Systemprogrammierung

24.10.2000

Literatur:

- Skript i4
- Silberschatz, Gelani: Operating System Concepts, Adison-Wesley, 1994
- Tannenbaum: Operation Systems – Design and Implementation
- Per Brinch Hansen: Operating System Principles, 1973'

Gliederung:

1. Einführung OK
2. Prozess-Management-Gundlagen OK
3. Prozess – Systeme
4. Bedienungssysteme -
5. Prozess-Synchronisation
6. Deadlocks
7. Memoy Management OK
8. Virtual Memory / Paging OK
9. File-System OK
10. Networking

SYSTEMPROGRAMMIERUNG 1

LITERATUR: 1
GLIEDERUNG: 1

1. EINFÜHRUNG 4

1.1. BEGRIFF DES BERIEBSSYSTEMS 4
1.2. SPEICHERHIERARCHIE 4
1.3. I/O-OPERATIONEN 4
1.3.1. 5
1.3.2. SPEICHERSTRUKTUREN 5
1.4. AUFBAU EINES BETRIEBSSYSTEMS 5
1.4.1. KOMPONENTEN EINES BETRIEBSSYSTEMS 5
1.4.2. SYSTEMAUFRUFE 5
1.4.3. SYSTEMPROGRAMME 6
1.5. BEISPIELARCHITEKTUREN 6
1.5.1. MS-DOS 6
1.5.2. UNIX 6
1.5.3. OS/2 6
1.5.4. SCHICHTENKONZEPTE 6

2. PROZESSVERWALTUNG 7

2.1. ZUM PROZESSBEGRIFF 7
2.2. SCHEDULING 7
2.3. INTERPROZESSKOMMUNIKATION 7

3. NEBENLÄUFIGE PROZESSE SYNCHRONISIEREN & KOORDINIEREN 8

3.1. ERZEUGER-VERBRAUCHER-PROBLEM 8
3.1.1. LÖSUNG: RINGPUFFER 8
3.1.2. ALLGEMEINER LÖSUNGSANSATZ UND NEUE SCHWIERIGKEITEN 8
3.2. DAS PROBLEM DES WECHSELSEITIGEN AUSSCHLUSSES 8
3.2.1. 2 UNTAUGLICHE VERSIONEN UND EINE LÖSUNG 8
3.2.2. DER BAKERY-ALGORITHMUS 8
3.2.3. ENQUEUE UND DEQUEUE (...) 9
3.2.4. SYNCHRONISATIONSMECHANISMEN MIT ATOMAREN OPERATIONEN 9
3.3. SEMAPHORE 9
3.3.1. DAS KONZEPT EINES SEMAPHORS: 9
3.3.2. ERZEUGER-VERBRAUCHER-PROBLEM 9

4. BEDIENUNGSSYSTEME (V) 11

5. CPU – SCHEDULING 12

5.1. GRUNDLEGENDES 12
5.2. SCHEDULING-STRATEGIEN 12
5.2.1. FIFO UND LIFO 12
5.2.2. SJF, SRPT 12
5.2.3. SEPT UND SERPT 12
5.2.4. PRIORITY SCHEDULING 13
5.2.5. RR (ROUND-ROBIN) UND PS (PROCESSOR-SHARING) 13
5.2.6. MULTILEVEL FEEDBACK QUEUEING 13

6. HAUPTSPEICHERVERWALTUNG 14

6.1. SPEICHERORGANISATION 14
6.1.1. SPEICHERHIERARCHIE 14
6.1.2. VIRTUELLER SPEICHER 14
6.2. SEGMENIERUNG 14
6.2.1. DAS PRINZIP DER SEGMENTIERUNG 14
6.2.2. SEGMENTIERUNGSSTRATEGIEN 14
6.3. BUDDY-SYSTEME 14
6.3.1. EINFACHE BUDDY-SYSTEME 14
6.3.2. GEWICHTETE BUDDY-SYSTEME: 14
6.4. DEMAND-PAGING 14
6.4.1. PRINZIP DES PAGINGS 14
6.4.2. DEMAND-PAGING-STRATEGIEN: 15
6.5. NICHT-DEMAND-PAGING 15
6.5.1. OBL (ONE-BLOCK-LOOK-AHEAD) DEMAND-PAGING-VERSION: 15
6.5.2. OBL ALS LOOK-AHEAD-VARIANTE: 15
6.6. DISKUSSION DER PAGING-ALGORITHMEN 15
6.6.1. KOSTEN VON PAGING-ALGORITHMEN 15
6.6.2. FIFO – ANOMALIE: (?) 15
6.6.3. STACK-ALGORITHMEN: (?) 15
6.6.4. PRIORITÄTSALGORITHMEN (?) 15

7. 16

8. DAS DATEISYSTEM 16

8.1. ALLGEMEINES ZUM DATEIKONZEPT 16
8.2. VERZEICHNISSTRUKTUR 16
8.2.1. SINGLE-LEVEL-VERZEICHNIS 16
8.2.2. TWO-LEVEL-VERZEICHNIS 16
8.2.3. VERZEICHNISBÄUME 16
8.2.4. AZYKLISCHER GRAPH: 16
8.3. ZUR IMPLEMENTIERUNG EINES DATEISYSTEMS 16
8.4. BELEGUNGSSTRATEGIEN 17
8.4.1. ZUSAMMENHÄNGENDE BELEGUNG 17
8.4.2. VERKETTETE BELEGUNG 17
8.4.3. INDIZIERTE BELEGUNG 17
8.5. SPEICHERPLATZVERWALTUNG 17
8.5.1. LINEARE LISTE 17
8.5.2. HASH-TABELLE: 17

1. Einführung

1.1. Begriff des Beriebssystems

Betriebssystem: Operating System = Programm, dass die Kontrolle und effektive Nutzung eines Rechners ermöglicht.

1.2. Speicherhierarchie

Die sog. Hierarchie bietet einen guten Kompromiss zwischen Kosten und Zugriffszeit.

Register - schnell
Cache
Haptspeicher
Platten - langsam

Typ Workstaion Okt. 1998: PII, 450 Mhz,

	Zeit	Kapazität [byte]	Zeit	Kapazität [Byte]
Register	10ns	256-1024	2.2ns	128
Prim.Cache	10ns	16-32k	2.2ns	32k
Sec.Cache	20ns	64k-1m	2.2ns	512k
RAM	34-100 ns	16M-128M	20ns	128M-64G
HD	16-50ms	100M-1G	10ms	10G

1.3. I/O-Operationen

I/O: Input/Output
Ereignisse werden der CPU mittels **Interrupt** signalisiert. Kann über Hardware oder Software erfolgen:
Bei einem Interrput stoppt die CPU den aktuellen Prozess, speichet den Zustand (Program Counte, Register, Files offen, ...). Danach macht die CPU an der gleichen Stelle weiter.

Aublauf einer I/O-Operation:

1. **Benutzerprozess** fodert I/O-Operation über einen System Call (**Interrput**) an.
2. System Call unterbricht den Benutzerprozess und gibt die **Kontrolle an das OS**, das die entsprechenden **I/O-Routine** startet. Diese I/O-Routine lädt die notwendigen Werte in die Register und Puffer des **Controllers** und startet die I/O-Operation des Controllers.
3. Datentransfer findet statt:
   CPU währenddessen:
   - wartet: snychronous I/O
   - Kontrolle zurück an OS: asynchronous I/O
4. Durch **Interrupt** meldet das I/O-Device das Ende des I/O.
5. Das OS ruft eine entsprechende Interrupt-Routine auf, und setzt den entsprechenden Benutzerprozess fort.

I/O-Operation ohne DMA:
I/O mit DMA: “direct memory access”: Datentransfer weitgehend ohne CPU.

1.3.1.

1.3.2. Speicherstrukturen

RAM: muss komplettes Programm aufnehmen. ABER:
- langsam
- flüchtig
- klein

Speicherhierarchie:
- Register
- Cache
- RAM
- Harddisk
- CD/RW
- DAT

1.4. Aufbau eines Betriebssystems

1.4.1. Komponenten eines Betriebssystems

Prozessverwaltung:
- create / kill
- Scheduling (CPU-Zeit verteilen)
- Synchronisieren
- Deadlocks beheben
- Interprozess-Kommunikation

Hauptspeicherverwaltung

Sekundärspeicherverwaltung
- Disk Scheduling (gleichzeitig mehrere schreiben)
- Zuteilung freie Bereiche

Dateiverwaltung

Kommandointerpreter

1.4.2. Systemaufrufe

Prozesse kommunizieren mit dem Betriebssystem.

Prozesskontrolle
Dateimanipulation
Gerätemanipulation
Kommuniaktion
Abruf von Informationen

1.4.3. Systemprogramme

Erweitern Funktionalität des OS (Editor, Command Shell, Compiler, ...)

1.5. Beispielarchitekturen

1.5.1. MS-DOS

- kein SUPERVISOR-Mode (jeder Prozess hat gleiche Rechte wie OS)
- kein Multitasking

1.5.2. Unix

besser. User → system-call → kernel → hardware

1.5.3. OS/2

etwa wie Unix

1.5.4. Schichtenkonzepte

hardwarenah ↔ andwendernah

Schichten austauschbar! (Modulkonzept)

T.H.E. : Hardware → CPU → RAM → Drivers → I/O → User
Venus: Hardware → Interpreter → CPU → ‘I/O → Virt.RAM → Drives → User

Schichen auch im Netzwerk!

ISO/OSI – Modell: 7 Schicjten:

- 7 Aussage
- 6 Übersetzung
- 5 Sprechen ?
- 4 Transport Aussage
- 3 Transport Sätze
- 2 Transport Wort
- 1 Transport Buchstabe

2. Prozessverwaltung

2.1. Zum Prozessbegriff

„Programm in der Ausführung“ (dynamisch)
Problem: 2 Prozesse wollen auf 1 Betriebsmittel zugreifen.

Nur 1 Prozess kann zu einem Zeitpunkt laufen, aber MULTITASKING.
möglicher Status eines Prozesses:

running:	läuft auf CPU.
ready:	wartet auf CPU
waiting	wartet auf Event (z.B. eigenbelegtes Gerät)
blocked	wartet, dass er ein Gerät benutzen darf
new	create()
killed	kill()
terminated	normal fertig.

PCB. Process Control Block: hat alle Infos über einen Prozess:

- **Status** (s.o.)
- **Programmzähler** (aktueller Befehl)
- **CPU-Scheduling** (Priority, Zeiger auf queues)
- **Speichermanagement** (letzte Registerwerte)
- **E/A-Status** (offene Geräte und Files)
- **Accounting** (Prozessnummer, needed time, ZeitBeschränkungen, …)

2.2. Scheduling

Single-Processing: nur 1 Prozess.
Multi-Processing: Prozess muss Kontrolle per Interrput abgeben.
Mutli-Tasking: OS verwaltet Kontrolle.

2.3. Interprozesskommunikation

- message-passing
- shared memory (schneller, aber kann inkonsistent werden!)

3. nebenläufige Prozesse synchronisieren & koordinieren

3.1. Erzeuger-Verbraucher-Problem

Prozess V (Verbraucher) ist auf eine Ausgabe von Prozess E (Erzeuger) angewiesen.

Lager mit begrenzer Kapazität. E legt ab (wenn nicht voll) , V entnimmt (wenn nicht leer).
PROBLEM: Inkonsistenz, wenn V etwas entnimmt, wenn E noch nicht den Bestand aktualisiert hat.

3.1.1. Lösung: Ringpuffer

IN/OUT per Modulo.
IN:=IN+1 MOD Max;
OUT:=OUT+1 MOD Max;

wenn in=out → leer,
wenn in+1 MOD max = out → voll. (letzter Platz darf NIE belegt werden!)

3.1.2. Allgemeiner Lösungsansatz und neue Schwierigkeiten

wir führen noch einen „Counter“ ein, der die Anzahl der Objekte im Ringpuffer darstellt.
Problem aber wie vorher: bevor der Counter upgedated wurde, greift ein 2. Prozess „dazwischen“ darauf zu, bevor der neue Wert des „counters“ aus dem Register zurückgeschrieben wurde.

3.2. Das Problem des wechselseitigen Ausschlusses

Programmteile, in denen globale Daten benutzt werden, als „kritisch“ markieren. Nun darf nur 1 Programm gleichzeitig sich in einem kritischen Bereich aufhalten!

3.2.1. 2 Untaugliche Versionen und eine Lösung

Wichtige Keiterien:

Mutual Exclusion: nur 1 Prozess im kritischen Bereich

Progress, : einer will rein →nur die, die auch rein wollen, werden gefragt.
Bounded Waiting : Prozess warten maxmal x Zeit.

3.2.2. Der Bakery-Algorithmus

2 Arrays mit allen Prozess-ID’s als Kastenbeschriftungen: einmal choosing mit TRUE/FALSE, einmal number mit „Warte-Nummern“.

Prozess 290 will jetzt in den kritischen Bereich.
Dafür setzt er choosing[290] auf TRUE, number[290]eins höher als die höchste vergebene Nummer bisher, und setzt choosing[290] zurück.

Nun probieren will ALLE choosings[] (also die Warteschlangen) durch, und machen jeweils erst dann weiter, wenn der aktuelle Prozess, den wir gerade betrachten, keine Wartenschleifenposition mehr vor unserem Prozess hat.
Haben wir dann alle Prozesse durchgetestet, so dass keiner mehr vor uns ist, dürfen wir loslegen.
Am Ende setzen wir dann unsere eigene Nummer wieder auf 0 ! (dann können die anderen wartenden Prozesse weitermachen).

3.2.3. Enqueue und Dequeue (...)

Array mit allen möglichen Prozessnummern. In A[0] steht die Nummer des aktuell laufenden Prozesses (z.B. 290), in A[290] steht dann ddie Nächste Prozess-ID (z.B. 5), etc.
A[0] wird entsprechend immer aktualisiert (von 290 auf 5, etc.)

3.2.4. Synchronisationsmechanismen mit atomaren Operationen

Möglichkeit 1: währen atomarem Befehl einfach Möglichkeit des Interrupts deaktivieren.
Möglichkeit 2: Test-And-Set: Prozedur: gib zuerst Eingabe zurück, setze sie dann auf True.
Möglichkeit 3: Swap: ????

3.3. Semaphore

3.3.1. Das Konzept eines Semaphors:

Wie im Parkhaus: (hier mit 1 Parkplatz): globale Variable. Will jemand rein, setzt er sie 1 runter. Wenn er rausgeht, 1 hoch.
Wenn sie 0 ist, darf niemand rein.

„busy waiting“: wenn ein Prozess in den Bereich möchte, wird er schlafengelegt und seine ID festgehalten. Er wird erst wieder geweckt, wenn er dann an der Reihe ist.

Init(S,1)	S := 1
Wait(S)	Sobald S größer 0, erniedrige S um 1 und mache DANN erst weiter!
Signal(S)	S := S+1

Dabei muss ggf. jeweils die Warteschlange aktualisiert werden.

Vor dem kritischen Bereich ist dann jeweils Wait(S) auszuführen, danach Signal(S).
Weitere Möglichkeit: Init(S,0), vor X: Wait(S), nach Y: Signal(S). → Y vor X ausgeführt.

3.3.2. Erzeuger-Verbraucher-Problem

N legt in Lager, M entnimmt aus Lager.

S = Zugriffs-Semaphor
F = volle Plätze
C = leere Plätze
F+C = Lagerkapazität (bleibt konstant)

Erzeuger:
WAIT(C): Erniedrige C um 1. Wenn vorher 0, warte.
WAIT(S), Einfügen, SIGNAL (S): dediziertes Einfügen.
SIGNAL(F): volle Plätze um 1 erhöhen.
Verbraucher: genau andersherum.

3.3.3. Reader-Writer-Probleme

z.B. Datenbanken: Schreiben nur exklusiv, lesen auch gleichzeitig möglich.

Lösung, dass keiner den anderen blockieren kann
Möchte ein Writer in den KB, so kommt kein neuer Reader in die Warteschlange. Haben alle Reader ihre Arbeit erledigt, so sind die Writer dran. Will jetzt ein Reader in den KB, kommt er nach Ende dieses WRITER-Prozesses an die Reihe.

Funktionsweise:

Nehmen wir an, „read“ dauert ziemlich lange. Der „W“-Semaphor ist nur dafür da, sicherzustellen, dass die Operationen vor und nach dem Leseprozess atomar erfolgen.
Jeder Leseprozess, der anfängt, erhöht nun n um 1. Der erste setzt zusätzlich „S“ auf 0.
Jeder Leseprozess, der aufhört, erniedrigt n um 1, der letzte setzt „S“ wieder auf 1.

Der Schreibprozess kommt nun erst an die Reihe, wenn S auf 1 sitzt.

..... ???

3.3.4. Das Fünf-Philosophen-Problem

4. Bedienungssysteme (V)

5. CPU – Scheduling

5.1. Grundlegendes

1 Prozess „running“, andere „waiting“. → welcher kommt nun dran?

- fair (nicht zu selten, nicht zu oft
- Priorities
- hoher CPU – Durchsatz
- möglichst kurze Response-Zeiten

alles das soll zusammenspielen!

5.2. Scheduling-Strategien

5.2.1. FIFO und LIFO

FIFO: „First Come, First Served“.
LIFO: neuester Prozess zuerst.

PROBLEM: haben wir einen langen und 2 kurze Prozesse, wäre es besser, die 2 kurzen zuerst drankommen zu lassen, da so die mittlerer Wartezeit verkleinert wird. Diese berechnet sich so:

Wir packen alle Prozesse hintereinander, und schauen uns die Zeiten nach jedem Prozess an (bis auf „nach dem letzten“ , da dieser ja da völlig alleine laufen kann).
Alle diese Zeiten teilen wir durch die Anzahl der Prozesse. → mittlere Wartezeit.

Im Durchschnitt ist dies aber egal.

VARIANZ = Durchschnitt der Quadrate der Abweichungen vom Mittelwet.
Ist hier bei LIFO wesentlich kleiner.

5.2.2. SJF, SRPT

SJF = shortest job first. Wir betrachten nur ganze Jobs!
Jobs unterteilen= „preemptives Multitasking“

SPT = shortest-remaining-time-first: jobs werden unterbrochen; der Job mit der kürzesten Restzeit kommt zuerst.

5.2.3. SEPT und SERPT

Shortest-Expected-Processing-Time-First
Shortest-Expected-Remaining-Processing-Time-First

Woher weiss man, wie lange der hob dauert?
Möglichkeit 1: Durchschnitt der letzten n jobs.
Möglichkeit 2: ältere Jobs weniger werten:

$\tau_{n+1} = a*\tau_n + (1-a)*\tau_n$

Wenn a=1 ist, wird nur der vorherige Prozess gewertet.
Wenn a=0,4 ist, gilt die länge des vorherigen Prozesses zu 60%, und alle vorherigen zu 40%.

Exkurs: digitale Sprachübertragung: Prediction der folgenden Waves.

5.2.4. Priority Scheduling

Prozesse hoher Priorität IMMER zuerst.
(auch preemptiv, d.h. mit Unterbrechung von lo-pri-jobs), möglich.
innerhalb der Klasse arbeiten wir mit FIFO (ältestes Job zuerst).

5.2.5. RR (Round-Robin) und PS (Processor-Sharing)

Idee: Lange Jobs auch "zerstückeln" in "Quanten": 1 Quantum ausführen, dann Job wieder hinten anstellen.

kleine Quanten → fast wie Processor Sharing (alles gleichzeitig)
große Quanten → fast wie FIFO (alle hintereinander in Warteschlange)

5.2.6. Multilevel Feedback Queueing

Prioritätsklassen: hoch = kleines Quantum, neidrig = großes Quantum. Prozess hangelt sich nun von oben bis unten runter; kurze Prozesse sind dafür schnell fertig.

(auf Deutsch: die „aufgeschobenen" Prozesse kommen – dann mit größerer Zeitscheibe – erst wieder dran, wenn gerade keine neuen Jobs kommen.)

6. Hauptspeicherverwaltung

6.1. Speicherorganisation

verschiedene Speicherarten

6.1.1. Speicherhierarchie

Cache → RAM → HD

6.1.2. Virtueller Speicher

klar.

6.2. Segmenierung

6.2.1. Das Prinzip der Segmentierung

Segmentnummer + Länge. → in Speicher.
Problem: alle Segmente völlig unterschiedlich groß. Daher:

6.2.2. Segmentierungsstrategien

oben: beste Strategien:

- Rotating First Fit (erste Lücke ab aktueller Pos.)
- First Fit (erste Lücke):
- Best Fit (beste Lücke)
- Worst Foit (größte Lücke)

Problem: Fragmentierung → Garbage Collection.

6.3. Buddy-Systeme

6.3.1. Einfache Buddy-Systeme

Idee: nur Segmente der Größe 2^i verwenden.
Wenn nun ein kleineres gebraucht wird, teile größeres auf.
Wenn eins frei wird, versuche mit benachbartem zusammenzulegen.

6.3.2. Gewichtete Buddy-Systeme:

auch die Aufteilen 3:1 zulassen. (2^r und $3*2^r$)

6.4. Demand-Paging

6.4.1. Prinzip des Pagings

Nur Speicherbereiche einheitlicher Größe
(im RAM: „Frames“, auf HD: „Seiten“) (gleich groß!!!)

Strategien:

- Demand-Paging: lade Seite, wenn benötigt (Seite fehlt = „Seitenfehler“)
- Demand-**Pre**-Paging: lade x folgende Seiten gleich mit.
- Look Ahead: vorausschauen, was vor. dmenächste benötigt wird.

6.4.2. Demand-Paging-Strategien:

Ziel: große Hit-Ratio (passende Seiten schon im RAM)

FIFO: älteste eingelegte Seite verdrängen.
LRU: (last recently used): älteste benutzte verdrängen
SECOND CHANCE: (spart Aufwand): bei 2. Aufruf BIT seten. Dann erst mal alle Seiten mit nicht gesetztem Bit auslagren (nach FIFO), wenn keine mehr da sind eben alle Bits löschen.
LFU: (last frequently used): niedrigste Nutzungshäufigkeit
(seit immer / seit h Zugriffen / seit letztem Seitenfehler)
CLIMB (Aufstieg bei Bewährung): Seite „steigt“ bei Aufruf 1 höher. Tiefste zuerst auslagern.
RANDOM: zufällig auslagern.
OPT: im Programmcode schauen, welche Page voraussichtlich in Zukunft am längsten nicht gebraucht werden wird (natürlich nicht immer möglich).

6.5. Nicht-Demand-Paging

Einpagen, was in der Nähe ist

6.5.1. OBL (One-Block-Look-Ahead) Demand-Paging-Version:

Seite im Speicher → an erste Position kicken.
Seite nicht im Speicher: lade sie an Pos.1, checke folgende:
wenn nicht im RAM, lade sie an letzte Pos.

6.5.2. OBL als Look-Ahead-Variante:

lade Seite i+1 auch nach, wenn Seite i bereits im Speicher ist.

6.6. Diskussion der Paging-Algorithmen

6.6.1. Kosten von Paging-Algorithmen

Von Speicherzustand S_n auf Zustand S_{n+1}
X_t: Anzahl nachgeladener Seiten, Y_t: Anzahl verdrängter Seiten-
Kosten für Verdrängung Proportional zu Kosten für Nachladung.
Nicht-Demand-Paging logischerweise immer schlechter als Demand-Paging.

Mehr Speicher heißt nicht immer besseres PAGING-Verhalten, z.B. bei FIFO nicht:

6.6.2. FIFO – Anomalie: (?)

??

6.6.3. Stack-Algorithmen: (?)

?? (neueste zuerst → oben auf Stack)

6.6.4. Prioritätsalgorithmen (?)

??
LRU, OPT, LIFO, LFU.
Jeder PA ist auch ein SA.

7. ...

8. Das Dateisystem

8.1. Allgemeines zum Dateikonzept

logische Anordnung von Daten.

DATEI: Name, Extension, Größe, Pos., Rechte, Date, Benutzer.
Alles in VERZEICHNISSTRUKTUR.

Operationen: lesen, schreiben, Zeiger Pos.,
Zusätzlich: Daten anhängen, umbenennen, kopieren.

Datei öffnen, schließen (für Zugriff) : → Open-File-Table.

Zugriff: Sequentieller, direkter Zugriff. (besser für große Dateien).

8.2. Verzeichnisstruktur

Partitionen, Verzeichnisse.

8.2.1. Single-Level-Verzeichnis

8.2.2. Two-Level-verzeichnis

Benutzer\Benutzerverzeichnis

8.2.3. Verzeichnisbäume

ROOT
aktuelles Verzeichnis
absoluter Pfad, relativer Pfad
del, deltree.

8.2.4. Azyklischer Graph:

1 Verzeichnis an mehreren Stellen (z.B. usern) „gemounted“. Zyklen sind zu vermeiden! ☺

8.3. Zur Implementierung eines Dateisystems

Harddisk: Block→Sektor (32 – 4096 Bytes)
Schichtensystem, von unten nach oben:
E/A-Kontrolle: HD → RAM.
Basic File System: Blöcke lesen / schreiben.
Dateiorganisationsmodul: Logische Dateiblöcke → Blöcke auf HD.
Logisches Dateisystem: Verzeichnisse, Dateinamen,

mounten (einhängen) von Dateisystemen nötig.
öffnen, schließen von Dateien nötig.

8.4. Belegungsstrategien

8.4.1. Zusammenhängende Belegung

alle Daten nur am Stück an beliebiger Pos. auf Platte (s.o.: Best Fit, ect.)

Problem: woher vorher Größe der Datei wissen?
auch „interne Fragmentierung“ – in der Datei – möglich.

8.4.2. Verkettete Belegung

Am Teil-Datei-Ende ist ein Zeiger, wo es weitergeht.
Vorteil: immer Platz da.
Nachteile:

- nur sequentielles Lesen möglich.
- auch Zeiger brauchen Speicher
- (Idee: mehrere Blöcke = 1 Cluster) → aber Verschwendung.
- Integrität schwer zu checken.

FAT: An Stelle m (für Block m) steht dann die Folge-Adresse, usw.
Freie Werte: 0.
+: schneller Zugriff,
–: ständiger Wechsel des Kopfes zwischen FAT und Daten. (Lösung: Cache)
–: FAT weg → alles weg.

8.4.3. Indizierte Belegung

Indexblock: ein extra Bock pro Datei, in dem alle Blocknummen der Datei stehen.
ggf. Verkettung von Indexblöcken notwendig.

8.5. Speicherplatzverwaltung

Freispeicherliste, implementiert als verkettete Liste oder BIT-Vektor.
Gruppierung freier Blöcke:

8.5.1. Lineare Liste

alle Verzeichisse in einer linearen Liste (?). Man muss lange suchen. → Cache?

8.5.2. Hash-Tabelle:

nicht zu kleine Hash-tabelle verwenden. Ggf. auf größre umsteigen, dann muss man aber alles neu berechnen.