Skript zur Vorlesung

Systemprogrammierung

bei Professor Ney

Wintersemester 2000/2001

RWTH-Aachen

verfaßt von

Sandip Sar-Dessai

Dieses Dokument wurde erstellt mit LYX Version 1.1.4fix3 und enthält die Vorlesungsinhalte der Vorlesung "Systemprogrammierung" bei Professor Ney im WS 2000/2001 an der RWTH-Aachen. Ich hoffe, mit dem Skripten der Vorlesung nicht die Rechte Dritter verletzt zu haben, falls dem jedoch so ist, bitte ich um Mitteilung.

Der Text spiegelt die von mir aufgezeichneten Unterlagen wieder. Ich kann daher weder die Vollständigkeit noch die Korrektheit gewährleisten. Sollten irgendwelche Fehler oder Unstimmigkeiten auftauchen, oder sollten Inhalte fehlen, bitte ich um Benachrichtigung unter der angegebenen Adresse.

Es sollte hier darauf hingewiesen werden, daß vorlesungsergänzend das vom Lehrstuhl I4 der RWTH-Aachen herausgegebene Skript "Systemprogrammierung" (Otto Spaniol u.a.) empfohlen wurde, das die hier beschriebenen Inhalte teilweise wesentlich genauer beschreibt.

Die aktuelle - weitestgehend fehlerbereinigte - Version dieses Dokuments läßt sich jederzeit von meiner Homepage runterladen.

Für Kritik, Verbesserungsvorschläge und insbesondere Korrekturen bin ich jederzeit offen.

Sandip Sar-Dessai

Download und Kontakt
Homepage: http://www.sandip.de
E-Mail: mails@sandip.de

2. Auflage, 3. April 2001

Inhaltsverzeichnis

Kapitel 1

Einführung

Betriebssystem ("Operating System"): Programm, das die komfortable und effiziente Nutzung eines Rechners ermöglicht.

1.1 Speicherhierarchie

Die sogenannte Hierarchie (siehe Abbildung 1.1) besitzt einen guten Kompromiß zwischen Kosten und Zugriffszeit. Die folgende Tabelle zeigt die Ausstattung einer Workstation von 1993 und des Pentium III von 1998:

	Workstation (1993)	Pentium III Xeon 450MHz (1998)
Register	10ns, 256B-1024B	2,2ns, 128B
Prim. Cache	10ns, 16KB-32KB	2,2ns, 32KB
Sek. Cache (SRAM)	20ns, 64KB-1MB	2,2ns, 512KB
Hauptspeicher (DRAM)	3,5-100ns, 16MB-128MB	20ns, 128MB-64GB
Harddisk	16-50ms, 100MB-1GB	10GB

1.2 I/O-Operationen und DMA

I/O (Input/Output) - Ereignis. Ereignisse werden der CPU mittels Interrupt signalisiert, der über Hardware oder Software erfolgen kann. Bei einem Interrupt stoppt die CPU den aktuellen

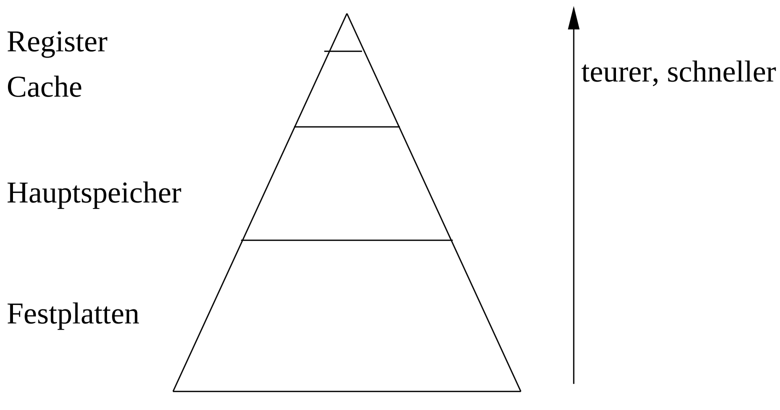


Abbildung 1.1: Speicherhierarchie

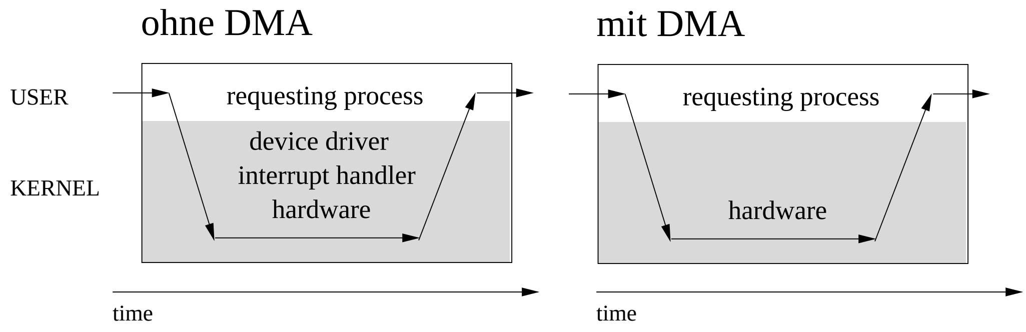


Abbildung 1.2: I/O-Operation mit und ohne DMA

Prozeß, speichert den Zustand und startet eine dem Interrupt entsprechende Routine. Nach der Ausführung dieser Interrupt-Routine setzt die CPU den unterbrochenen Prozeß an der gleichen Stelle fort.

Ablauf einer I/O-Operation:

1. Benutzerprozeß fordert I/O-Operation über einen Systemcall an.
2. System-Call unterbricht den Benutzerprozeß und gibt die Kontrolle an das OS, das die entsprechende 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. Die CPU wartet währenddessen (synchrones I/O) oder gibt die Kontrolle zurück an das OS (asynchrones 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 unterbrochenen Benutzerprozeß fort.

DMA (Direct Memory Access): I/O-Operation ohne dauernde Überwachung durch die CPU; Steuerung erfolgt nach einer geeigneten Initialisierung durch die Kontrolleinheit des I/O-Device, vgl. Abbildung 1.2. Vorteil: Die CPU wird entlastet und ist frei für andere Aufgaben.

1.3 Komponenten des Betriebssystems

Wir unterscheiden typischerweise:

Prozeß-Verwaltung (siehe 2)

- Neue Prozesse erzeugen und alte vernichten
- CPU-Zeit auf mehrere Prozesse verteilen
- Prozesse synchronisieren
- Deadlocks vermeiden und beheben
- Mittel für die Interprozeß-Kommunikation bereitstellen

Hauptspeicherverwaltung (siehe 7)

- Belegte und freie Speicherbereiche verwalten
- Entscheiden, welcher neue Prozeß bei frei werdendem Speicherplatz geladen wird
- Entscheiden, wieviel Speicherplatz einem Prozeß zugewiesen oder entzogen wird

Sekundärspeicherverwaltung

- Disk-Scheduling: Sammeln mehrerer I/O-Anfragen und Abarbeiten in der günstigsten Reihenfolge
- Speicherplatz-Zuteilung, um die Zugriffszeit klein zu halten
- Verwalten der freien Speicherplätze

File System (siehe 10)

- Geeignete Abbildung der logischen Struktur auf die physikalische Struktur des Speichers
- Erstellen, Manipulation und Löschen von Dateien und Verzeichnissen
- Automatisches Sichern wichtiger Daten (Backup)

Kommando-Interpreter

Schnittstelle zwischen Betriebssystem und Benutzer, damit der Benutzer die Funktionalität des OS nutzen kann.

Realisierung: entweder direkter Teil des OS oder separates Programm, sogenanntes Systemprogramm (bspw. die Shell der Kommando-Interpreter bei UNIX, siehe 1.4).

Systemaufrufe

Ein Systemaufruf ist beispielsweise das Lesen eines Files

```
count = read (filename, buffer, n_bytes)
```

Liest eine Anzahl von `n_bytes` Bytes aus dem File `filename` in den Array `buffer`. Nach dem Lesen sollte `count=n_bytes` sein. Andernfalls ist ein Fehler aufgetreten.

Weitere Beispiele für Systemaufrufe:

- Prozeß-Kontrolle: create/terminate process, wait/signal event, allocate/free memory
- File System: create/delete files, open/close file, read/write file
- Gerätemanipulation: request/release device, read/write/reposition device
- Prozeß-Kommunikation: create/delete communication connection, send/receive message
- Abruf von Informationen: get attributes of process/file/date, get/set time/date

Schichtenkonzept

Die folgende Tabelle zeigt das Schichtenkonzept des System THE (THE=Technische Hogeschool Eindhoven):

n	Layer
5	User Programs
4	Buffering for I/O-Devices
3	Operator console device driver
2	Memory management
1	CPU Scheduling
0	Hardware

Ziel: Modularität

Idee: Die Schicht n greift auf die Dienste der darunter liegenden Schichten $n-1, n-2, ...$ zurück.

Vor- und Nachteile des Schichtenkonzepts:

- Debugging und Verifikation jeder Schicht wird vereinfacht
- Verlust an Effizienz
- Definition der Schichten ist nicht einfach

1.4 Betriebssystem UNIX

Geschichte von Unix: Viele der UNIX-Konzepte gehen auf MULTICS zurück.

- ca. 1965: OS MULTICS (Multiplexed Information and Computing Service) am MIT (und in den Bell Labs, GE, ...)
- 1969: UNIX (UNICS: Uniplexed Information and Computing Service) auf DEC PDP7 von K. Thompson implementiert in Assembler, später auf der PDP11.
- 1973: nach wenig erfolgreichen Versuchen mit den Sprachen BCPL, B, NB wird UNIX in C implementiert (Thompson, Ritchie).

Vorteile (damals):

- Lesbarkeit und Portierbarkeit des Codes
- Kleiner Umfang, Modularität und klares Design
- Fast keine Kosten für Universitäten

Praktische Aspekte:

- File-Aufbau: Abfolge von Bytes, darüber hinaus gibt es keine Strukturierung
- Gute Kombinierbarkeit der verschiedenen Systemroutinen (oder auch Shell-Kommandos)

`pwd`	Ausgabe des aktuellen Verzeichnisses (print working directory)
`cd dirname`	Verzeichniswechsel nach `dirname`, ohne Parameter: Ins Homeverzeichnis
`ls file`	Dateien auflisten; `-l` mit Attributen, `-a` alle Dateien
`cp file file`	Dateien kopieren
`mv file file`	Dateien umbenennen oder verschieben
`rm file`	Dateien löschen
`cat file ...`	ggf. mehrere Dateien nacheinander auf dem Bildschirm ausgeben
`page/more file`	Dateien seitenweise ausgeben
`wc file`	Zeilen, Zeichen und Worte zählen; `-l` nur Zeilen, `-c` nur Zeichen, `-w` nur Worte
`head file`	Anfang von `file` ausgeben; `-n` Zeilenzahl, `-c` Bytezahl
`tail file`	Ende der Datei ausgeben; `-n`, `-c` wie head, `-f` ständig weiterausgeben (wachsende Files)
`tr s1 file`	`-d` Zeichen in `s1` löschen, `-s`: Mehrfache Buchstaben von `s1` durch einen ersetzen
`tr s1 s2 file`	Zeichen in `s1` durch die in `s2` ersetzen; `-c s1` komplementieren
`sort file`	Sortieren; `-b` führende Spaces ignorieren, `-d` alphabetisch, `-n` numerisch, `-f` ignore case
`diff file file`	Unterschiede zwischen den Dateien ausgeben (Zeilen die nur in einer Datei vorkommen)
`paste file ...`	Jeweils Dateizeilen in eine zusammenfügen (tab-getrennt); `-s` Eine Datei in eine Zeile
`cut file`	gibt Teile der Datei aus; `-dc` Trenner `c`, `-fa,b,...` Felder a,b,...
`grep expr file`	Gibt passende Zeilen aus
`uniq file`	Gleiche Zeilen unterdrücken; `-c`: Anzahl vor Zeile; `-w n` nur `n` Zeichen
`mail`	Mails verschicken

Tabelle 1.1: Einige UNIX Shell Commands

Systemaufbau

1. Hardware: terminal controllers, device controllers, memory controllers, terminals, discs and tapes, physical memory
2. Kernel Interface to the Hardware
3. Kernel: signals, file system, CPU Scheduling, terminal handling, swapping, page replacement, character I/O-systems, block I/O-systems, demand paging, terminal drivers, disc and tape devices, virtual memory
4. System-Call Interface to the Kernel
5. System Programs: shell and commands, compilers and interpreters, system libraries
6. Users

Beispiele für UNIX-Shellkommandos zeigt Tabelle 1.1.

UNIX war ursprünglich (auch) als Werkzeug für die Textverarbeitung konzipiert.

Beispiel: Erstellen der 1000 häufigsten Wörter aus einem natürlichsprachigen Text (in Datei `my_text`):

```
cat my_text | tr -sc '[A-Za-z]' '\012' | sort | uniq -c |\
sort -n | tail -1000 > vocabulary_list
```

Anmerkungen hierzu:

- Pipesymbol '`|`': Ausgabe des linken Kommandos wird Eingabe des rechten Kommandos
- Continuesymbol '\' am Ende einer Zeile: Kommando in nächster Zeile weiterführen
- Outputumleitung '>': Umleitung des Terminal-Outputs in eine Datei
- Die Bedeutungen der Kommandos sind der Tabelle zu entnehmen.

Kapitel 2

Grundlagen des Prozeß-Management

Prozeß: Programm in Ausführung zu einem bestimmten Zeitpunkt, d.h. ein Prozeß umfaßt den Programm-Code, den aktuellen Befehl und den Wert aller Programm-Variablen (einschl. Dateien). Auf Hardware-Ebene umfaßt ein Prozeß:

- text section: Programmcode mit Program Counter (PC), CPU-Register etc.
- data section: globale Variablen
- process stack: Prozeduraufrufe inkl. aktueller Prozedurparameter, lokale Variablen

Eine grundsätzliche Unterscheidung wird in der Bearbeitung von Befehlen getroffen:

- sequentieller Prozeß: Ein Befehl pro Zeiteinheit
- paralleler Prozeß: Mehrere Befehle pro Zeiteinheit

Beispiel: Addition von acht Zahlen $a_1 ... a_8$:

- sequentiell ($n-1$ Zeiteinheiten): (1) $a_1 + a_2$ (2) $a_1 + a_2 + a_3$ ... (7) $a_1 + ... + a_8$
- parallel ($ld\, n$ Zeiteinheiten): (1) a_1+a_2, a_3+a_4,... (2) $a_1+...+a_4$, $a_5+...+a_8$ (3) $a_1+...+a_8$

Unterscheide außerdem:

- Benutzerprozeß: vom Benutzer aufgerufene Programme, Compiler und Kommando-Interpreter, der dem Benutzer zugeordnet ist.
- OS-Prozeß: Verwaltung des Hauptspeichers, des File-Systems, der Peripherie, ...

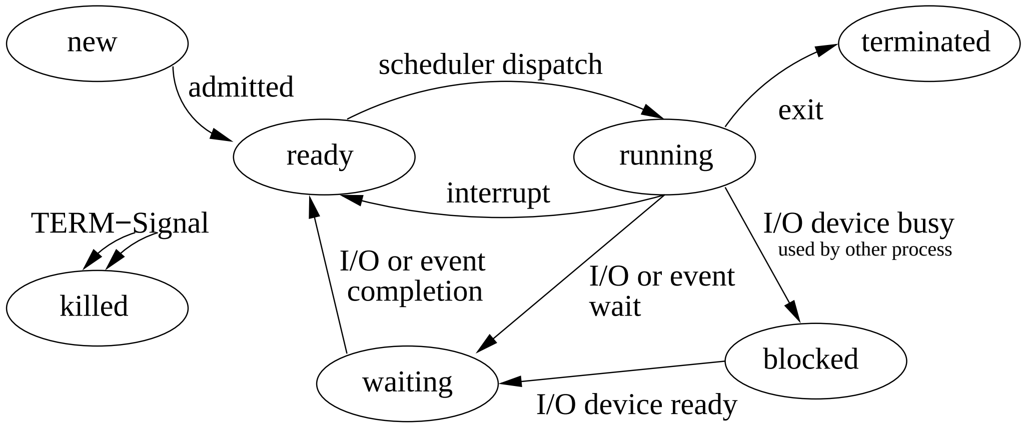


Abbildung 2.1: Prozeßzustände. CPU wird benötigt für Running und Blocked.

2.1 Prozeßzustände und Prozeß-Kontrollblock

Ein Zustand kann sich in unterschiedlichen Zuständen befinden, wie Abbildung 2.1 illustriert.

Die Prozeßzustände:

- **new**: Ein neuer Prozeß wird gestartet
- **ready**: Prozeß ist bereit zur Ausführung und wartet auf einen freien Prozessor
- **running**: Prozeß wird gerade vom Prozessor bearbeitet (CPU ist belastet)
- **waiting**: Prozeß wartet auf eigenes Betriebsmittel (bspw. auf Ende eines Lesevorgangs)
- **blocked**: Prozeß wartet auf von fremdem Prozeß belegtes Betriebsmittel (CPU ist belastet)
- **terminated**: Prozeß wurde vollständig ausgeführt und beendet
- **killed**: Prozeß wurde durch ein entsprechendes Signal beendet

Jeder Prozeß hat einen PCB (process control block), der folgendes enthält:

- Prozeß-Zustand
- Befehlszähler (PC) und CPU-Register (condition code, index, stack, general purpose etc.)
- Memory Management Information: base/limit registers, page/segment table, ...
- CPU Scheduling Info: priority, Parameter für das Process Scheduling, ...
- I/O-Status Info: files, tapes, devices, ...
- Accounting Information

2.2 Threads

Als Erweiterung des Konzepts Prozeß wird das Konzept Thread eingeführt: ein Prozeß (heavy weight process) besteht aus mehreren Threads (light weight process), die alle dasselbe Text- und Datensegment besitzen, jedoch separaten Stack und CPU-Register (inkl. Befehlszähler) haben.

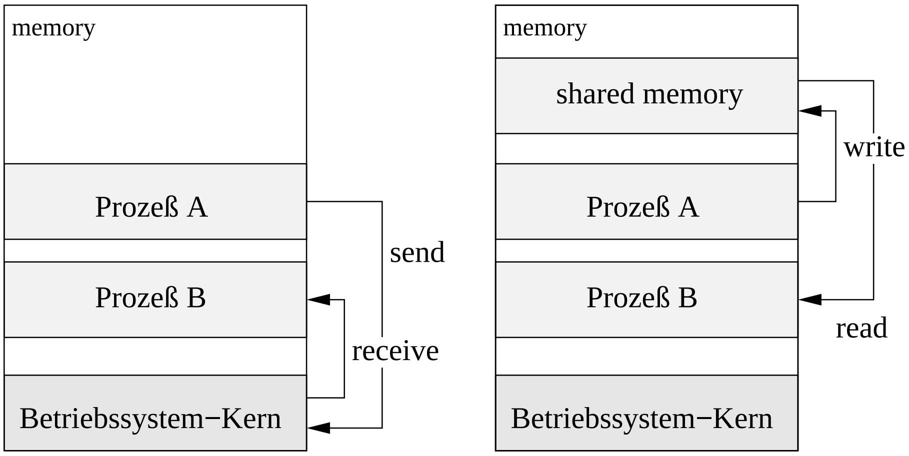


Abbildung 2.2: Interprozeßkommunikation: Message Passing (links) und Shared Memory (rechts)

2.3 Interprozeß-Kommunikation

Die Kommunikation zwischen einzelnen Prozessen wird wichtig, wenn diese zusammenarbeiten. Man unterscheidet dabei folgende Grundkonzepte (vgl. hierzu auch Abbildung 2.2):

- **Message Passing**: Über das Betriebssystem wird eine Verbindung zwischen A und B aufgebaut (Systemcalls: getHostID, open/accept/close connection).
- **Shared Memory**: Unabhängig vom OS haben A und B Zugriff auf einen gemeinsamen Speicherbereich (Shared Memory). Der Vorteil ist die Schnelligkeit, allerdings ist Synchronisation erforderlich.

Kapitel 3

Prozeß-Scheduling

Die Aufgabe des Prozeß-Scheduling ist es, die CPU-Zeit unter mehreren wartenden Prozessen aufzuteilen. Neben einer fairen Zuteilung sollte das Process Scheduling auch den Wechsel zwischen CPU-Bursts und I/O-Bursts berücksichtigen.

Kriterien für Scheduling:

- Auslastung der CPU
- Durchsatz (Throughput): Anzahl der abgearbeiteten Jobs (Prozesse) pro Zeiteinheit
- Faire Behandlung: Jeder Prozeß sollte im Mittel gleichen CPU-Zeitanteil erhalten
- Ausführungszeit (Turnaround-Time): Wartezeit und Bearbeitungszeit (Bedienzeit)
- Wartezeit (Waiting Time): Der Scheduler hat im wesentlichen Einfluß auf die Wartezeit
- Antwortzeit bei interaktiven Systemen (Response Time): Zeit zwischen Anfrage (Request) und Antwort bzw. Reaktion

Unterscheide:

- non-preemptive Scheduling (run-to-completion-strategy: nicht-unterbrechende Strategie): Jeder begonnene Job wird stets ohne Unterbrechung zum Ende geführt.
- preemptive Scheduling (unterbrechende Strategie): Ein begonnener Job kann unterbrochen und später (u.U. mit weiteren Unterbrechungen) fortgesetzt werden

3.1 Einfache Scheduling Strategien

In der Queue seien N Jobs mit (exakt geschätzten) Bearbeitungszeiten $t_1, ..., t_n, ..., t_N$, der Index n gibt die Reihenfolge in der Queue an: Job P_n (mit Zeit t_n) vor Job P_{n+1} (mit Zeit t_{n+1}).

Strategien:

- **FCFS (first come first served)**: FIFO-Strategie
- **LCFS (last come first served)**: LIFO-Strategie
- **SJF (shortest job first)**: SJN-Strategie (shortest job next)

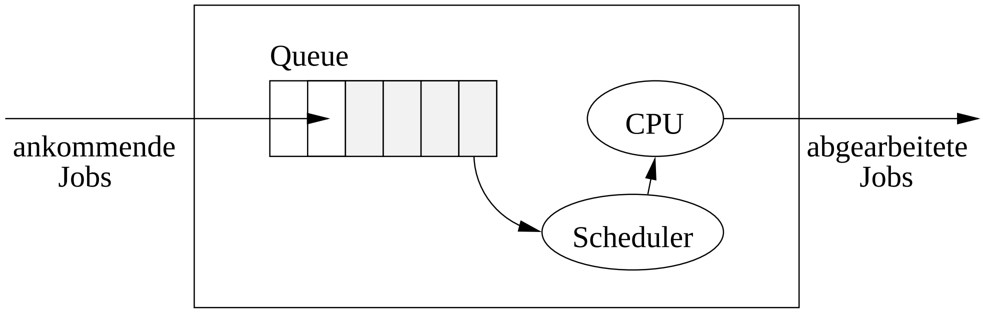
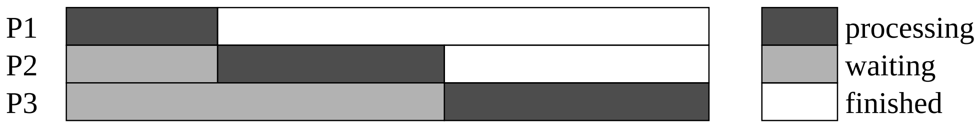


Abbildung 3.1: Einfache Scheduling-Strategien



3.1.1 FCFS und LCFS

Ausführungszeit aller Jobs: $T_{FCFS} = \sum_{n=1}^{N}(N+1-n)t_n$, $T_{LCFS} = Nt_n + ... + t_1 = \sum_{n=1}^{N} nt_n$.

In der Regel keine wechselseitige Abhängigkeit zwischen Ausführungszeit und Reihenfolge des Ankommens in der Queue. Im statistischen Mittel gilt dann: $\langle T_{LCFS}\rangle = \langle T_{FCFS}\rangle$.

3.1.2 SJF

Die Jobs werden nach der Bearbeitungszeit t_n sortiert, so daß wir die permutierte Folge t_{K_n} erhalten: $t_{K_n} \leq ... \leq t_{K_1}$. Die Ausführungszeit aller Jobs beträgt dann: $T_{SJF} = \sum_{n=1}^{N} nt_{K_n}$.

Optimalität von SJF: Jede non-preemptive Scheduling Strategie kann als Permutation $K_1...K_N$ $(= K_1^N)$ der ursprünglichen Job-Reihenfolge $1..N$ interpretiert werden. Die resultierende Gesamtausführungszeit ist $T(K_1^N) = \sum_{n=1}^{N} nt_{K_n}$. Diese Zeit ist minimal (wegen der Gewichtung mit n) für die Permutation K_1^N falls $t_{K_n} \leq ... \leq t_{K_1}$.

Bearbeitungszeit t_N Wie wird t_N ermittelt?

- I.A. wird der Benutzer eine realistische Schätzung abgeben: gibt er eine zu kurze Zeit an, wird der Job bei Überschreitung abgebrochen, gibt er eine zu lange Zeit an, werden andere Jobs vorgezogen.
- Schätzung aus früheren Werten: aus echter Jobzeit (Jobs desselben Benutzers) bzw. CPU-Bursts (frühere CPU-Bursts desselben Jobs).
  t_N: genauer Wert, τ_N: geschätzter Wert. Es ist

$$\begin{aligned}
\tau_{n+1} &= at_n + (1-\alpha)\tau_n \\
&= at_n + (1-\alpha)(at_{n-1} + (1-\alpha)\tau_{n-1}) \\
&= \left(\alpha \cdot \sum_{i=0}^{n}(1-\alpha)t_{n-i}\right) + (1-\alpha)^{n+1}\tau_0
\end{aligned}$$

  Die früheren Messungen t_{N-i} $(i = 0..n)$ gehen mit dem Gewicht $\alpha(1-\alpha)^i$ in den Schätzwert τ_{n+1} ein (exponentielle Mittelung/Wichtung).
  $\alpha \to 1$: Vergangene Messungen werden “vergessen”.
  $\alpha \to 0$: Vergangene Messungen werden “behalten”.

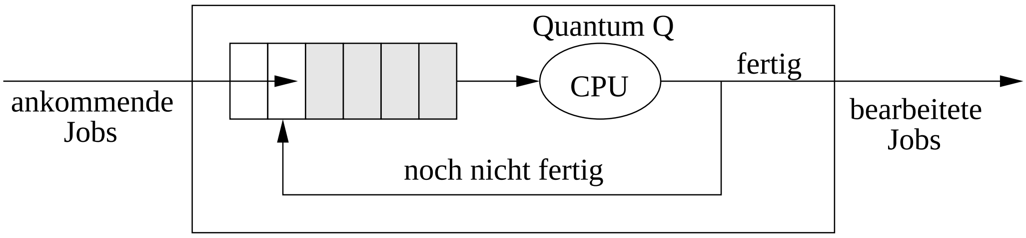


Abbildung 3.2: Round-Robin-Verfahren

3.2 Verfeinerte Scheduling Strategien

Preemptives Scheduling versucht vor allem, das Problem der sogenannten Langläufer zu vermeiden:

- Ist der Langläufer in Bearbeitung, kommt für lange Zeit kein anderer Job zum Zuge
- Der Langläufer kommt u.U. nicht zum Zuge, da laufend Jobs mit kürzeren Bearbeitungszeiten eintreffen

3.2.1 Preemptives SJF-Scheduling (SRPT)

Shortest Remaining Processing Time First (SRPT): Ein Job, der sich in Bearbeitung befindet, wird unterbrochen, falls ein neuer Job ankommt, und dessen Bearbeitungszeit kürzer ist als die Restzeit des gerade laufenden Jobs.

Man kann zeigen, daß SRPT zu einer minimalen mittleren Ausführungszeit führt (ohne Berücksichtigung des Overheads durch Prozeß-Wechsel, d.h. work-concerning).

3.2.2 Round-Robin-Verfahren

Zeitscheibenverfahren, siehe Abbildung 3.2:

$Q \to 0$: Jeder der n Prozesse erhält $\frac{1}{n}$ der CPU-Zeit. Ergebnis: Processor-Sharing.
$Q \to \infty$: Ein laufender Prozeß wird nicht unterbrochen. Ergebnis: FCFS.

3.2.3 Priority-Scheduling

Jedem Prozeß wird eine Priorität zugewiesen.

Strategie: **Highest Priority First (HPF)** - Vorrang hat die höchste nichtleere Prioritätsklasse. Innerhalb einer Prioritätsklasse: FCFS.

Varianten: preemptiv/non-preemptiv - Bei der preemptiven Variante wird ein laufender Job unterbrochen, sobald ein Job mit höherer Priorität ankommt.

3.2.4 Multilevel-Feedback-Queue-Scheduling

Im Gegensatz zum Priority-Scheduling sind die Stufen hier Zeitscheibenlängen, diese Strategie ist also eine Kombination von Round-Robin und Priority Scheduling, siehe Abbildung 3.4.

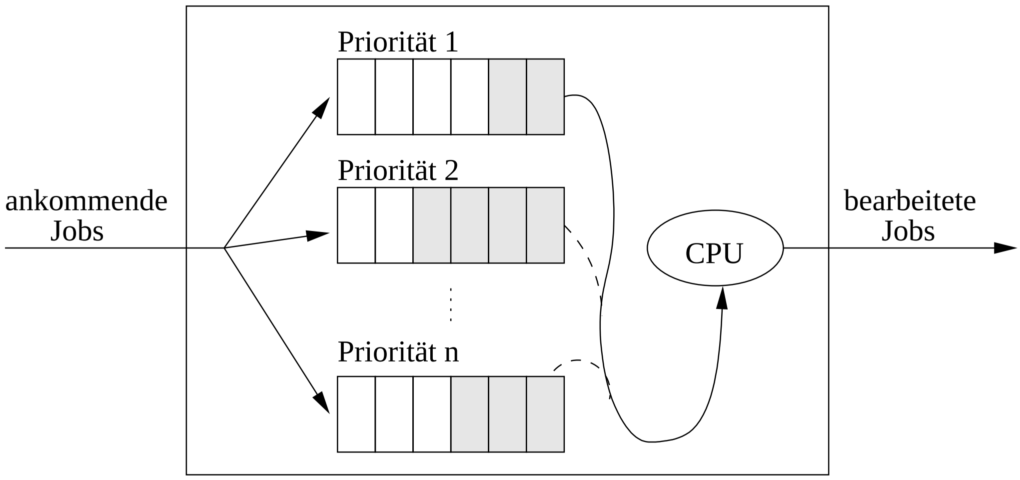


Abbildung 3.3: Priority Scheduling

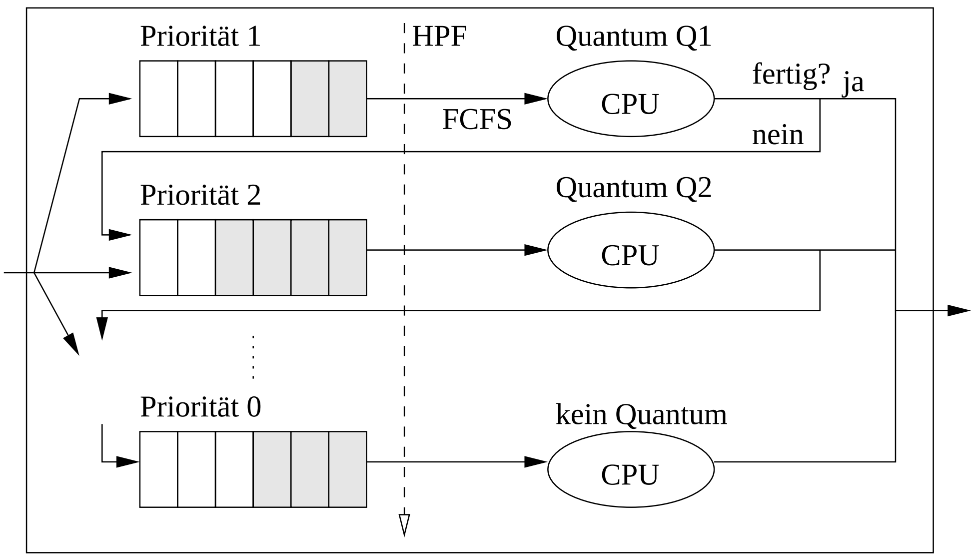


Abbildung 3.4: Multilevel-Feedback-Queue-Scheduling. "CPU" steht für denselben/dieselben Prozessor(en): Prozesse niedrigerer Priorität werden nur bearbeitet, wenn kein Prozeß höherer Priorität vorhanden ist.

Kapitel 4

Bediensysteme

Die folgende Tabelle gibt einige Bediensysteme an:

Bedienungssystem	**Kunden**
Batch-Betriebssystem	Jobs oder Prozesse
Computer-Netzwerk	Jobs oder Prozesse
Parallel-Computer	Teilprozesse (Teilaufgaben)
Terminal-Beetriebssystem	Interaktive Kommandos
Plattenspeicher	Lese-/Schreib-Anfragen
Kommunikationsnetzwerk	(zu sendende) Nachrichten

Quantitive Beschreibung von Bediensystemen:

- Ankunftsstrom: Wie sind die Ankunftszeiten (bzw. die zeitl. Abstände) der Kunden?
- Bedienzeit: Wie sind die Bedienzeiten/Systemzeiten der Kunden?
- Bedienstrategie: Welche Scheduling-Strategie wird benutzt?
- Wieviele Server (CPUs)?
- Wieviele Plätze hat die Queue?
- Wie groß ist die Kundenpopulation?

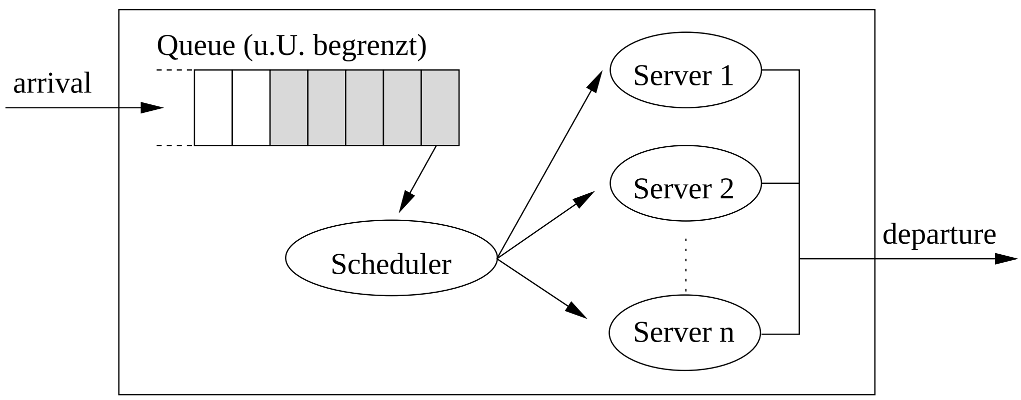
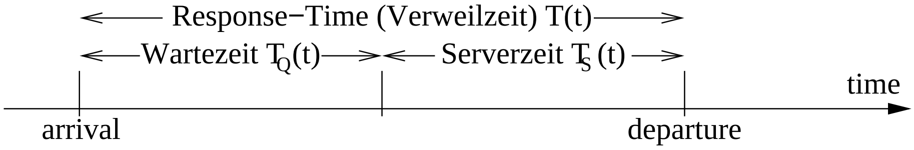


Abbildung 4.1: Allgemeines Bedienungssystem

Schematisch durchläuft ein Kunde (Ankunft zur Zeit t) folgende Abschnitte:



- $T_Q(t)$ und $N_Q(t)$: Wartezeit und aktuelle Anzahl Kunden in der Queue
- $T_S(t)$ und $N_S(t)$: Serverzeit und aktuelle Anzahl Kunden in den CPUs
- $T(t) = T_Q(t) + T_S(t)$ und $N(t) = N_Q(t) + N_S(t)$: Verweilzeit bzw. Gesamtstrom

Entsprechend gilt für die Mittelwerte:

- $E\{T\} = E\{T_Q\} + E\{T_S\}$
- $E\{N\} = E\{N_Q\} + E\{N_S\}$

4.1 Kendall-Notation

Kendall-Notation eines allgemeinen Bediensystems: $F_1|F_2|n_1|n_2|n_3 - S$

- F_1: Verteilungstyp der Zwischenankunftszeiten (Zeit zw. 2 ankommenden Jobs)
- F_2: Verteilungstyp der Bedienzeiten (Zeit in den Servern)
- n_1: Zahl der Server
- n_2: max. Anzahl der Kunden im System (also Server- und Queueplätze)
- n_3: Größe der Kundenpopulation (max. Anzahl Kunden, die bedient werden wollen)
- S: Schedulingstrategie, bspw. FCFS, LCFS, Priority, Random

F_1 und F_2 werden typischerweise gewählt aus

- M (marker- oder memoryless distribution): Exponentialverteilung
- D (deterministic distribution): Zeiten sind genau bekannt
- G (general distribution): allgemeine Verteilung

Konvention: Sofern Größen außer F_1/F_2 nicht spezifiziert wurden, gilt: $n_1 = n_2 = n_3 = \infty$, $S = FCFS$.

Beispiele:

- $M|M|c$: System mit c Servern und $n_2 = n_3 = \infty$ sowie FCFS-Strategie
- $M|M|c|m$: System mit c Servern, Kapazität m und $n_3 = \infty$ sowie FCFS-Strategie, d.h. falls die Zahl $N(t)$ der Kunden im System gleich m ist wird jeder neuankommende Kunde abgewiesen (bspw. Telefon-Netz mit c Leitungen: $M|M|c|c$ keine Warteschlange - d.h. ein Kunde wird abgewiesen, falls alle c Leitungen besetzt sind)

- $M|M|1||k$: System mit einem Server und max. n Kunden sowie $n_2 = \infty$ (bspw. ein Rechner mit k Kunden am interaktiven Terminal).
- $G|D|1$: Kaffee-Automat
- $D|G|1$: Arztbesuch mit Terminabsprache
- $G|G|1$: Arztbesuch ohne Terminabsprache

4.2 Grundlagen der Wahrscheinlichkeitsrechnung

Um Erwartungswerte für einzelne Bediensysteme angeben zu können (bspw. für Bedienzeiten, Kunden im System oder Zwischenankunftszeiten etc.) sind einige stochastische Grundlagen einzuführen. Hier wird letztlich die M-Verteilung (Exponentialverteilung) genauer beleuchtet.

Unter dem Erwartungswert (oder Mittelwert) versteht man das wahrscheinlichste Ergebnis für den nächsten Wert, während mit der Varianz die (quadratisch gewichtete) Streuung um der Werte um den Erwartungswert versteht.

4.2.1 Binomialverteilung

Ausgangspunkt: Experiment mit binärem Ausgang, bspw. Werfen einer Münze. π: erfolgreich (1), $1-\pi$: erfolglos (0).

Frage: Mit welcher Wahrscheinlichkeit beobachten wir k Erfolge bei n Versuchen (jeder mit Erfolgswahrscheinlichkeit π), i.Z. $p(k|n,\pi)$?

Wir können Ergebnis als Binärzahl darstellen, bspw. 111...1000...0 (mit k Einsen und $n-k$ Nullen). Dieses Ergebnis hat die Wahrscheinlichkeit $\pi^k \cdot (1-\pi)^{n-k}$. Insbesondere gibt es $\begin{pmatrix} n \\ k \end{pmatrix} = \frac{n!}{k!(n-k)!}$ (Binomialkoeffizient) mögliche Binärworte mit k Einsen und $n-k$ Nullen.

Damit erhalten wir für $p(k|n,\pi)$:

$$p(k|n,\pi) = \begin{pmatrix} n \\ k \end{pmatrix} \cdot \pi^k \cdot (1-\pi)^{n-k} \qquad k = 0, ..., n$$

Normierung:

$$\sum_{k=0}^{\infty} p(k|n,\pi) = 1 \quad \text{mit}\, p(k|n,\pi) = 0\, \text{fuer}\, k > n$$

Mittelwert/Erwartungswert:

$$E\{k\} := \sum_{k=0}^{\infty} k \cdot p(k|n,\pi) = n \cdot \pi$$

Varianz:

$$Var\{k\} := E\{(k - E\{k\})^2\} = \sum_{k=0}^{\infty} (k - n\pi)^2 \cdot p(k|n,\pi) = n \cdot \pi \cdot (1-\pi)$$

Beispiel: Wahrscheinlichkeit, in $n = 5$ Versuchen aus einem Skatspiel $k = 0, 1, 2, ..., 5$ Herzkarten zu ziehen ($\pi = \frac{8}{32} = \frac{1}{4}$):

k	0	1	2	3	4	5	> 5	Σ
$p(k\|n,\pi)$	$0,237$	$0,396$	$0,264$	$0,088$	$0,015$	$0,001$	$0,000$	$1,000$

Erwartungswert und Varianz:

$$E\{k\} = 0 + 0,396 + 0,528 + 0,264 + 0,06 + 0,005 = 1,253 \approx n\pi$$

$$Var\{k\} = 1,253^2 \cdot 0,237 + 0,253^2 \cdot 0,396 + 0,747^2 \cdot 0,264 + ... + 3,747^2 \cdot 0,001 = 0,941 \approx n\pi(1-\pi)$$

4.2.2 Poisson-Verteilung

Die Poisson-Verteilung Ergibt sich aus Binomialverteilung im Grenzfall "seltener" Ereignisse, also $\pi \to 0$ und $n \to \infty$, wobei $\vartheta = n\pi$ endlich bleibt:

$$n \gg k: \quad \begin{pmatrix} n \\ k \end{pmatrix} \approx \frac{n^k}{k!}, (1-\pi)^{n-k} \approx e^{-\pi(n-k)} \approx e^{-n\pi} = e^{-\vartheta}$$

Damit erhalten wir in dieser Näherung

$$\begin{pmatrix} n \\ k \end{pmatrix} \pi^k (1-\pi)^{1-k} \longrightarrow \frac{\vartheta^k}{k!} e^{-\vartheta}$$

Normierung:

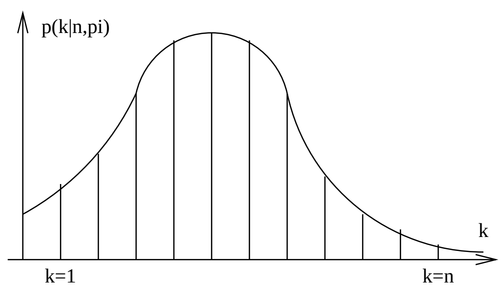
$$p(k|\vartheta) = e^{-\vartheta} \cdot \frac{\vartheta^k}{k!} \quad k = 0, 1, ...$$

Mittelwert/Erwartungswert:

$$E\{k\} = \sum_{k=0}^{\infty} k \cdot p(k|\vartheta) = \sum_{k=0}^{\infty} (k \cdot e^{-\vartheta} \cdot \frac{\vartheta^k}{k!}) = ... = \vartheta$$

Varianz:

$$Var\{k\} = \sum_{k=0}^{\infty} (k-\vartheta)^2 \cdot p(k|\vartheta) = ... = \vartheta$$



4.2.3 Poisson-Prozeß

Von der Poisson-Verteilung kommen wir zum Poisson-Prozeß, indem wir Ereignisse innerhalb eines Zeitintervalls $[0;t]$ betrachten und den Parameter ϑ proportional zur Zeitdauer t annehmen:

$$E_t\{k\} = \vartheta = \lambda \cdot t \quad \text{mit einer zeitunabh. Konstante } \lambda$$

Normierung: $p_t(k|\lambda) = e^{-\lambda t} \cdot \frac{(\lambda t)^k}{k!} \quad k = 0, 1, ...$

Mittelwert: $E_t\{k\} = \lambda \cdot t$

Varianz: $Var_t\{k\} = \lambda \cdot t$

Eigenschaften des Poisson-Prozesses:

- Proportional zur Intervall-Länge t, also $t \sim p_t(k|\lambda)$
- Proportional zur Intensitätsrate λ, also $\lambda \sim p_t(k|\lambda)$
- Unabhängig von der Zeit, also $p_{[0,t]}(k|\lambda) = p_{[\tau;\tau+t]}(k|\lambda)$
- Unabhängig von vergangenen oder zukünftigen Ereignissen
- Es treffen nie zwei Ereignisse gleichzeitig ein.

Das Eintreffen der Kunden in einem exponentialverteilten Bediensystem $(M|M|...)$ ist ebenfalls ein Poisson-Prozeß. Das heißt, die Wahrscheinlichkeit, daß in einem Zeitabschnitt $[0;t]$ genau k Kunden im System sind (bei einer Ankunftsrate λ), beträgt $p_t(k|\lambda)$. Damit gilt für den Erwartungswert für die Kundenzahl $E\{N\} = \lambda E\{T\}$ (analog für $E\{N_Q\}$).

4.2.4 Exponentialverteilung

Übergang von derEreignisverteilung zur Zeitverteilung. Wir betrachten die Wahrscheinlichkeit $p_t(k = 0|\lambda)$, daß kein Ereignis im Zeitintervall $[0;t]$ stattfindet: $p_t(k = 0|\lambda) = e^{-\lambda t}$, welches der Wahrscheinlichkeit des Komplementärereignisses $\overline{p_t}(k = 0|\lambda)$ (mindestens ein Ereignis) entspricht:

$$\overline{p_t}(k = 0|\lambda) = p_t(k > 0|\lambda) = 1 - p_t(k = 0|\lambda) = 1 - e^{-\lambda t}$$

Die Wahrscheinlichkeit, daß die Zeit, die zwischen zwei Ereignissen durchschnittlich verstreicht, höchstens t ist, beträgt also

$$Pr(\tau \leq t) = F(t|\lambda) = 1 - e^{-\lambda t} \qquad t \in [0;\infty[$$

Es handelt sich um eine kontinuierliche Zufallsgröße mit der Verteilungsdichte

$$f(t|\lambda) = \frac{d}{dt}F(t|\lambda) = \lambda e^{-\lambda t}$$

Mittelwert:

$$E\{t\} = \int_0^\infty t \cdot f(t|\lambda)\, dt = \int_0^\infty \lambda t e^{-\lambda t}\, dt = \frac{1}{\lambda}$$

Varianz:

$$Var\{t\} = E\{(t - \frac{1}{\lambda})^2\} = \int_0^\infty (t - \frac{1}{\lambda})^2 \lambda e^{-\lambda t} dt = ... = \frac{1}{\lambda^2}$$

Analog hierzu läßt sich die Wahrscheinlichkeit, daß kein Ereignis in einer Intervallänge t auftritt, angeben als

$$Pr(\tau > t) = e^{-\lambda t}$$

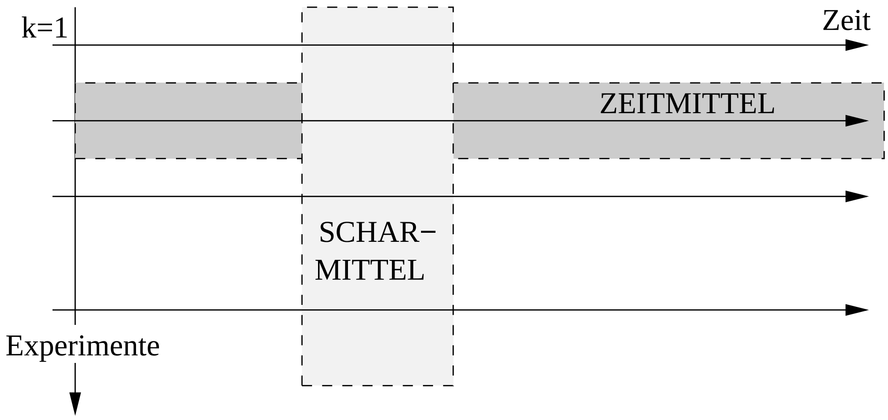


Abbildung 4.2: Zeitmittel und Scharmittel

Es gibt folgende Äquivalenz zwischen Exponentialverteilung und Poisson-Prozessen:

Zwischenankunftszeiten τ mit Exponentialverteilung $\Leftrightarrow$ Zählprozeß n_t ist Poisson-Prozeß.

Wie schon festgestellt, ist das Eintreffen der Kunden in einem exponentialverteilten Bediensystem ein Poisson-Prozeß. Entsprechend sind die Zwischenankunftszeiten exponentialverteilt. Die Wahrscheinlichkeit, daß die Zwischenankunftszeit $E\{T_A\}$ bei einer Ankunftsrate λ mindestens T ist, beträgt also

$$Pr(E\{T_A\} > t) = e^{-\lambda t}$$

4.2.5 Scharmittel und Zeitmittel

Zwei Arten von Mittelwerten (vgl. Abbildung 4.2) :

- Scharmittel: Mittelung über das statistische Ensemble (=Experimente). Für feste Zeit t gilt

$$\frac{1}{t}E\{n_t\} = \frac{1}{t}\sum_{n=0}^{\infty} p(n_t|\lambda t) \cdot n_t = \frac{1}{t}\lambda t = \lambda$$

- Zeitmittel: Mittelung über die Zeit. Für festen Exponenten k des Prozesses gilt $\lim_{t\to\infty} \frac{n_t}{t} = \lambda$ (siehe unten). Für diesen Fall gilt dann Scharmittel=Zeitmittel (ergodische Prozesse), in allgemeinen Fällen ist das nicht erfüllt ($\rightarrow$Ergodentheorie).

Berechnung des Zeitmittels

Wir betrachten den Kehrwert:

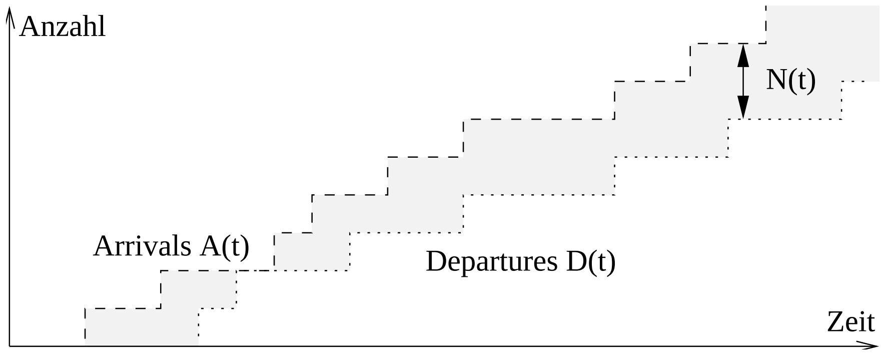
$$\begin{aligned}
\lim_{t\to\infty} \frac{t}{n_t} &= \lim_{I\to\infty} \frac{1}{I}\sum_{i=1}^{I} \tau_i \\
&= \lim_{\Delta\tau_k\to 0} \sum_k \Delta\tau_k \cdot f(\tau_k) \cdot \tau_k = \int_0^\infty d\tau \cdot f(\tau) \cdot \tau \\
&= \int_0^\infty d\tau \cdot \lambda\tau e^{-\lambda\tau} = \frac{1}{\lambda}\int_0^\infty u e^{-u} du = \frac{1}{\lambda}
\end{aligned}$$

mit $f(\tau) = \frac{d}{d\tau}p(n_\tau > 0|\lambda) = \frac{d}{d\tau}(1 - e^{-\tau\lambda}) = \lambda e^{-\lambda\tau}$ und $u = \lambda\tau$, $du = \lambda\, d\tau$..

4.3 Little'sche Formel

System zur Zeit t:

$A(t)$: Zahl der Kunden, die bis zur Zeit t angekommen sind
$D(t)$: Zahl der Kunden, die bis zur Zeit t abgefertigt worden sind
$N(t)$: Zahl der Kunden, die zur Zeit t im System sind.



Die Funktionen $t \to A(t)$ und $t \to D(t)$ sind monoton wachsend, es gilt offensichtlich:

$$N(t) = A(t) - D(t) \geq 0$$

Wir betrachten die mittlere Zahl N_t der Kunden im System pro Zeiteinheit

$$N_t = \frac{1}{t} \int_0^t N(t')\, dt'$$

Mit den Definitionen $A_t := \frac{A(t)}{t}$ (mittlere Zahl ankommender Kunden bis zur Zeit t) und $T_t := \frac{1}{A(t)} \cdot \int_0^t N(t')\, dt'$ (mittlere Verweilzeit der Kunden bis zur Zeit t) gilt aber $N_A = A_t T_t$.

Das System soll sich im Gleichgewicht befinden (stationärer Zustand), d.h. es sollen die Grenzwerte existeieren:

- Ankunftsintensität: $\lambda =< A_t >= \lim_{t\to\infty} A_t$
- Mittlere Verweilzeit: $< T_t >= \lim_{t\to\infty} T_t$
- Mittlere Kundenzahl $< N_t >= \lim_{t\to\infty} N_t$

Die Little'sche Formel besagt dann (für den stationären Zustand):

$$< N_t >=< A_t > \cdot < T_t >= \lambda \cdot < T_t >$$

Eine entsprechende Gleichung gilt auch für die Queue, falls es sich um ein System ohne Preemption handelt.

4.4 Allgemeine M/G/1-Systeme

Kunden: $N_t = N_{Q,t} + N_{S,t}$, Zeiten $T_t = T_{Q,t} + T_{S,t}$.

Zeitabh. Größen $\to$ Zufallsgrößen, bspw. $N(t)-$Gleichgewicht $\to N =< N(t) > -$Erwartungswert $\to E\{N\}$

Für diese Mittelwerte gilt:

$$E\{N\} = E\{N_Q\} + E\{N_S\} \quad \text{sowie} \quad E\{T\} = E\{T_Q\} + E\{T_S\}$$

Ziel: Ausdrücken durch die Verteilungen F_1 und F_2 des Systems $M|G|1$ mit

- Ankunftsstrom; λ Kunden pro Zeiteinheit, exponentialverteile Zwischenankunftszeiten T_Q, $E\{T_Q\} = \frac{1}{\lambda}$
- Bedienzeiten: μ Kunden pro Zeiteinheit, allgemeine Verteilung der Bedienzeiten T_S, $E\{T_S\} = \frac{1}{\mu}$. $E\{T_S^2\}$. $E\{T_S^2\}$ ("zweites Moment") sei dabei bekannt.

Die Auslastung ist definiert als $\rho := \frac{E\{T_S\}}{E\{T_Q\}} = \frac{\lambda}{\mu}$.

Um die vier unabhängigen Größen zu bestimmen, sind schon drei Größen bekannt: $E\{T_S\} = \frac{1}{\mu}$, $E\{N\} = \lambda \cdot E\{T\}$ (Little'sche Formel). Wir brauchen eine weitere Gleichung, bspw. $E\{N\}$ als Funktion der Verteilungen M und G. Dies liefert die Formel von Pollaczek-Khintschin:

$$E\{N\} = \rho + \frac{\lambda^2}{2(1-\rho)} \cdot E\{T_S^2\}$$

Mit dem Variationskoeffizienten $C_S^2 := \frac{Var\{T_S\}}{E\{T_S^2\}}$ ergibt sich aus $E\{N\} = \lambda \cdot E\{T\}$:

$$E\{N\} = \rho + \rho^2 + \frac{1 + C_S^2}{2(1-\rho)} = \lambda E\{T\}$$

Also:

$$\begin{aligned}
E\{T\} &= \frac{1}{\lambda} E\{N\} = \frac{1}{\mu} + \frac{\lambda}{2(1-\rho)} \cdot E\{T_S^2\} \\
E\{T_Q\} &= E\{T\} - E\{T_S\} = \frac{\lambda}{2(1-\rho)} \cdot E\{T_S^2\} \\
E\{N_Q\} &= \lambda \cdot E\{T_Q\} = \frac{\lambda^2}{2(1-\rho)} \cdot E\{T_S^2\} \\
E\{N\} &= \lambda \cdot E\{T\} = \rho + \frac{\lambda^2}{2(1-\rho)} \cdot E\{T_S^2\} \\
& \ldots
\end{aligned}$$

4.5 M/M/1-System und Verwandte Systeme

4.5.1 Analyse eine exponentialverteilten Systems

In den vergangenen Abschnitten sind alle Vorraussetzungen gegeben worden, um ein exponentialverteiltes Bediensystem zu analysieren. Hier werden nun die einzelnen Schritte aufgeführt, exemplarisch wird dieses Verfahren für das $M|M|1$-System im nächsten Abschnitt durchgeführt.

Zunächst ist es hilfreich, das angegebene System in Kendall-Notation anzugeben. Hieraus lassen sich dann die Anzahl der Server (n_1), der Queueplätze ($n_2 - n_1$) sowie die der Kunden (n_3) ablesen. Weiterhin wichtig sind Ankunftsrate λ und Bedienrate μ (jeweils auf einen Kunden bzw. Server bezogen).

Sind diese nicht gegeben, lassen sie sich über die Zwischenankunftszeit $E\{T_A\} = \frac{1}{\lambda}$ bzw. über die Bedienzeit $E\{T_S\} = \frac{1}{\mu}$ errechnen.

Als nächstes muß man ein Zustandsdiagramm aufstellen. Dabei stellt man die möglichen Systemzustände als Knoten. Ein Zustand entspricht der Anzahl der Kunden im System (entsprechend die Beschriftung $0, 1, \ldots$). Die Kundenzahl im System ist ggf. beschränkt durch die Queue und die Kundenpopulation.

Als nächstes werden die Zustandsübergänge als gerichtete Kanten zwischen benachbarten Knoten eingefügt:

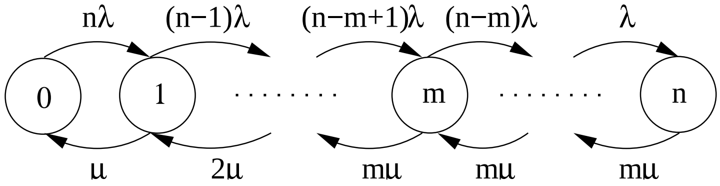


Abbildung 4.3: Exemplarische Erstellung eines Zustandsdiagramms

- Die Ankunftsrate bewirkt den Wechsel in höhere Zustände; ist die Population unbegrenzt, ist sie stets λ. Bei $n_3 < \infty$ jedoch nimmt sie mit wachsender Zahl von Kunden im System ab. Der erste Übergang (von 0 auf 1) geht dann mit $n_3\lambda$ vor sich, danach geht die Geschwindigkeit schrittweise bis λ zurück, danach ist sie Null.
- Die Bedienrate bewirkt den Wechsel in niedrigere Zustände; gibt es im System n_1 Server, so wird die maximale Auslastung folglich erst bei n_1 Kunden im System erreicht. Entsprechend steigt die Bedienrate von zunächst μ schrittweise auf $n_1\mu$ an und bleibt dann gleich.

Natürlich braucht man nicht alle Knoten aufzutragen, sondern nur diejenigen, wo es zu einer gravierenden Änderung kommt, die sich in einer Änderung der Formeln niederschlägt. Exemplarisch ist dies in Abbildung 4.3 dargestellt.

Nun werden Bilanzgleichungen für den stationären Zustand aufgestellt. Es ergeben sich dabei unterschiedliche Gleichungen für die unterschiedlichen Teilbereiche des Diagramms. Die Zustandswahrscheinlichkeit für den Zustand i wird mit π_i bezeichnet. In den Gleichungen stellt man die Wahrscheinlichkeit, daß der Zustand von den umliegenden aus erreicht wird, und die, daß der Zustand verlassen wird, gegenüber. Im nächsten Abschnitt wird dies beispielhaft für ein $M|M|1$-System vorgeführt.

Nun werden aus den einzelnen Bilanzgleichungen allgemeine Formeln für die π_i ermittelt (ggf. bereichsweise), und zwar als $\pi_i = \pi_0 \cdot f(i)$. Dies läßt sich durch sukzessives Einsetzen bewerkstelligen. π_0 gewinnt man dann durch Ausnutzung der Normierungsbedingung:

$$\sum \pi_i = 1 \Leftrightarrow \sum \pi_0 f(i) = 1 \Leftrightarrow \pi_0 \cdot \sum f(i) = 1 \Leftrightarrow \pi_0 = \left(\sum f(i)\right)^{-1}$$

Es sei darauf hingewiesen, daß in der Summenauflistung $\pi_0 = 1 \cdot \pi_0$ auch vorkommt, also $f(0) = 1$ nicht vergessen! Eine Tabelle einer π_i-Formeln findet sich in Abschnitt 4.5.3.

Da π_0 nun bekannt ist, lassen sich die π_i leicht berechnen. Der Rest ist nun einfach:

1. Zwischenankunftszeit und Bedienzeit: $E\{T_A\} = \frac{1}{\lambda}$, $E\{T_S\} = \frac{1}{\mu}$ (exponentialverteilt)
2. Kundenzahl im System: $E\{N\} = \sum i \cdot \pi_i$ (wegen Definition des Erwartungswert)
3. Verweilzeit im System: $E\{T\} = \frac{1}{\lambda}E\{N\}$ (Little'sche Formel)
4. Wartezeit: $E\{T_Q\} = E\{T\} - E\{T_S\} = E\{T\} - \frac{1}{\mu}$ (Summenformel für Verweilzeit)
5. Kunden in der Queue: $E\{N_Q\} = \lambda E\{T_Q\}$ (Little'sche Formel)
6. Kunden im Server: $E\{N_S\} = E\{N\} - E\{N_Q\}$ (Summenformel für Kundenzahl)

Wichtig sind auch die folgenden Werte:

- Auslastung: $\rho = \frac{\lambda}{\mu}$ (wobei μ die Gesamtbedienrate, also egtl. $n_1\mu$ ist bei n_1 Servern)

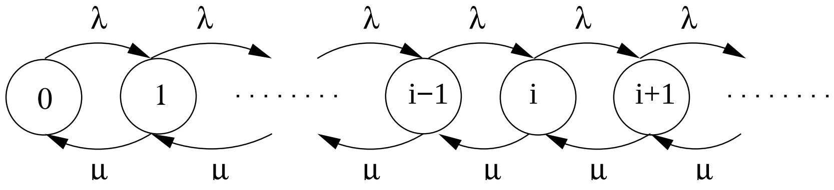


Abbildung 4.4: Zustandsdiagramm M/M/1-Bediensystem

- Abweisungswahrscheinlichkeit (bei begrenzter Queue mit Länge k und mehr als k Kunden):

$$a = \lambda\pi_k$$

- Stabilität (die Systemanalyse gilt nur für stabile Systeme): Ein System mit begrenzter Queue oder Kundenpopulation ist stets stabil. Für ein System mit unendlich vielen Zuständen gilt: Das System ist stabil genau dann wenn $\rho < 1$.
- Durchsatz (Wahrscheinlichkeit, daß das System genutzt wird):

$$d = \sum \mu_i \cdot \pi_i =_{(\mu\,\text{konst})} \mu(1-\pi_0)$$

4.5.2 Analyse des M/M/1-Systems

Gegeben seien exponentialverteilte Zwischenankunftszeit $E\{T_A\}$ und Bedienzeit $E\{T_S\}$. Zunächst ist dann $\lambda = \frac{1}{E\{T_A\}}$ und $\mu = \frac{1}{E\{T_S\}}$. Mit einem Server, unbegrenzter Queue und unbegrenzter Kundenpopulation $(M|M|1|\infty|\infty)$ ergibt sich das in Abbildung 4.4 gezeigte Zustandsdiagramm.

Sei das System in einem stabilen Zustand, dann kann man die Bilanzgleichungen aufstellen:

$$\begin{aligned}
\lambda\pi_0 &= \mu\pi_1 \\
(\lambda+\mu)\pi_1 &= \lambda\pi_0 + \mu\pi_2 \\
&\dots \\
(\lambda+\mu)\pi_i &= \lambda\pi_{i-1} + \mu\pi_{i+1} \text{ (fuer } i > 0)
\end{aligned}$$

Damit ergibt sich zunächst $\pi_0 = \frac{\mu}{\lambda}\pi_1$, eingesetzt in die zweite Gleichung

$$(\lambda+\mu)\pi_1 = \lambda\pi_0 + \mu\pi_2 = \mu(\pi_1+\pi_2) \Leftrightarrow \lambda\pi_1 = \mu\pi_2 \Leftrightarrow \pi_1 = \frac{\mu}{\lambda}\pi_2$$

Insgesamt ergibt sich $\pi_i = \rho\pi_{i-1}$ (für $i > 0$), durch rekursives Einsetzen erhält man $\pi_i = \rho^i\pi_0$.

Mit der Normierungsbedingung $\sum \pi_i = 1$ ergibt sich:

$$\sum \pi_i = 1 \Leftrightarrow \sum \rho^i\pi_0 = 1 \Leftrightarrow \pi_0 \cdot \sum \rho^i = 1 \Leftrightarrow \pi_0 \cdot \frac{1}{1-\rho} = 1 \Leftrightarrow \pi_0 = 1-\rho$$

Mit dem nun bekannten π_0 lassen sich die π_i ausrechnen als $\pi_i = (1-\rho)\rho^i$. Damit ergibt sich:

1. $E\{N\} = \sum i\pi_i = (1-\rho)\sum i\rho^i = \dots = \frac{\rho}{1-\rho}$
2. $E\{T\} = \frac{1}{\lambda} \cdot E\{N\} = \frac{\rho}{\lambda(1-\rho)} = \frac{1}{\mu(1-\rho)} = \frac{1}{\mu-\lambda}$
3. $E\{T_Q\} = E\{T\} - \frac{1}{\mu} = \frac{\rho}{1-\rho}\frac{1}{\mu}$
4. $E\{N_Q\} = \lambda E\{T_Q\} = \frac{\rho^2}{1-\rho}$
5. $E\{N_S\} = E\{N\} - E\{N_Q\} = \frac{\rho}{1-\rho} - \frac{\rho^2}{1-\rho} = \rho$

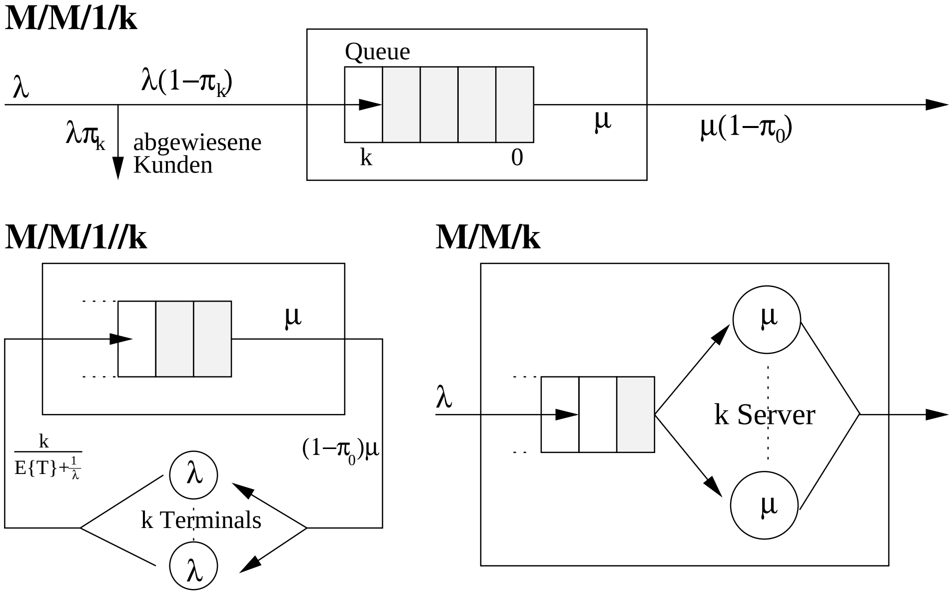


Abbildung 4.5: Einige M/M/1-Systeme.

4.5.3 Weitere exponentialverteilte Systeme

Abbildung 4.5 zeigt einige M/M/k-Systeme :

1. **M/M/1/k**: Bounded Buffer System. Queue mit k Plätzen; Kunden, die ankommen, wenn die Queue voll ist, werden abgewiesen.
2. **M/M/1//k**: Terminal-System. System mit endlicher Anzahl von Kunden k; da hier nur k Terminals zur Verfügung stehen, können nur maximal k Kunden im System sein.
3. **M/M/k**: Multiuser-System mit k Servern; bei unendlich großer Queue werden alle ankommenden Kunden möglichst schnell bearbeitet.
4. **M/M/k/k**: Telefonsystem für k Kunden (M/M/k-Verlustsystem). Da die Queue hier begrenzt ist, werden Kunden abgewiesen, die ankommen, wenn die Queue voll ist.

Die folgende Tabelle zeigt eine Übersicht der Formeln für die Zustandswahrscheinlichkeiten einiger exponentialverteilter Systeme. Die Erwartungswerte lassen sich dann gewohnt schnell berechnen.

	M/M/1	M/M/1/k	M/M/1//k	M/M/k
ρ	$\frac{\lambda}{\mu}$	$\frac{\lambda}{\mu}$	$\frac{\lambda}{\mu}$	$\frac{\lambda}{k\mu}$
π_0	$1-\rho$	$\frac{1-\rho}{1-\rho^{k+1}}$	aus π_i	aus π_i
π_i	$(1-\rho)\rho^i$	$\pi_0 \cdot \rho^i$	$\frac{k!}{(k-i)!}\rho^i\pi_0$	$\frac{(k\rho)^i}{i!}\pi_0 \quad (k \leq i)$ $\frac{k^k\rho^i}{i!}\pi_0 \quad (k > i)$

hierbei bedeutet "aus π_i", daß nach Berechnung der π_i in Abhängigkeit von π_0 der konkrete Wert für π_0 wie normal mittels der Normierungsbedingung $\sum \pi_i = 1$ berechnet werden kann.

Kapitel 5

Prozeß-Synchronisation

Bei (pseudo-)parallelen Prozessen unterscheidet man

- unabhängige Prozesse: keine gegenseitige Abhängigkeit
- kooperierende Prozesse: gegenseitige Abhängigkeit und Beeinflussung, typischer Fall: zeitliche Abstimmung von Operationen oder Austausch von Ergebnissen

5.1 Erzeuger-Verbraucher-Problem

Es gibt zwei Prozesse:

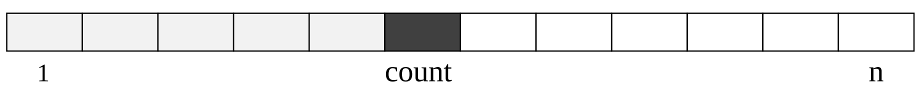
- Erzeuger (Producer): produziert Daten, die in endlichem Puffer abgelegt werden.
- Verbraucher (Consumer): entnimmt die Daten aus dem Puffer und verarbeitet sie.

Probleme der Synchronisation und Koordination:

- Wenn der Puffer voll ist, kann der Producer nichts ablegen
- Wenn der Puffer leer ist, kann der Consumer nichts entnehmen
- Die Daten im Puffer dürfen nicht gleichzeitig von Producer und Consumer verändert werden, um Inkonsistenzen zu vermeiden.

Die dritte Bedingung ist hierbei extrem wichtig; sofern sie mißachtet wird, können Daten mehrfach bearbeitet werden oder verloren gehen.

Fehlerhafte Lösung: LIFO-Speicher



```
var buffer:  array[1..n] of item;
var nextp, nextc:  item;
var count:  integer := 0;
```

Producer:

```
repeat
  produce an item in nextp;
  while (count=n) do noop;
  count:=count+1;
  buffer[count]:=nextp;
until false;
```

Consumer:

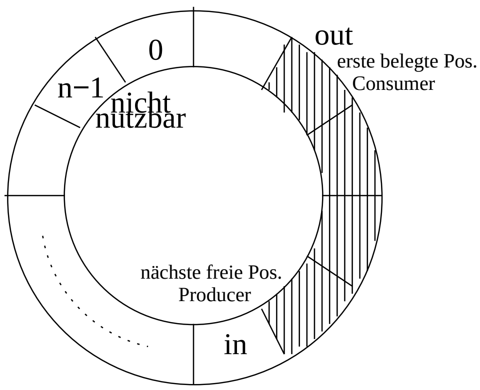
```
repeat
  while (count=0) do noop
  nextc:=buffer[count];
  count:=count-1;
  consume the item in nextc;
until false;
```

Probleme:

- Die Ankunftsreihenfolge wird bei Abarbeitung nicht eingehalten (LIFO statt FIFO)
- Inkonsistenzen: Daten gehen verloren bei Unterbrechung zwischen den kursivgedruckten Bereichen. Abhilfe: diese Programmteile müssen atomar sein.

Korrekte Lösung: Ringspeicher

Ringspeicher (FIFO-Speicher): Von den n angelegten Plätzen können maximal $n-1$ gefüllt werden. Der Speicher ist genau dann leer, wenn $in = out$, und genau dann voll, wenn $(in+1)\, mod\, n = out$.




```
var buffer:  array[0..n-1] of item;
    nextp, nextc:  item;
    in, out:  [0..n-1]:=0;
```

Producer:

```
repeat
  produce an item in nextp;
  while ((in+1) mod n=out) do
    noop;
  buffer[in]:=nextp;
  in:=(in+1) mod n;
until false;
```

Consumer:

```
repeat
  while (in=out) do noop;
  nextc:=buffer[out];
  out:=(out+1) mod n;
  consume the item in nextc;
until false;
```

Warum gibt es keine Probleme? Es gibt keine Variable, die von beiden Prozessen verändert wird (allerdings werden Variablen von beiden Prozessen gelesen: in/out).

Neuer Lösungsansatz: globale Variable `count` und n nutzbare Speicherplätze statt bisher $n-1$. Ziel: Erweiterung auf mehrere Consumer/Producer sollte möglich sein.

```
var buffer:  array[0..n-1] of item;
    nextp, nextc:  item;
    in, out:  [0..n-1] := 0;
    count:  int := 0;
```

Producer:

```
repeat
  produce an item in nextp;
  while (count=n) do noop;
  buffer[in]:=nextp;
  in:=(in+1) mod n;
  count:=count+1;
until false;
```

Consumer:

```
repeat
  while (count=0) do noop
  nextc:=buffer[out];
  out:=(out+1) mod n;
  count:=count-1;
  consume the item in nextc;
until false;
```

Problem: Veränderung der Variablen `count` muß atomar sein, andernfalls kommt es zu Inkonsistenzen. Das geht nur dann, wenn auch die Maschinencode-Umsetzung atomar ist:

```
MOV reg, count;
INC reg, 1;              bzw. DEC reg, 1;
DEC count, reg;
```

Race-Condition: Situation, in der mehrere Prozesse gemeinsame Daten "gleichzeitig" manipulieren und das Ergebnis von der Reihenfolge der Ausführung abhängt.

5.2 Wechselseitiger Ausschluß und kritischer Bereich

engl. mutual exclusion and critical section

Bei kooperierenden Prozessen $P_0..P_{n-1}$ muß es Ziel sein, unkontrollierte "gleichzeitige" Manipulation von gemeinsamen Daten zu vermeiden. Dazu hat jeder Prozeß einen Bereich (kritischer Bereich, KB), in dem er auf gemeinsame Daten zugreift, umschlossen von Bereichen, in denen er nur auf lokale Daten zugreift (unkritischer Bereich, UKB). Die Forderung lautet dann, daß sich zu jedem Zeitpunkt nur ein Prozeß in seinem KB befinden darf.

Der Programmcode für Prozeß P_i sieht demnach wie folgt aus:

```
repeat
  entry section;
  critical section;
  exit section;
  remainder section;
until false;
```

Eine geeignete Synchronisation erfüllt drei Anforderungen:

1. **Mutual Exclusion**: Zu jedem Zeitpunkt darf sich nur ein Prozeß in seinem KB befinden.
2. **Progress Requirement**: Falls sich kein Prozeß im KB befindet und es Prozesse gibt, die ihren KB betreten möchten, dann wird die Entscheidung, welcher Prozeß den KB betreten darf, nur unter diesen Prozessen getroffen und außerdem in endlicher Zeit. Insbesondere darf es keinen Einfluß haben, ob ein Prozeß in seinem UKB stirbt.
3. **Bounded Waiting**: Ein Prozeß darf nicht unendlich lange auf Einlaß in den KB warten müssen, d.h. es gibt eine natürliche Zahl n, so daß jeder Prozeß, nachdem er Einlaß in den KB gefordert hat, den anderen Prozessen maximal n-mal den Vortritt läßt. Die schließt aus logischen Gründen Prioritätsregelungen aus.

5.3 Petersen-Algorithmus

Der Petersen-Algorithmus (1981) ist eine reine Softwarelösung des Synchronisationsproblems für zwei Prozesse und soll hier in vier Stufen entwickelt werden. Der Code wird jeweils angegeben für P_i $(i = 0, 1)$.

Stufe 1: gemeinsame Synchronisationsvariable

Ansatz: gemeinsame Synchronisationsvariable `turn`, falls `turn=i`, darf Prozeß P_i den KB betreten.

```
var turn:  [0..1] := 1;

repeat
  while (turn≠i) do noop;
  critical section;
  turn:=1-i;
  remainder section;
until false;
```

Eigenschaften:

- Mutual Exclusion: wegen turn kann nur ein Prozeß seinen KB betreten.
- Progress Requirement ist verletzt: der Algorithmus fordert striktes Alternieren; stirbt P_i im UKB, kann P_{1-i} höchstens noch einmal in seinen KB.

Stufe 2: Setzen eines Flags im Kritischen Bereich

Ansatz: Statt `turn` wird eine boolesche Variable `flag[i]` für jeden Prozeß P_i verwendet (lesbar von beiden Prozessen). Vor KB-Eintritt setzt P_i diese Variable auf `true` und nach dem KB-Eintritt auf `false`.

```
var flag:  array[0..1] of boolean := false;

repeat
  while flag[1-i] do noop;
  flag[i]:=true;
  critical section;
  flag[i]:=false;
  remainder section;
until false;
```

Eigenschaften:

- Mutual Exclusion ist verletzt: Gelangen beide Prozesse gleichzeitig in die Schleife, hat noch keiner sein Flag gesetzt, und beide können den KB betreten.

Stufe 3: Verletzung der Mutual Exclusion beheben

Ansatz: Setzen des Flags vor der WHILE-Schleife:

```
var flag:  array[0..1] of boolean := false;

repeat
  flag[i]:=true;
  while flag[1-i] do noop;
  critical section;
  flag[i]:=false;
  remainder section;
until false;
```

Eigenschaften:

- Mutual Exclusion ist erfüllt
- Progress Requirement ist verletzt: Setzen die Prozesse unmittelbar nacheinander ihr Flag, bleiben sie in der Schleife gefangen (Deadlock).

Stufe 4: Petersen-Algorithmus

Ansatz: Verwendung von `turn` und `flag`: P_i beantragt KB-Einlaß mittels `flag[i]:=true`, dann setzt er `turn:=1-i` (läßt also dem anderen Prozeß vortritt), und wartet bis der andere Prozeß den KB verlassen hat. Erst dann betritt er den KB:

```
var turn:  [0..1];
    flag:  array[0..1] of boolean := false;

repeat
  flag[i]:=true;                             Pi will in KB
  turn:=1-i;                                 Pi sagt "P1-i hat Vortritt"
  while (flag[1-i] and turn=1-i) do noop;    Falls P1-i in KB will und
  critical section (Pi);                       Vortritt hat wartet Pi
  flag[i]:=false;                            Pi sagt "Habe KB verlassen"
  remainder section (Pi);
until false;
```

Eigenschaften:

- Mutual Exclusion, Progress Requirement und Bounded Waiting sind erfüllt.

5.4 Bakery Algorithmus

Bakery Algorithmus (Lemport 1974), Erweiterung auf n Prozesse.

Prinzip: Ausgabe von Wartenummern für wartende Prozesse. Der Prozeß mit der kleinsten Wartenummer ist als nächstes dran. Bei mehreren Prozessen mit gleicher Wartenummer hat der Prozeß mit der niedrigsten Prozeßnummer Vorrang. Die Prozesse seien P_i $(i = 0..n-1)$.

`number[i]` ist die Wartenummer für den KB-Eintritt
`choosing[i]` ist `true`, falls P_i Wartenummer ziehen möchte, sonst `false`.

$(a, b) < (c, d)$ heißt: entweder $(a < c) \vee ((a = c) \wedge (b < d))$

```
var number:  array[0..n-1] of integer := 0;
    choosing:  array[0..n-1] of boolean := false;

repeat
  choosing[i]:=true;                       "Ich ziehe Nummer"
  number[i]:=1+max{number[]};               Nummer ziehen (ggf. nicht eindeutig)
  choosing[i]:=false;
  for j := 0 to n-1 do                     Überprüfe andere Prozesse:
    while (choosing[j]) do noop;            Warte auf gültige Nummer
    while (number[j]>0 and                  Will er rein?
           (number[j],j) < (number[i],i)    Ist er vorher dran?
           ) do noop;
  end;
  critical section;
  number[i]:=0;                             Fertig!
  remaining section;
until false;
```

5.5 Hardware für Synchronisation

Varianten:

- Interrupt-Ausschalten
- Test and Set - Befehl
- Swap-Befehl

5.5.1 Interrupt-Ausschalten

Mit ausgeschalteten Interrupts wird der Prozeßwechsel:

```
repeat
  disable interrupts
  critical section;
  enable interrupts;
  remainder section;
until false;
```

Schwerwiegende Nachteile:

- Ein Benutzerprozeß sollte nicht über Interrupts verfügen können. Bei Mißbrauch oder Fehlern drohen zu große Risiken.
- Bei Multiprozessor-Systemen ist diese Methode entweder nicht möglich oder zu umständlich.
- Mutual Exclusion und Progress Requirement sind erfüllt.

5.5.2 Test and Set

Dieser Befehl entspricht der atomaren Hardwarefunktion

```
function TestAndSet (var target: boolean): boolean;
begin
  TestAndSet := target;
  target := true;
end;
```

Die Variable `target` wird gesetzt (und returniert) und - falls sie ungleich `true` ist - auf `true` gesetzt. Innerhalb der Funktion kann keine Unterbrechung eintreten.

Einfache Lösung Diese Lösung erfüllt zwar Bounded Waiting nicht, da keine strikte Reihenfolge eingehalten wird (der Test hat keine Ablaufordnung!). Sie zeigt aber die grundsätzliche Funktionsweise der Test-and-Set-Anweisung.

```
var lock: boolean := false;

repeat
  while TestAndSet(lock) do noop;
  critical section;
  lock:=false;
  remainder section;
until false;
```

Korrekte Lösung Diese Lösung erfüllt alle drei Anforderungen.

```
var waiting: array [0..n-1] of boolean :=false;  Pi wartet nicht
    lock: boolean :=false;                       KB frei

localvar j: [0..n-1];
         key: boolean;

repeat
  waiting[i]:=true;                      “Warte”
  key:=true;
  while(waiting[i] and key) do           Solange wartend und KB belegt...
    key:=TestAndSet(lock);
  waiting[i]:=false;
  critical section;
  j:=(i+1) mod n;                        Pi prüft jeden Prozeß Pj und findet
  while (j!=i and not waiting[j]) do      den ersten wartenden Prozeß
    j:=(j+1) mod n;                      Dann entweder
  if (j!=i) then                         es wartet kein Prozeß (KB wird frei)
    lock:=false;
  else                                   oder
    waiting[j]:=false;                   an wartenden Prozeß Signal: warte nicht
  remainder section;
until false;
```

5.5.3 Swap-Befehl

Dieser Befehl ist atomar und entspricht der Hardwarefunktion zu

```
procedure Swap (var a, b:  boolean);
var temp:  boolean;
begin
  temp:=a;
  a:=b;
  b:=temp;
end;
```

Analog zur Test-And-Set-Anweisung kann man wie folgt mit Swap synchronisieren (hier wieder die einfache Variante ohne Bounded Waiting):

```
var lock:  boolean := false;

localvar key:  boolean := true;

repeat
  key:=true;
  repeat
    Swap(lock, key);
  until not key;
  critical section;
  lock:=false;
  remainder section;
until false;
```

5.6 Semaphore

Ansatz: Es gibt spezielle Systemaufrufe des OS für die globale (Zähler-)Variable S:

wait(S)

`while` ($S \leq 0$) `do noop;`	zw. Ende der Schleife und Dekr. keine Unterbr.
$S := S - 1$;	atomar

signal(S)

$S := S + 1$;	atomar

Wechselseitiger Ausschluß mit Semaphoren

```
var mutex:  integer := 1;      Semaphor

repeat
  wait(mutex);
  critical section;
  signal(mutex);
  remainder section;
until false;
```

Synchronisation mit Semaphoren

Problem: Zwei Prozesse mit festgelegter zeitlicher Abfolge (erst P_1, dann P_2)

```
var synch:  integer := 0;
```
Semaphor

Prozeß P_1***:***

```
statements;
signal(synch);
```

Prozeß P_2***:***

```
wait(synch);
statements;
```

Busy Waiting bei Semaphoren

In der bisherigen Implementation ist das "Busy Waiting" ein Nachteil der Semaphore: das Warten auf Einlaß verbraucht CPU-Zeit. Bei kurzer Wartezeit ist dies zwar akzeptabel, bei langer Wartezeit sollte sich der wartende Prozeß jedoch "schlafen legen" (d.h. er wird blockiert).

Implementierung:

```
type semaphore = record
  lock:  boolean := false;
  value: integer := 1;
  list:  queue of processes := ();
end;
```

wait(S)

```
while (TestAndSet(S.lock)) do noop;
S.value:=S.value-1;
if (S.value<0) then begin
  add this process to S.list;
  block this process;
end;
S.lock:=false;
```

signal(S)

```
while (TestAndSet(S.lock)) do noop;
if (S.value<0) then begin
  remove a process P from S.list;
  wakeup(P);
end;
S.value:=S.value+1;
S.lock:=false;
```

Ist die `value`-Komponente dabei positiv, so gibt sie an, wieviele Prozesse noch in den KB dürfen; ist sie dagegen negativ, gibt sie an, wieviele auf Einlaß warten.

5.7 Klassisches Synchronisationsproblem

1. **Erzeuger-Verbraucher-Problem**
   Lösung mit Semaphoren: `mutex` (Schutz des Speichers), `full` (zählt die vollen Pufferplätze) und `empty` (zählt die leeren Pufferplätze), wobei `full+empty=max` invariant bleibt.

2. **Reader-Writer-Probleme**
   Anwendung: Verwaltung von Datenbanken, bspw. Reservierungssystemen; 3 Varianten:

   (a) Reader haben Priorität vor Writern, außer wenn gerade ein Writer in seinem KB ist.
   Problem: "starvation" - Reader verbünden sich, Writer kommen nicht dran

   (b) Writer haben Priorität
   Problem: "starvation" der Reader

   (c) Alternation von Reader- und Writer-Phasen ("Fairness")

3. **Dining Philosophes Problem**
   Ein eher theoretisches Problem: Fünf Philosophen sitzen an einem runden Tisch, vor ihnen jeweils ein Teller, und zwischen jedem Teller liegt eine Gabel. Die Philosophen denken eine gewisse Zeit lang und werden dadurch hungrig. Daher greifen sie zu zwei Gabeln (rechts und links vom Teller), und essen, sobald sie beide Gabeln haben. Schließlich denken sie weiter.
   Probleme: Deadlock (bspw. wenn jeder die rechte Gabel ergreift) sowie Starvation

5.8 High Level Konstrukte

Der Nachteil der Semaphore ist, daß sie problemanfällig sind. Da die Aktivierung und Aufhebung der Semaphor-Sperren direkt vom Programmierer implementiert werden, kommt es sehr häufig zu Fehlern, die dann zu Deadlocks führen. Um dies zu umgehen, gibt es einige High-Level Konstrukte, die wesentlich einfacher zu handhaben sind als Semaphore.

5.8.1 Bedingte kritische Regionen

Prozeß entspricht lokalen Daten und einem sequentiellen Programm. Globale Daten (deklariert als `shared type`) sind allen Prozessen zugänglich.

Typischer Fall:

`region v when B do S`

`B=true`: Prozeß betritt KB, sofern dieser frei ist. Falls der KB nicht frei ist, wird er blockiert.

`B=false`: Prozeß wartet (wird blockiert), bis `B=true` und somit der KB betreten werden kann

Sequentielle Prozeßabfolge Zwei Prozesse P_1 und P_2, die sequentiell ablaufen müssen (d.h. P_1, P_2 oder P_2, P_1):

P_1: **region** v **when true do** S_1

P_2: **region** v **when true do** S_2

Erzeuger-Verbraucher-Problem Lösung mittels kritischer Regionen

```
var buffer:  shared record
              pool:  array[0..n-1] of item;
              count, in, out:  int;
            end;
```

Producer:

```
repeat
  produce an item in nextp;
  region buffer when count<n
    do begin
      pool[in]:=nextp;
      in:=(in+1) mod n;
      count:=count+1;
    end;
until false
```

Consumer:

```
repeat
  region buffer when count>0
    do begin
      nextc:=pool[out]
      out:=(out+1) mod n;
      count:=count-1;
    end;
  consume item in nextc;
until false;
```

5.8.2 Monitor-Konzept

Ein Monitor ("Überwacher") ist ein abstrakter Datentyp, der sowohl die zu schützenden Daten als auch die zugehörigen Zugriffs- und Synchronisationsmechanismen umfaßt, die "procedure entries" und "conditions" heißen.

Die Conditions sind Warteschlangen wie die Semaphoren, aber ohne Zähler. Auf Conditions sind die folgenden Operationen definiert:

- `wait(cond)`: der ausführende Prozeß blockiert sich selbst und wartet bis ein anderer Prozeß eine `signal`-Operation ausführt.
- `signal(cond)`: Der nächste Prozeß in der Warteschlange wird reaktiviert. Ein Signal hat keinen Effekt, falls die Warteschlange leer ist, daher sollte dieser Fall ggf. zusätzlich abgesichert werden.

Kapitel 6

Deadlocks

Ein Deadlock tritt auf, wenn mehrere Prozesse auf ein Ereignis warten, daß nicht mehr eintreten wird. Dies ist bspw. der Fall, wenn ein Prozeß auf eine Ressource wartet, die von einem anderen Prozeß belegt wird, und dieser seinerseits auf eine vom ersten Prozeß belegte Ressource wartet.

6.1 Ressource-Allocation-Graph

Ein Ressource Allocation Graph ist ein Graph, der einen Prozeßzustand in Hinblick auf die Ressourcen grafisch darstellt. Dazu gibt es zweierlei Arten von Knoten: Prozesse $P_1, ..., P_n$ und Ressourcentypen $R_1, ..., R_n$ (von denen jeder Typ in mehreren Exemplaren vorkommen kann, falls dies so ist, werden “Unterknoten” in den Ressourcenknoten gemalt). Anforderungen und Zuteilungen werden durch gerichtete Kanten ausgedrückt:

- $P_i \rightarrow R_j$: Prozeß P_i fordert Ressource R_j an (und zwar 1 Exemplar) und wartet auf Zuteilung
- $R_j \rightarrow P_i$: Die Ressource R_j (1 Exemplar) ist dem Prozeß P_i zugeordnet

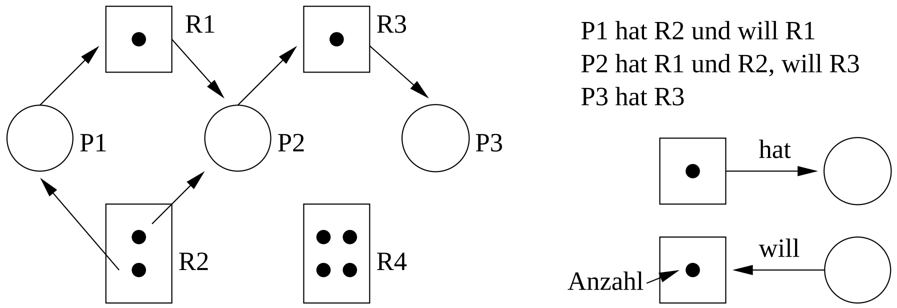
Zeitliche Veränderung:

1. Einfügen einer Anforderung: $P_i \rightarrow R_j$
2. Umwandeln in Zuteilung: $R_j \rightarrow P_i$
3. Löschen der Zuteilung: $R_j \quad P_i$

Wait-For-Graph

Entsteht durch Weglassen der Ressourcentypen. $P_i \rightarrow P_j$ bedeutet dann: P_i wartet auf die Freigabe von durch P_j belegten Ressourcen.

Erstes Beispiel (keine Zyklen) Es gibt keinen Deadlock (P_3 arbeitet bis Ende, P_2 bekommt R_3 und arbeitet bis Ende, P_1 bekommt R_1 und arbeitet bis Ende)

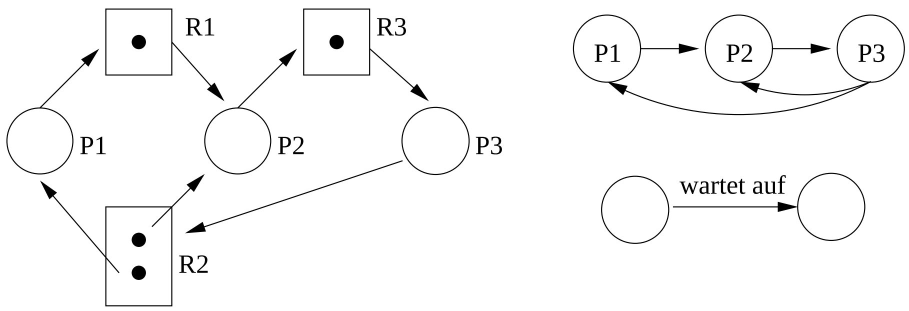


Zweites Beispiel (Graph mit 2 Zyklen) Es gibt einen Deadlock, weil neben P_1 und P_2 jetzt auch P_3 warten muß und keiner der drei Prozesse fortgeführt werden kann. Allgemein gilt:

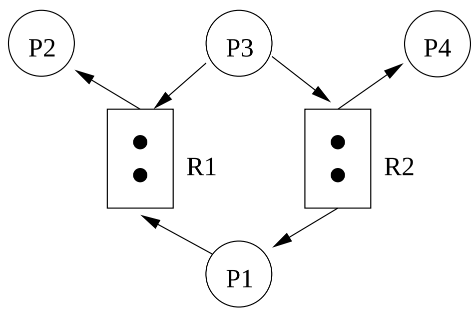
$$\text{Kein Zyklus im Graph} \Rightarrow \text{kein Deadlock}$$

Falls jeder Ressourcentyp nur ein Exemplar hat gilt sogar

$$\text{Zyklus im Graph} \Leftrightarrow \text{Deadlock}$$



Drittes Beispiel (Zyklus ohne Deadlock) Es gibt keinen Deadlock (P_4 wird beendet, P_3 erhält R_2 und arbeitet bis Ende, P_1 erhält R_1 und wird beendet, P_2 wird beendet).



6.2 Charakterisierung von Deadlocks

Es gibt vier notwendige Bedingungen für einen Deadlock:

1. **Circular Wait**: Es muß eine zyklische Kette wartender Prozesse geben.
2. **Exclusive Use** (Mutual Exclusion): Die fraglichen Ressourcen können nur von einem Prozeß belegt werden. Jeder andere Prozeß muß warten, bis die Ressource frei wird.
3. **Hold and Wait** (Folgerung aus 1): Es gibt mindestens einen Prozeß, der eine oder mehrere Ressourcen belegt hat, behält, und auf die Zuteilung von durch wartende Prozesse blockierten Ressourcen wartet.
4. **No Preemption**: Eine zugeteilte Ressource kann einem Prozeß nicht entzogen werden. Nur der Prozeß selbst kann die Ressource freigeben.

Es gibt verschiedene Möglichkeiten, auf Deadlocks zu reagieren:

- **Deadlock-Prevention**: Der Eintritt von mindestens einer der vier Bedingungen wird unmöglich gemacht.
- **Deadlock-Avoidance**: Jeder Prozeß muß seinen maximalen Ressourcenbedarf anmelden. Mit dieser Information wird geprüft, ob eine Ressourcenanforderung gewährt wird.
- **Deadlock-Detection**: Deadlocks werden zugelassen, aber es werden Methoden bereitgestellt, um Deadlocks zu erkennen und zu beheben.
- **keine Maßnahmen**: Die Möglichkeit von Deadlocks wird als so klein eingestuft, daß das Problem ignoriert wird (bspw. UNIX, Linux).

6.3 Deadlock-Prevention

Man versucht, eine der vier notwendigen Vorraussetzungen unmöglich zu machen:

- **Ressourcen-Sharing**, um Bedingung 2 zu vereiteln. Geht häufig aus praktischen Gründen nicht.
- **Preemption** (gegen 4), d.h. Eingriffe von außen sind möglich. Schlecht, weil "Gewaltmaßnahme".
- **"Alles oder nichts"** (gegen 3), Prozesse fordern alle möglicherweise benötigten Mittel an. Sehr ineffizient.
- **Unmöglichkeit eines Zyklus** (gegen 1), algorithmisch häufig durch Einführung einer Ressourcen-Hierarchie verwirklicht. Die Ressourcen werden in Klassen $K_1, ...$ eingeteilt. Wenn nun ein Prozeß eine Ressource der Klasse K_r (und niedrigerer Klassen) belegt, darf er nur auf Ressourcen aus höheren Klassen warten ($K_{r+1}, ...$). Sollte er Ressourcen niedrigerer Klassen benötigen, muß er zunächst entsprechend höhere belegte Ressourcen freigeben.

6.4 Deadlock-Avoidance

Vorraussetzung ist, daß jeder Prozeß seinen maximalen Ressourcenbedarf vorher anmeldet. Vor jeder Zuteilung einer Ressource prüft dann das OS, ob garantiert das Eintreten der Circular-Wait-Bedingung verhindert werden kann und damit ein Deadlock unmöglich wird.

“Sicherer Zustand”: Es gibt eine Reihenfolge von Prozessen, so daß ein Prozeß niemals mehr Ressourcen benötigt als frei sind, wenn alle vorherigen Prozesse in der Liste beendet wurden.

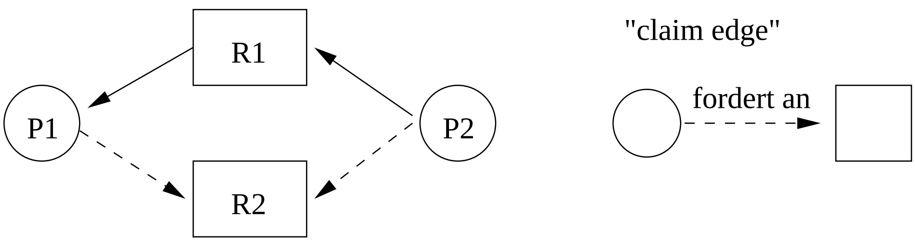
Beispiel: 3 Prozesse und 1 Ressourcetyp mit 12 Exemplaren

	P_1	P_2	P_3	frei	P_1	P_2	P_3	frei
max.	10	4	9		10	4	9	
aktuell	5	2	2	3	5	2	3	2
P_2	5	4	2	1	5	4	3	0
	5	0	2	5	5	0	3	4
P_1	10	0	2	0				
	0	0	2	0				
P_3	0	0	9	3				
	0	0	0	12				

Der Zustand (5,2,2) ist also sicher, der Zustand (5,2,3) jedoch nicht.

Erweiterung des Ressource-Allocation-Graph

Kanten für potentielle Anforderungen (“claim edges”) werden gestrichelt dargestellt.



6.4.1 Bankers Algorithmus

Der Algorithmus verfährt in ähnlicher Weise wie ein Banker, der sicher gehen will, daß er die Kreditwünsche aller seiner Kunden in einer geeigneten Reihenfolge befriedigen kann.

Beispiel: 5 Prozesse (P1..P5) und 3 Ressourcetypen (A,B,C) mit (10,5,7) Exemplaren.

	Allocation (A,B,C)	Maximum (A,B,C)	Need (A,B,C)
P1	(0,1,0)	(7,5,3)	(7,4,3)
P2	(2,0,0)	(3,2,2)	(1,2,2)
P3	(3,0,2)	(9,0,2)	(6,0,0)
dP4	(2,1,1)	(2,2,2)	(0,1,1)
P5	(0,0,2)	(4,3,3)	(4,3,1)
Sum	(7,2,5)		
Available	(3,3,2)		

(P2,P4,P5,P3,P1) ist sichere Reihenfolge:

	Allocation (A,B,C)	Need (A,B,C)	Available (A,B,C)
P2	(2,0,0)	(1,2,2)	(3,3,2)
P4	(2,1,1)	(0,1,1)	(5,3,2)
P5	(0,0,2)	(4,3,1)	(7,4,3)
P3	(3,0,2)	(6,0,0)	(7,4,5)
P1	(0,1,0)	(7,4,3)	(10,4,7)

Weitere Beispiele:

- Available(3,3,2), P2 fordert (1,0,2)
  Wird erfüllt, da der neue Zustand sicher ist.
- Available(2,3,0), P5 fordert (3,3,0)
  Nicht erfüllbar, weil Ressourcen nicht verfügbar sind
- Available(2,3,0), P1 fordert (0,2,0)
  Wird nicht erfüllt, weil der resultierende Zustand nicht sicher wäre (nachprüfen!)

Der Bankers Algorithmus

Prozesse P_i $(i = 1..n)$, Ressourcentypen R_j $(j = 1..m)$

Vier Arrays:

- `Max[i,j]`: maximaler Bedarf des Prozesses P_i an R_j, der bei Prozeßstart bekannt sein muß
- `Alloc[i,j]`: Anzahl Exemplare von R_j, die P_i belegt
- `Avail[j]`: Zur Zeit verfügbare Exemplare von R_j
- `Need[i,j]:=Max[i,j]-Alloc[i,j]`: Möglicher Bedarf des Prozesses P_i an R_j

Der Bankers Algorithmus besteht aus zwei Teilalgorithmen. Der Ressource-Request-Algorithmus gewährt letztlich die Ressourcen. Der Safety-Algorithmus wird von diesem Algorithmus lediglich genutzt, um die Sicherheit einer Zuteilung zu überprüfen. Er spielt aber für die Deadlock Detection eine bedeutende Rolle. Hier wird zunächst der Safety-Algorithmus, dann der Ressource-Request-Algorithmus präsentiert:

Safety-Algorithmus

```
var Work[1..m]:=Avail[];
    Finish[1..n]:=(false,...);

repeat
  Suche einen Prozeß Pi mit...
    Finish[i]=false, d.h. Pi ist noch nicht markiert
    Need[i,j]≤Work[j] ∀j, d.h. die restlichen Anforderungen sind erfüllbar
  Falls kein solcher Prozeß existiert, Ende "Zustand unsicher"
  Finish[i]:=true, d.h. markiere Pi als terminiert
  Work[j]:=Work[j]+Alloc[i,j] ∀j, d.h. Pi's Ressourcen sind nach Ende verfügbar
  Falls alle Finish[i]=true, Ende "Zustand sicher"
until false
```

Ressource-Request-Algorithmus

`Request[i,j]=k` "Prozeß P_i fordert k Exemplare des Ressourcentyps R_j an"

Der Algorithmus läuft in 3 Schritten ab:

1. Überprüfe: `Request[i,j]`$\leq$`Need[i,j]` $\forall j$
   sonst Fehler "P_i überschreitet vereinbartes Maximum"
2. Überprüfe: `Request[i,j]`$\leq$`Avail[j]` $\forall j$
   sonst muß P_i warten, da die angeforderten Ressourcen nicht verfügbar sind
3. Teile P_i versuchsweise die angeforderten Ressourcen zu:
   - `Avail[j]:=Avail[j]-Request[i,j]` $\forall j$
   - `Alloc[i,j]:=Alloc[i,j]+Request[i,j]` $\forall j$
   - `Need[i,j]:=Need[i,j]-Request[i,j]` $\forall j$

   Prüfe mit dem Safety-Algorithmus, ob dieser vorläufige Zustand sicher ist:
   - ja: mache die Zuordnung endgültig.
   - nein: mache die Zuweisung rückgängig; P_i muß warten.

Eigenschaften:

- Maximaler Bedarf muß bekannt sein
- Menge der Prozesse bleibt konstant, d.h. es gibt keine neuen Prozesse
- Eine Verbesserung kann erzielt werden, wenn das Maximum dynamisch gehandhabt wird.

6.5 Deadlock-Detection

Fall A: Jeder Ressource-Typ hat genau ein Exemplar

Ressource Allocation Graph, Wait-For-Graph

Fall B: Jeder Ressource-Typ hat mehrere Exemplare

Modifikation des Safety-Algorithmus: statt der maximalen Restanforderung jedes Prozesses P_i, ausgedrückt durch `Need[i,j]`, verwendet man die aktuelle Anforderung `Request[i,j]` in der Vergleichsoperation. Ergebnis:

- `Finish[i]=false`: P_i ist am Deadlock beteiligt
- `Finish[i]=true` $\forall i$: es liegt kein Deadlock vor

Kapitel 7

Hauptspeicherverwaltung

Hauptspeicher ist Ressource, die vom OS verwaltet wird.

7.1 Einfache Verfahren

Overlay-Technik (Monoprogramming) Es gibt Teile des Programms, die niemals gleichzeitig im Hauptspeicher benötigt werden. Diese Teile bekommen denselben Adreßbereich zugeordnet, d.h. sie werden "übereinandergelegt".

Problem: Entwurf dieser Overlay-Struktur erfolgt durch den Programmierer

Dynamic Loading (bei Mono- und Multiprogramming) Eine Subroutine wird erst dann in den Hauptspeicher geladen, wenn sie aufgerufen wird.

Typische Anwendung: Subroutinen für Sonderfälle, Fehlerbehandlung

Dynamic Linking (bei Multiprogramming) Library-Routinen, die von mehreren Prozessen benutzt werden, sollen nur einmal im Hauptspeicher sein. Wird auch Dynamically Linked Library (DLL) oder Shared Libraries genannt.

7.2 Segmentierung

Sicht des Programmierers: logischer Adressraum ist zweidimensional (segment, offset).

Beispiel:

Segment No.	Segment Name	Base	Limit (# Zellen)
0	subroutine A	1400	1000
1	subroutine B	6300	400
2	main program	4300	400
3	stack	3200	1100
4	symbol table	4700	1000

Der Hauptspeicher wird hierbei also in Teile (sogenannte Partitionen) beliebiger Größe unterteilt, die beliebig auf dem Speicherbereich verteilt sein dürfen.

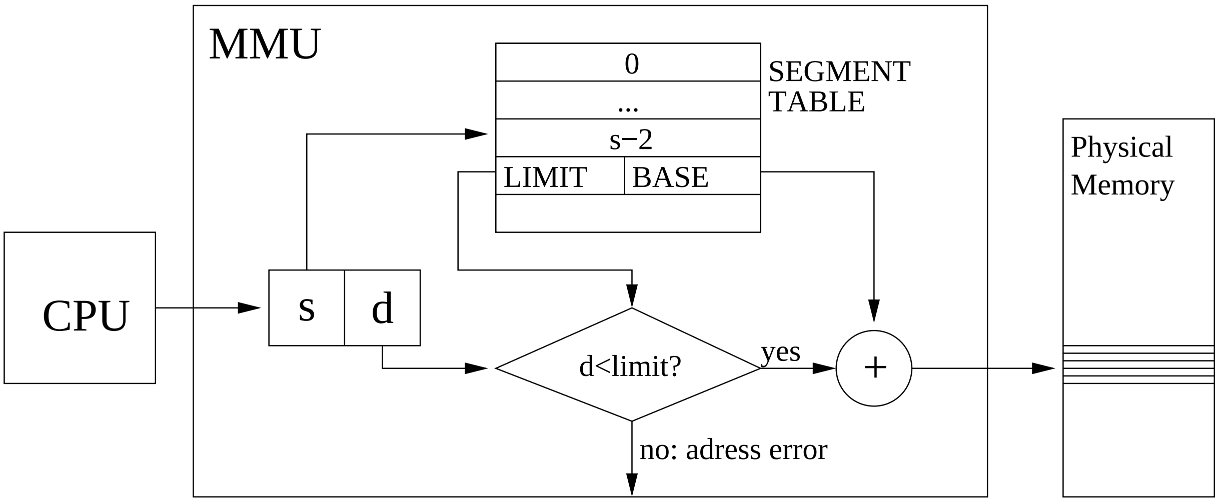


Abbildung 7.1: Funktion der Memory Management Unit (MMU) bei Segmentierung

Belegungsstrategien:

- **First Fit**: das erste passende Loch wird ausgewählt
- **Best Fit**: das kleinste passende Loch wird ausgewählt
- **Worst Fit**: das größte Loch wird ausgewählt
- **Rotating First Fit**: wie First Fit, der Speicher wird jedoch jeweils ausgehend von der letzten Plazierung zirkulär durchsucht.

7.2.1 Fragmentierung

Bei der Segmentierung ergibt sich die externe Fragmentierung als offensichtliches Problem. Damit bezeichnet man die Tatsache, daß zwischen den verschieden großen Partitionen "Löcher" verbleiben. Es kann somit vorkommen, daß - obwohl in der Summe genug freier Speicher vorhanden ist - keines der Löcher groß genug ist, um eine Partition aufzunehmen.

Verwaltung des freien Speichers:

- Bitmaps: 0 frei, 1 belegt
- Linked List: Segmente:=belegte/freie Speicherbereiche

  Liste der Segmente mit Einträgen Flag (belegt/frei), Startadresse, Größe, Zeiger zum nächsten Eintrag

  Dabei ist ein Sortieren nach Startadresse vorteilhaft

Maßnahmen gegen die Fragmentierung:

- Garbage Collection
- Compaction

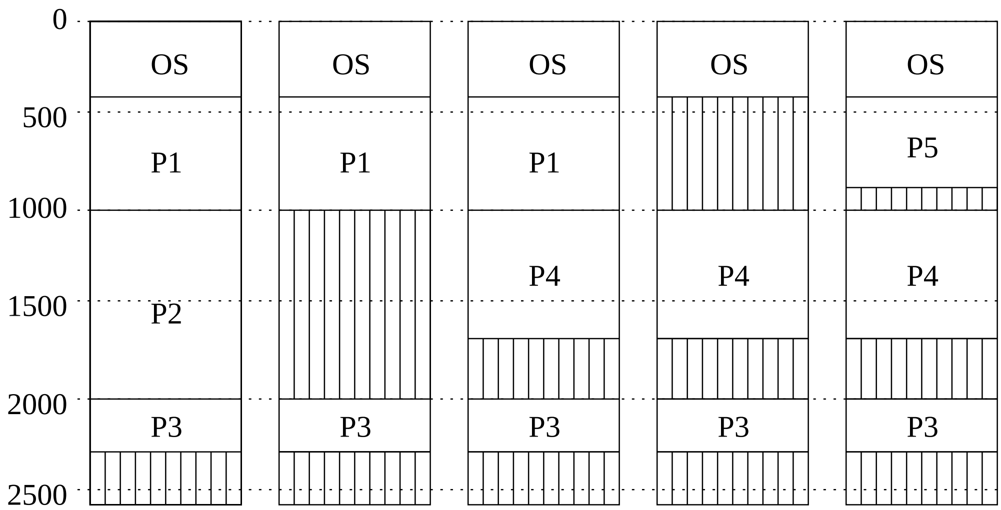


Abbildung 7.2: Beispiel zur Speicherbelegung. Prozesse: [1] 600kB 10ms [2] 1000kB 5ms [3] 300kB 20ms [4] 700kB 8ms [5] 500kB 15ms

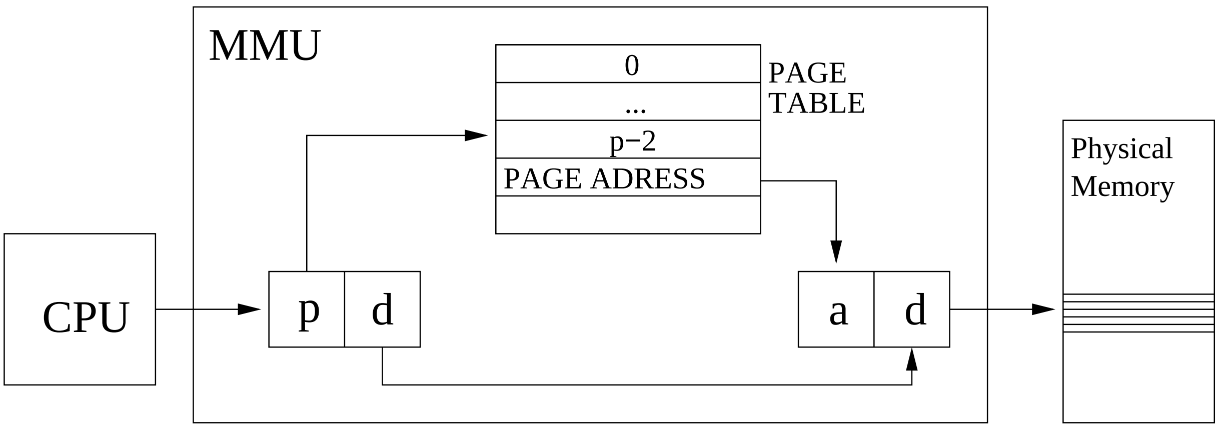


Abbildung 7.3: Funktion der Memory Management Unit (MMU) beim Paging

7.3 Paging

Unterteilung des Adreßraums in Blöcke fester Größe

- Physikalischer Speicher: Frames (Rahmen)
- Logischer Speicher: Pages (Seiten)
- Adresse: (p|d) mit p=page number, d=page offset

Beim Paging kommt es zur sogenannten internen Fragmentierung. Diese entsteht dadurch, daß innerhalb der Frames Speicher ungenutzt bleibt, weil die Programm- und Datensegmente normalerweise nicht den vollständigen Frame belegen.

7.3.1 Page Table

Früher konnte die Page Table in Spezialregistern der CPU gehalten werden, weil sie genügend klein war. Heutzutage ist sie zu groß und wird daher im Hauptspeicher gehalten. Nachteil dabei sind die stets nötigen zusätzlichen 2 Speicherzugriffe, also Verdopplung aller Speicherzugriffszeiten.

Abhilfe bietet spezielle, sehr teure Hardware, der Translation Lookaside Buffer (TLB). Er ist eine Art assoziativer Speicher, bei dem für ein vorgegebenes Datum der Vergleich mit allen Speicherzellen simultan erfolgt. Wegen der hohen Kosten des TLB wird nicht die gesamte Page Table, sondern nur ein Teil im TLB gehalten.

Unterscheidung: TLB hit/miss. Mittlere Zugriffszeit (hit ratio q, $20ns$ TLB, $100ns$ Mem) ist

$$q \cdot (20ns + 100ns) + (1-q)(20ns + 2 \cdot 100ns)$$

Bei $q = 0\%$ sind dies durchschn. $220ns$, bei $q = 80\%$ nur $140ns$, bei $q = 98\%$ lediglich $122ns$.

Kapitel 8

Virtueller Speicher

Ein Prozeß kann ausgeführt werden, ohne daß er sich vollständig im Hauptspeicher befindet. Die Teile des Prozesses, die sich nicht im Hauptspeicher befinden, liegen auf der Festplatte (hier: virtueller Speicher mittels demand paging; segmentation wäre auch möglich).

Pager: Lagert die Pages eines Prozesses vom Hauptspeicher auf die Festplatte aus und umgekehrt

Swapper: Bearbeitet ganze Prozesse

Erwartung: Lokalität der Speicherreferenzen

8.1 Swapping

Als Swapping bezeichnet man die Prozeßauslagerung vom Speicher auf die Festplatte und umgekehrt.

Multiprogramming erfordert Swapping, da nicht alle Prozesse gleichzeitig im Hauptspeicher gehalten werden können.

Swapping kostet viel Zeit; bei Seek Time von 15ms, Latency Time von 8ms und Transfer Time von 1ms/KB benötigt ein Programmcode von 1MBytes 1sec23msec für die Auslagerung (Einlagerung).

Wegen dieser hohen Zeitkosten ist es wichtig, daß Prozeßwechsel zwischen Prozessen stattfinden, die im Speicher sind.

8.2 Demand Paging

Erst bei Bedarf werden die Pages eines Prozesses von der Festplatte in den physikalischen Speicher geladen ("Pure Demand Paging").

Markierung: Welche Pages sind im Hauptspeicher und welche nicht?

Für jeden Eintrag in der Page Table wird ein valid/invalid bit gesetzt. Falls die Page nicht im Speicher ist, führt die Hardware einen Page Fault Trap aus. Schritte:

1. Page Fault Trap
   (a) MMU stellt fest, daß das invalid-Bit in der Page Table eine fehlende Seite anzeigt
   (b) MMU erzeugt einen Trap zum OS

   (c) CPU-Ressourcen und Prozeßstates werden ermittelt
   (d) Prüfung, ob die Page-Referenz legal ist und wo sich die Seite auf der HD befindet
   (e) Auffinden eines freien Frames für die neue Seite
   (f) Anfordern des Datentransfers von der Platte

2. Einlesen der Page

   (a) optimal: Prozeß wird blockiert, so daß die CPU von anderen Prozessen genutzt werden kann
   (b) Warten in der Queue der Festplatte (ggf. vorher andere Leseanforderungen?)
   (c) Warten: Seek- und Latency-Time
   (d) Eigentlicher Datentransfer
   (e) Interrupt signalisiert an CPU: I/O completed

3. Fortführung des Prozesses

   (a) OS empfängt die Meldung und aktualisiert die Page Table
   (b) Falls der Prozeß blockiert war, wird er in die Ready-Queue gebracht
   (c) Warten auf Zuteilung der CPU
   (d) Laden der CPU-Register, Prozeßstates und der aktualisierten PageTable.
   (e) Wiederaufnahme des Befehls, bei dem der Page Fault aufgetreten ist.

Komplikationen

Falls kein Frame mehr frei ist, wird zunächst eine auszulagernde Page aus dem Speicher gewählt, danach wird diese auf die Platte ausgelagert (2 Page Transfers), siehe auch 8.3.

Kosten eines Page Faults

dramatische Unterschiede: Speicher: $T_M = 100ns$, Festplatte: $T_D = 25ms$.

Die mittlere Zugriffszeit (bei Wahrscheinlichkeit eines Page Faults q) beträgt:

$$T = (1-q) \cdot T_M + q \cdot T_D \approx T_M + q \cdot T_D \text{ für } T_M \ll T_D$$

Es gilt dann für (q/T): $(0/100ns)$, $(10^{-6}/125ns)$, $(10^{-4}/2600ns)$, $(10^{-2}/250\mu s)$

8.3 Seitenersetzung

Frage: Welche der im Memory vorhandenen Pages soll auf die Platte ausgelagert werden?

Ideales Ziel: Die Zahl der Page Faults soll minimal sein

Wir betrachten die Folge der Speicheradressen-Aufrufe eines Prozesses ("Reference String").

Zur Veranschaulichung werden Änderungen der Frames bei den Strategiebeispielen auf Basis eines überall gleichen Referenzstrings angegeben. Sternchen-Reihen bedeuten, daß keine Änderung stattfand (die angefragte Seite also bereits im Speicher war).

8.3.1 Optimaler Algorithmus

Hypothetischer "Blick in die Zukunft": Ersetze die Page, die für die längste Zeit nicht gebraucht werden wird

Request	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	*	2	*	2	*	*	2	*	*	2	*	*	*	7	*	*
Frame 2		0	0	0	*	0	*	4	*	*	0	*	*	0	*	*	*	0	*	*
Frame 3			1	1	*	3	*	3	*	*	3	*	*	1	*	*	*	1	*	*

Optimalerweise fallen (im Beispiel) 6 (+3) Page Faults an.

8.3.2 FIFO-Algorithmus

Jeder Page wird ihre Ladezeit angehängt, bei Bedarf wird die älteste Page ausgelagert

Request	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	*	2	2	4	4	4	0	*	*	0	0	*	*	7	7	7
Frame 2		0	0	0	*	3	3	3	2	2	2	*	*	1	1	*	*	1	0	0
Frame 3			1	1	*	1	0	0	0	3	3	*	*	3	2	*	*	2	2	1

Es fallen bei dieser Strategie (im Beispiel) 12 (+3 beim anfänglichen Laden) Page Faults an.

Beim FIFO-Algorithmus tritt die sogenannte Belady-Anomalie auf: ein Erhöhen der Framezahl führt nicht zwangsläufig zu einer Verringerung der Page Faults, in wenigen Fällen ist dann sogar mit mehr Page Faults zu rechnen.

8.3.3 LRU-Algorithmus

LRU = Least Recently Used

Approximation an optimalen Algorithmus durch Verwenden der Zeit, zu der eine Page zuletzt benutzt wurde.

Es wird also die Page ersetzt, die am längsten nicht benutzt wurde.

Request	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
Frame 1	7	7	7	2	*	2	*	4	4	4	0	*	*	1	*	1	*	1	*	*
Frame 2		0	0	0	*	0	*	0	0	3	3	*	*	3	*	0	*	0	*	*
Frame 3			1	1	*	3	*	3	2	2	2	*	*	2	*	2	*	7	*	*

Hier fallen also (im Beispiel) 9 (+3) Page Faults an.

Bei dem LRU-Algorithmus kann die Belady-Anomalie nicht auftreten.

Approximation an LRU-Algorithmus: Referenz-Bit

Die meisten Prozessoren verfügen über eine Hardware in Form eines sogenannten Referenzbits (R-Bit) für jeden Eintrag in der Page Table. Diese Bits werden bei Initialisierung auf Null gesetzt, und auf Eins, sobald die Seite mittels Read/Write referenziert wurde.

Basis-Algorithmus:

In regelmäßigen Zeitintervallen (bspw. 100ms) wird das R-Bit jeder Page ausgelesen und zurückgesetzt. Jede Page hat ein eigenes Byte, in dem über dieses Bit (mittels Shiften) Buch geführt wird.

Beispiel:

1	1	1	1	1	1	1	1	8mal benutzt
1	1	0	0	0	0	0	0	die letzten beiden Male benutzt
0	1	1	1	0	1	1	1	beim vorletzten Mal benutzt [...]

Dieses Byte wird als `unsigned int` interpretiert, und die Seite mit dem niedrigsten Wert wird ersetzt.

8.3.4 Second Chance Algorithmus

Entspricht dem FIFO-Algorithmus mit Benutzung eines Usebits.

Bei der nach FIFO ausgewählten Page wird das Usebit geprüft. Falls es Null ist, wird die Page ersetzt. Falls es Eins ist, bekommt die Page eine "second chance"; die nächste Page in der FIFO-Reihenfolge wird betrachtet.

Alle Usebits werden auf Null gesetzt wenn entweder ein Page Fault auftrat oder alle Usebits gesetzt sind (in letzterem Falle wäre keine weitere Information aus den Usebits zu gewinnen).

Zusätzlich wird R auf Null gesetzt und die Ladezeit wird auf die aktuelle Zeit gesetzt. Implementiert wird der Algorithmus als zyklische Warteschlange; ein Spezialfall tritt auf, wenn alle Referenzbits Eins sind, dann ist der Algorithmus ein reiner FIFO-Algorithmus.

Verbesserter Second Chance Algorithmus

Zusätzlich zu dem Usebit unterstützt die Hardware ein M-Bit (M=modify). Dieses wird auf Null gesetzt bei Initialisierung und beim Auslagern auf die Platte, und auf Eins bei einem Schreibbefehl auf die Page.

Beachte: Das Usebit wird in regelmäßigen Zeitintervallen zurückgesetzt, das M-Bit nicht.

Klassifizierung:
Klasse 1: R=M=0; Klasse 2: R=0, M=1; Klasse 3: R=1, M=0; Klasse 4: R=M=1

Die Page in der niedrigsten nichtleeren Klasse (Reihenfolge 1,2,3,4) wird ersetzt.

Kapitel 9

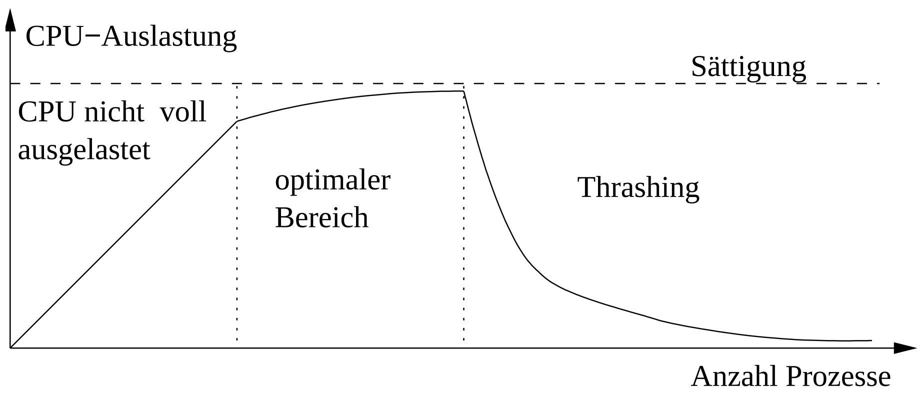
Virtueller Speicher und Multiprogramming

Die wichtigste Anwendung des VM (mit demand paging) ist Multiprogramming mit dem Ziel der effektiven Nutzung der Ressourcen Hauptspeicher und CPU.

Gegenläufige Anforderungen:

- zu wenig Prozesse ⇒ mangelnde Auslastung
- zu viele Prozesse ⇒ zu wenig Speicher pro Prozeß, daher viele Page Faults ("Thrashing")

Folgende CPU-Auslastung in Abhängigkeit von der Zahl der Prozesse ergibt sich:



Thrashing

Ein Prozess verwendet (deutlich) mehr Zeit auf das Laden der benötigten Pages als auf die Ausführung von Instruktionen.

9.1 Optimale Seitengröße

Speicher Overhead = Page Table Size + Internal Fragmentation:

$$F(p) = \frac{s \cdot e}{p} + \frac{p}{2}$$

mit s = mittlerer virtueller Memory-Bedarf pro Prozeß, e = Größe eines Eintrags in der Page-Table, p = Page Size. Optimal ist hierbei $p = \sqrt{2es}$.

Bei $e = 4B$ und $s = 512kB$ ergibt sich bspw. $p = 2kB$ als optimale Größe.

9.2 Optimale Framezahl

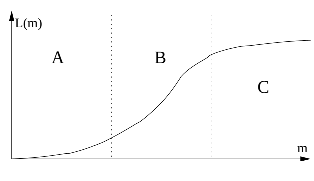
Damit möglichst viele Prozesse aktiv sein können, ohne daß es zum Thrashing kommt, ist eine möglichst günstige Zuweisung der zur Verfügung stehenden Frames an die Prozesse notwendig. Pauschal gilt hierbei:

- Wieviele Frames weist man einem Prozess zu? Die Zahl der Frames sollte der Zahl aktiver Pages entsprechen. Eine Page heißt aktiv, wenn es sehr wahrscheinlich ist, daß diese Seite referenziert wird.
- Wann soll ein neuer Prozeß gestartet werden? Falls genügend Frames für die aktiven Pages frei sind.
- Wann ist ein aktiver Prozeß stillzulegen? Falls ein Page Fault auftritt und alle Frames von aktiven Pages belegt sind.
- Wann soll sich die Speicheraufteilung ändern? Wenn bei einem Prozess die Zahl der aktiven Pages sinkt und ein anderer Prozess mehr aktive Pages benötigt.

9.2.1 Lifetime-Funktion

Die Lifetime-Funktion $L(m)$ eines Prozesses gibt die mittlere Zeit zwischen zwei aufeinanderfolgenden Page Faults bei m zur Verfügung stehenden Frames an. Sie dient hierbei dabei zur Bestimmung der Menge der aktiven Pages.

Typische Form:



Die Kurve läßt sich hierbei grob in drei Bereiche unterteilen:

- Bereich A: Zuwenige Frames, daher extrem viele Page Faults und sehr geringe Lifetime
- Bereich B: Der Engpaß wird behoben (mehr Frames führen zu deutlicher Lifetime-Steigerung)
- Bereich C: Sättigung; weitere Frames führen zu einer kaum höheren Lifetime.

Optimal ist eine Wahl der Framegröße im Übergangsbereich zwischen B und C.

9.2.2 Working Set Modell

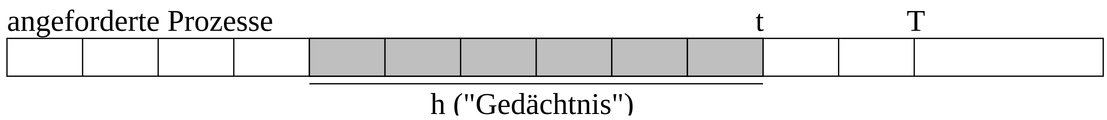
Mit dem Working-Set-Modell wird die Lifetime-Funktion approximiert, d.h. mit ihm kann man ermitteln, welches die aktiven Pages eines Prozesses sind. Hierbei betrachtet man den Referenzstring eines Prozesses:

$$r_1^T = r_1 ... r_t r_{t+1} ... r_T$$

Der Working-Set $W(t, h)$ für diesen Prozeß ist

$$W(t,h) := \bigcup_{\tau=t-h+1}^{t} r_\tau$$

Er ist also die Menge der Pages, die der Prozeß zur Zeit t in den letzten h Page-Referenzen angesprochen hat.



Bedingungen:

- Lokalität des Prozesses: Speicherzugriffe erfolgen im wesentlichen lokal, d.h. in eng benachbarten Adressbereichen.
- Die zukünftigen Speicherzugriffe stehen eng mit denen der unmittelbaren Vergangenheit in Verbindung.

Strategien:

- Fenstergröße h gut wählen (siehe 9.2.3)
- Zahl der Frames m zur Zeit t sollte $m := |W(t, h)| \leq h$ sein.
  Steuere die Zahl der aktiven Prozesse über der Zeit t; falls genügend Frames frei sind, aktiviere neuen Prozeß, falls jedoch die Summe aller Working Sets zu groß ist, lege einen Prozeß still.
- Einzelheiten der Speicheraufteilung: bei Page Faults lagere diejenigen Pages aus, die zu keinem Working-Set gehören, d.h. ein Prozeß wächst auf Kosten eines anderen Prozesses. Falls dies nicht möglich ist, ist das System überlastet und ein Prozeß (z.B. der anfordernde) sollte stillgelegt werden.

Beim Multiprogramming gibt es zwei Strategien für die Seitenersetzung:

- Lokale Strategie: Jedem Prozeß wird eine feste Zahl von Frames zugewiesen
- Globale Strategie: Die verfügbaren Frames werden unter allen Prozessen aufgeteilt, so daß die Zahl der Frames eines Prozesses variabel ist.

9.2.3 Wahl der Fenstergröße h

Der Gesamtdurchsatz D wird durch die Verfügbarkeit von zwei Ressourcen bestimmt:

- CPU (mit Hauptspeicher) mit T =mittlere Rechenzeit pro Prozeß
- Paging Station (PS) mit T_{PS} =mittlere Bedienzeit pro Page Fault

Bei einer CPU-Zeit T belegt jeder Prozeß die Paging Station für die Zeit $\frac{T}{L(m)}T_{PS}$, wobei $\frac{T}{L(m)}$ die erwartete Gesamtzahl der Page Faults für die Lifetime-Funktion ist.

Der Gesamtdurchsatz D wird (näherungsweise) durch jede der beiden Ressourcen nach oben beschränkt:

$$D \leq min\left\{\frac{1}{T}, \frac{L(m)}{T} \cdot \frac{1}{T_{PS}}\right\} = \frac{1}{T} \cdot min\left\{1, \frac{L(m)}{T_{PS}}\right\}$$

Beide Ressourcen werden gut ausgenutzt bei der Wahl

$$L(m) = T_{PS}$$

Wähle also die Fenstergröße entsprechend.

Betrachte die Warteschlangensysteme M/M/1 (memoryless) oder besser M/D/1 (deterministisch, d.h. konstante Bedienzeit) mit der Auslastung $\rho = \frac{\lambda}{\mu}$ (λ=mittlere Ankunftsrate und $\mu = \frac{1}{T_{PS}}$ =mittlere Bedienrate).

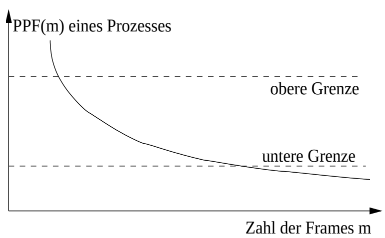
Für die mittlere Kundenzahl $E\{N\}$ gilt:

- M/M/1: $E\{N\} = \frac{\rho}{1-\rho}$
- M/D/1: $E\{N\} = \frac{\rho}{1-\rho} \cdot (1 - \frac{\rho}{2})$

Eine gute Auslastung des Gesamtsystems (CPU+Paging-Station) ergibt sich, wenn $E\{N\} = 1$, d.h. wenn die Paging-Station belegt ist und kein Prozeß warten muß ("50%-Regel"):

- M/M/1: $\rho = 50\%$
- M/D/1: $\rho = 2 - \sqrt{2} = 59\%$

Da das Working-Set-Modell eine aufwändige Hardware verlangt nutzt man oft die folgende Vereinfachung:



Die zugeteilten Frames pro Prozeß, d.h. die Zahl der aktiven Prozesse, werden mittels PPF (Page Fault Frequency) gesteuert.

- obere Grenze: Falls die PFF die obere Grenze überschreitet, erhöhe m. Falls dies nicht möglich ist (da alle Frames belegt sind), lege diesen Prozeß still.
- unter Grenze: Falls die PFF die untere Grenze unterschreitet, gib ein (geeignetes) Frame dieses Prozesses frei und starte eventuell einen zusätzlichen Prozeß.

Kapitel 10

File System

10.1 Dateikonzept

Sekundärspeicher: Magnetplatte, Magnetband, optische Platte

Wir ignorieren (zunächst) die physikalischen Eigenschaften und definieren die logische Struktur aus der Sicht des Benutzers mittels File-Konzept:

Datei:

- Sammlung von Daten
- Name: (möglichst) eindeutige Identifikation
- nichtflüchtige (=permanente) Speicherung
- existiert außerhalb des Adressraums von Prozessen und ist daher wichtig für den Austausch von Daten zwischen Prozessen

Festplattenspeicher ist wichtig für die Funktionalität des OS:

- Ausführbare Prozesse (inkl. OS-Prozesse)
- Prozeß-Scheduling, Swapping u. Paging verwenden die Festplatte
- oft (bspw UNIX): Kommando-Interpreter verwendet die ausführbaren Programme auf Festplatte

Art von Daten in Files:

- numerische Daten
- Texte (alphanumerisch)
- binäre Daten (ausführbares Programm, alle 8-Bit-Kombinationen)
- ...

File-Typen:

- Textfile
- Sourcefile (einer Programmiersprache)
- Objectfile
- Executable / Binary File (ausführbarer Code)
- ...

oft: entsprechende Extension im Dateinamen

File-Attribute:

- Name
- Typ des Files (oft: File-Extension)
- Position des Files im System
- Größe (in Bytes/Blocks)
- Protection Codes (bspw. UNIX: read/write/execute für owner/group/world)
- Zeitangaben: Erzeugung, letzte Modifikation, letzter Zugriff
- Besitzer

File-Operationen:

Das OS stellt System Calls zur Verfügung, u.a. für create, delete, open, close, read, write, seek (Positionierung des File-Pointers), truncate, ...

Die Operationen verwenden folgende Informationen:

- **File Pointer** (current file position pointer)
  zeigt auf die aktuelle Position des Files, auf der die nächste read/write-Operation erfolgen soll
- **File Open Count**
  Wird bei open/close-Operationen hoch/runtergezählt, so daß mehrere Prozesse auf dieselbe Datei zugreifen können und darüber Buch geführt wird
- **Position der Datei auf der Festplatte**
  unmittelbarer Zugriff auf Daten der Datei (ohne zusätzliches Lesen der entspr. Informationen von Festplatte). Diese Informationen werden vom OS in einer sog. open-file-table gehalten, der Index in dieser Tabelle heißt file descriptor.

10.2 Verzeichnisse

Namenskonvention unter UNIX:

- absoluter Dateiname ist `/1stdir/2nddir/.../nthdir/file.ext`
- relativer Dateiname (bspw. in zwei darüberliegenden Ebenen): `nthdir/file.ext`

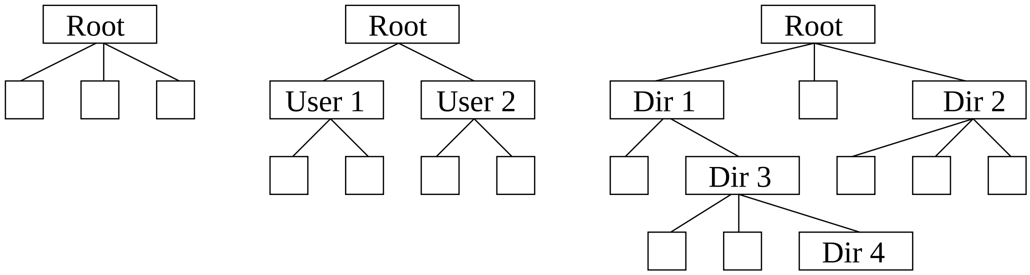


Abbildung 10.1: Verzeichnisorganisation: (1) single level (2) two level (3) Baum

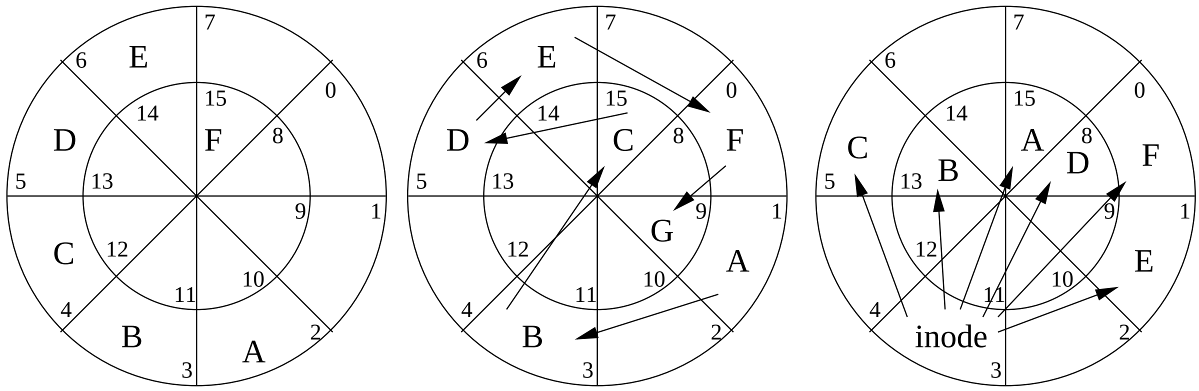


Abbildung 10.2: Belegungsstrategien: (1) zusammenhängende (2) verkettete (3) indizierte Belegung

10.3 Belegungsstrategien

Das Betriebssystem faßt mehrere Sektoren zu einem Block zusammen, oft 4KByte=4096Byte. Es kann dabei verschiedene Arten des Zugriffs unterstützen: direct access und sequential access.

Aufgabe: Die Blöcke einer Datei liegen auf der Platte u.U. beliebig und verteilt.

Strategien:

- zusammenhängende Belegung (contiguous allocation)
- verkettete Belegung (linked allocation)
- indiziert Belegung (indexed allocation)

10.3.1 Zusammenhängende Belegung

Jede Datei belegt eine entsprechende Zahl von zusammenhängenden (im Sinne der Spur) Blöcken auf der Platte (siehe Abb. 10.2).

Eigenschaften:

- Ähnlich wie beim Hauptspeicher führt diese Strategie zu externer Fragmentierung.
- bei Dateierstellung ist normal nicht bekannt, wie groß die Datei wird; daher läßt sich schwer angeben, wieviel Platz reserviert werden muß.

- Eine solche Reservierung ist ohnehin nicht unbedingt sinnvoll, da der dynamische Charakter einer Datei nicht berücksichtigt wird.
- Sequentieller und direkter Zugriff ist möglich

10.3.2 Verkettete Belegung

Jede Datei ist eine verkettete Liste von Blöcken. Das Verzeichnis enthält einen Pointer zum ersten Block (u.U. letzter Block, Dateigröße). Siehe hierzu auch Abbildung 10.2.

Eigenschaften:

- Die Zeiger verbrauchen (wenn auch vergleichweise wenig) Speicherplatz
- Die Beschädigung eines Zeigers führt schnell zu großen Schäden
- Diese Strategie unterstützt nur den direkten Zugriff
- Eine Verfeinerung stellt die Dateibelegungstabelle (File Allocation Table, FAT) dar, die die Verkettungsstruktur der Blöcke in einer Tabelle zusammenfaßt. Dadurch wird der direkte Zugriff leichter, allerdings sind mehr Kopfbewegungen (zwischen FAT und Datenblöcken) nötig, und ein Verlust der FAT führt zu vollständigem Datenverlust.

10.3.3 Indizierte Belegung

Jede Datei verfügt hierbei über einen sogenannten Indexblock, der alle Block-Adressen enthält (siehe Abb. 59). Sofern mehr Block-Adressen zu speichern sind, als der Indexblock aufnehmen kann, werden mehrere Indexblöcke verkettet.

- Sowohl direkter als auch sequentieller Zugriff sind leicht realisierbar
- Bei kleinen Dateien kommt es zur Platzverschwendung
- Es ist schwer, eine ideale Größe für Indexblöcke anzugeben

UNIX Inodes

Das Dateisystem von UNIX basiert auf Inodes: jeder Datei ist eine kleine Tabelle auf der Platte zugeordnet. Neben Dateiattributen (wie Name, Zugriffsrechte etc.) enthält sie 12 direct pointers sowie je einen single indirect, double indirect und triple indirect pointer, die zusammen auf alle Blöcke der Datei verweisen.

Die direct pointer zeigen dabei direkt auf Datenblöcke, während single indirect pointer auf einen Block zeigen, dessen Inhalt direct pointers sind. Entsprechend zeigen double indirect pointer auf einen Block mit single, und triple indirect pointer auf einen Block mit double indirect pointers. Damit läßt sich die maximale Gesamtgröße einer Datei berechnen (Blockgröße 4 KBytes, Pointergröße 4 Bytes):

	Inhalt	Zeigeranzahl	Referenzierte Daten
direct pointer	Daten	12	48 KBytes
indirect pointer	1024 direct pointer	1.024	4 MBytes
double indirect pointer	1024 indirect pointer	1.048.576	4 GBytes
triple indirect pointer	1024 double indirect pointer	1.073.741.824	4.096 GBytes
max. Gesamtgröße		1.074.791.436	>4.100 GBytes

Vorteilhaft ist die Flexibilität (Indirektion wird nur bei großen Dateien benötigt).

Index