Professor Dr.-Ing. Stefan Kowalewski

Systemprogrammierung
WS 2004/2005
Zusammenfassung

Ulrich Loup
Ulrich.Loup@rwth-aachen.de

Vorbemerkungen:

- Diese Zusammenfassung ist privat erstellt worden und erhebt deshalb keinen Anspruch auf Vollständigkeit oder Richtigkeit!
- Das Dokument enthält im Wesentlichen die Punkte der Zusammenfassung, die Prof. Kowalewski in der letzten Vorlesung gegebenen hat.
- Das Layout des Dokuments ist so gewählt, dass es beidseitig auf DinA4 Papier gedruckt und gelocht werden kann.

Inhaltsverzeichnis

1 Aufbau des Systems

Ein Rechner besteht heutzutage *hardwaremäßig* aus verschiedenen Ein- und Ausgabegeräten sowie verschiedenen Speichern. Jedes Gerät besitzt eine **Kontrolleinheit** (**Device-Controller**), der den (evtl. gleichzeitigen) Zugriff auf das Gerät steuert. Alle Kontrolleinheiten besitzen *Puffer* und sind mit der **CPU** über den **Systembus** verbunden. Eine Eingabe/Ausgabe-Operation, kurz **E/A-Operation**, passiert bei *asynchron* arbeitenden Systemen wie folgt:

1. Unterbrechen des Benutzerprozesses, Aufruf der E/A-Routine.
2. Datenweitergabe an den Controller, aktivieren der controllerseitigen E/A-Operation.
3. Arbeit des Controllers (Datenübertragung zum Gerät, CPU kann andere Aufgaben übernehmen $\rightsquigarrow$ asynchron.
4. Controller sendet **Interrupt**-Signals bei beendeter Ausführung, CPU arbeitet entsprechende Routine ab.
5. Benutzerprozess wird daraufhin fortgesetzt.

Diese Aufzählung ist ein typischer Ablauf bei einer Ausgabe, eine Eingabe funktioniert allerdings analog, nur wird zuerst das entsprechende Interrupt-Signal ausgelöst. Falls die Operation *synchron* abläuft, wartet die CPU während der Controller arbeitet. Dies ermöglicht eine sehr viel einfachere Systemverwaltung, ist aber weitgehend ineffizient. Eine weitere Geschwindigkeitssteigerung kann durch **DMA** (Direct Memory Access) erreicht werden, bei welchem die Controller direkt in den Speicher schreiben können, was der CPU wieder Arbeit abnimmt. Eine häufig eingesetzte Methode zur Strukturierung von komplexen Systemen sind **Schichtenmodelle**. Eine *Schicht* stellt jeweils Dienste für eine darüberliegende Schicht zur Verfügung und nutzt die Funktionalität der darunterliegenden Schicht über *wohldefinierte Schnittstellen*. Der Vorteil liegt in der extrem hohen Wartungsfreundlichkeit durch einfaches Austauschen von Schichten. Ein Schichtenmodell für Kommunikationsprotokolle stellt das **ISO/OSI-Schichtenmodell** (International Standard Organization/Open Systems Interconnection) dar. Hier liegen wie oft die hardwarenahen Schichten unten und die anwendungsnahen Schichten oben. Diese *sieben Schichten* sind die folgenden:

7 Anwendung
6 Darstellung
5 Sitzung
4 Transport
3 Netz
2 Sicherung
1 Bittransport

Hierbei sind die letzten drei Schichten die netzbezogenen und die darüberliegenden die anwendungsbezogenen Schichten.

2 Prozessverwaltung

2.1 Prozesszustände

Ein **Prozess** ist ein Programm im Stadium der Ausführung. Es gibt entweder *Benutzerprozesse* oder *Systemprozesse*, welche beide unterschiedliche Zustände annehmen können $\rightsquigarrow$ Tabelle 1. Es kann immer nur ein Prozess gleichzeitig auf einem Prozessor ausgeführt werden.

Tabelle 1: Prozesszustände

running	Prozessanweisungen werden gerade ausgeführt
ready	Prozess ist bereit, wartet aber auf freien Prozessor
waiting	Prozess wartet auf Ereigniss
blocked	Prozess wartet auf Freigabe eines Betriebsmittels
new	Prozess wird gerade erzeugt
killed	Prozess wird zwangsweise abgebrochen
terminated	Prozess hat Ausführung beendet

2.2 Zustandsübergänge

Die Zusammenhänge zwischen den einzelnen Prozesszuständen bzw. die Zustandsübergänge können durch ein **Prozesszustandsdiagramm** dargestellt werden, welches ein kantenbeschrifteter, gerichteter Graph mit den Prozesszuständen als Knoten ist. Eine Kante $e = (A, B)$ zwischen zwei Zuständen A und B besagt dann, dass von A der Zustand B erreicht werden kann, wenn die Aktion oder Bedingung eintritt, mit der e beschriftet ist. In Abbildung 1 ist ein Beispiel-Zustandsdiagramm dargestellt.

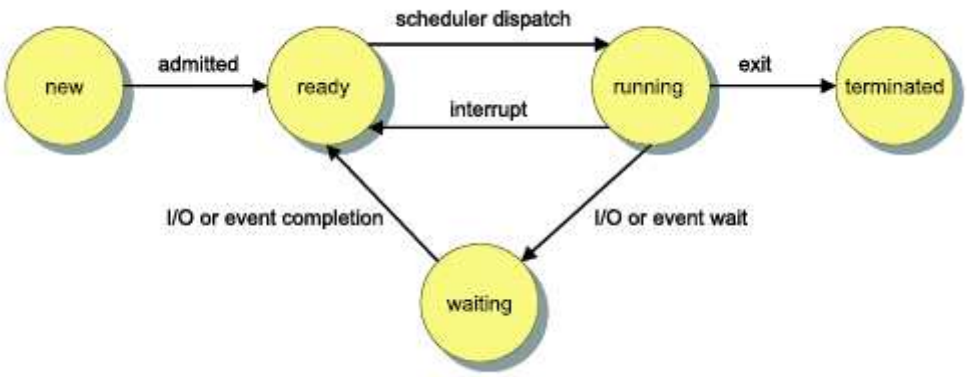

Abbildung 1: Beispiel für Prozesszustandsdiagramm

2.3 Prozesskontrollblock

Die Betriebssystem-relevante Information eines Prozesses ist in einem **Prozesskontrollblock** (PCB) abgelegt, welcher etwa die folgenden Informationen enthält:

2.4 Message Passing und Shared Memory

Es gibt zwei unterschiedliche Ansätze für die Kommunikation zwischen Prozessen:

- **Message Passing**: Aufbau einer direkten Verbindung zwischen zwei Prozessen vermittelt über den Betriebssystemkern; Kommunikation über Befehle wie *open-connection*, *close-connection*, *send* oder *recieve*.

Status	aktueller Prozesszustand
Programmzähler	aktuelle Adresse der nächsten Instruktion
CPU-Scheduling	Parameter für Prozess-Scheduling (Warteschlangenzeiger etc.)
Speichermanagement	Informationen zur Speicherverwaltung (Seitentabellen etc.)
E/A-Status	Liste von benötigten E/A-Geräten
Accounting	benötigte CPU-Zeit, Prozessnummern etc.

- **Shared Memory**: Prozesse haben über den Betriebssystemkern Zugriff auf einen gemeinsamen Speicherbereich, in welchem Nachrichten hinterlassen und gelesen werden können ($\rightsquigarrow$ Synchronisationsprobleme).

2.5 Threads

Ein **Thread** ist ein „Prozess in einem Prozess“, wobei er nur die von dem eigentlichen Prozess zur Verfügung gestellten Betriebsmittel und Speicherbereiche benutzen darf. Ein Prozess, in dem mehrere Threads zugelassen sind, heißt **Multi-Threading-Prozess**, das Gegenteil **Single-Threading-Prozess**. Der Vorteil von Multi-Threading-Prozessen ist der geringe Zeitverlust bei der Inter-Thread-Kommunikation, was mehr Geschwindigkeit bringt. Außerdem kann das Scheduling auf Anwendungsebene laufen und es müssen keine Sicherheitsvorrichtungen implementiert werden, was wieder Geschwindigkeitsvorteile liefert.

3 Prozesssynchronisation

In einem System arbeiten oft viele Prozesse miteinander „verzahnt“ (*interleaved*), d.h., sie bedienen sich gegenseitig ihrer Ergebnisse und teilen Betriebsmittel und Speicher. Falls zwei Prozesse eine Variable gleichzeitig verändern können, so bezeichnet man das als **Inkonsistenz**. Es gibt einige klassische Problemstellungen für den Umgang mit Konflikten, die bei Prozessen auftreten können.

3.1 Erzeuger-Verbraucher-Problem

Problemstellung: Es gibt Erzeuger- und Verbraucher-Prozesse, welche beide auf ein gemeinsames endliches Lager S zugreifen können. Die Erzeuger schreiben (produzieren) und die Verbraucher lesen (verbrauchen) und es gelten die folgenden notwendigen Bedinungen für eine korrekte Lösung:

1. Wenn das Lager voll ist, kann der Erzeuger nichts ablegen.
2. Wenn das Lager leer ist, kann der Verbraucher nichts entnehmen.
3. Der Lagerbestand kann nicht gleichzeitig von Erzeuger und Verbraucher verändert werden.

Probleme:

- *bei* 1 *Erzeuger und* 1 *Verbraucher:* die Variable für den Lagerfüllstand S verursacht Inkonsistenzen, da bei einem gleichzeitigen Lagerzugriff beider Prozesse es passieren kann, dass S von beiden Prozessen eingelesen wird, aber nicht in der richtigen Reihenfolge wieder zurückgeschrieben wird. Dann kann z.B. der Lagerstand sich nicht verändert haben, da eine Einheit verbraucht und eine erzeugt wurde, aber nur ein Prozess hat den Lagerstand verändert (*Lost Update*).

- *Lösung mit Ringpuffer:* Jeder der zwei Prozesse hat eine eigene Variable IN bzw. OUT. Der Zugriff auf den Ringpuffer läuft über die Modulo-Operation.
- *mehrere Erzeuger*: Lösung mit Ringpuffer nicht möglich, da dann wieder mehr als ein Prozess auf ein und dieselbe Variable zugreifen können, was zu dem gleichen Fehler führt.

3.2 Wechselseitiger Ausschluß

Der Programmcode von Prozessen läßt sich in **kritische** und **unkritische Bereiche** einteilen. Ein kritischer Bereich (KB) ist eine Phase des Programms, die auf gemeinsame Daten zugreift. Beim Erzeuger-Verbraucher-Problem dürfen sich nicht mehrere Prozesse im kritischen Bereich (Lagerzugriff) aufhalten sondern immer nur der Erzeuger oder der Verbraucher, was auch **wechselseitiger Ausschluß** (als isolierte Problemstellung) genannt wird. Es stehen die Mechanismen *Sperre* und *Entriegelung* für einen Prozess zur Verfügung, der in den kritischen Bereich will. Es muss weiterhin gelten, dass diese Mechanismen jeweils *atomare Operationen*, d.h., unteilbar sind. Die folgenden Bedingungen müssen für eine korrekte Lösung des wechselseitigen Ausschlußproblems gelten:

1. **Mutual-Exclusion** (höchstens ein Prozess im KB)
2. **Progress** (jeder darf in den KB)
3. **Bounded-Waiting** (endliche Wartezeit vor Eintritt in KB nach Erlaubnis)

Ist *Progress* verletzt so auch *Bounded-Waiting*. Außerdem darf kein Prozess in seinem KB sterben, da sonst sofort *Bounded-Waiting* verletzt wäre.

3.2.1 Lösung 1: Bakery-Algorithmus

Der Algorithmus arbeitet wie folgt:

1. Prozess P_i signalisiert, dass er Nummer ziehen will (`choosing[i]:=TRUE`).
2. P_i zieht größte Nummer (kann auch bei mehreren Prozessen die selbe sein).
3. P_i setzt (`choosing[i]:=FALSE`).
4. Warten, dass alle Prozesse vor ihm die Nummer gezogen haben (`while choosing[j] do noop`).
5. Warten, bis P_i an der Reihe
   (a) via Nummer (`while number[j] <> 0`).
   (b) via Alphabet bzw. Prozessnummer (`and (number[j], j) < (number[i], i)`).
   (c) P_i tritt in kritischen Bereich ein und setzt danach die Nummer zurück (`number[i] := 0`).

3.2.2 Lösung 2: Bakery-Algorithmus mit enqueue/dequeue

Diese Variante des Bakery-Algorithmus ist durch eine **FIFO-Warteschlange** realisiert. Das Einfügen an den Anfang *enqueue* und das Entnehmen vom Ende der Warteschlange *dequeue* sind jeweils atomare Operationen. Der Algorithmus läuft dann für einen Prozess P_i wie folgt:

1. P_i setzt sich in die Warteschlange (`enqueue(i)`)
2. P_i wartet, bis er an der Reihe ist (`while F[0] <> i do noop`)
3. P_i geht in kritischen Bereich, entfernt sich danach wieder von der Warteschlange (`dequeue(i)`)

3.3 Semaphore

Semaphore sind ein höhersprachliches Konzept, Probleme wie das wechselseitige Ausschluß-Problem auf kompfortabele Weise zu lösen. Ein Semaphor ist eine (Integer-)Variable verknüpft mit drei Methoden, welche *atomar* realisiert sind. Es gibt auch eine Semaphor Variante, welche zusätzlich eine Warteschlange assoziiert hat, die wartende Prozesse aufnehmen und vor dem *busy waiting* bewahren soll. Die Drei Operationen des Semaphors `S` einmal ohne einmal mit assoziierter Warteschlange sind die folgenden:

1. `init(S, start)`

   Anfangswert setzen, `start`= 1 bei **binärem Semaphor**, `start`= ∞ bei **Zählsemaphor**.

2. `wait(S)`

   (a) warten bis Zugang erlaubt (etwa `S > 0`), dann selbst sperren (etwa `S := S-1`)

   (b) falls Zugang erlaubt, in KB eintreten, sonst in Warteschlange eingliedern

3. `signal(S)`

   (a) Sperre aufheben (etwa `S := S+1`)

   (b) bekanntmachen, dass KB verlassen wurde (etwa `S := S+1`) und falls KB weiterhin betreten werden kann, nächsten Prozess aus Warteschlange wecken

Eine Anwendung sieht in der Regel für einen Prozess P_i und den Semaphor `S` wie folgt aus:

```
wait(S);
"Kritischer Bereich";
signal(S);
Ünkritsicher Bereich";
```

3.3.1 Semaphor-Lösung des Erzeuger-Verbraucher-Problems

Semaphore eigenen sich besonders zur Lösung des Erzeuger-Verbraucher-Problems mit $N > 1$ Erzeugern und $M > 1$ Verbrauchern. Dazu braucht man zunächst einen binären Semaphor S, welcher den Zugriff auf das Lager darstellt (es darf ja nur ein Prozess gleichzeitig zugreifen). Den Lagerbestand kann man dann in zwei Zählsemaphore E und V aufteilen, wobei $E + V = MAX$ während der gesamten Laufzeit invariant sein soll (MAX = maximaler Lagerbestand). Das bedeutet für die Initialisierung:

```
init(S,1); init(E,MAX); init(V,0);
```

Programmablauf für Erzeuger und Verbraucher:

- Der Erzeuger kann nur in das Lager einliefern, falls $E > 0$ (`wait(E)`). Ist dies der Fall, betritt er mit `wait(S)` das Lager, verlässt es nach Ablieferung und gibt es wieder frei mit `signal(s)`. Dann teilt er dem Verbraucher über `signal(V)` mit, dass ein neues Element im Lager liegt.
- Der Verbraucher hingegen prüft analog den Semaphor V vor Eintritt des Lagers, da er ja nur Elemente entnehmen kann, falls $V > 0$, der Erzeuger also etwas ins Lager geliefert hat. Nach dem Lagerzugriff (über S gesteuert), gibt der Verbraucher dem Erzeuger ein Signal, dass das Lager jetzt einen Platz mehr frei hat mittels `signal(E)`.

3.3.2 Semaphor-Lösung des Reader-Writer-Problems

Das *Reader-Writer-Problem* besteht aus dem Konflikt zwischen *Writern*, die in einer gemeinsamen Datei Updates machen wollen, und *Readern*, die (gleichzeitig) die Datei Lesen wollen. Es gibt drei unterschiedliche Problemstellungen für diesen Sachverhalt, welche unterschiedliche Prioritätsregelungen haben:

> **Erstes R/W-Problem**: Die Reader dürfen immer lesen und die Writer unterbrechen.
>
> **Zweites R/W-Problem**: Die Writer dürfen immer schreiben und die neuankommenden Reader verbieten.
>
> **Drittes R/W-Problem**: Es wird zwischen Lese- und Schreibphasen abgewechselt, in welchen jeweils Reader bzw. Writer die exklusiven Rechte haben (fairer Ansatz).

Das *erste* R/W-Problem kann dann mittels zweier Semaphore und einer Zählvariablen gelöst werden:

- Semaphor W mit `init(W,1)` für den Exklusiv-Zugriff auf die Datei
- Semaphor Z mit `init(Z,1)` für den Exklusiv-Zugriff auf die Zählvariable `n`
- Zählvariable `n := 0` für die Anz. gleichzeitig aktiver Leser

Ein Writer braucht hierbei lediglich zu warten, ob der Zugriff erlaubt wird (`wait(W)`), dann schreibt er und gibt die Zugriffsrechte wieder frei (`signal(W)`). Der Reader hingegen muss sich zuerst mit

```
wait(Z);
n := n+1;
```

als Reader anmelden. Falls er der erste Reader ist, reiht er sich über die Bedingung

```
if(n = 1) then wait(W)
```

in die zu Z assoziierte Warteschlange ein, so dass als nächstes alle n Reader Zugriff haben. Nach dem Lesevorgang wird mit

```
wait(Z);
n := n-1;
if(n = 0) then signal(W);
signal(Z);
```

die Readerzahl aktualisiert und evtl. die Sperre für die Datei aufgehoben, falls kein Reader mehr wartet.

3.3.3 Semaphor-Lösung des Fünf-Philosophen-Problems

Gegeben sind fünf Prozesse $P_0, \ldots, P_4$ (Philosophen), die sich um die fünf Betriebsmittel $s_0, \ldots, s_4$ (Stäbchen) bewerben, wobei immer zwei BM $s_i, s_{(i+1) \bmod 5}$ von P_i (für linke und rechte Hand) benötigt werden. Schematisch sei also der unkritische Bereich das Denken der Philosophen und der kritische der Eßvorgang. Bei der naiven Lösung, welche vorsieht, dass ein Philosoph einfach hintereinander ein Stäbchen (dargestellt durch jeweils einen Semaphor) anfordert, führt zu einem Deadlock. Dann lässt sich z.B. eine Situation konstruieren, bei der jeder Philosoph ein Stäbchen besitzt und immer noch auf eins wartet (Circular-Wait verletzt), da z.B. alle Philosophen Rechtshänder sind. Diese

Verklemmung kann man beheben durch Definition eines Linkshänders, wodurch einfach die notwendige Bedingung Circular-Wait vereitelt wird (siehe Abschnitt 4). Eine weitaus kompliziertere Lösung benutzt den Zwischenzustand *hungrig*, in dem ein Philosoph sich für seine Stäbchen anmeldet. Realisiert werden kann dieser Ansatz mittels eines Semaphors S (essen) und Semaphore $h[0], \dots h[4]$ (hungrig) und den Kontrollvariablen $c[0], \dots c[4]$ mit $c[i] \in \{0 = denken, 1 = hungrig, 2 = essen\}$.

Tipps zu Semaphor-Aufgaben

- Bei komplizierteren Vorgängen benötigt man oft zusätzlich zu den Semaphoren *Kontrollvariablen*, welche z.B. Zählvariablen oder Zustandsvariablen sind.

- In einem Automaten werden die einzelnen Threads (z.B. `Geld()`, `Drucken()` beim Fahrkartenautomat oder `auto()` bei der Kreuzung) oft durch Funktionen in einer zusammenhängenden Klasse deklariert.

- Aufgaben, bei denen mehrere Prozesse (≥ 2) beteiligt sind, liegt es nahe, einen Ticket-Semaphor *mutex* zu benutzen. ($\rightsquigarrow$ Modell der *Bedingten Kritischen Region*)

3.4 Petrinetze

Ein **Petrinetz** ist ein Transitions-System (S, T, E), welches ein anschauliches und mathematisches Mittel zur Darstellung von Sachverhalten in einem System ist (allgemeiner: Netz/Graph). Es gilt

S: **Stellen** (durch Kreise dargestellt), können bestimmte Anzahl von **Token** (schwarze Punkte in den Kreisen) aufnehmen.

- Es heißt $\bullet v = \{w \mid (w, v) \in E\}$ der *Vorbereich*, $v\bullet = \{w \mid (v, w) \in E\}$ der *Nachbereich* einer Stelle/Transition v.

  T: **Transitionen**: Übergang zwischen Stellen (durch Rechtecke dargestellt). Falls Eingangsbedingungen erfüllt (Stellen im Vorbereich der Transition enthalten genug Token, Stellen im Nachbereich haben genug Platz), **feuert** Transition, d.h., die durch das Eingangskantengewicht vorgegebene Anzahl von Token wird vom Vorbereich entfernt, und die Stellen im Nachbereich werden mit der durch die Ausgangskantengewichte bestimmten Anzahl von Token gefüllt.

  E: **Kanten/Pfeile**: gerichtete, gewichtete Kanten zwischen Stellen und Transitionen und umgekehrt, Gewichtung ist Funktion $W : E \to \mathbb{N}$

- Eine *Markierung* $M : S \to \mathbb{N}$ ist die Anzahl der Token auf allen Stellen, darstellbar durch die Menge $\{(s, n) \mid s \in S, n \in \mathbb{N}\}$, M_0 heißt *Anfangsmarkierung.*

- Zu einer Markierung M ist $T_{akt} \subseteq T$ die Menge der Transitionen, die aktiviert ist, d.h., die Transitionen die feuern können.

Diese Formalisierung kann noch weiter mit mathematischen Begriffen ausgebaut werden, um größere Aussagekraft zu erhalten, was hier nicht benötigt wird.

3.5 Bedingte Kritische Regionen

Eine **Kritische Region** ist ein weiteres hochsprachiges Konzept für die Prozess-Synchronisation. Da ein geschützter Bereich aber nur auf Grund einer Bedingung betreten werden darf, wird das Konstrukt der Kritischen Region mit einer *Boolschen Bedingung* verknüpft, was zum Begriff der **Bedingten Kritischen Region** führt. Man geht davon aus, dass es eine gemeinsame Variable `v` gibt und eine Bedingung B, bei deren Eintritt der Prozess *ungestört* einen Programmteil S ausführt:

`region v when` B `do` S;

Der Prozess wartet also, bis B eintritt, dann kann er den Block S ausführen, in dem er `v` sicher verändern kann. Eine solche Bedingte Kritische Region kann mittels Semaphore implementiert werden. Dazu benötigt man drei Semaphore mit assoziierter Warteschlange:

1. **mutex**: Ein Ticket (binär), dass unter den Prozessen, die in den KB wollen, kursiert.
2. **firstdelay**: Warteschlange, welche diejenigen Prozesse beinhaltet, deren Bedingung nicht beim ersten Mal erfüllt war.
3. **seconddelay**: Warteschlange mit den Prozessen, die beim zweiten Mal auch keine gültige Bedingung hatten.

Es ist immer nur der Prozess aktiv, der gerade das Ticket hat. Falls seine Bedingung erfüllt ist, darf er in den KB, ansonsten wird er in die zu *firstdelay* assoziierte Warteschlange geschickt. Jeder Prozess, der den KB verlässt, weckt den ältesten wartenden Prozess aus der firstdelay-Warteschlange, der sich und alle anderen Prozesse von firstdelay nach seconddelay schickt. Dort werden die Bedingungen der Reihe nach wieder getestet, wobei die Prozesse mit Mißerfolg nach firstdelay zurück wandern, die anderen treten in den KB ein.

3.6 Monitore

Hier handelt es sich um ein weiteres hochsprachliches Konzept, Prozesse zu synchronisieren. Ein **Monitor** ist ein Objekt, welches eine Menge von Prozeduren und Daten bzw. Variablen enthält, welche immer nur von einem Prozess gleichzeitig benutzt werden dürfen. Die Anwendung eines Monitors ist denkbar einfach, da viele Operationen in einer Prozedur zusammengefasst werden können, welche mit z.B. Semaphoren sehr viel komplexere Programmcodes produzieren. Da eine bedingte Sperrvorrichtung wie bei den bedingten kritischen Regionen benötigt wird, werden Monitore mittels *Bedingungsvariablen* vom Typ `condition` implementiert. Diese sind ähnlich zu den Semaphoren, nur ruft eine `signal`-Anweisung keinen direkten Effekt hervor sondern weckt lediglich einen gerade wartenden Prozess. Aus diesem Grund muss mittels zusätzlicher Variablen vor jedem `wait`-Befehl eine Abfrage stattfinden, ob bereits ein Signal gegeben wurde, d.h., die Eigenschaften der `condition`-Variable sind zusätzlich über andere Variablen ausgedrückt (`condition`-Variable allein für Sperrung und Entsperrung zuständig).

3.6.1 Monitor-Lösung des dritten Reader-Writer-Problems

Es soll nach jedem Lesevorgang wieder ein Schreibvorgang möglich sein, bzw. nach jedem Schreibvorgang wieder ein Leser Zugriff erhalten dürfen. Hierzu wird ein Monitor `RW` implementiert, der eine Zählvariable `readercount`, zwei Bedingungsvariablen `okread, okwrite` und eine Boolsche Variable `busywrite` enthält. Außerdem gibt es vier Prozeduren:

- `startread` : Reader darf lesen, falls der gegenwärtige Writer zu Ende geschrieben hat oder keiner schreiben möchte (`if (busywrite or okwrite.nonempty) then wait(okread);`)und es wird `readercount` um `1` erhöht.
- `endread` : `readercount` wird wieder erniedrigt, falls keine Reader mehr warten, wird die Schreibfreigabe gegeben (`if readercount = 0 then signal(okwrite);`).
- `startwrite` : Writer darf analog zu den Readern nur schreiben, wenn kein Reader lesen möchte bzw. kein Writer schon schreibt:

  ```
  if (busywrite or (readercount > 0)) then wait(okwrite);
  ```

  Ist dies erfüllt, wird die Variable `busywrite := TRUE` gesetzt.
- `endwrite` : Nach Setzen `busywrite := FALSE` wird entweder die Schreib- oder die Lesephase gestartet, je nach dem, was gerade gebraucht wird (vornehmlich aber die Lesephase):

  ```
  if okread.nonempty then signal(okread)
  else signal(okwrite);
  ```

Dieses Verfahren ermöglicht eine faire Abwechslung zwischen Readern und Writern.

4 Deadlocks

Ein **Deadlock** oder eine **Verklemmung** ist eine Situation, bei der Prozesse auf andere Prozesse warten, dass *Betriebsmittel* (BM) freigegeben werden und keiner den Zustand aufheben kann. Ein Systemzustand $(P_1, \ldots, P_n)$ heißt **sicher** *gdw.* es für jeden Prozess P_i eine Permutation $\sigma \in S_n$ gibt, sodass alle P_j für $j < i$ ihre BM freigegeben haben und P_i mit maximalen Anforderungen von BM weitergeführt werden kann. Der Startzustand ist daher sicher.

4.1 graphische Darstellung

Eine anschauliche Darstellungsart für Situationen von BM-Anforderung (**Request**) und Belegung eines BM durch einen Prozess (**Allocation**) ist ein **Request-Allocation-Graph** (V, W, E_V, E_W) mit

V: runde Knoten, die den Prozess darstellen,

W: eckige Knoten, die das Betriebsmittel darstellen,

E_V: Pfeile $e = (a, b) \in V \times W$ (*Request*),

E_W: Pfeile $e = (a, b) \in W \times V$ (*Allocation*).

Die Vereinfachung dieses Graphen-Modells im Hinblick auf die Analyse von Deadlocks liefert einen sogenannten **Wait-For-Graphen**, bei dem einfach die BM-Knoten $w \in W$ weggelassen werden und nur Kanten eingezeichnet werden, welche die Form

$$(v_1, v_2) \in V \times V$$

haben, wobei $(v_1, w) \in E_V$ und $(w, v_2) \in E_W$ ist.

4.2 Deadlock Bedingungen

Es gibt die folgenden vier Bedingungen, welche *alle* notwendig für einen Deadlock sind:

1. **Circular-Wait:** Request-Allocation-Graph bzw. Wait-For-Graph hat Kreis (zyklisch)
2. **Exclusive-Use:** kein BM-Sharing (gleichzeitiges Benutzen von BM)
3. **Hold-and-Wait:** BM-Anforderung erlaubt ohne vorherige BM-Freigabe
4. **No-Preemption:** gezwungener BM-Entzug nicht möglich

Folglich kann kein Deadlock auftreten, wenn nur eine der Bedingungen nicht erfüllt ist. Es gibt nun vier Strategien mit Deadlocks umzugehen:

1. **Prevention:** Vermeidung eines Deadlocks durch Verbieten einer Deadlock-Bedingung
2. **Avoidance:** Umgehen eines Deadlocks durch Zusatzabfragen
3. **Detection:** Aufspüren von Deadlocks und nachträgliche Beseitigung
4. **Ignoring:** Annahme, dass Deadlocks nicht auftreten $\rightsquigarrow$ Vogel-Strauß-Ansatz (von den meisten Systemen z.B. Windows NT, Unix benutzt)

4.3 Prevention

Die folgenden Ansätze verhindern einen Deadlock durch Ausschließen einer der vier Bedingungen:

1. *Exclusive-Use:* unerwünscht, da Sharing wichtiger Systembestandteil
2. *zu Preemption:* Zulassen von gewaltsamem BM-Entzug
3. *zu Hold-and-Wait:*
   - BM-Freigabe vor Neuanforderung
   - alle BM gleichzeitig anfordern $\rightsquigarrow$ ineffizient
4. *zu Circular-Wait:* Klassenhierarchie von BM (d.h. alle BM niedrigerer Klassen freigeben, bevor ein BM höherer Priorität angefordert wird)

4.4 Avoidance

Die Deadlock-Vermeidung kann mittels zweier Ansätze realisiert werden. Einmal kann vor jeder BM-Zuteilung geprüft werden, ob

- die Gefahr eines Deadlocks besteht,
- der Nachfolgezustand sicher ist.

Das geschieht i.d.R. durch *worst-case* Annahmen der BM-Anforderungen aller Prozesse $\rightsquigarrow$ **MRU** (Maximum Ressource Usage). Zur Entscheidung, ob ein Zustand sicher ist, seien die MRU Werte und die Anzahl der freien BM der Prozesse gegeben. Dann muss man die Prozesse so anordnen, dass ein Prozess immer sein Maximum an BM von den freigegebenen BM des Vorgängerprozesses (= MRU des Vorgängers) tilgen kann (Banker's Algorithmus).

4.5 Banker's Algorithmus

Der **Banker's Algorithmus** ist ein Algorithmus, um festzustellen, ob ein *Zustand sicher* ist und findet daher Anwendung in der *Deadlock-Avoidance* (Prüfung, ob Folgezustand sicher) und der *Deadlock-Detection* (Prüfung, ob ein Deadlock vorliegt und welche Prozesse beteiligt sind). Die Prozesse seien bezeichnet mit $P_i,\ i \in [1,n]$ und die Betriebsmitteltypen mit $B_j,\ j \in [1,m]$, das System sei im Zeitpunkt t:

$$\begin{aligned} Q^{max}_{j,s}(t) &:= \text{maximal zusätzlich von } P_j \text{ anforderbare BM des Typs } B_s \\ Q_{j,s}(t) &:= \text{aktuell von } P_j \text{ angeforderte BM des Typs } B_s \\ H_{j,s}(t) &:= \text{aktuell von } P_j \text{ gehaltene BM des Typs } B_s \\ V_s(t) &:= \text{aktuell verfügbare BM des Typs } B_s \end{aligned}$$

Um zu prüfen, ob die Anforderungen der gegebenen Prozesse zum Zeitpunkt t realisierbar sind für gegebene Tabellen $Q(t)$ und $H(t)$ (d.h. Prozesse gegen BM aufgetragen), muss einfach die Summe der Tabellen $Q(t)+H(t)$ gebildet werden, wobei $(Q(t)+H(t))_{j,s} = (Q(t))_{j,s} + (H(t))_{j,s}$. Sei $M \in \mathbb{N}^m$ der BM-Vorrat, so ist die BM-Anforderung insgesamt realisierbar, falls für jeden Eintrag $(Q(t)+H(t))_{j,s} \leq (M)_s$ gilt. Es werden im Folgenden die Tabellen als Vektoren je Prozess mit m Einträgen betrachtet, welche komponentenweise verglichen werden, d.h., jeder Schritt wird für jedes der m BM gemacht. Der Algorithmus setzt sich aus zwei Teilen zusammen:

1. **Sicherheitsprüfung:** Test, ob Zustand $(P_{\sigma(1)}, \ldots, P_{\sigma(n)}),\ \sigma \in S_n$ sicher ist.
2. **BM-Zuweisung:** Probieralgorithmus, der unter Verwendung der Sicherheitsprüfung eine sichere Zuweisung von BM für den aktuellen Systemzustand findet

Hier wird nur die *Sicherheitsprüfung* genauer betrachtet: Seien die Prozesse $P_1, \ldots, P_2$ unmarkiert und der Restvorrat an Betriebsmitteln gegeben durch $V(t) = M - \sum_{i=1}^{s} (H(t))_i$. Der Algorithmus arbeitet wie folgt:

(1)
- Suche nächsten unmarkierten Prozess P_i mit $Q^{max}_i(t) \leq V(t)$.
- Falls alle markiert, gehe nach (3).

(2)
- Arbeite P_i ab, d.h., $V(t) := V(t) + H_i(t)$.
- Markiere P_i, gehe nach (1).

(3)
- Falls $\forall i = 1, \ldots, n$ P_i markiert : Zustand in Zeitpunkt t sicher.
- Sonst : Zustand in Zeitpunkt t unsicher $\Rightarrow$ *Deadlock!*

Die Laufzeit des Algorithmus ist quadratisch in n und kann mit dem Verfahren von Holt auf $O(n\ \log n)$ weiter nach unten beschränkt werden.

5 Scheduling

Das **CPU-Scheduling** beschäftigt sich mit Strategien für die Reihenfolge, in der Prozesse im *ready*-Zustand in den *running*-Zustand geholt werden (d.h. ausgeführt werden). Diese Strategien wägen zwischen einer gewissen *Fairness*, einem *Prioritätssystem* oder dem *Leistungsverhalten* im Hinblick auf die CPU ab.

5.1 Strategien

Es gibt folgende gebräuchliche Strategien:

- *FIFO*: Vorgehen nach dem *FCFS*-Prinzip (first come first serve), Verwaltung der Prozesse in einer Warteschlange
- *LIFO*: gegenteiliges Vorgehen, jüngster Prozess wird als erster bearbeitet
- *LIFO-PR*: LIFO *preemptiv*
  1. LIFO-*Preemptive-Resume*: unterbrochener Prozess läuft an der selben Stelle weiter, wo er aufgehört hat
  2. LIFO-*Preemptive-Repeat*: unterbrochener Prozess wird neu gestartet
- *SJF* (Shortest-Job-First): Bevorzugung der Prozesse mit niedrigster Laufzeit (auch: SPF, Shortest Processing Time First)
- *SRPTF* (Shortest-Remaining-Processing-Time-First): laufender Prozess wird durch einen noch kürzeren Neuankömmling unterbrochen $\rightsquigarrow$ optimal bzgl. der mittleren. Wartezeit
- *SEPT* (Shortest-Expected-Processing-Time-First)
  SERPT (Shortest-Expected-Remaining-Processing-Time-First): Schätzung der Prozesslaufzeit durch
  - Durchschnitt der letzten n Prozesse
  - 80% des Maximalwerts der letzten n Prozesse etc.
  - *Exponential Averaging*: $\tau_{n+1} = \alpha \cdot t_n + (1-\alpha)\tau_n$ mit t_i die Laufzeit und τ_i die geschätzten Laufzeit des i-ten Prozesses
- *HPF* (Highest Priority First): Der älteste Prozess der höchsten nichtleeren Prioritätsklasse wird als erstes genommen.
- *RR* (Round Robin): Jeder Prozess bekommt das gleiche Quantum Q auf einer Zeitscheibe, die aktiven Prozesse befinden sich in einer FIFO-Warteschlange.
- *Multilevel-Feedback-Queueing*: Es gibt Prioritätsklassen, welche je eine Zeitscheibe besitzen, deren Quanti mit wachsender Priorität kleiner werden. Ein neuankommender Prozess wird zunächst in die höchste Priorität eingeordnet, falls er nicht fertig geworden ist, in die nächst tiefere.

Strategien, welche vernachlässigbare Umschaltzeiten zwischen den Prozessen haben, heißen **work conserving**. Eine *nicht*-work conserving Strategie ist z.B. LIFO-Preemptive-Repeat. Eine Strategie heißt **preemptiv**, falls es möglich ist, einen *running*-Prozess zu unterbrechen. Ein Schedule kann anschaulich auf einer Zeitachse durch rechteckig dargestellte Prozesse dargestellt werden (**Gantt-Chart**).

5.2 Mehrprozessorsysteme

Es gebe m identische Prozessoren in einem System. Hier gibt es kaum optimale Strategien wie z.B. *LPT* (Longest-Processing-Time-First). Ein Schedule ist eine Aufteilung von n Prozessen auf die m Prozessoren. Ein Schedule ist das beste, falls es kein Schedule gibt, dessen Dauer zur Terminierung des letzten Prozesses geringer ist. Es gibt jedoch entgegen der Annahme, dass mit mehr Prozessoren

die Leistung steigt, viele **Anomalien**. Als Schranke für die optimale Laufzeit L_{opt} kann man für ein Schedule mit Laufzeit L_{akt} die folgende Abschätzung angeben:

$$\frac{L_{akt}}{L_{opt}} \leq 2 - \frac{1}{m}$$

6 Speicherverwaltung

Der Speicher in einem Rechner ist hierarchisch gegliedert, d.h, es gibt schnelle, teure Speicher wie den CPU-Cache sowie langsame, erschwingliche Hintergrundspeicher wie Festplatten. Der Hauptspeicher ist ein schneller, flüchtiger Speicher (RAM), in dem heutzutage die Programme arbeiten. Festplatten dienen nur zur Auslagerung oder Datenspeicherung, da sie nicht-flüchtig sind. Um bestmögliche Leistung aus einem Multiprogramming-System zu holen, muss nicht nur die Scheduling-Strategie möglichst optimal sein, sondern der Speicherzugriff muss schnell von statten gehen. Da aber der Einfachheit halber, ein unendlicher (oder sehr großer) Arbeitsspeicher für die Ausführung von Programmen angenommen wird, benötigt man den Begriff des **Virtuellen Speichers**, d.h., der *logische Adressraum* erstreckt sich nicht nur über den Hauptspeicher sondern auch über Bereiche der Festplatte (d.h. den *physikalischen Adressraum*). Falls Daten nicht im Hauptspeicher vorliegen, müssen sie nachgeladen werden. Da man aber nur einen geringen Geschwindigkeitsverlust in Kauf nehmen will, ist ein effizientes Verfahren erforderlich, welches die Daten des Hauptspeichers so zusammenstellt, dass ein Optimum an Geschwindigkeit erzielt wird. Dazu gibt es die folgenden Ansätze.

6.1 Segmentierung

Der logische Adressraum wird in Segmente unterteilt. Ein **Segment** ist eine logische Einheit zusammenhängender Daten (z.B. Unterprogramm) und besitzt eine Nummer und eine (variable) Länge. Zu jedem Programm gibt es eine **Segmenttabelle**, in der jedem Segment eine Nummer s und ein Speicherbereich im physikalischen Speicher zugeordnet ist, d.h., eine *Basis* b bestehend aus der ersten Adresse des Segments und der Länge l des Bereichs (die letzte Adresse eines Segments ist folglich $b + l - 1$). Eine logische Adresse ist daher von der Form

$$(s, d), \quad d \text{ Offset mit } b \leq d < b + l.$$

Durch das Nachladen (von der Festplatte) und Verdrängen (aus dem Hauptspeicher) unterschiedlich langer Segmente entstehen unterschiedlich lange Lücken zwischen den vorhandenen Segmenten im Hauptspeicher. Es gibt die folgenden Strategien zur **Platzierung nachgeladener Segmente** in die Lücken:

- *First-Fit* (FF): Platzierung in die erste passende Lücke
- *Best-Fit* (BF): Platzierung in die kleinste passende Lücke
- *Worst-Fit* (WF): Platzierung in die größte passende Lücke
- *Rotating-First-Fit* (RFF): wie FF nur Suche der nächsten Lücke ab der zuletzt besetzten Lücke (exkl. dieser)

Es gilt nach einer Reihe von Simulationen („>“ = *besser als*)

$$RFF > FF > BF > WF.$$

Da viele kleine, unbrauchbare Lücken bei einer Segmentierung des Speichers entstehen, benötigt man **Garbage-Collection**-Verfahren, welche die Lücken zusammenfassen $\rightsquigarrow$ *Defragmentierung.*

6.2 Paging

Beim **Paging** wird der Speicher ebenfalls in Blöcke (= **Seiten**) unterteilt, welche aber alle eine feste Größe haben. Der Vorteil liegt darin, dass nun keine *externe Fragmentierung* durch Lücken mehr entsteht, allerdings kann es innerhalb einer Seite einen Verschnitt an Platz geben, der durch kleinere Programmteile entsteht (*interne Fragmentierung*). Es gibt drei Strategien für das **Nachladen von Seiten** aus dem Hintergrundspeicher:

- *Demand-Paging:* Seitenaustausch im Falle eines *Seitenfehlers* (betreffende Seite fehlt im Hauptspeicher)
- *Demand-Prepaging:* wie Demand-Paging nur Nachladen mehrerer Seiten
- *Look-Ahead:* auch Nachladen, wenn kein Seitenfehler vorliegt

6.2.1 Demand-Paging-Strategien

Bei diesen Strategien betrachtet man einen schon voll belegten Speicher. Beim Nachladen sollen dabei die unwichtigsten Seiten verdrängt werden. Der logische Adressraum sei im folgenden als $N = [0, n-1]$ (d.h. er hat die Seiten $0, \ldots, n-1$), der physikalische Adressraum als $M = [0, m-1]$ (d.h. er hat die Seitenrahmen $0, \ldots, m-1$) bezeichnet, wobei im Normalfall $n \gg m$. Die bei einem Programmaufruf erforderliche Folge von Seiten sei durch den *Referenzstring* ω bezeichnet, wobei

$$\omega = s_1 \ldots s_k \in N^k.$$

Weiter sei $S_t = \{i \in N \mid i \text{ belegt Seitenrahmen in } M\}$ der *Speicherzustand* zum Zeitpunkt t. Ein Verfahren, welches die geforderten Seitentausch-Vorgänge realisiert kann formal als *endlicher Automat A ohne Ausgabe* charakterisiert werden. Dazu sei $Q = \{q = (q_1, \ldots, q_k) \mid q_i \in N\}$ die Menge der Kontrollzustände, deren Elemente die Anordnung der gespeicherten Seiten im logischen Adressraum darstellen, es ist

$$A = (N, \{S_t\} \times Q, q_0, g_A),$$

wobei q_0 der Startzustand und

$$\begin{array}{rcl} g_A : N \times \{S_t\} \times Q & \rightarrow & \{S_t\} \times Q \text{ die Überführungsfunktion mit} \\ (r_i, S, q) & \stackrel{g_A}{\mapsto} & (S', q'). \end{array}$$

D.h., beim Zugriff auf die Seite r_i wird durch den Automaten der Zustand (S, q) in den Zustand (S', q') überführt. Wie allerdings dieser neue Zustand aussieht, hängt zum einen davon ab, ob ein *Seitenfehler* vorliegt, zum anderen aber auch von der Art der Paging-Strategie:

$(r_i, S, q) \mapsto (S, q)$	kein Seitenfehler: $r_i \in S$
$(r_i, S, q) \mapsto (S \cup \{r_i\}, q = (r_i, q_1, \ldots, q_k))$	Speicher noch nicht voll: $r_i \notin S, \lvert S\rvert \leq m$
$(r_i, S, q) \mapsto (S \cup \{r_i\} \setminus \{q_m\}, q = (r_i, q_1, \ldots, q_{m-1}))$	Seitenfehler und Speicher voll: $r_i \notin S, \lvert S\rvert = m$

Zu den einzelnen Strategien:

- *FIFO:* Älteste Seite wird verdrängt.
- *LRU (last recently used):* Wie FIFO, bei Anforderung einer bereits vorhandenen Seite wird diese jedoch wieder an den Anfang der Warteschlange gesetzt (Verjüngung).

- *Second-Chance:* Einführen eines *UseBits*, welches bei wiederholter Seitenanforderung gesetzt wird und gelöscht bei Seitenfehler oder wenn alle UseBits gesetzt sind; es wird die älteste Seite ohne UseBit verdrängt.
- *LFU (least frequently used:* Verdrängen nach Nutzungshäufigkeit
  - seit Beginn des Referenzstrings
  - innerhalb der letzten h Zugriffe
  - seit letztem Seitenfehler
- *Climb:* wie LRU, nur steigt die wiederholt aufgerufene Seite nur um einen Platz nach oben
- *Random:* zufällige Wahl der zu verdrängenden Seite
- *OPT:* optimale Strategie, Verdrängen der am längsten nicht mehr gebrauchten Seite (geringster Vorwärtsabstand)

6.2.2 Nicht-Demand-Paging-Strategien

In der ersten **Nicht-Demand-Paging-Strategie** wird die Tatsache verwendet, dass Programme durch

- Sequentielle Programmdurchläufe
- lineare Programmcodes
- geographisch geordnete Daten

sehr leicht absehbare Seitenaufrufe tätigen. Eine charakteristische Strategie ist der **OBL**-Algorithmus (One-Block-Look-Ahead). Für diesen gibt es zwei unterschiedliche Ansätze:

1. *Demand-Preparing-Version:*
   - Seite schon im Speicher: analog zu LRU
   - Seitenfehler: Falls Folgeseite noch nicht im Speicher, an letzte Position laden, sonst wie LRU vorgehen.
2. *Look-Ahead-Version:* Analog zur Demand-Preparing-Version, allerdings wird die schon im Speicher vorhandene Folgeseite wie bei LRU verjüngt.

Allgemein ist aber zu jedem Nicht-Demand-Paging-Algorithmus ein Demand-Paging-Algorithmus konstruierbar, welcher höchstens genauso viele Seitenfehler verursacht. Die Paging-Algorithmen können auch noch auf andere Arten Klassifiziert werden. Dazu sei $S(A, m, \omega) = S(m, \omega)$ die Menge der Seitennummern, welche nach Abarbeitung des Referenzstrings ω durch den Automaten A im Speicher mit m Seitenrahmen stehen.

- **Stack-Algorithmen:** Es gilt $S(m, \omega) \subseteq S(m+1, \omega)$, wobei mit wachsendem m die Anzahl der Seitenfehler sinkt. $\Rightarrow$ FIFO, Climb sind keine Stack-Algorithmen aber LRU, LIFO und OPT sind Stack-Algorithmen
- **Prioritätsalgorithmen:** Es wird unabhängig von m eine Prioritätsliste benutzt, welche bestimmt, welche der Seiten verdrängt wird. Beispiele für die Wahl der Priorität:

- LRU: wachsende Rückwärtsdistanz
- OPT: wachsende Vorwärtsdistanz
- LIFO: wachsende Zeit des Eintritts in den Hauptspeicher
- LFU: abnehmende Häufigkeit der Benutzung

6.3 Buddy-Systeme

Die **Buddy-Systeme** bieten eine dynamische und möglichst Lücken-füllende Speicheraufteilung. Dazu wird die Speichergröße einer 2er-Potenz 2^{max} angenommen. Eine Speicheranforderung einer bestimmten Größe wird einer möglichst passenden 2er-Potenz 2^p zugeordnet. Es wird nun der Speicher solange in kleinere 2er-Potenzen aufgeteilt, bis die passende gefunden ist. Diese Zerlegung wird in einem Binärbaum dargestellt, welcher *Buddy-Baum* heißt. Eine Aufteilung eines Knotens der Größe 2^p teilt den Speicher in zwei **Buddies** von jeweils der Größe 2^{p-1} auf, es sind also zwei kleinere Blöcke frei geworden ($2^{p-1} + 2^{p-1} = 2 \cdot 2^{p-1} = 2^p$). Der darüberliegende Knoten kann aber nicht mehr benutzt werden. Um auf die aktuellen freien Blöcke einer Potenz 2^p schnell zugreifen zu können, werden Listen L_p eingeführt, welche diese Blöcke in Binärdarstellung enthalten. Beim Plazieren eines neuen Segments der (am besten passenden) Größe 2^p können daher die folgenden zwei Fälle auftreten:

- $L_p \neq \emptyset$: Plaziere Segment und lösche Block aus Liste.
- $L_p = \emptyset$: Durchsuche nächst größere Listen $L_{p+1}, L_{p+2}, \ldots$ nach freiem Block, welcher dann auf passende Größe heruntergespaltet wird.

Sollen umgekehrt Segmente freigegeben werden, so wird zuerst beim zugehörigen Buddy geprüft, ob er frei ist. Falls ja, werden beide zu einem neuen Freien Knoten verschmolzen, weswegen diese Prozedur ein weiteres Mal durchzuführen ist. Es alle Buddies sukzessive verschmolzen. In der graphischen Darstellung wählt man für freie Buddies Rechtecke für andere Knoten Kreise. Im folgenden Beispiel sind zwei Anforderungen an einen 128 MB großen Speicher: dargestellt

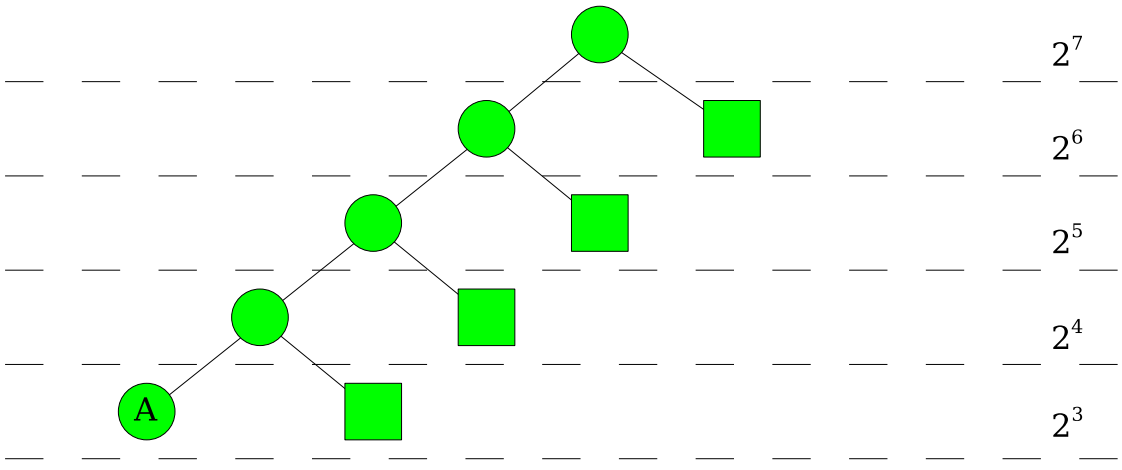

Abbildung 2: Anforderung A mit 4 MB

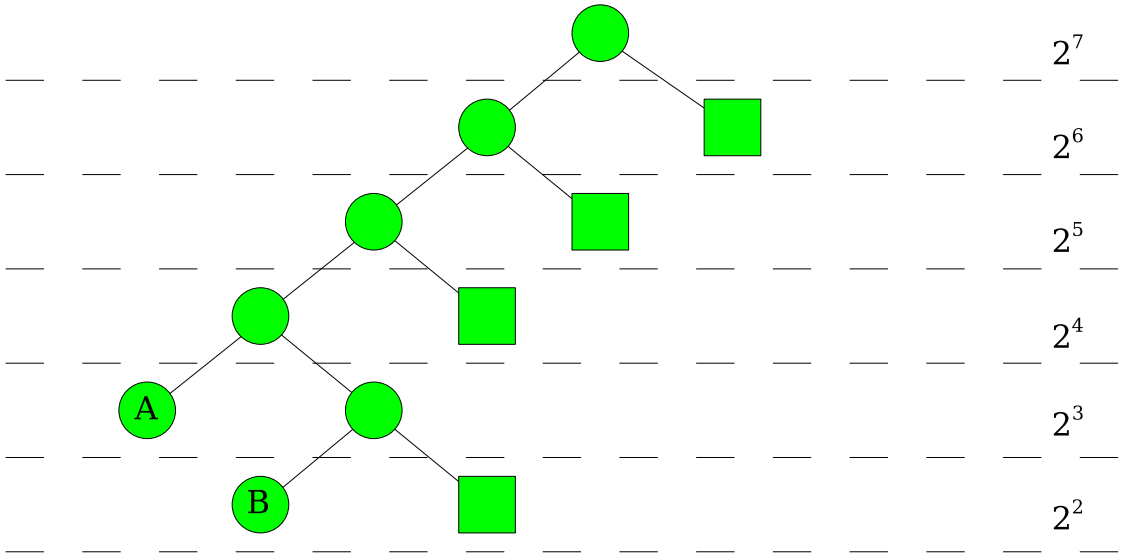

Abbildung 3: Anforderung B mit 6 MB

6.4 gewichtete Buddy-Systeme

Eine feinere Aufteilung des Speichers wird über **gewichtete Buddies** erreicht. Hier wird die Größe eines Knotens nicht halbiert, sondern etwa im Verhältnis $1:3$ geteilt, d.h., ein Block der Größe 2^{r+2} in Blöcke der Größe 2^r und $3 \cdot 2^r$ ($2^{r+2} = 2^r \cdot 2 \cdot 2 = 2^r \cdot (1+3) = 2^r + 3 \cdot 2^r$). Ein Buddy der Größe $3 \cdot 2^r$ wird im Verhältnis $2:1$ aufgeteilt ($3 \cdot 2^r = (2+1)2^r = 2^{r+1} + 2^r$). Für die Zuweisung in gewichteten Buddy-Systemen (auch bei ungewichteten einsetzbar) wird folgender Algorithmus verwendet:

7 Speicherzuteilung bei Multiprogramming

Multiprogramming (oder Multitasking) bedeutet, dass mehrere Programme gleichzeitig eine gemeinsame CPU benutzen. Neben Scheduling und Speicherverwaltung ist vor allem auch die Speicherzuteilung von Prozessen für eine gute Systemleistung wichtig. Zu dieser Betrachtung sei n der Multiprogramminggrad also die Anzahl gleichzeitig ausführbarer Programme, m die Anzahl der Seitenrahmen pro Prozess und D der Systemdurchsatz (d.h. die Leistung des Systems). Bei zu kleinem n werden zu viele Ressourcen verschwendet, ist n zu groß, so werden auf Grund der festen Speichergröße einem Prozess weniger Seitenrahmen zugeordnet. In diesem Fall würde der Verwaltungsaufwand für das Paging unverhältnismäßig groß werden, was **Thrashing** (übersetzbar mit „Prügeln“) genannt wird. Möchte man Prozessen Speicher zuteilen, wird das mittels **aktiver Rahmen** gemacht, d.h. Seitenrahmen, bei denen die Wahrscheinlichkeit hoch ist, dass sie in naher Zukunft benötigt werden. Ein Prozess bekommt also immer seine momentan aktiven Rahmen zugeordnet und neue Prozesse können erst bei genügend Platz für ihre aktiven Rahmen geladen werden. Ein Prozess soll deaktiviert werden, falls er einen Seitenfehler verursacht, aber keine Seite auf Grund ihrer Aktivität momentan ein Tauschkandidat ist. Der Speicherbereich für die aktiven Rahmen eines Prozesses darf im Falle einer entsprechenden Anforderung und Inaktivität von Seiten anderer Prozesse auf Kosten eben dieser erweitert werden.

Vor.:	$u \in \{2^0, 2^1, \ldots, 2^m, 3 \cdot 2^0, \ldots, 3 \cdot 2^{m-2}\}$ die gesuchte Blockgröße.
(0)	Setze h :=Index der Liste für u.
(1)	Falls $L_h = \emptyset$, setze $h := h + 1$ und gehe zu (2).
	Sonst: Gehe zu (4).
(2)	Falls $h > 2m$ Ausgabe *Anforderung unerfüllbar*, Stopp.
	Sonst: Gehe zu (3).
(3)	Falls $L_h = \emptyset$, setze $h := h + 1$ und gehe zu (2).
	Sonst: Setze $L_h := L_h \setminus \{u'\}$ und
	$\rightarrow$ Zerlege u' in gewichtete. Buddies u_1, u_2
	$\rightarrow$ Zerlege das kleinsten u_i weiter bis j gefunden mit $u_j = u$
	$\rightarrow$ Gehe zu (4).
(4)	Weise Block zu: $L_h := L_h \cup \{u\}$, Stopp.

Eine Darstellung der Abhängigkeit von der Rahmenzahl m und der mittleren Zeit zwischen aufeinanderfolgenden Seitenfehlern ermöglicht die Beurteilung einer Speicherzuteilungs-Strategie. Diese Funktion heißt **Lifetime-Funktion** $L(m)$. Der Graph der Funktion ist meist *sigmodal*, d.h., in „S“-Form.

7.1 Working-Set-Strategie

Als **Working-Set** zum Zeitpunkt t bezeichnet man die Menge der letzten h referenzierten Seiten eines Prozesses:

$$W(t,h) = \bigcup_{i=t-h+1}^{t} \{r_i\}, \quad \omega = r_1 r_2 \ldots, r_t, \ldots, r_T$$

Ziel ist es, durch Betrachtung der unmittelbar zurückliegenden, zuletzt referenzierten Seiten Aufschluß über ein Optimum von m bzgl. $L(m)$ zu bekommen. Es folgt

$$h \leq h' \quad \Rightarrow \quad |W(t,h)| \leq |W(t,h')| =: w(t,h').$$

Die **Working-Set-Strategie** (WS), die eine optimale Speicheraufteilung im Sinne der Lifetime-Funktion gewährleisten soll, ist durch folgende Schritte beschrieben:

1. Fenstergröße h wählen.
2. Jedem Prozess Rahmen des aktuellen Working-Sets zuweisen, bei freiem Speicher ggf. neue Prozesse starten.
3. Bei Seitenfehlern temporär eine Seite außerhalb der aktuellen Working-Sets verdrängen.
4. Bei Seitenfehlern ohne Tauschkandidat, Prozess deaktivieren.

Problematisch ist hierbei die Wahl der Fenstergröße h. Es gibt folgende Ansätze:

- **Knie-Kriterium:** Wahl von h so, dass $w(t,h) \approx m_{opt}$, wobei m_{opt} die Seitenzahl an der Knie-/Sattelstelle der Lifetime-Funktion ist. Es gilt $\frac{L(h)}{h} \geq \frac{L(h^*)}{h^*}\ \forall h^* \in N$.
- **L = S-Kriterium:** Wahl von $L(m) = S$, wobei S benötigte Zeit zur Bedienung eines Seitenfehlers (Swaptime).
- **50%-Kriterium:** Wahl von h so, dass Paging-Station zu etwa 50% ausgelastet.

7.2 VOPT

VOPT soll die optimale Speicherbelegungsstrategie sein. Das Verfahren arbeitet wie die Working-Set-Strategie, benutzt allerdings ein *Vorwärtsfenster* der Länge h:

$$V(t,h) := \bigcup_{i=t+1}^{t+h} \{r_i\}$$

Falls die Seite $r_i \in V(t,h)$, soll sie gehalten, andernfalls verdrängt werden. Es ist ersichtlich, dass VOPT und WS bzgl. der produzierten Seitenfehler keinen Unterschied bilden. Daher wird das Kostenmaß erweitert, indem nicht nur die Kosten eines Seitenfehlers sondern auch der Aufwand zum Halten einer Seite im Speicher berücksichtigt wird: $C_{VOPT} = Seitenfehlerkosten + Seitenhaltekosten$. Dann ist allerdings WS suboptimal, was die Betrachtung der mittleren zugeteilten Seitenzahl liefert. Es gilt:

Für die Fenstergröße $h = \frac{R}{U}$ ist VOPT der optimale Paging-Algorithmus bzgl. C_{VOPT}.

8 Datei- und Verzeichnissysteme

Im Folgenden werden einige Ansätze für den Aufbau von **Verzeichnisstrukturen** gegeben:

1. **Single-Level-Verzeichnis:** nur ein Verzeichnis, das alle Dateien beinhaltet
2. **Two-Level-Verzeichnis:** erste Ebene sind Benutzerverzeichnisse, zweite Ebene deren Benutzerdateien
3. **Verzeichnisbäume:** Baumstruktur von Verzeichnissen, Wurzel (root) ist Stammverzeichnis, es gibt einen
   - *absoluten Pfad*, der den Weg zu einem Verzeichnis von der Wurzel an beschreibt,
   - *relativen Pfad*, der den Weg relativ zum aktuellen Verzeichnis beschreibt.
4. **DAG**: Gerichteter, azyklischer Graph ist flexibler ($\rightsquigarrow$ Shares etc.), aber viel schwerer zu implementieren wegen der Sicherstellung, dass keine Zykel vorkommen, da sonst etwa Suchalgorithmen instabil werden.

Implementiert werden kann ein Verzeichnis mittels z.B.

- linearer Listen (langsam),

- Hashtabellen.

Dateisysteme werden in einem Schichtenmodell organisiert:

6 Anwendungsprogramme
5 logisches Dateisystem
4 Dateiorganisationsmodul
3 Basic-File-System
2 E/A-Kontrolle
1 Hardware

1. *E/A-Kontrolle:* Treiber, technische Seite der Datenübertragung
2. *Basic-File-System:* blockorientierte Anweisungen für die hardwarenahen Treiber
3. *Dateiorganisationsmodul:* Schnittstelle zwischen logischem und physikalischem Dateisystem
4. *logisches Dateisystem:* Verzeichnisstruktur, Dateistruktur

Eine Festplatte, welche als Hintergrundspeicher für Dateisysteme fungiert, ist folgendermaßen aufgebaut:

- magnetische Speichermethode
- mehrere übereinander liegende Scheiben, Schreib-/Leseköpfe
- Scheibe hat Spuren (Tracks) mit mehreren Sektoren
- Zylinder ist die Menge aller durch eine Position des Schreib-/Lesekopfarms erreichbaren Spuren
- Block ist Menge von Sektoren, Einheit für Übertragung zum Controller

Es gibt nun drei verschiedene Strategien, Dateien auf der Festplatte zu speichern:

1. **Zusammenhängende Belegung:** Jede Datei liegt in einem zusammenhängenden Feld von Blöcken. Probleme: externe Fragmentierung, Speicherplatz für eine Datei statisch, kann also nicht immer den Anforderungen entsprechen.
2. **Verkettete Belegung:** Blockposition variabel, jede Datei bildet verkettete Liste von Blöcken. Probleme: auf Grund sequentieller Suche langsam, jeder Block verschwendet wegen Zeiger auf den nächsten Block. Speicher. Lösungsansätze:
   - Bildung von *Clustern*, welche mehrere Blöcke vereinen und untereinander verlinkt sind
   - *FAT* (File Allocation Table): Zu Datei gehöriger Block in Tabelle am Anfang des Datenträgers, welche zu jedem Block den Folgeblock enthält (und spezielle EoF Markierungen bzw. 0 bei freien Blöcken). Problem: häufige Bewegung des Schreib-/Lesekopfs zwischen Tabelle und Block.
3. Zu Dateien und Verzeichnissen gibt es je einen Indexblock, der alle erforderlichen Blockverweise enthält.
   - Problem: Wahl der Größe des Indexblocks

- Lösung: Kombination von Verkettung von Indexblöcken und verschiedenen Ebenen mit unterschiedlichen Indizierungsmethoden ($\rightsquigarrow$ UNIX Inode)

Um schnell z.B. beim schreiben einer Datei auf freie Blöcke zugreifen zu können, bedient man sich der sogenannten **Free-Space-List**. Diese oft als *Bitvektor* implementierte „Liste“ enthält zu jedem Block einen entsprechenden Eintrag, ob er frei ist, oder nicht.

9 Binden und Laden von Programmen

Damit Programme laufen können, müssen sie zunächst durch einen **Binder** (**Linker**) aus ihrer Modularität in eine kompakte Form gebracht werden, um dann vom **Lader** in den Speicher gebracht zu werden. Für den Linker gibt es zum einen die Möglichkeit, die Programmmodule beim Laden eines Programms (*Linkage-Loader*) oder unmittelbar nach Übersetzen der Module (*Linkage-Editor*) zu binden. Der Linkage-Editor erstellt aus den einzelnen Programmmodulen und evtl. Steuerbefehlen ein fertiges verschiebbares Programm, d.h., von der Form, dass es der Loader in den Hauptspeicher laden kann. Der (verschiebende) Lader (*Relocating Loader*) hat dann die Aufgabe, die relativen Adressen des Programmcodes in absolute Adressen im Hauptspeicher umzurechnen und somit einen ausreichend großen Speicherbereich im zu lokalisieren.

10 Systemsicherheit

In dem Zusammenhang der **Sicherheit** werden die zu schützenden Betriebsmittel, Speicherbereiche etc. als *Objekte* bezeichnet, auf welchen *Prozessoperationen* unter bestimmten Rechten ausgeführt werden dürfen oder nicht. Ein **Zugriffsrecht** ist ein Tupel $(obj, \{r_i\} =: R$, wobei obj ein Objekt und R die Menge der erlaubten Operationen auf diesem Objekt ist. Dann definiert sich ein **Schutzbereich** als eine Menge von Zugriffsrechten. Ein Zuordnung eines Prozesses zu einem Schutzbereich heißt *statisch*, falls diese nicht änderbar für die Prozessdauer ist. Ansonsten heißt sie *dynamisch*, was wesentlich schwieriger zu implementieren ist. Die Realisierung solcher Schutzbereiche kann auf Benutzerebene, Prozess- oder sogar Funktionsebene erfolgen, auf welcher ein Funktionsaufruf innerhalb eines Prozesses eine Änderung der Schutzmechanismen hervorrufen kann. Die Verwaltung mehrerer Schutzbereiche erfolgt über eine **Zugriffsmatrix** $Z \in R \times \{obj\}$, welche selbst auch als Objekt verfügbar ist. Die Idee ist dabei, dass der *Eigentümer eines Objektes Z* in seiner *Spalte* ändern darf, er darf also Rechte für die Prozesse, die auf sein Objekt zugreifen können, vergeben. Andererseits kann die *Kontrolle eines Schutzbereiches* die entsprechende *Zeile* von Z ändern, also Rechte bearbeiten. Solche Matrizen können mittels folgender Mechanismen implementiert werden:

- *Zugriffslisten für Objekte:*
  - jedes Objekt hat eine Liste
  - jeder Schutzbereich mind. ein Recht
  - es gibt einen *default* Eintrag zur Reduzierung der Daten
- *Capability-Listen:*
  - jeder Schutzbereich hat eine Objekt-Rechte-Liste
  - jedes Objekt hat mind. ein Recht

  - Zugriff erfolgt nur über die Capability (physikalischer Name bzw. Adresse des Rechts)

- *Schlüssel-Schloß-Mechanismus:*
  - jedes Objekt hat eine Schloß-Recht-Liste
  - jeder Schutzbereich hat eine Schlüssel-Recht-Liste
  - passen Schlüssel auf Schloß (beides Bitmuster), hat Prozess Zugriff auf Objekt

Um eine Ebene höher das System vor bestimmten Personen zu schützen, ist eine **Authentifizierung** des Benutzers nötig, welche über Mechanismen wie *Passwort-Eingabe*, *Fingerabdruck*, *Chipkarte* etc. erfolgen kann. Der unsichere Vorgang bei einer Authentifizierung ist die Übertragung der Zugangsdaten über z.B. das Internet oder andere Netze. Aus diesem Grund werden die Daten so **chiffriert/kodiert**, dass die kodierten Daten für unberechtigte Personen nicht entzifferbar sind. Das erreicht man ebenfalls über Schlüssel-Schloß-Mechanismen, bei welchen ein Schlüsselwort zur Kodierung der zu schützenden Daten dient, Kodier- und Dekodieralgorithmus aber nicht unbedingt geheim sind.

11 Kommunikationsmodelle in verteilten Systemen

Da das OSI-Schichtenmodell stellt einen zu hohen Verwaltungsaufwand bei der Betrachtung einfacher Kommunikation in **verteilten Systemen** dar. Es gibt zwei weitere Ansätze, einfache und damit effiziente Kommunikationsmodelle zu entwerfen:

1. **Client/Server-Modell:**
   - Server: Nimmt Anfragen von Client entgegen, sendet Antwort.
   - Client: Schickt Anfragen an Server.
   - Jede send/recieve-Operation blockiert den Prozess.
   - *machine-process-Adressierung*: $IP.Prozessnummer$ oder $Prozessnummer@IP$
   - *ortstransparente Adressierung*:
     - Lokalisierungspaket broadcasten (an alle schicken)
     - Server meldet Position.
     - Kommunikation läuft wie gewohnt ab.
   - *ortstransparente Adressierung mit Name-Server/Trader*:
     - Anfrage an den Name-Server nach der Adresse des gewünschten Servers
     - Name-Server meldet Position.
     - Kommunikation läuft wie gewohnt ab.
2. **Remote-Procedure-Call:** Aufruf entfernter Unterprogramme ermöglicht *lokal* erscheinende verteilte Berechnungen. Der Vorgang läuft folgendermaßen ab:
   - Möchte ein Client-Prozess auf einem entfernten System ein Unterprogramm aufrufen, informiert er zunächst den *Client-Stub*, welcher ein anderer lokaler Prozess ist, mit den nötigen Daten und Parametern.
   - Der Client-Stub („Stellvertreter-Prozess“) generiert aus der Anfrage eine Nachricht und schickt diese über eine Anfrage an den lokalen System-Kern an das entfernte System. Danach blockiert der Client-Stub, während er auf die Antwort wartet.

- Der Kern des Server-Systems hingegen leitet die Nachricht an den *Server-Stub* weiter, der die Daten auspackt und das entsprechende Unterprogramm aufruft. Der Server-Stub wird dann inaktiv, bis das Ergebnis vorliegt.
- Die Ergebnis-Daten werden dann über den selben Weg wieder in eine Nachricht verpackt und über den Server-System-Kern an den Client geschickt, der dann seine Arbeit fortsetzen kann.