Betriebssysteme I, Studiengang Informatik

Hans-Peter Oser*

16. Februar 2009

Inhaltsverzeichnis

*Professor an der Fachhochschule Nordwestschweiz

Abbildungsverzeichnis

1 Was ist ein Betriebssystem?

Unter einem Betriebssystem versteht man eine Ansammlung von Steuerungsprogrammen und Hilfsroutinen, die die Benutzung eines Rechners und der daran angeschlossenen Geräte für den Menschen vereinfachen. Man kann sich vorstellen, dass ein Betriebssystem zwischen den Benutzer (bzw. sein Programm) und die Hardware tritt, die die spezifizierte Aufgabe tatsächlich ausführt. Dem Benutzer wird, durch die Software des Betriebssystems ein Rechner vorgespiegelt, der zu wesentlich komplexeren Operationen in der Lage ist, als es die reine Hardware wäre. Dadurch wird die Aufgabe, ein bestimmtes Programm zu schreiben, oder auch nur ein bestehendes Programm zur Ausführung zu bringen, wesentlich vereinfacht. Die Hauptaufgabe eines Betriebssystems ist aber nicht Programme zur Ausführung zu bringen, sondern die Verwaltung der Betriebsmittel eines Rechnersystems. Unter Betriebsmitteln eines Rechnersystems verstehen wir:

- den, die Prozessoren
- den Hauptspeicher
- die Massenspeicher
- die Peripheriegeräte

Betriebssysteme unterscheiden sich nicht stark von anderen Programmen. Die Schwierigkeiten, die bei der Konstruktion von Betriebssystemen auftreten, treten auch bei der Konstruktion von Compilern oder grossen Anwendungsprogrammen auf.
Betriebssysteme haben eine historische Bedeutung erlangt in zweifacher Hinsicht:

- Gr366sse und Komplexität der Software: Viele moderne Betriebssysteme beinhalten einen Software Aufwand von 50 bis 100 Arbeitsjahren.
- Mehrprogrammbetrieb: Mehrere Benutzer können die vorhandenen Betriebsmittel gleichzeitig benützen.

2 Schichtenmodell eines Betriebssystems

Ein Betriebssystem besteht aus den folgenden Schichten:

- Anwenderprogramme:
  Als Anwenderprogramme bezeichnet man die Programme zur Lösung eines Anwenderproblems. Zum Beispiel:
  - Tabellenkalkulation (spread sheets, work sheets)
  - Buchhaltung
  - Zeichenprogramme (drafting, CAD)
- Dienstprogramme:
  Die meisten Beschreibungen der Betriebssysteme erklären die Anwendung der Dienstprogramme. Was verstehen wir unter einem Dienstprogramm? Dienstprogramme sind Programme, die den Betrieb eines Rechnersystems vereinfachen. Beispiele sind:
  - Compiler

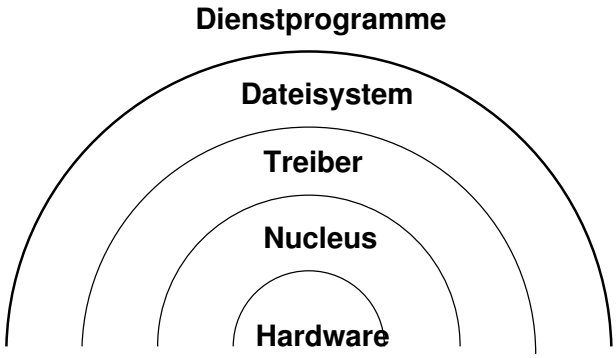


Abbildung 1: Schichtenmodell eines Betriebssystems

  - Editor
  - Debugger
  - Dateiunterhalts-Programme
  - Linker

  Der Zugriff auf das Betriebssystem durch Anwender- und Dienstprogramme erfolgt:

  - entweder direkt, mit einem Systemaufruf (system call) oder
  - indirekt über ein Dienstprogramm

- Dateisystem:
  Das Dateisystem verwaltet die Massenspeicher. Durch das Dateisystem ist es möglich, die Dateien logisch (mit einem Namen) anzusprechen.

- Treiber:
  Treiber oder Treiberprogramme bilden eine Brücke zwischen dem am Rechner angeschlossenen Peripheriegerät (Hardware) und dem Nucleus einerseits und dem Dateisystem oder Anwenderprogramm andererseits.

- Nucleus oder Kernel
  Programme zur Verwaltung der Betriebsmittel bei simultaner Benutzung des Prozessors durch mehrere Anwendungen.

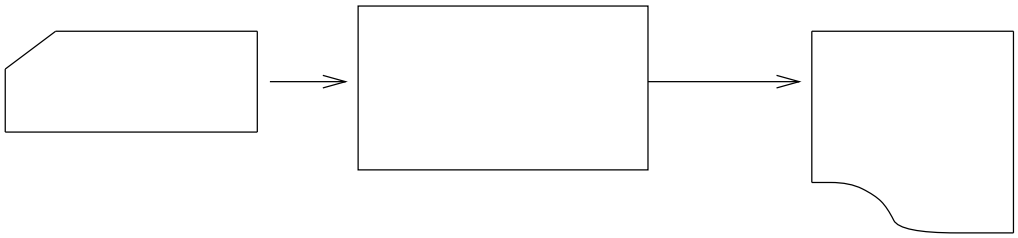


Abbildung 2: Batch System

3 Verschiedene Arten von Betriebssystemen

In diesem Kapitel wird ein Überblick über die verschiedenen Arten von Betriebssystemen gegeben. In chronologischer Reihenfolge werden die Charakteristiken der verschiedenen Betriebssysteme dargestellt.

3.1 Stapel Systeme (Batch Systems)

Die Hauptaufgabe dieses Betriebssystems ist die Initialisierung des Systems für den Start eines Batch-Job. Aus den Betriebssystem-Schichten sind die Treiber, zum Teil ein Dateisystem und ein Dienstprogramm zur Steuerung und Initialisierung des Batch-Jobs vorhanden.

Charakterisierung des Betriebssystems:

- Rein sequentielles Arbeiten: Eingeben-Ausführen-Ausgeben
- Einbenützer Betriebssystem

Organisation des Betriebs:

- Einfachste Organisationsform:
  Open Shop! Jeder Benutzer reserviert sich den Rechner für z.B. 15 Minuten
  - Effektive Prozessorzeit (Annahme) 1 Minute
  - Leistung des Systems kann charakterisiert werden durch:
    - Prozessorauslastung = Rechenzeit/Gesamtzeit
    - Durchsatz = Anzahl der Stapelaufträge pro Zeiteinheit
    - Bei Open Shop: Prozessorauslastung: 7%,
      Durchsatz: 4 Aufträge/Stunde
- Closed Shop: Stapelaufträge werden einem Operateur übergeben.
  - Stapelauftrag braucht nur noch 1.8 Min. (Ein-Ausgabe-Bearbeitung)
  - Die Prozessorausnützung wird auf 55% verbessert, bei 33 Aufträgen/h.
- Verbesserung möglich, durch Verwendung von 3 Systemen: Eingabe auf Band - Bearbeitung - Ausgabe von Band (90%, 55 Aufträge). → Nachteile: mehrere Operateure notwendig zum Führen des Systems!

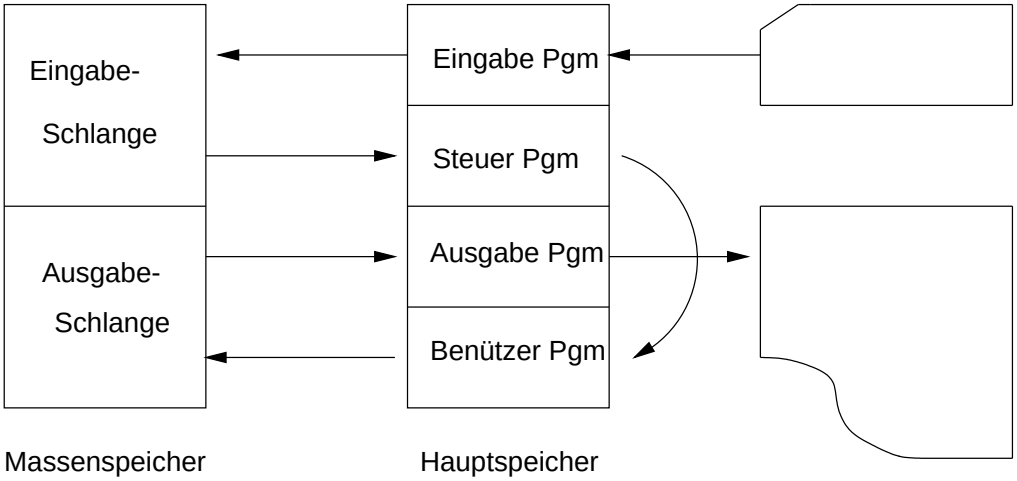


Abbildung 3: Spooling System

3.2 Spulensysteme (Spooling)

- Mehrprogramm Betriebssystem

Zur besseren Auslastung des Systems sind mehrere Programme gleichzeitig aktiv. Neben den bereits besprochenen Schichten *Treiber*, *Dateisystem* und *Dienstprogramme* wird hier noch ein Verwalter der Betriebsmittel ein *Kernel* oder *Nucleus* benötigt.

- Wahlfreier Zugriff auf Massenspeicher
- $\Longrightarrow$ Spooling
- Interrupts

Mit diesem Konzept, werden die Eingabe, Ausgabe und Bearbeitung überlagert $\Longrightarrow$ Mehrprogramm System = Multiprogramming.

3.3 Interaktive Systeme (Timesharing Systems)

Die Grundidee dieser Systeme ist es, den Prozessor gleichzeitig durch eine Menge von Benutzern über Datensichtstationen als komfortable Tischrechner benützen zu lassen. Diese Benutzer haben direkten Zugriff auf ihre Programme und Daten, die auf dem Massenspeicher verfügbar gehalten werden. Jeder Benutzer kommuniziert direkt mit dem Betriebssystem (Shell), das seine Betriebsmittel jedem Benutzer für kurze Zeitabschnitte zuteilt. In diesen Betriebssystemen sind die Schalen: *Nucleus*, *Treiber*, *Dateisystem*, und *Dienstprogramme* vorhanden. Im Nucleus dieser Betriebssysteme müssen Stammfunktionen (primitives) für die Synchronisierung und Kommunikation der einzelnen Anwenderprogramme vorhanden sein. Besondere Problemkreise in diesen Betriebssystemen sind:

- Schutz vor Fehlern anderer Benutzer
- Schutz der persönlichen Daten vor Zugriff von Unberechtigten

- Steuerung der Prozessorzuteilung (meist Reihum)
- Speicherverwaltung

3.4 Prozessrechner Systeme

Die Aufgabe dieser Systeme ist es, mit fest vorgegebenen Algorithmen innerhalb bestimmter, zu garantierender Zeiten auf äussere Signale zu reagieren. Alle zeitkritischen Programme müssen speicherresident gehalten werden. Ereignisgesteuert schaltet der Nucleus zwischen den Programmen hin und her. Zugunsten von Schnelligkeit wird oft auf auf Komfort verzichtet.

3.5 Vernetzte Workstations

Die Aufgabe dieser Betriebssysteme ist es, einem Benutzer alle im Netzwerk vorhandenen Dienste und Betriebsmittel möglichst einfach zur Verfügung zu stellen. Das eigentliche Betriebssystem hat die Schalen: Nucleus, Treiber (Kommunikation) und Dateisystem. Im Betriebssystem eingebaut ist ein Windowsystem (X11), mit dem man, in je einem Fenster auf ganz verschiedene Rechner und Betriebssysteme zugreifen kann.

3.6 Cluster

Die Betriebsmittel von zwei oder mehr Rechnern können zu einem System zusammengefügt werden. Die Kopplung der beiden Systeme erfolgt über ein schnelles Netzwerk oder spezielle Hardware. Das Scheduling des Clusters erlaubt die Rechenlast gleichmässig auf die einzelnen Cluster Systeme zu verteilen. In diesen Betriebssystemen sind alle genannten Schichten vorhanden. Zusätzlich braucht es eine Schicht für das *load balancing* die meist auf die Schicht *Dateisystem* aufgesetzt ist.

4 Parallele Aktivitäten

Falls ein Rechner mehrere Anwendungen parallel verarbeiten kann, so führt er jede Anwendung als separate Aktivität aus. Diese einzelnen Aktivitäten werden in der Praxis, je nach dem als:

- Prozess
- Thread oder
- Task

bezeichnet. In JAVA sind *Threads* implementiert. Mit Threads kann man Programme erzeugen die *gleichzeitig* mehrere Aufgaben erledigen. Threads ermöglichen sogenannte Parallelprogrammierung. Wenn mehrere Aufgaben gleichzeitig erledigt werden spricht man von Multitasking oder Multiprogramming. Im folgenden Text wird der Begriff *Prozess* verwendet um allgemeine Aktivitäten zu bezeichnen. Bei konkreten Algorithmen, die mit JAVA formuliert werden wird der Begriff *Thread* verwendet. Der genaue Unterschied zwischen diesen beiden Begriffen wird später erklärt.

4.1 Was ist das *ein Prozess*?

Die Arbeit die ein Prozessor erledigt kann auf folgende zwei Arten betrachtet werden:

arbeitsorientiert = man verfolgt den Prozessor bei seiner Arbeit.

aufgabenorientiert = Programme sind logische Einheiten, der Prozessor wird diesen als Betriebsmittel zugeteilt.

Auch in anderen Wissensgebieten wird oft eine aufgabenorientierte Gliederung von Fakten angewandt (Beispiel: Weltgeschichte).

$\Longrightarrow$ Bei einem Prozessorsystem führt die aufgabenorientierte Betrachtungsweise zu viel einfacheren Strukturen. Ein Programm kann teilweise abgearbeitet sein, ohne dass der Prozessor zur Verfügung steht.

Definitionen:

Ein *Programm* ist ein statisches Textstück, das eine Folge von Aktionen spezifiziert, die von einem oder mehreren Prozessoren auszuführen sind.

Ein *Prozess* ist eine durch ein Programm spezifizierte Folge von Aktionen, deren erste begonnen, deren letzte aber noch nicht abgeschlossen ist. In manchen Betriebssystemen wird anstelle von Prozess auch der Begriff *Task* oder *Thread* verwendet.

Weitere mögliche Umschreibungen eines *Prozesses*:

- Ein Programm in Ausführung
- Eine asynchrone Aktivität
- Die *Belebung* einer Prozedur
- Wird manifestiert durch einen Prozess-Steuerblock (pcb)
- Einheit der ein Prozessor zugewiesen werden kann
- Zu bearbeitende Einheit (dispatchable unit)

4.2 Prozess Zustände

Während seiner Existenz durchläuft ein Prozess eine Reihe von Zuständen. Verschiedene Ereignisse können eine Änderung des Prozess-Zustandes bewirken.

Prozesszustände

ausführend (running): Der Prozess besitzt im Augenblick den Prozessor.

bereit (ready): Der Prozess könnte den Prozessor gebrauchen, falls dieser verfügbar wäre.

blockiert (blocked): Der Prozess wartet auf ein bestimmtes Ereignis, bevor er weiterarbeiten kann.

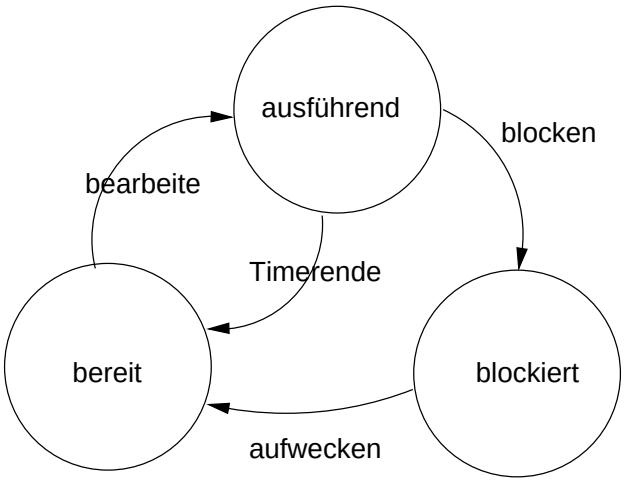


Abbildung 4: Einfaches Prozess Zustandsdiagramm

Der Einfachheit halber betrachten wir ein System mit nur einem Prozessor. Nur ein Prozess kann in einem solchen System gleichzeitig *ausführend* sein. Verschiedene Prozesse können *bereit* sein. Zur Verwaltung der Prozesse erstellen wir eine Bereit-Liste in die alle Prozesse die *bereit* sind nach Prioritäten geordnet eingetragen werden. Die Prozesse die *blockiert* sind, werden dort eingetragen wo sie warten. Eine geordnete Liste ist sinnlos, da man die Sequenz der erwarteten Ereignisse nicht kennt.

4.3 Übergänge von einem Prozesszustand in einen anderen (Transitions)

Sobald ein Prozess zuvorderst in der Bereit-Liste steht, und der Prozessor verfügbar wird, erhält dieser Prozess den Prozessor. Die Zuweisung des Prozessors wird durch den *Bearbeiter* (= dispatcher) vorgenommen:

bearbeite(Prozessname): bereit/ready → ausführend/running

Damit ein bestimmter Prozess den Prozessor nicht monopolisieren kann, gibt es in vielen Betriebssystemen einen Timer der periodisch die Prozesse unterbricht, und dem Betriebssystem (Bearbeiter) den Prozessor zurückgibt:

Timerende(Prozessname): ausführend → bereit

Falls ein ausführender Prozess eine E/A Operation startet, so gibt dieser Prozess den Prozessor freiwillig frei bis zum Abschluss der E/A Operation:

blocken(Prozessname): ausführend → blockiert

Sobald die E/A Operation beendet ist, wird der Prozess wieder aufgeweckt:

aufwecken(Prozessname): blockiert → bereit

Nur der Übergang *blocken* wird vom Prozess selbst verlangt, alle übrigen Übergänge werden durch das Betriebssystem gesteuert.

4.4 Der Prozess Steuerblock pcb

Mit dem pcb wird ein Prozess einem Betriebssystem bekanntgemacht. Der pcb ist eine Datenstruktur die nachstehendes enthält:

- aktueller Zustand des Prozesses
- eindeutige Identifikation des Prozesses
- Prozess Priorität
- Zeiger zum Speicher dieses Prozesses
- Zeiger zu den zugewiesenen Betriebsmitteln
- Bereich zum Retten der Register

Der pcb definiert den Prozess dem Betriebssystem gegenüber. Damit ein Prozesswechsel schnell stattfinden kann, haben verschiedene Prozessorsysteme spezielle Register die immer zum aktuellen pcb zeigen, oder Hardware Instruktionen die einen Prozesswechsel durchführen können.

4.5 Operationen auf dem Objekt *Prozess*

Damit das Betriebssystem die Prozesse führen kann, sind eine Reihe von Operationen notwendig:

erzeugen: (create) eines Prozesses

vernichten: (destroy) eines Prozesses

aufschieben: (suspend) eines Prozesses

wiederaufnehmen: (resume) eines Prozesses

ändern: (change) der Prozesspriorität

blockieren: (block) eines Prozesses

aufwecken: (wakeup) eines Prozesses

bearbeiten: (dispatch) eines Prozesses

Das Erzeugen eines Prozesses verlangt unter anderen nachstehende Operationen:

- Zuweisung eines Namens zum Prozess
- Einfügen dieses Prozesses in die Liste der bekannten Prozesse
- Bestimmung der anfänglichen Priorität
- Erzeugen eines pcb
- Zuweisung der anfänglichen Betriebsmittel

Ein Prozess kann einen weiteren Prozess erzeugen. Einen so erzeugten Prozess nennt man *Sohn-Prozess* (child), der erzeugende Prozess wird *Vater-Prozess* (parent) genannt.

Die Vernichtung eines Prozesses bedeutet seine vollständige Auslöschung aus allen Tabellen, alle Betriebsmittel werden zurückgegeben.

Ein aufgeschobener Prozess kann nicht fortfahren, bis er von einem anderen Prozess wiederaufgenommen wird. Das Aufschieben ist eine wichtige Operation. Normalerweise dauert die Aufschiebung nur kurze Zeit. Oft erfolgt eine Aufschiebung durch das Betriebssystem um einer zu grossen Systembelastung zu begegnen. Bei längerer Aufschiebung sollten die Betriebsmittel freigegeben werden. Hauptspeicher sollte sofort freigegeben werden, gewisse Geräte können einem Prozess noch eine gewisse Zeit belassen werden.

Wiederaufnehmen eines Prozesses bedeutet ein Fortfahren am Ort wo er aufgeschoben wurde.

Das Vernichten eines Prozesses ist komplizierter, falls der Prozess neue Prozesse erzeugt hat. Oft werden die *Sohn-Prozesse* ebenfalls zerstört. In anderen Fällen sind diese unabhängig von den *Vater-Prozessen* und existieren weiter.

4.6 Aufschieben und Wiederaufnehmen

Aufschieben und Wiederaufnehmen sind wichtige Operationen zur Führung der Prozesse. Damit werden:

- Bei schlechtem Funktionieren des Gesamtsystems einzelne Prozesse aufgeschoben, bis das Problem behoben ist.
- Bei zweifelhaften Teilresultaten, der betreffende Prozesses aufgeschoben bis das richtige Funktionieren sichergestellt ist.
- Bei grossen Belastungsschwankungen in einem System einzelne Prozesse zeitweilig aufgeschoben und sobald möglich wiederaufgenommen.

Mit der Einführung der beiden Operationen *Aufschieben* und *Wiederaufnehmen* werden die Prozessübergänge erweitert und zwei neue Zustände eingeführt: *aufgeschoben-blockiert* und *aufgeschoben-bereit*. Unterhalb der gestrichelten Linie sind aufgeschobene- oberhalb sind aktive Zustände. Das Aufschieben erfolgt durch einen anderen Prozess. Bei einem Mehrprozessorsystem kann ein ausführender Prozess einen auf einem anderen Prozessor ausführenden Prozess aufschieben. Ein Prozess der *bereit* oder *blockiert* ist, kann nur von einem anderen Prozess aufgeschoben werden.

Folgende 5 Zustandsübergänge sind neu:

aufschieben(Prozessname): bereit → aufgeschoben-bereit/suspended-ready

Ein Prozess kann nur einen anderen damit aufschieben.

wiederaufnehmen(Prozessname): aufgeschoben-bereit → bereit

Ein Prozess der aufgeschoben ist, kann nur durch einen andern Prozess bereit gesetzt werden.

aufschieben(Prozessname): blockiert → aufgeschoben-blockiert

Ein Prozess der blockiert ist kann durch einen anderen Prozess aufgeschoben werden.

wiederaufnehmen(Prozessname): aufgeschoben-blockiert → blockiert

Ein aufgeschoben-blockierter Prozess kann durch einen anderen Prozess wiederaufgenommen werden.

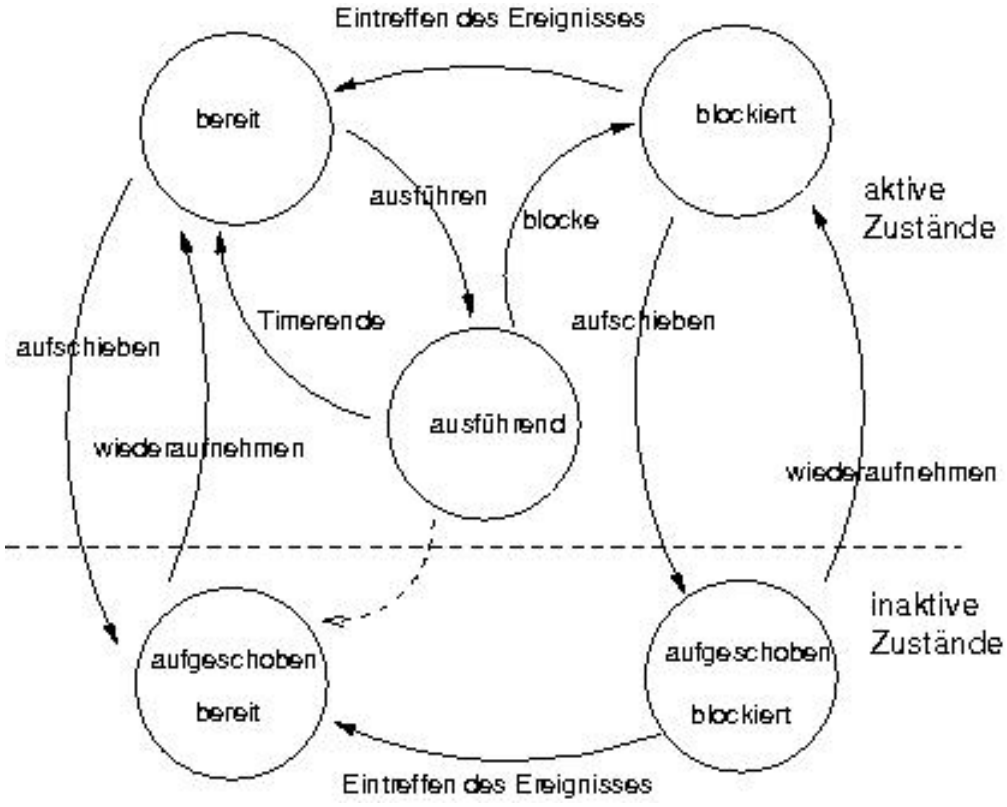


Abbildung 5: Vollständiges Prozess Zustandsdiagramm

eintreffen eines Ereignisses(Prozessname): aufgeschoben-blockiert → aufgeschoben-bereit

Warum sollen blockierte Prozesse aufgeschoben werden?
Anstelle des Aufschiebens eines blockierten Prozesses könnte man vorerst auf das Ereignis warten. Das Ereignis könnte aber nie eintreten, oder lange verzögert werden. Nach dem Aufschieben kann der Prozess auf Disk ausgelagert werden und sein Hauptspeicher freigesetzt werden. **Aufschieben und Wiederaufnehmen wird vor allen zur Steuerung der Prozesse verwendet!**

4.7 Unterbrechungsbehandlung

Sobald eine Unterbrechung (Interrupt) auftritt, so

- übernimmt das Betriebssystem die Steuerung
- das Betriebssystem rettet den Zustand des aktuellen Prozesses, dieser wird oft im pcb abgelegt.
- Das Betriebssystem analysiert die Unterbrechung und übergibt die Steuerung der Interrupt Service Routine ISR.

Eine Unterbrechung kann durch ein externes Ereignis, oder durch einen Prozess ausgelöst werden. Nur freigegebene (enabled) Unterbrechungen werden zugelassen; gesperrte (disabled) Unterbrechungen bleiben schwebend (pending). Das Betriebssystem kann externe Unterbrechungen sperren. Sobald die Unterbrechungsbehandlung abgeschlossen ist, wird der Prozessor entweder:

- dem unterbrochenen Prozess (nicht präemptiver Prozess) oder
- dem Prozess mit der höchsten Priorität (präemptiver Prozess) zugeteilt.

4.8 Der Nucleus eines Betriebssystems

Alle die Prozesse betreffenden Operationen werden durch den Nucleus (oder Kernel) des Betriebssystems gesteuert. Im allgemeinen stellt der Nucleus nur einen kleinen Teil des Betriebssystems dar; er ist aber einer der meistbenützten Programmteile. Aus diesem Grunde muss der Nucleus immer speicherresident bleiben. Eine der wichtigsten Funktionen ist die Unterbrechungsbehandlung. Während der Behandlung der Unterbrechungen werden gewisse Unterbrechungen gesperrt. Bei einem grossen Anfall von Unterbrechungen können die Unterbrechungen für einen grossen Anteil der Zeit gesperrt sein. Dies kann zu schlechten Unterbrechungs-Antwortzeiten führen; aus diesem Grunde: Die Unterbrechungsbehandlung des Nucleus beinhaltet nur das absolute Minimum. Alle weitergehenden Bearbeitungen werden mit einem Prozess erledigt.

Nucleus-Funktionen:

- Unterbrechungs-Behandlung
- Erzeugen und Vernichten von Prozessen
- Ändern der Prozess-Zustände
- Zuweisung der Prozessoren zu den Prozessen (dispatching)
- Aufschieben und Wiederaufnehmen von Prozessen

- Prozess Synchronisation
- Interprozess Kommunikation
- Manipulation an den pcb's
- Unterstützung der E/A
- Unterstützung der Speicherverwaltung
- Unterstützung des Dateisystems
- Unterstützung von Abrechnungen über Betriebsmittelbenützung

Diese Nucleusfuktionen werden als *primitives* (Stammfunktionen) bezeichnet.

4.9 Threads in JAVA

Ein Thread ist ein einzelner sequentieller Steuerfluss innerhalb eines Programms. Jeder Thread hat seinen eigenen Context (Instruktionszeiger, Stack). Ein Thread kann auf folgende beiden grundsätzlichen Methoden erzeugt werden:

- Subclassing und Überschreiben der *run* Methode
- Verwenden der *Runnable* Interfaces

Subclassing und Überschreiben der *run* Methode

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

Der Class Constructor *SimpleThread* nimmt einen String und übergibt diesen der Superclass. Die nächste Methode ist die *run* Methode, dies ist **die** Methode eines Threads: dort wird die Arbeit des Threads bestimmt. Die *SimpleThreadDemo* Klasse erzeugt zwei Threads und lässt diese parallel ablaufen.

```
public class SimpleThreadDemo {
    public static void main (String[] args) {
        new SimpleThread("FHA").start();
        new SimpleThread("FHNW").start();
    }
}
```

In der Main Methode werden sofort zwei Threads gestartet und folgender Output generiert:

```
0 FHA
0 FHNW
1 FHNW
2 FHNW
1 FHA
3 FHNW
2 FHA
3 FHA
4 FHNW
4 FHA
DONE! FHNW
DONE! FHA
```

Verwenden des *Runnable* Interfaces
Die Verwendung des *Runnable* Interfaces ist unbedingt notwendig falls man bereits von einer anderen Methode ableitet. Das Programm *ColorBoxes.java* ist aus dem Buch **Thinking in Java** von Bruce Eckel und zeigt die Verwendung des *Runnable* Interfaces.

```
// ColorBoxes.java
// Using the Runnable Interface
import java.awt.*;
import java.awt.event.*;


class CBox extends Canvas implements Runnable{
    private Thread t;
    private int pause;
    private static final Color[] colors={
        Color.black, Color.blue, Color.cyan, Color.darkGray, Color.gray,
        Color.green, Color.lightGray, Color.magenta, Color.orange,
        Color.pink, Color.red, Color.white, Color.yellow
    };
    private Color cColor = newColor();
    private static final Color newColor(){
        return colors[(int) (Math.random() * colors.length) ];
    }
    public void paint(Graphics g){
        g.setColor(cColor);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
    public CBox(int pause){
        this.pause = pause;
        t = new Thread(this);
        t.start();
    }
    public void run(){
        while(true) {
            cColor = newColor();
            repaint();
            try{
                t.sleep(pause);
            } catch(InterruptedException e) {}
        }
    }
}
```

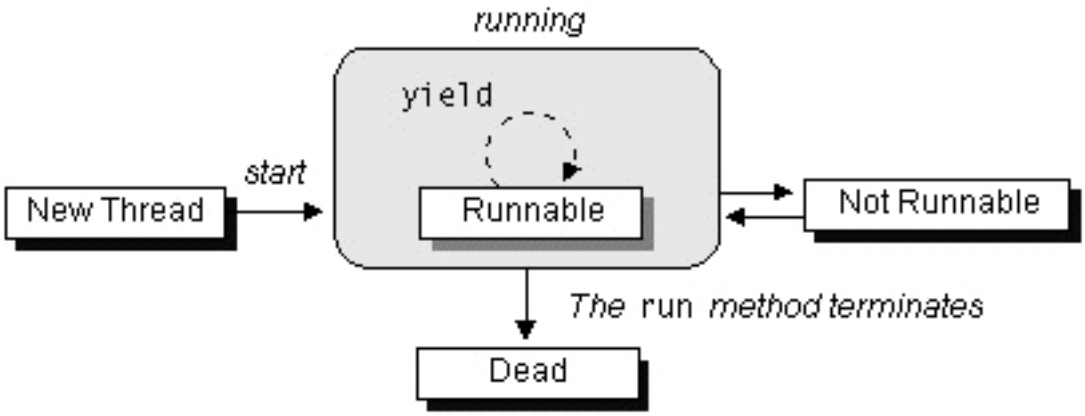


Abbildung 6: Thread Life Cycle

```
public class ColorBoxes extends Frame {
    public ColorBoxes(int pause, int grid){
        setTitle("ColorBoxes");
        setLayout(new GridLayout(grid, grid));
        for (int i =0; i <grid * grid; i++)
            add(new CBox(pause));
        addWindowListener(new WindowAdapter(){
                public void windowClosing(WindowEvent e) {
                    System.exit(0);
                }
        });
    }
    public static void main(String[] args){
        int pause = 50;
        int grid =      8;
        if(args.length >0)
            pause = Integer.parseInt(args[0]);
        if(args.length > 1)
            grid = Integer.parseInt(args[1]);
        Frame f = new ColorBoxes(pause,grid);
        f.setSize(500,400);
        f.setVisible(true);
    }
}
```

Lebenszyklus der Threads
Ein Java-Thread durchläuft minimal (abhängig vom darunterliegenden Betriebssystem) die Zustände wie in der Abbildung 6 gezeigt.

Thread Priorität
JAVA unterstützt ein sogenanntes *fixed priority scheduling*. Dabei wird der Prozessor jeweils dem Thread mit der höchsten Priorität zugeteilt. Die Priorität liegt im Bereich `MIN_PRIORITY` und `MAX_PRIORITY`. Je höher der integer Wert liegt desto grösser ist die Priorität. Das Scheduling ist präemptiv. Das strikte Einhalten der Priorität ist aber nicht garantiert. Die Priorität eines Threads kann mit der Methode *setPriority* gesetzt werden.

5 Gleichzeitige Aktivitäten

In diesem Kapitel wird der Zugriff auf gemeinsame Daten betrachtet. Für die dabei entstehenden Probleme wird Schritt für Schritt eine Lösung aufgezeigt. Die in diesem Kurs verwendeten Algorithmen werden in der Sprache JAVA formuliert.

Die fortschreitende Verkleinerung und Kostensenkung bei der Rechnerhardware bringt einen zunehmenden Trend zu grosser Parallelität mit sich. Viele Aufgaben werden heute noch rein *logisch-parallel* durchgeführt; in Zukunft wird vermehrt *physische Parallelität* eingesetzt werden (Multiprocessing). Parallele Verarbeitung ist aus verschiedenen Gründen interessant:

+ Es ist einfacher sich auf eine einzelne Aktivität zu konzentrieren. Das Formulieren eines Programmes für die einzelne Aktivität ist einfach.
- Das Festlegen welche Aktivitäten parallel ablaufen können ist nicht immer einfach. Das Debuggen von parallelen Prozessen ist komplex.
- Asynchrone Prozesse beeinflussen sich gegenseitig. Diese Beeinflussung erhöht die Komplexität des Systems.
- Es ist schwieriger die Korrektheit paralleler Programme zu beweisen.

5.1 Zugriff auf globale (gemeinsame) Daten

Wir betrachten ein System mit dem Adressen von Kunden erfasst werden. In diesem System will man jederzeit wissen wieviele Kunden bereits im System vorhanden sind. Die Erfassung geschieht mit einem Programm, viele Arbeitsplätze sind mit diesem Programm verbunden. Für jeden Arbeitsplatz ist ein Thread vorhanden. Jedes Mal wenn eine Adresseingabe abgeschlossen ist, muss eine globale Variable `ADDRESS_COUNT` um eins erhöht werden. Jeder Thread enthält folgenden Code auf Maschinencode Ebene:

```
LOAD  ADDRESS_COUNT
ADD   1
STORE ADDRESS_COUNT
```

Nehmen wir an `ADDRESS_COUNT` sei 7856. Der Thread A hat eben die Instruktion LOAD ausgeführt. Nun wird dem Thread A der Prozessor entzogen. Der Thread B führt nun die obigen 3 Instruktionen aus. `ADDRESS_COUNT` hat jetzt den Wert 7857. Noch weitere Threads können die obige Sequenz ausführen bis der Thread A wiederum den Prozessor erhält. `ADDRESS_COUNT` ist sicher $\geq$ 7857. Sobald der Thread A den Prozessor erhält, erhöht er seinem Accumulator um eins und speichert 7857 in `ADDRESS_COUNT` ab. Richtigerweise sollte `ADDRESS_COUNT` mindestens 7858 enthalten. Wegen dem uneingeschränkten Zugriff auf die Variable `ADDRESS_COUNT` ist ein Fehler entstanden.

$\Longrightarrow$ Dieses Problem kann behoben werden in dem man den Threads jeweils ein ausschliessliches Zugriffsrecht auf die Variable `ADDRESS_COUNT` zuweist. Während ein Thread die Variable erhöht, müssen alle anderen Threads, die das ebenfalls tun möchten, warten bis der Zugriff auf die Variable abgeschlossen ist. Dies nennt man: *wechselseitiger Ausschluss* oder *mutual exclusion*.

Ein wechselseitiger Ausschluss muss nur erzwungen werden falls man auf gemeinsame (shared) Daten zugreift. Alle übrigen Operationen können parallel (gleichzeitig) ausgeführt werden. Falls ein Thread auf gemeinsame Daten zugreift sagt

man: der Thread befindet sich in einem *kritischen Abschnitt* befinden. Das Erzwingen des wechselseitigen Ausschlusses ist eines der Schlüsselprobleme bei der Parallel-Programmierung.
Viele verschiedene Lösungen sind für das Problem des *wechselseitigen Ausschlusses* erdacht worden, reine Software Lösungen oder Hardware Lösungen.

Beim Eintritt in einem kritischen Abschnitt bekommt der Thread vom Kernel einen speziellen Status zugeordnet. Der Thread bekommt exklusives Zugriffsrecht auf gemeinsame Daten. Andere Threads die auch auf diese Daten zugreifen möchten müssen warten. $\Longrightarrow$ Kritische Abschnitte müssen so schnell wie möglich abgearbeitet werden, damit andere Threads nicht unnötig verzögert werden!

Beispiel für einen wechselseitigen Ausschluss:

```
public class MutalExclusion {
  private static int linesentered = 0;
  public class LineCounter extends Thread {
    public LineCounter() {
        start();
    }
    public void run()
      {
      while (true)
          {
            getnextlinefromterminal();
            entermutualexclusion();
              linesentered = linesentered+1;
            exitmutualexclusion();
            processtheline();
          }
      }
  }

  LineCounter mythread1 = new LineCounter();
  LineCounter mythread2 = new LineCounter();

  public static void main(String[] args){
      MutalExclusion mymutex = new MutalExclusion();
  }
}
```

Die Anweisungen `entermutualexclusion()` und `exitmutualexclusion()` die in diesem Beispiel angewandt werden, umschliessen die kritischen Abschnitte. Diese Anweisungen werden als *Stammfunktionen für wechselseitigen Ausschluss* (primitives for mutual exclusion) bezeichnet. Falls *mythread1* und *mythread2* gleichzeitig die Anweisung `entermutualexclusion()` ausführen, so kann nur einer der beiden weiterfahren, der andere muss warten. Wir nehmen an, dass der *Gewinner* zufällig ausgewählt wird.

5.2 Implementierung des wechselseititigen Ausschlusses

Der holländische Mathematiker Dekker hat einen Algorithmus vorgeschlagen, der von Dijkstra zu nachstehenden Programmen entwickelt wurde.

```
public class VersionOne {
    public static int threadnumber = 1;
    public class threadone extends Thread {
       public threadone() {
          start();
       }
       public void run(){
          while (true) {
                       while ( threadnumber == 2)
                       {}
                       criticalsectionone();
                       threadnumber = 2;
                       otherstuffone();
          }
       }
    }
    public class threadtwo extends Thread {
       public threadtwo() {
           start();
       }
       public void run(){
            while (true) {
                        while ( threadnumber == 1)
                        {}
                        criticalsectiontwo();
                        threadnumber = 1;
                        otherstufftwo();
            }
       }
    }

    threadone mythread1 = new threadone();
    threadtwo mythread2 = new threadtwo();

    public static void main(String[] args){
           VersionOne myv1 = new VersionOne();
    }
}
```

Der wechselseitige Ausschluss ist garantiert. Zuerst muss *threadone* den kritischen Abschnitt durchlaufen, falls *threadstwo* zu diesem Augenblick bereits bereit ist für den Eintritt in den kritischen Abschnitt, wird er verzögert. Nachdem *threadone* den kritischen Abschnitt verlassen hat, muss *threadtwo* in den Abschnitt eintreten, selbst dann, wenn *threadtwo* nicht bereit ist, andernfalls kann *threadone* nicht mehr eintreten. Im allgemeinen führt die obige Lösung zu einer Verlangsamung des Systems. In dieser Lösung wurde mit nur einer Synchronisationsvariabeln gearbeitet (= `threadnumber`).
In der nachstehenden zweiten Lösung werden zwei Variabeln zur Synchronisation herangezogen: `t1inside` und `t2inside`. `t1inside` ist *true* falls *threadone* sich im kritischen Abschnitt befindet; analoges gilt für *threadtwo* und `t2inside`.

```
public class VersionTwo {
    public static boolean t1inside = false;
    public static boolean t2inside = false;

    public class threadone extends Thread {
        public threadone() {
            start();
        }
        public void run(){
           while (true) {
                        while ( t2inside)
                        {}
                        t1inside = true;
                        criticalsectionone();
                        t1inside = false;
                        otherstuffone();
           }
        }
    }
    public class threadtwo extends Thread {
        public threadtwo() {
            start();
        }
        public void run(){
            while (true) {
                        while ( t1inside)
                        {}
                        t2inside = true;
                        criticalsectiontwo();
                        t2inside = false;
                        otherstufftwo();
            }
        }
    }

    threadone mythread1 = new threadone();
    threadtwo mythread2 = new threadtwo();

    public static void main(String[] args){
            VersionTwo myv2 = new VersionTwo();
    }
}
```

In dieser Version bleibt *threadone* aktiv blockiert solange `t2inside` *true* ist. Sobald *threadtwo* die Variable `t2inside` zu *false* setzt, kann *threadone* in den kritischen Abschnitt eintreten.
Die oben gezeigte Version hat aber einen schwerwiegenden Fehler: Der wechselseitige Ausschluss ist nicht garantiert! Da beide Threads parallel ablaufen, ist es möglich, dass *threadone* `t2inside` testet und diese den Zustand *false* hat. Bevor nun *threadone* `t1inside` *true* setzen kann, könnte *threadtwo* die Variable t1inside testen und diese ebenfalls *false* vorfinden. *threadone* und *threadtwo* würden nun beide in den kritischen Abschnitt eintreten. Das Problem bei dieser Version ist,

dass zwischen der Prüfung der Variablen und dem Setzen genug Zeit vorhanden ist, dass der andere Thread dazwischen kommen kann.

In der untenstehenden Lösung wird versucht, dass nur ein Thread über den *while-test* hinaus kommt.

```
public class VersionThree {
    public static boolean t1wantstoenter = false;
    public static boolean t2wantstoenter = false;

    public class threadone extends Thread {
        public threadone() {
            start();
        }
        public void run(){
           while (true) {
               t1wantstoenter = true;
               while ( t2wantstoenter)
               {}
               criticalsectionone();
               t1wantstoenter = false;
               otherstuffone();
           }
        }
    }
    public class threadtwo extends Thread {
        public threadtwo() {
            start();
        }
        public void run(){
            while (true) {
                t2wantstoenter = true;
                while ( t1wantstoenter)
                {}
                criticalsectiontwo();
                t2wantstoenter = false;
                otherstufftwo();
            }
        }
    }

    threadone mythread1 = new threadone();
    threadtwo mythread2 = new threadtwo();

    public static void main(String[] args){
            VersionThree myv3 = new VersionThree();
    }
}
```

Mit dieser Version wurde ein Problem gelöst, aber ein neues eingeführt. Falls jeder Thread sein Flag setzt bevor er testet, so wird jeder Thread das Flag des anderen bereits gesetzt finden und wird für immer in der while-Schleife hängen bleiben. Das obige Programm ist ein Beispiel für einen 2-Thread-Deadlock.

Das Problem mit der Version 3 ist, dass jeder Thread in seiner while Schleife hängen bleiben kann. Wir müssen einen Weg finden, um aus dieser Schleife auszubrechen. In der nachfolgenden Version 4 wird dies gemacht, indem jeder Thread seine Flags nur für eine kurze Zeitperiode setzt:

```
public class VersionFour {
    public static boolean t1wantstoenter = false;
    public static boolean t2wantstoenter = false;

    public class threadone extends Thread {
        public threadone() {
            start();
        }
        public void run(){
           while (true) {
               t1wantstoenter = true;
               while ( t2wantstoenter){
                  t1wantstoenter = false;
                  delay(random,fewcycles);
                  t1wantstoenter = true;
               }
               criticalsectionone();
               t1wantstoenter = false;
               otherstuffone();
           }
        }
    }
    public class threadtwo extends Thread {
        public threadtwo() {
            start();
        }
        public void run(){
            while (true) {
                t2wantstoenter = true;
                while ( t1wantstoenter){
                   t2wantstoenter = false;
                   delay(random,fewcycles);
                   t2wantstoenter = true;
                }
                criticalsectiontwo();
                t2wantstoenter = false;
                otherstufftwo();
            }
        }
    }

    threadone mythread1 = new threadone();
    threadtwo mythread2 = new threadtwo();

    public static void main(String[] args){
            VersionFour myv4 = new VersionFour();
    }
}
```

Der wechselseitige Ausschluss ist garantiert, ein Deadlock ist ausgeschlossen. Aber ein anderes Problem könnte auftreten: $\Longrightarrow$ *unbestimmtes Aufschieben!* Der eine der beiden Threads könnte periodisch aufgerufen werden. Der andere Thread könnte immer gerade nach dem Anlaufen des ersten Threads versuchen in den kritischen Abschnitt einzutreten und anschliessend verzögert werden bis der erste Thread erneut angelaufen ist. Natürlich ist ein solches Threadverhalten unwahrscheinlich, aber die Tatsache, dass dies so geschehen könnte, zeigt uns, dass Version 4 unbrauchbar ist.

Der Dekker-Algorithmus löst das Problem des wechselseitigen Ausschlusses auf elegante Art:

```
public class Dekker {
    public static int favoredthread = 1;
    public static boolean t1wantstoenter = false;
    public static boolean t2wantstoenter = false;

    public class threadone extends Thread {
        public threadone() {
            start();
        }
        public void run(){
           while (true) {
               t1wantstoenter = true;
               while ( t2wantstoenter){
                  if (favoredthread == 2){
                       t1wantstoenter = false;
                       while (favoredthread == 2)
                       {}
                       t1wantstoenter = true;
                   }
               }
               criticalsectionone();
               favoredthread=2;
               t1wantstoenter = false;
               otherstuffone();
           }
        }
    }
    public class threadtwo extends Thread {
        public threadtwo() {
            start();
        }
        public void run(){
            while (true) {
               t2wantstoenter = true;
               while ( t1wantstoenter){
                  if (favoredthread == 1){
                       t2wantstoenter = false;
                       while (favoredthread == 1)
                       {}
                       t2wantstoenter = true;
                   }
               }
```

```
                criticalsectiontwo();
                favoredthread=1;
                t2wantstoenter = false;
                otherstufftwo();
            }
        }
    }

    threadone mythread1 = new threadone();
    threadtwo mythread2 = new threadtwo();

    public static void main(String[] args){
            Dekker mydk = new Dekker();
    }
}
```

Funktionsweise des Programms:

threadone setzt sein Flag und zeigt damit an, dass er in den kritischen Bereich eintreten möchte. Dann kommt *threadone* zum while-Test wo untersucht wird, ob *threadtwo* ebenfalls eintreten will. Falls dies nicht der Fall ist, überspringt *threadone* den while-Schleifenkörper und tritt in den kritischen Abschnitt ein. Falls beide Threads gleichzeitig in die while Schleife eintreten, so kann nur der Favorit in den kritischen Abschnitt eintreten. Sobald ein Thread den kritischen Abschnitt verlässt, setzt er den anderen Thread zum Favoriten.

$\Longrightarrow$ Was geschieht wenn *threadone* der Prozessor entzogen wird gerade nach Abschluss der inneren while - Schleife?

Resultat: Allenfalls kann *threadtwo* erneut eintreten,
aber ein fehlerhaftes Verhalten des
Programms ist ausgeschlossen!

Nachdem dieser Algorithmus bereits seit 1965 bekannt war, hat G.L. Peterson im Jahre 1981 eine einfachere Lösung für dieses Problem gefunden:

```
public class Peterson{
    public static int favoredthread = 1;
    public static boolean t1wantstoenter = false;
    public static boolean t2wantstoenter = false;

    public class threadone extends Thread {
        public threadone() {
            start();
        }
        public void run(){
           while (true) {
              t1wantstoenter = true;
              favoredthread = 2;
              while (t2wantstoenter && (favoredthread == 2))
              {}
              criticalsectionone();
              t1wantstoenter = false;
              otherstuffone();
           }
        }
```

```
    }
    public class threadtwo extends Thread {
        public threadtwo() {
            start();
        }
        public void run(){
            while (true) {
              t2wantstoenter = true;
              favoredthread = 1;
              while (t1wantstoenter && (favoredthread == 1))
                {}
                criticalsectiontwo();
                t2wantstoenter = false;
                otherstufftwo();
            }
        }
    }

    threadone mythread1 = new threadone();
    threadtwo mythread2 = new threadtwo();

    public static void main(String[] args){
            Peterson mypeter = new Peterson();
    }
}
```

5.3 Hardware Lösung des wechselseitigen Ausschlusses

Der Dekker Algorithmus ist eine reine Softwarelösung. Auf der Hardware Ebene gibt es zwei Konzepte für den WSA:

1. wechselseitiger Ausschluss bei einem Einprozessorsystem: *Das Unterbrechungssystem wird ausgeschaltet.*
2. WSA bei Ein- und Multiprozessorsystemen: *die Instruktion* LOCK = (*testandset*).

Die Instruktion LOCK(L)
Falls in der Hardware eine Instruktion existiert, die eine Variable liest, den gelesenen Wert im Speicher oder einem Register abspeichert und die gelesene Variable zu einem verlangten Wert setzt; so kann mit dieser Instruktion ein wechselseitiger Ausschluss programmiert werden. Diese Instruktion, die ohne Unterbrechung abgearbeitet werden muss, wird in der Literatur als: *testandset* bezeichnet.

```
testandset (a,L)

- liest den Wert der booleschen Variabeln L
- kopiert diesen Wert in Variable a
- setzt die Variable L zu 'true'
```

Mit dieser Instruktion kann der wechselseitige Ausschluss wie folgt formuliert werden:

```
public class TestAndSet {
  private static boolean active = false;
```

```
  public class ThreadOne extends Thread {
      private boolean onecannotenter = false;
      public ThreadOne() {
        start();
      }
      public void run(){
        while (true){
            onecannotenter = true;
            while (onecannotenter) {
                testandset(onecannotenter, active);
            }
            criticalsectionone();
            active = false;
            otherstuffone();
        }
      }
  }
  public class ThreadTwo extends Thread {
      private boolean twocannotenter = false;
      public ThreadTwo() {
        start();
      }
      public void run(){
        while (true){
            twocannotenter = true;
            while (twocannotenter) {
                testandset(twocannotenter, active);
            }
            criticalsectiontwo();
            active = false;
            otherstufftwo();
        }
      }
  }

  ThreadOne mythread1 = new ThreadOne();
  ThreadTwo mythread2 = new ThreadTwo();

  public static void main(String[] args){
        TestAndSet mytas= new TestAndSet();
  }

}
```

ThreadOne stützt seine Entscheidung, ob er in den kritischen Abschnitt eintreten darf, auf die logische Variable `onecannotenter` ab. Solange die Variable `active` *true* ist, bleibt ThreadOne in der inneren while-Schleife. Der wechselseitige Ausschluss ist garantiert.

```
public class Lock {
    private boolean L = false;
    public void LOCK(){
        boolean lockset;
        lockset = true;
```

```
        while (lockset){
          testandset(lockset,L);
        }
    }
    public void UNLOCK(){
        L = false;
    }
}
```

Testandset bei INTEL Prozessoren:

Bei diesen Prozessoren muss verhindert werden, dass der aktive Prozessor während der Instruktion *testandset* unterbrochen wird. Eine solche Unterbrechung ist durch eine DMA-Anforderung oder durch einen anderen Prozessor, der den Bus verlangt, möglich. Damit eine Instruktion ohne Unterbrechung abgearbeitet wird, muss der Instruktionsprefix `lock` verwendet werden. Die *testandset* Instruktion auf diesen Prozessoren hat die mnemonische Bezeichnung `XCHG`.
Die Implementierung der inneren While-schleife könnte wie folgt aussehen:

```
         mov  al,onecannotenter
warte:   lock xchg al,active
         test al,al
         jnz  warte
```

5.4 Semaphore

Die beschriebenen Synchronisierungsmassnahmen sind in vielfacher Hinsicht schwerfällig und undurchsichtig.

- Wenn ein Thread sich in einem kritischen Abschnitt befindet, sollten nicht die anderen Threads ständig *um Erlaubnis zum Eintritt in den kritischen Abschnitt* fragen. Es sollte möglich sein, die wartenden Threads zu deaktivieren, bis sie durch die *Fertig-Meldung* des seinen kritischen Abschnitt verlassenden Threads aufgeweckt werden.
- Der wechselseitige Ausschluss sollte nicht im Anwenderprogramm kodiert sondern mit den Stammfunktionen (primitives) des Nucleus garantiert werden.

Für den wechselseitigen Ausschluss hat Dijkstra das Semaphor-Konzept entworfen. Ein Semaphor ist eine geschützte Variable auf die nur mit den Operationen P und V, und eine allfällige Semaphor-Initialisierung, zugegriffen werden kann. Man unterscheidet *Binäre Semaphore*, die nur die Werte 0, 1 annehmen können, und *Zähler Semaphore*, die nicht-negative Werte annehmen können.

Die P - Operation auf dem Semaphor S, geschrieben als P(S) *P = passeren* arbeitet wie folgt:

```
if  (S.value > 0)
    S.value = S.value-1;
else (wait(on S));
```

Die V - Operation auf dem Semaphor S, geschrieben als V(S) *V = vrijmaken* arbeitet wie folgt:

```
            if  (one or more activities are waiting on S)
                (let one of these activities proceed);
            else S.value = S.value +1;
```

Im allgemeinen verwendet man eine FIFO Schlange zur Verwaltung der wartenden Threads. Wie wir schon bei der Operation *testandset* gesehen haben, müssen auch die Operationen P und V ohne Threadwechsel (indivisable) abgearbeitet werden. Die Implementierung von P und V bildet einen kritischen Abschnitt.

Unter Verwendung der Operationen P und V kann das Problem des wechselseitigen Ausschlusses wie folgt gelöst werden:

```
import Concurrent.*;

public class SemaphoreExample {
  private static Semaphore myS = new Semaphore(1);
  public class ThreadOne extends Thread {
      public ThreadOne() {
        start();
      }
      public void run(){
        while (true){
            myS.P();
            criticalsectionone();
            myS.V();
            otherstuffone();
        }
      }
  }
  public class ThreadTwo extends Thread {
      public ThreadTwo() {
        start();
      }
      public void run(){
        while (true){
            myS.P();
            criticalsectiontwo();
            myS.V();
            otherstufftwo();
        }
      }
  }

  ThreadOne mythread1 = new ThreadOne();
  ThreadTwo mythread2 = new ThreadTwo();

  public static void main(String[] args){
     SemaphoreExample mysem= new SemaphoreExample();
  }
}
```

5.5 Semaphore in java 1.5

Ab java 1.5 sind Semaphore in `java.util.concurrent` enthalten. Diese Semaphore haben folgende Funktionalität:

Konstruktoren:

```
Semaphore(int permits) // permits = Wert auf den Semaphore initialisiert wird.
Semaphore(int permits, boolean fair) // fair = FIFO Queue der wartenden Threads
```

P-Operationen und deren Erweiterungen:

```
void acquire()
void acquire(int permits)
void acquireUninterruptibly()
void acquireUninterruptibly(int permits)
boolean tryAcquire()
boolean tryAcquire(int permits)
boolean tryAcquire(int permits, long timeout, TimeUnit unit)
boolean tryAcquire(long timeout, TimeUnit unit)
int drainPermits()
```

V-Operationen und deren Erweiterungen:

```
void release()
void release(int permits)
```

Infomations-Operationen

```
int availablePermits()
Collection<Thread> getQueuedThreads()
int getQueueLength()
boolean hasQueuedThreads()
boolean isFair()
```

Das vorherige Semaphore Beispiel Programm mit Dijksta Semaphoren für WSA kann unter Verwendung der JAVA 1.5 Semaphore wie folgt codiert werden:

```
import java.util.concurrent.*;

public class Semaphore15Example {
  private static Semaphore myS = new Semaphore(1, true);
  public class ThreadOne extends Thread {
      public ThreadOne() {
        start();
      }
      public void run(){
        while (true){
            myS.acquireUninterruptibly();
            try{
              criticalsectionone();
            }
            finally{
              myS.release();
            }
            otherstuffone();
        }
      }
```

```
  }
  public class ThreadTwo extends Thread {
      public ThreadTwo() {
        start();
      }
      public void run(){
        while (true){
            myS.acquireUninterruptibly();
            try{
              criticalsectiontwo();
            }
            finally{
              myS.release();
            }
            otherstufftwo();
        }
      }
  }

  ThreadOne mythread1 = new ThreadOne();
  ThreadTwo mythread2 = new ThreadTwo();

  public static void main(String[] args){
     Semaphore15Example mysem= new Semaphore15Example();
  }
}
```

6 Kooperierende Aktivitäten

In diesem Kapitel wird die Zusammenarbeit von Aktivitäten näher untersucht. Es werden verschiedene Lösungen der Synchronisation vorgestellt. Probleme die sich durch die Zusammenarbeit von Aktivitäten ergeben werden analysiert.

6.1 Synchronisation

Wenn eine asynchrone Aktivität eine E/A verlangt, wird diese blockiert bis zum Abschluss des Transfers. Nach dem Transfer wird diese wieder aufgeweckt, entweder durch das Betriebssystem oder eine andere Aktivität.

Allgemeiner: Oft möchte eine Aktiviät auf das Eintreffen eines Ereignisses aufmerksam gemacht werden.

Mit Semaphoren ist das Anzeigen von Ereignissen möglich!

```
import Concurrent.*;

public class BlockAndWakeup {
  private static Semaphore EventOfInterest = new Semaphore(0);
  public class ThreadOne extends Thread {
      public ThreadOne() {
        start();
      }
      public void run(){
            preliminarystuffone();
            EventOfInterest.P();
            otherstuffone();
      }
  }
  public class ThreadTwo extends Thread {
      private boolean twocannotenter = false;
      public ThreadTwo() {
        start();
      }
      public void run(){
            preliminarystufftwo();
            EventOfInterest.V();
            otherstufftwo();
      }
  }

    ThreadOne mythread1 = new ThreadOne();
    ThreadTwo mythread2 = new ThreadTwo();

    public static void main(String[] args){
        BlockAndWakeup baw = new BlockAndWakeup();
    }

}
```

Das obige Beispiel funktioniert auch wenn *ThreadTwo* die V Operation durchführt bevor *ThreadOne* zur P Operation kommt: durch die V Operation wird das Semaphor von 0 auf 1 erhöht; die nachfolgende P Operation erniedrigt das Semaphor

von 1 auf 0. In diesem Falle muss *ThreadOne* nicht warten.

Oft sind aber mehrere Aktivitäten an einem bestimmten Ereignis interessiert. Mit den P und V Operationen kann aber nur eine einzige Aktivität nach einem Ereignis aufgeweckt werden. In verschiedenen Nuclei sind Stammfunktionen vorhanden um beim Eintreffen eines Ereignisses alle wartenden Aktivitäten aufzuwecken. Diese Stammfunktionen nennen wir: WAIT und POST.

Die Wait Operation auf dem Ereignis S, WAIT(S), funktioniert wie folgt:

```
if  (S.value > 0)
  S.value = 0;
else {wait on S}
```

Die POST - Operation auf dem Ereignis S, POST(S), funktioniert wie folgt:

```
if   (one or more processes are waiting on S)
   let all of them proceed;
else S.value = 1;
```

6.2 Petri-Netze

Das Petri Netz ist ein sehr anschauliches Beschreibungs- und Modellierungsmittel für Systeme mit parallelen Aktivitäten.

Definition: Das Petri Netz ist ein gerichteter Graph N = (T,P,A,I). Dabei gilt:

T = (t1.....tn) eine Menge von Ereignissen
P = (p1.....pn) eine Menge von Bedingungen
A = (a1.....an) eine Menge gerichteter Kanten
I: P Anfangsmarkierung auf den Bedingungen, jede Bedingung kann eine Marke (token) enthalten, damit wird diese Bedingung wahr.

6.3 Produzent-Konsument Beziehung

In einem sequentiellen Programm, können bei einem Prozeduraufruf Daten von einem Programmteil zu einem anderen übergeben werden. Diese Datenübergabe aber findet in der selben Aktivität statt. Die Datenübergabe zwischen Aktivitäten ist etwas komplexer. Eine Aktivität der Daten generiert; nennen wir *Produzent*. Ein Aktivität der Daten braucht (benötigt); nennen wir *Konsument*. Zwei Aktivitäten können im einfachsten Falle über eine einzige Variable miteinander verkehren.

Beispiel:

> Der Produzent macht einige Berechnungen und schreibt sein Resultat in die gemeinsame Variable; der Konsument liest diese Variable und druckt diese aus. Wenn die Arbeitsgeschwindigkeiten von Produzent und Konsument genau übereinstimmen geht alles gut. Sobald aber nur ein kleiner Geschwindigkeitsunterschied vorhanden ist, so kann es vorkommen, dass ein Resultat nicht oder doppelt ausgedruckt wird. Wir verlangen, dass die beiden Aktivitäten richtig zusammenarbeiten, unabhängig von der Arbeitsgeschwindigkeit der beiden Aktivitäten. Das Erzwingen dieser Zusammenarbeit nennt man *Synchronisation*.

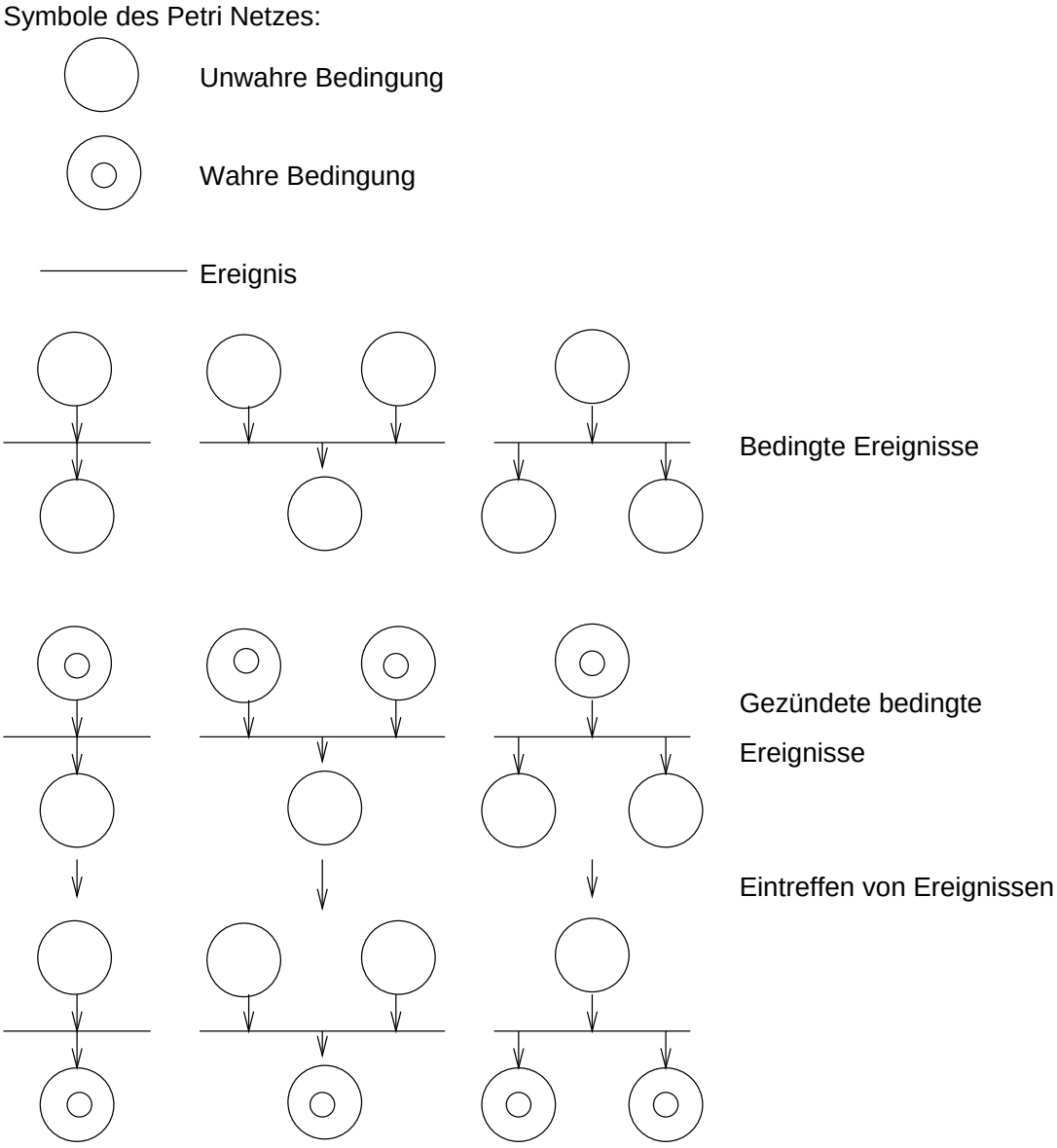


Abbildung 7: Petri Netz Symbole

```
import Concurrent.*;

public class ProducerConsumer {
  private static Semaphore NumberDeposited = new Semaphore(0);
  private static Semaphore ExclusiveAccess = new Semaphore(1);
  private static int numberbuffer;
  public class Consumer extends Thread {
      private int nextresult;
      public Consumer() {
        start();
      }
      public void run(){
        while (true){
            NumberDeposited.P();
            ExclusiveAccess.P();
            nextresult = numberbuffer;
            ExclusiveAccess.V();
            write (nextresult);
        }
      }
  }
  public class Producer extends Thread {
      private int nextresult;
      public Producer() {
        start();
      }
      public void run(){
        while (true){
           calculatenextresult();
           ExclusiveAccess.P();
           numberbuffer = nextresult;
           ExclusiveAccess.V();
           NumberDeposited.V();
        }
      }
  }

    Consumer mythread1 = new Consumer();
    Producer mythread2 = new Producer();

    public static void main(String[] args){
        ProducerConsumer pc = new ProducerConsumer();
    }
}
```

In diesem Beispiel werden Semaphore für zwei verschiedene Zwecke benützt:

ExclusiveAccess:	für das Sicherstellen des wechselseitigen Ausschlusses.
NumberDeposited:	für die Synchronisation der Aktivitäten.

6.4 Zählersemaphore

In verschiedenen Betriebssystemen sind Zählersemaphore unterstützt. Sie eignen sich vor allem für die Verwaltung eines Pools von gleichartigen Betriebsmitteln. Das

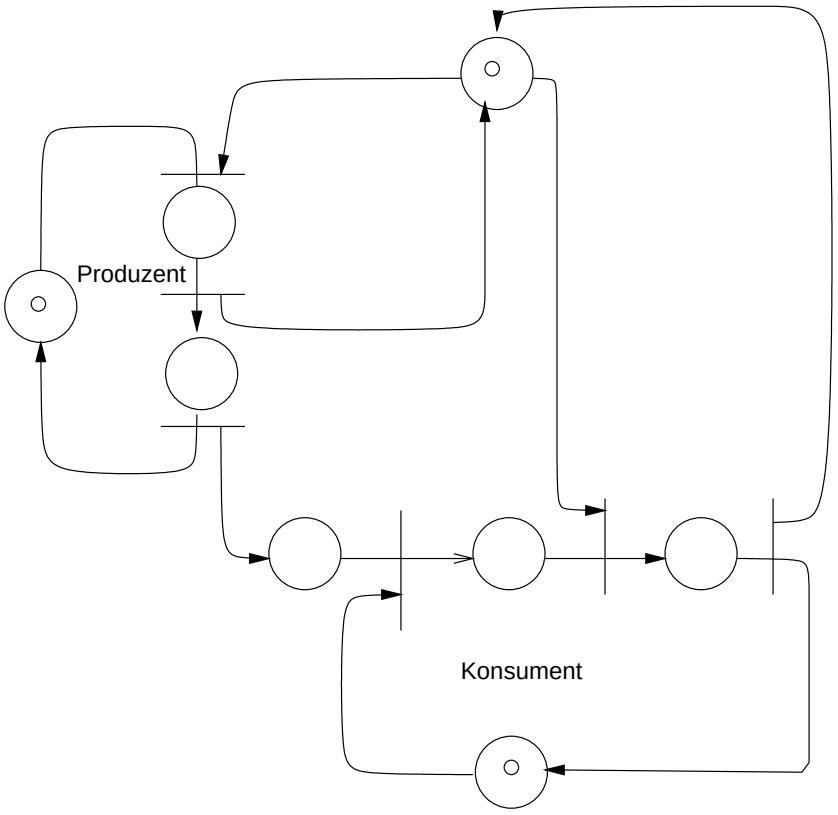


Abbildung 8: Petri Netz Beispiel

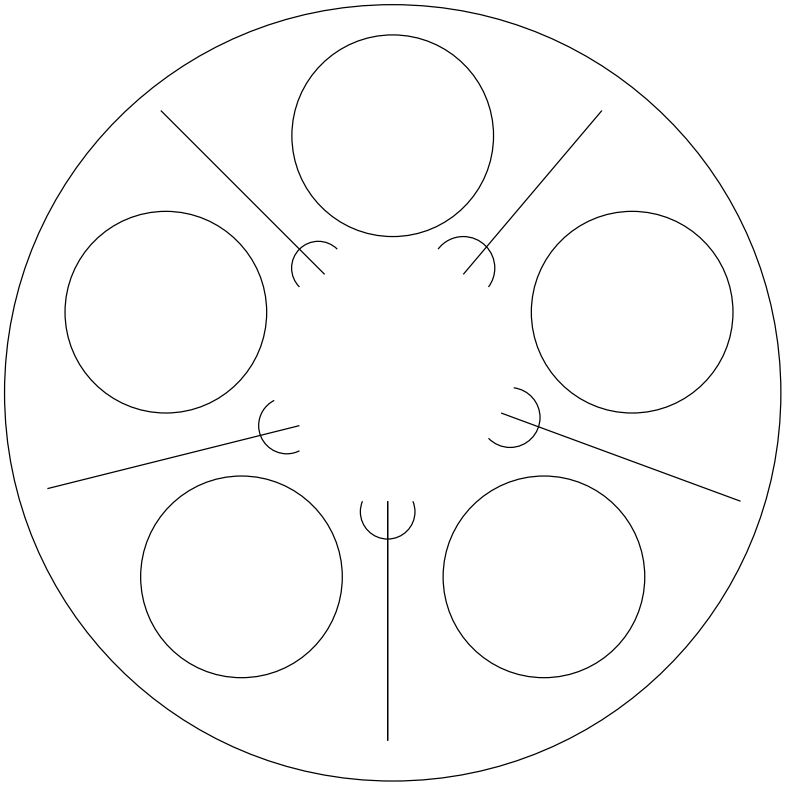


Abbildung 9: Die fünf Philosophen

Semaphor wird dabei initialisiert auf die Anzahl der vorhandenen Betriebsmittel. Jede P Operation erniedrigt den Wert des Semaphors um 1; jede V Operation erhöht den Wert des Semaphors um 1. Solange der Semaphorwert $>$ 0 ist, sind noch Betriebsmittel vorhanden, und die Aktivitäten müssen nicht warten.

6.5 Private Semaphore

Wir betrachten hier das von Dijkstra erstmals gelöste Problem der 5 Philosophen, die an einem runden Tisch sitzen. Vor jedem Philosophen steht ein Teller Spaghetti. Damit ein Philosoph essen kann, braucht er zwei Gabeln. Somit kann ein Philosoph nur dann essen, wenn keiner der Nachbarn isst. Die Philosophen durchlaufen zyklisch die drei Zustände: *denken*, *hungrig werden* und *essen*. Es soll ein Algorithmus gefunden werden für den Philosophen i $(0 <= i <= 4)$.
Die einfachste Form einer Lösung wäre die Einführung von Semaphoren für jede Gabel. Jeder Philosoph i hätte dann ein Semaphor links fork[i] und ein Semaphor rechts fork[i+1]. Alle Semaphore sind zu Beginn auf 1 initialisiert.

```
import Concurrent.*;

public class Philosopher1 {
  private static Semaphore[] fork =
          {new Semaphore(1),new Semaphore(1),
           new Semaphore(1),
           new Semaphore(1),new Semaphore(1)};
```

```
  public class Philosopher extends Thread {
      private int i;
      public Philosopher( int index) {
        i = index;
        start();
      }
      public void run(){
        while (true){
          think(i);
          get_hungry(i);
          fork[i].P();
          fork[(i+1) %  5].P();
          eat(i);
          fork[(i+1) % 5].V();
          fork[i].V();
         }
      }
      private void eat(int i){
      }
      private void get_hungry(int i){
      }
      private void think(int i){
      }
  }

  Philosopher [] myphilo = {new Philosopher(0),
                            new Philosopher(1),
                            new Philosopher(2),
                            new Philosopher(3),
                            new Philosopher(4)};


  public static void main(String[] args){
     Philosopher1 myph= new Philosopher1();
  }
}
```

Diese Lösung kann aber zu einem Deadlock führen. Eine brauchbare Lösung ist im nachstehenden erklärt:

Wir führen für jeden Philosophen *i* eine Statusvariable *status* ein, die die Werte: *denken*, *hungrig-werden* und *essen* annehmen kann. Damit der Übergang *hungrig-werden* → *essen* vollzogen werden kann, muss gelten:

status[(i+1) % 5] <> *essen* und
status[(i-1) % 5] <> *essen*

Neben dem Statusvektor *status* benützt man zwei Arten von Semaphoren zur Lösung der Aufgabe:

- 5 private Semaphore `private[i]`
- ein Verwaltungssemaphor `mutex`

```
import Concurrent.*;
```

```
public class Philosopher2 {
  private static Semaphore[] privat =
            {new Semaphore(0),new Semaphore(0),
             new Semaphore(0),
             new Semaphore(0),new Semaphore(0)};
  private static Semaphore mutex = new Semaphore(1);
  final static int denken  = 0;
  final static int hungrig_werden = 1;
  final static int essen = 2;
  private static int[] status = { denken, denken, denken,
                                  denken, denken};
  public class Philosopher extends Thread {
      private int i;
      public Philosopher( int index) {
        i = index;
        start();
      }

      private void eattest(int i){
            if (status[i] == hungrig_werden){
                  if ((status[(i+4) % 5] != essen) &&
                     (status[(i+1) % 5] != essen)){
                         privat[i].V();
                         status[i] = essen;
                   }
            }
      }

      public void run(){
        while (true){
              think(i);
              get_hungry(i);
              mutex.P();
              status[i]= hungrig_werden;
              eattest(i);
              mutex.V();
              privat[i].P();
              eat(i);
              mutex.P();
              status[i] = denken;
              eattest((i+1) % 5);
              eattest((i+4) % 5);
              mutex.V();

        }
      }
      private void eat(int i){
      }
      private void get_hungry(int i){
      }
      private void think(int i){
      }
  }
```

```
  Philosopher [] myphilo = {new Philosopher(0),
                            new Philosopher(1),
                            new Philosopher(2),
                            new Philosopher(3),
                            new Philosopher(4)};


  public static void main(String[] args){
     Philosopher2 myph= new Philosopher2();
  }
}
```

Auf einem privaten Semaphor darf nur der 'Eigentümer' P - Operationen durchführen. Alle Aktivitäten aber dürfen V Operationen darauf ausführen.

Private Semaphore werden vor allem für die Zuteilung von Betriebsmitteln verwendet, die nur in beschränkter Anzahl vorhanden sind.

Hausaufgabe: Ein bestimmtes Betriebsmittel ist nur in n-facher Ausführung vorhanden. Es können sich alle k Aktivitäten um diese Betriebsmittel bewerben. Kodieren Sie die beiden Methoden 'int anfordern(i)' und 'boolean freigeben(b,i)' wobei i = Aktivitäts-Index b = Betriebsmittel-Index

6.6 Barriers

Auf einer shared Memory Plattform braucht man oft *barriers*. *barriers* sind eine Erweiterung der Semaphore. Kein Thread darf die *barrier* überschreiten bis alle Threads dort angekommen sind. Im untenstehenden Beispiel gibt es zwei Konstruktoren und auch zwei *gate()* Methoden für den ein- oder mehrdimensionalen Fall. Die *gate()* Methode lässt den Thread an der Barrier warten.

```
package Concurrent;


// a Barrier for m*n threads: all must arrive before any are released
public class Barrier  implements Runnable {

   private int m = -1, n = -1;
   private Semaphore arrive = null;
   private Semaphore[][] release = null;
   private Thread thread = null;

   public Barrier(int m, int n) {
      System.out.println("Barrier for m=" + m + " n=" + n + " threads");
      this.m = m;
      this.n = n;
      arrive = new Semaphore(0);
      release = new Semaphore[m][n];
      for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)
         release[i][j] = new Semaphore(0);
```

```
      thread = new Thread(this);
      thread.setDaemon(true);
      thread.start();
   }

   public Barrier(int n) { this(1, n); }


   public void gate(int i, int j) {
      arrive.V();
      release[i][j].P();
   }

   public void gate(int j) { this.gate(0, j); }


   public void run() {
      while (true) {
         for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)
            arrive.P();
// The Second semaphore must be an array to avoid race conditions
// due to arbitrary context switches and relative CPU speeds.
         for (int i = 0; i < m; i++) for (int j = 0; j < n; j++)
            release[i][j].V();
      }
   }
}
```

6.7 Monitore

Das Programmieren des wechselseitigen Ausschlusses mit dem Dekker Algorithmus aber auch mit Dijkstra's Semaphoren ist nicht einfach. Die fehlerhafte Programmierung eines wechselseitigen Ausschlusses kann aber mehrere Aktivitäten beeinflussen.
Um die Schwierigkeiten bei der Bedienung von Semaphoren zu vermeiden, wurde von Brinch Hansen und Hoare ein neues Sprachkonstrukt entwickelt: der MONITOR. Mit dem Monitor soll eine korrekte Abwicklung der Semaphor-Operationen erzwungen werden, ohne dass die Operationen P und V explizite programmiert werden. Die P und V Aufrufe werden durch einen Compiler erzeugt. Folgende Echtzeitsprachen unterstützen das Monitor-Konzept:

Concurrent Pascal: Brinch Hansen 1975

Portal: Landis & Gyr 1978

Modula 2: N. Wirth, ETH Zürich, 1975

JAVA: Sun Microsystems

Ein Monitor ist ein Programmteil, der Daten und Methoden/Prozeduren enthält. Damit eine Aktivität in den Monitor eindringen kann, muss diese ein *monitor entry* aufrufen. Viele Aktivitäten können Monitor entry's desselben Monitors aufrufen; der wechselseitige Aussschluss bleibt garantiert. Nur eine Aktivität ist gleichzeitig im Innern des Monitors. Aktivitäten, die in einen Monitor eintreten wollen der bereits

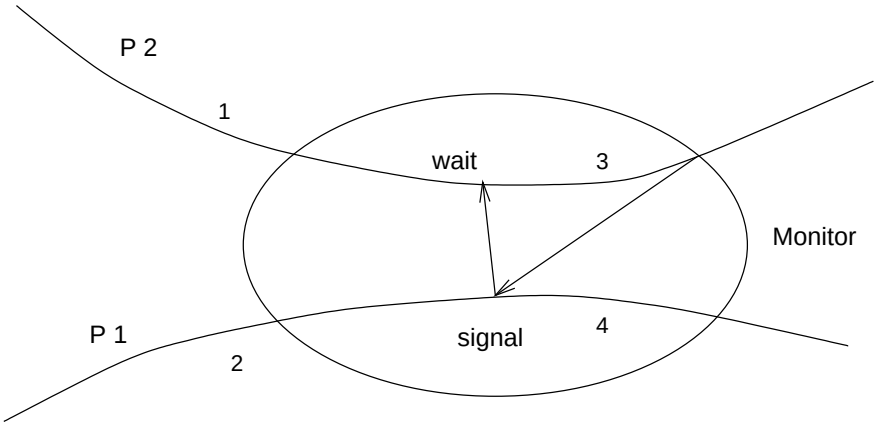


Abbildung 10: Funktionsweise des Monitors

besetzt ist, müssen warten. Die Prozeduren/Methoden eines Monitors können auf globale Daten zugreifen. Die lokalen Daten des Monitors sind aber von aussen nicht zugänglich (verstecken von Information).

Im Innern eines Monitors ist es aber unter Umständen nötig, dass eine Aktiviät wartet. Eine solche Aktivität wird, durch den Monitor, aus dem Monitor ausgelagert. Auf diese Weise bleibt der Monitor nicht blockiert und eine anderere Aktivität kann in den Monitor eintreten. Falls eine anderere Aktivität den Wartenden befreit, so kann diese, sobald der Monitor frei ist, diesen wieder betreten.

Für die Synchronisation im Innern des Monitors müssen folgende Bedingungen gelten:

- Während dem Befreien darf keine andere Aktivität dazwischen treten.
- Um zu verhindern, dass die 'Befreiungsbedingung' durch die befreiende Aktivität wieder geändert wird, verlangt man, dass die Steuerung sofort an die befreite Aktivität übergeht. Das bedeutet, dass die wartende Aktivität Priorität gegenüber der neu eingetretenen hat.

Zur Steuerung werden 3 Semaphore verwendet:

- mutex
- urgent
- condition

Im Innern des Monitors können die Methoden:

- condition.Wait() = condition.P()
- condition.Signal() = condition.V() (hat nur eine Auswirkung falls eine Aktivität wartet)

verwendet werden.

Die drei Semaphore werden wie folgt initialisiert:

- mutex = 1
- urgent = condition = 0

Das Semaphor urgent ist ein Hilfssemaphor. Sobald der Produzent dem Konsumenten etwas signalisiert (signal, V) so wird der Produzent auf diesem Semaphor warten.
Implementierung des Monitors mit 3 Semaphoren:

Es ist interessant festzustellen, dass das Verlassen des Monitors über den Ausgang *signal(condition)* das Semaphor mutex nicht berührt. Die aufgeweckte Aktivität kann danach den Monitor betreten. Beim Verlassen weckt er den, auf der Queue *urgent* wartenden auf. Dieser setzt beim Verlassen des Monitors das Mutex zurück

Vereinfachte Version des Monitors

Wir stellen fest, dass falls die Operation *signal(condition)* erst zu dem Zeitpunkt gemacht wird, wo die Aktivität den Monitor verlässt, ist das Semaphor urgent überflüssig. Mit der Elimination dieses Semaphors steigern wir das Echt-Zeitverhalten des Monitors.

6.7.1 Monitor Beispiele

a) Einfache Betriebsmittel-Zuteilung

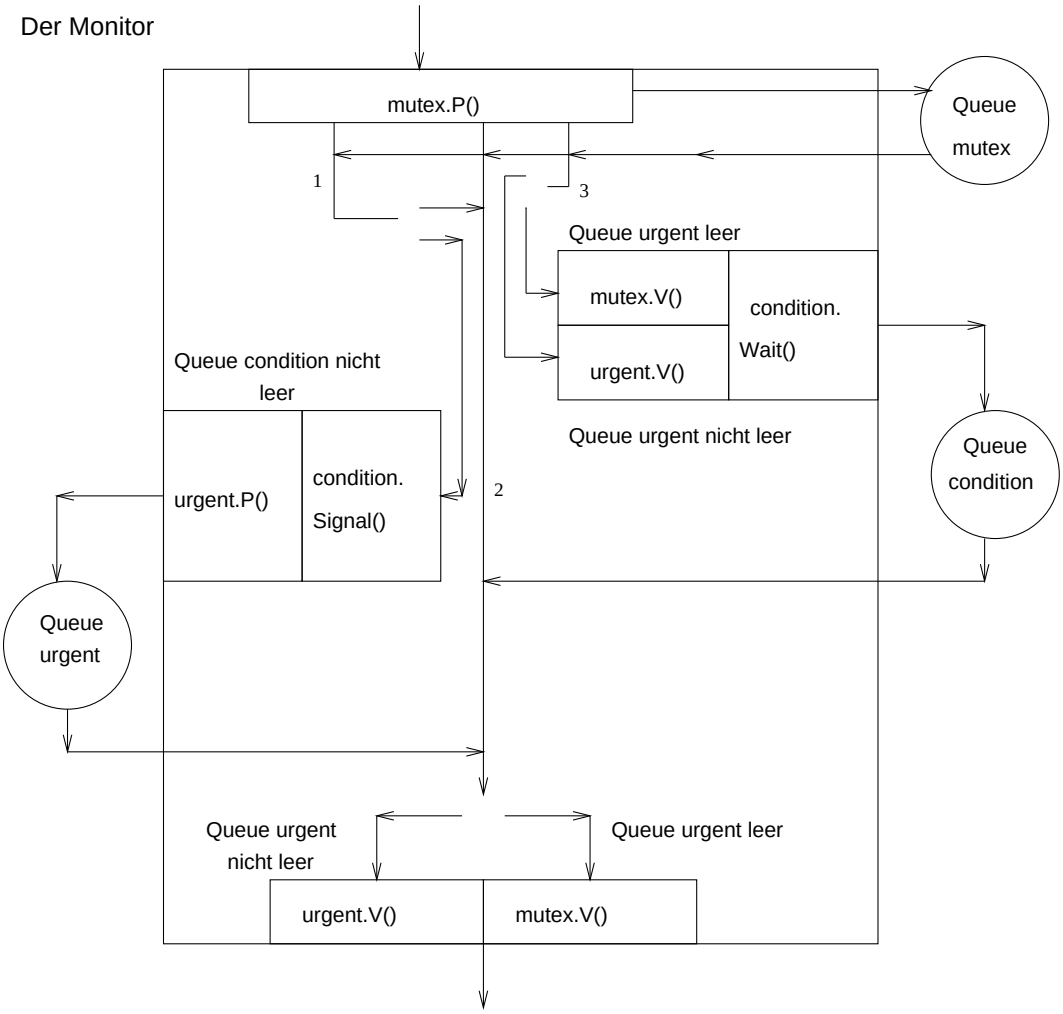
```
import Concurrent.*;

public class ResourceAllocator extends monitor {
    private boolean  resourceinuse = false;
    private Condition  resourceisfree = new Condition();
    public ResourceAllocator(){
    }
    public void getresource() {// monitor entry, takes the monitor lock
              if (resourceinuse)
                   resourceisfree.Wait();
              resourceinuse= true;
    }
    public void returnresource() {// monitor entry, takes the monitor lock
              resourceinuse= false;
              resourceisfree.Signal();
    }
}
```

Das Schöne an diesem Monitor ist, dass dieser benützt werden kann ohne die P und V Operationen für wechselseitigen Ausschluss. Statt dem Monitor, hätten wir geradesogut P und V Operationen verwenden können.

b) Der Ringpuffer

Viele Systeme bestehen aus einer Anzahl asynchroner Aktivitäten. Oft ist es nötig, dass einzelne Aktivitäten miteinander Daten austauschen. Ein Ringpuffer kann verwendet werden um Daten zwischen einem Produzenten und einem Konsumenten



1: Prozess sendet signal

2: Prozess benützt Monitor

3: Prozess erwartet signal

Abbildung 11: Der vollständige Monitor

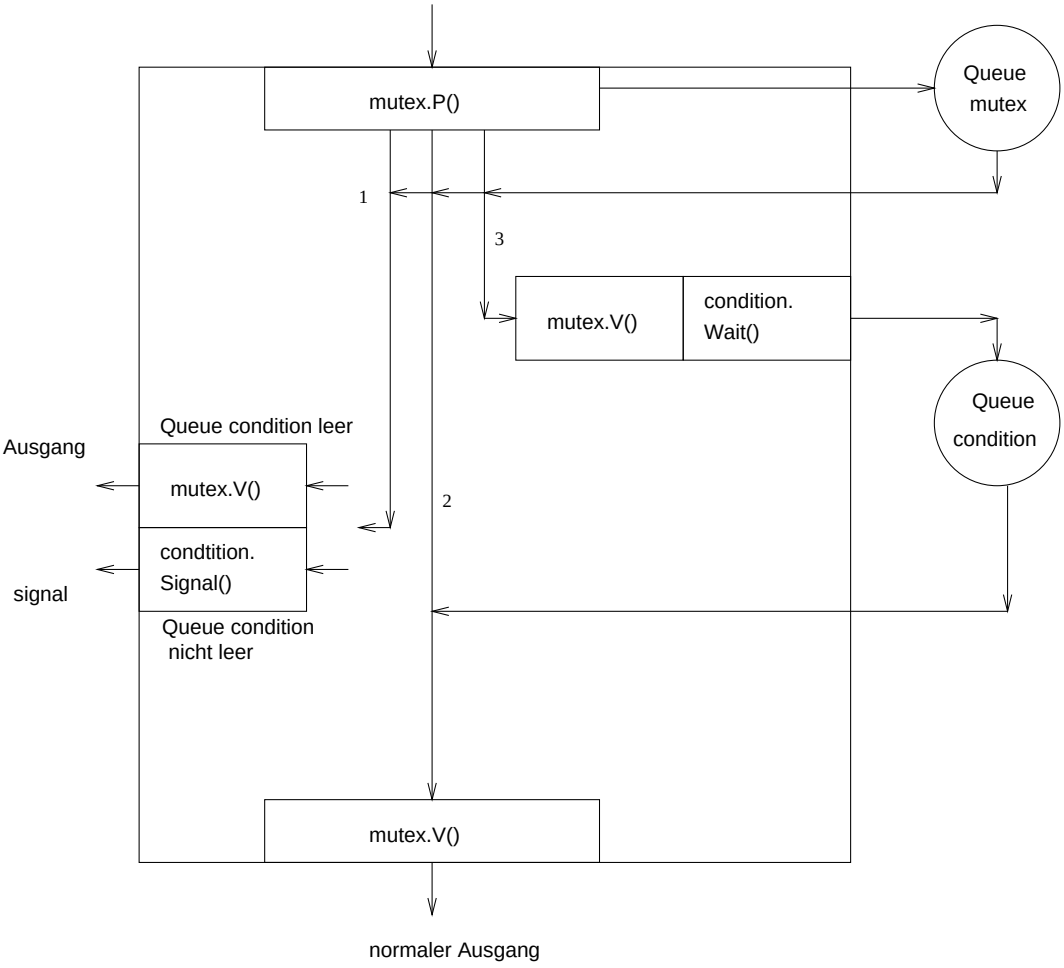


Abbildung 12: Vereinfachte Version des Monitors

auszutauschen. Oft will der Produzent Daten abgeben bevor der Konsument bereit ist diese anzunehmen; oder der Konsument möchte Daten entgegennehmen, bevor diese vom Produzenten erzeugt wurden. Die Synchronisierung zwischen Produzent und Konsument ist also sehr wichtig. Ein Ringpuffer hat eine feste Grösse und wird allgemein durch einen Vektor (array) fester Grösse gebildet. Das Element, das auf dasjenige mit dem grössten Index folgt, ist das erste Element (wrap around). Durch die feste Grösse des Ringpuffers, ist es möglich, dass der Produzent den Puffer voll vorfindet, und warten muss, bis der Konsument Elemente entnommen hat.

```
import Concurrent.*;

public class RingBufferMonitor extends monitor {
    private int slots;
    private Object [] ringbuffer;
    private int   slotsinuse = 0;
    private int   nextslottofill = 0;
    private int   nextslottoempty = 0;
    private Condition  ringbufferhasdata = new Condition();
    private Condition  ringbufferhasspace = new Condition();
    public RingBufferMonitor(int i){
        ringbuffer = new Object[i];
        slots = i;
    }
    public void fillaslot(Object slotdata) {// monitor entry, takes the  lock
          if (slotsinuse == slots)
                ringbufferhasspace.Wait();
          ringbuffer[nextslottofill] = slotdata;
          slotsinuse= slotsinuse+1;
          nextslottofill= (nextslottofill + 1) % slots;
          ringbufferhasdata.Signal();
    }
    public Object emptyaslot (){// monitor entry, takes the monitor lock
          Object slotdata;
          if (slotsinuse == 0)
                ringbufferhasdata.Wait();
          slotdata = ringbuffer[nextslottoempty];
          slotsinuse= slotsinuse-1;
          nextslottoempty=(nextslottoempty+1) % slots;
          ringbufferhasspace.Signal();
          return slotdata;
    }
}
```

Diese Art der Pufferung wird in Betriebssystemen oft zur Steuerung von 'Spooling' benützt. Der Ringpuffer kann anstelle eines speicherresidenten Vektors auch ein Bereich auf dem Massenspeicher sein. Dem Produzenten sagt man den oft auch 'Spooler' der Konsument wird oft 'Despooler' genannt.

c) readers and writers

In Rechnersystemen kommt es häufig vor, dass Aktivitäten vorhanden sind (genannt: readers) die gewisse Daten nur lesen und andere Aktivitäten (genannt:writers) die auf diese Datenbestände schreiben. Ein Beispiel dafür sind Flugreservartions-

systeme: viele Anfragen werden oft gemacht, bis schliesslich eine Buchung stattfindet. Da die 'readers' die Information der Datenbasis nicht ändern, können viele gleichzeitig auf die Datenbasis zugreifen. Da aber der 'writer' die Datenbasis verändert, muss er exklusiven Zugriff auf die Datenbasis erhalten. Falls ein 'writer' aktiv ist, so darf weder ein anderer 'reader' noch ein anderer 'writer' aktiv sein. Dieser exklusive Zugriff muss aber nur auf Satzebene sichergestellt werden. Ein exklusiver Zugriff auf die ganze Datenbasis ist nicht nötig. Die vorliegende Lösung stammt von Hoare und Gorman. Sie ist anwendbar für einen Satz oder eine ganze Datenbasis.

```
import Concurrent.*;

public class  ReadersAndWriters extends monitor {
    private int  readers = 0;
    private boolean  someoneiswriting = false;
    private Condition readingallowed = new Condition();
    private Condition writingallowed = new Condition();

    public void beginreading() {// monitor entry, takes the monitor lock
              if ((someoneiswriting) || queue(writingallowed))
                 readingallowed.Wait();
              readers= readers+1;
              readingallowed.Signal();
    }
    public void finishreading() {// monitor entry, takes the monitor lock
              readers= readers-1;
              if (readers == 0)  writingallowed.Signal();
    }
    public void beginwriting()  {// monitor entry, takes the monitor lock
              if ((readers > 0) || someoneiswriting)
                    writingallowed.Wait();
              someoneiswriting= true;
    }
    public void finishwriting() {// monitor entry, takes the monitor lock
              someoneiswriting= false;
              if (queue(readingallowed))
                   readingallowed.Signal();
              else
                   writingallowed.Signal();
    }
    }
```

Zu Beginn des Lesens ruft der *reader* die Methode `beginreading`; nach dem Lesen ruft er die Methode `finishreading`. Solange keine Aktivität die Schreiben möchte wartet, kann jeder *reader* mit Lesen beginnen. Damit ein *writer* nicht unnötig lange aufgeschoben wird, darf ein *reader* nicht mehr beginnen sobald ein *writer* wartet. Sobald alle aktiven *readers* die Methode `finishreading` aufgerufen haben wird der wartende *writer* gestartet. Sobald der *writer* die Methode `finishwriting` aufruft, werden mit Signal die *readers* wieder aufgeweckt; wobei jeder *reader* seinerseits wieder einen *reader* weckt.

Dass ein wartender *writer* Priorität hat über die wartenden *reader* ist zwingend, damit die *writer* nicht endlos aufgeschoben werden können.

6.8 Wechselseitiger Ausschluss und Synchronisation in Anwenderprogrammen

Die Verwendung von Monitoren ist etwas schwerfällig und führt im allgemeinen auch zu unnötiger Sequentialisierung. Anderseits aber ist es in der Praxis ausserordentlich häufig, dass innerhalb eines kritischen Abschnittes gewartet werden muss. Der Monitor ermöglicht eine robuste und einfache Lösung für dieses Problem. Die Lesbarkeit von Programmen, bei denen sowohl für den Wechselseitigen Ausschluss wie auch für die Synchronisation die Semaphor-Operationen P und V verwendet werden, lässt zu wünschen übrig. In der heutigen Anwender-Programmierung werden aus diesem Grunde, zur Verbesserung der Lesbarkeit wie auch für den einfachen Gebrauch und robuste Programme folgende neue Datentypen und Stammfunktionen verwendet:

- Wechselseitiger Ausschluss:
  - Klasse *mutex*
  - Methoden
    - void lock();
    - void unlock ();
- Synchronisation:
  - Klasse *condition* = binäres Semaphor mit speziellem Synchronisationsverhalten
  - Methoden:
    - void wait(mutex m);
    - void signal();
    - void broadcast();

6.8.1 Das Mutex

Das Mutex wird verwendet für den wechselseitigen Ausschluss. Der Konstruktor von *mutex* initialisiert das binäre Semaphor automatisch auf 1. Mit *lock* versucht man in den kritischen Abschnitt einzutreten, mit *unlock* verlässt man diesen. Wenn eine Aktivität sich im kritischen Abschnitt befindet, so sagt man die Aktivität *hält* das Mutex.

6.8.2 Die Condition Variable

Eine Condition Variable ist immer mit einem Mutex verbunden; es ist aber auch möglich, dass mehrere Condition Variabeln zu einem Mutex gehören.
Die Wait Operation gibt das Mutex frei und blockiert die Aktivität. Die Wait Operation kann nicht unterbrochen werden. Die Signal Operation hat nur eine Auswirkung, falls mindestens ein Aktivität blockiert ist: in diesem Falle wird mindestens eine Aktivität aufgeweckt. Die Broadcast Operation funktioniert wie Signal ausser, dass alle blockierten Aktivitäten aufgeweckt werden. Sobald eine auf Wait blockierte Aktivität aufgeweckt wird, so führt er erneut die Lock Operation auf dem Mutex aus und fährt in seiner Arbeit fort. Es ist denkbar, dass das Mutex zu diesem Zeitpunkt nicht frei ist; in diesem Falle wird die Aktivität auf dem Mutex blockiert. Die folgenden Programm-Fragmente sollen die Verwendung von Mutex und Condition Variables erläutern:

Herausnehmen aus einer Liste

```
private condition nonempty = new condition();
mutex m = new mutex();

m.lock();
WHILE (head == null)  nonempty.wait(m);
topElement = head;
head = head.next;  /* head->next */
m.unlock();
```

Einfügen in eine Liste:

```
m.lock();
newElement.next= head; /* newElement->next */
head = newElement;
nonempty.signal();
m.unlock();
```

6.9 Mutex und Condition Variable in jsr166

Ab der JAVA Version 1.5 sind sowohl Mutexe wie auch Condition Variables in *java.util.concurrent.locks* enthalten. Im einfachsten Falle wird die Klasse *ReentrantLock* verwendet. Diese Klasse erlaubt einem Thread mehrfach eine *lock* Operation auszuführen, ohne zu blockieren, falls er den *Lock* schon besitzt. Die Klasse *ReentrantLock* enhält folgende Methoden:

Konstruktoren:

```
ReentrantLock()
ReentrantLock(boolean fair)
```

lock/unlock Methoden und deren Erweiterung

```
int getHoldCount()
Thread getOwner()
Collection<Thread> getQueuedThreads()
boolean hasQueuedThread(Thread thread)
boolean hasQueuedThreads()
boolean isFair()
boolean isHeldByCurrentThread()
boolean isLocked()
void lock()
void lockInterruptibly()
boolean tryLock()
boolean tryLock(long timeout, TimeUnit unit)
void unlock()
Condition newCondition()
Collection<Thread> getWaitingThreads(Condition condition)
int getWaitQueueLength(Condition condition)
boolean hasWaiters(Condition condition)
```

Methoden auf der Condition

```
void await()
boolean await(long time, TimeUnit unit)
```

```
long awaitNanos(long nanosTimeout)
void awaitUninterruptibly()
boolean awaitUntil(Date deadline)
void signal()
void signalAll()
```

Beispielprogramm

```
import java.util.concurrent.locks.*;

 class BoundedBuffer {
   final Lock mutex = new ReentrantLock();
   final Condition notFull  = mutex.newCondition();
   final Condition notEmpty = mutex.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     mutex.lock();
     try {
       while (count == items.length)
         notFull.await();
       items[putptr] = x;
       if (++putptr == items.length) putptr = 0;
       ++count;
       notEmpty.signal();
     } finally {
       mutex.unlock();
     }
   }

   public Object take() throws InterruptedException {
     mutex.lock();
     try {
       while (count == 0)
         notEmpty.await();
       Object x = items[takeptr];
       if (++takeptr == items.length) takeptr = 0;
       --count;
       notFull.signal();
       return x;
     } finally {
       mutex.unlock();
     }
   }
 }
```

6.10 Ereigniszähler (Event Counters)

Die bisherigen Lösungen für das Produzenten-Konsumenten Problem enthielten kritische Abschnitte, die mit Monitoren oder Semaphoren geschützt werden mussten. Mit Ereignis-Zählern (Reed und Kanodia, 1979) ist ein wechselseitiger Ausschluss nicht nötig.

Drei Operationen sind definiert auf dem Ereignis-Zähler e

1. e.read() liest den aktuellen Wert des Ereignis-Zählers e

2. e.advance() erhöht den Wert von e um eins (kann nicht unterbrochen werden).

3. e.await(v) wartet bis e den Wert v oder mehr hat.

Die hier gezeigte Produzent-Konsument Lösung verwendet die Operation Read nicht, andere Sychronisationsprobleme machen diese Operation dennoch notwendig.

```
import Concurrent.*;

public class EventCounterExample {
  private eventcounter inputcounter = new eventcounter();
  private eventcounter outputcounter = new eventcounter();
  private int slots = 0;
  private Object [] buffer;
  private int inputsequence = 0;
  private int  outputsequence = 0;

  public EventCounterExample(int i){
      buffer = new Object[i];
      slots = i;
  }

  public class producer extends Thread {
      public producer(){
        start();
      }
      public void run(){
        Object  item;

        while (true){
           item = produce();
           inputsequence = inputsequence+1;
           outputcounter.await(inputsequence-slots);
           buffer[(inputsequence-1) % slots]= item;
           inputcounter.advance();
        }
      }
  }

  public class consumer extends Thread{
     public consumer(){
        start();
     }
      public void run(){
        Object    item;

        while (true){
           outputsequence = outputsequence+1;
           inputcounter.await(outputsequence);
           item= buffer[(outputsequence-1 % slots)];
           outputcounter.advance();
           consume(item);
        }
```

```
      }
  }

  producer my_producer = new producer();
  consumer my_consumer = new consumer();

  public static void main(String[] args ){
    EventCounterExample evct = new EventCounterExample(10);
  }
}
```

Die Ereignis-Zähler werden nur erhöht, nie erniedrigt. Sie beginnen immer bei 0. In unserem Beispiel werden zwei Ereignis-Zähler verwendet. *inputcounter* zählt die totale Anzahl der produzierten Elemente seit dem Start. *outputcounter* zählt die Anzahl der konsumierten Elemente seit dem Start. Aus diesem Grunde muss outputcounter <= inputcounter sein. Sobald der Produzent ein neues Element erzeugt hat, prüft er ob noch Platz im Puffer vorhanden ist, indem er den Await - Systemaufruf benützt. Zu Beginn ist outputcounter = 0 und (inputsequence-slots) negativ damit wird der Produzent nicht blockiert. Falls es dem Produzenten gelingt (slots+1) - Elemente zu erzeugen bevor der Konsument startet, so muss der Produzent warten bis outputcounter = 1 ist. Dies ist aber nur möglich durch das Konsumieren eines Elementes durch den Konsumenten. Die Logik des Konsumenten ist noch einfacher. Bevor das n-te Element konsumiert werden kann, muss mit Await(inputcounter,n) auf das Erzeugen des n-ten Elementes gewartet werden.

6.11 Rendezvous

Als Rendezvous bezeichnet man eine Sychronisationstechnik bei der sich ein Sender und ein Empfänger gleichzeitig treffen um ein Objekt zu übergeben.

```
import java.util.concurrent.*;


// like bounded buffer multiple producers and multiple
// consumers with one buffer slot except producer must
// wait for the consumer to extract the message

public final class Rendezvous {

/*
 * This class implements a rendezvous between the sender and reveiver:
 * they meet at the same time inside an object instantiated from this
 * class and a message object is passed from sender to receiver.  This
 * version works inside a single JVM, that is, it requires shared memory.
 * To send an object from one JVM to another requires object serialization.
 *
 */

   private Object theMessage = null;
   private final Object sending = new Object();
   private final Object receiving = new Object();
    private final Semaphore senderIn = new Semaphore(0,true);
    private final Semaphore receiverIn = new Semaphore(0,true);
```

```
    private final Semaphore senderMutex = new Semaphore(1,true);
    private final Semaphore receiverMutex = new Semaphore(1,true);

   public Rendezvous() {}


   public final void send(Object m) {
      if (m == null) {
         System.err.println("null message passed to send()");
         throw new NullPointerException();
      }
      senderMutex.acquireUninterruptibly();
         theMessage = m;
         senderIn.release();
         receiverIn.acquireUninterruptibly();
     senderMutex.release();
   }

   public final Object receive() {
      Object receivedMessage = null;
      receiverMutex.acquireUninterruptibly();
         senderIn.acquireUninterruptibly();
         receivedMessage = theMessage;
         receiverIn.release();
       receiverMutex.release();
      return receivedMessage;
   }
}
```

6.12 Wechselseitiger Ausschluss und Synchronisation in JAVA

Jedes Objekt in JAVA (das heisst `Object`) enthält einen *lock*, der in Anspruch genommen werden kann, in dem man eine Methode als `synchronized` kennzeichnet. Mehrere Methoden eines Objektes können als `synchronized` bezeichnet werden:

```
class Counter {
private long count = 0;

   public synchronized void up()
   {
      count = count + 1;
   }
   public synchronized void down()
   {
      count = count - 1;
   }
}
```

Nur ein Thread kann gleichzeitig über die Methoden `up()` oder `down()` auf das Objekt zugreifen. Der Wechselseitige Ausschluss wird mit der Kennzeichnung `synchronized` sichergestellt. Dabei ist zu beachten, dass pro Objekt ein *lock* vorhanden ist und nicht für die Klasse *Counter*. Falls man alle Methoden einer Klasse als `synchronized`

deklariert, so kann man sagen, dass eigentlich eine Art *MONITOR* vorliegt.
Falls man den Wechselseitigen Ausschluss feingranularer gestalten möchte, so kann man den `synchronized`-Block verwenden. Dabei wird dem Block `synchronized` das entsprechende Objekt als Parameter übergeben:

```
class Counter {
private long count = 0;

   public void up()
   {
      synchronized(this) {
         count = count + 1;
      }
   }
   public void down()
   {
      synchronized(this) {
         count = count - 1;
      }
   }
}
```

Oft muss aber im Innern eines kritischen Abschnittes gewartet werden. Der wartende Thread muss nachher auch wieder geweckt werden. Zu diesem Zweck gibt es in JAVA die folgenden Methoden:

wait() Mit dieser Methode kann auf einem Objekt gewartet werden. `wait()` darf nur verwendet werden falls man den *lock* auf das entsprechende Objekt besitzt. Achtung: es gibt *nur* eine Warteschlange pro Objekt!

notify() Entspricht dem *signal* des Monitors. Damit kann ein wartender Thread aufgeweckt werden. Auch `notify()` darf nur verwendet werden wenn man den *lock* des entsprechenden Objektes besitzt (`wait()` und `notify()` sind Teil der Basisklasse `Object`).

notifyAll() Entspricht dem *broadcast* Aufruf.

Ringpuffer als Monitor:

```
public class RingBuffer {
    private int slots=20;
    private int ringbuffer[] = new int[slots];
    private int slotsinuse = 0;
    private int nextslottofill = 0;
    private int nextslottoempty = 0;

    public synchronized void fillaslot ( int value) {

        while (slotsinuse == slots) {
            try {
                wait();
            } catch (InterruptedException e){
            }
        }
            ringbuffer[nextslottofill] = value;
```

```
            slotsinuse= slotsinuse+1;
            nextslottofill= (nextslottofill + 1) % slots;
            notifyAll();
    }

    public synchronized int emptyaslot() {

        int contents;
        while (slotsinuse == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
            contents = ringbuffer[nextslottoempty];
            slotsinuse= slotsinuse-1;
            nextslottoempty=(nextslottoempty+1) % slots;
            notifyAll();
            return contents;
    }

}
```

Man könnte nun versucht sein den *lock* feingranularer setzen, analog dem Kapitel *Wechselseitiger Ausschluss und Synchronisation in Anwenderprogrammen* und erhält dadurch das folgende Programm:

```
public class RingBuffer {
    private int slots=20;
    private int ringbuffer[]= new int[slots];
    private int slotsinuse = 0;
    private int nextslottofill = 0;
    private int nextslottoempty = 0;

    public void fillaslot ( int value) {

        synchronized(this) {
           while (slotsinuse == slots) {
              try {
                wait();
              } catch (InterruptedException e){
                }
           }
            ringbuffer[nextslottofill] = value;
            slotsinuse= slotsinuse+1;
         nextslottofill= (nextslottofill + 1) % slots;
         }
         notifyAll();
    }

    public int emptyaslot() {

        int contents;
        synchronized(this) {
```

```
            while (slotsinuse == 0) {
               try {
                 wait();
               } catch (InterruptedException e) {
                 }
            }
             contents = ringbuffer[nextslottoempty];
             slotsinuse= slotsinuse-1;
          nextslottoempty=(nextslottoempty+1) % slots;
          }
          notifyAll();
          return contents;
    }

}
```

Aber: Dieses Programm wird eine `IllegalMonitorStateException` erzeugen weil das `notifyAll` aufgerufen wird ohne dass man den *lock* auf dem Objekt besitzt. Als Vereinfachung könnte man versucht sein, anstatt von `notifyAll()`, die Methode `notify()` zu verwendet werden, das kann aber zu einem Deadlock führen. Die korrekte Lösung mit dem `synchronized()`-Block ist wie folgt:

```
public class RingBuffer {
    private int slots=20;
    private int ringbuffer[] = new int[slots];
    private int slotsinuse = 0;
    private int nextslottofill = 0;
    private int nextslottoempty = 0;


    public void fillaslot ( int value) {

        synchronized(this) {
           while (slotsinuse == slots) {
              try {
                wait();
              } catch (InterruptedException e){
                }
           }
            ringbuffer[nextslottofill] = value;
            slotsinuse= slotsinuse+1;
            nextslottofill= (nextslottofill + 1) % slots;
            notifyAll();
        }
    }

    public int emptyaslot() {

        int contents;
        synchronized(this) {
           while (slotsinuse == 0) {
              try {
                wait();
              } catch (InterruptedException e) {
```

```
                }
            }
             contents = ringbuffer[nextslottoempty];
             slotsinuse= slotsinuse-1;
             nextslottoempty=(nextslottoempty+1) % slots;
             notifyAll();
         }
         return contents;
    }

}
```

Mit der Methode `interrupt()` ist es möglich, einen in `wait()` blockierten Thread aufzuwecken. Dazu das folgende Beispielprogramm:

```
// Wake a Thread using interrupt method
//
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

class Blocked extends Thread {
    public synchronized void run() {
        try{
            wait(); // blocks
        } catch(InterruptedException e) {
            System.out.println("InterruptedException");
        }
        System.out.println("Exiting run()");
    }
}

public class Interrupt extends Applet {
    private Button interrupt = new Button("Interrupt");
    private Blocked blocked = new Blocked();
    public void init() {
        add(interrupt);
        interrupt.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent e) {
                    System.out.println("Button pressed");
                    if(blocked == null) return;
                    Thread remove = blocked;
                    blocked = null; // release Thread
                    remove.interrupt(); // send interrupt
                }
            });
        blocked.start();
    }
    public static void main(String[] args) {
        Interrupt applet = new Interrupt();
        Frame aFrame = new Frame("Interrupt");
        aFrame.addWindowListener(new WindowAdapter() {
                public void windowClosing
                            (WindowEvent e) { System.exit(0);}});
        aFrame.add(applet, BorderLayout.CENTER);
```

```
        aFrame.setSize(200,100);
        applet.init();
        applet.start();
        aFrame.setVisible(true);
    }
}
```

In JAVA ist es mit wenig Aufwand möglich die Dijkstra-Semaphore zu implementieren:

```
package Concurrent;

public class Semaphore {
    // implements the classical Dijkstra semaphore operations
    // semvalue is the semaphore value, where  semvalue >= 0
    // waiters() is the count of threads waiting in the wait queue
    // value() is value of the Dijkstra semaphore
    // if newvalue() < 0 then abs(newvalue()) is the count of
    // threads in the wait queue

   protected int value = 0;
   protected int waiter = 0;
   protected int semvalue = 0;

   public Semaphore() {value = 0; semvalue = 0;}

   public Semaphore(int initial) {
      if (initial < 0) throw new IllegalArgumentException("initial<0");
      value = initial;
      semvalue = initial;
   }

   public synchronized void P() {

      waiter++;
      while(value<= 0){
             try {
               wait();
             } catch (InterruptedException e) {
                 System.err.println
                 ("Semaphore.P(): InterruptedException, wait again ");
               }
      }
      if (semvalue > 0) semvalue--;
      waiter--;
      value--;
   }

   public synchronized void V() {
      if (waiter == 0) semvalue++;
      notify();
      value++;
    }
```

```
   public synchronized int value() {
      return semvalue;
   }

   public synchronized int newvalue() {
      return semvalue-waiter;
   }

   public synchronized int waiters() {
      return waiter;
   }

   public synchronized String toString() {
      return String.valueOf(value);
   }


   public synchronized void tryP() throws WouldBlockException {
      if (value > 0) this.P();  // nested locking, but already have lock
      else throw new WouldBlockException();
   }

   public synchronized void interruptibleP() throws InterruptedException {
      waiter++;
      while (value <= 0) {
         try { wait(); }
         catch (InterruptedException e) {
            System.err.println
               ("Semaphore.interruptibleP(): InterruptedException");
            waiter--;      // backout, i.e., restore semaphore value
            throw e;
         }
      }
      waiter--;
      value--;
      if (semvalue > 0) semvalue--;
   }
}
```

Mit diesem Semaphor lassen sich Mutexe und Condition Variablen erzeugen. Hier folgt ein Vorschlag für die Mutexe:

```
package Concurrent;


public final class Mutex extends Semaphore {

   public Mutex() {super(1);}  // constructor


    public synchronized void unlock() {
      if (value() != 0) throw new
              IllegalArgumentException("illegal semaphor value = " + value());
```

```
      super.V();
   }
   public synchronized void lock() {
      if (value() > 1) throw new
              IllegalArgumentException("illegal semaphor value = " + value());
      super.P();
   }
}
```

Und hier der Vorschlag für Condition Variablen:

```
package Concurrent;

public final class ConditionVariable extends Semaphore{

   public ConditionVariable() {super(0);}

   public void waitCV(Mutex m) {
      synchronized(this) {
         m.unlock();
         super.P();
      }
      m.lock();
   }

   public synchronized void sendCV() {
       if (newvalue() < 0) {
          super.V();
       }
   }

   public synchronized void broadcastCV() {
       while (newvalue() < 0){
             super.V();
       }
   }
}
```

6.13 Gruppierung von Threads

Jeder JAVA Thread ist Mitglied einer *thread group*. Thread groups geben die Möglichkeit mehrere Threads in ein einziges Objekt zu vereinen und diese Threads gleichzeitig zu bearbeiten.

Default Thread Group

Falls ein Thread erzeugt wird ohne eine Gruppe im Konstruktor anzugeben so wird dieser Thread in die gleiche Gruppe aufgenommen wie der Thread der diesen erzeugt hat. Die erste Thread Gruppe die erzeugt wird vom System heisst *main*. Um einen Thread einer speziellen Gruppe zuzuordnen gibt es die folgenden Konstruktoren:

```
public Thread(ThreadGroup group, Runnable runnable)
public Thread(ThreadGroup group, String name)
public Thread(ThreadGroup group, Runnable runnable, String name)
```

Die Threadgruppe kann gelesen werden mit:

```
theGroup = myThread.getThreadGroup();
```

Auf den Threadgruppen sind folgende Operationen möglich:

- `getMaxPriority` and `setMaxPriority`
- `getDaemon` and `setDaemon`
- `getName`
- `getParent` and `parentOf`
- `toString`
- `destroy`

Beispiel eines Programms das auf eine Thread Gruppe zugreift:

```
public class MaxPriorityTest {
    public static void main(String[] args) {

        ThreadGroup groupNORM = new ThreadGroup(
                                    "a group with normal priority");
        Thread priorityMAX = new Thread(groupNORM,
                                "a thread with maximum priority");

        // set Thread's priority to max (10)
        priorityMAX.setPriority(Thread.MAX_PRIORITY);

        // set ThreadGroup's max priority to normal (5)
        groupNORM.setMaxPriority(Thread.NORM_PRIORITY);

        System.out.println("Group's maximum priority = " +
                           groupNORM.getMaxPriority());
        System.out.println("Thread's priority = " +
                           priorityMAX.getPriority());
    }
}
```

Access Restrictions Die ThreadGroup selbst hat keine Zugriffsbeschänkungen. Aber es gibt eine Methode `checkAccess` die mit dem Security Manager zusammenarbeitet. Beim Aufruf dieser Methode wird eine Security Exception geworfen falls der Zugriff auf eine folgenden Methoden nicht erlaubt ist:

- `ThreadGroup(ThreadGroup parent, String name)`
- `setDaemon(boolean isDaemon)`
- `setMaxPriority(int maxPriority)`
- `destroy`

Für alle Thread Methoden siehe API Dokumentation.

Literatur

[1] Concurrent Programming Using Java, Stephen J. Hartley, Oxford University Press.

[2] Thinking in JAVA, Bruce Eckel, Prentice Hall

6.14 Liveness

Die Fähigkeit eines Programms mit mehreren parallelen Aktivitäten *zeitgerecht* zu arbeiten wird mit Liveness bezeichnet. Die häufigsten Probleme von Liveness sind:

- Starvation und Livelock
- Deadlock

6.14.1 Starvation und Livelock

Obwohl Starvation und Livelock weniger häufig auftreten als ein Deadlock, sind es Probleme die gelegentlich auftreten und die man vermeiden muss.

Starvation
Starvation ist eine Situation bei der ein Thread nicht mehr regelmässig auf die gemeinsamen Betriebsmittel zugreifen kann und keinen Fortschritt mehr erreicht. Das kann durch *gierige* Threads geschehen, die diese Betriebsmittel monopolisieren. Starvation ist aber auch möglich wegen der (zu) tiefen Priorität eines Threads.

Livelock
Ein Thread reagiert oft als Antwort auf die Aktion eines anderen Threads. Wenn die Aktion des anderen Threads ebenfalls von der Aktion eines anderen Threads abhängt so kann ein Livelock entstehen. Beispiel aus dem täglichen Leben: zwei Personen begegnen sich im Korridor: die eine Person weicht nach rechts aus, die andere nach links!

6.14.2 Deadlock

Eine Aktivität in einem Multiprogramming System befindet sich in einem Deadlock, falls diese auf ein Ereignis wartet, das nie eintrifft. Die Haupttätigkeit eines Betriebssystems ist die Betriebsmittel- Verwaltung. Falls Betriebsmittel unter mehrere Benutzer aufgeteilt werden, so erhalten diese die Betriebsmittel zur exklusiven Benutzung. Dabei ist es durch einen Deadlock möglich, dass ein Benutzer seine Arbeit nie abschliessen kann. In diesem Kapitel untersuchen wir das Problem des Deadlocks unter folgenden Gesichtspunkten:

- Deadlock Verhinderung (prevention)
- Deadlock Vermeidung (avoidance)
- Deadlock Entdeckung (detection)
- Deadlock Erholung (recovery)

Die unbestimmte Verzögerung (indefinite postponement) kann den gleichen Effekt wie ein Deadlock haben. Die Korrektur von Deadlock Situationen hat ihren Preis: als Overhead, oder bei der Wiederholung der Arbeit von Aktivitäten.

6.14.3 Deadlock Beispiele

a) Verkehrssituation als Deadlock:

b) Deadlock mit Betriebsmitteln

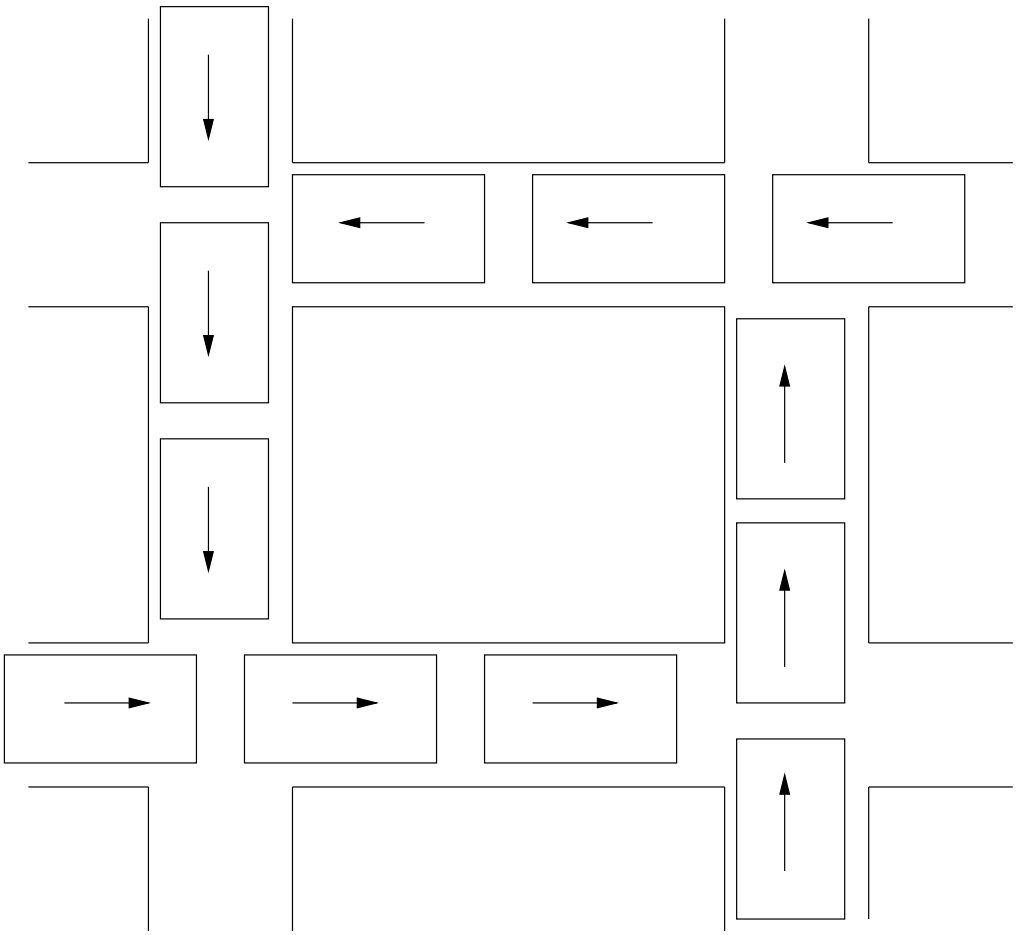


Abbildung 13: Deadlock in Verkehrssituation

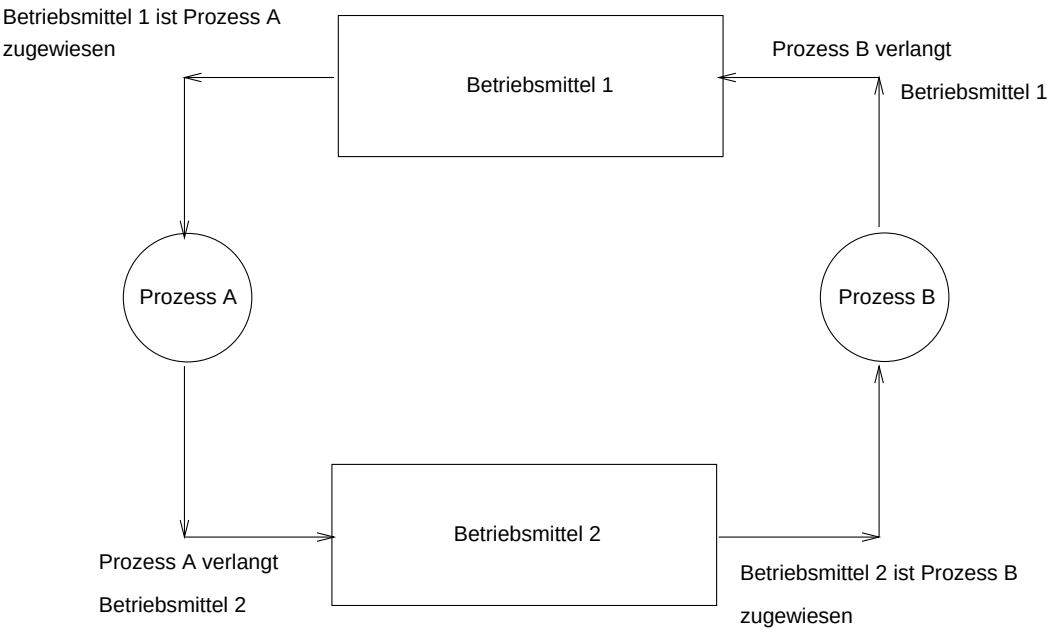


Abbildung 14: Deadlock bei Betriebsmitteln

c) Deadlock in Spulen-Systemen (Spooling Systems)

Spooling wird angewandt um Datenverarbeitung und E/A zu entkoppeln. In gewissen älteren Systemen muss beim Drucker Spooling zuerst der gesamte Output auf den Massenspeicher geschrieben werden, bevor das Ausdrucken beginnt. Falls nicht genügend Platz auf dem Massenspeicher zur Verfügung steht, kann die Deadlock Situation eintreten. Falls mehrere Aktivitäten gleichzeitig Daten auf den Drucker ausgegeben haben, kann für die Erholung vom Deadlock der Abbruch aller am Ausdrucken beteiligter Aktivitäten notwendig sein. Eine Methode zur Verhinderung solcher Deadlocks ist: jederzeit genügend Platz auf dem Massenspeicher zur Verfügung zu stellen. Heutige Betriebssysteme sind meistens so programmiert, dass beim Spooling kein Deadlock mehr möglich ist.

Betriebsmittel Begriffe:

präemptive Betriebsmittel = CPU, Hauptspeicher

nicht präemptive Betriebsmittel = dedizierte Betriebsmittel, z.B. Band-Stationen

geteilte (shared) Betriebsmittel = Hauptspeicher, CPU, zT. Dateien

reentrant Code = Falls Programm nur einmal im Hauptspeicher aber der Code mehrfach inkarniert ist.

serially reusable Code = Code der nur einfach inkarniert werden kann.

6.14.4 Notwendige Bedingungen für einen Deadlock

Coffman, Elphick und Shoshani nennen folgende 4 Bedingungen die notwendig sind für einen Deadlock:

- Aktivitäten verlangen exklusiven Zugriff auf die verlangten Betriebsmittel. (Wechselseitiger Ausschluss)
- Aktivitäten besitzen bereits Betriebsmittel während sie auf die Zuteilung neuer Betriebsmittel warten. (Warten auf..)
- Die Betriebsmittel können den Aktivitäten vor Beendigung der Arbeit nicht entzogen werden. (Betriebsmittel nicht präemptiv)
- Es existiert eine Ringliste der Aktivitäten, jede Aktivität hält ein oder mehrere Betriebsmittel, die von der nächsten Aktivität in der Kette verlangt werden. (circular wait).

6.14.5 Deadlock Verhinderung

Falls eine der obigen Bedingungen verneint werden kann ist der Deadlock ausgeschlossen. Havender schlug die folgenden Strategien vor für die Verhinderung von Deadlocks:

a) Jede Aktivität muss alle ihre Betriebsmittel auf einmal verlangen; sie kann erst fortfahren wenn sie alle erhalten hat.

b) Falls eine Aktivität, die bereits Betriebsmittel besitzt, weitere Betriebsmittel braucht, so muss sie zuerst alle ihre Betriebsmittel zurückgeben. Anschliessend muss sie alle Betriebsmittel zusammen anfordern.

c) Alle Betriebsmittel sind eindeutig numeriert. Die Betriebsmittel werden verlangt in der Reihenfolge aufsteigender Betriebsmittelnummern.

a) Kann aber zu einer Betriebsmittel-Verschwendung führen. Falls eine Aktivität, die total n Betriebsmittel benötigt, in Teilaktivitäten zerlegt werden kann, die m $<=$ n Betriebsmittel benötigen, kann die Betriebmittelauslastung verbessert werden.

b) Durch die Rückgabe der Betriebsmittel kann die gesamte bisherige Arbeit der Aktivität verloren gehen. Falls dies aber selten auftritt, kann dies vielleicht toleriert werden. Es ist möglich, dass bei dieser Methode Aktivitäten unbestimmt aufgeschoben werden.

c) Mit dieser Regel kann nie ein *circular wait* auftreten. Eine kleine Variation dieses Algorithmus ist, dass ein Prozess nie ein Betriebsmittel verlangen soll mit einer tieferen Nummer als die Betriebsmittel die er schon hat.

6.14.6 Deadlock Vermeidung

Falls die vier Bedingungen für einen Deadlock gegeben sind, kann mit einer sorgfältigen Betriebsmittel-Zuweisung ein Deadlock dennoch vermieden werden.

Betriebsmittel Trajektorien

Die Grundüberlegung bei der Deadlock Vermeidung ist das Verbleiben im *sicheren Zustand* (save state). Als Beispiel nehmen wir zwei Prozesse A und B die sich um zwei unterschiedliche Betriebsmittel (zum Beispiel einen Drucker und einen Plotter) bewerben.

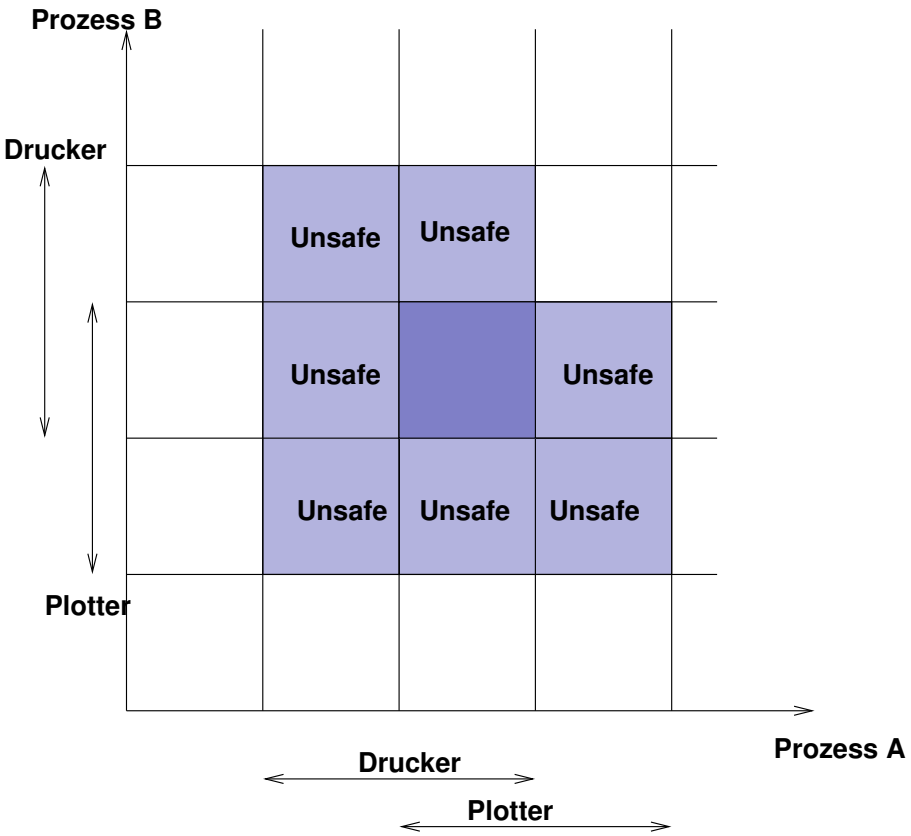


Abbildung 15: Betriebsmittel Trajektorien

In Abbildung 15 ist in horizontaler Richtung Prozess A aufgetragen, und in vertikaler Richtung Prozess B. Der abgebildete Zustandsraum kann nur in Richtung *rechts* oder in Richtung *oben* beschritten werden. Sobald man in ein *Unsafe* Gebiet kommt, kann man deshalb einem Deadlock nicht mehr entweichen. Der Algorithmus für die Deadlock Vermeidung muss also alle Gebiete die *Unsafe* sind umgehen.

Das berühmteste Beispiel für die Deadlock Vermeidung ist der Banker-Algorithmus von Dijkstra. Ein Bankier leiht Geld aus und erhält Rückzahlungen. Er arbeitet mit einer gegebenen Menge Geld. (= Kapital) Übersetzt in ein Betriebssystem geht es beim Banker-Algorithmus um die Verwaltung einer Menge gleichartiger Betriebsmittel.
Jeder Kunde spezifiziert im voraus seinen maximalen Kapitalbedarf. Der Bankier akzeptiert den Kunden falls sein Bedarf das Kapital nicht überschreitet. Während einer Transaktion eines Kunden kann dieser nur eine Geldeinheit nach der anderen verlangen. Manchmal muss ein Kunde warten, bevor er eine Geldeinheit geleiht bekommt. Der Bankier garantiert aber eine endliche Wartezeit. Der Kunde seinerseits garantiert die Rückzahlung seines Kredits in endlicher Zeit.
Die aktuelle Situation des Bankiers ist sicher, falls er alle seine Kunden in die Lage versetzen kann ihre Transaktionen in endlicher Zeit abzuschliessen. Andernfalls ist die Situation unsicher.

Situation eines Kunden: Anspruch = Bedarf - Kredit

Situation des Bankiers: Liquidität = Kapital - Summe der Kredite

Sichere Situation:

Lq = 2	Lq = 4	Lq = 8	Lq = 10
P = 4(4)	P = 4(4)		
Q = 2(1)			
R = 2(7)	R = 2(7)	R = 2(7)	

Der Bankier hat 10 Geldeinheiten (Lq). Er verteilt diese unter die drei Kunden P, Q und R. R = 2(7) bedeutet, dass R bereits 2 Geldeinheiten besitzt und Anspruch auf 7 weitere Einheiten hat.

Unsichere Situation:

Lq = 1	Lq = 3
P = 4(4)	P = 4(4)
Q = 2(1)	
R = 3(6)	R =3(6)

→ Lösung des Problems: Hausaufgabe!

Schwächen des Banker-Algorithmus:

- Feste Anzahl von Betriebsmitteln
- Feste Anzahl von Benutzern der Betriebsmittel
- Betriebsmittel werden in endlicher Zeit zugewiesen (lange Wartezeiten)
- Rückgabe der Betriebsmittel in endlicher Zeit ist notwendig
- Maximale Anzahl der Betriebsmittel müssen im voraus angegeben werden.

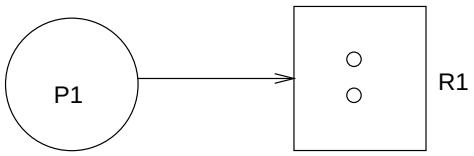


Abbildung 16: P1 verlangt R1

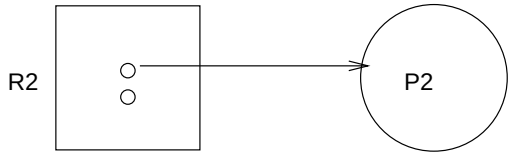


Abbildung 17: R2 wurde Aktivität P2 zugewiesen

6.14.7 Deadlock Erkennung

Deadlock Erkennungsalgorithmen untersuchen ob ein 'circular wait' vorliegt. Falls Erkennungsalgorithmen implementiert werden erzeugen diese einen sog. 'runtime-overhead'. Die Untersuchung ob ein Deadlock vorliegt geschieht am einfachsten mit Betriebsmittel-Zuweisungs-Graphen. Das sind gerichtete Graphen die Betriebsmittel-Anforderungen und -Zuweisungen darstellen:

a) P1 verlangt ein Betriebsmittel vom Typ R1:

b) Ein Betriebsmittel des Typs R2 wurde Aktivität P2 zugewiesen:

c) Aktivität P3 verlangt Betriebsmittel R3 das Aktivität P4 zugewiesen wurde.

d) Aktivität P5 hat das Betriebsmittel R5 zugewiesen erhalten, das von Aktivität P6 verlangt wird. P6 hat das Betriebsmittel R4 erhalten, das von Aktivität P5 verlangt wird (circular wait).

   Für die Feststellung ob ein Deadlock vorliegt können die Graphen reduziert (vereinfacht) werden. Aktivitäten deren Betriebsmittel-Anforderungen befriedigt werden können, werden aus dem Graphen eliminiert. Sobald ein Graph nicht weiter reduziert werden kann, hat man die Aktivitäten die sich im Zustand 'Deadlock' befinden gefunden.

$\Longrightarrow$ siehe file-locking in UNIX

6.14.8 Deadlock Erholung

In den meisten Systemen werden die Aktivitäten die im 'Deadlock' sind vom System entfernt. Damit können die Betriebsmittel wieder freigesetzt werden.

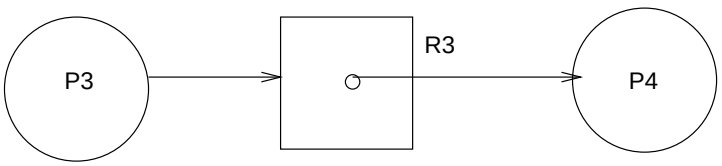


Abbildung 18: P3 verlangt R3 das Aktivität P4 zugewiesen ist

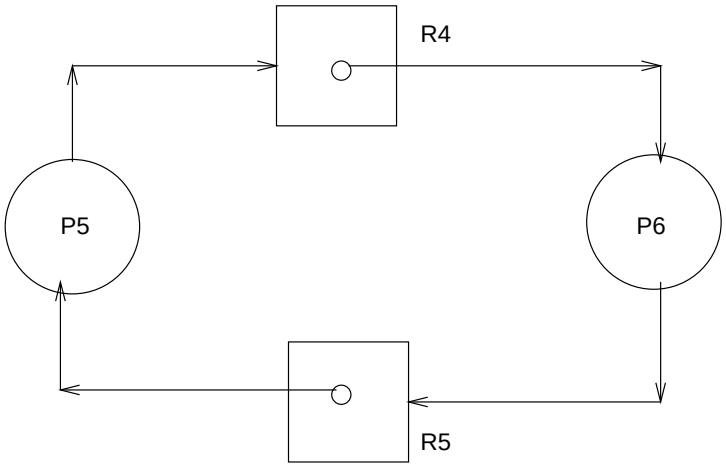


Abbildung 19: Circular wait

7 Entwurf paralleler Aktivitäten

Bei der Modularisierung von Software wird versucht, Einheiten von grosser Kohäsion zu bilden. Dabei werden verschiedene Arten der Kohäsion unterschieden:

- funktionale
- sequentielle
- kommunikative
- prozedurale
- zeitliche
- logische

Modularisierung wird erst handhabbar falls die Interaktion zwischen den Modulen begrenzt wird. Im Gegensatz zur sequentiellen Programmierung können in einer Umgebung mit mehreren Aktivitäten Interaktionen zwischen Modulen auftreten, die funktionell und konstruktiv unabhängig sind. Dabei geht es oft um das Aufteilen von Betriebsmitteln (Speicher, Prozessor, usw.).

Gesucht wird eine Vorgehensweise die es erlaubt für jedes Problem eine einfache Lösungs-Struktur zu finden, sodass die Software erstellt und einfach gehandhabt werden kann. MASCOT wurde genau für diese Anforderung entworfen (MASCOT = Modular Approach to Software Construction Operation and Test).

7.1 MASCOT-2 Einführung

Bei MASCOT sind die kooperierenden, parallelen Aktivitäten Startpunkt für die Modularisierung. Zusätzlich stützt man sich beim Entwurf mit MASCOT auf den Datenfluss ab. Für jede Meldung wird zwischen der aktiven Aktivität (Quelle) und der passiven Aktivität (Senke) unterschieden. Eine Aktivität könnte grundsätzlich uneingeschränkt auf alle Daten zuzugreifen, was aber im Interesse der Modularisierung unterlassen werden sollte. MASCOT besteht aus einem sehr einfachen Modularitätskonzept. Folgende Hauptkomponenten werden in MASCOT unterschieden:

- die Aktivität
- der Interkommunikations Daten Bereich (IDA = Intercommunication Data Area)

Eine Aktivität besteht aus einem Prozess, Task oder Thread. Jede Kommunikationsanforderung einer Aktivität wird explizit definiert und gesteuert. Die Datenkommunikation zwischen den Aktivitäten findet unter Verwendung der IDA's statt. Bei den IDA's unterscheidet man zwischen:

- Kanal (Channel) und
- Pool

Ein Kanal übergibt Daten in Form von *Messages* von Aktivität zu Aktivität oder von Gerät zu Aktivität. Ein Kanal hat zwei unidirektionale Schnittstellen: die Eingabe Schnittstelle für Aktivitäten die Daten erzeugen und die Ausgabe Schnittstellen für Aktivitäten die konsumieren. In einem Kanal können unkonsumierte Meldungen vorhanden sein.

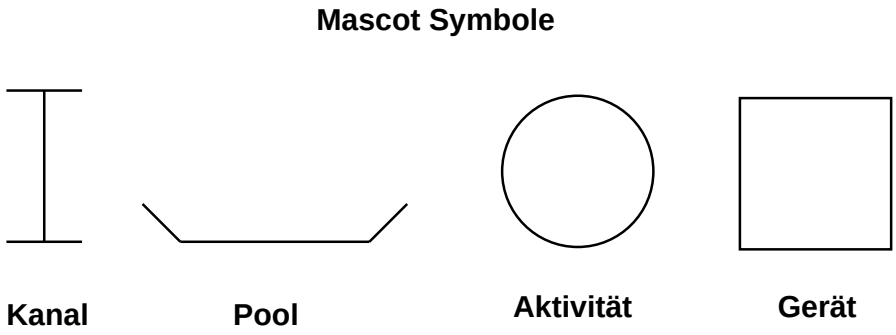


Abbildung 20: MASCOT Symbole

Ein Pool enthält Daten für Auskunftszwecke. Diese können sehr oft gelesen werden. Von Zeit zu Zeit können diese Auskunftsdaten aufgefrischt werden. Dies geschieht durch überschreiben der alten Information. Innerhalb der beiden IDA Klassen sind verschiedene Typen möglich. Einzelne Kanäle können für die Übergabe von Buchstaben, oder irgendwelchen anderen Datentypen aufgebaut werden. Ein Pool kann zum Beispiel definiert werden um den Katalog eines Massenspeichers zu beinhalten.

In der Praxis wird die Charakteristik einer IDA durch einen Satz von Operationen und Zugriffsmechanismen (Funktionen/Methoden) definiert.

Mit Pfeilen wird der Datenfluss dargestellt. Es gilt die Konvention, dass auf der einen Seite des Kanals die Produzenten und auf der anderen die Konsumenten sind. Die Verwendung der obigen Symbole erfolgt mittels des ACP (Activity Channel Pool) Diagrammes.

Das ACP Diagramm dient demselben Zweck wie eine Produktionszeichnung im Bereich des klassischen Engineering.
Hauptunterschied zwischen MASCOT und anderen Methoden:

Konventionell werden parallele Aktivitäten und deren Kommunikation in einem Programm festgelegt. In MASCOT werden zuerst die Aktivitäten und deren Kommunikation definiert. Die Programmierung hat nur eine untergeordnete Bedeutung. Die Spezifikation einer Aktivität (der Quellentext einer Programmversion dieser Aktivität) wird *Root* genannt. Bei den Kanälen und Pools gibt es keine speziellen Namen für die Spezifikation.

Bei der Realisierung des ACP Diagramms als Ausführungs- Netzwerk wird mit MASCOT folgendes durchgeführt:

- Jeder Aktivität wird nur erlaubt auf diejenige Menge von Kanälen und Pools zuzugreifen für die eine Verbindung im ACP Diagramm besteht.

- Bei jeder IDA wird geprüft ob es sich um den richtigen Datentyp handelt. Diese Überprüfungen werden durch die Zugriffsoperationen in den einzelnen Aktivitäten duchgeführt.

Spezifikation der Einzel-Aktivitäten:

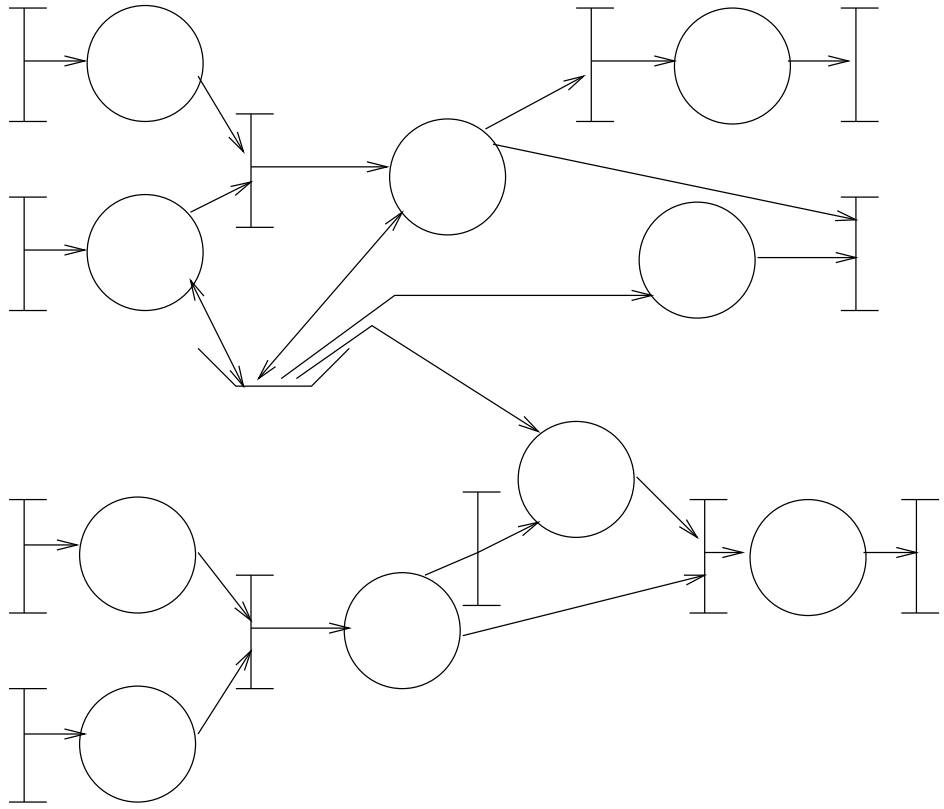


Abbildung 21: ACP Diagramm

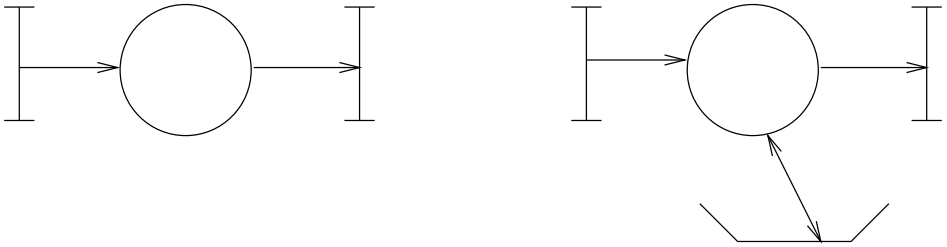


Abbildung 22: Einzel-Aktivitäten

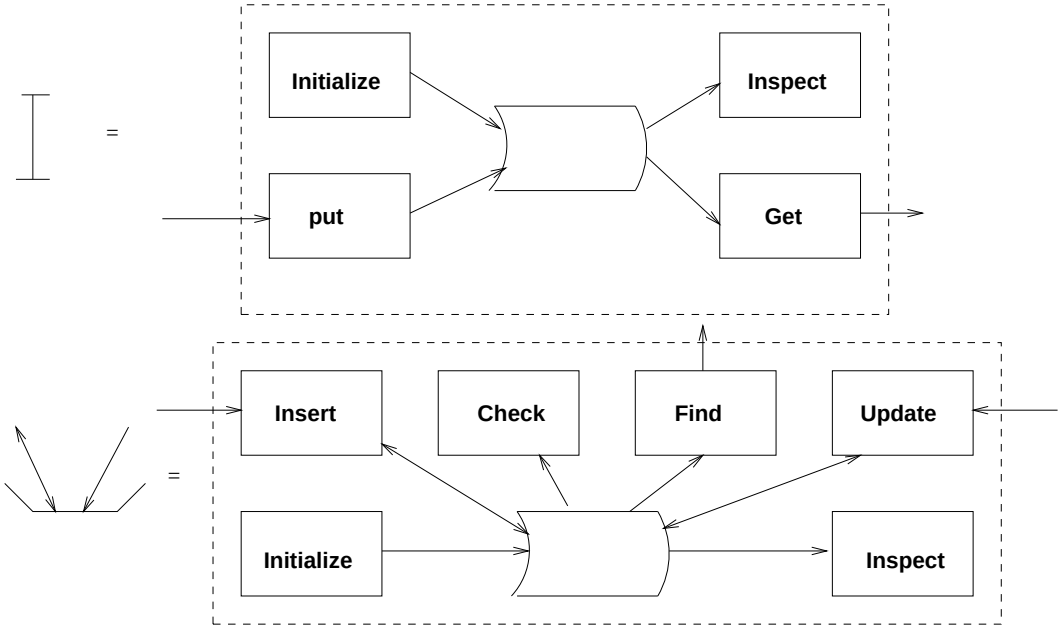


Abbildung 23: ACP Beispiele

Der Begriff des Programmes wird in MASCOT ersetzt durch den Begriff Subsystem. Um eine geordnete Kommunikation unter den einzelnen MASCOT Aktivitäten zu erreichen, ist ein gewisses Mass an Synchronisation notwendig. Die Synchronisation und alle andere Kommunikation zwischen Aktivitäten wird erreicht durch Zugriffsmechanismen die in den IDA's versteckt werden. Diese Zugriffsmechanismen definieren das Interface zwischen der Root und den Kanälen und Pools. Sie bestehen aus einem Satz von Operationen die von der Root benützt werden können ohne die genaue Struktur der IDA zu kennen. Damit ist es möglich, die Root als sequentielles Programm zu erstellen. Die Kanäle und Pools zusammen mit den Zugriffsmechanismen enthalten all die Aspekte der parallelen Verarbeitung.

MASCOT Maschine

All die Werkzeuge die bis jetzt erklärt wurden, werden in ihrer Gesamtheit als MASCOT Maschine bezeichnet. Zur Programmentwicklung mit MASCOT braucht man konventionelle Compiler, Linker und Dateisysteme und die MASCOT vorgehensweise. Die Programm Sprache hat eine untergeordnete Rolle in MASCOT. Es muss aber möglich sein die:

- Root mit allen Prozeduren zu codieren und die
- IDA Spezifikationen festzulegen.

Der Compiler sollte die Kompilation von einzelnen Aktivitäten ermöglichen. Um den Zusammenbau der Komponenten optimal zu unterstützen, sollten Linker und Compiler möglichst alle Spezifikationen auf ihre Kompatibilität überprüfen.

MASCOT als Entwicklungs-Hilfsmittel

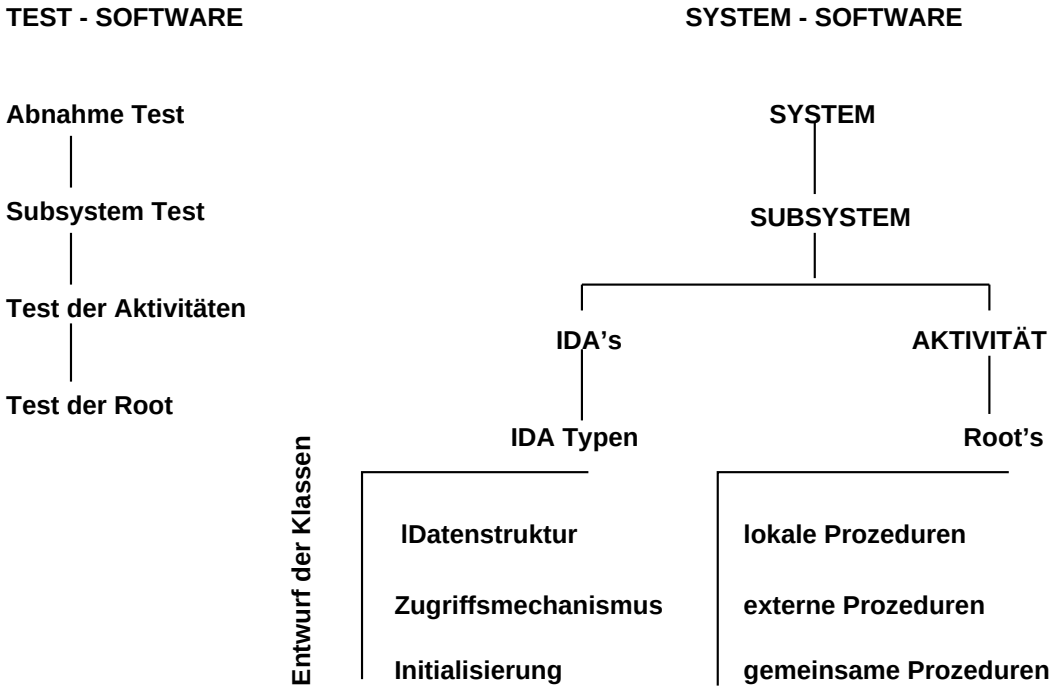


Abbildung 24: Entwurfs-Hierarchie

MASCOT gibt Unterstützungswerkzeuge für das Arbeiten mit Software während des gesamten SW-Lifecyle. Dabei sollte wie folgt vorgegangen werden:

1. Aus der technischen Spezifikation wird ein Gesamt-Entwurf der Software erstellt, der mit dem ACP Diagramm dokumentiert wird. Mit diesem Diagramm können der Software Aufwand abgeschätzt und die Termine festgelegt werden.

2. Jede Aktivität, Kanal und Pool wird weiter zerlegt mit den in der Figur 24 angegebenen Schritten. Am Ende der Phase 2 ist alle Software in einer höheren Sprache erstellt.

3. In dieser Phase wird implementiert und getestet. Implementierung bedeutet die Übersetzung des Programmtextes vom Manuskript in maschinenlesbare Form, die getestet werden kann. Darin sind also Kompilation, Editieren, erneut Kompilieren und Linken eingeschlossen. Beim Testen werden alle in Phase 1 festgelegten Module ausgetestet und der Abnahmetest durchgeführt.

7.2 Grobentwurf von Software mit MASCOT-2

In der Praxis wird die reine top-down Methode kaum angewandt. Man braucht ein iteratives Vorgehen. In MASCOT steht das ACP Diagramm im Vordergrund. Als erste Tätigkeit muss das richtige ACP Diagramm erstellt werden, dieses stellt das wesentliche Resultat des Entwurfs dar.

Startpunkt

Die Spezifikation eines Systems definiert die externen Funktionen des Systems. Im allgemeinen besteht keine direkte Beziehung zwischen diesen externen Funktionen und denjenigen die einzelne Software Module verrichten. Die Beziehung kann ausgedrückt werden in Form einer Matrix (ext. Funktionen/sw Funktionen). Diese Matrix zusammen mit der Spezifikation der externen Funktionen kann verwendet werden zum Entwurf der System- und Integrationstests.

Wie erfolgt der Entwurf

Man betrachtet den Datenfluss von allen Eingabegeräten zu allen Ausgabegeräten indem man das ACP Diagramm erstellt. Damit entsteht eine sehr gute Übersicht und man erhält eine gute Basis für die Diskussion im Projektteam. Beim Erstellen des ACP Diagramms muss für jedes Element im Netzwerk der genaue Zweck festgelegt und das Verständnis für die gesamte Software vorhanden sein. Das ständige Fragen nach dem Zweck ermöglicht eine Verdeutlichung der Systemspezifikationen und führt zwangsläufig auf ein iteratives Vorgehen beim Entwurf. Bei diesem Vorgehen werden viele Systembeschränkungen sichtbar werden. Diese bilden einen Teil der Systemspezifikation sind aber nicht notwendigerweise auch Systemfunktionen. Das Prozedere die Spezifikationen in ein ACP Diagramm umzuwandeln ist die schwierigste und anspruchsvollste in der Parallelprogrammierung. Dabei werden: Kreativität, Erfahrung und Kenntnis der Anwendung verlangt. Die Verfeinerung des Entwurfs im kleinere und weniger komplexe Funktionen, damit die einzelnen Komponenten handhabbar werden, setzen gute Übersicht voraus. In folgenden Fällen ist der Entwurf nochmals zu überprüfen:

a) eine Aktivität die mehr als einen Kanal liest.

b) ein Pool in den zwei oder mehr Aktivitäten schreiben

c) ein Kanal der von zwei oder mehr Aktivitäten gelesen wird (Es sei denn diese haben identische Funktion).

Das ACP Diagramm kann eine grosse Anzahl von Komponenten enthalten. Nachdem festgestellt ist, dass die Spezifikationen erfüllt sind, muss das System in Bezug auf die Antwortzeiten analysiert werden. Vielleicht bringt die grosse Modularität zu grossen Overhead mit sich, sodass ein teilweiser Neu-Entwurf notwendig wird:

Dabei soll in folgender Reihenfolge vorgegangen werden:

- Ketten von Aktivitäten
- Ansammlungen von Aktivitäten bei denen ein Kombinieren der individuellen Funktionen einfach möglich ist.
- Ansammlung von Aktivitäten bei denen ein Kombinieren der individuellen Funktionen weniger offensichtlich ist.

Falls nach diesen Schritten das Projekt immer noch nicht innerhalb der Systemspezifikationen liegt, sollten diese nochmals zur Diskussion gestellt werden.
Die Testbarkeit des Systems stellt ein gutes Mass für die durchgeführte Modularisierung dar. Die Module eines gut entworfenen Netzwerkes können auch gut getestet werden.

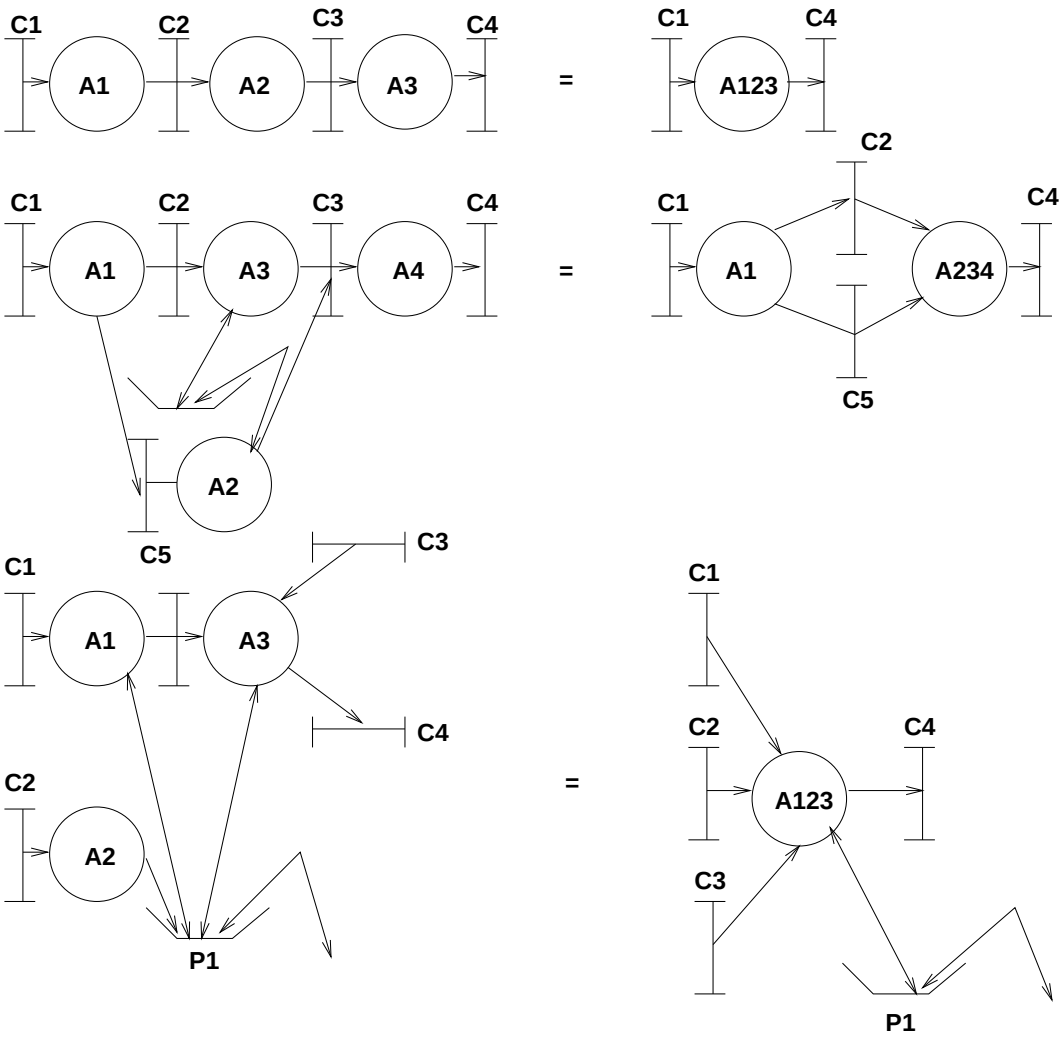


Abbildung 25: ACP Vereinfachungen

Grosse Systeme

Für grosse komplexe Systeme kann es notwendig sein, einen oder mehrere Zwischenstufen einzuschalten. Achtung: dies ist nur ein konzeptionelles Hilfsmittel, MASCOT-2 erlaubt keine Implementierung von hierarchischen Stufen. Das ACP Diagramm stellt den Entwurf dar, der nicht hierarchisch gegliedert ist.

Eine Möglichkeit der Gruppierung ist die Gliederung in Subsysteme. Falls mehrere Stufen der Beschreibung notwendig sind kann man mit SIDA's (Subsystem Interkommunikations Daten Bereich) ein Entwurf durchgeführt werden. Dabei gehen auf der höheren Stufe alle IDA's im Innern des Subsystems verloren. Auf der/den unteren/n Stufe/n werden dann weitere SIDA's und auf der untersten Stufe die IDA's entworfen.

Organisation des Teams

Das ACP Diagramm ermöglicht das Studium alternativer Implementierungs-Sequenzen. Bei einem grossen Team ist es denkbar, dass jeder Modul einzeln geprüft wird. Bei einem kleinem Team kann bei den Tests bereits getestete Software mit verwendet werden. Erst das fertige ACP Diagramm ermöglicht die genaue Bestimmung der Hardware Konfiguration und eine Abschätzung des Software Aufwands.

Software Prototypen

Software Prototypen müssen dieselbe Struktur haben wie das vorgeschlagene System. Sie müssen schnell und frühzeitig im Projektablauf erstellt werden. Falls die Erstellung eines solchen Prototypen möglich ist, helfen diese bei der Entwicklung der endgültigen SW ganz beachtlich.

Zusammenfassung

Die Resultate der ersten Phase sind:

a) Klarere Anforderungen

b) Ein Entwurf als ACP Diagramm

c) Verständnis für jedes ACP Element

d) Eine Implementierungs-Sequenz

e) Test Strategie für

   - Gesamt-Test
   - Integrationstests (Subsysteme)
   - ACP Netzwerk Elemente

f) Abschätzung der Betriebsmittel

g) Schnittstellen (Interface Details)

h) Software Prototypen oder Vorschlag zu deren Realisierung

7.3 Detailentwurf mit MASCOT

Entwurf der IDA's

Hier müssen die genauen Anforderungen an die IDA zusammengestellt werden. In vielen Fällen ist es möglich eine Standard IDA einzusetzen. Der Entwurf der Datenstruktur und der zugehörigen Zugriffsmechanismen muss gleichzeitig erfolgen (in der OO-Programmierung wird dies beim Programmieren der Klasse sichergestellt), da diese beiden eng miteinander verknüpft sind.

Entwurf der Root

Hier können die bekannten Regeln der Programmiermethodik angewandt werden. Alle Interprozess-Kommunikation ist in den IDA's versteckt. Die Root ist also nur ein sequentielles Programm. Am Ende des Detailentwurfs müssen alle Software -Komponenten so programmiert sein und alle Entscheidungen getroffen sein, dass keine Iterationen mehr nötig sind. Es ist nicht nötig, dass die gewählte (höhere) Programmsprache mit der Implementierungs-Sprache übereinstimmt.

Entwurf der Tests
Bei diesen Tests muss folgendes vorbereitet oder überprüft werden:

- die Konsistenz des ACP Netzwerkes
- Generierung von Testdaten
- Sequenz der Kommandos die nötig sind um die SW zu steuern
- die Ausgabedaten

IDA Tests

Diese Tests können ziemlich unabhängig von der übrigen SW durchgeführt werden. Die entsprechenden Test-Roots bestehen aus Aufrufen der verschiedenen Zugriffsmechanismen. Wichtig ist die Sicherstellung, dass die Synchronisation nicht durch das geschickte Scheduling oder das wählen einer bestimmten Priorität zustande kommt.

Root Tests

Bei diesen Tests wird jedes Programm funktionell überprüft. Dabei können die bekannten Testmethoden angewandt werden.

Test der Aktivitäten

Hier wird die Root als Prozess getestet. Es muss ein Test- Subsystem erstellt werden das die notwendigen Stimuli liefert.

Abnahme Test

Hier ist es denkbar, dass Simulatoren benötigt werden. Solche Simulatoren (oder Testsoftware) können aber auch beim Testen von Aktivitäten helfen.

Zusammenfassung

Der Detailentwurf soll folgende Resultate liefern:

- a) Detaillierter Entwurf der einzelnen Komponenten (IDA, Root)
- b) Detailierte Testprozedere

7.4 Implementierung und Test

Das Resultat dieser Phase ist ist das vollständig funktionierende, und dokumentierte System. Mit den bisherigen Mitteln wurde für die Implementierung und Tests etwa 50% der Zeit benötigt. Mit MASCOT sollte hier eine wesentliche Reduktion möglich sein.

- Die erste Hürde die in dieser Phase überwunden werden muss besteht im Kompilieren der Software. Hier kann mit der komfortablen Programmierumgebung viel Zeit gewonnen werden.
- Die einzelnen Prozeduren die von den Aktivitäten verwendet werden werden getestet.
- Nun werden die Aktivitäten getestet. Dabei werden Test-Subsysteme benötigt. Falls Simulationen angewandt werden so sollen diese nur die Schnittstellen-Spezifikation befriedigen und keine Synchronisationen beinhalten.

Zusammenfassung

Folgende Resultate sind nach dieser Phase vorhanden:

- Quellenprogramme und kompilierte Form aller Softwaremodule
- Link-Listen aller Module
- Eingabedaten für die Tests
- Testresultate

7.5 MASCOT-3

Das MASCOT-2 hat seine Stärke beim Entwurf einfacher Systeme. Für einen Entwurf eines grossen im Netzwerk verteilten Systems treten aber eine Reihe von Mängel zu Tage:

- Bei MASCOT-2 besteht eine eindeutige Verbindung zwischen einer bestimmten IDA und einer bestimmten Aktivität.
- MASCOT-2 kennt keine hierarchischen Strukturen, eine Gliederung in Subsystem und Sub-Subsysteme wird nicht unterstützt.

Bei MASCOT-2 wurde zuerst immer das ACP Diagramm erstellt. Neu in MASCOT-3 wird zuerst die Hierarchie der Subsysteme mit den IDA's und den Servern entworfen. Erst später erfolgt dann das Entwerfen der einzelnen Subsysteme mit den Aktivitäten.

Beispiel eines Systementwurfs mit MASCOT-3

Bei diesem Diagramm können folgende Unterschiede festgestellt werden:

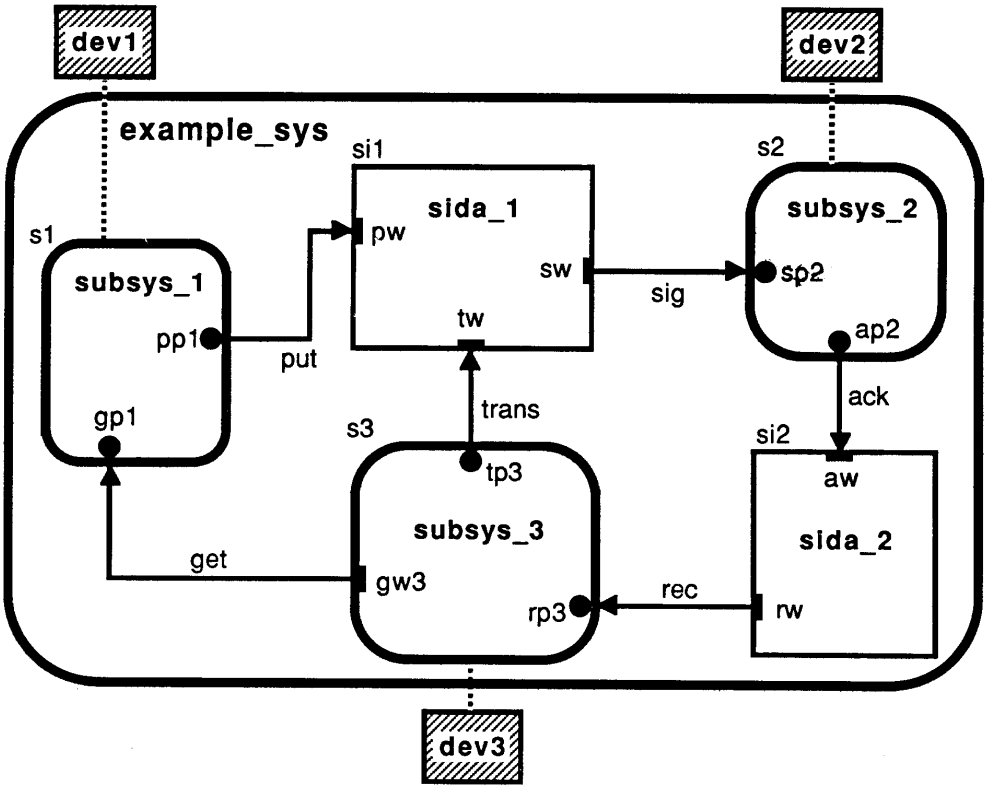


Abbildung 26: MASCOT-3 Gesamtsystem

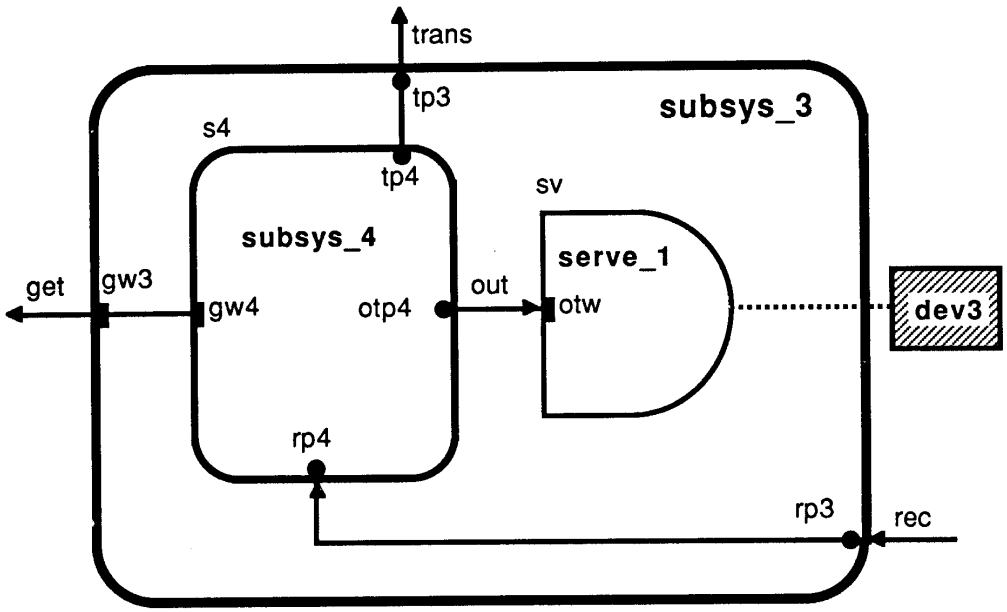


Abbildung 27: Subsystem 3

- Die Koppelung zwischen IDA und Subsystem ist nicht mehr fest: Das Subsystem hat einen *Port* mit dem es auf eine (irgend eine) IDA zugreift. Als Gegenstück hat die IDA ein *Window* über das eine Aktivität zugreifen kann. Mit Pfeilen wird die Datenrichtung festgelegt (wie bisher).
- Das Gesamtsystem ist hierarchisch aufgebaut. Es besteht aus zwei IDA's und drei Subsystemen.

In jedem Subsystem ist mindestens eine Aktivität vorhanden. Das entworfene System kann weiter verfeinert werden, indem man die einzelnen Subsysteme untersucht.

Subsystem 3:

Dieses Subsystem enthält ein zusätzliches Subsystem (subsys 4) und einen Server. In MASCOT ist ein Server ein Element, das mit einem externen Gerät kommunizieren kann. Server sind eng mit IDA's und Aktivitäten verwandt. Das Symbol soll an etwas erinnern, das zwischen Aktivität und IDA steht. Man kann sich einen Server als Hardware-Treiber vorstellen.

Das Subsystem 4:

Insgesamt kann MASCOT-3 als Erweiterung und Verfeinerung von MASCOT-2 verstanden werden. Für einfache kleine Systeme ist MASCOT-2 eindeutig einfacher. Grosse Systeme können nur mit MASCOT-3 entworfen werden.

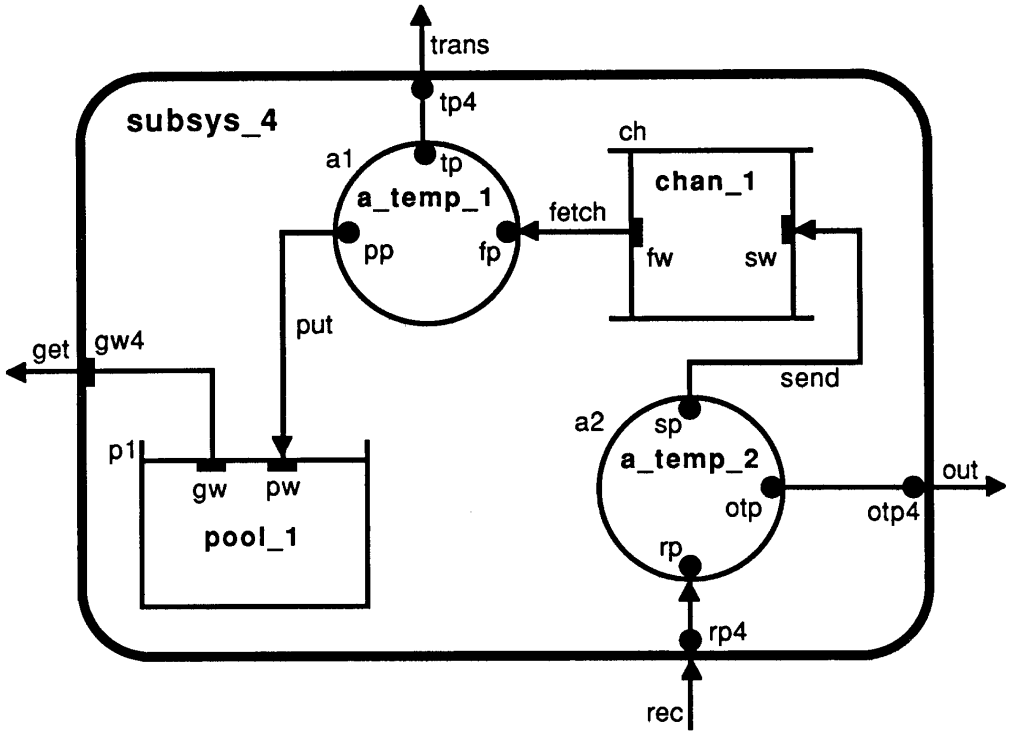


Abbildung 28: Subsystem 4

8 Speicherverwaltung

Die Organisation und die Verwaltung des Speichers ist eine der Schlüssel-Aufgaben beim Entwurf eines Betriebssystems. Man unterscheidet zwischen Hauptspeicher (storage, memory) und Sekundärspeicher (Disk, Tape). In diesem Kapitel werden die verschiedenen bekannten Methoden der Speicherverwaltung diskutiert- (realer und virtueller Hauptspeicher).

8.1 Realer Hauptspeicher

Historisch gesehen ist Hauptspeicher ein sehr teures Betriebsmittel. Lange Zeit war man gezwungen möglichst viele Aufgaben mit wenig Speicher zu lösen.

8.1.1 Speicher Organisation

Mit Speicher-Organisation bezeichnet man die Gliederung des Hauptspeichers. Nachstehende Fragen müssen von der Speicherverwaltung beantwortet werden:

- Ein oder mehrere Benutzer gleichzeitig im Hauptspeicher?
- Bei mehreren Benutzern: ist die Partition jedes Benutzers gleich gross?
- Legen wir die Grösse der Partitions ein für allemal fest?
- Arbeiten gewisse Benutzer in festen Partitions?
- Muss eine einzelne Partition aus zusammenhängendem Speicher bestehen oder kann diese sich aus mehreren Teilen zusammensetzen?

8.1.2 Speicher Verwaltung

Unabhängig von der gewählten Speicher-Organisation müssen wir die Strategien festlegen zur Erzielung der optimalen Leistung. Die Speicherverwaltung bestimmt wie sich eine gegebene Speicher- Organisation unter verschiedenen Umständen verhält:

- Wann wird ein neues Programm in den Hauptspeicher geladen?
- Wird das Programm geladen, wenn verlangt; oder wird es vorausschauend geladen?
- Wohin im Hauptspeicher wird ein neues Programm geladen?
- Falls ein neues Programm geladen werden soll und der Hauptspeicher ist voll: welches der anderen Programme wird verdrängt?

8.1.3 Speicher Hierarchie

Bei einfacheren Micro- und Mini- Rechnern haben wir die zweistufige Speicher-Hierarchie: Hauptspeicher - Massenspeicher. Grössere Systeme arbeiten mit der dreistufigen Hierarchie: Cache - Hauptspeicher - Massenspeicher.

Der Cache - Speicher

Virtueller Speicher: Versuch den Adressraum des (schnelleren) Hauptspeichers um den Adressraum des (langsameren) Sekundär-Speichers zu erweitern. Gesamter Speicher hat scheinbar die Geschwindigkeit des Hauptspeichers.

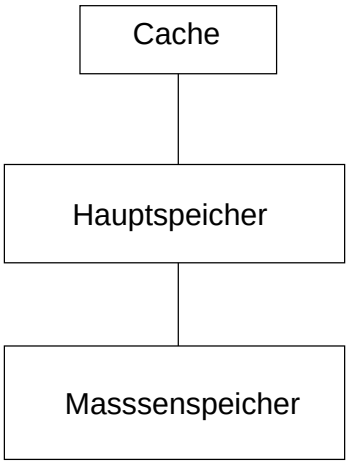


Abbildung 29: Cache und Hauptspeicher

Nach dem gleichen Prinzip kann man einen schnellen Speicher zwischen Hauptspeicher und Prozessor legen. → Hauptspeicher erhält scheinbar die Geschwindigkeit des Cache. Aus Kostengründen ist Cache viel kleiner als Hauptspeicher.

Cache und Hauptspeicher bilden eine Einheit; dem Programmierer ist es gleichgültig ob ein referiertes Wort im Cache oder Hauptspeicher ist.
→ Cache ist für Programmierer transparent!

8.1.4 Strategien der Speicherverwaltung

Teure Betriebsmittel müssen intensiv verwaltet werden damit diese den optimalen Nutzen erbringen. Speicherverwaltungs Strategien können in folgende Kategorien unterteilt werden:

- Lade Strategien (fetch strategies)
  - Demand fetch
  - Anticipatory (vorwegnehmend) fetch
- Platzierungs Strategien (placement strategies)
- Ersetzungs Strategien (replacement strategies)

Die Lade Strategien befassen sich mit dem Zeitpunkt wann ein neues Programm oder neue Daten in den Hauptspeicher geladen werden sollen. Während vieler Jahre war 'Demand fetch' die einzig verwendete Strategie. Demand Fetch bedeutet, dass erst geladen wird, falls das entsprechende Programm oder die Daten benötigt werden. Heute weiss man, dass mit Anticipatory fetch die Systemleistung vergrössert werden kann. Die Platzierungsstrategien befassen sich mit der Frage wo im Hauptspeicher das zu ladende Programm abzuspeichern sei. Mögliche Strategien sind: first-fit, best-fit, worst-fit. Die Ersetzungsstrategien kümmern sich um die Frage welches Programm oder welche Daten ausgelagert werden können, um Raum zu schaffen für ein neues Programm.

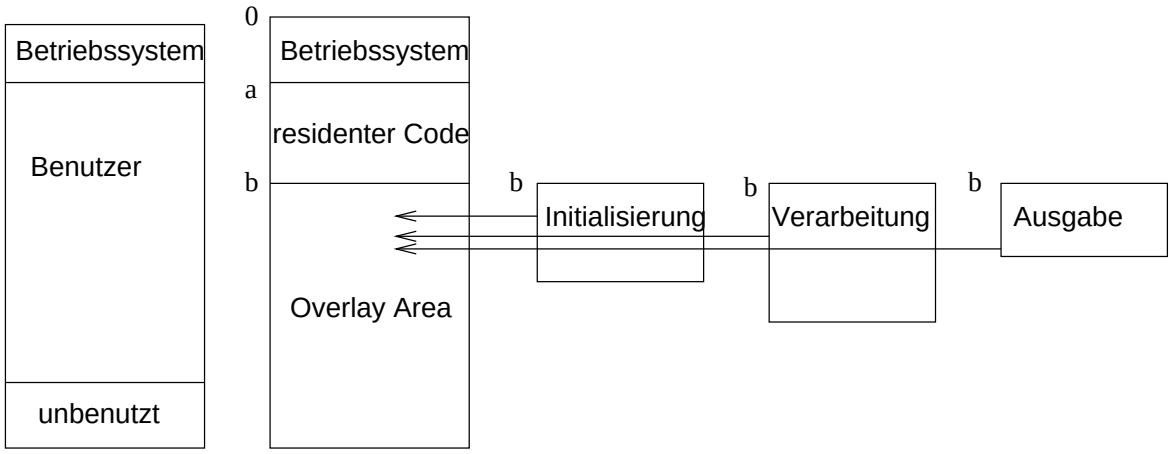


Abbildung 30: Overlays

8.1.5 Zusammenhängender Speicher in Einbenützer-Systemen

Die ersten Rechnersysteme waren Einbenützer Systeme. Die Grösse der Programme ist durch den vorhandenen Hauptspeicher begrenzt. Durch die Verwendung der Overlay Technik ist es möglich grössere Programme auszuführen, als der vorhandene Hauptspeicher.

Overlays geben dem Programmierer die Möglichkeit begrenzten Hauptspeicher zu erweitern. Das Planen und Gliedern eines Programmes in Overlays ist aufwendig und zeitintensiv. Virtuelle Speicherverwaltungen machen das Arbeiten mit Overlays überflüssig.

Schutz in Einbenützer Systemen

In einem Einbenützer System mit zusammenhängendem Speicher hat der Benutzer volle Kontrolle über den gesamten Hauptspeicher. Damit kann das Betriebssystem jederzeit zerstört werden. Ein Schutz in einem Einbenützer System ist möglich durch den Einsatz eines 'Grenzregisters' das den Beginn des Benützerspeichers anzeigt (Anfang der Segmentierung). Der Aufruf von Betriebssystem-Funktionen kann über spezielle Unterbrechungen (sog. supervisor calls) geschehen.

8.1.6 Multiprogramming mit festen Partitions

Mit Multiprogramming kann die Zentraleinheit besser ausgenützt werden. Um den maximalen Nutzen aus dem Multiprogramming zu ziehen, müssen mehrere Jobs gleichzeitig im Hauptspeicher vorhanden sein. Sobald ein Job E/A verlangt, wird die CPU sofort zum nächsten Job weitergegeben. Im allgemeinen verlangt ein Multiprogramming Betrieb mehr Speicher als ein Einbenützer System.

Absolutes Laden

In den ersten Systemen war der Hauptspeicher unterteilt in eine Anzahl fester Partitions. Jede Partition konnte einen Job aufnehmen. Die einzelnen

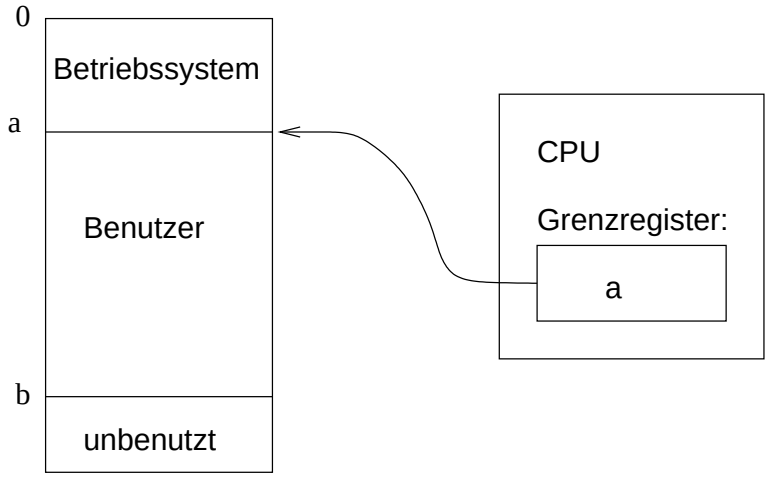


Abbildung 31: Einbenutzer Systeme

Programme wurden aufbereitet zur Ausführung in einer spezifischen Partition. Falls zur Zeit der Ausführung die Partition belegt war, musste der Job warten auch wenn eine andere Partition frei war.

Relozierbares Laden

Bei diesen Systemen kann jeder Job in irgend eine freie Partition geladen werden. Mit einem speziellen Ladeprogramm, das die Programme reloziert, werden die Jobs jeweils geladen.

Speicher-Verwaltungen mit festen Partitions nützen, bei statistischem Arbeitsanfall, den vorhandenen Speicher im allgemeinen schlecht aus. Solche Verwaltungen werden aber heute noch in Prozessrechnern verwendet.

Schutz in Multiprogramming Systemen: Mit 'Grenzregistern' kann überwacht werden, dass kein Job seinen Adressbereich überschreitet.

Fragmentierung: Dies tritt bei allen Speicher-Verwaltungen auf. In Systemen mit festen Partitions entsteht diese, da die vorhandenen Programme den Speicher der Partition nicht ausnützen.

8.1.7 Multiprogramming mit variabler Partitiongrösse

Der vorhandene Speicher kann mit variabler Partitiongrösse besser ausgenützt werden. Die einzelne Partition muss weiterhin zusammenhängend sein. Es gibt keine Beschränkung der Grösse eines Jobs, ausser dass dieser die Grösse des vorhandenen Hauptspeichers nicht übersteigen darf. Im Prinzip gibt es hier keine Speicher-Verschwendung, da jeder Job nur soviel Speicher bekommt wie er braucht. Nach dem Abschluss hinterlassen die Jobs Löcher im Hauptspeicher. Diese können von anderen Jobs wieder verwendet werden. Mit der Zeit können aber die Löcher immer kleiner und immer zahlreicher werden. Dabei kann es vorkommen, dass kein Platz vorhanden ist für einen bestimmten Job, obwohl die Summe der vorhandenen Löcher den Job aufnehmen könnte.

8.1.8 Kompaktifizierung

Benachbarte Löcher können einfach zu einem Loch zusammengefasst werden. Falls aber zu viele isolierte Löcher vorhanden sind kann man die vorhandenen Jobs zusammenschieben, sodass ein einziges Loch entsteht (=Kompaktifizierung). Im englischen bezeichnet man dies oft auch als 'garbage collection'.

Nachteile der Kompaktifizierung:

- Braucht Betriebsmittel die der Produktion entzogen werden.
- Während der Kompaktifizierung werden alle Jobs angehalten.
- Kompaktifizierung bedeutet ebenfalls Relokation, dazu wird Information benötigt die normalerweise nicht speicherresident ist. → bei guten Platzierungsstrategien muss erst kompaktifiziert werden, falls der Speicher ohnehin fast voll ist.

8.1.9 Positionierungsstrategie

Diese Strategie bestimmt, wo im Hauptspeicher neue Programme und Daten positioniert werden. Die bekanntesten dieser Strategien sind:

best fit Auswahl des kleinsten Lochs. Diese Strategie lässt grosse Löcher lange bestehen, während auf der anderen Seite eine Vielzahl kleiner, unbrauchbarer Überreste entsteht (Besteuerung der Armen).

worst fit Auswahl des grössten Lochs. Dieses Verfahren bringt alle Löcher auf dieselbe Grösse. Das einzelne Loch kann zu klein sein zur Aufnahme eines neuen Segmentes (Besteuerung der Reichen).

first fit Auswahl des ersten hinreichend grossen Lochs. Dieses Verfahren ist das effizienteste. Damit die Löcher gleichmässig im Speicher verteilt werden, empfiehlt es sich den Speicher ringförmig zu organisieren und jede neue Suche beim Ziel des vorangehenden zu beginnen.

8.1.10 Multiprogramming mit Swapping

In den bisherigen Systemen sind die Programme im Speicher verblieben bis ihre Ausführung beendet war. Bei einem System mit Swapping sind nur eine beschränkte Zahl von Jobs gleichzeitig im Hauptspeicher möglich. Sobald ein Job nicht weiter lauffähig ist, gibt den Speicher frei und wird vom Betriebssystem ausgelagert (swapped out). Das Betriebssystem lädt dann den nächsten Job in den Speicher (swapped in). Normalerweise wird ein Job viele Mal ein- und ausgelagert bis er beendet ist. Swapping Systeme sind heute durch Paging Systeme ersetzt worden.

8.2 Virtueller Speicher

Mit dem Begriff 'virtueller Speicher' wird normalerweise die Fähigkeit verbunden mehr Speicher zu adressieren als physisch vorhanden ist. Die beiden bekanntesten Methoden bei der Implementierung von virtuellem Speicher sind: Paging und Segmentierung.

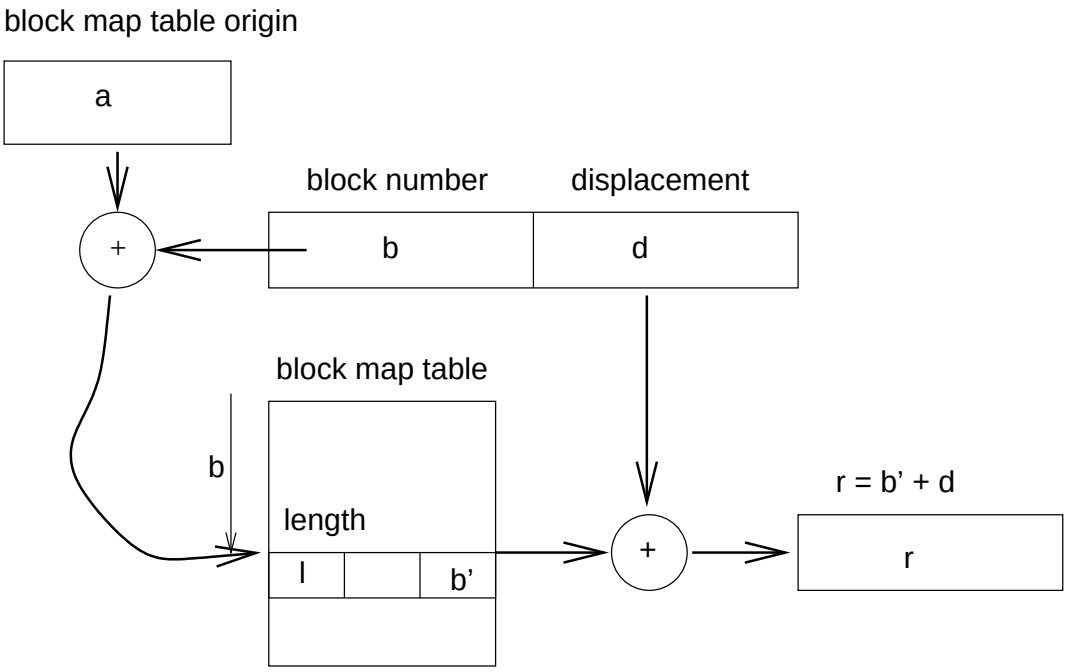


Abbildung 32: Blockweises Falten

8.2.1 Grundlegende Konzepte

Der physikalische Aufbau des virtuellen Speichers besteht normalerweise aus der zweistufigen Hierarchie von Haupt- und Massenspeicher. Der wesentliche Aspekt ist die unterschiedliche Zugriffszeit von ca. 60ns beim Haupt- und 10 ms beim Massenspeicher. Wichtig für die nachfolgenden Betrachtungen ist, dass das Verhältnis der Zugriffszeiten $> 10^4$ (=gross) ist. Man unterscheidet den physischen oder realen Adressraum R und den logischen oder virtuellen Adressraum V. Das Benutzerprogramm arbeitet im logischen Adressraum. Bei der Ausführung des Programmes wird der logische Adressraum auf den physischen gefaltet.

8.2.2 Blockweises Falten

Zum Falten des virtuellen auf den realen Hauptspeicher braucht man Tabellen. Würde dieses Falten wortweise durchgeführt, so wären diese Tabellen so gross wie der logische Adressraum. Der Speicher wird in Blöcke aufgeteilt. Die Speicherverwaltung ist jederzeit auf dem laufenden wo im realen Hauptspeicher welche virtuellen Blöcke sind. Falls alle Blöcke gleich gross sind spricht man von Paging, andernfalls von Segmentierung. Gewisse Systeme kombinieren diese beiden Techniken.

8.2.3 Paging

Haupt- und Massenspeicher werden in Blöcke gleicher Grösse unterteilt. Eine virtuelle Adresse V besteht aus einem geordneten Paar (p,d), bei dem p die Seitennummer des virtuellen Adressraumes und d das Displacement innerhalb der Page darstellt.

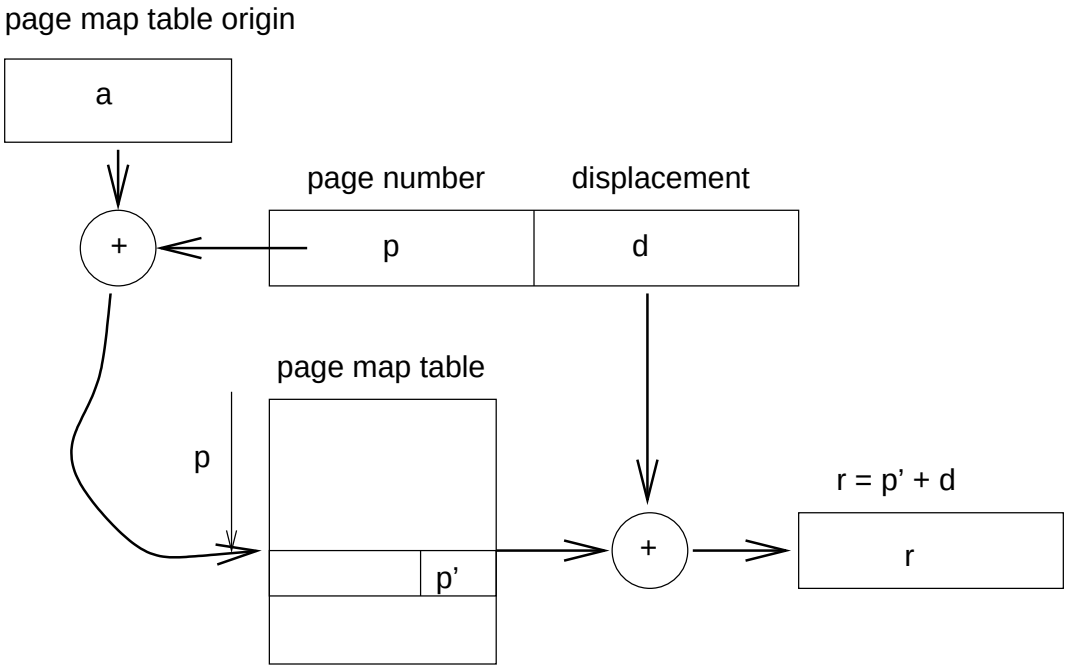


Abbildung 33: Paging

a) Direktes Falten

In der Pagetabelle wird nun beim Displacement p nachgesehen ob diese Page im Hauptspeicher ist; falls ja wird die reale Seitennummer p' ausgelesen und zur Adressierung verwendet. Andernfalls wird ein Page-Fault generiert und die entsprechende Seite geladen.

Die Länge der Seiten ist normalerweise eine Zweierpotenz.

b) Assoziatives Falten

Eine andere Methode ist, die ganze Page-Map in einen Assoziativ- Speicher zu legen, dessen Zugriffszeit eine Grössenordnung geringer ist als die des Hauptspeichers. Damit kann die Zugriffszeit auf den Hauptspeicher verkleinert werden. Diese Lösung ist im allgemeinen heute noch zu teuer. Bei verschiedenen Rechner-Architekturen wird heute häufig ein gemischtes direktes und assoziatives Falten durchgeführt. Dabei befinden sich, die physischen Adressen der am häufigsten verwendeten Seiten im Assoziativ-Speicher.

8.2.4 Segmentierung

Aus dem Kapitel 'Realer Hauptspeicher' wissen wir, dass Programme mit den Strategien 'first fit', 'worst fit' und 'best fit' platziert werden können bei einer Speicherverwaltung mit variabler Partitiongrösse. Diese Strategien sind notwendig, falls der Speicher zusammenhängend sein muss. Bei der Segmentierung lassen wir diese Beschränkung fallen. Ein Programm darf gleichzeitig viele separate Blöcke, unterschiedlicher Grösse, des Hauptspeichers belegen. Damit entstehen einige interessante Probleme: der Schutz des Programmes vor anderen Benutzern, oder die

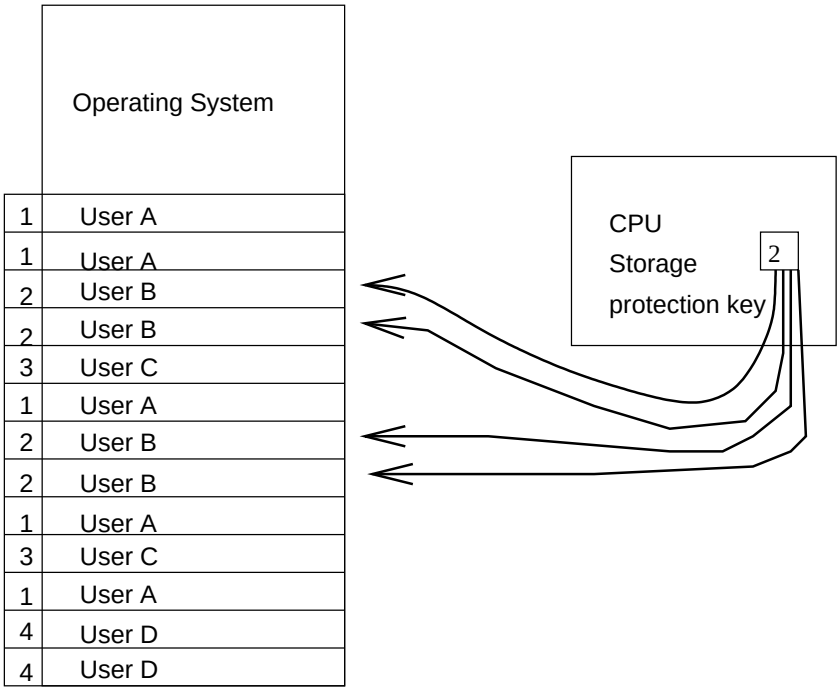


Abbildung 34: Speicherschutz

Zugriffs-Begrenzung eines Programmes. Eine Möglichkeit diese Probleme zu lösen sind sog. 'Speicherschutz Schlüssel' (storage protection keys).

Die virtuelle Adresse in einem segmentierten System wird mit V = (s,d) bezeichnet. Die Umsetzung in die physische Adresse zeigt untenstehende Figur:
Zugriffsschutz im segmentierten System

Es ist unvernünftig jedem Programm unbegrenzten Zugriff zum Speicher zu ermöglichen. Einer der attraktiven Vorteile der segmentierten Speicher ist der Zugriffsschutz. Man unterscheidet:

Zugriffstyp	Abkürzung	Erklärung
Read	r	Dieser Block darf gelesen werden
Write	w	Dieser Block darf verändert werden.
Execute	e	Dieser Block darf ausgeführt werden.
Append	a	An diesen Block darf Information angehängt werden.

Jeder dieser 4 Typen kann für jedes Segment spezifiziert werden. Dabei gibt es selbstverständlich Kombinationen die keinen Sinn machen: nur Write und Execute, nur Write.

Ein typischer Eintrag in der Segmenttabelle könnte wie folgt aussehen:

wobei:

v= 0:	Segment nicht im Hauptspeicher
v= 1:	Segment im Hauptspeicher
a=	Adresse des sekundär Speichers
l=	Länge des Segmentes
r w e a:	Zugriffs-Steuerbits
s'=	reale Basis Adresse des Segmentes

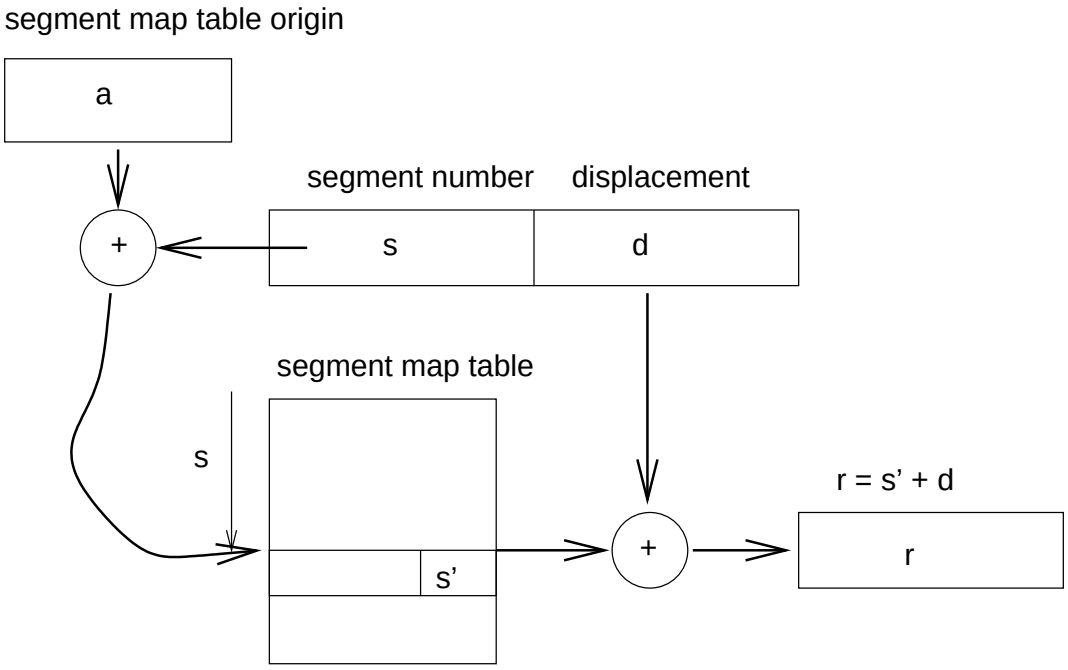


Abbildung 35: Segmentierung

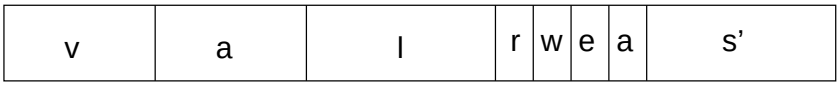


Abbildung 36: Segment Tabelleneintrag

8.2.5 Paging und Segmentierung kombiniert

Damit werden die Vorteile der beiden Techniken kombiniert. Die Adressierung ist 3-Dimensional: V= (s,p,d)

Dieser Übersetzungs-Prozess von der virtuellen zur realen Adresse, geht davon aus, dass alle Information genau dort ist wo diese hingehört. Viele Fehler sind aber möglich:

- Das Segment könnte nicht im Hauptspeicher sein: → Missing Segment Fault: Das Betriebssystem lokalisiert das Segment auf dem Sekundär-Speicher, generiert eine Page Tabelle für das Segment und lädt das Segment.
- Die Page könnte nicht im Hauptspeicher sein: → Missing Page Fault!
- Die Adresse könnte ausserhalb des Segmentes liegen: →Segment Overflow
- Es könnte ein Zugriff verlangt sein der nicht erlaubt ist: → Protection Fault

Siehe Abbildung 37.

8.3 Virtuelle Speicherverwaltung

In diesem Kapitel werden die Auswirkungen von verschiedenen:

- Lade-Strategien (fetch strategies)
- Platzierungs-Strategien (placement strategies)
- Ersetzungs-Strategien (replacement strategies) diskutiert.

8.3.1 Einfluss der Ersetzungsstrategie

Üblicherweise sind in einem mit Paging arbeitenden Rechnersystem zu jeder Zeit alle vorhanden Seiten belegt. Die Speicherverwaltung muss dann entscheiden, wo im Hauptspeicher Platz gemacht wird für eine neue Seite. Zur formalen Beschreibung ist es zweckmässig folgende Grössen einzuführen:

Vorwärts-Abstand: d(x,t) einer Seite x zur Zeit t, ist der zeitliche Abstand zum nächsten Zugriff auf x.

Rückwärts-Abstand: b(x,t) einer Seite x zur Zeit t, ist der zeitliche Abstand zum letzten Zugriff auf x.

Belady's optimal algorithm

Die ausgewählte Seite hat den grössten Vorwärts-Abstand. Dieser Algorithmus verlangt Kenntnis der zukünftigen Referenzen. Aus diesem Grunde gilt dieser Algorithmus als nicht realisierbar. Man benützt diesen Algorithmus als Mass für die Güte aller Ersetzungsstrategien, da er das Vorkommen von Seiten-Fehlern minimalisiert. In der Praxis wird oft versucht, den Vorwärts-Abstand durch den Erwartungswert des Vorwärts-Abstands zu ersetzen.
Ersetzungsstrategie = FIFO

In diesem Falle erhält jede Seite beim Laden in den Hauptspeicher einen Zeitstempel. Bei diesem Algorithmus wird diejenige Seite ausgewählt, die am längsten

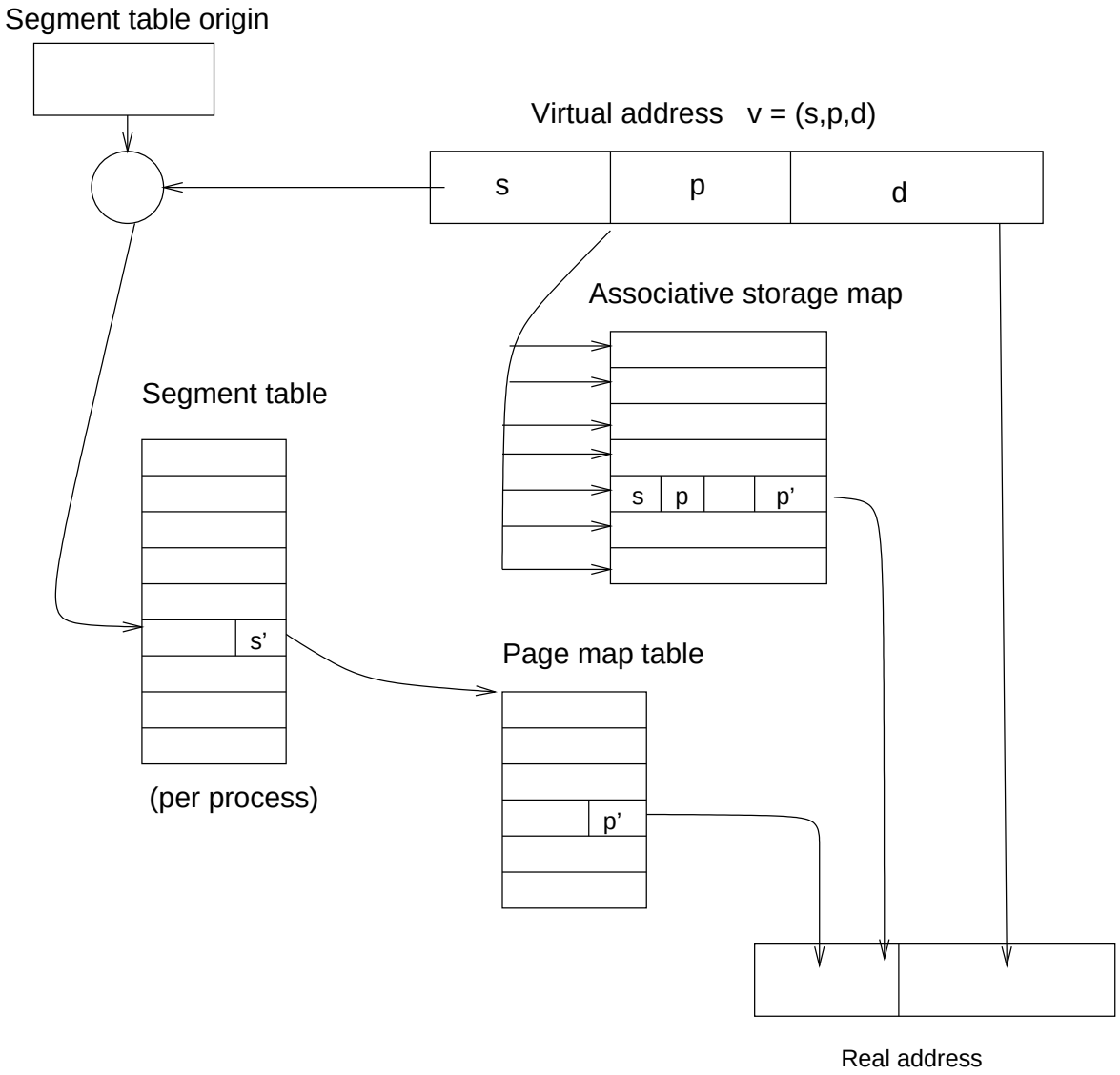


Abbildung 37: Segmentierung und Paging

im Hauptspeicher. ist. Auf den ersten Blick scheint diese Strategie brauchbar. Unglücklicherweise werden dabei auch stark benützte Seiten ersetzt. In einem grossen Teilnehmer-System ist es üblich, dass 'Shared Code' verwendet wird. Bei Verwendung der FIFO Strategie werden auch solche 'Shared Pages' ersetzt, dabei entstehen unnötige Seiten-Fehler. Neben diesem Nachteil besteht bei FIFO noch das Erweiterungs-Problem: Von einem gutartigen Paging Algorithmus darf erwartet werden, dass die Anzahl der Seiten-Fehler abnimmt, falls der Hauptspeicher vergrössert wird. Folgendes Beispiel zeigt, dass dies bei FIFO nicht unbedingt der Fall ist:

- Ein Adressraum bestehe aus den Seiten: (1,2,3,4,5)
- Die Bedarfs-Reihenfolge sei: (1,2,3,4,1,2,5,1,2,3,4,5)

a) Der Speicherraum bestehe aus 3 Seiten:

   Es ergeben sich folgende Speicher-Zustände:

```
1    1    1    2    3    4    1    1*   1*   2    5    5*
     2    2    3    4    1    2    2    2    5    3    3
          3    4    1    2    5    5    5    3    4    4
```

b) Der Speicherraum bestehe aus 4 Seiten:

```
1    1    1    1    1*   1*   2    3    4    5    1    2
     2    2    2    2    2    3    4    5    1    2    3
          3    3    3    3    4    5    1    2    3    4
               4    4    4    5    1    2    3    4    5
```

Ersetzungsstrategie = Least-Recently-Used (LRU)

Bei dieser Strategie versucht man den Vorwärts-Abstand zu approximieren durch: E(d(x,t)) = b(x,t).
Man stützt sich auf der heuristischen Überlegung ab, dass die unmittelbare Vergangenheit eine gute Grundlage ist zum abschätzen der Zukunft ist. LRU verlangt, dass für jede Seite eine Zeitangabe vorhanden ist, die bei jedem Zugriff nachgeführt werden muss. Dies erfordert einen beachtlichen Aufwand. Aus diesem Grunde wird LRU relativ selten implementiert. Bei heuristischen Überlegungen in Betriebssystemen muss man vorsichtig sein. Die LRU-Page könnte diejenige Seite sein auf die der nächste Zugriff erfolgt (bei einer grossen Schleife).
Ersetzungsstrategie = Least-Frequently-Used (LFU)

Eine Approximation von LRU ist die LFU Strategie. Man befasst sich mit der Häufigkeit des Seitenzugriffs. Auch hier, wie bei LRU, ist die Gefahr von falschen Entscheidungen gross: Die zuletzt geladene Seite ist wahrscheinlich die am wenigsten häufig verwendete Seite!
Ersetzungsstrategie = Not-Used-Recently (NUR)

Auch diese Strategie approximiert LRU. NUR ist eine Strategie die recht einfach zu implementieren ist. Für jede Page werden zwei zusätzliche Bits verwendet:

a) Reference-Bit: 0 = Page wurde nicht referiert 1 = Page wurde referiert.

b) Modified-Bit: 0 = Page wurde nicht modifiziert 1 = Page wurde modifiziert.

Das modified-Bit wird oft auch 'dirty-Bit' genannt. Die NUR Strategie arbeitet wie folgt:
Zu Beginn werden alle Reference-Bit zu Null gesetzt. Sobald eine Referenz auf eine Page erfolgt, wird das entsprechende Bit 1 gesetzt. Zu Beginn sind alle Modified-Bit zu Null gesetzt. Sobald ein Wort in einer Page geändert wird, wird das Modified-Bit zu eins gesetzt. Falls eine zu ersetzende Page ausgewählt werden soll, so wird eine Seite gesucht mit dem Reference-Bit = 0. Nach Möglichkeit sollte auch das Modified Bit = 0 sein (das Ersetzen einer ungeänderten Page erfordert kein Zurückschreiben der Page auf den Massenspeicher). Da aber der Hauptspeicher im allgemeinen recht intensiv benützt wird, so werden früher oder später alle Reference-Bit gesetzt sein. Damit verlieren wir die Möglichkeit die gewünschte Seite zu ersetzen. Um das zu umgehen, werden die Reference-Bit periodisch rückgesetzt. Bei NUR teilen wir die vorhandenen Pages in vier Kategorien ein:

a) unreferenziert, unmodifiziert

b) unreferenziert, modifiziert

c) referenziert, unmodifiziert

d) referenziert, modifiziert

Bei der Suche nach einer zu ersetzenden Seite geht man die Kategorien in der Reihenfolge a, b, c, d durch.

8.4 Lokalität

Programme neigen dazu den Speicher in nicht uniformer Weise zu referenzieren. Man bezeichnet dieses Verhalten als Lokalität. Ein Programm beschränkt dabei seine Zugriffe während eines gegebenen Zeitabschnitts auf nur eine Teilmenge aller Seiten seines Adressraumes. Es ist bemerkenswert, dass ein Programm seine Lokalität während längerer Zeit nicht oder nur wenig ändert. Starke Änderungen bleiben auf punktuelle Phasen-Übergänge beschränkt.
Man unterscheidet zwischen zeitlicher und räumlicher Lokalität. Zeitliche Lokalität bedeutet, dass eine einmal referierte Speicherzelle mit grosser Wahrscheinlichkeit in nächster Zukunft erneut referiert wird. Die Ursache sind:

- Schleifen
- Unterprogramme
- Stacks
- Zähl- und Summenvariablen.

Räumliche Lokalität bedeutet, dass Speicher-Zugriffe dazu tendieren immer die gleichen Speicherzellen oder in der Nähe liegende Speicherzellen zu referieren. Die Ursachen sind:

- Array Zugriffe
- Ausführung von sequentiellem Code
- Die Gewohnheit der Programmierer Variablen zu gruppieren.

Die signifikanteste Aussage der Lokalität ist, dass ein Programm effizient ablaufen kann solange sein favorisiertes Subset der Pages sich im Hauptspeicher befindet. Falls die Anzahl der zur Verfügung stehender Seiten um eine gewisses Mass verkleinert wird, so ändert die Page-Fault Rate in einem bestimmten Zeitintervall nur

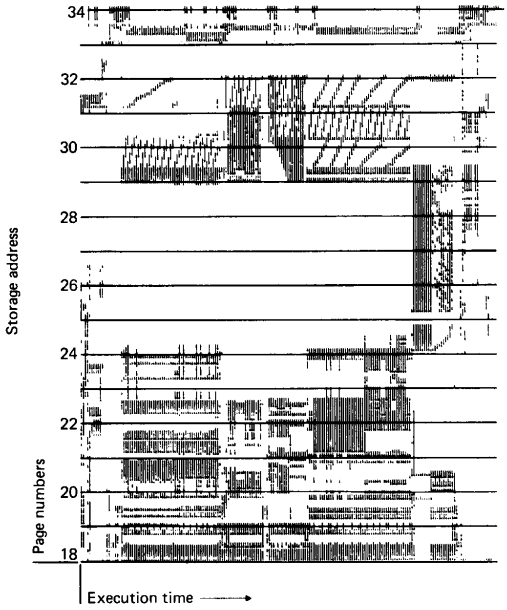


Fig. 9.2 Storage reference pattern exhibiting locality. (Reprint by permission from *IBM Systems Journal*. © 1971 by International Business Machines Corporation.)

Abbildung 38: Lokalität

unwesentlich. Wird aber die Anzahl der zur Verfügung stehenden Pages über ein bestimmtes Mass reduziert, so nimmt die Anzahl der Page-Faults dramatisch zu.

8.5 Das Working Set

Denning hat eine Theorie über das Verhalten von Paging Systemen aufgestellt und diese 'Working Set' Theorie über das Programmverhalten genannt. Das 'Working Set' eines Programmes ist derjenige Satz von Pages die Speicherresident sein müssen für die effiziente Ausführung des Programmes. Falls das 'Working Set' eines Programmes nicht resident gehalten werden kann so kann dies zum Seitenflattern (Thrashing) führen. Die 'Working Set' Speicherverwaltungs-Strategie versucht das 'Working Set' aller aktiven Prozesse im Speicher zu halten. Die Entscheidung ob ein neuer Prozess aktiviert werden kann, hängt davon ab, ob genügend Hauptspeicher für sein 'Working Set' vorhanden ist. Diese Entscheidung wird oft mit heuristischen Mitteln durchgeführt, da genaue Angaben über das 'Working Set' neuer Prozesse fehlen. Das Working Set eines Prozesses W(t,w) besteht aus einem Satz von Seiten die der Prozess zwischen dem Zeitpunkt t-w bis t benützt. Die Variable w ist die Fenstergrösse des 'Working Set'. Die Bestimmung von w ist kritisch, je grösser w desto mehr Hauptspeicher ist notwendig.

Das 'Working Set' eines Prozesses ändert während seiner Ausführung. Eine gewisse Zeit bleibt das WS stabil, dann kommt eine Phase des Übergangs von einem Working Set auf ein anderes usw (siehe Abbildung 41).

Das Hauptproblem ist das Festsetzen der Fenstergrösse w. Denning schlägt vor

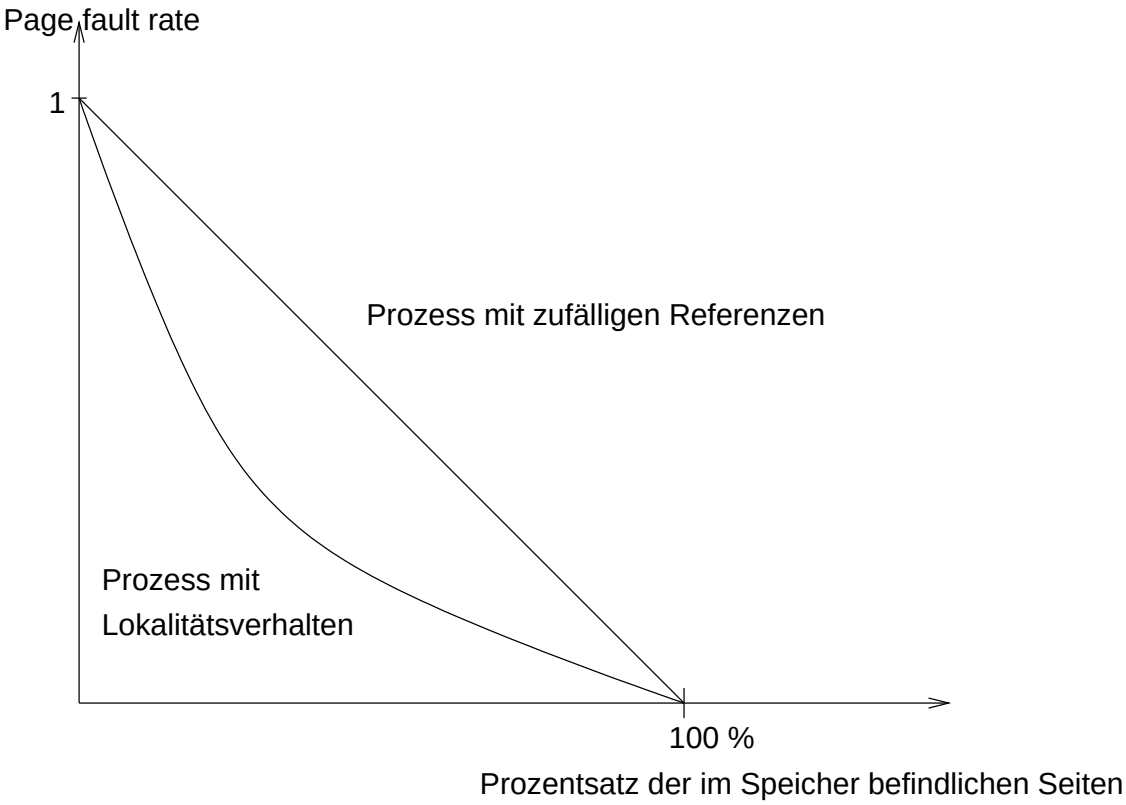


Abbildung 39: Page Faults

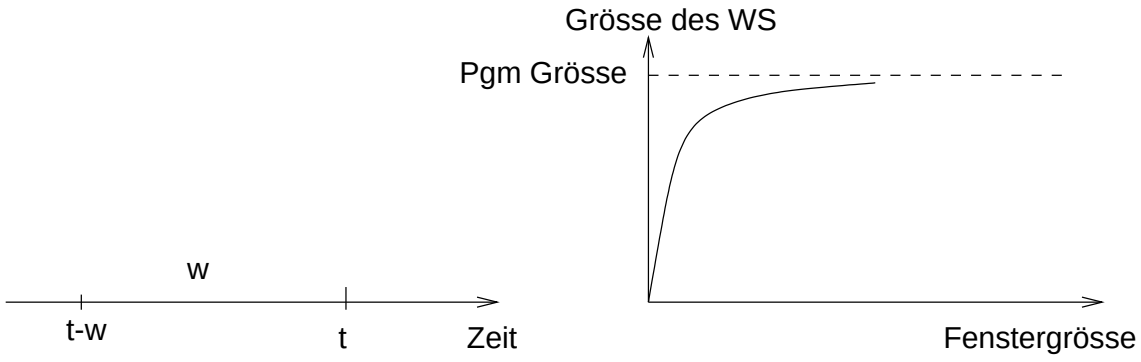


Abbildung 40: Working Set

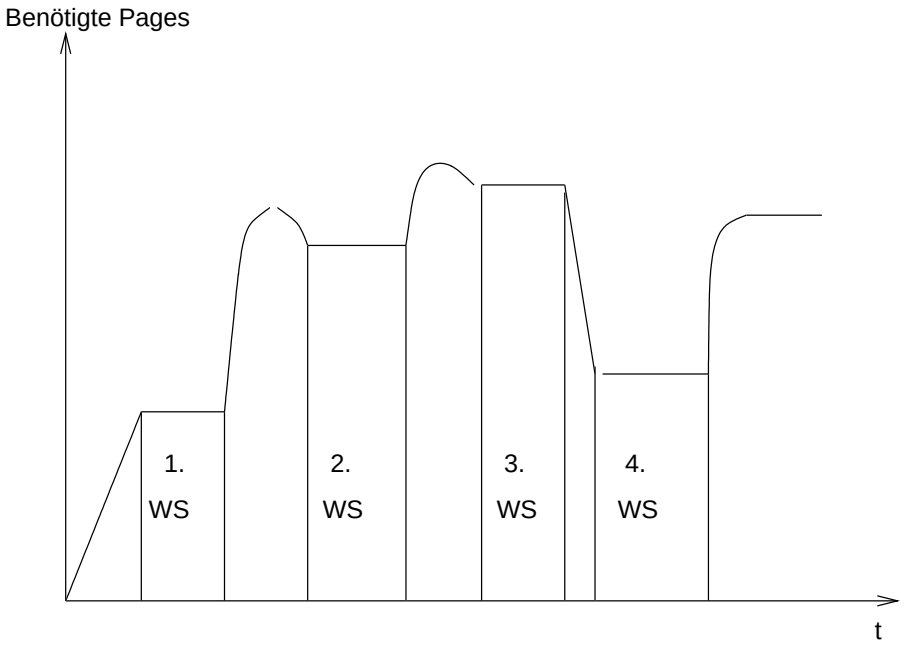


Abbildung 41: Aenderung des Working

diese so zu wählen, dass diese der doppelten Übertragungszeit auf den Massenspeicher entspricht.

8.6 Demand Paging

Normalerweise werden Pages nur auf Verlangen geladen. Dies geschieht aus folgenden Gründen:

- Der Programmfluss kann nie genau vorausgesagt werden.
- Bei Demand Paging werden keine überflüssigen Pages geladen.
- Die Entscheidung welche Pages zusätzlich zu laden sind verursacht Verwaltungsaufwand.

Demand Paging ist nicht ohne Nachteile. Jedes Mal wenn eine neue Page referiert wird muss der Prozess bis zum Laden der Page warten. In der untenstehenden Figur wird das Raum-Zeit Produkt eines Programmes dargestellt. Eines der Ziele bei einem Speicherverwaltungs-Entwurf ist ein möglichst kleines Raum-Zeit Produkt für die Prozesse (siehe Abbildung 42)

8.7 Anticipatory Paging

Die Hardware wurde in den letzten Jahren erheblich billiger. Der relative Wert von Rechnerzeit zu Anwenderzeit wurde in den letzten Jahren erheblich reduziert. Der Entwerfer eines Betriebssystems versucht heute vorwiegend die Wartezeit des Anwenders zu reduzieren. Mit Anticipatory Paging kann dieses Ziel erreicht werden. Bei Anticipatory Paging versucht das Betriebssystem die in Zukunft benötigten Pages vorausschauend zu laden. Falls die richtigen Pages geladen werden kann die

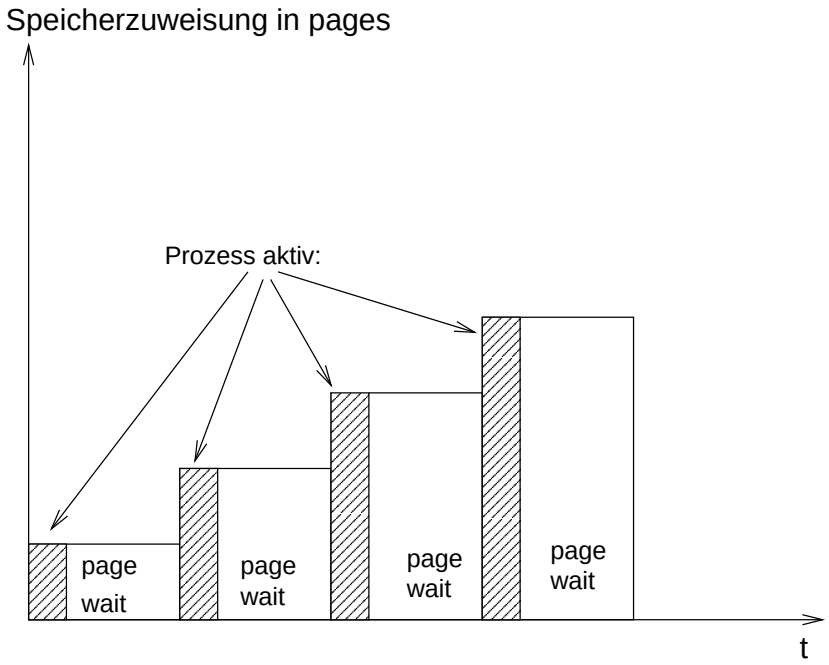


Abbildung 42: Raum Zeit Produkt

totale Ausführungszeit eines Programmes erheblich reduziert werden. Folgende Vorteile ergeben sich mit Anticipatory Paging:

- Bei richtigen Entscheidungen wird die Ausführungszeit eines Prozesses erheblich reduziert.
- Falls die Entscheidung mit wenig Aufwand implementiert werden kann, kann der Gesamt-Durchsatz durch die Maschine erhöht werden.
- Mit immer billiger werdender Hardware kann man sich mehr Hauptspeicher leisten, die Auswirkungen von teilweise falschen Entscheidungen sind damit kleiner.

8.8 Page Release

Unter einer 'Working Set' Speicherverwaltung geben die Programme die referierte Page explizite an. Nicht mehr verwendete Seiten verbleiben eine gewisse Zeit im Hauptspeicher. Sobald eine Seite nicht mehr benützt wird, könnte der Benutzer diese freigeben. Damit könnte der vorhandene Speicher besser genutzt werden. Zukünftige Compiler sollten diese Freigaben durchführen.

8.9 Einfluss der Seitengrösse

Alle Systeme mit virtuellem Speicher lassen einen Teil des zugeteilten Adressraumes ungenützt. Bei einem System mit Paging ist ein Teil der letzten Seite unbenutzt (siehe Abbildung 43).

Die obenstehende Abbildung legt nahe, die Seitengrösse möglichst klein zu halten. Bei einer Seitengrösse von 1 Wort ist kein Verschnitt mehr vorhanden! Die Gründe die gegen kleine Seiten sprechen sind:

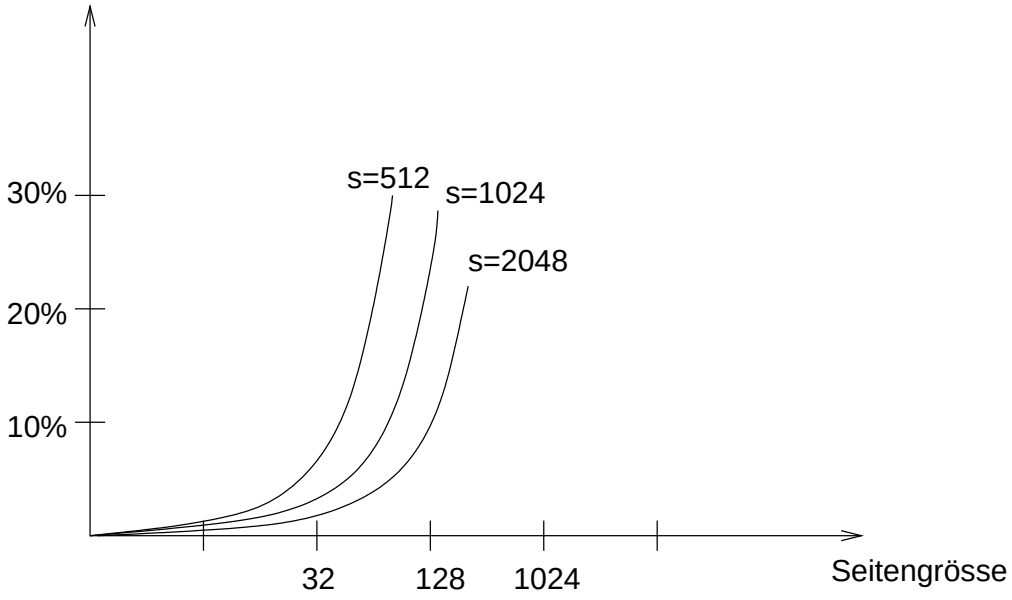


Abbildung 43: Einfluss der Seitengrösse

- Platz-Aufwand für Page-Tabelle
- Aufwand für Seiten Ein- und Auslagern.

Der Overhead für den Seiten-Transport sinkt mit wachsender Seitengrösse. Wir müssen die Seitengrösse derart wählen, dass ein günstiger Kompromiss zwischen Fragmentierung (Raum-Effizienz) und Transport-Overhead (Zeit-Effizienz) gefunden werden kann.

8.10 Linux Speicherverwaltung

8.10.1 Realer Speicher

Der physische Speicher unter Linux ist gegliedert in *nodes* und *zones* ein NUMA (Non Uniform Memory Architecture) hat mehrere nodes, ein x86 System nur einen node. Beim x86 unterscheidet man die folgenden *zones*

ZONE_DMA: First 16MB of memory

ZONE_NORMAL: 16MB - 896MB

ZONE_HIGHMEM: 896 MB - End

Die meisten Betriebssystem Operationen können nur in der *ZONE_NORMAL* durchgeführt werden. Um den Speicher in *ZONE_HIGHMEM* verwenden zu können müssen Teile davon in die *ZONE_NORMAL* gefaltet werden.
Jede Zone hat drei Niveaus (Watermarks) die vom System überwacht werden:

pages_low Beim Erreichen dieser Marke wird der *kspwapd* synchron aktiviert um Seiten auszulagern.

pages_min Beim Erreichen dieser Marke wird der *kswapd* gestartet. Normalerweise ist *pages_min* doppelt so gross wie *pages_low*.

pages_high Der kpaged wird erst deaktiviert falls *pages_high* Seiten frei sind. Normalerweise ist *pages_high* dreimal so gross wie *pages_low*.

8.10.2 Verwaltung des realen Speichers

Der Speicher wird mit dem *Binary Buddy Allocator* vorgenommen. Dabei wird der Speicher so unterteilt, dass nur Speicherblockgrössen einer ganzen Zweierpotenz entstehen. Speicher kann also nur vergeben werden in der Grösse von 131072, 65536, 32768, 16384, 8192, usw Bytes (siehe Abbildung 44). Dadurch werden die Probleme Fragmentierung minimiert und die Effizienz der Speichervergabe und Verwendung verbessert gegenüber anderen Strategien. Zusätzlich werden die bereits vergebenen Speicherblöcke mit einer Cacheverwaltung bewirtschaftet.

8.10.3 Virtual Memory

Linux hat eine virtuelle Speicherverwaltung. Die Seitengrösse beträgt 4kByte beim x86. Grundsätzlich hat Linux eine dreistufige Speicherverwaltung. Falls man allerdings nur mit 4GB logischem Adressraum arbeitet wird fällt eine der Page Tables weg und es verbleibt eine zweistufige Verwaltung (siehe Abbildung 45).

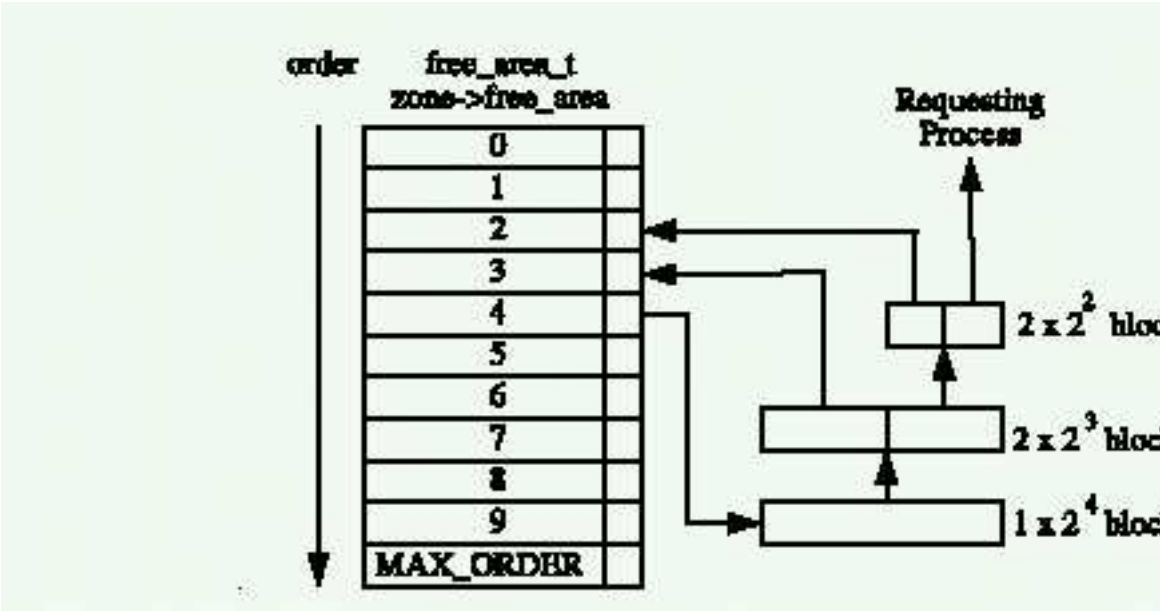
Der virtuelle Adressraum ist unterteilt in den *user space* und *system space*. In einem x86 System sind die ersten 3 GB für den *user space* reserviert. Die genaue Aufteilung ist in der Abbildung 46 dargestellt.

8.10.4 Verwaltung des Adressraums

Damit die Speicherwaltung effizient ist genügen die *page tables* allein nicht. Für jedem Prozess stehen wie bereits gesagt 3 GB Adressraum zur Verfügung. Jeder Prozess hat also seinen eigenen Adressraum. Zur genaueren Beschreibung seines Adressraums gibt es pro Prozess die `mm_struct`. Darin ist vereinfachend folgendes enthalten:

- Zeiger auf das pgd (Page Global Directory)
- Anfang und Ende des Codes
- Anfang und Ende der Daten
- Anfang und Ende des Stacks
- Anfang und Ende des Heaps
- Zeiger auf eine verkettete Liste mit allen *memory regions*

Normalerweise braucht aber ein Prozess zusätzliche Daten: zum Beispiel Daten aus Dateien, dynamische Daten, Zugriff auf Netzwerk Puffer usw. Meistens sind diese Daten verstreut irgendwo im virtuellen Adressraum. Zur Verwaltung dieser Daten gibt es in Linux die *memory regions*. Im Adressraum eines Prozesses gibt es normalerweise mehrere *memory regions*. der Vorteil der *memory regions* neben den kleineren *page tables* ist auch , dass diese von mehreren Prozessen benutzt werden können und daher mit einer speziellen Verwaltung in einem Cache gehalten werden (siehe *slab*).


order
free_area_t
zone->free_area
0
1
2
3
4
5
6
7
8
9
MAX_ORDER
Requesting
Process
2 x 2^2 bloc
2 x 2^3 bloc
1 x 2^4 bloc

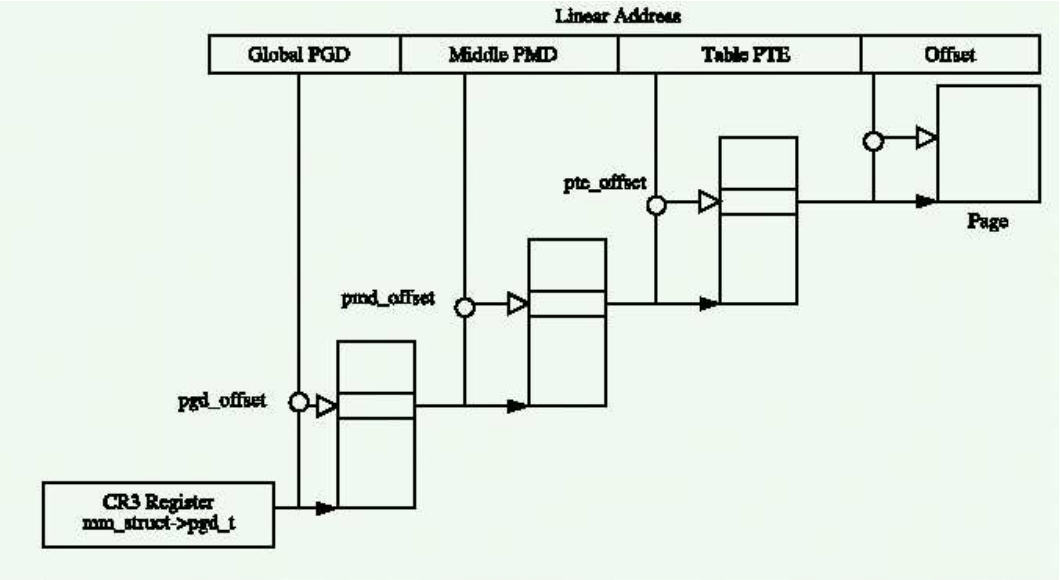


Abbildung 45: Linux Page Tables

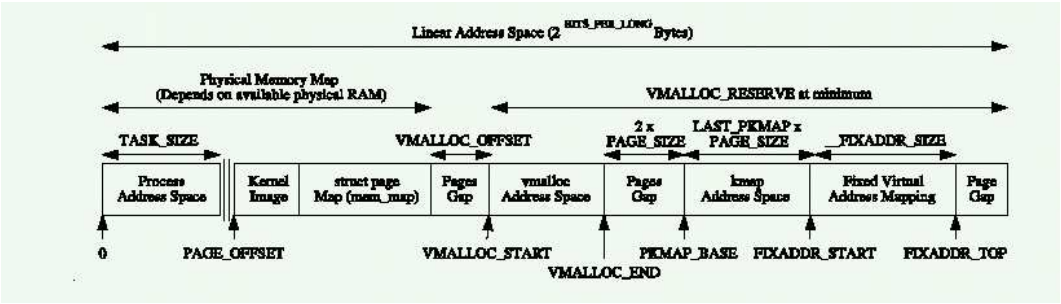


Abbildung 46: Linux virtual address assignment

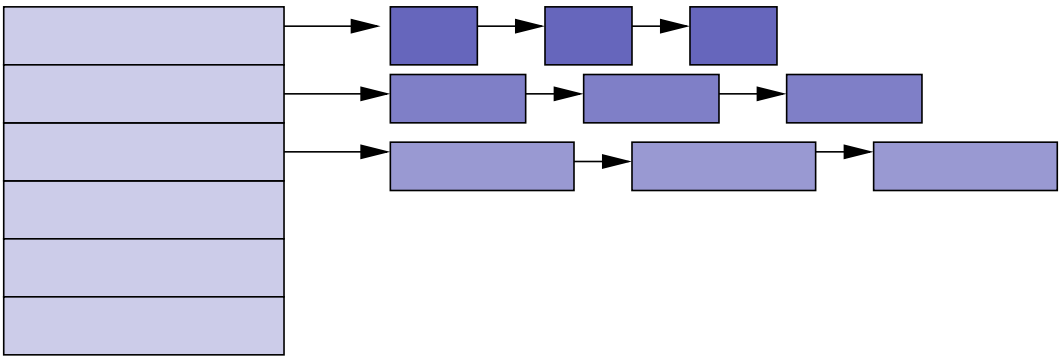


Abbildung 47: Slab Allocator

8.10.5 Slab Allocator

Die Idee des *slab* ist, dem System einen Cache von häufig verwendeten Objekten zur Verfügung zu halten. Dies ist in Linux ganz ähnlich wie in Solaris implementiert. Die einzelnen gleichen Datenstrukturen die mit dem Buddy Algorithmen alloziert wurden werden dabei jeweils mit einer doppelt verketteten Liste zusammengefasst (siehe Abbildung 47).

8.10.6 Page Cache

Alle Seiten die vom Disk gelesen wurden sind solange sie im Speicher sind im *page cache* aufgeführt. Damit soll unnötiges Lesen von der Festplatte vermieden werden. Das *page cache* ist unterteilt in die *active list* und die *inactive list*. Zuerst kommt eine Page immer in die *active list*. Von Zeit zu Zeit muss das Cache geschrumpft werden um Speicher zu sparen. Dabei werden all jene Seiten von der *active list* zur *inactive list* verschoben, deren *reference bit* nicht gesetzt ist. Gleichzeitig werden die *reference bit* zurückgesetzt.

8.10.7 Ersetzungsstrategie

Auf den ersten Blick ist die Ersetzungsstrategie bei Linux LRU. Das ist aber nicht ganz richtig weil das *page cache* nicht in LRU Manier organisiert ist. Die Idee hinter der *active list* ist dass diese die Working Sets der residenten Prozesse enthält. Zur

Verbesserung (Erweiterung) des Cache ist zusätzlich die *inactive list* vorhanden. Falls eine neue Seite benötigt wird so nimmt man diese vom Ende der *inactive list* (siehe Abbildung 48).

8.11 Shared Libraries (a.out)

Die effiziente Verwendung von Haupt- und Massenspeicher wird immer wichtiger. Neben den bereits besprochenen Speicherverwaltungstechniken des Betriebssystems spielen dabei die Shared Libraries eine wichtige Rolle.
Die richtige Verwendung von Shared Memory kann dazu führen, dass die a.out Datei weniger Platz auf dem Massenspeicher und bei der Ausführung auch weniger Hauptspeicher belegt. In vielen Rechnersystemen werden Shared Libraries unterstützt: VMS (shared images), UNIX System V usw.

8.11.1 Was ist eine Shared Library

Eine Shared Library ist eine Datei die Objektcode enthält, der von mehreren gleichzeitig ablaufenden a.out Dateien verwendet werden kann. Wenn ein Programm kompiliert und gelinkt wird mit einer Shared Library so wird der Code der Library nicht in die a.out Datei kopiert. Anstelle des Codes wird eine spezielle Sequenz die '.lib' heisst in den Code aufgenommen. Bei der Ausführung der a.out Datei wird mit der in der '.lib' Sequenz vorhandenen Information der Code der entsprechenden Shared Library in den Speicher geladen.
Die Implementierung hinter diesem Konzept ist eine Shared Library die aus zwei Teilen besteht:

- der 'host shared library'
- der 'target shared library'

Die 'host shared library' wird durch den Linker in die a.out Datei eingebunden. Die 'target shared library' wird durch die a.out Datei in den Hauptspeicher geladen falls diese nicht schon im Hauptspeicher ist. Die Verwendung von Shared Libraries offeriert verschiedene Vorteile:

- Einsparung von Massenspeicher, da die a.out Dateien kürzer sind.
- Einsparung von Hauptspeicher, da die Shared Library nur einmal geladen wird.
- Einfacher Unterhalt der a.out Dateien. Eine Fehlerkorrektur in der Shared Library wirkt sich sofort auf die Ausführung aller a.out Dateien aus ohne, dass ein Neulinken notwendig ist.

8.11.2 Speicherbelegung mit und ohne Shared Library

Unter UNIX System V kann eine a.out Datei erzeugt werden entweder durch Verwendung der sogenannten 'archive libraries' oder durch Verwendung der 'shared libraries'. Im ersten Fall werden die unaufgelösten externen Referenzen des Codes durch den Inhalt der entsprechenden Objektdatei der 'archive library' ergänzt. Der Objektcode besteht im allgemeinen aus einem Programmteil (.text) und einem Datenteil (.data). Der Programmteil wird in den .text Teil, der Datenteil in den .data Teil der a.out Datei übernommen.
Im Falle der Shared Library kopiert der Linker nur einen kleinen Teil der Shared Library in den .text und .data Bereich von a.out. In diesem Teil erfolgt die Initialisierung der importierten Symbole und das Laden/Falten der Shared Library in den

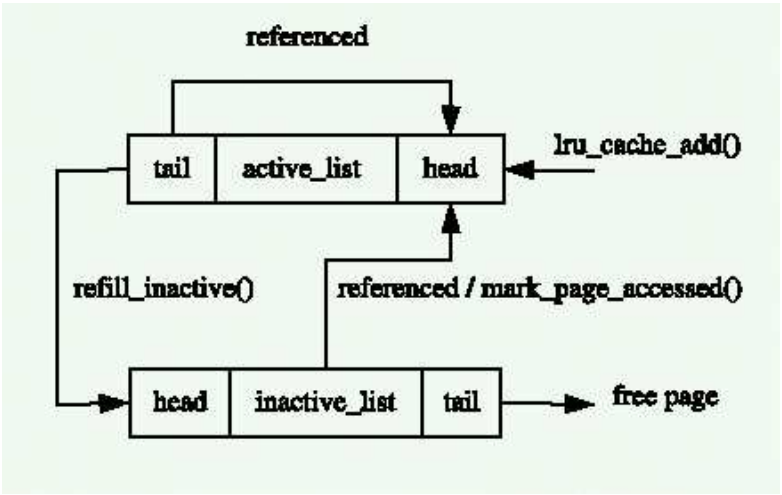


Abbildung 48: page cache

Adressraum von a.out (selbstverständlich werden auch die externen Referenzen befriedigt). Zu diesem Zweck wird .lib zu a.out gelinkt.

```
$ cat hello.c
main()
{
        printf("Hello\n");
}
# linux
$ gcc -static -o unshared hello.c
# True UNIX64
$ cc -non_shared -o unshared hello.c
$ gcc  -o shared hello.c
$ size unshared shared
   text    data     bss     dec     hex filename
    907     224      24    1155     483 shared
 201038    5064    3340  209442   33222 unshared
$
```

a.out mit 'archive libraries':

FILE HEADER
program **.text**
library **.text** for **printf(S)**
program **.data**
library **.data** for **printf(S)**
SYMBOL TABLE
STRING TABLE

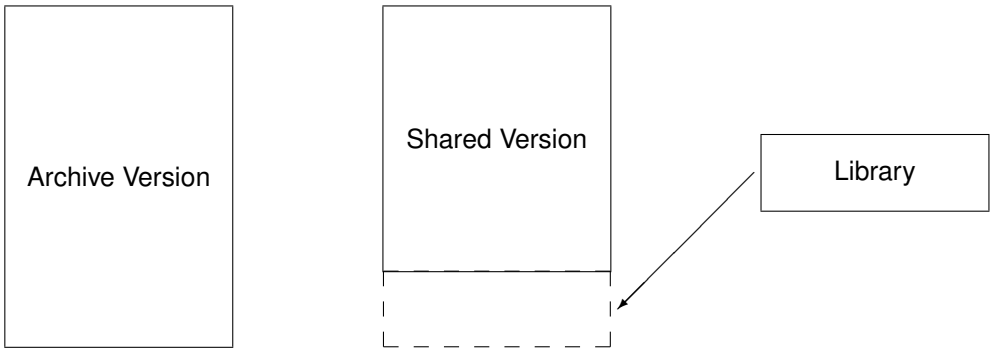
a.out mit Shared Libraries:

FILE HEADER
program **.text**
program **.data**
.lib
SYMBOL TABLE
STRING TABLE

Prozesse mit und ohne Shared Library:

Die Host- und Target Library

Jede Shared Library besteht aus zwei Teilen: der host library, die verwendet wird beim Linken und der target library, die auf der 'target maschine' für die Ausführung von a.out verwendet wird.
Als 'host maschine' bezeichnet man denjenigen Rechner auf dem a.out gebildet wird. Die 'target maschine' ist der Rechner auf dem a.out ausgeführt wird.
Die 'host library' ist wie eine archive library aufgebaut. Jeder Eintrag in ihr definiert Daten- und Textsymbole in der Symboltabelle. Der Linker durchsucht diese Symboltabelle, wenn diese library verwendet wird.



Library code referenced by **.lib**

Die Suche gilt Symbolen die im Programm verwendet, dort aber nicht definiert wurden. Im Falle von Shared Libraries wird aber der Library Code nicht in a.out kopiert. Es werden nur die externen Symbole aufgelöst.
Die target library gleicht einer a.out Datei. Das Betriebssystem liest diese in den Hauptspeicher, falls ein Prozess diese Shared Library benötigt. Im .lib Abschnitt von a.out ist spezifiziert welche Shared Libraries benötigt werden. Bevor ein Prozess gestartet wird, werden all seine benötigten Shared Libraries geladen.

Die Branch Table

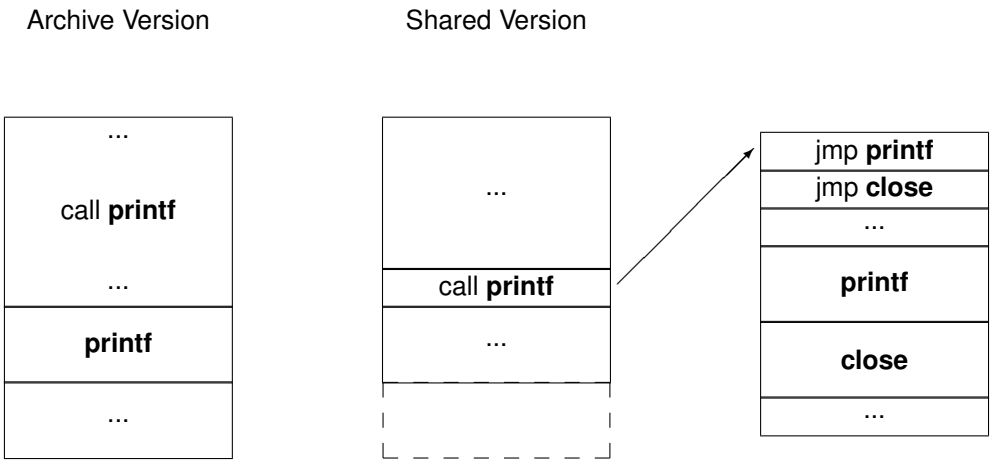
Beim Linken werden die externen Symbole aufgelöst. Jede a.out Datei enthält also Adressen. Was geschieht aber wenn die library geändert wird und Symbole ihren Wert ändern?
Es ändert sich nichts! Zu diesem Zweck beginnt jede Shared Library mit einer Branch Table. Eine Branch Table assoziiert Text Symbole mit absoluten Adressen. Auf diese Weise ändern sich die Werte dieser Symbole nicht, selbst wenn die Library geändert wird.

Shared Libraries können aber auch den Hauptspeicherbedarf erhöhen. Während der Textbereich einer Shared Library von mehreren Prozessen gemeinsam verwendet wird, bekommt jeder Prozess eine Kopie des Datenbereichs. Falls nun ein Prozess allein eine Shared Library verwendet, und dieser Prozess überdies noch nur einen kleinen Teil der Shared Library braucht, so benötigt dieser Prozess mehr Hauptspeicher als wenn er ohne Shared Library gelinkt worden wäre. Denn sobald eine Shared Library benötigt wird, so wird der gesamte Text- und Datenbereich dieser Library geladen.

Bauen einer Shared Library

Zum Bauen einer Shared Library sind die folgenden 6 Schritte notwendig:

1. Wählen einer Adresse für die Library
2. Wählen des Pfadnamens für die Shared Library
3. Wählen des Library Inhalts
4. Eventuell umschreiben von bestehendem Library Code



Library code referenced by **.lib**

5. Erstellen der Library Spezifikation
6. Erstellen der Host- und Target Library unter Verwendung von mkshlib

Bei der Festlegung der Adresse ist nachstehende Tabelle zu berücksichtigen. Für jeden Eintrag in dieser Tabelle ist ein Adressbereich von 0x400000 = 4MB reserviert (max. Segmentgrösse des iAPX386).
Beim Erstellen einer kommerziellen Version einer Shared Library wird empfohlen auch eine 'archive' Version dieser Library mitzuliefern, weil es Anwendungen gibt bei denen die Verwendung einer Shared Library Nachteile bringt.

Start Adresse	Inhalt	Target Pathname
0xA0000000 . . . 0xA3C00000	Reserviert für AT&T UNIX System Shared Library C Library AT&T Networking Library	 /shlib/libc_s /shlib/libnsl_s
0xA4000000 0xA4400000 0xA4800000 0xA4C00000	Generic Database Library	Unassigned
0xA5000000 0xA5400000 0xA5800000 0xA5C00000	Generic Statistical Library	Unassigned
0xA6000000 0xA6400000 0xA6800000 0xA6C00000	Generic User Interface Library	Unassigned
0xA7000000 0xA7400000 0xA7800000 0xA7C00000	Generic Screen Handling Library	Unassigned
0xA8000000 0xA8400000 0xA8800000 0xA8C00000	Generic Graphics Library	Unassigned
0xA9000000 0xA9400000 0xA9800000 0xA9C00000	Generic Networking Library	Unassigned
0xAA000000 . . .	Generic - to be defined	Unassigned
0xB0000000 . . .	For private use	Unassigned

8.12 Shared Libraries (ELF)

ELF ist eine Abkürzung für *Executable and Linking Format* das ist ein Binärformat, das von USL (UNIX System Laboratories) entwickelt wurde und heute in Solaris, System V R4 und Linux verwendet wird. Der Hauptvorteil ist die grössere Flexibilität. Wenn eine Shared Library richtig installiert ist, werden alle Programme, die danach starten automatisch die neue Shared Library verwenden. Shared Libraries ermöglichen:

- aktualisieren vom Libraries und weiterhin verwenden der alten Bibliotheken.
- Überschreiben von speziellen Library Funktionen eines Programms
- Durchführen von Library Änderungen im laufenden Betrieb

8.12.1 Konventionen

Damit Shared Libraries alle diese gewünschten Eigenschaften unterstützen können, müssen eine Anzahl von Konventionen und Richtlinien befolgt werden. Man muss den Unterschied zwischen dem Namen einer Library *soname* und dem *real name* verstehen (und wie sie aufeinander einwirken). Man muss auch wissen, wo sie in das Dateisystem abgelegt werden.

Shared Library Names
Jede Shared Library hat einen speziellen Namen, den *soname*. Der *soname* hat den Präfix `lib`, gefolgt vom Namen der Library, und dem String `.so`, gefolgt von einem Punkt und einer Versionsnummer. Der *real name* ist der Dateiname der Shared Library, dieser enthält in seinem Dateinamen alle Subversionsnummern. Normalerweise sind die *soname*s auf die *real names* gelinkt.

```
oser@oshome:/usr/lib> ls -l libcrypt
lrwxrwxrwx    1 root     root           18 2004-01-23 19:12 libcrypto.so.0
                                                          -> libcrypto.so.0.9.7
-r-xr-xr-x    1 root     root      1143807 2004-03-02 18:34 libcrypto.so.0.9.7
```

Daneben gibt es noch den *linker name* das ist der *soname* ohne den Präfix `lib` und ohne `.so` und den Versionsnummern. Im obigen Beispiel heisst der *linker name* `crypto`.

Wo werden Shared Libraries abgelegt?
In einem Unix System werden die Shared Libraries in einem der folgenden Kataloge abgelegt:

- `/lib`
- `/usr/lib`
- `/usr/local/lib`

Falls eine Shared Library in einem anderen Katalog abgelegt ist, so kann man die Variable `LD_LIBRARY_PATH` setzen, damit diese gefunden wird. In Linux sind die Kataloge wo Shared Libraries zu finden sind in der Datei `/etc/ld.so.conf` spezifiziert.
Das Laden der Shared Libraries unter Linux wird durch `/lib/ld-linux.so.X` durchgeführt.

8.12.2 Speicherbelegung mit Shared Libraries (ELF)

ELF Libraries können an irgend eine freie Speicheradresse des Prozesses geladen oder gefaltet werden, da der Code *Position Independant* ist. Es gibt deshalb auch keine Zuweisung von Adressen zu Libraries wie im Falle von a.out.

8.12.3 Erzeugen einer Shared Library

Es ist einfach eine *shared library* zu erzeugen. Man compiliert alle Module mit `-PIC` (Position Independent Code) und linkt danach die Library mit einem einzigen Kommando. Um eine Library `libfoo.so` zu erzeugen geht man wie folgt vor:

```
$  gcc -fPIC -c *.c
$  gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$  ln -s libfoo.so.1.0 libfoo.so.1
$  ln -s libfoo.so.1 libfoo.so
$  export LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH
```

Einige Detail sind zu beachten beim Erzeugen von Shared Libraries:

- Die Shared Library darf nicht gestrippt werden, da die Symboltabelle beim Linken/Laden verwendet wird.
- Der einzige Nachteil von ELF ist, dass das System zwischen 1 bis 5% langsamer wird. Das kommt daher, dass der *Position Independent Code* ein Register braucht zur Aufnahme des jeweiligen Offsets. Damit verbleibt für den Code der Anwendung ein Register weniger.

8.12.4 Inkompatible Shared Libraries

Eine Shared Library kann inkompatibel werden aus folgenden Gründen:

- Die Funktionalität einer Funktion ändert
- Zufügen von Daten zu irgendwelchen *struct* ausser am Ende
- Das Interface einer Funktion ändert

Wenn keine der obigen Verstösse vorkommen so ist hat Shared Library das gleiche ABI (Application Binary Interface) und bleibt kompatibel.

8.12.5 Linken von Programmen unter Verwendung von Shared Libraries

Wenn mit dem Linker *ld* ein ausführbares Programm erzeugt wird unter Linux werden per Default Shared Libraries verwendet (es wird dynamisch gelinkt). Die Symbole der Shared Library sind in der Datei des *real name* abgelegt. Das Einschliessen von Libraries wird bei *ld* mit der Option `-l` verlangt. Hier muss jetzt der *linker name* angegeben werden. Beispiel für das Erzeugen eines ausführbaren Programms *mypgm*:

```
ld -o mypgm mypgm.o -L /home/loginname -lfoo
```

Hier wird die Shared Library `libfoo.so` eingeschlossen, die sich im Katalog `/home/loginname` befindet. Die Option `-L` ist notwendig wenn sich die Shared Library nicht in einem Standard Katalog befindet.

8.12.6 Unaufgelöste Symbole beim Linken

Wenn man linkt so ist es möglich, dass nicht alle notwendigen Shared Libraries spezifiziert werden und der Linker mit einer Fehlermeldung abbricht. Wie findet man aber die fehlende Shared Library? Hier eine einfache Methode: Nehmen wir an, dass das Symbol `xyz` nicht gefunden wurde. Man verwendet ein Terminalfenster und gibt nun folgendes kleines Script ein:

```
cd LibraryDirectory
for i in lib*
do
echo $i
nm $i|grep xyz
done
```

Damit geht man alle Shared Libraries im Katalog *LibraryDirectory* durch, gibt den Namen aus und sucht mit *nm* nach dem gewünschten Symbol. Falls man das Symbol findet, weiss man auf Grund der Ausgabe `echo $i` in welcher Shared Library es sich befindet.

Literatur

[1] Understanding the Linux Memory Manager, Mel Gorman, University of Limerick, Ireland; `http://www.csn.ul.ie/~mel/projects/vm/`.

[2] Modern Operating Systems, Andew S. Tanenbaum, Prentice Hall

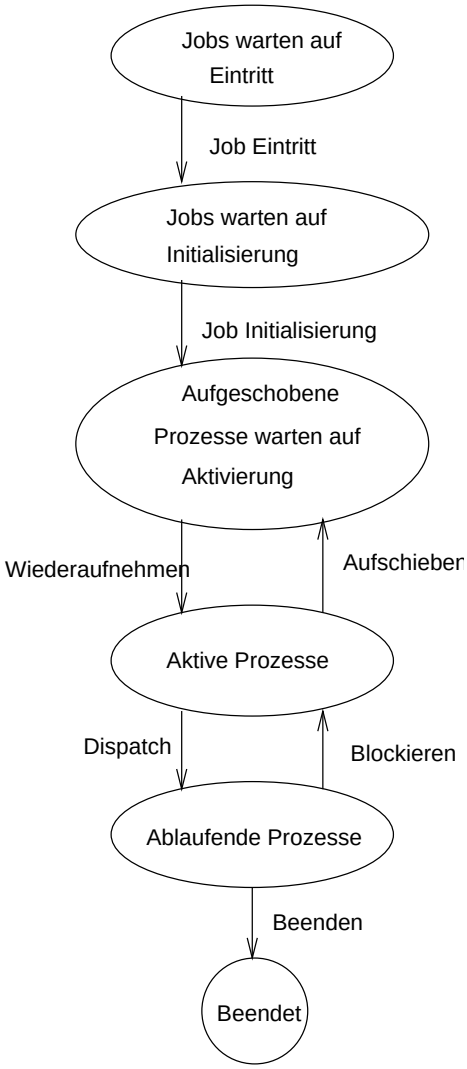


Abbildung 49: Ebenen der Ablaufsteuerung

9 Ablaufsteuerung (Scheduling)

Unter Ablaufsteuerung versteht man die Zuweisung eines physischen Prozessors zu einem Prozess. In diesem Kapitel wird die Frage behandelt wann der Prozessor welchem Prozess zugewiesen werden soll.

9.1 Ebenen der Ablaufsteuerung

Wir unterscheiden 3 verschiedene Ebenen:

9.2 Ziele der Ablaufsteuerung

Die Ablaufsteuerung soll:

- fair sein und niemals unbestimmt aufschieben.

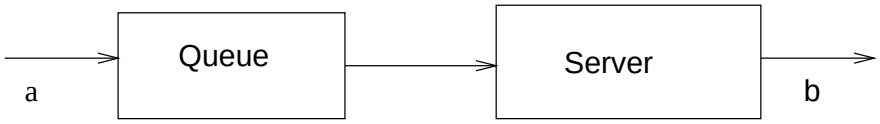


Abbildung 50: Scheduling Wahrscheinlichkeitsmodel

- den Durchsatz durch das System möglichst gross halten.
- die Anzahl der interaktiven Benutzer möglichst gross machen.
- voraussagbar sein; ein Job soll in der gleichen Zeit abgeschlossen werden unabhängig von der Systembelastung.
- nur minimalen Overhead erzeugen.
- den Ausgleich finden zwischen Antwortzeit und Benutzung der vorhandenen Betriebsmittel: Echtzeitsysteme= schnelle Antwort aber schlechte Ausnützung der Betriebsmittel.
- verlangte Prioritäten durchsetzten.
- Prozesse die wichtige Betriebsmittel halten bevorzugen, auch wenn diese niedere Priorität haben.

Viele dieser Ziele stehen im Konflikt miteinander, was die Ablaufsteuerung zu einem komplexen Problem werden lässt.

9.3 Kriterien der Ablaufsteuerung

Um die Ziele der Ablaufsteuerung zu erreichen, muss für jeden Prozess folgendes berücksichtigt werden:

- Wie stark ist der Prozess E/A gebunden? Braucht er die CPU nur kurz vor der E/A?
- Wie stark ist der Prozess CPU gebunden? Braucht er die CPU bis zum Ende seines Quantums?
- Ist der Prozess interaktiv oder batch?
- Wie wichtig ist eine schnelle Antwortzeit?
- Welche Priorität hat der Prozess?
- Wie häufig entstehen Page-faults bei der Ausführung?
- Wie oft wurde ein Prozess 'preempted' zugunsten eines Prozesses höherer Priorität.
- Wieviel CPU Zeit hat der Prozess bereits verbraucht?

9.4 Wahrscheinlichkeitsmodelle

Das Scheduling ohne Preemptions kann mit folgendem Wahrscheinlichkeitsmodell beschrieben werden:
Damit ein solches System mathematisch untersucht werden kann, muss folgendes bekannt sein:

- Wahrscheinlichkeitsverteilung für das Eintreffen der Aufträge.
- Wahrscheinlichkeitsverteilung für Rechenzeiten der Aufträge.

Annahme:	a =	mittlere Ankunftsrate
	b =	mittlere Abfertigungsrate
	p(x,t) =	Wahrscheinlichkeit für das Auftreten von x Aufträgen in der Zeitspanne t.

$p(x,t) = \frac{e^{-at}(at)^x}{x!}$

Wahrscheinlichkeit für das Ausbleiben eines Auftrages während der Zeitspanne t:

$p(0,t) = e^{-at}$

Wahrscheinlichkeit für das Eintreffen eines Auftrages während der Zeitspanne t:

$1 - p(0,t) = 1 - e^{-at}$

Die Wahrscheinlichkeit für die Abfertigung eines Auftrages während der Zeitspanne t:

$B(t) = 1 - e^{-bt}$

- Zeitspanne zwischen zwei Aufträgen: $\frac{1}{a}$
- Zeitspanne zwischen zwei Abfertigungen (Bearbeitungszeit): $\frac{1}{b}$

Ein eintreffender Auftrag wird im allgemeinen nicht sofort bearbeitet, er wartet vorerst im Warteraum (Queue). Ausser der Bearbeitungszeit vergeht die Wartezeit W, die umso grösser ist je länger die Queue ist. Die Summe von Bearbeitungs- und Wartezeit bezeichnet man als Verweilzeit. Little bestimmte die mittlere Verweilzeit als:

$T = \frac{1}{a} * (\frac{\rho}{(1-\rho)}) = \frac{1}{b-a}$
wobei $\rho = \frac{a}{b}$

Abschätzen des Einflusses der Warteschlange auf die Verweilzeit: Ohne Warteschlange: Verweilzeit = Bearbeitungszeit!

$\frac{T}{\frac{1}{b}} = \frac{b}{b-a} = \frac{1}{1-\rho}$

oder verbal: Warteverhältnis= Verweilzeit/Bearbeitungszeit

Das Systemverhalten wird vorwiegend durch den Wert $\rho = \frac{a}{b}$ bestimmt. Wenn a sich b nähert, rückt die Wahrscheinlichkeit dass die Warteschlange nicht leer ist in die Nähe von 1; die Verweilzeit wächst unbeschränkt.

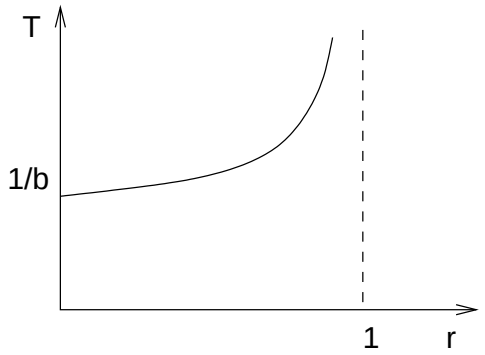


Abbildung 51: Warteverhältnis

9.5 Scheduling Verfahren

Wir betrachten ein System mit einer Warteschlange (Bearbeitungsart = FIFO), das zwei Sorten von Aufträgen bearbeitet: 'kleine' und 'grosse'. Beide Auftragsklassen haben Poissonverteilung bezüglich Ankunfts- und Abfertigungsrate. Man kann zeigen, dass die Überlagerung dieser beiden Auftragsklassen eine poissonverteilte neue Klasse ergibt mit:

$a = a_1 + a_2$ und $\rho = \rho_1 + \rho_2$

Die mittlere Verweilzeit ist damit:

$T = \frac{1}{a} * \frac{\rho}{(1-\rho)}$

Diese Strategie (FIFO) ist den kleinen Aufträgen gegenüber ungerecht. Eine mögliche Alternative ist die 'Shortest Job First' (SJF) Strategie. Dabei wird der Prozess mit der kürzesten Rechenzeit als nächster bearbeitet.

Wartezeiten in Funktion der Bearbeitungszeit der beiden einseitigen Strategien FIFO und SJF:

Shortest Remaining Time (SRT)
Falls bei SJF Preemption angewendet wird, so wird jeweils derjenige Prozess bearbeitet, der die geringste verbleibende Rest-Rechenzeit hat. Dies kann zu Preemtion führen beim Eintreffen neuer Jobs. Die Wartezeiten verhalten sich noch extremer als bei SJF.

Prioritäten

Prioritäten können durch das Betriebssystem zugewiesen, extern zugewiesen oder gekauft werden. Sie können statisch oder dynamisch sein. Dynamische Prioritäten bedeutet, dass das Betriebssystem die Priorität im Laufe der Bearbeitungszeit anpasst. Batch oder TS Benutzer eines Systems können sich für ihre Jobs eine höhere Priorität kaufen.

Deadline Scheduling

Jobs müssen so gesteuert werden, dass sie auf einen speziellen Zeitpunkt beendet sind. Deadline Scheduling ist komplex aus mehreren Gründen:

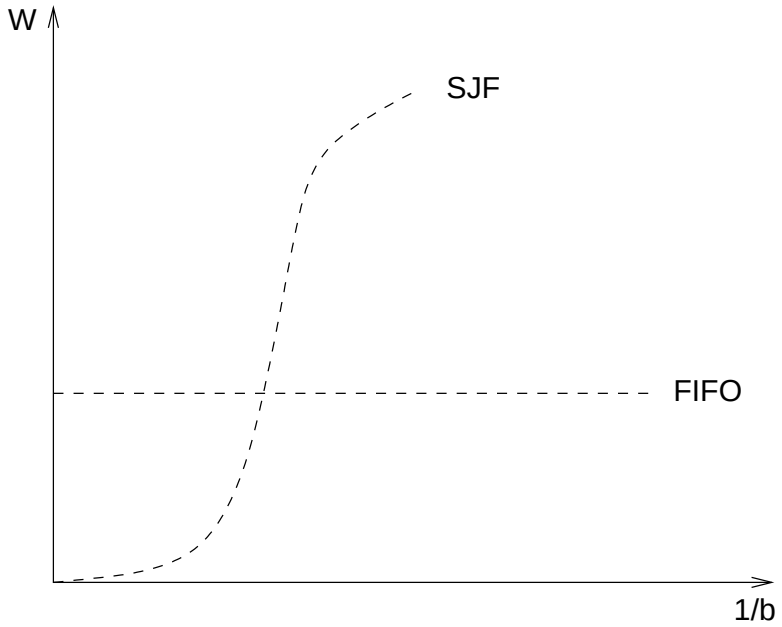


Abbildung 52: Wartezeiten in Funktion der Bearbeitungszeit

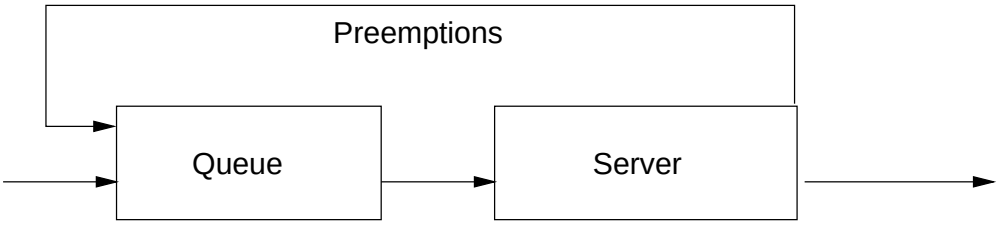


Abbildung 53: Round-Robin (RR) und Processor Sharing (PS)

- Der Benutzer muss genaue Betriebsmittel-Bedürfnisse bekanntgeben.
- Andere Benutzer sollten nicht schwerwiegend benachteiligt werden.
- Richtiges Deadline Scheduling ist meistens ein Optimierungsproblem (Zeitlicher Overhead).

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

Bei RR Scheduling erhält jeder Job ein bestimmtes Quantum Q an Rechenzeit, falls er nicht fertig wird erfolgt Preemption und er wird an den Anfang der Queue gestellt. Falls man den durch die Preemption verursachten Overhead vernachlässigt, und $Q \rightarrow 0$ gehen lässt, so erhält man PS Scheduling.

Vergleich der Wartezeiten von PS, RR und FIFO Scheduling:

Falls man bei RR die Anzahl der aufzunehmenden Jobs begrenzt, kann die Warteschlange in zwei Hälften geteilt werden. RR Scheduling erfolgt nur im rechten Teil der Queue, neue Jobs werden links eingefügt. Neue Jobs werden nur ange-

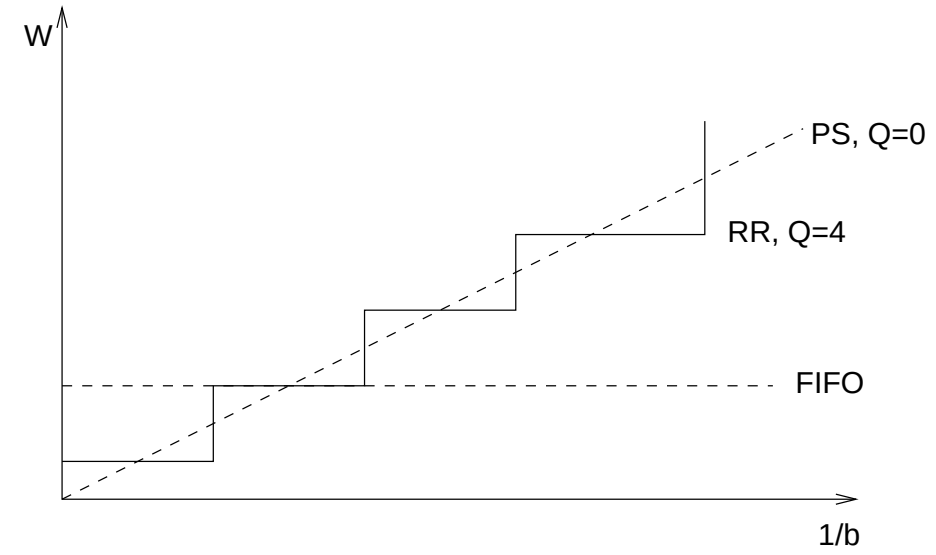


Abbildung 54: Vergleich der Wartezeiten

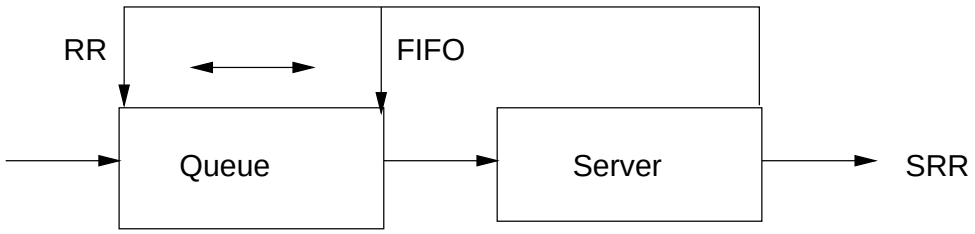


Abbildung 55: Selfish-Round-Robin

nommen, falls ihre Priorität durch warten diejenige der aktiven Jobs angenommen hat. Durch geeignete Wahl der Prioritätsfunktion lässt sich mit diesem Verfahren sowohl RR wie auch FIFO realisieren. Dieses Verfahren wird als Selfish-Round-Robin bezeichnet (SRR).

Highest Response Ratio Next (HRN)

Brinch-Hansen hat diese Strategie entworfen. Damit soll die grosse Benachteiligung grosser Jobs in SJF korrigiert werden. HRN ist eine nonpreemptive Scheduling-Strategie die jedem Job eine Priorität zuweist die eine Funktion ist der Bearbeitungszeit und der Wartezeit. Die dynamische Priorität der HRN berechnet sich als:

$$Prioritt = \frac{(Wartezeit+Bearbeitungszeit)}{Bearbeitungszeit}$$

→ Siehe Figuren Brinch Hansen!

Multilevel Feedback Queues
Im Zeitpunkt da ein Prozess erstmals die CPU zugeteilt erhält weiss der Scheduler im allgemeinen nicht wieviel Rechenzeit dieser Prozess braucht. Der Scheduling Algorithmus sollte:

- Kurze Jobs bevorzugen
- E/A gebundene Jobs bevorzugen damit die Peripherie optimal genutzt wird.
- Die Natur des Jobs möglichst schnell erkennen und das Scheduling anpassen.

Mit Multilevel Feedback Queues ist dies möglich. Ein neuer Job tritt in die Queue der höchsten Priorität ein. Nach dem Ende seines ersten Quantum wird er in die nächstniedrige Queue eingereiht usw. Falls er sein Quantum nicht aufgebraucht hat, und auf E/A wartet, kommt er nach beendigtem Warten in dieselbe Queue zurück. Multilevel Scheduling ist eine Möglichkeit die Ablaufsteuerung adaptiv an die Prozesse anzupassen.

9.6 Linux Scheduling

Der hier angegebene Buchausschnitt zeigt die wesentlichen Teile des Linux Scheduling im Kernel 2.4.20.
URL= `http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html`
Der Linux 2.6 Scheduler:
URL=`http://www.oser.org/~hp/bsyI_aufgaben/linux_cpu_scheduler.pdf`

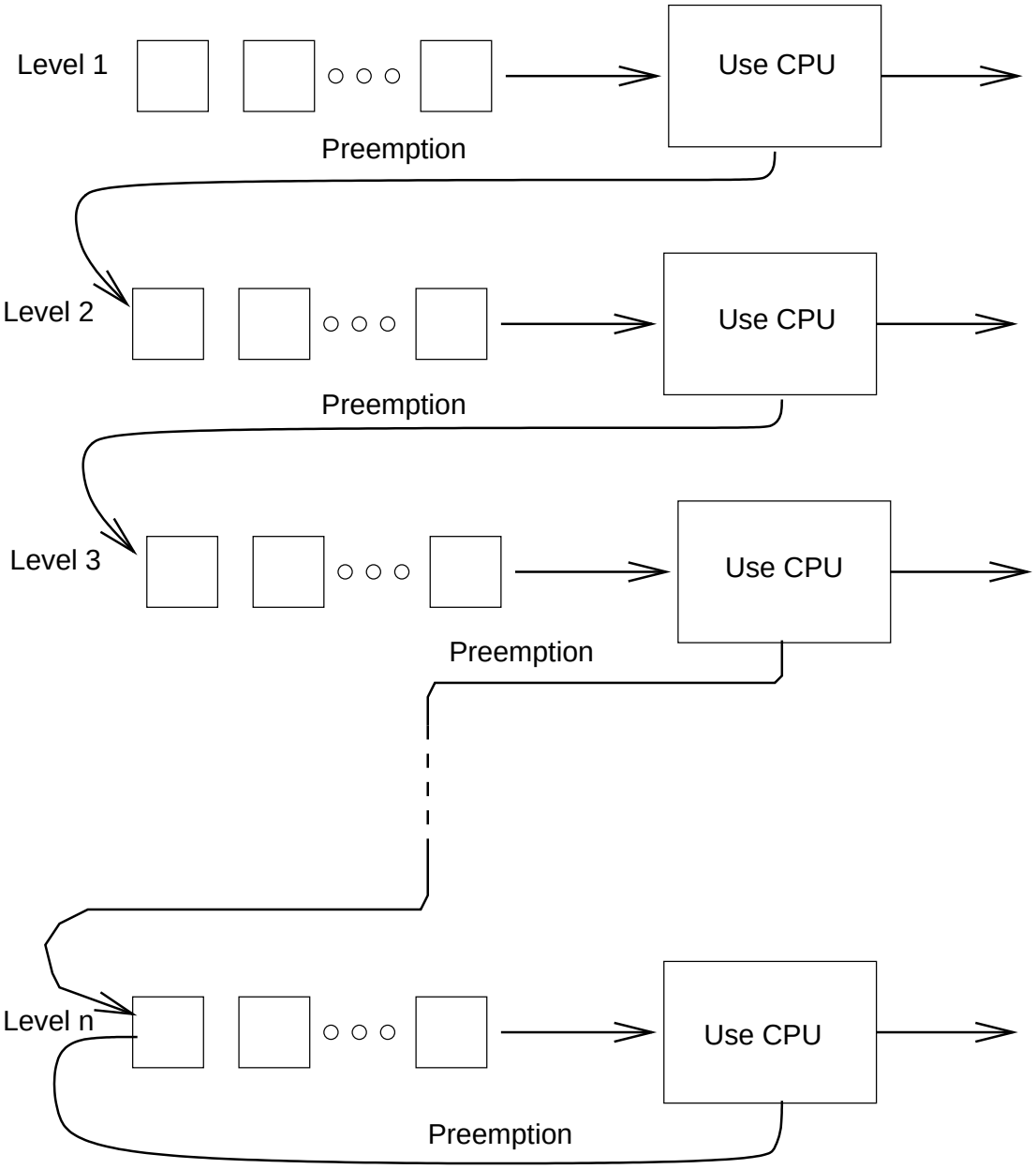


Abbildung 56: Multilevel Feedback Queues