L4
Mikrokernel der 2. Generation

Vortrag zum Seminar „Konzepte von Betriebssystemkomponenten“

Alexander Würstlein
snalwuer@cip.informatik.uni-erlangen.de

Friedrich-Alexander-Universität Erlangen-Nürnberg

24. Juli 2006

Inhalt

Kapitelübersicht

Allgemeines zu L4

- Mikrokernel der 2. Generation
- soll vor allem die Performanceprobleme der ersten Generation beseitigen
- entworfen und implementiert von Jochen Liedtke
- mehrere Implementierungen mit kompatibler API verfügbar

Kapitelübersicht

Was ist ein Mikrokernel?

Definition

Ein Mikrokernel ist ein Kernel, der nur aus solchen Subsystemen besteht, die für die gewünschten Funktionen unbedingt Teil des Kernels sein müssen.

Anforderungen

- Interaktivität: Mehrere Benutzerprozesse, Interrupts
- Unabhaengigkeit: Subsysteme können unabhängig voneinander Garantien einhalten.
- Integrität: Ein Subsystem kann sich auf die Garantien eines anderen verlassen; zwei Subsysteme müssen sicher kommunizieren können.
- Die Hardware benutzt seitenbasierte Speicherverwaltung.
- Der Kernel stellt dieselben Abstraktionen auf allen Architekturen zur Verfügung.

Adressraumverwaltung

Ist Teil des Kernels weil...

... Trennung der Speicherbereiche sonst nicht implementiert werden könnte.

... Adressraumverwaltung privilegierte Operationen erfordert.

- besteht aus 3 grundlegenden Operationen, `grant`, `map` und `flush`
- eventuell auch Zugriffsrechte (read/write)
- Auslagerung („paging“) und Speicherverwaltung sind ausserhalb des Kernels implementiert.
- E/A-Ports und „memory mapped I/O“ können ebenfalls ausserhalb des Kernels behandelt werden.

Threads

- Threads sind Aktivitäten, die innerhalb eines Adressraumes ausgeführt werden.
- Sie werden charakterisiert durch:
  - Registerinhalte (vor allem IP und SP)
  - Speicherbereich

Im Kernel weil...

... gewünschte Trennung der Speicherbereiche sonst nicht realisiert werden kann. Jede Änderung im Speicherbereich eines Threads muss vom Kernel kontrolliert werden können.

IPC

Im Kernel weil...

... bei der Kommunikation zwischen Threads Adressgrenzen überschreitend Daten ausgetauscht werden müssen.

- klassisch durch Austausch von Nachrichten
- andere Verfahren wie Prozeduraufrufe können durch Nachrichtenaustausch nachgebildet werden.
- Interrupts werden auf Nachrichten abgebildet.

„unique identifier“

- Threads müssen Kommunikationspartner kennen
- entweder Threads oder Kommunikationskanäle muessen UIDs haben

Im Kernel weil...

... nur der Kernel Integrität sicherstellen kann

Nicht im Kernel

- Speichermanagment: Server ausserhalb des Kernels verteilt Speicher.
- Auslagerungs-Server
- Gerätetreiber: Hardwarezugriffe über ge-`map`-ten Speicher und Interrupt-Nachrichten.
- Kommunikation über Rechnergrenzen: Protokolle können über IPC benutzt werden und ihrerseits Gerätetreiber nutzen.
- UNIX-Server: Systemaufrufe können emuliert werden.

Kapitelübersicht

Kontextwechsel

- „widely believed“: Adressraumwechsel und Threadwechsel sind prinzipiell teuer.
- Wird scheinbar durch Messungen bestätigt.
- Wechsel zerlegbar in 2 Vorgänge:
  - Wechsel zwischen Kernel und User
  - Adressraumwechsel

Kernel-User-Wechsel

Wie funktionierts?

- Stack wechseln
- in privilegierten Modus des Prozessors wechseln
- eigentlich nur ein spezieller Funktionsaufruf

Was kostets?

- Messung per `getpid` auf einem 50 MHz486er
- Mach: 18 μs$\hat{=}$900 Takte, davon 800 Takte overhead
- L3[1]: maximal bis 180 Takte, minimal 123, davon 15 bzw. 57 overhead
- Ursache: Andere Implementierung

[1] Vorgänger zu L4

kurzer Ausflug: TLB

Definition

Der Translation Lookaside Buffer (TLB) ist ein Bestandteil der MMU, der die Zuordnung von logischen zu physikalischen Adressen cached. Ein tagged TLB speichert zusaetzlich zu jedem Paar aus logischer und physikalischer Adresse eine ID fuer den Adressraum.

- Bei einem untagged TLB wird der Inhalt des TLB bei einem Adressraumwechsel ungültig („TLB flush“).
- Wird bei einem Speicherzugriff die Adresse nicht im TLB gefunden („TLB miss“), muss in der Seitentabelle nachgesehen werden.
- Die Kosten dafuer sind hoch.
- Die Kosten fallen je nach „working set“ eines Threads mehrmals an.

Adressraumwechsel

- Kosten und Verfahren sehr stark architekturabhängig
- Kosten ergeben sich aus Wechsel der Segment- oder Seitentabelle und TLB flush
- bei tagged-TLB-Architekturen sehr billig weil TLB flush wegfällt
- bei Architekturen ohne tagged TLB im schlimmsten Fall einige hundert Takte overhead
- durch geschickte Wahl des Verfahrens sind die Kosten aber minimierbar

Threadwechsel

- im Beispiel RPC, bei dem 1 byte verschickt wird
- Kosten setzen sich aus Adressraumwechsel, system call, Parameterübergabe und Stackwechsel zusammen
- schnell genug um auch Interrupts zu bearbeiten

Messwerte

System	CPU, MHz	RPC time (round trip)	cycles/IPC (oneway)
full IPC semantics			
L3	486, 50	10 μs	250
QNX	486, 33	76 μs	1254
Mach	R2000, 16.7	190 μs	1584
SRC RPC	CVAX, 12.5	464 μs	2900
Mach	486, 50	230 μs	5750
Amoeba	68020, 15	800 μs	6000
Spin	Alpha 21064, 133	102 μs	6783
Mach	Alpha 21064, 133	104 μs	6916
restricted IPC semantics			
Exo-tlrpc	R2000, 16.7	6 μs	53
Spring	SparcV8, 40	11 μs	220
DP-Mach	486, 66	16 μs	528
LRPC	CVAX, 12.5	157 μs	981

Table 2: *1-byte-RPC performance*

Speichereffekte

- memory cycle overhead per instruction (MCPI)
- bei Mach laut Messungen bis zu 0.25 Takte pro Prozessorbefehl
- Ursache sind gegenseitige Verdrängungen aus dem Cache zwischen Kernel und Anwendungen
- als Lösung wird eine Verkleinerung des „cache working set“ angegeben

Messwerte

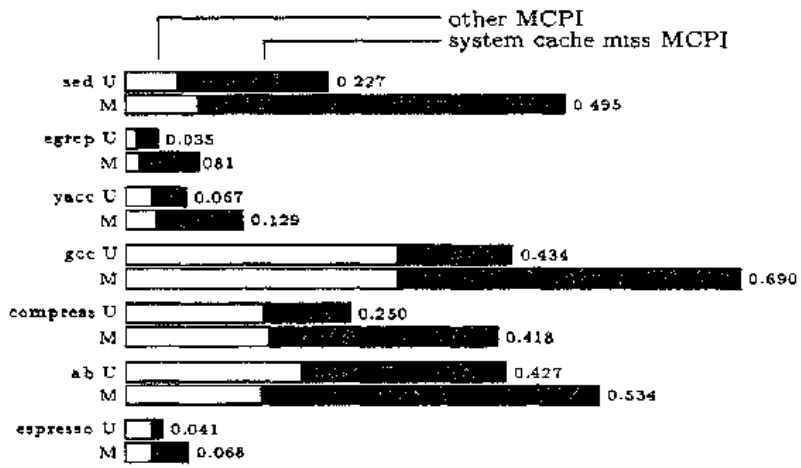


Figure 3: *Baseline MCPI for Ultrix and Mach.*

Portabilität von Mikrokerneln

Probleme

- früher: Mikrokernel hatten eine Abstraktion für die Hardware
- Abstraktion erzeugt overhead und verhindert spezielle Optimierungen
- speziell Adressraumwechsel sollten für die Architektur optimiert sein
- selbst zwischen 486 und Pentium bestehen signifikante Unterschiede

Portabilität von Mikrokerneln

Auswirkungen

- anderer Prozessor bedingt anderes Design
- Mikrokernel sind nicht portabel
- aber ein prozessorspezifischer Mikrokernel kann Basis fuer ein portables Betriebssystem sein

Kapitelübersicht

andere Ansätze

andere Ansätze

- Synthesis: „Compiler“ im Kernel integriert, der bei Systemaufrufen zur Laufzeit optimierten Code für diesen speziellen Systemaufruf mit diesen speziellen Parametern generiert
- Spin: Erweiterung dazu, ermöglicht vom Benutzer geschriebene zur Laufzeit übersetzte Systemaufrufe
- Exokernel: auf die R2000-Architektur zugeschnitten, stellt aber keine Abstraktion bereit sondern nur architekturabhängige Bausteine („primitives“)

Zusammenfassung

Zusammenfassung

- Mikrokernel bieten nur einen minimalen Satz an Abstraktionen
- Für unsere angegebenen Anforderungen: Adressräume, Threads, IPC, UIDs
- Das Design muss genau auf die Eigenheiten der Architektur rücksicht nehmen

aus der Presse

auf slashdot.org

OSTG | SourceForge | ThinkGeek | ITMJ | Linux.com | NewsForge | freshmeat | Newsletters | Jobs | Broadband | Whitepapers | X

Slashdot DON'T FEAR THE PENGUINS

▶ Login | Create Account | Subscribe

▼ Sections
Main
Apple
AskSlashdot
Books
Developers
Games
Hardware
Interviews
IT
Linux
Politics
Science
YRO
▼ Vendors
AMD
▼ Help
FAQ

Virtualized Linux Faster Than Native?

Posted by ScuttleMonkey on Wed May 31, '06 07:31 AM
from the **too-specialized-to-count** dept.

^switch writes

> "Aussies at NICTA have developed a para-virtualized Linux called Wombat that they claim outperforms native Linux. From the article: 'The L4 Microkernel works with its own open source operating system Iguana, which is specifically designed as a base for use in embedded systems.'"

Specific performance results are also available from the NICTA website.

Quellen

J. Liedtke
On μ-Kernel Construction.
ACM Operating Systems Review, Dezember 1995, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95).

H. Hartig et. al.
The Performance of μ-Kernel-Based Systems.
ACM Operating Systems Review, Dezember 1996, Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '96).

L4
L^{4}Linux, DROPS und Benchmarks

Vortrag zum Seminar „Konzepte von Betriebssystemkomponenten“

Alexander Würstlein
arw@arw.name

Friedrich-Alexander-Universität Erlangen-Nürnberg

24. Juli 2006

Inhalt

Kapitelübersicht

Allgemeines zu L4Linux

- Modifiziertes Linux als Prozess in L4
- Paravirtualisierung: Hardwarezugriff über modifizierte Treiber, die Mechanismen des Wirtssystems nutzen
- Entwickelt für die Benutzung in „DROPS“

Implementierung von L4Linux

- Änderungen an Linux nur im architekturabhängigen Teil
  - Interrupthandler
  - Speicherverwaltung
- L4 unverändert
- Linux-Gerätetreiber unverändert
- Linux-Benutzerprozesse sind einzelne L4-Prozesse
- Syscalls werden auf L4-IPC abgebildet
- Scheduling durch L4
- Binärkompatibel zu x86 Linux

Kapitelübersicht

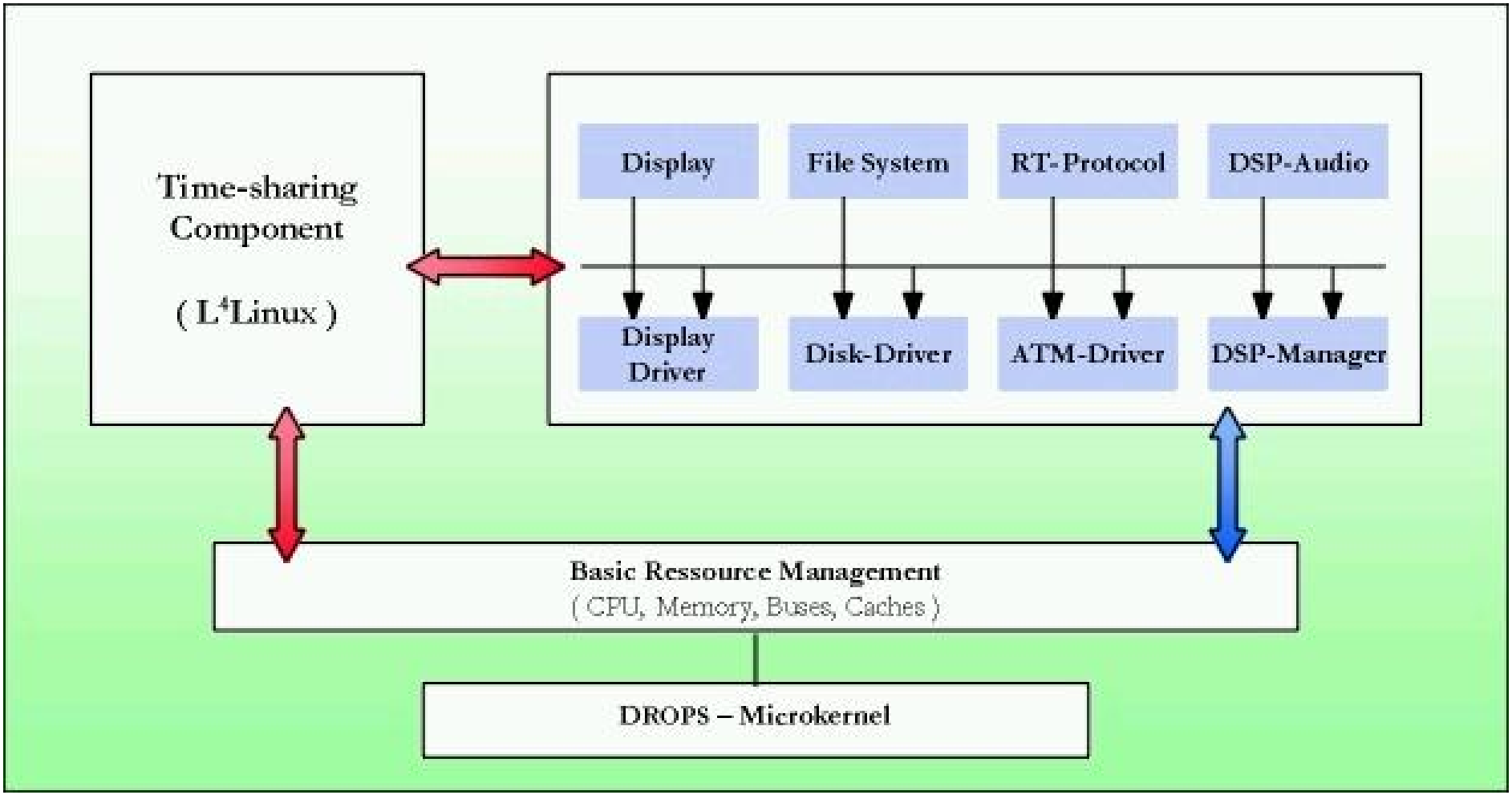
DROPS

- Echtzeitsystem zusammen mit benutzerinteraktivem „time-sharing“ L^{4}Linux
- Aufbauend auf L4 und L^{4}Linux
- Managment von Systemresourcen

Akronym

Dresden
Real-time
Operating
System Project

DROPS


Time-sharing Component
(L⁴Linux)
Display
File System
RT-Protocol
DSP-Audio
Display Driver
Disk-Driver
ATM-Driver
DSP-Manager
Basic Ressource Management
(CPU, Memory, Buses, Caches)
DROPS – Microkernel

Implementierung von DROPS

- Resourcentrennung für CPU, Speicher, Netzwerk, SCSI, Cache uvm.
- Resourcentrennung erfordert Gerätetreiber in L4
- Treiber in Linux durch Stubs ersetzt
- Managmentprozess in L4 erhält beim Booten alle Resourcen
- Dieser delegiert an Echtzeitprozesse und L^4Linux
- gezähmtes („tamed“) L^4Linux

Kapitelübersicht

Benchmarks

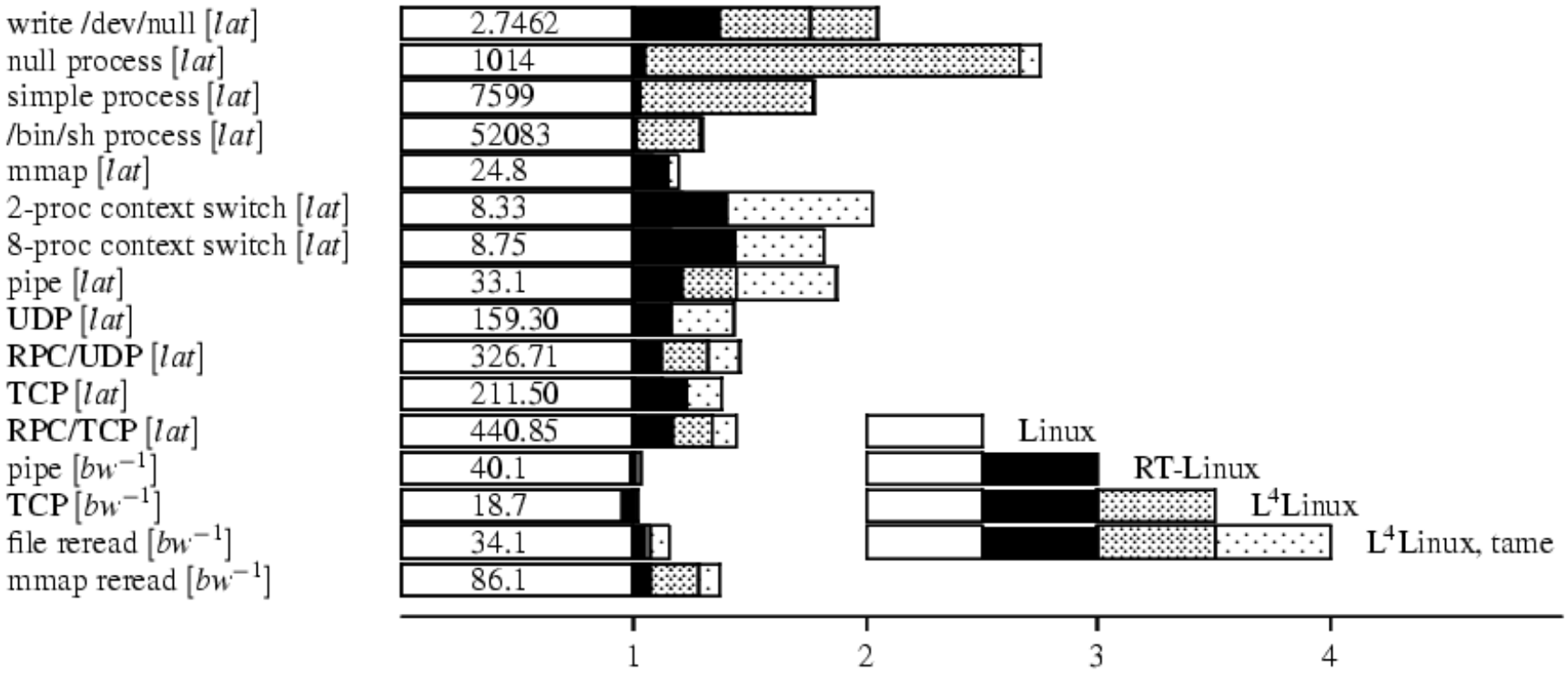
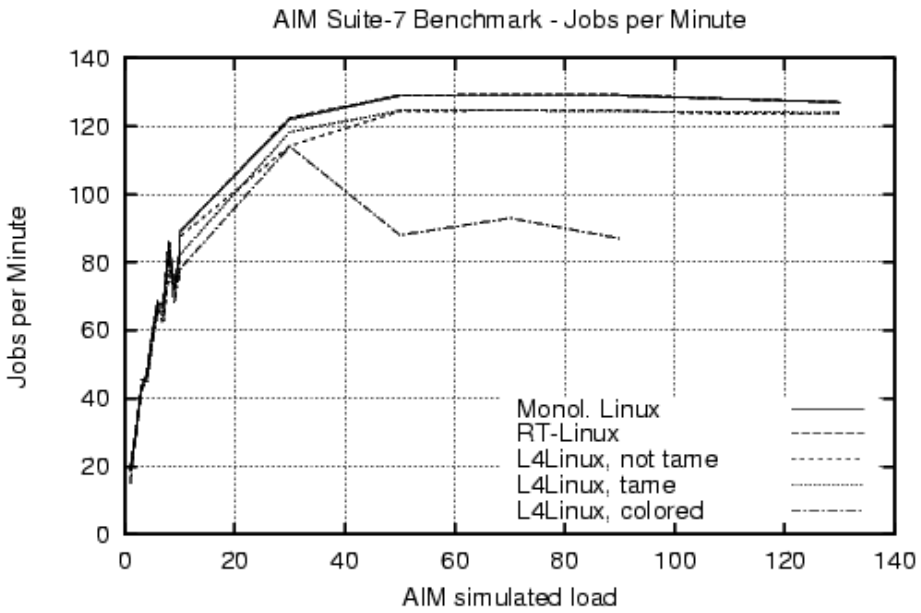


Fig. 4. *hbench:OS* results, normalized to native Linux. These are presented as slowdowns: A shorter bar is a better result. $[lat]$ is a latency measurement, $[bw^{-1}]$ the inverse of a bandwidth one. The numbers in the white boxes show the absolute values for native Linux in microseconds (for latencies) and in MB/s (for bandwidths), respectively. Hardware is a 133 MHz Pentium.

Benchmarks



kernel	relative average throughput	maximum throughput [jobs per minute]
monolithic Linux	100 %	129.6
RT-Linux	99.8 %	129.3
L⁴Linux	96.2 %	124.7
L⁴Linux, tame	96.1 %	123.7
L⁴Linux, tame and with colored cache	80.7 %	114.8

Fig. 5. AIM Multiuser Benchmark Suite VII. Left: Jobs completed per minute depending on AIM load units. Right: Average throughput (normalized to monolithic Linux) and maximum throughput of the various Linux kernels.

Benchmarks

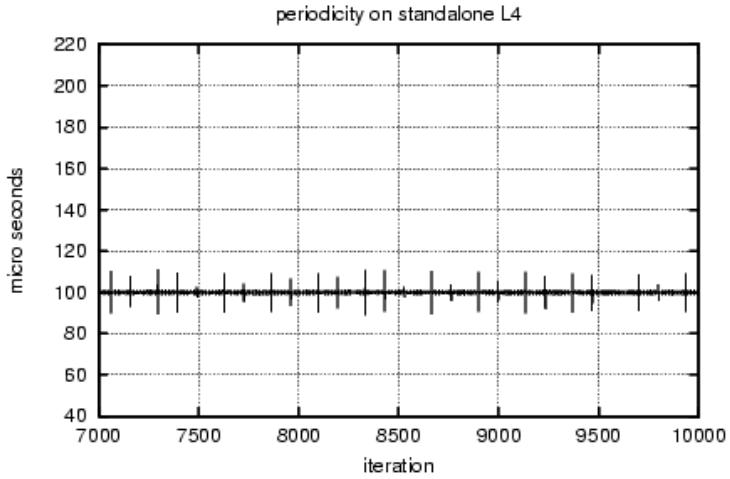


Fig. 6. Periodicity of a real-time task on stand-alone L4

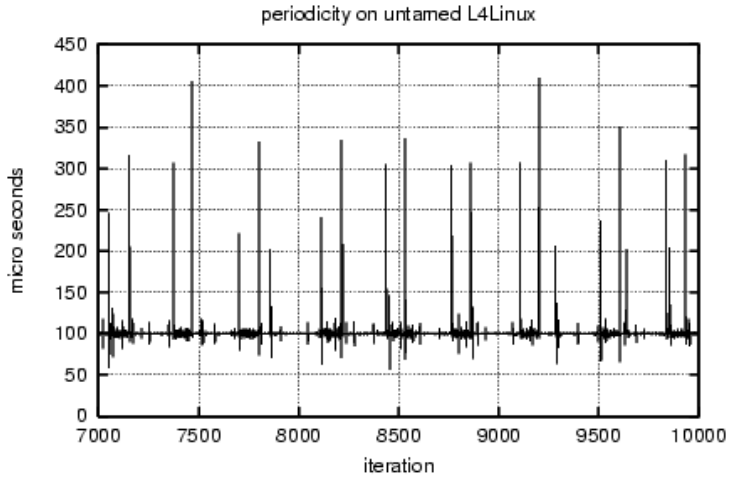
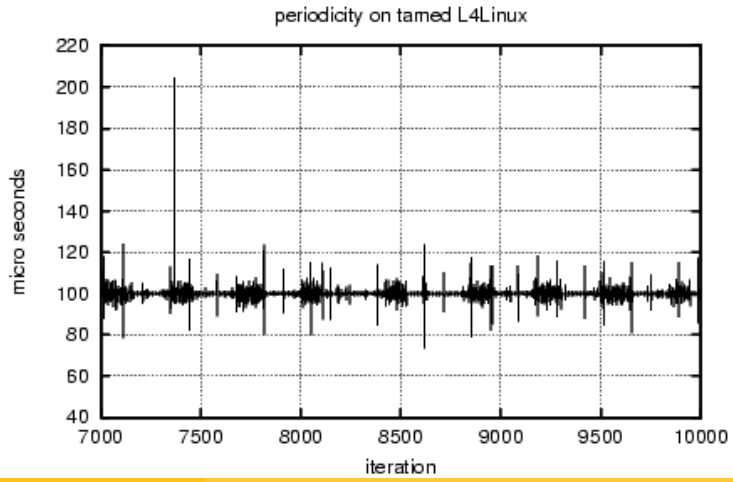


Fig. 7. Periodicity of a real-time task running besides original L^4Linux



Kapitelübersicht

Zusammenfassung

- L4 deutlich schneller als Mach
- L^{4}Linuxmit etwa 5 % Einbußen gegenüber nativem Linux
- Aber bei speziellen Operationen oft nur halb so schnell
- Echtzeitsystem koexistiert mit L^{4}Linux

Quellen

J. Liedtke
On μ-Kernel Construction.
ACM Operating Systems Review, Dezember 1995, Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95).

H. Härtig et. al.
The Performance of μ-Kernel-Based Systems.
ACM Operating Systems Review, Dezember 1996, Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '96).

Quellen


H. Härtig, M. Hohmuth, J. Wolter
Taming Linux.
Proceedings of PART '98: The 5th Australasian Conference on Parallel and Real-Time Systems, September 1998, Springer.


div. Autoren
`http://os.inf.tu-dresden.de/`