L4
Mikrokernel der 2. Generation

Ausarbeitung zum Seminar „Konzepte von Betriebssystemkomponenten“

Alexander Würstlein
arw@arw.name

24. Juli 2006

1 Einleitung

Mikrokernel der ersten Generation wie beispielsweise Mach haben vor allem mit Performanceproblemen zu kämpfen. Während Kritiker des Konzepts Mikrokernel dies auf eine prinzipielle, nichtbehebbare Schwäche von Mikrokernen zurückführen, wurde L4 zu dem Zweck entwickelt, zu zeigen, daß die bestehenden Probleme der ersten Generation vor allem auf die schlechte Implementierung und einige Designfehler der speziellen Kernel zurückzuführen sind.

Prinzipiell bezeichnet L4 vor allem eine API, die von verschiedenen Implementierungen benutzt wird. Diese API wie auch die erste Implementierung „L4/x86“ wurden von Jochen Liedtke als Nachfolger von L3 geschaffen.

Implementierungen von L4 existieren für die Architekturen x86, ARM, Alpha und R4000; praktische Verwendung findet L4 beispielsweise in Projekten wie L4Linux, wo ein oder mehrere Linux-Kernel paravirtualisiert als Threads unter einem L4-Kernel laufen oder Hurd-L4, ein Versuch das Betriebssystem[1] Hurd auf L4 zu portieren.

2 Mikrokernel

Definition (Mikrokernel). Ein Mikrokernel ist ein Kernel, der nur aus solchen Komponenten besteht, die für die gewünschte Funktionalität unbedingt Teil des Kernels sein müssen.

Diese Definition ist die beim Entwurf von L4 zugrundegelegte, natürlich gibt es abweichende Ansichten. Aus der Definition ergibt sich die Notwendigkeit für alle einzelnen im Kernel enthaltenen Teilsysteme zu diskutieren, ob diese jeweils im Sinne der Definition enthalten sein dürfen. Ausserdem ist es natürlich notwendig, sich über die Anforderungen im Klaren zu sein.

[1] Hurd ist eigentlich eine Sammlung von Serverprozessen, die zusammen mit einem Mikrokernel die Funktionalität eines traditionellen Unix-Makrokernels bereitstellen

2.1 Anforderungen

L4 soll als Basis für ein traditionelles Betriebssystem dienen, welches auf gebräuchlichen Rechnerarchitekturen wie etwa x86 laufen soll. L4 stellt wie eingangs erwähnt insbesondere eine API dar, diese soll natürlich auf allen Architekturen gleich angeboten werden. Es wird angenommen, daß alle diese Architekturen eine seitenbasierte Speicherverwaltung benutzen.

Es soll interaktiver Betrieb möglich sein und einzelne Programme sollen einander nicht „stören“ können. Daraus ergibt sich, daß mehrere Benutzerprozesse möglich sein sollen, die von Benutzereingaben und ähnlichem durch Interrupts erfahren.

Die „Störungsfreiheit“ verlangt etwas eingehendere Erklärung: Zuerst sollte ein Fehlverhalten[2] eines Prozesses keine Auswirkungen auf das Funktionieren anderer Prozesse haben. Unabhängigkeit ist die Anforderung, daß Prozesse unabhängig voneinander ihre Aufgaben erfüllen können, also Garantien die sie gegeben haben einhalten können.

Aber es besteht noch eine weitergehende Anforderung, nämlich die der Integrität: Es muss zwei Prozessen möglich sein, sich auf die Garantien des jeweils anderen zu verlassen. Der Unterschied zur vorhergehenden Anforderung ist, daß dazu der Austausch der Garantie, also die Kommunikation zwischen den Prozessen nicht durch Dritte verfälscht oder belauscht [3] werden kann.

2.2 Enthaltene Teilsysteme

2.2.1 Adressraumverwaltung

Die Adressraumverwaltung ist Bestandteil des Kernels, da ansonsten eine Trennung der Speicherbereiche nicht möglich wäre. Auf den genannten Architekturen erfordert die Adressraumverwaltung privilegierte Operationen, deren Ausführung nicht an ein ausserhalb des Kernels befindliches System abgegeben werden kann, ohne dieses System damit „zum Kernel zu befördern“, was den Forderungen nach Integrität und Unabhängigkeit zuwiderlaufen würde.

Ähnlich wie in Mach existieren drei grundlegende Operationen, `grant`, `map` und `flush`. Diese sind ausreichend, um weitere Aufgaben wie Speicherverwaltung und Paging an Prozesse ausserhalb des Kernels auszulagern. Ziel ist es hierbei, eine Hierarchie von Speicherverwaltern schaffen zu können, wobei der erste solche Prozess vom Kernel den kompletten nicht-Kernel-Speicher delegiert bekommt, welchen er dann mit denselben Operationen weiter unterteilen und delegieren kann.

Kommunikation mit der Hardware erfolgt oft aus Geschwindigkeitsgründen über Speicherbereiche[4], aus denen beispielsweise eine Netzwerkkarte ein zu sendendes Packet liest und ein empfangenes ablegt. Die Verwaltung und Nutzung dieser Speicherbereiche muss ebenfalls nicht im Kernel erfolgen, es genügt, diese an entsprechende Prozesse delegieren zu können.

[2] z.B. Zugriff auf eine ungültige Adresse im Speicher

[3] dies ist nötig für Garantien wie: „der folgende Schlüssel ist zufällig und geheim“

[4] Ports, memory mapped I/O

2.3 Prozesse / Threads

Prozesse[5] sind Aktivitätsträger, die in einem Adressraum arbeiten. Sie „bestehen“ aus einem Satz von Registern (vor allem Zeiger auf die nächste Anweisung und den Stack, üblicherweise mit `IP` und `SP` bezeichnet) und ihrem Speicherbereich.

Prozesse müssen im Kernel verwaltet werden, da ansonsten eine Trennung der Speicherbereiche nicht realisiert werden kann. Ein anderer Prozess könnte fremde Speicherbereiche seinen eigenen hinzufügen. Daher muss jede Änderung im Speicherbereich eines Threads vom Kernel kontrolliert werden.

2.3.1 IPC

Damit Prozesse untereinander kommunizieren können müssen Daten zwischen ihren verschiedenen abgetrennten Speicherbereichen bewegt werden. Daher muss der Kernel diesen Austausch bewerkstelligen, im Falle von L4 werden dafür Nachrichten ausgetauscht.

Andere Verfahren zur Kommunikation wie etwa Prozeduraufrufe oder shared memory können über den Nachrichtenaustausch nachgebildet oder vermittelt werden.

Zur Interruptbehandlung gibt es als Besonderheit einen speziellen Nachrichtentyp, der mit einer speziellen Absender-ID versehen ist und an den den Interrupt behandelnden (Gerätetreiber-)Prozess verschickt wird. Dadurch ist im Kernel keine gesondere Behandlung notwendig ausser der Übersetzung von Interrupts in Nachrichten.

2.3.2 UIDs

Natürlich müssen bei der Kommunikation die jeweiligen Kommunikationspartner bekannt sein, ein Dritter darf beispielsweise nicht eine Absender-ID fälschen dürfen und somit eigene Daten in fremde Verbindungen einschleusen. Daher stellt der Kernel eine eindeutige Identifizierungsmöglichkeit für Kommunikationskanäle oder Kommunikationspartner bereit.

2.4 Nicht enthaltene Teilsysteme

Wie bereits oben erwähnt befinden sich das Speichermanagment und das Paging ausserhalb des Kernels. Auch Gerätetreiber können ihre Aufgaben über Interruptnachrichten und zugeordnete Speicherbereiche erfüllen, ohne Bestandteil des Kernels sein zu müssen.

Kommunikation kann auch über Rechnergrenzen hinaus durch entsprechende Prozesse ausgedehnt werden, die Nachrichten entgegennehmen, diese in passende Protokolle verpacken und an die Netzwerkgerätetreiber weitergeben. Auf dem entfernten Rechner nimmt eine umgekehrt arbeitende Gegenseite diese Packete entgegen und wandelt diese in entsprechende Nachrichten um.

[5] ich benutze hier die Begriffe Threads und Prozesse synonym

Ausserdem können Systemaufrufe anderer Betriebssysteme, beispielsweise Unix, durch einen Prozess entgegengenommen und auf die Mechanismen von L4 abgebildet werden.

3 L4 und die Probleme der ersten Generation

In einem Mikrokernelbetriebssystem treten häufiger als in einem Betriebssystem mit Makrokernel Prozess- und Kontextwechsel auf. Zur Illustration folgendes Beispiel eines empfangenen Netzwerkpacketes:

In einem Makrokernel wie Linux empfängt der Netzwerktreiber das Packet nach einem Interrupt durch die für die Hardware typischen Mechanismen. Er gibt den Datenteil des Packets weiter an den Protokollstack, der je nach enthaltenen Protokollen die entsprechenden Behandlungsroutinen aufruft, bis das Packet schliesslich entweder vom Kernel behandelt wird[6] oder an einen Prozess übergeben wird.

Anders in einem Mikrokernel. Nachdem der Interrupt von der Netzwerkkarte ausgelöst wurde, generiert der Kernel eine entsprechende Interruptnachricht und stellt diese an den Gerätetreiberprozess zu. Um die Nachricht zu empfangen ist ein Kontextwechsel erforderlich, und falls sich der Gerätetreiberprozess noch nicht in Ausführung befindet auch ein Prozesswechsel. Nach der entsprechenden Verarbeitung werden die Daten des Packets an den entsprechenden, das jeweilige Protokoll behandelnden Prozess weitergeleitet. Dazu ist die Übermittlung einer Nachricht, also ein Kontextwechsel und ein Prozesswechsel nötig.

Ist mehr als ein Prozess an der weiteren Verarbeitung des jeweiligen Packets beteiligt macht dies nach demselben Muster weitere Kontext- und Prozesswechsel bei jedem Nachrichtenaustausch nötig.

3.1 Kontextwechsel

Es ist also klar, daß langsame Kontextwechsel die Performance des Systems stark beeinträchtigen können. Weit verbreitet ist jedoch die Annahme, dass Kontextwechsel prinzipiell teuer seien. Zur genaueren Betrachtung zunächst folgende Einteilung: Ein Kontextwechsel zerfällt in 2 Teile: den Wechsel zwischen User und Kernel und den Wechsel zwischen Adressräumen.

3.1.1 Kernel-User-Wechsel

Der Wechsel zwischen Kernel- und Userspace[7] erfolgt vor allem durch 2 Vorgänge: den Wechsel des Stacks und die Umschaltung zwischen dem privilegierten und unprivilegierten Modus des Prozessors, also im ganzen ein spezieller Funktionsaufruf.

Um seine Behauptung zu belegen, daß diese Wechsel nicht so teuer sind wie angenommen gibt Liedtke als Beispiel Zeiten für den Systemaufruf `getpid` an, welcher die Prozess-ID des aufrufenden Prozesses zurückliefert, ein Vorgang, der

[6] z.B. ICMP echo request aka. ping

[7] ein solcher Wechsel ist bei jedem Systemaufruf erforderlich

beinahe keinen eigentlichen Aufwand bedeutet. Daher ist dieser Funktionsaufruf besonders gut geeignet, den Overhead bestimmter Implementierungen zu messen, Liedtke gibt für L3, den Vorgänger zu L4, einen Overhead von 15 bis 57 Takten an, während sich der Overhead für Mach auf etwa 800 Takte beläuft.

3.1.2 Adressraumwechsel

Bei Adressraumwechseln ist zuerst eine nähere Betrachtung der zugrundeliegenden Architektur nötig, da deren Performance sehr stark architekturabhängig ist. Verantwortlich dafür ist der Translation Lookaside Buffer (TLB). Dieser ist ein Bestandteil der MMU, der die Zuordnung von logischen zu physikalischen Adressen zwischenspeichert, da ein Zugriff auf die Seitentabelle unverhältnismässig teuer wäre.

Der TLB ordnet einer gegebenen logischen Adresse eine physikalische Adresse zu. Daher wird bei einem Adressraumwechsel, bei dem sich ja die logischen Adressen ändern, der Inhalt des TLB ungültig, der TLB muss geleert werden („TLB flush“).

Es existiert jedoch bei einigen Architekturen ein sogenannter „tagged TLB“, bei dem zusätzlich eine ID für den jeweiligen Adressraum, in dem eine logische Adresse gültig ist, mitgeführt wird. Bei solchen Architekturen ist der TLB flush unnötig, vorausgesetzt der TLB hat eine ausreichende Kapazität bleiben die Einträge eines anderen Adressraums erhalten und sind nach dem Zurückwechseln in diesen Adressraum auch wieder gültig und benutzbar. Insbesondere bei häufigen Wechseln zwischen einigen wenigen Prozessen[8] werden dadurch gegenüber Architekturen mit untagged TLB die Vorteile deutlich.

Die gesamten Kosten für einen Adressraumwechsel ergeben sich aus dem Wechsel der Segment- und Seitentabelle und dem eventuell nötigen TLB flush. Bei Architekturen ohne tagged TLB können sich hier bis zu einigen hundert Takten Overhead ergeben, die aber durch geschickte Wahl des Verfahrens minimierbar sind. Beispielsweise gibt Liedtke ein Verfahren an, wie man auf PowerPC-Prozessoren das Verhalten eines tagged TLB simulieren kann und somit trotz einer Architektur mit untagged TLB fast dieselbe Performance erreicht wie mit tagged TLB.

3.1.3 Prozesswechsel

Um die Performance bei einem Prozesswechsel zu messen wurde ein Prozeduraufruf, der 1 byte verschickt auf verschiedenen Systemen verglichen. In Tabelle 1 ist jeweils die Zeit für das Hin- und wieder Zurückschicken und die Anzahl der CPU-Takte für den einmaligen Weg angegeben. Da sich die Kosten aus Adressraumwechseln, Systemaufrufen und der zur Parameterübergabe nötigen Kommunikation zusammensetzen kann man hieran die Performance der bisher genannten Mechanismen vergleichen. Die Performance in diesen Bereichen ist ausreichend, um in L4 auch Interrupts auf die genannte Weise zu bearbeiten.

[8] Beispiel: Ein Serverprozess, der Anfragen aus dem Netz beantwortet und sich daher häufig mit dem Netzwerkkartentreiber und dem Protokollstack abwechselt

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*

Abbildung 1: Performancemessung von 1-byte-RPC-Aufrufen

3.2 Speichereffekte

Als eine weitere Ursache für Performanceprobleme bei Mikrokerneln wie Mach gibt Liedtke die gegenseitige Verdrängung von Kernel und Userprozessen aus dem Cache an. Hierbei sollen beiMach bis zu 0.25 Takte pro Prozessorbefehl an den sogenannten „memory cycle overhead per instruction“ (MCPI) verloren gehen.

Gelöst werden kann dieses Problem durch eine Verkleinerung der Arbeitsmenge des Kernels.

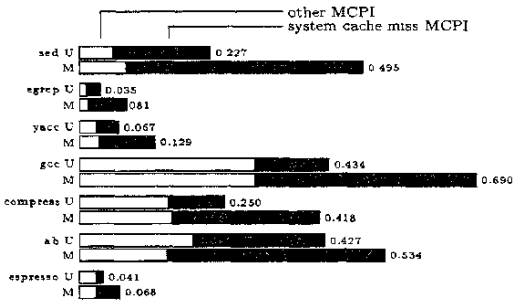

Figure 3: *Baseline MCPI for Ultrix and Mach.*

Abbildung 2: MCPI von Ultrix (Makrokernel) und Mach (Mikrokernel)

3.3 Portabilität

Mikrokernel der ersten Generation haben, um einen möglichst grossen Teil des Kernels hardwareunabhängig zu halten eine Abstraktionsschicht innerhalb des Kernels eingeführt. Dies erzeugt jedoch zusätzlichen Overhead und verhindert spezielle Optimierungen auf die Gegebenheiten der Hardware. Diese speziellen Optimierungen sind jedoch besonders bei Adressraumwechseln unerlässlich. Hierbei spielen selbst vermeintlich kleine Unterschiede in der Architektur, wie die zwischen Pentium und 486 eine Rolle.

Daher bedingt ein anderer Prozessor immer ein anderes Design des Mikrokernels um dem Rechnung zu tragen. Somit sind Mikrokernel prinzipiell nicht portabel. Allerdings ist es möglich, einen Mikrokernel als Basis für ein portables Betriebssystem zu nutzen.

4 Zusammenfassung

Mikrokernel sind, wie wir gesehen haben, die Kernel mit der jeweils kleinstmöglichen Menge an Funktionen, die die Aufgaben die an einen Kernel gestellt werden, noch erfüllen. L4 stellt für die gestellten Anforderungen eines interaktiven Systems mit unabhängigen Prozessen mit garantierter Integrität als Abstraktionen Adressräume, Prozesse, IPC und eindeutige IDs zur Verfügung. Um eine akzeptable Performance zu erreichen muss die jeweilige Implementierung eines Mikrokernels speziell auf die zugrundeliegende Architektur zugeschnitten sein.

Literatur

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

[2] 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).