Speichermanager vorgesehen hat. Die Bandbreite der Kommunikation über diesen gemeinsamen Speicherbereich wird praktisch nur vom Speicherdurchsatz des Rechners begrenzt.

Diese Beispiele zeigen, daß es durchaus Sinn machen kann, L4Linux in der Praxis sinzusetzen, sobald von den Vorteilen Gebrauch gemacht werden kann.

5 Weiterentwicklung, Zukunft

Die bisherigen Ausführungen haben gezeigt, daß das L4-Konzept zu einer deutlichen Geschwindigkeitssteigerung führt, daß mehrere Implementationen dieses Interfaces für verschiedene Prozessoren bereits reif für den praktischen

Einsatz sind, und daß momentan mehrere Systeme auf Basis von L4 realisiert werden.

Auch nach dem unerwarteten Tod von Jochen Liedtke Mitte 2001 im Alter von nur 48 Jahren bleibt die Forschung an dieser Stelle nicht stehen. Das liegt zum einen an den Erfahrungen bei der Implementierung der L4-Kerne bzw. der Personalities, und zum anderen an der fortschreitenden Prozessor-Entwicklung. Der praktische Einsatz von L4-Kernen hat beispeilsweise gezeigt, daß das Konzept der Clans & Chiefs für gewissen Anwendungen nicht flexibel genug ist, es wurde deshalb ein aufwärtskompatibles Konzept namens "Elphinstone-Jaeger Redirection Model" entworfen. Insbesondere die Entwicklung neuer 64-Bit-Prozessoren wie der IA-64-Linie oder des IBM Power-4 (viele Register, expliziter Parallelismus, 64-Bit-Speicher) erfordern sowohl Optimierungen des ABI für diese Prozessoren als auch eine komplette Neuentwicklung besonderer Algorithmen hierfür, damit L4 auch auf diesen Prozessoren eine hohe Geschwindigkeit erreichen kann. All diese Eigenschaften soll die Version 4 des L4-Kern-Interfaces, welches bereits spezifiziert ist, zusätzlich unterstützen. An der Universität von Karlsruhe soll dann ein L4-Kern der Version 4 unter dem Namen Pistachio entwickelt werden.

Aufgrund des hohen Implementierungsaufwandes in Assembler forscht man des weiteren daran, in wie weit die C++-Implementierung die Geschwindigkeit der Assembler-Implementierung erreichen kann. Sollte sich herausstellen, daß man in C++ prinzipiell zu weit von der Assembler-Performance entfernt ist, so plant man, zusätzlich einen L4-Kern der Version 4 in Assembler zu implementieren.

Wie man sieht, bleibt die Entwicklung nicht stehen. Die bald bevorstehende Einführung von 64-Bit-Prozessoren für Arbeitsplatz-Rechner macht das Bedürfnis effizienter Kerne für diese Architekturen besonders relavant. Und die Ergebnisse der Forschung an der Version 4 des L4-Interfaces sowie die des DROPS- und des SawMill-Projektes werden möglicherweise die Konstruktionsmethoden von Betriebssystemen revolutionieren.

Literatur

[1] Liedtke, J.: Improving IPC by kernel design. In *14th ACM Symposium on Operating System Principles (SOSP)*, Seiten 175-188, Asheville, NC, Dezember 1993

[2] Liedte, J.: On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, Seiten 237-250, Copper Mountail Rersort, CO, Dezember 1995.

[3] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., Wolter, J.: The performance of μ-kernel based systems. In *16th ACM Symposium on Operating System Principles (SOSP)*, Seiten 66-77, Saint-Malo, Frankreich, Oktober 1997

[4] Liedtke, J.: L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021.- Sankt Augustin: GMD, 1996

[5] Hohmuth, M.: Linux-Emulation auf einem Mikrokern. Diplomarbeit, TU Dresden, 1996

[6] Tanenbaum, A.S.: Modern Operating Systems.- New Jersey, Prentice Hall, 2001.

[7] Diedrich, Dr. Oliver: Linus skaliert nicht. In *c't 4/2002*, Seite 40.

[8] Inside Mac OS X: Kernel Environment.- Cupertino: Apple Computer, 2000.

[9] Dannowski, U. et al.: The L4Ka Vision.- Universität Karlsruhe, 2001