Portierung von Linux auf den μ-Kern L4

Martin Borriss* Michael Hohmuth Jean Wolter Hermann Härtig

Zusammenfassung

Im Kontext des DROPS (**D**resden **R**ealtime **Op**erating **S**ystem)-Projektes wurde der monolithische Linuxkern auf den μ-Kern L4 portiert.

Im Mittelpunkt dieses Beitrags steht die Erläuterung von Entwurfsentscheidungen der Portierung.

Leistungsmessungen, die im Anschluß an Optimierungen durchgeführt wurden, zeigen, daß den Vorteilen der μ-Kern-Architektur keine wesentlichen Leistungseinbußen gegenüberstehen.

1 Einleitung

Die Forschung an der TU Dresden, Lehrstuhl Betriebssysteme, konzentriert sich auf echtzeitfähige Betriebssysteme, die auf den μ-Kernen L3 und L4 [Lie95] basieren. Ziel ist die Entwicklung einer Multi-Server-Architektur, die Anwendungen mit Quality-Of-Service-Anforderungen unterstützt [Uni96].

Um mit dem μ-Kern L4[1] effektiv arbeiten und entwickeln zu können, soll eine Standard-Unix-API zur Verfügung stehen. Dazu sollte ein klassischer monolithischer Kern, Linux [Uni97], auf den μ-Kern L4 portiert werden.

Im einzelnen wurden folgende Ziele verfolgt:

- Portierung des Linux-Kerns auf den μ-Kern L4, so daß dieser als Anwendungsprogramm im Nutzermodus läuft.
- Vollständige Binärkompatibilität mit der Originalimplementation des Linuxkerns, die eine Modifikation bestehender Anwendungen unnötig macht.
- Aufzeigen der Effizienz der Kombination L4 und Linux, ohne dabei auf Funktionalität oder Sicherheit zu verzichten und ohne den μ-Kern zu ändern.
- Erster extensiver Praxistest des L4-μ-Kerns, um neue Konzepte zu testen und verbliebene Schwächen zu beseitigen.

2 Entwurf und Realisierung

Linux läuft auf den Prozessoren Intel x86, Motorola 68k, Digital Alpha, PowerPC und SPARC. Dabei existiert eine klare Trennung zwischen maschinenabhängiger und -unabhängiger Schicht. Eine Änderung des maschinenunabhängigen Teils sollte bei der vorgestellten Portierung möglichst vermieden werden[2]. Gleichfalls unverändert bleibt der L4-μ-Kern.

2.1 Entwurfsschwerpunkte

Beim Entwurf einer solchen Portierung gilt es generell zu beachten, wo potentielle Engpässe in der Systemleistung auftreten können. Insbesondere die Interaktion zwischen Nutzertasks und dem UNIX-Kern bestimmt die Struktur der Emulation. In klassischen UNIX-Systemen greifen Anwendungen nach einem Wechsel des

*Email: Martin.Borriss@inf.tu-dresden.de

[1] L4 ist ein μ-Kern der zweiten Generation, entwickelt von J.Liedtke an der GMD. Konstruiert nach dem Minimalitätsprinzip, zeichnet sich L4 durch extrem schnelle IPC aus.

[2] Der Linuxkern unterliegt einer äußerst schnellen Entwicklung. Durch die Beschränkung auf die Änderung des architekturabhängigen Teils können neue Kernversionen leicht in das Design eingepaßt werden. Gegenwärtig wird Linux 2.0.21 verwendet.

Ausführungsmodus auf Betriebssystemdienste zu [Vah96]. Im Gegensatz dazu werden in einer Emulation die Dienste des emulierten Systems durch Server bereitgestellt.

Eine geradlinige Herangehensweise legt nahe, Linux-Anwendungen auf μ-Kerntasks abzubilden, während die Dienste des Linuxkernes durch genau eine weitere Task erledigt werden.

Grundlegend ist der Datentransfer zwischen Linuxkern und Anwendung. Linux nutzt dazu den *copyin/copyout* Mechanismus[3].

2.2 Ein erster Entwurf

Um den *copyin/copyout* Mechanismus effizient umzusetzen, wurden – anstelle genau einer Linuxservertask systemweit – pro Nutzertask eine Linuxservertask vorgesehen. Dieser Ansatz vermeidet die Kosten des Ein- und Ausblendens verschiedener Nutzerbereiche in den Kernadreßraum, da jede Linuxservertask Zugriff auf den Adreßraum der zugehörigen Nutzertask hat.

Die zugrundeliegende Annahme ist, daß Interprozeßkommunikation und Kontextwechsel in L4 wenig aufwendig sind. Die entstandene Struktur (*Task-Paar-Modell*) ist in Abbildung 1 vorgestellt.

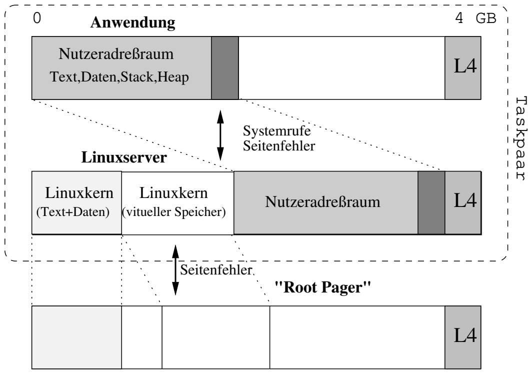

Abbildung 1: Erster Entwurf: Das *Task-Paar-Modell*.

[3]Um Daten zwischen Nutzer- und Kernbereich zu kopieren, muß der Nutzeradreßraum im Adreßraum der Kerntask verfügbar sein. In einem Design mit genau einer Linuxservertask müssen daher Nutzerbereiche ein- und ausgeblendet werden.

Nach der Realisierung dieses Entwurfs wurde festgestellt, daß folgende Probleme existieren, die die Systemleistung negativ beinflussen:

- Die Abbildung auf eine vergleichsweise große Anzahl L4-Tasks bedingt einen höheren Ressourcenverbrauch (z. B. Seitentabelleneinträge).
- Interprozeßkommunikation ist wesentlich langsamer, wenn die Kommunikationspartner in großen Adreßräumen residieren[4]. Zwischen Linuxservertask und Linuxnutzertask kann wegen der Nutzung großer Adreßräume nur eine „langsame“ Kommunikation stattfinden.
- Durch die quasiparallele Ausführung mehrerer Linuxservertasks ist zusätzliche Synchronisation auf Kernebene notwendig[5] [Loe93]. Das *Task-Paar-Modell* impliziert den Synchronisationsaufwand eines voll preemptiven Unixkerns.

  Dies erzeugt zusätzlichen Overhead und ist Quelle von schwer zu findenden Fehlern.

Die Durchführung von Benchmarks und die Analyse kritischer Pfade, die bei der Abarbeitung realer Applikationen auftreten, stimulierten ein alternatives Design. Dieses vermeidet die meisten der aufgeführten Probleme und wird im nachfolgenden Abschnitt beschrieben.

2.3 Verbesserter Entwurf

Da die Existenz multipler Instanzen des Linuxservers als Hauptursache für hohen Ressourcenverbrauch und Synchronisationsoverhead identifiziert wurde, beinhaltet das Neudesign eine Rückkehr zum natürlichen Ansatz

[4]Innerhalb kleiner Adreßräume kann L4 auf Pentium-Prozessoren wesentlich schneller IPC realisieren. Anstelle einer Umschaltung von Seitenverzeichnissen ist nur ein Umschalten eines Segmentregisters notwendig (Simulation eines *tagged TLB*) .

[5]Der Linuxkern ist durch ein binäres Semaphor geschützt. Im Kern schlafende Prozesse (zum Beispiel bei blockierenden I/O-Befehlen) werden zudem mittels „condition variables“ in Warteschlangen verwaltet.

mit genau einer Linuxservertask (siehe Abbildung 2).

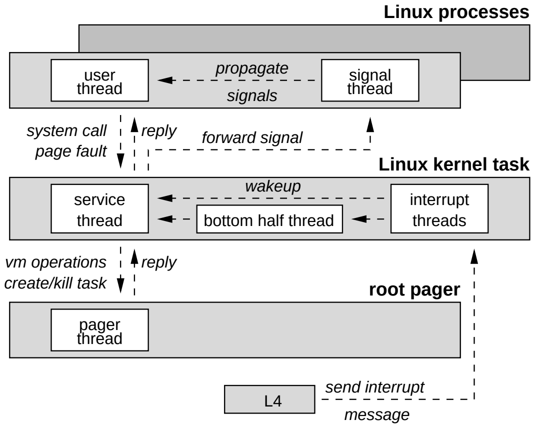

Abbildung 2: Linux auf L4 - Taskstruktur.

Die Speicherverwaltung in einem L4-basierten System erfolgt mit Hilfe externer Tasks („external pager"). Linux auf L4 verwaltet den Speicher hierarchisch: Eine „root pager"-Task löst Seitenfehler für die Linuxservertask auf, welche wiederum als Pager für Linuxnutzertasks dient.

Insgesamt konnte der architekturunabhängige Teil von Linux mit Ausnahme des Scheduling völlig unverändert gelassen werden. Die Anpassung des architekturabhängigen Teils umfaßte etwa 12000 Zeilen C-Quellkode.

Im folgenden sollen die Aspekte der Anpassung des Interruptsystems, der Systemrufschnittstelle, der Signalzustellung, des Scheduling und der Speicherverwaltung betrachtet werden.

2.3.1 Task- und Threadstruktur

Aus Sicht des μ-Kernes L4 hat das Linuxsubsystem nunmehr folgende Struktur:

1. Alle geforderten Linuxkerndienste werden durch einen einzigen Thread behandelt. Dessen Kontext kann umgeschaltet werden, um ihn auf verschiedene Linuxkernaktivitäten zu multiplexen.

2. Für jede Interruptquelle steht ein dedizierter Thread zur Verfügung (siehe Abschnitt 2.3.4).

3. Ein weiterer Thread steht für die Abarbeitung der zeitunkritischen Teile der Interruptbehandlungsroutinen zur Verfügung.

4. Linuxnutzertasks bestehen aus zwei Threads. Dabei arbeitet ein Thread die primäre Nutzeraktivität ab, während ein zweiter Thread für die Signalbehandlung (Abschnitt 2.3.3) erforderlich ist.

Eine zusätzliche Synchronisation ist unnötig. Wie im monolithischen Linux werden lediglich kurze kritische Abschnitte innerhalb des Kerns vor Interrupts geschützt.

2.3.2 Systemrufe

Systemrufe werden mittels eines *Trampolinmechanismus* an die Linuxservertask weitergeleitet. Dadurch wird eine vollständige Binärkompatibilität mit dem Original-Linux gewahrt[6].

Die C-Bibliothek bringt hierbei Systemrufnummer und eventuelle Parameter in Registern unter und löst einen Softwareinterrupt (*int 0x80h*) aus. Dieser Trap in den L4-Kern kann dort nicht behandelt werden, so daß eine Ausnahme[7] generiert und an die verursachende Task zurückgesendet wird. Eine Ausnahmebehandlungsroutine erkennt hier den Systemruf. Die Parameter des Systemrufes werden in einem von Linuxnutzertask und Linuxservertask gemeinsam genutzten Bereich abgelegt. Eine sich im Adreßraum der Nutzertask befindende Emulationsbibliothek gibt den Systemruf per IPC an die Linuxservertask weiter.

Die Linuxservertask besitzt eine Schnittstelle, zu der Systemrufanforderungen, Seitenfehler- und Ausnahme-Nachrichten per L4-IPC zugestellt werden. Zwischen den kommunizierenden Prozessen können weitere Informationen (Parameter, Identifikatoren, Statusinformationen) über einen gemeinsamen Speicherbereich ausgetauscht werden.

[6] Ein weiterer Leistungsgewinn könnte durch eine modifizierte C-Bibliothek erreicht werden, die den Systemruf direkt an den Linuxserver weiterleitet.

[7] Als Reaktion auf dieses Ereignis erzeugt der i386 die Ausnahme 13: *General Protection Exception*.

Kommunikation in L4 ist synchron; somit wartet der aufrufende Prozeß auf die Antwort des Linuxservers bei Beendigung des Systemrufes.

2.3.3 Signale

Signale werden benutzt, um Prozesse asynchron über Ereignisse zu informieren oder ihren Zustand zu ändern. Die Zustellung von Signalen erfolgt bei der Rückkehr vom Kern- in den Nutzermodus durch Manipulation des Stacks[8]. Im Original-Linux wird durch regelmäßigen Aufruf der `schedule()`-Funktion durch den Timerinterrupt eine geringe Latenz bei der Signalzustellung erreicht.

Um Gleiches für Linux auf L4 zu gewährleisten, wird ein *Fake-Interrupt-Mechanismus* benutzt: Ein dedizierter Thread innerhalb der Nutzertask wird über das zuzustellende Signal benachrichtigt und erzwingt den Kerneintritt des sich im Nutzermodus befindenden Nutzerthreads[9]. Bei der Rückkehr von diesem *Null-Systemruf* werden fällige Signale zugestellt.

2.3.4 Interrupts

L4 sieht vor, daß die Behandlung von Hardwareinterrupts auf Nutzerebene erfolgt. Bei Eintreffen eines Interrupts wird ein Thread per IPC benachrichtigt, welcher die Interruptbehandlungsroutine implementiert. Interruptbehandlungsroutinen unter Linux sind zweigeteilt in *Top Half* und *Bottom Half*:

Top Halves. Diese werden sofort nach Eintreffen des Interrupts abgearbeitet. Sie sind nur durch andere Top Halves unterbrechbar. Oft bestätigen diese lediglich den Interrupt und markieren eine oder mehrere Bottom Halves zur Ausführung. Für jede Interruptquelle existiert in unserem Entwurf ein dedizierter Thread.

[8]Die Adressen der entsprechenden Signalbehandlungsroutinen werden auf dem Stack abgelegt, so daß der Prozeß zunächst diese ausführt.

[9]Unkooperative Tasks – die etwa den Signalthread beenden – sind aufgrund der besonderen Semantik der Unix-Signale SIGKILL und SIGSTOP dennoch beherrschbar.

Bottom Halves. Die zeitunkritischen Teile der Interruptbehandlungsroutinen werden durch *einen* Thread ausgeführt.

Top-Half-Threads besitzen eine höhere statische Priorität als der Bottom-Half-Thread und der Linuxserverthread. Damit wird implizit die Linux-Semantik bezüglich Verdrängbarkeit von Interruptbehandlungsroutinen erzwungen.

2.3.5 Scheduling

Linux realisiert in der `schedule()`-Funktion ein dynamisches Scheduling. `schedule()` kann bei der Abarbeitung von blockierenden Systemrufen direkt aufgerufen werden, zumindest aber nach Ablauf der Zeitscheibe von 10ms ist der Aufruf durch den Timerinterrupt garantiert.

Folgende Entwurfsalternativen ergeben sich hier:

1. Durchsetzen der Schedulingstrategie von Linux. Dafür ist in L4 ein *Preemption Handler*-Mechanismus vorgesehen, mit dem beliebige Schedulingstrategien auf Nutzerebene realisiert werden können. Zum gegenwärtigen Zeitpunkt ist dieser Mechanismus jedoch noch nicht einsatzbereit.

2. Scheduling auf μ-Kernebene. Während Schedulingentscheidungen durch L4 getroffen werden, beschränkt sich die `schedule()`-Funktion auf das Multiplexen des Linuxserverthreads auf die verschiedenen Aktiväten, die aus Systemrufen resultieren.

Implementiert wurde die zweite Variante. Dabei wurden verschiedene Strategien für den Linuxscheduler getestet [HHL+97]. Eine Strategie, bei der nach Abarbeitung eines Systemdienstes der zugehörige Nutzerprozeß aktiviert wird, erweist sich als befriedigend und entspricht dem Verhalten des monolithischen Linux.

3 Leistungsmessungen

Um zu zeigen, daß μ-Kern-basierte Systeme durchaus effizient implementierbar sind, wurde der Optimierung kritischer Pfade und der Leistungsmessung großes Augenmerk geschenkt [HHL+97].

Sowohl Mikro-Benchmarks wie *lmbench* [MC96] als auch Applikations-Benchmarks wie *AIM* [Tec] wurden verwendet, um Original-Linux, Linux-L4 und MK-Linux[10] [dPSR96] hinsichtlich der Leistung zu vergleichen.

3.1 Optimierung kritischer Pfade

Die Analyse des Laufzeitverhaltens und daraus abgeleitete Verbesserungen des Entwurfs sind die wesentliche Ursache für das zufriedenstellende Laufzeitverhalten der Linuxemulation. Im einzelnen:

- **Allgemeine Optimierungen.** Hierzu zählen die *inline*-Generierung von Funktionsaufrufen, Nutzung der vom Compiler angebotenen Optimierungsstufen, „Handtuning" kritischer Codepassagen und Ausnutzen des multiskalaren Designs des Pentium-Prozessors. Insbesondere wenn die projektierte von der gemessenen Laufzeit abwich, wurde nach Cache- und TLB-Effekten gesucht.

- **Systemrufpfad.** Die Abarbeitung von Systemrufen ist aufgrund des aufwendigen Trampolinmechanismus (vgl. Abschnitt 2.3.2) ein Punkt, bei welchem Performanceeinbußen unvermeidlich waren. Die Analyse der zunächst schlechten Zahlen führte zum Neuentwurf der Emulation und zu Experimenten mit der Schedulingstrategie (vgl. Abschnitt 2.3.5). Weiterhin wurden die Ausrichtung von Datenstrukturen (*alignment*) verbessert und Datenzugriffe minimiert.

[10] MK-Linux ist eine Mach-basierte Linuxemulation, entwickelt von der OSF. Der Linuxserver ist hierbei „colocated", d. h., er läuft im Adreßraum des μ-Kerns.

3.2 Ergebnisse der Leistungsmessungen

Die Ergebnisse des `getpid()`-Systemrufes sind in Tabelle 1 dargestellt[11].

System	Time	Cycles
Linux monolithisch	2 μs	223
Linux L4	5 μs	753
MK-Linux (colocated)	15 μs	2050

Tabelle 1: `getpid()`-Systemruf

Als weiteres Experiment diente die Zeit, die für das Übersetzen des Linuxkerns benötigt wird. Linux benötigte 11:00 min, Linux-L4 11:48 min.

Messungen mit dem Applikationsbenchmark AIM [Tec] resultierten bei höchstmöglicher Last in einem Overhead von 19.5% von Linux-L4.

Der gemessene Overhead auf Applikationsebene von 7-20% ist – obwohl wesentlich niedriger als bei vergleichbaren Projekten – signifikant [HHL+97]. Zukünftige Optimierungen könnten die Speicherverwaltung betreffen.

4 Bewertung und Ausblick

Die entstandene Linuxportierung ist binärkompatibel zu Linux für i386. Benchmarks und der praktische Einsatz haben gezeigt, daß keine wesentlichen Einbußen hinsichtlich Performance mit der Nutzung von Linux auf dem L4-μ-Kern verbunden sind.

Demgegenüber stehen Vorteile wie erhöhte Modularität, hohe Sicherheit und – ganz wesentlich – die Möglichkeit eines Multi-Server-Designs zur Unterstützung von Anwendungen mit QoS-Anforderungen.

[11] Für alle Messungen in diesem Abschnitt wurde ein Pentium PC (P133, 64MB, 256kB PB Cache) verwendet.

Literatur

[dPSR96] F. B. des Places, N. Stephen, and F. D. Reynolds. Linux on the OSF Mach3 microkernel. In *FSF sponsored Conference on Freely Distributed Software*, 1996.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel based systems. In *ACM Symposium On Operating System Principles*, 1997.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *ACM Symposium On Operating System Principles*, 1995.

[Loe93] Keith Loepere, editor. *OSF Mach Server Writer's Guide, Revision 4*. OSF Mach series. Open Software Foundation, 1993.

[MC96] L. McVoy and C.Staelin. Lmbench: Performance tools for performance analysis. In *USENIX Annual Technical Conference*, 1996.

[Tec] AIM Technology. *AIM Multiuser Benchmark Suite VII.*

[Uni96] Dresden University. DROPS – Dresden Realtime Operating Frontpage. http://os.inf.tu-dresden.de/project/frontpage.html, 1996.

[Uni97] Helsinki University. Linux Operating System. http://www.cs.helsinki.fi/linux/, 1997.

[Vah96] Uresh Vahalia. *UNIX internals : the new frontiers.* Prentice Hall, 1996.

Autoren

Dipl.-Inf. Martin Borriss
Prof.Dr.rer.nat. Hermann Härtig
Dipl.-Inf. Michael Hohmuth
Dipl.-Inf. Jean Wolter

Technische Universität Dresden
Fakultät Informatik
Tel. +49-351-463 8401
Fax. +49-351-463 8284
Email: {borriss,haertig,hohmuth,jw5}@os.inf.tu-dresden.de