Zusammenfassung "Betriebssysteme" bei Prof. Dr. Gheis (WS 1999/2000)

Michael Jaeger

8.10.2000

1 Aufbau von Betriebssystemen

1.1 Aufgaben eines Betriebssystems

Die Hauptaufgaben eines Betriebssystems sind

- Programmausführung (laden, starten, beenden)
- Zuteilung der Ressourcen (CPU, Speicher, Drucker), unter Beachtung der Fairness und Vermeidung von Konflikten und Verklemmungen
- Ein- und Ausgabe zur Peripherie (Koordination und Verbergen)
- Kommunikation mit anderen Rechnern (Daten, E-Mail, Browser)
- Zugriffsschutz (Verhinderung unerlaubter Zugriffe)
- Abrechnung (Erfassung+Aufbereitung des Ressourcenverbrauchs)

In der Entwicklung von Betriebssystemen spricht man von verschiedene Generationen:

1. **single user** Der Rechner wird exklusiv von nur einem Benutzer belegt.
2. **batch Betrieb** Die Verarbeitung wird in Aufträge aufgeteilt, die einzeln der Reihe nach abgearbeitet werden (keine Interaktion).
3. **mulitprogramming** Es können mehrere Programme (quasi-)gleichzeitig ausgeführt werden. Hier wurde bereits ein Augenmerk auf die Protabilität gerichtet.
3. **timesharing** Das Betriebssystem kann zwischen mehreren Prozessen umschalten. Dies ermöglicht aus Benutzersicht eine Quasi-Parallelität.
3. **multiuser** Mehrerer Benutzer können (quasi-)gleichzeitig an einem Rechner arbeiten - ermöglicht durch multiprogramming. Es wird zwischen dem Benutzer und dem Kern unterschieden.
4. **client-server** Aufteilung des Programms in Darstellung und Verarbeitung - kann auf verschiedene Rechner umgelegt werden.
4. **Verteilte Systeme** Das gesamte Betriebssystem wird auf mehrere Rechner verteilt. Zur Unterstützung einer derartigen Aufteilung wird sog. middleware benötigt.

1.2 Strukturen

Monolithisch Unterteilung Kernel-/User-Mode (Übergang durch System Calls), keine klare Struktur

System Call spezieller Aufrufmechanismus zum Schutz des BS-Kerns, BS-spezifisch, z.B. für Prozeßverwaltung, Schutzmechanismen, Statusabfragen, Interprozesskommunikation etc.

Schichten / Ringe strukturierte Aufteilung nach funktionalen Gesichtspunkten, bestimmen Schutzsphären

Mikrokerne minimale Funktionen im BS-Kern, BS-Funktionen laufen als Anwendungen auf dem Mikrokern (auch Dateisystem und Terminals), IPC über Nachrichten, klare Struktur, leicht verteilbar

Verteiltes Betriebssystem und Netzbetriebssystem Kommunikation über Rechnernetz, Knoten nur bedingt autark

1.3 Prozesse

Prozeß Programm (Einheit für das BS) in Ausführungsphase ($\rightarrow$ Benutzer), belegt Prozessor ($\rightarrow$ Betriebssystem), Prozeßkontrollblock ($\rightarrow$ Systemprogrammierer), Basismodell für die Ausführung von Aktionen in einem Rechnersystem, Sammlung von Code, Daten, Register, Stackzeiger, Instruktionszeiger etc.

Prozeßkontrollblock (PCB) Datenstruktur mit notwendigen Informationen zu einem Prozeß: Prozeßnr., Programmzähler, program status word, verbrauchte Zeit, Erzeugungszeitpunkt, Zeiger auf Nachrichten, Textsegmentzeiger, Datensegmentzeiger, Stackzeiger, aktuelles Verzeichnis, Signalstatus, Register etc.

Pozeßtabelle Liste der Prozesse im BS mit Zeiger auf PCBs

program status word (PSW) zeigt den Zustand eines laufenden Prozesses an, Instruktionszeiger, zugänglich im Debugging-Modus

Speicherbereiche eines Prozesses Programmtext, Programm-Daten, Stack, Laufdaten (BS).

Adressräume Der Adreßraum eines Prozesses enthält den Programmtext (Code), statische Daten, dynamische Daten und den Stack, wobei die dynamischen Daten nach oben und der Stack un unten „wächst"´.

Zustände eines Prozesses neu $\rightarrow$ bereit $\rightarrow$ aktiv $\rightarrow$ terminiert bzw. aktiv $\rightarrow$ bereit mittels Unterbrechung und aktiv $\rightarrow$ wartend $\rightarrow$ bereit mittels Blockierung

fork $\leftrightarrow$ exec `fork` erzeugt eine identische Kopie des aufrufenden Prozesses mit neuem Adressraum, während `exec` einen Prozeß erzeugt, der in demselben Adressraum läuft.

Threads Threads in einem Prozeß arbeiten auf dem gleichen Adressraum, haben aber eigene Register, Stack, Instruktionszähler etc. und verursachen damit nicht den Aufwand normaler Prozesse.

Kernel-level Threads sind dem BS bekannt und können wie Prozesse verwaltet werden

User-level Threads werden im user space von sog. Thread-Bibliotheken realisiert und sind dem BS unbekannt (nur der umgebende Prozeß ist bekannt)

2 Prozeß-Scheduling

Um die quasi-parallele Ausführung von Prozessen zu ermöglichen, schaltet ein sog. scheduler zwischen den Prozessen hin und her. Dabei sollte sein Vorgehen fair, effizient (was die Prozessorleistung angeht), die Wartezeit minimierend und den Durchsatz maximierend sein.

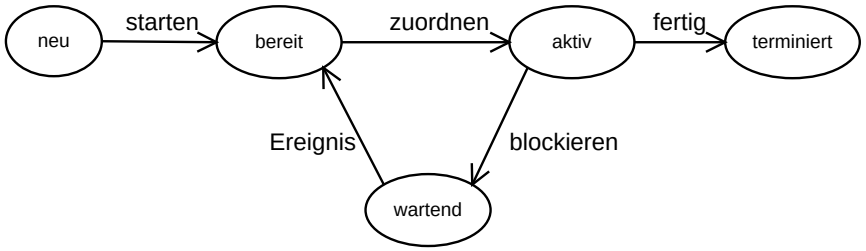
In Batch-Systemen ist z.B. der Durchsatz maximal, die hohen Verweilzeiten der Prozesse aber ziemlich schlecht. Im Gegensatz dazu werden beim time-sharing die Verweilzeiten optimiert. Damit nun der Scheduler zwischen den Prozessen umschalten kann, erzeugt die CPU in regelmäßigen Abständen eine Unterbrechung, so dass das BS die Kontrolle erhält und einen Prozesswechsel vornehmen kann. Die Strategien für den Wechsel von Prozessen sind vielfältig und werden nun genauer besprochen.

2.1 Scheduling

Scheduling Steuerung der Abarbeitungsreihenfolge der Prozesse, Zuordnung der Prozesse zur CPU

Strategien Fairness (kein „Verhungern"´), wenig Aufwand für Scheduling, Lastbalancierung der Ressourcen, gracefully degrading (bei schwerer Last), Möglichkeit, Prioritäten zu vergeben

Modellierung deterministische / stochastische Modelle → rechnergestützte Simulation, Messung an existierenden Systemen



2.2 Deterministische Modellierung

Gegeben sind N Tasks und M Prozessoren. Gesucht ist eine Zuordnung der N Tasks zu den M Prozessoren (z.B. ein Schedule mit kürzester Ausführungszeit).

A-Schedule Optimales Schedule für die Gesamtausführungszeit, nicht unterbrechend.
Nummeriert alle Tasks mit gegebenem Nummerierungsalg. und ordnet einen Task einem Prozessor zu, falls dieser frei wird und alle Vorgänger ausgeführt sind und er die höchste verbliebene Nummer trägt. Der Nummerierungsalg. geht dabei Bottom-up vor.
Ist nicht optimal für unterschiedliche Ausführungszeiten.

B-Schedule Optimal für den Spezialfall: G hat Baumstruktur, gleiche Ausführungszeiten, nicht unterbrechend, M beliebig.
Nummeriert alle Tasks mit ihrer Ebene im Präzedenzbaum von der Wurzel aus. Wenn ein Prozessor frei wird, odne den Task zu, für den alle Vorgänger ausgeführt sind und der die höchste verbliebene Nummer trägt.

Prioritätslisten Scheduling hängt ab von Prioritäten, die den Tasks zugeordnet werden

Preemptive Scheduling Prozesse können beliebig unterbrochen und fortgesetzt werden

Processor Sharing Alle lauffähigen Prozesse erhalten $\frac{1}{N}$ der Kapazität

Variablen und Konstanten der deterministischen Modellierung:

- r_i = Verweilzeit von T_i
- r_{max} = maximale Verweilzeit aller T_i (=$t(S)$)
- r = mittlere Verweilzeit aller T_i
- N = Anzahl der Tasks im Tasksystem
- n = mittlere Anzahl nicht beendeter Tasks
- $< r[j] >$ = monoton wachsende Folge der Verweilzeit ($r[0] = 0$)

Es gilt $\frac{n}{N} = \frac{r}{r_{max}}$.

Shortest Processing Time First (SPTF) optimal hinsichtlich der mittleren Verweilzeit

2.3 Stochastische Modellierung

Es werden die zufälligen Schwankungen bei Ankünften und Abarbeitungszeiten der Tasks modelliert (Warteschlangentheorie). Dabei sind folgende Parameter von Bedeutung:

$Prob(x > t) = e^{-\lambda t}$ Wahrscheinlichkeit, dass der Abstand zwischen zwei Ankünften grßser als t ist

$E[x] = \frac{1}{\lambda}$ Mittelwert der Zeit zwischen zwei Ankünften

λ Mittlere Anzahl der Ankünfte; mittlere Bedienrate (Bsp.: 4 Kunden pro Stunde, Bearbeitungszeit im Schnitt 12 min $\Rightarrow \frac{1}{\mu} = 12min \Rightarrow \mu = 5$)

μ Mittlere Anzahl der möglichen Ankünfte; mittlere Ankunftsrate

$Prob(N \geq n) = \rho^n$ Wahrscheinlichkeit, dass mindestens n Benutzer im System sind

$\frac{1}{\mu}$ Mittlere Bedienzeit

$\rho = \frac{\lambda}{\mu}$ Mittelwert der Poissonverteilung; Auslastung des Systems; Intensität

$E[x] = \frac{1}{\lambda}$ Mittelwert der Zeit zwischen zwei Ankünften

$E[N] = \frac{\rho}{1-\rho}$ Mittlere Anzahl der Kunden im System

$E[Q] = E[N] - \rho$ Mittlere Anzahl der Kunden in der Warteschlange

$E[S] = \frac{1}{\mu}$ Mittlere Bedienzeit

$E[W] = E[N] \cdot E[S] = \frac{\rho}{1-\rho} \cdot \frac{1}{\mu}$ Mittlere Wartezeit

$E[T] = E[W] + E[S] = \frac{1}{\mu}(\frac{\rho}{1-\rho} + 1) = \frac{1}{\mu(1-\rho)}$ Zeit, bis der Prozeß abgefertigt ist

$Prob(x > t) = Prob(\#(t) = 0) = e^{-\lambda t}$ Abstand zwischen zwei Ankünften

$Prob(x \leq t) = 1 - \lambda t$ Verteilung der Zwischenankunftszeiten für den Poissonprozess

$Prob(R \leq t | x \geq y) = \frac{Prob(R \leq t, x \geq y)}{Prob(x \geq y)}$ Ankunftsprozess, bei dem wir bereits y Zeiteinheiten gewartet haben - wie ist die Wartezeit verteilt?

Geometrische Verteilung Prob(A tritt beim k-ten Versuch ein)=$p_k = p(1-p)^{k-1}$

2.3.1 M|M|1-Systeme

Unter der Markoffeigenschaft versteht man, dass „Die Übergangswahrscheinlichkeit $p_{i,j}$ durch die Zustände i, j eindeutig beschrieben ist. Sie läßt sich als eine Menge von Zuständen Z und der Übergangsmatrix Q darstellen."Die Übergangswahrscheinlichkeit $p_{i,j}$ ist also nicht von der Vergangenheit sondern nur vom aktuellen Zustand abhängig.

Jede Wahr.vert., für die gilt, dass $P(X_n = i) = P(X_0 = i) = p_i$ ist, heißt „stationäre Verteilung". In diesem Fall legt die Anfangsverteilung die Verteilung der Kette selbst fest. Damit eine Markovkette einen stationären Zustand besitzt, muss sie irreduzibel, aperiodisch und rekurrant sein.

Beim M|M|1-Modell ist λ die mittlere Ankunftsrate, μ die mittlere Bedienrate, die Zischenankunftszeiten und die Bedienzeiten sind negativ exponentialverteilt mit den Mittelwerten $\frac{1}{\lambda}$ und $\frac{1}{\mu}$, die Auslastung beträgt ρ.

$$A(t) = \begin{cases} 0, & t \leq 0 \\ 1 - e^{-\lambda t}, & t>0 \end{cases} \quad , \quad B(t) = \begin{cases} 0, & t \leq 0 \\ 1 - e^{-\mu t}, & t > 0 \end{cases}$$

Es gilt weiterhin

- $\rho = \frac{\lambda}{\mu} \Rightarrow p_n = \rho^n(1-\rho)$
- $E[N] = \frac{\rho}{1-\rho}$, $Var[N] = \frac{\rho}{(1-\rho)^2}$
- $E[Q] = \frac{\rho^2}{1-\rho}$, $Var[Q] = \frac{\rho^2(1+\rho-\rho^2)}{(1-\rho)^2}$

3 Prozeß-Scheduling

3.1 Kritische Abschnitte

Beim Multitasking greifen konkurrierende Prozesse möglicherweise störend auf gemeinsame Ressourcen zu. Als Lösung kann man KA einführen, so dass Prozesse einen exklusiven Zugriff auf gemeinsamen Speicher haben → mutual exclusion.

Eine Lösung muss folgenden Anforderungen genügen:

1. höchstens ein Prozeß im KA
2. funktioniert für beliebig viele Prozesse
3. kein Blockieren anderer Prozesse ausserhalb des KA (mit beliebigen Ausführgeschwindigkeiten)
4. jeder Prozeß muss irgendwann einmal in den KA eintreten

Als **race-conditions** werden dabei zeitkritische Abläufe bezeichnet.

Lösungsmöglichkeiten für kritische Abschnitte sind:

3.1.1 Busy Waiting

Algorithmus Mit Sperrvariable S oder mittels streng alternierendem Zugriff oder durch Setzen von Flaggen. Das Sperren von Unterbrechungen sollte allerdings nur dem Kernel möglich sein. Lamports „Bakery Algorithmus"´ funktioniert mit n Prozessen und einem int-array `wählt[1:N]=0`, in dem sich ein Prozeß eintragen kann. Dabei werden alle die Prozesse vorgelassen, die eine kleinere Nummer haben.

Algorithmus von Dekker verwendet die Variablen `p1bereit, p2bereit, favorit`. Wenn ein Prozeß bereit ist, jedoch der Favorit der andere ist, so wird er auf `nicht bereit` gesetzt und wartet (busy), bis der Prozeß selber der Favorit ist. Dann wird er wieder auf `bereit` gesetzt und der KA ausgeführt.
Diese Methode ist korrekt!

Wechselseitiger Ausschluss (Peterson) Mit `aktiv, p1bereit, p2bereit` wird in einer while-Schleife überprüft, ob der andere Prozeß bereit ist und der eigene Prozeß aktiv → warten.
In der Hardware wird der wechselseitige Ausschluss mit der Operation `TSL` unterstützt.
Diese Methode ist korrekt!

Da busy waiting Ressourcen verschwendet, kann es zu unerwarteten Effekten kommen. So kann z.B. ein Prozeß mit hoher Priorität beim Warten einen Prozeß mit niedriger Priorität im KA ausbremsen (Prioritätsinvertierung).

3.1.2 Sperrvariablen

Das Problem beim Einsatz von Sperrvariablen ist, das es zu Fehlern beim Synchronisieren kommen kann: P_1 liest V_1 (Wert 0) → P_2 setzt $V_1 = 1$ → P_1 und P_2 im KA!

3.1.3 Schlafen und Wecken

Zum Beispiel beim Erzeuger-Verbraucher-Problem: Wenn der Puffer voll ist, legt sich der Produzent schlafen und wird vom Verbaucher geweckt, falls wieder Platz ist. Auch hier kann es aber zu Problemen aufgrund von KA kommen.

3.1.4 Semaphore

Spezielle Integer Variablen, die mit einer Prozeßwarteschlange und zwei unteilbaren Operationen $P()$ und $V()$ verbunden sind:
$P(S)$: IF $S > 0$ THEN $S = S - 1$ ELSE warten;
$V(S)$: IF Prozesse in Warteschlange THEN einen aufwecken ELSE $S = S + 1$;
Ein binärer Semaphor kann nur die Werte 0 und 1 aufnehmen (Mutex)

3.1.5 Monitore

Da die Programmierung mit Semaphoren u.U. sehr schwierig ist, wurden Monitore eingeführt. Ein Monitor ist

- ein Konstrukt einer Programmiersprache
- eine Menge von Prozeduren und Daten, die in einem speziellen Modul gekapselt sind
- Prozesse rufen Prozeduren im Monitor auf

- nur höchstens ein Prozeß kann in einem Monitor akiv sein

Vor- und Nachteile sind:

\+ vereinfachte Programmierung

\+ unterstützt "‘information hiding"´

\- Frage: Verletzt SIGNAL die Monitor-Bedingung? Binch-Hansen: SIGNAL() muss letzte Anweisung des Monitors sein; Hoare: aufgeweckter Prozeß läuft, aufweckender blockiert sofort.

Man kann Monitore aber auch mit Semaphoren programmieren. So hat jeder Monitor

- SEMAPHORE mutex=1 // regelt den exklusiven Zugriff
- SEMAPHORE next=0 // für Prozeß der SINGAL() sagt und deshalb blockieren muss
- INTEGER next-anz=0 // zählt Prozesse an "‘next"´

Ein Benutzerprozess muss dann folgendermassen erweitert werden:

- P(mutex) // exklusiver Zugriff
- Proc() // eigentliche Prozedur
- IF next-anz > 0 THEN V(next) ELSE V(mutex) // Prozeß von innen oder aussen

Die Prozeduren SIGNAL und WAIT müssen auch noch neu geschrieben werden. Dies wird durch die Variablen

- SEMAPHORE x-sem $= 0$ // am Anfang keine Prozesse
- INTEGER c-anz $= 0$ // Zähler für wartende Prozesse

3.1.6 Pfadausdrücke

Erlauben eine genauere Steuerung der Reihenfolge und Parallelität der Prozesse und sind in Programmiersprachen integriert. Die Notation ist wie folgt:

P1 , P2 paralleles Arbeiten
Mit Semaphoren: $< L\ P_1, P_2\ R > \rightarrow < L\ P_1\ R > < L\ P_2\ R >$

P1 ; P2 sequentielles Arbeiten
Mit Semaphoren: $< L\ P_1; P_2\ R > \rightarrow < L\ P_1 \vee (S_1) > < P(S_1) P_2\ R >$ mit $s_1 := 0$

n : (P1) Einschränkung: höchstens n aktive Inkarnationen von P1
Mit Semaphoren: $< L\ n : (P_1)\ R > \rightarrow < P(S_2)\ L\ P_1 R\ V(S_2)$ mit $s_2 := n$

[P1] Aufhebung einer Einschränkung
Mit Semaphoren $< L\ [P_1]\ R > \rightarrow < PP(c, s, L)\ P_1\ VV(c, s, R) >$

Auch Pfadausdrücke können mit Hilfe von Semaphoren realisiert werden.

3.1.7 Deadlock

Beim Zugriff auf Ressourcen unterscheidet man zwischen dem preemptable und dem non-preemptable Zugriff und zwischen dem shared access und der mutual exclusion.
Die 4 Bedingungen, die alle erfüllt sein müssen, damit es zu einer Deadlock-Situation kommt sind:

1. **Wechselseitiger Ausschluss:** Jedes Betriebsmittel wird entweder von genau einem Prozeß belegt oder ist verfügbar → Lösung: Ressourcen gemeinsam nutzbar machen
2. **Belegungs- und Wartebedingung:** Ein Prozeß kann während er bereits Betriebsmittel belegt weitere anfordern → Lösung: alle Ressourcen müssen vor Start belegt sein
3. **Ununterbrechbarkeit:** Betriebsmittel, die von einem Prozeß belegt werden können nicht entzogen werden, sondern müssen explizit vom belegenden Prozeß wieder freigegeben werden → Lösung: es muss erlaubt sein, einem Prozeß Ressourcen zu entziehen
4. **Zyklische Wartebedingung:** Es muss eine zyklische Kette aus zwei oder mehr Prozessen existieren, so dass jeder Prozeß ein Betriebsmittel anfordert, das ein anderer belegt → Lösung: entweder ein Prozeß darf immer nur eine Ressource belegen, oder man nummeriert die Ressourcen und befriedigt Anforderungen nur in numerisch aufsteigender Reihenfolge

Um einen Deadlock vermeiden zu können, muss vor der Zuteilung geprüft werden, ob ein Deadlock entstehen kann. Mann kann z.B. eine Art von Ressource betrachten, von der N Instanzen vorhanden sind. Prozesse fordern Ressourcen an und das BS entscheidet über die Zuteilung. Ein **sicherer Zustand** ist dabei, wenn es eine mögliche Zuteilungsreihenfolge gibt, um alle Prozesse zufriedenstellen zu können. Daraus folgt:

3.1.8 Banker's Algorithmus

Der Algorithmus prüft vor dem Zuteilen einer Ressource alle möglichen Nachfolgezustände, und teilt die Ressource nur dann zu, wenn es eine Möglichkeit gibt, alle Anforderungen zu erfüllen. Eine Zuteilung wird also nur dann vorgenommen, wenn ein sicherer Zustand entsteht.

Im Banker's Algorithmus werden folgende Variablen verwendet:

M #Prozesse

N #Ressourcentypen

$E[e_j]$ #verfügbare Ressourcen pro Typ

$B[b_j]$ #belegte Ressourcen pro Typ

$A[a_j]$ #freie Ressourcen pro Typ

$C[c_{ij}]$ #zugeteilte Ressourcen pro Typ und Prozeß

$R[r_{ij}]$ #noch erforderlichen Ressourcen pro Typ und Prozeß

Nach Definition gilt

$$A \leq B \Leftrightarrow \forall j : a_j \leq b_j$$

Damit kann man den Banker's Algorithmus wie folgt formulieren: Sei $s := s_0$

1. Suche Zeile Z in R mit $Z \leq A$.
   Falls keine solche Zeile existiert, dann kann kein Prozeß beendet werden (s ist unsicher, da ein Deadlock möglich ist).

2. Nimm an, Prozeß der Zeile Z erhält alles, was er fordert und terminiert. Markiere diesen Prozeß und simuliere Zuteilung, s :=Folgezustand.

3. Wiederhole Schritte 1. und 2. bis alle markiert sind (d.h. s_0 ist sicher) oder Deadlock-Gefahr entsteht (d.h. s_0 ist unsicher).

Die Schwächen des Banker's Algorithmus sind die Annahmen, dass nur eine feste Anzahl von Ressourcen und Prozessen vorhanden sind, dass jeder Prozeß seine Maximalanforderungen a-priori kennt, und dass seine Laufzeit sehr schlecht ist.

4 Speicherverwaltung

4.1 Swapping

Das Verschieben von Prozessen auf Platte und umgekehrt heisst Swapping. Dabei wird der Hauptspeicher in Einheiten fester Länge (Blöcke) eingeteilt und in einer Bitmap vermerkt, ob ein Block belegt ist. Dabei stellen sich Designfragen wie "'Wie gross sind die Allokationseinheiten zu wählen?"', "'Wie groß ist die Bitmap?"' und "'Wie gross ist der Suchaufwand?"'.

Die Speicherverwaltung wird mit Hilfe von verketteten Listen realisiert. Bei der Auswahl des freien Speichers werden folgende Strategien verwendet: First Fit, Next Fit, Best Fit und Worst Fit. Dadurch kommt es zu Fragmentierungen (im Schnitt die Hälfte des allokierten Speichers).

Eine Form des Swappings ist das sog. **Buddy System**, bei dem Lücken die Grösse von Zweier-Potenzen haben und systematisch angepasst werden.

4.2 Virtueller Speicher

Ziel ist die Entkopplung der Programme vom physisch vorhandenem realen Hauptspeicher. Dabei werden immer nur die Seiten des virtuellen Adressraums eingelagert, die auch wirklich benötigt werden. In diesem Zusammenhang sind Segmente vom Benutzer definierte lineare Speicherbereiche und Pages die aus Sicht des BS zu verwaltenden Speicherblöcke. Der Hauptspeicher ist aufgeteilt in Seitenrahmen, wobei ein Seitenrahmen genau eine Seite speichert.

4.2.1 Paging

Paging ist ein Speicherverwaltungskonzept für Betriebssysteme mit Einheiten fester Länge. Die Seiten korrespondieren nicht mit der logischen Struktur von Prozessen. Die korrespondierenden Einheiten zu den Seiten im physikalischen Speicher heissen Seitenrahmen und haben dieselbe Grösse wie die Seiten. Da Programme vollständig im virtuellen Adressraum arbeiten, muss jede virtuelle Adresse von der MMU in eine physikalische Adresse umgewandelt werden. In einer Seitentabelle wird dabei gespeichert, welche Seiten sich gerade wo im Hauptspeicher befinden. Bei einem page fault, der entsteht, wenn eine angeforderte Seite nicht im Hauptspeicher verfügbar ist, muss eine eingelagerte Seite ausgelagert werden, damit Platz für die neue Seite entstehen kann.

4.2.2 Segmente

Segmente stellen zusammenhängende Adressbereiche mit variabler Grösse dar und sind dann von Vorteil, wenn ein Prozeß mehrere getrennt Adressräume benötigt, die unabhängig voneinander beliebig wachsen und schrumpfen können. Ausserdem können Prozeduren in eigenen Segmenten neu compiliert werden, ohne das ganze System neu zu compilieren, wenn sie in einem eigenen Segment an Adresse 0 liegen. Dies ermöglicht z.B. sog. shared libraries. Ein Programmierer muss sich aber anders als beim Paging der angewandten Technik bewusst sein.

Segmente sind Behälter für logisch zusammengehörige Informationen und bilden eine Einheit bzgl. des Zugriffsschutzes. Bei Swapping werden sie als Ganzes ein- bzw. ausgelagert. Sie werden in einer Segmenttabelle eines Prozesses verwaltet, wobei eine Adresse aus einer Segmentnummer S und einem Offset d besteht. Sie können geschützt werden und lassen sich von mehreren Prozessen gemeinsam nutzen. Oft wird eine Kombination von Segmentierung und Paging eingesetzt (dabei werden einzelne Seiten von Segmenten ausgelagert): Prozeßtabelle $\rightarrow$ Segmenttabelle $\rightarrow$ Seitentablle $\rightarrow$ realer Speicher. Beim gemeinsamen Nutzen von Daten und Prozeduren werden die Zeiger von der Segmenttabelle auf die Seitentabelle auf die geiche Tabelle gebogen.
Die Memory Management Unit (MMU) unterstützt das BS hardwareseitig bei der Verwaltung des realen und virtuellen Speichers. Dabei geht es vor allem um die Umrechnung der virtuellen Adresse in reale.

Ist der virtuelle Hauptspeicher sehr groß, so werden mehrstufige Seitentabellen eingesetzt, wo Unterabschnitte auch ausgelagert werden können.

Beim Workingset Modell werden in den verschiedenen Phasen der Ausführung eines Prozesses immer eine Untermenge aller möglichen Seiten referenziert. Dieses Workingset sollte möglichst ganz im Speicher gehalten werden.

4.2.3 Vergleich Paging$\leftrightarrow$Segmentierung

Gesichtspunkt	Paging	Segmentierung
Muss sich der Programmierer der Technik bewusst sein?	Nein	Ja
Anzahl der benutzten linearen Adressräume	1	Viele
Kann er gesamte Adressraum die Grösse des physikalischen Speichers übersteigen?	Ja	Ja
Können Prozeduren und Daten auseinandergehalten werden?	Nein	Ja
Können Tabellen, deren Grösse fluktuiert, leicht angepasst werden?	Nein	Ja
Ist die gemeinsame Benutzung von Prozeduren unterhalb von Benutzern möglich?	Nein	Ja
Warum wurde die Technik eingeführt?	Um grösseren Adressraum ohne zusätzliche Kosten für physikalischen Speicher zu erhalten	Um die Trennung zwischen Programmen und Daten in logische unabhängige Adressräume zu erlauben und um die gemeinsame Benutzung und den Schutz zu unterstützen

4.2.4 Seitenersetzungsstrategien

Optimaler Alg. Ersetze diejenige Seite, die am längsten nicht mehr benötigt wird

Zufälliger Alg. Ersetzte eine beliebige Seite

FIFO Ersetze die Seite, die am längsten im Speicher ist.
Bei der FIFO-Anomalie von Belady kann es vorkommen, dass die Anzahl der Seitenfehler trotz grösserem Speicher steigt.

LRU Ersetze diejenige Seite, die am längsten nicht gebraucht wurde.

Second Chance (LRU Annäherung mit FIFO) Benutzt das Referenziert-Bit der Einträge in der Seitentabelle. Arbeitet wie FIFO, nur dass bei einem Referenziert Bit mit Wert 1 selbiges erst auf 0 gesetzt wird, und dann weitergegangen wird.

Seitenersetzungen hängen ab von (ω, A, M) mit:

- ω: Referenzfolge eines Prozesses
- A: Seitenersetzungsalgorithmus
- M: Speichergrösse
- N: Anzahl der referenzierbaren Seiten

Als Stack-Algorithmen gelten LRU, Optimal etc. FIFO ist kein Stack-Alg.

5 Dateisysteme

Das BS legt die innere Struktur der Dateien, den Namensraum, die Ordnungs- und Gruppierungsmöglichkeiten, die Zugriffsformen, Schutzvorrichtungen etc. fest.

Unter Unix gibt es vier verschiedene Dateitypen:

regular files Benutzerdaten (ASCII, binary)

directory files Strukturinformationen für das Dateisystem

character special files modellieren serielles E/A-Gerät

block special files modellieren E/A-Geräte im Block-Modus

Eine Binäre Datei besteht in Unix aus dem Kopf (Informationen zu Textgrösse, Datengrösse, Entry Point, Grösse der Symboltabelle, Flags etc.) und den Relocation Bits, die beim Positionieren im Arbeitsspeicher verwendet werden.

5.1 Plattenverwaltung

Es stellen sich beim Entwurf die Frage "'Interne Fragmentierung $\leftrightarrow$ Verwaltungsaufwand"´ bei der Blockgrösse, "'verkettete Listen (direkter Zugriff) $\leftrightarrow$ Bitmaps (kompakter)"´ bei der Freispeicherverwaltung.

5.2 UNIX-Dateisystem

Jede Datei wird durch einen eigenen I-Node verwaltet. Dieser enthält die Dateiattribute und Zeiger auf die Lokation der zugehörigen Dateiblöcke. Ein I-Node ist wie folgt aufgebaut:

Mode	Typ und Zugangsrechte
File Owner	Eigentümer
Group	Gruppe des Eigentümers
Timestamp 1	...gelesen
Timestamp 2	...geschrieben
Timestamp 3	I-Node verändert
Size	Grösse in Byte
Block Count	Anzahl der belegten Blöcke
Reference Count	Anzahl der Verweise
Zeiger auf Datenblöcke	

Verzeichnisse werden ebenfalls über I-Nodes verwaltet. Der Verzeichniseintrag einer Datei besteht aus der Nummer des zugehörigen I-Nodes und dem Names als ASCII-Zeichenkette.
Bei der Implementierungs wird die Platte in Blöcke fester Länge aufgeteilt. Der Boot Block (Nummer 0) enthält den Start-Up-Code und die Initialisierungen. Der Super Block (Nr. 1) enthält Informationen zur Gestaltung des Dateisystems, die Anzahl der I-Nodes, Grösse der Platte in Blöckenm, Freelist etc. Die I-Nodes werden aufsteigend ab 1 nummeriert und sind jeweils 64 Bytes lang. Datenblöcke sind konsekutiv für eine Datei.

Boot	Super	I-Nodes	Daten	Daten	...

Die Tabelle mit den I-Nodes befindet sich im Hauptspeicher. I-Node 1 wird zur Verwaltung fehlerhafter Blöcke verwendet. I-Node 2 zeigt per Konvention auf das root-Verzeichnis. Ausserdem wird eine Tabelle der geöffneten Dateien verwaltet. Zusätzlich hat jeder Prozeß eine Tabelle mit Dateideskriptoren.