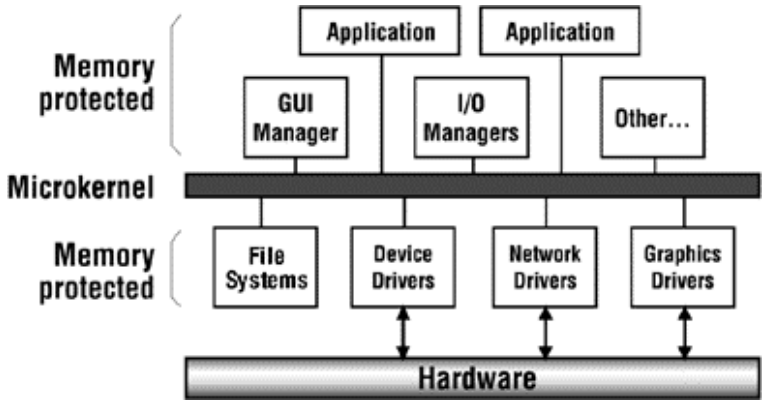
Betriebssysteme und Microkern

Seminararbeit für das Seminar:
"Ausgewählte Komponenten für Betriebssysteme"



Andreas Steinel <lnxbil@cs.uni-sb.de>

7. Oktober 2004

Geschrieben von Andreas Steinel[1] im September 2004

Das Bild auf der Titelseite wurde von folgdener Webseite genommen und entstannt der Dokumentation von QNX:

`http://www.freeos.com/articles/4160/`

[1] Andreas Steinel <lnxbil@cs.uni-sb.de>

Inhaltsverzeichnis

1 Abriss **2**

2 Übersicht **3**
2.1 Was macht ein Betriebssystem? . . . 3
2.2 Anwendungsorientierte Arten . . . 3
2.2.1 Miniatursysteme: . . . 3
2.2.2 Einzelprogramm- und Mehrprogrammsysteme: . . . 3
2.2.3 Einzelbenutzer- und Mehrbenutzersysteme: . . . 3
2.2.4 Großrechner: . . . 4
2.3 Technisch orientierte Arten . . . 4

3 Betriebssystemkonzepte **5**
3.1 Prozesse/Tasks . . . 5
3.2 Interrupt-Behandlung . . . 5
3.3 Prozess-Verwaltung und -Umschaltung . . . 5
3.4 Speicherverwaltung . . . 6
3.5 Abstraktion von Geräten . . . 6

4 Monolithischer Kern **7**
4.1 Leistungsmerkmale . . . 7
4.2 Weiterentwicklung . . . 7
4.2.1 Schichten-Modell . . . 7
4.2.2 Kapselung von Diensten . . . 7
4.2.3 Client-Server-Modell . . . 8
4.2.4 Modularisierung und Sandboxing . . . 8
4.3 Überblick . . . 8

5 Microkern **9**
5.1 Client-Server-Modell bei Microkern . . . 9
5.1.1 Single-Server . . . 9
5.1.2 Multi-Server . . . 9
5.1.3 Betriebssystem-Server . . . 9
5.1.4 Zwischenprozesskommunikation . . . 9
5.2 Microkern . . . 10
5.2.1 Erste Generation . . . 10
5.2.2 Zweite Generation am Beispiel L4 . . . 10

6 Zusammenfassung **12**

7 Quellen **13**

1 Abriss

Diese Seminararbeit geht auf den Begriff Betriebssystem ein, deren verschiedene Arten wie Teilzeit- und Echtzeitbetriebssysteme, deren Anwendungsgebiete wie z.B. Heimcomputer und Großrechner mit Betriebssystembeispielen.
Im nächsten Teil der Arbeit werden die Konzepte wie z.B. virtueller Speicher, Prozesse und Tasks, Threads, Benutzer-Modus und System-Modus erklärt und deren Bedeutung erläutert. Der darauf folgende Abschnitt handelt von der Beziehung zwischen dem monolithischen Kern und Microkern. Dabei werden Aspekte wie Designziele, Erfahrungen und Portabilitätsfaktoren diskutiert.
Eine kurze Übersicht über bisherige Microkern-Implementierungen und eine Zusammenfassung runden die Arbeit ab.

2 Übersicht

2.1 Was macht ein Betriebssystem?

Einfach ausgedrückt tut ein Betriebssystem (engl. *Operating System (OS)*) nichts anderes, als sich um die Hardware zu kümmern. Es stellt die Schnittstelle zwischen der Hardware und der darauf aufbauenden Software dar. Als Hardware wären die typischen als Hardware bezeichneten Komponenten eines Computer zu nennen, die man fast täglich in der Hand hält, bzw. drauf schaut:

- **Eingabegeräte:**
  Maus, Tastatur, Trackball
- **Ausgabegeräte:**
  Monitor, Drucker
- **Datenmedien:**
  CD-ROM, DVD, USB-Sticks oder sogar noch Disketten

Aber auch die zu den wichtigen **Betriebsmitteln** gehörenden Komponenten der Hardware sind ausschlaggebend. Im eigentlichen Sinne sind nur die folgenden Teile für eine Rechenmaschine notwendig:

- **Rechnereinheiten** (engl. *Central Processing Unit*, CPU):
  verschiedene Architekturen wie die Weit verbreiteten im **Computer-Alltag** (*x86, ia64, amd64, PowerPC, Sparc, Alpha*), **High-End/Performance-Computing (HPC)** (S390 und andere Mainframes) als sowohl auch die **Embedded-Prozessoren**, z.B. im Handy (ARM, RISC, MIPS)
- **Speicher** (engl. *Memory*):
  Dieser kann sich auch 2 verschiedenen Arten von Speicher zusammensetzen: Die erste Kategorie ist Speicher in Form vom **Hardware-Speicherbausteinen (Random Access Memory, RAM)** und die zweite Art wäre der **Swap-Speicher** auf der Festplatte (siehe 3.4)

2.2 Anwendungsorientierte Arten

Diese Betriebssysteme werden katalogisiert durch ihre Anwendung:

2.2.1 Miniatursysteme:

Alle Arten von eingebetteten Computern verfügen über solche Betriebssysteme. Einige bekannte Beispiele wären z.B. Handys, Autos, aber auch Hardware-Router oder sonstiges.

Betriebssystembeispiele: MuLinux

2.2.2 Einzelprogramm- und Mehrprogrammsysteme:

Diese Systeme wurden so konzipiert, dass nur ein bestimmtes Programm gleichzeitig darauf laufen kann. Dieses Programm ist in der Lage, die gesamte Hardware des Rechners ßu sehen". Nun gibt es Systeme, die nur für ein Programm gebaut wurden: Kassenautomaten oder sonstiges. Es gibt aber auch Mehrprogrammsysteme, die die Ausführung mehrerer Anwendungen erlaubt, jedoch nicht zur gleichen Zeit. Auf solche Systeme kann man z.B. Textverarbeitung betreiben oder ein Computerspiel spielen. Als Beispiel wäre hier MS-DOS zu nennen, auf dem man ja bekanntlich nur ein Programm zur gleichen Zeit ausführen konnte und auch Windows 3.x, dass nur sog. *kooperatives Multitasking* konnte.

Betriebssystembeispiele: MS-DOS

2.2.3 Einzelbenutzer- und Mehrbenutzersysteme:

Sie stellen die Weiterentwicklung der Einzel- und Mehrprozessorsysteme dar. Alle gängigen Betriebssysteme wie Linux, BSD, BeOS und auch Windows wären hier zu nennen. Sie bieten die Möglichkeit, mehrere Programme gleichzeitig auszuführen, was einen enormeren Anspruch an das Betriebssystem stellt, als es bei den Einzel- und Mehrprogrammsystemen der Fall war. Für diese Systeme wurde meistens ein Computer mit nur einem Prozessor verwendet (Home-Bereich) und zwei, vier, acht oder auch mehr im

Server-Bereich. Bei diesen Systemen kam *präemptives-Multitasking* (bei Uni-Prozessor-Systemen) und *echtes-Multitasking* (bei Mehr-Prozessor-Systemen)

Betriebssystembeispiele: Linux, BSD, Windows ab NT

2.2.4 Großrechner:

Heutzutage in 2 Arten vertreten als **hardwareseitige Rechneranlagen** immenser physikalischer Grössen, wie z.B. Mainframes von IBM und Sun aber auch als **Rechen-Cluster**, die eine Zusammenschaltung vorhandener Rechneranlagen, aber auch nur von einzelnen PC's sein können. Hier wurde auch *echtes Multitasking* verwendet.

Betriebssystembeispiele: IBM S390

2.3 Technisch orientierte Arten

Hier werden zwei grundlegende Unterscheidungen gemacht.

- Echtzeitbetriebssysteme (engl. *Real-Time OS*)
- Teilzeitbetriebssysteme (engl. *Time-Sharing OS*)

Bei einem Echtzeitbetriebssystem ist es möglich, die Ausführung eines Programms in einer gewissen Zeitspanne zu garantieren. Dadurch ermöglichen sich neue Anwendungsgebiete, wie z.B. Industrieroboter. Bei diesen ist das Timing wichtig: Wenn der Roboterarm eine Zehntel Sekunde zu spät stoppt könnte das Fertigungsteil schon beschädigt sein. Im Gegensatz dazu gibt es die Teilzeitbetriebssysteme, wie alle gängigen PC-Versionen, bei denen die Rechenzeit in Zeitscheiben eingeteilt wird und somit jedem Prozess eine gewisse Anzahl von Zeitscheiben zur Verfügung stehen. Sozusagen bekommt jedes Programm ein Stück vom Kuchen ab.

3 Betriebssystemkonzepte

Grundlegend unterscheidet man bei Betriebssystemen immer folgende beiden Typen:

- Monolithische Systeme
- Microkern Systeme

Dabei verwenden beide Ansätze gewisse Konzepte, mit denen man die Aufgaben in Komponenten zerlegen kann:

3.1 Prozesse/Tasks

Ein Prozess besteht aus einer Task, die den Adressraum für eine Anwendung bereitstellt und mindestens einem Thread. Ein Thread ist der in Ausführung befindlicher Code eines Prozesses (engl. *execution flow*). Es ist möglich mehrere Threads in einem Prozess ablaufen zu lassen.
In jedem dieser Threads werden Verwaltungsinformationen für das Betriebssystem abgelegt z.B. bei Linux: die PID (*Prozess ID*), die PID des Vaterprozesses (*PPID*), Real-User-ID, Arbeitsverzeichnis, File-Deskriptor-Tabelle, Signal-Slots oder Umgebungsvariablen die für das auszuführende Programm wichtig sind.
In dem Task werden die folgende Komponenten des Programms abgelegt (hier vereinfachtes `a.out`-Format):

- argv-Struktur
- envp-Struktur
- Benutzer-Stack
- Benutzer-Heap
- nicht initialisierter Daten (*Daten-Segment*)
- initialisierte Daten (*Daten-Segment*)
- Programm-Code (*Text-Segment*)

Jeder Prozess läuft in seinem eigenen Adressraum, der nur von Thread gelesen und beschrieben werden kann, die auf dem gleichen Adressraum arbeiten. Beim Laden eines Programms wird der Programmcode ab einer fixen niedrigen Adresse platziert. Diese ist nur lesbar für den Prozess. Oberhalb des Text-Segmentes wird der initialisierte Datenbereich abgelegt, welcher beschreibbar ist. Darauf wird nun der Bereich der nicht-initialisierten Daten angelegt, welcher globale, nicht initialisierte Programm- variablen enthält. Nun folgt der Heap, welcher in Richtung höherer Adressen wächst und der Programm-Stack, der entgegen dem Heap wächst. Nun folgt noch die Umgebungsvariablen und die Übergebenen Programm-Parameter.

3.2 Interrupt-Behandlung

Ein Interrupt ist ein von der Hardware geworfene Unterbrechung, die vom Betriebssystem abgehandelt werden muss. Normalerweise registriert ein Hardware-Bauteil eine Interrupt-Leitung und ein Treiber im Betriebssystem kann mittels dieser Leitung feststellen, ob z.B. neue Daten vorliegen und somit auf den Interrupt reagieren. Als Beispiel wäre hier der Timer-Interrupt zu nennen, der von einem Quarz-Kristall-Bauteil periodisch in einem bestimmten Intervall ausgelöst wird.

3.3 Prozess-Verwaltung und -Umschaltung

Es wird grundsätzlich unterschieden zwischen Programmcode im **System-Modus** oder im **Benutzer-Modus**. Der Wechsel zwischen diesen beiden Modi wird **Kontext-Wechsel** genannt. Jeder Benutzer-Prozess arbeitet im Benutzer-Modus.
Da alle Prozesse und auch Threads vom Betriebssystem verwaltet werden müssen, muss auch eine Mechanismus existieren, der regelt, wie die Prozesse/Threads umgeschaltet werden. Diese Aufgabe übernimmt der **Scheduler**. Er verwaltet die Prozesse und damit auch die Threads, die als nächstes auf eine freie CPU

zum Ablauf kommen sollen. Dabei wird auf die Priorität des Prozesses geachtet und dem entsprechend die Warteschlange umsortiert. Für den Kontext-Wechsel ist der **Dispatcher** verantwortlich, der mittels hardwaregestützten Register-Operationen seine Arbeit verrichtet.
Auf neuen System gibt es sogar **präemption**, die es ermöglicht ein Programm an jeder Stelle zu unterbrechen (auch Treiber in monolithischen Systemen, System-Modus) und danach ein anderen Prozess auf die CPU zu lassen.[1]

3.4 Speicherverwaltung

Die Speicherverwaltung heutiger Rechner ist dadurch gekennzeichnet, dass der Speicher nicht mehr alleine durch die eingebaute Grösse des Arbeitsspeichers bestimmt wird. Es kann (und wird normalerweise auch) eine Swap-Datei oder eine Swap-Partition benutzt um die Speicherkapazität zu erhöhen.
Technisch gesehen wurde eine virtuelle Speicherverwaltung eingeführt. Die Hauptgründe hierfür waren die Sicherheitsaspekte, mit denen endlich die Trennung von Programmen im Bezug auf den Speicher möglich war. Nun ist es nicht mehr möglich auf Speicherstellen eines anderen Programms zu schreiben.
Beim Start des Betriebssystems wird die Speicherverwaltung gestartet und nun existieren für den Kern zwei verschiedene Arten von Speicher-Zugriffen möglich:

- direkter physikalischer Speicher-Zugriff: [2]
  Hier kann wie früher sofort auf eine physikalische Speicher-Adresse zugegriffen werden.
- indirekter virtueller Speicher-Zugriff: [3]
  Speicher-Zugriff erfolgt auf eine virtuelle Adresse im Speicher. Diese Adresse muss erst durch spezielle Kern-Funktionen der Speicherverwaltung aufgelöst werden und somit zu einer physikalischen Adresse umgewandelt werden.

Diese Unterscheidung ist nur für den Kernel möglich. Ein Prozess hat immer nur virtuelle Adressen, die dann intern mittels Kern-Funktionen abgebildet (engl. *mapped*) werden.
Falls von einem Programm A eine virtuelle Adresse angefordert wird, welche noch nicht auf physikalischen Speicher abgebildet worden ist, wird ein *Page-Fault*-Interrupt ausgelöst. Dieser führt nun dazu, dass der Kern in den Kernel-Modus umschaltet und damit eine freie Speicher-Seite nachläd und die virtuelle Adresse (und die folgenden Adressen bis zur Grösse der Speicher-Seite) auf diesen virtuellen Speicher im Prozess-Adressraum abbildet und danach die Kontrolle wieder an das Programm A abgibt, also den Kernel-Modus verlässt und wieder in den User-Modus wechselt, sodass es mit seiner Programmierung fortfahren kann.
Der Swapping-Mechanismus wird benutzt, um schon lange nicht mehr benötigten Speicher-Inhalte des physikalischen Speichers auf in die Swap-Datei auszulagern, um wieder mehr Hauptspeicher zur Verfügung zu stellen. Er ändern die virtuellen Adressen jedoch nicht, sodass die Benutzer-Prozesse davon nichts mitbekommen. Alles geschieht vollkommen transparent. Wenn jetzt nun ein Programm B wieder auf die virtuelle Adresse zugreifen will, die ausgelagert wurde, kommt wieder ein *Page-Fault* und der Kern läd die Speicher-Seite wieder in den physikalischen Speicher und das Programm B fährt fort.
Virtueller Speicher wird auch benutzt, um Funktionen von Programm-Bibliotheken(*shared-Libraries*) in den Adressraum einzublenden und somit die Speicherverwaltung zu optimieren. Es ist weniger Platz verbrauchend, wenn gleicher Code nur einmal im physikalischen Speicher liegt, als wenn dies mehrfach der Fall ist.

3.5 Abstraktion von Geräten

Vereinfachung der Schnittstellen durch Abstraktion und einheitlich API. Dadurch ist es möglich auf Hardware mit generischen wie z.B. `read` und `write` beim Dateisystem zu lesen und zu schreiben, obwohl das zugrunde liegende Dateisystem ein komplett verschiedenes Design und Handhabung hat.
Die Programmierung auf Benutzer-Seite ist somit viel leichter zu handhaben, da man über interne Strukturen des Kernels nicht bescheide wissen muss.

[1] Als Beispiel hierfür wäre z.B. eine Endlosschleife in einem Treiber bei monolithischen Systemen. Ohne präemption würde das System stillstehen. Mit präemption könnte man das System immer noch benutzen.
[2] Im Linux-Kern mittels `kmalloc`
[3] Im Linux-Kern mittels `vmalloc`

4 Monolithischer Kern

Ein monolithischer Kern ist ein Zusammenschluss von allen Treibern und Dingen, die man von einem Betriebssystem erwarten würde.

4.1 Leistungsmerkmale

Alle Funktionen arbeiten im Kernel-Modus, indem sie Zugriff auf den kompletten Adressraum des Kernels haben.
Auf die einzeln Funktionen kann man mittels **System-Call** zugreifen. Dieser ist durch einen **Down-Call** realisiert, d.h. die Anforderung kommt aus dem User-Space (Benutzer-Seite des Systems) und führt hinunter in den Kernel-Space(Kernel-Seite des Systems) und bleibt auch dort (somit kein **Up-Call**), bis das Ergebnis zurückgesendet wird.
Es gibt keine Trennung der einzelnen Treiber voneinander. Somit kann jeder Treiber Daten eines anderen Treibers oder sonstige interne Datenstrukturen des Kern beschädigen oder zerstören. Durch diese Bauart ist jedoch möglich direkt und ohne Umwege mit den Treibern zu kommunizieren und somit arbeitet der Kern sehr schnell. Leider kann man diese Art eines Kerns nicht einfach und ein paar Funktionen erweitern, da man das ganze System verstehen muss. Außerdem muss man wegen der Hardwarenähe sehr viel neu schreiben, wenn man den Kern portieren will.

4.2 Weiterentwicklung

Um die Schwachpunkte des monolithischen Ansatzes zu verbessern, hat man sich Gedanken gemacht, wie man es machen könnte:

4.2.1 Schichten-Modell

Zum einen gab es das **Schichten-Modell**, mit dem man den monolithischen Kern in Schichten eingeteilt hat und somit eine Struktur in den bislang totalen Wirrwarr hineingebracht hat. Damit wurde es möglich Abstraktionsschichten wie z.B. virtuelle Dateisysteme einzuführen, mit denen man auf jegliche Art von Dateisystemen mit den gleichen Operationen (`read`,`write`,...) zugreifen kann. Dadurch wurde der Monolith:

- flexibler
  da jeweils eine neue Schicht auf eine bereits vorhandene aufgesetzt werden kann
- weniger komplex
  da nun eine Teilung in logische Einheiten möglich war

Die Performance ist immer noch recht gut, jedoch etwas schlechter als bei reinen Monolithen, da der Weg einer oberen in eine untere Schicht über alle Schichten erfolgen muss (Pfadlänge). Jedoch waren Aspekte wie gegenseitige Treiber-Beeinflussung mit dem Schichten-Modell immer noch nicht gelöst.

4.2.2 Kapselung von Diensten

Ein weiterer Verbesserungsvorschlag basierte auf der **Kapselung von Diensten**. Damit war es möglich den Treibern einen eigenen Adressraum zu geben, da sie als Benutzer-Prozess im Benutzer-Adressraum laufen. Dadurch konnte man einige der Mißstände des Monoliths lösen:

- flexibler
  es musste nur ein neuer Treiber-Prozess gestartet werden
- weniger komplex
  Trennung von Treibern und Auslagerung in den Benutzer-Adressraum
- sicherer
  kein Treiber kann auf Daten eines anderen Treibers zugreifen.

Performance ist niedriger, da zwischen den Prozessräumen kommuniziert und umgeschaltet werden muss. Es ist keine direkte Kommunikation aufgrund der getrennten Adressräume möglich.

4.2.3 Client-Server-Modell

Als Weiterentwicklung der Kapselung von Diensten wird das **Client Server Modell** benutzt. Hierbei laufen die Treiber auch im Benutzer-Adressraum, in sog. Servern. Jeder andere Benutzer-Prozess arbeitet als Client und kann die Dienste, die von den Servern angeboten werden benutzen. Unter einem Server wird aber nicht nur ein Treiber, sondern auch höhere Betriebssystemfunktionalität verstanden. Man kann z.B. einen *Pager* auch als Benutzer-Prozess realisieren.

4.2.4 Modularisierung und Sandboxing

Bei Modularisierung handelt es sich um eine Technik, bei der zur Laufzeit des Betriebssystems teile hinzugefügt oder entfernt werden können. Alle modernen monolithischen Systeme verwenden diesen Ansatz, wobei man dann meistens von **Hybrid-Monolithischen-Systemen** spricht.
Hierfür existiert auch eine Technik namens **Sandboxing**, die hier nur sehr kurz angesprochen wird, mit der es möglich ist, durch Verwendung von speziellen Programmiersprachen wie z.B. Modula, Treiber-Module zu übersetzen, die strikten Kontrollen im Bezug auf die Allokation von Speicher unterliegen.

4.3 Überblick

	Ur-Monolith	Schicht-Modell	Client-Server-Modell	Hybrid-Monolith
Portabilität	schlecht	mittelmässig	sehr gut	gut
Performance	gut	gut	mittelmässig	gut
Robustheit	schlecht	schlecht	gut	schlecht
Flexibilität	schlecht	mittelmässig	gut	gut
Sicherheit	schlecht	schlecht	gut	mittelmässig

5 Microkern

Das Client-Server-Modell wird nun verwendet, um eine Ära der Betriebssystemkerne einzuführen. Dabei wird versucht, so wenig wie möglich im Kern zu machen und so viel wie möglich auszulagern. Aber es ist nicht genau festgelegt, welche Funktionalität man auslagern sollte und welche nicht. Daher gib es viele verschiedene Implementierungen von Microkern-Betriebssystemen, die sich alle Microkern nennen, jedoch unterschiedliche Auffassung des Begriffes vertreten.
Der Ansatz wurde in den 80er Jahren von immer mehr Betriebssystem-Entwicklern aufgefasst und diente Hauptsächlich der Vereinfachung der Betriebssystem-Konstruktion, Verbesserung der Sicherheitskonzepte und verwenden als Verteilte Systeme (siehe 5.2.1) In der heutigen Zeit bietet sich der Microkern (hauptsächlich L4) an verifiziert zu werden, da eh sehr schlank ist und somit recht einfach verifiziert werden kann. Darüberhinaus ist es einfach einen Microkern zu portieren, als einen Monolith.
Jedoch schauen wir zuerst einmal auf die Client-Server Architektur eines Microkerns:

5.1 Client-Server-Modell bei Microkern

Der Microkern bietet durch Systemaufrufe eine Schnittstelle an, mit der man mit dem Kern kommunizieren kann. Direkt auf dieser Schnittstelle beginnt die Server- und Client-Schicht. Server sind hierbei Treiber oder ausgelagerte höhere Betriebssystem-Funktionen des Kerns. Wie schon erwähnt sind alle normalen Benutzer-Prozesse Clients.
Ein weiterer Vorteil des Client-Server-Modells liegt darin, dass alle Server, die keine Treiber sind, komplett Plattform unabhängig sind und somit nicht portiert werden müssen.
Bei der Server-Architektur wird zwischen zwei Arten unterschieden:

5.1.1 Single-Server

Hierbei existiert ein einziger Server, der alle höheren Betriebssystemfunktionalitäten bereitstellt. Dies könnte man mit einem monolithischen Kern auf dem Microkern vergleichen, jedoch läuft dieser im Benutzer-Adressraum.

5.1.2 Multi-Server

Mehrere Server teilen die Betriebssystemfunktionalität untereinander und stellen diese als Dienste bereit. Auf diese Dienste kann nun ein Client oder ein anderer Server-Prozess zugreifen.
Dabei wird nochmal unterschieden zwischen

- horizontalem Zugriff (Unabhängige Server, jeder kommuniziert nur direkt mit dem Kern)
- linearem Zugriff (Server können voneinander abhängig sein)

5.1.3 Betriebssystem-Server

Nun gibt es noch sog. Betriebssystem-Server (*OS personality*). Diese stellen ein komplettes Betriebssystem in einem oder mehreren Server dar. Damit ist es möglich verschiedene Betriebssysteme gleichzeitig auf einer Maschine auszuführen. Im Gegenteil zu VMWare oder ähnlichen Projekten wird nicht ein Rechner emuliert, sondern der Code laüft parallel zum anderen Betriebssystem auf dem Prozessor. Der Geschwindigkeitsunterschied ist natürlich enorm: ein Hybrid-OS, wie es auch genannt wird, ist wesentlich schneller als ein emuliertes.

5.1.4 Zwischenprozesskommunikation

Kommuniziert wird bei einem Microkern mittels Zwischenprozesskommunikation (IPC), da direkte Kommunikation mittels globalen Variablen oder ähnlichem wegen den getrennten Adressräumen nicht möglich ist. Dabei werden zwei Arten von IPC unterschieden:

- synchrones IPC (Warten auf Antwort)
- asynchrones IPC (nicht Warten auf Antwort)

Die Kommunikation wird durch einen sog. Trampolin Prinzip übermittelt: Der IPC-Sender schickt die Nachricht an den Kern. Dieser leitet die Nachricht an den IPC-Empfänger weiter. Dieser sendet die Antwort wieder an den Kern, der sie an den urspünglichen Sender zurückschickt.

5.2 Microkern

Bei den Microkernen muss man zwischen zwei Generationen unterscheiden. Die erste Generation entsprach nicht den Vorstellungen der Betriebssystem-Entwickler, da sie sehr langsam war. Man schob den Schwarzen Peter auf die Microkern Architektur, was jedoch nicht stimmte. In der zweiten Generation wurde genau diese falsche Prämisse widerlegt.

5.2.1 Erste Generation

Am Beispiel Mach

Als ein Beispiel für die erste Generation von Microkernen kann man das Mach-Betriebssystem heranziehen. Es wurde 1989 stückweise aus BSD (Berkeley Software Distribution) 4.2 und später 4.3 Code zusammengesetzt und war erst in der Version 3 ein reiner Microkern. Entwickelt wurde es Carnegie Mellon University (CMU) in den USA.
Die Ziele dieses Betriebssystems war es viele Architekturen zu unterstützen, Netzwerk-Transparenz zur Verfügung zu stellen, Parallelismus bei Programmen und größere Adressräume zu benutzen, da dies früher nicht so selbstverständlich war wie es heute ist. Vor allem sollte es aber eine Plattform für die Entwicklung neuerer Betriebssysteme sein. Leider war Mach wegen des großen Anteils an Berkeley Code sehr langsam und groß geraten. Eingführten Konzepte waren u.A. Ports, Port Rights und Memory Objects.

Am Beispiel Amoeba

Amoeba ist ein Betriebssystem, das darauf ausgelegt war, auf vielen Rechnern zu laufen und diese als einziges System erscheinen zu lassen. Es entstand 1981 an der *Vrije Universiteit* in Amsterdam. Entwickelt wurde es vor allem von *A. Tanenbaum*. Es basiert nicht auf Unix, sondern wurde komplett neu entwickelt. Ziel war es ein komplett verteiltes, transparentes System zu schaffen, das Gruppen-Kommunikation erlaubt und Shared Memory unterstützt. Es wurden außerdem Prozessorpools benutzt, die nur aus Prozessoren, Speicher und einer Netzwerkschnittstelle verfügten. Diese konnten sogar heterogen sein. Im Gegensatz zu den Rechnerpools gab es auch Terminals, von denen man arbeiten konnte. Das ganze Amoeba-System schien nach außen immer als ein System und nicht als ein Verbund von Rechnern.

5.2.2 Zweite Generation am Beispiel L4

Die Zweite Generation schloss die Performancelücken, die die Erste Generation auf warf. Jochen Liedtke entwickelte 1995 den Microkern L4 mit einem Minimalprinzip: Nur die Prozess-Verwaltung, Speicherverwaltung und schnelle Zwischenprozesskommunikation sollen im Kern Platz finden. Außer dem drei eben erwähnten Abstraktionen sind sich nur 7 Systemaufrufe im Kern. Diese erste L4 Implementierung legte den Grundstein für die gesamte L4-Familie, die auf dieser ersten Implementierung aufbauten.

Geschwindigkeitsverbesserungen

Die L4-Entwickler fanden den Flaschenhals, der für die enormen Geschwindigkeitsunterschiede zwischen den monolithischen Systemen und der 1. Generation von Microkernen verantwortlich war. Es war die unzulängliche Implementierung der Zwischenprozesskommunikation (IPC). L4 wurde nun extra auf eine sehr schnelle Zwischenprozesskommunikation ausgelegt.
Ein Erweiterung der Zwischenprozesskommunikation wurde auch eingeführt, sog. **leichte Zwischenprozesskommunikation**, bei der innerhalb eines Prozesses Daten zwischen Threads ausgetauscht werden konnten, ohne einen Adressraum-Wechsel durchzuführen.

externe Dienste

Der Pager wurde in der L4-Familie als Server-Dienst in den Benutzer-Adressraum ausgelagert. Sobald nun ein Seiten-Fehler auftritt wird vom Kern eine Nachricht generiert und an den Pager geschickt. Dieser läd die Seite in den Ziel-Prozess ein und somit ist der Seiten-Fehler abgehandelt.

Bei den Servern wurde auch noch eine andere Methode eingeführt: Das reintegrieren eines Servers in den System-Adressraum. Somit können Geschwindigkeitskritische Aufgaben schneller erledigt werden. Ob dies nun von Nützen ist, auf die Vorteile von Servern zu verzichten liegt somit aber immer noch bei den System-Administratoren.

Flexpages

Das neue Speicher-Konzept von **Flexpages** beschreibt ein Speicher-Objekt beliebiger Größe in einem Adressraum. Es besitzt eine Basisgröße und eine Länge.
Flexpages können mittels `grant`, `flush` und `map` in fremde Prozess weitergegeben, entzogen oder gespiegelt werden konnten.

Clans und Chiefs

Eine kurze Daseins-zeit in der L4-Familie hatte das `Clans und Chiefs` Konzept. Dies soll hier nur der Vollständigkeit halber sehr kurz beschrieben werden.
Mit diesem Konzept war es möglich eine Art Nachrichtentransferbaum zu erzeugen. Jede Nachricht wurde immer an den Chief eines Clans gesendet. Jeder Prozess war in einem Clan und hat einen Chief oder ist selbst der Chief. Somit wird die Nachricht immer so lange an den höhergeordneten Chief weitergegeben, bis der Ziel-Prozess im eigenen Clan oder deren Unter-Clans liegt.
Leider war es nicht performant genug und wurde deshalb wieder aus neueren L4 Implementierungen gestrichen.

Ressourcen-Belegung

Bei L4 bekommt jeder Prozess, der als erstes eine Ressource, wie z.B. Speicher oder Interrupts, belegt, die kompletten Rechte an der Ressource, es sei denn, sie gehört schon einem anderen Prozess. Somit arbeitet der Pager, auch sigma0 genannt, auf dem kompletten physikalischen Speicher. Er wird auch 1:1 gemappt, sodass alle virtuellen Adressen des Pager-Adressraumes genau den physikalischen Adressen entsprechen. Sigma0 wird also direkt nach dem booten des L4-Kerns geladen und ist somit der 1. Prozess der nicht-Kern-Speicher verwendet.

Systemaufrufe

Auszug der 7 Systemaufrufe mit Verwendung des Clans und Chiefs Konzepts:

- ipc
  Erzeugen und versenden einer Zwischenprozessnachricht
- id_nearest
  ID des nächsten Kommunikationspartners (Chief) herausfinden
- fpage_unmap
  Flexpage aus dem Speicher-Bereich freigeben
- thread_switch
  freiwillig CPU abgeben
- thread_schedule Schedule Mechanismen beeinflussen
- lthread_ex_regs
  Thread-Wechsel durchführen (u.A. auch Register austauschen)
- task_new
  Neuen Task kreieren

Verifikation

Der L4-Kern, mit seinen 7 Systemaufrufen und gerade mal 12K Größe im Speicher, eignet sich hervorragend für eine Verifikation. Je kleiner und unkomplizierter ein Gebilde ist, desto leichter ist es zu verifizieren. Die Verifikation des L4-Microkerns am Lehrstuhl Paul, Universität des Saarlandes, steht kurz vor der Vollendung.

6 Zusammenfassung

Betriebssysteme bilden den Grund-Stock jedes komplexen technischen Gerätes, sei es Handy, Fernseher oder auch der Computer. Wir benutzen sie tagtäglich und sie sind nicht mehr wegzudenken.
Sei es der monolithische oder der microkernsche Ansatz, beide besitzen Tasks und Threads, Interrupt-Mechanismen oder allgemeine Abstraktionen. Wir haben gesehen, dass ich die Ur-Monolithen wegen ihrer viel zu hohen Komplexität und Fehleranfälligkeit nicht mehr für heutige Systeme eigenen. Es wurde nach Verbesserungen gesucht und diese in Form des Schichten-Modells und Sandboxing auf der monolithischen Seite, als auch mit Auslagerung und Client-Server-Modell auf der microkernschen Seite gefunden.
Die Microkern-Geschichte kann man in zwei Abschnitte zerteilen, der Ersten Generation und der Zweiten Generation, die sich durch enorme Geschwindigkeitsdefizite unterscheiden. Erst mit der Zweiten Generation war der Schritt in die richtige Richtung getan.
Die wesentlichen Unterschiede zwischen dem Microkern und dem Monolith waren die wesentlichen Sicherheitsverbesserungen auf Seite der Microkernen. Dort wurde mittels eigenen Adressräumen die Komplexität und Fehleranfälligkeit der einzelnen Betriebssystemkomponenten verringert.
Da in der heutige Zeit die Sicherheit mehr denn je im Vordergrund steht, man nehme sich nur mal die unzähligen Viren-Attacken, werden den Microkernen auch immer mehr populärer.

7 Quellen

1) **A. S. Tanenbaum:** *Modern Operating Systems"*,
Prentice Hall 1992.

2) **Jochen Liedtke:** *Toward Real Microkernels"*,
Communications of the ACM, 39(9), pp. 70-77, September 1996

3) **Jochen Liedtke:** *"µ-Kernels Must And Can Be Small"*,
5th IEEE International Workshop on Object-Orientation in Operating Systems (IWOOOS), Seattle, WA, October 1996

4) **Jochen Liedtke:** *Ön µ-Kernel Construction."*,
In Proceedings of the Fifteenth ACM Symposium on Operating System Principles.
ACM Press, New York, NY, 1995, pp. 237-250.

5) **R. Spurk:** *"Betriebssystempraxis 2004"*,
Spezialvorlesung, Universität des Saarlandes