Konzepte von Betriebssystemkomponenten
Unix

Christian Kollee
(sichkoll@cip.informatik.uni-erlangen.de)

23.10.2002

1 Der Kernel

Auf Grund seiner ursprünglichen Entwicklungsziele (klein und effizient) wurde UNIX mit einem monolithischen[1] Kern (siehe Abbildung 1) entwickelt. Die Vorteile die sich aus diesem Design ergaben, waren einerseits eine effiziente Kommunikation innerhalb des Kernes, andererseits war jeder Bestandteil unmittelbar in der Lage priviliegierte Befehle auszuführen.

Im Laufe der Zeit wuchs der Kern, durch Implementierung neuer Treiber (Netzwerk, div. Gerätetreiber, etc.), immer stärker an und die Nachteile eines monolithischen Kernels wurden sichtbar:

- Fehlende Modularität, d.h. jede noch so kleine Änderung am Kern erfordert eine Übersetzung und einen Neustart des Rechners. Fehler können dabei katastrophale Folgen (Kernel panic) mit sich ziehen.
- Keine intere Struktur, d.h. es existieren zwischen den einzelnen Kernelkomponenten keinerlei Schutzmechanismen, die einen unbefugten Zugriff verhindern.

Es wurde versucht die fehlende Modularität durch die Einführung ladbarer Kernelmodule (z.B. für Gerätetreiber) zu simulieren. Dies wiederum führt zu neuen Problemen (fehlerhafte oder korrupierte Module), welche auf den fehlenden internen Abstraktionsschichten basieren.

Mittlerweile existieren auch UNIX-Betriebssysteme welche nicht mehr dem monolitischen Ansatz folgen, sondern auf einer Microkernelarchitektur basieren (MINIX, GNU/Hurd).

[1] Das heißt, daß alle Funkionen des Betriebssystems (z.B. Dateisystem, Speicherverwaltung, etc.) in diesem Kern implementiert werden.

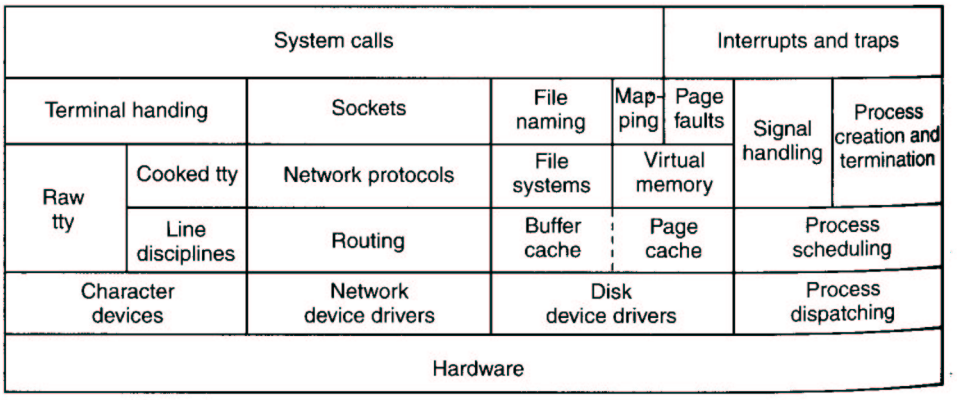

Abbildung 1: Kernelstruktur von BSD4.4 [2]

2 Prozesse

2.1 Aufbau eines Prozesses

Das Verhalten jedes Prozesses unter UNIX wird durch die in Tabelle 1 enthaltenen Bestandteile definiert. Die Textsection wird bei der Übersetzung generiert (Compiler+Linker), Daten Section und Stack erst, wenn das Programm in den Speicher geladen wird.

2.2 Erzeugung von Prozessen

Erzeugt werden Prozesse unter UNIX mittels `fork`-System Call. Dabei wird ein neuer Speicherbereich belegt, in dem exakte Kopien der Textsection, Datasection und des Stacks angelegt werden. Von diesem Moment an arbeiten beide Prozesse nahezu autark voneinander weiter. Verändert also einer der Beiden seine Variablen oder geht in einen anderen Zustand über, so bemerkt der andere Prozess nichts von diesen Vorgängen (Ausnahme: Der Vaterprozess wartet auf die Beendigung seinens Kindes).

2.3 Process Identifier

Bei seiner Erzeugung bekommt jeder Prozess eine eindeutigen Process IDentifier (PID) zugewiesen. Dabei handelt es sich um einen Zeiger in eine Tabelle mit allen Prozessen, die sich derzeit im System befinden. Immer dann, wenn ein Prozess sich innerhalb eines System Calls auf einen anderen Prozess bezieht, wird die PID des Zielprozessses übergeben.

Section (Segment)	Beschreibung
Text	Eigentlicher Programmcode. Ein Zeiger (Program Counter) auf den Addressbereich enthält dabei den nächsten auszuführenden Befehl.
Data	Globale Variablen
Stack	Temporäre Daten

Tabelle 1: Prozessbestandteile

2.4 Prozesskommunikation

Kommunikation zwischen Prozessen kann unter UNIX auf unterschiedlichen Wegen erfolgen:

1. Pipes: Dabei wird ein Kanal zwischen den beiden kommunizierenden Prozessen geschaffen, in den einer der Beiden schreiben und der Zweite drauas lesen kann.

2. Signals: Prozesse können Mitgliedern Ihrer eigenen Prozessgruppe[2] Signale (Software Interrupts) schicken. Jeder Prozess kann entscheiden, wie er auf ein solches Signal reagiert. Er kann es ignorieren, abfangen[3] oder durch den Prozess beendet werden. Soll das Signal abgefangen werden, wird ein *signal handler* aufgerufen. Nach der Ausführung kehrt das Programm an die Stelle des Aufrufes zurück. POSIX nennt 12 Signale, die jedes UNIX zur Verfügung stellt (z.B. SIGKILL, SIGQUIT, u.a.).

3. Shared Memory: Das Konzept des **shared memory** kann in UNIX nicht direkt eingesetzt werden. Jedoch besteht die Möglichkeit über einKonzept der Speicherverwaltung diese Kommunikationsart nachzubilden.

3 Speicherverwaltung

3.1 Shared Text Segments

Da UNIX als Multiuser-System entwickelt wurde, bestand von Anfang an die Möglichkkeit, daß zwei Benutzer das gleiche Programm zur gleichen Zeit ausführen. Um hier eine Redundanz der identischen Textsegmente zu vermeiden, wird es nur einmal in den physikalischen Speicher geladen und daraufhin über virtuelle Adressen der jeweiligen Prozesse gemappt.

[2] Väterprozesse und deren Vorgänger, Geschwisterprozesse und Kindprozesse gehören zur selben Prozessgruppe

[3] SIGKILL kann weder ignoriert noch abgefangen werden

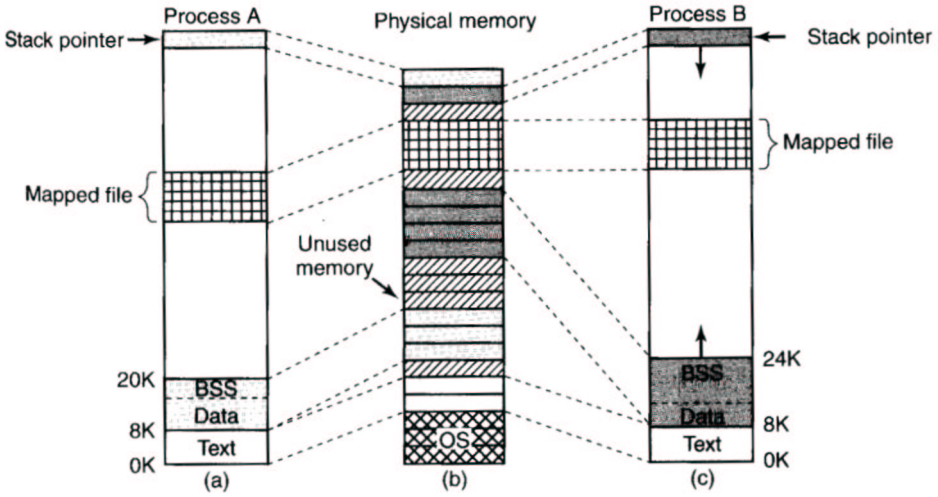

Abbildung 2: Speicherverwaltungskonzepte [2]

3.2 Memory-Mapped-Files

Viele Versionen von UNIX gestatten es, Dateien auf einen Teil des Adressraumes eines Proesses abzubilden. Damit vereinfacht sich einerseits der Zugriff auf die Datei (es kann wie auf normalen Speicher zugegriffen werden), andererseits können mehrere Prozesse die gleiche Datei mappen und so das Konzept des *shared memory* simulieren.

4 Input/Output

UNIX I/O Devices (Festplatten, Drucker, Netzwerk...) werden als **special files** unter */dev* angelegt. Dabei werden (normalerweise) die gleichen System Calls benutzt, welche auch zum Zugriff auf gewöhnliche Dateien benutzt werden.

Man unterscheidet zwei Arten von special files:

1. Blockdevices sind Geräte, welche einen direkten Zugriff auf alle Blöcke ermöglichen. Somit kann z.B. direkt auf Block 12 zugegriffen werden, ohne erst die Blöcke 0 bis 11 gelesen zu haben (Festplatte, Diskette).

2. Characterdevices werden für Geräte benutzt, die einen konstanten Zeichenstrom produzieren bzw. akzeptieren. Beispiele: Maus, Tastatur.

Alle Einträge unter /dev sind mit einem Gerätetreiber verknüpft, welcher das entsprechende Gerät verwaltet. Dabei bekommt jeder Treiber eine sogenannte Major Device Number zugwiesen. Unterstützt ein Treiber mehrere

Geräte (z.B. zwei Festplatten), so erhält jedes noch eine Minor Device Number. Über Major und Minor Device Number ist jedes Gerät eines Rechners eindeutig gekennzeichnet.

Literatur

[1] Silberschatz, Galvin, Gagne. Operating System Concepts, John Wiley & Sons, Inc., 6. ed. 2002

[2] Tanenbaum, Andrew. Modern Operating Systems, Prentice Hall, 2. ed. 2001

[3] Nutt, Gary. Operating Systems - A Modern Perspective, Addison-Wesley, 2. ed. 2002

[4] Dr. Hauck, Franz. Systemprogrammierung I - Folien, IMMD4, WS 2001/02