Seminarausarbeitung zu MACH

im Proseminar Konzepte von Betriebssystemkomponenten

Katrin Giese
katti@andariel.informatik.uni-erlangen.de

13. Juli 2006

Kurzzusammenfassung

Dieses Dokument soll einen Überblick über Ziele und Design von Mach geben und speziell auf die Kommunikation und Speicherverwaltung in Mach eingehen.

1 Einführung

1.1 Monolithischer Kernel vs. Mikrokernel

Im Verlauf der Entwicklung wurden immer mehr Funktionen dem Betriebssystemkernel hinzugefügt. Dadurch entstanden immer größere und komplexere monolithische Kernel, wobei alle Kernel-Funktionen von überall aufgerufen werden können (siehe Abbildung 1). Sie besitzen keine Struktur und sind sehr stark fehleranfällig. Schon kleine Änderungen können zu unbeabsichtigten Nebenwirkungen an anderen Stellen führen. Zudem wird der Speicher, der vom Kernel belegt wird, immer gehalten, auch wenn er nicht gebraucht wird.

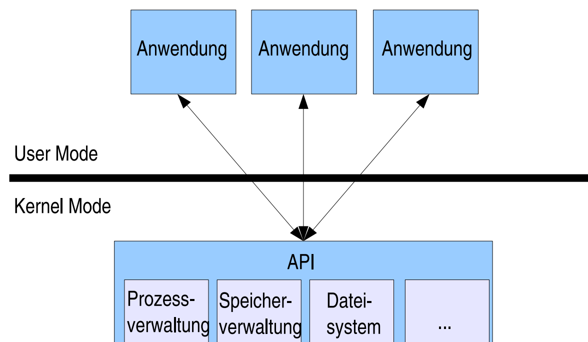

Abbildung 1: Monolithischer Kernel

Aufgrund der Fehleranfälligkeit und der sehr hohen Komplexität wurde ein neues Konzept entwickelt, der Mikrokernel. Das Konzept bestand darin, die Funktionen des Kernels zu reduzieren und somit minimal zu halten, sowie alle anderen Funktionen von anderen Programmen im Benuzteradressraum ausführen zu lassen.

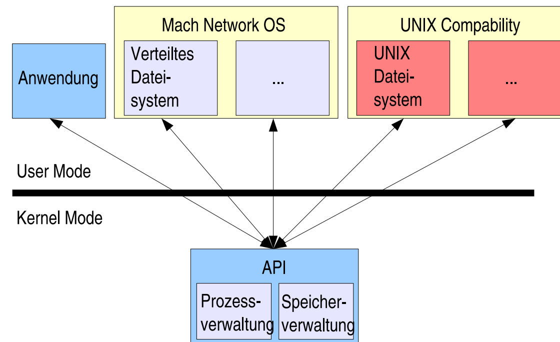

Abbildung 2: Mikrokernel-Architektur, um UNIX-Kompatibilität zu erreichen. Sämtliche „typischen“ Betriebssytem-Dienste werden als Userspace-Server implementiert.

Einer der ersten Mikrokernel ist Mach, dessen Wurzeln in RIG (Rochester Intelligent Gateway) liegen. Entwickelt wurde Mach von Richard Rashid, der ein Betriebssystem als Menge kooperierender Prozesse ansah.

1.2 Ziele

Es gab fünf wesentliche Ziele von Mach.

1. Bereitstellen einer Basis zur Entwicklung anderer Betriebssysteme (z.B. UNIX)
2. Unterstützung großer, spärlich bzw. wenig benutzter Adressräume
3. transparenter Zugriff auf Netzwerkbetriebsmittel
4. Ausnutzen von Parallelität sowohl im System als auch in den Anwendungen
5. Übertragbarkeit von Mach auf eine Vielzahl von Maschinen

2 Grundlagen

Mach wurde entwickelt um vorhandene Betriebssysteme zu emulieren, die durch eine Software-Schicht realisiert wurden, die aber im Benutzermodus liegen. Die Idee dazu war, verschiedene Mechanismen bereitzustellen, die für ein lauffähiges System benötigt werden. So wurden alle Funktionen, die ein Betriebssystem bereitstellt, zum Beispiel die Prozesse und die Dateiverwaltung, in die Benutzerebene verlagert. Dadurch beschränkten sich die Aufgaben des Mikrokernels auf Prozess- und Speicherverwaltung, sowie I/O-Dienste und Interprozesskommunikation. Um jenes zu realisieren stellt der Mikrokernel fünf grundlegende Konzepte zur Verfügung.

1. Prozesse
2. Threads
3. Ports
4. Nachrichten
5. Speicherobjekte

3 Mach

3.1 Prozesse

Die Prozesse in Mach bestehen aus einem Adressraum, Ports und einer Menge von Threads, wobei die Ports zur Kommunikation mit dem Kernel dienen. Anhand der Ports können Prozesse die Dienste des Kernels nutzen, indem sie Nachrichten an den Prozess-Port schicken. Diese Dienste besitzen Stellvertreter-Prozeduren in der Programmbibliothek, die wiederum aus einer Dienstdefinition durch den MIG (Mach Interface Generator) generiert werden. Somit werden weniger Systemaufrufe benötigt und deren Anzahl auf das Minimum gesenkt. Zusätzlich besitzen Prozesse eine Emulationsadresse, die dem Kernel mitteilt, an welcher Stelle sich das Emulationspaket im Adressraum des Prozesses befindet. Dies ist notwendig um die UNIX-Systemaufrufe emulieren zu können. Die Emulationsadressen werden nur einmal festgelegt. Danach werden sie nur an Kindprozesse weitervererbt. Im Gegensatz zu UNIX besitzen Prozesse in Mach keine uid, gid, Signalmaske und kein Wurzelverzeichnis. Ebenfalls haben sie kein Array von Dateideskriptoren.

Threads werden durch Register dargestellt, die den Instruction-Pointer (IP), Stackpointer und die Zustandsinformation enthalten. Threads können sich in in einem der Zustände „ausführbar“, „blockiert“ oder „rechnend“ befinden. Ausserdem besitzen sie auch Ports um thread-spezifische Kerndienste aufzurufen. Zusätzlich ist noch zu beachten, dass nur der Kernel die Adressräume kontrollieren kann, damit keine Inkonsistenzen auftreten können. Dies geschieht entweder durch Verdrängung und/oder über Kernel-Threads.

3.2 Kommunikation

In Mach gelangen die Nachrichten nicht direkt an den Empfänger. Sie werden immer an einen Port des Empfängers gesendet. Man kann einen Port als geschützten Briefkasten betrachten, der innerhalb des Kernels realisiert wird. Die Nachrichten, die an die Ports verschickt werden, werden in geordneten Listen gespeichert, deren Längen nicht fixiert sind.

Mach besitzt drei verschiedene Port-Typen. Der Bootstrap-Port initialisiert neue, leere Ports. Der Exception-Port leitet Fehlermeldungen an Prozesse weiter. Registrierte Ports dienen zur Kommunikation zwischen Prozessen und Standard-System-Servern, da ausgelagerte Funktionen auch auf dem Server liegen können, auf den diese Prozesse dann zugreifen können.

Damit Prozesse und Threads kommunizieren können, muss der Mikrokernel IPC unterstützen, die nach der klassischen Sender-Empfänger-Methode arbeiten. Dabei werden die Nachrichten über den Mikrokernel verschickt. Die Kommunikation in Mach ist unidirektional, das heißt an einem Port können mehrere Threads schreiben, aber nur einer lesen. Dabei ist zu beachten, dass der Datenaustausch asynchron ist. Das bedeutet, dass Nachrichten zuverlässig und sortiert sein müssen. Das heisst, Nachrichten kommen sicher und in der richtigen Reihenfolge beim Empfänger an und gehen nicht auf dem Transportweg verloren.

Nachrichten sind in Kopf- und Datenteil zerlegt. Der Kopf enthält alle Informationen für den Transport und Zustellung der Nachricht, wie die Größe der Nachricht, die den Zielport sowie den Antwortport, den Funktionscode und die Nachrichtenart. Dabei gibt die Größe die Gesamtgröße von Kopf- und Datenteil in Bytes an. Der Zielport bestimmt, wohin die Nachricht geschickt werden soll und der Antwortport gibt an, wohin eine eventuelle Antwort geschickt werden muss. Der Funktionscode und die Nachrichtenart sind nur für den Empfänger relevant. Der Datenteil besteht aus einem oder mehreren Datenobjekten, welche die Datenbeschreibung und die eigentlichen Datenelemente beinhalten. Unter Datenbeschreibung versteht man die Anzahl der Datenelemente sowie Größe der Daten in Bits und Typ der Datenelemente.

Meist wird der Funktionscode für RPC (remote procedure call) verwendet, um die auszuführende Funktion zu bestimmen. Der Empfänger stellt anhand der Größe der Nachricht einen Puffer bereit, in dem die empfangene Nachricht gespeichert wird.

Prozesse können anhand von RPC mit Hilfe von Nachrichten Funktionen von anderen Prozessen aufrufen. RPC wird auch vom Kernel verwendet. Dazu wird eine Definitionsdatei für den MIG angelegt. Aus der Definitionsdatei wird eine Unterroutine vom MIG erzeugt, der die RPCs in Form einer Nachricht an ein anderes Programm schickt. Dort angelangt werden diese Nachrichten von Routinen, die vom MIG erzeugt wurden, ausgewertet. Nun kann die enstprechende Funktion aufgerufen werden.

3.2.1 Nachrichtenversand über Netzwerk

Der Nachrichtenversand über Netzwerk verfolgt dieselben Sender-Empfänger-Prinzipien wie IPC, wobei das Versenden von Nachrichten nicht vom Kernel sondern vom Server erledigt wird. Der Kernel stellt die Nachrichten einem lokalen Nachrichtenserver zu, der daraufhin den Zielrechner sucht und die Nachricht für den weiteren Versand vorbereitet. Das heißt, er verpackt die Nachricht in ein Netzwerkprotokoll und wandelt, wenn notwendig, die Datentypen um. Ebenfalls verschlüsselt er die zu versendenden Daten. Nur der Nachrichtenserver ist für die Authentifizierung des Zielsystems zuständig, was jedoch abhängig vom Nachrichtenserver ist.

3.2.2 Beispiel

Wenn zum Beispiel Prozess X vom Rechner A über den Port Z auf Prozess Y von Rechner B zugreifen will, ist eine Sendecapability notwendig. Da aber die Capabilities lokal sind, existiert ein Proxyport auf dem Rechner A. Dadurch besitzt Prozess X die Sendecapability und der lokale Nachrichtenserver die Receivecapability. Zudem besitzt der Nachrichtenserver von Rechner B die Sendecapability und Prozess Y die Receivecapability auf dem Port Z. Ebenfalls ist eine globale eindeutige Nummer für den Zielport Z im Netzwerk gegeben. Diese Nummer nennt man auch Netzwerkname. Dadurch kann der Nachrichtenserver von Rechner A den Rechner B als Zielrechner bestimmen. Letztendlich nimmt der Rechner B den Netzwerknamen entgegen und schickt die Daten als Nachricht an den Zielport Z.

3.3 Speicher

Eine Besonderheit in Mach sind die Speicherobjekte. Sie stellen eine Datenstruktur oder eine Datei dar und werden im Adressraum eines Prozesses abgebildet, wobei sie eine oder mehrere Seiten belegen können. Speicherobjekte bilden die Basis für die Verwaltung des virtuellen Speichers in Mach. Falls ein Prozess auf eine Seite zugreift, auf dem er keinen Zugriff hat, tritt ein Seitenfehler auf. Diesen fängt der Kernel ab und kann ihn entweder selber behandeln oder er schickt eine Nachricht zu einem Server auf der Benutzerebene, der dann die fehlende Seite einlagert.

Die Speicherverwaltung in Mach ist in drei Komponenten unterteilt:

1. Das *pmap*-Modul wird im Kernel ausgeführt und beschäftigt sich mit der Verwaltung der MMU.

2. Der maschinen-unabhängige Kernelcode verwaltet die Seitenfehler sowie die Speicherfunktion *adress map* und ersetzt die fehlenden Seiten.

3. Der Speicherverwalter bzw. externe Pager wird im Benutzeradressraum ausgeführt und verwaltet den logischen Speicher und den Hintergrundspeicher.

Die Kommunikation zwischen dem Speicherverwalter und dem Kernel erfolgt über ein wohldefiniertes Protokoll, welches ermöglicht auch eigene Speicherverwalter zu realisieren. Die Vorteile darin sind zum einen die Anwendung von speziellen Ersetzungsstrategien für Systeme mit speziellen Anforderungen und zum anderen die Verkleinerung und Vereinfachung des Kernels. Jedoch kann es auch sein, dass die Speicherverwaltung durch eigene Speicherverwalter komplizierter wird, da der Kernel sich vor fehlerhaften oder böswilligen Speicherverwaltern schützen muss und der wechselseitige Ausschluss durchgesetzt werden muss, da sonst die Gefahr einer *race condition* entsteht.

Der virtuelle Speicher wird in Mach in Regionen (oder auch *regions* genannt) eingeteilt. Somit sind nur die Adressen gültig, die auch in einer Region vorhanden sind. Speicherobjekte werden auf eine unbenutzte Region des virtuellen Speichers abgebildet. Die Besonderheit hierbei ist, dass mit gewöhnlichen Maschineninstruktionen gelesen oder geschrieben werden kann. Sie stellen somit ein Schlüsselkonzept dar.

In einem gemeinsam benutzten Speicher sehen Threads innerhalb eines Prozesses den gleichen Adressraum. Sie können gemeinsam ein Speicherobjekt benutzen. In einem Multiprozessorsystem sind es eine Menge kooperierender Prozesse, die parallel auf verschiedenen CPUs zur Ausführung kommen, anstatt quasiparallel bzw. über time sharing auf eine CPU abzulaufen.

Ebenso wichtig ist die Erzeugung von Prozessen. Dabei muss aber der Kindprozess nicht identisch dem erzeugendem Prozess bzw. Prototyp sein. Da aber nicht jeder Kindprozess die Seiten des Vaterprozess benötigt, stellt Mach die Möglichkeit bereit, Vererbungsattribute für Regionen zu setzen.

1. Region wird im Kindprozess nicht benötigt
2. Region wird vom Prototyp und Kindprozess gemeinsam benutzt
3. Region im Kindprozess ist Kopie der Region des Prototyps

Wenn nun die Region des Kindprozesses eine Kopie der Region des Prototyps ist, verwendet Mach einen cleveren Trick, genannt copy-on-write. Dabei sind die Seiten der Region des Kindprozesses zuerst nur lesbar. Solange der Kindprozess nur lesend auf die Seiten zugreift, werden keine Seiten kopiert. Aber wenn es auf eine Seite schreibend zugreifen will, entsteht ein Schutzfehler. Daraufhin lagert das Betriebssystem eine Kopie der gewünschten Seite in den Adressraum des Kindprozesses. Dadurch wurde die lesbare Seite ersetzt. Somit werden auch wirklich nur die Seiten kopiert, auf dem der Kindprozess schreiben will. Aber durch copy-on-write ist die Verwaltung komplizierter und es werden mehrere Kern-Unterbrechungen (*Traps*) benötigt. Ebenfalls muss man unterscheiden, ob der Trap durch einen Seitenfehler oder durch copy-on-write verursacht wurde.

Damit die Speicherobjekte keinen Fehler verursachen, werden sie von externen Speicherverwaltern überwacht. Die Einbindung eines solchen Speicherobjekts in ein Adressraum erfolgt über das *mappen*. Daraufhin sendet der Prozess eine Nachricht an den Speicherverwalter und fordert damit den Speicherverwalter auf die Abbildung vorzunehmen. Dafür werden drei Ports zur Verfügung gestellt. Der Erste ist der Objekt-Port, der vom Speicherverwalter erzeugt wird und dient zur Informationsweitergabe (Seitenfehler und andere Ereignisse) an den Kernel. Der Zweite ist der Kontroll-Port, der vom Kernel erzeugt wird und der zum Senden von Antworten verwendet wird. Der Dritte Port ist der Namens-Port, der ebenfalls vom Kernel erzeugt wird und zur Objektidentifizierung dient. Gleichzeitig gibt er noch den Speicherbereich an. Objekte werden hierbei über den Speicherverwalter eingelagert. Der Speicherverwalter liefert eine Capability für den Objekt-Port. Der Kernel erzeugt daraufhin den Kontroll- und den Namens-Port und sendet eine initiale Nachricht an den Objekt-Port. Diese

Nachricht teilt dem Speicherverwalter mir, dass der Kernel nun die beiden anderen Ports angelegt hat. Der Speicherverwalter sendet daraufhin eine Nachricht mit Daten über den Objekt-Port zurück. Nun sind alle Seiten vorhanden, die initial nicht les- und schreibbar sind. Das bedeutet, wenn ein Thread auf eine Seite eines Objekts zugreifen will, tritt ein Seitenfehler auf und ein Trap wird ausgelöst. Der Kernel teilt dem Speicherverwalter über eine Nachricht an den Objekt-Port mit, dass er die Seite, auf die der Thread zugreifen wollte, bereitstellen muss. Der Speicherverwalter überprüft zuerst, ob der Zugriff überhaupt rechtmäßig ist. Wenn nicht, schickt der Speicherverwalter eine Fehlermeldung zurück. Ansonsten fügt er die Seite in den Adressraum ein. Der Kernel bekommt einen Zeiger auf die Seite und weiß somit, dass die Seite nun verfügbar ist. Er blendet nun die Seite in den Adressraum des Threads ein, der auf die Seite zugreifen wollte. Der Thread kann dann entblockiert werden.

Um aber sicherzustellen, dass ständig genügend freie Seitenrahmen zur Verfügung stehen, werden sie anhand eines Kernel-Threads überprüft, der Paging-Dämon genannt wird. Dabei wird der Paging-Dämon von Zeit zu Zeit geweckt. Wenn nun nicht genügend freie Seitenrahmen vorhanden sind, werden alte veränderte („*dirty*“) Seiten zum Speicherverwalter des Objekts gesendet. Der Speicherverwalter schreibt dann die Seiten auf die Festplatte und teilt es dem Kernel mit, sobald er fertig ist. Als Ersetzungsalgorithmus verwendet Mach die *second chance* - Strategie.

4 Zusammenfassung

Die Vorteile von Mach sind zum einen, dass fast alles im Benutzeradressraum liegt und nur der Mikrokernel an sich maschinen-abhängig ist. Durch seine extreme Flexibilität ist es sogar möglich Operationen für den Echtzeitbetrieb hinzuzufügen. Ebenfalls kann man mehrere Betriebssysteme gleichzeitig auf dem Mikrokernel laufen lassen. Auch der Schutz zwischen Prozessen ist gewährleistet. Doch leider schneidet Mach in Bezug auf Performanz nicht sehr gut ab, was aber nicht Ziel von Richard Rashid war. Der Mikrokernel muss sehr oft zwischen Kernel- und Benutzermodus umschalten. Und die Kontrolle über Prozesse bewirken häufigere Threadwechsel und somit auch Adressraumwechsel, was den Betrieb verlangsamt.

Die bekanntesten Anwendungen von Mach3.0 sind HURD und Darwin, wobei HURD im GNU-System als Kernel fungiert, aber nicht stabil genug ist, so dass man es nicht in Produktivsystemen einsetzen kann. Darwin hingegen ist kompatibel zu 4.4BSD und Kernel von MacOS X.

Ebenfalls kann man Mach in Sprite und Mike finden. Sprite war ein verteiltes Betriebssystem, das zu Forschungszwecken entwickelt wurde. Mike hingegen ist eine Art Laufzeitumgebung für verteilte Software, die in einer C++-ähnlichen Syntax geschrieben werden kann.

5 Literaturverzeichnis

Literatur

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian an M. Young. Mach: A New Kernel Foundation for UNIX Development. USENIX Association, June 1986.

[2] A. Tanenbaum. Moderne Betriebssysteme. Hanser Verlag. 1994

[3] http://www.usenix.org/publications/library/proceedings/mach3/bernadat.html

[4] http://www.usenix.org/publications/library/proceedings/mach3/castro.html

[5] http://www.usenix.org/publications/library/proceedings/mach3/kupfer.html