Zusammenfassung für die mündliche Diplomprüfung im Fach Betriebssysteme

Naja von Schmude

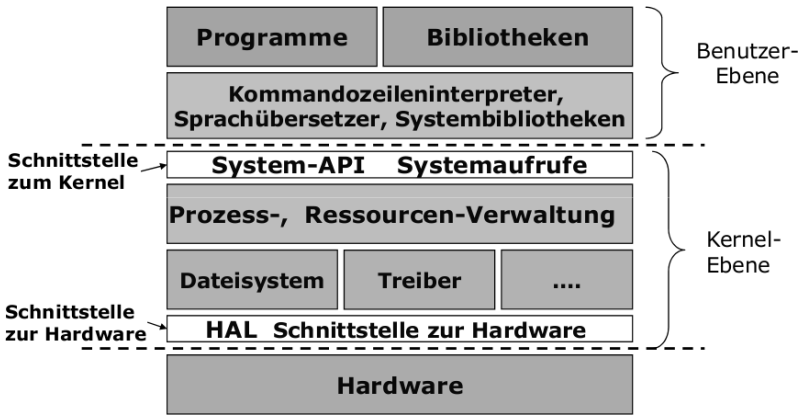
3. März 2011

Inhaltsverzeichnis

Dies ist meine Zusammenfassung des Lernstoffs für die mündliche Diplomprüfung im Bereich praktische Informatik an der Freien Universität Berlin. Sie umfasst den Stoff der Vorlesung Betriebssysteme aus dem Sommersemester 2009 bei Prof. Esponda, und wurde entsprechend mit der zweiten Auflage „Moderne Betriebssystem“ von Andrew S. Tanenbaum ergänzt.

1 Einführung

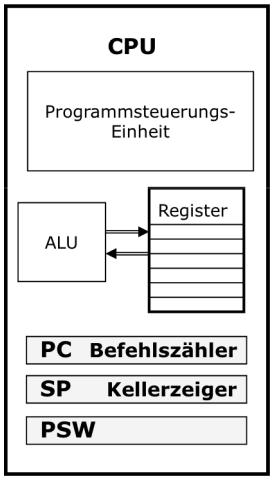
- Ein *Betriebssystem* verwaltet vorhandene Geräte und bietet Benutzerprogrammen einfache Schnittstelle zur Hardware. Betriebssystem abstrahiert also für den Benutzer/ Programmierer von der Hardware und der damit verbundenen Komplexität und stellt eigenen Instruktionssatz zu Verfügung.
- Einordnung des Betriebssystems ins Gesamtsystem; Kernelebene ist das eigentliche Betriebssystem. (Grenzen verschwimmen aber, da z.B. Shell, Compiler etc. oft sehr eng verbunden sind)



- Motivation: Effizient erhöhen, Parallele Hardware ausnutzen, Energieverbrauch senken, Sicherheitsprobleme lösen, neue Geräte = neue Betriebssysteme, Anwendungen besser, wenn Betriebssystem besser
- Zwei Sichtweisen auf Betriebssysteme

1. Betriebssystem ist virtuelle Maschine, die von Hardware abstrahiert (top down Sicht) => portable Software möglich
2. Betriebssystem als Ressourcenmanager, der Zuteilung von Prozessoren, Rechenzeit, Speicher, I/O in geordneter und kontrollierter Weise vornimmt (bottom up Sicht) => endliche Hardware wird als unendliche virtuell behandelt

- *CPU* ist zentrale Recheneinheit des Computers. Wichtige Register der CPU: *PC* (Program counter), zeigt auf Speicheradresse des nächsten Befehls, *SP* (stack pointer) zeigt das Ende des aktuellen Kellers im Speicher, *PSW* (program status word) enthält Kontrollbits, wie z.B. den Modus, Priorität usw.



- Ausführungsmodi der CPU

  Kernel mode Jeder Befehl des Befehlssatzes kann ausgeführt werden und alle Hardwareeigenschaften benutzt.

  User mode Nur ein Teil der Befehle sind nutzbar, Hardware nur bedingt ansprechbar.

  Das Betriebssystem läuft im Kernel mode, Benutzerprogramme im User mode, wo sie pauschal keine Ein/Ausgabe durchführen können. Um trotzdem diese Funktionalität zu nutzen, muss das Benutzerprogramm einen *system call* durchführen. Der Systemaufruf springt in den Kernel und übergibt zeitweise die Kontrolle an das Betriebssystem, das stellvertretend die gewünschte Funktionalität erbringt. Nach der Ausführung springt man zurück in den user mode und das Benutzerprogramm erhält die Kontrolle zurück.

1.1 Betriebssystemkonzepte

- Ein Schlüsselkonzept jeden Betriebssystems ist der *Prozess*. Der Prozess ist ein Programm in Ausführung. Prozesse werden vom Betriebssystem in der *Prozess-*

tabelle verwaltet. Prozesse können mittels *interprocess communication* miteinander kommunizieren.

- *Speicherverwaltung*: Mehrere im Hauptspeicher geladenen Programme müssen gegenseitig geschützt werden und der Speicher muss entsprechend zugewiesen werden. Konzept des *virtuellen Speichers* fasst Hauptspeicher und Festplatte virtuell zusammen, sodass auch Programme, die größer als der Hauptspeicher sind, geladen werden können und transparent Daten hin und hergeschoben werden können.
- Das Betriebssystem abstrahiert von Hardware/ Platten und stellt abstraktes Modell geräteunabhängiger Dateien zu Verfügung. Die Dateien werden hierarchisch in einem *Dateisystem* verwaltet. Das Betriebssystem stellt dem Benutzer entsprechende Systemaufrufe zu Verfügung, mit denen Dateien geöffnet, gelesen, geschrieben usw. werden können.
- Das Betriebssystem muss gewisse *Sicherheitskonzepte* umsetzen, so dass z.B. Dateien vor unbefugtem Zugriff geschützt werden.

1.2 Systemaufrufe

- Menge von *Systemaufrufe* bildet die Schnittstelle zwischen Betriebssystem und Benutzerprogrammen.
- Systemaufrufe bieten Erzeugen und Zerstören von Prozessen, Erzeugen, Löschen, Lesen und Schreiben von Dateien und I/O Aufgaben an.
- POSIX bietet einen festgeschriebenen Satz an Systemaufrufen.
- Prozessmanagement:
  - `pid = fork()` erstellt komplette Kopie des aktuellen Prozess (einschließlich aller Variablen, Dateideskriptoren und Register usw.). Im Elternprozess liefert `pid` die Prozess-ID des neuerzeugten Prozesses, im Kindprozess 0.
  - `pid = waitpid(pid, &statloc, options)` wartet auf die Beendigung eines Kindprozesses.
  - `s = execve(name, argv, environp)` ersetzt kompletten Prozess im Speicher durch andere Datei mit Startparametern und Umgebungsvariablen.
- Datei/ Verzeichnisverwaltung:
  - `fd = open(file, options)` öffnet eine Datei unter dem angegebenen Pfad. Optionen können sein, ob man lesen, schreiben oder lesen und schreiben will usw. Als Rückgabe gibt es den *file descriptor*.
  - `s = close(fd)` schließt die Datei wieder.
  - `n = read(fd, buffer, nbytes)` liest eine gewisse Anzahl von Bytes von der Datei und `n = write(fd, buffer, nbytes)` schreibt Bytes in die Datei.

- `position = lseek(fd, offset, whence)` positioniert den Schreib/ Lesekopf an die gewünschte Stelle.
- Genauso gibt es Befehle um Verzeichnisse zu erstellen usw. wie `mkdir(name, mode)`.

- Sonstiges:
  - `chdir(dirname)` wechselt das Arbeitsverzeichnis des Prozesses
  - Man kann unter UNIX den Zugriffsmodus ändern mit `chmod`
  - Mit `kill(pid, signal)` können Signale an Prozesse gesendet wird, ohne Angabe eines Signals wird der Prozess beendet.

1.3 Nebenläufigkeitskonzepte

- Man unterscheidet zwischen *echter Parallelität* und *virtueller Parallelität*. Bei ersterer laufen Prozesse auf unterschiedlichen CPUs, bei letzterer laufen mehrere Prozesse auf derselben CPU, so dass die Parallelität hier irgendwie simuliert werden muss.
- Bei *single program* wird nur ein Programm ausgeführt. Mehrere Prozesse laufen streng sequentiell nacheinander ab.
- Beim *task switching* (Multiprogramming) liegen mehrere Programme im Arbeitsspeicher. Durch Umschalten zwischen den Prozessen wird die Ausführung eines Programms fortgesetzt.
- *Multitasking* führt das task switching weiter. Dabei schaltet das Betriebssystem so schnell zwischen den Prozessen um, dass der Eindruck der Gleichzeitigkeit entsteht. Jeder Prozess hat dabei seinen eigenen Adressraum.
- Beim *Multithreading* gibt es innerhalb eines Prozesses mehrere Progammfäden, die parallel innerhalb des Programms laufen. Alle Threads teilen sich den gleichen Adressraum, die Threads werden dabei von dem Prozess verwaltet.

1.4 Betriebssystemstruktur

Die Architektur wird hauptsächlich durch die Anforderungen an ein Betriebssystem bestimmt. Dabei unterscheidet man zwischen der *Policy*, die angibt, was das Betriebssystem leisten soll und dem *Mechanism*, das sagt wie es implementiert werden soll.

Monolithisch Es gibt eigentlich keine Struktur innerhalb des Betriebssystems. Das Betriebssystem ist eine Menge von Prozeduren, von denen jede Prozedur jede andere aufrufen kann. (Beispiel: Linux)

Geschichtet Betriebssystem wird in verschiedene Schichten aufgeteilt, von denen die höheren Schichten auf die darunter liegenden Schichten aufbauen. Damit verbunden sind die Funktionen in den unteren Schichten primitiver, die Komplexität nimmt nach oben hin zu.

Mikrokernel Der eigentliche Kern soll so klein wie möglich gehalten werden, d.h. möglichst viel Funktionalität soll außerhalb des Kerns durch Benutzerprogramme realisiert werden. Diese Funktionalitäten werden durch *Server-Prozesse* angeboten, ein Prozess, der den Dienst in Anspruch nehmen will, wird *Client-Prozess* genannt. Mikrokern muss so nur Kommunikation zwischen Server-Prozess und Client-Prozess realisieren. (Beispiel: Mac OS teilweise)

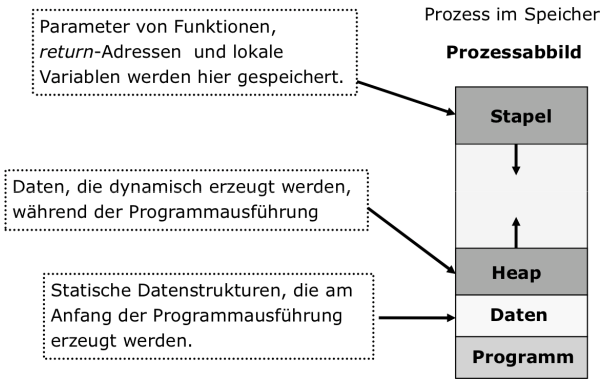
Objektorientiert Betriebssystem hat modulare Struktur. Es gibt ein zentrales Kern-Modul und verschiedene ladbare Module, die alle durch eine saubere Schnittstelle definiert sind. Erweiterung von Modulen in höheren Ebenen ist leicht möglich. (Beispiel: Solaris, Mac OS teilweise)

Virtuelle Maschine Eine virtuelle Maschine bietet eine exakte Kopie der nachten Hardware, sodass sie alle Eigenschaften der Hardware ebenfalls aufweist. Mehrere VMs können so auf einer physischen Maschine laufen, die jeweils unterschiedliche Betriebssysteme benutzen können. (Beispiele: VMware, Java VM)

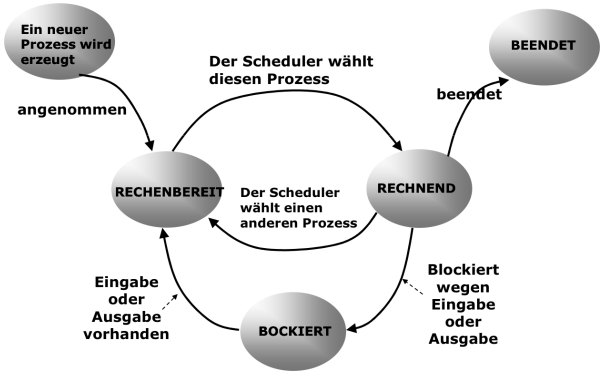
2 Prozesse und Threads

2.1 Prozesse

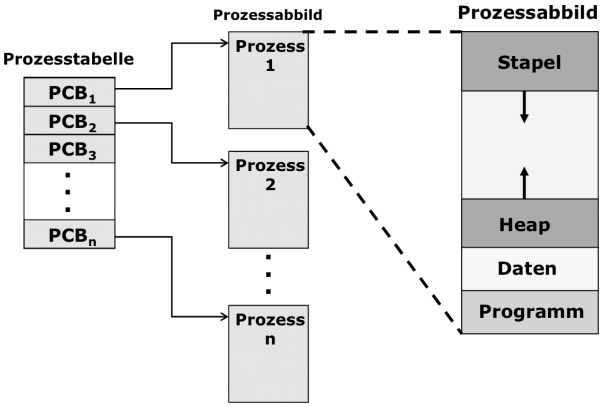
- Ein Prozess hat ein Programm zugrunde liegen, welches durch den Prozess ausgeführt wird. Der Prozess bezeichnet dabei die Aktivität der Ausführung inkl. der Ein- und Ausgaben und dem aktuellen Zustand.



- Ein Prozess kann sich in unterschiedlichen *Zuständen* befinden. Den Übergang zwischen *rechenbereit* und *rechnend* erledigt dabei der Scheduler, der bestimmt, welcher Prozess wann und wie lange CPU-Zeit bekommt (das natürlich möglichst fair usw.).



- Prozesse werden vom Betriebssystem über eine *Prozesstabelle* verwaltet, die pro Prozess einen Eintrag enthält. Ein solcher Eintrag wird als *process control block* (PCB) bezeichnet und enthält die aktuellen Informationen über den Zustand des Prozesses wie PID, PC, SP, PSW, Register, Speicherbelegung, Prozesszustand, benutzte CPU-Zeit, Eltern/ Kindprozesse, offener Dateien usw.



2.2 Threads

- Ein Thread bezeichnet einen *Programmfaden*. Das Threadmodell erweitert das Prozessmodell um die Möglichkeit mehrere voneinander unabhängige Programmfäden in demselben Prozess (d.h. mit gleichem Speicherbereich, gleiche geöffnete Dateien und Ressourcen usw.) laufen zu lassen. Daher bezeichnet man Threads auch als *leichtgewichtige Prozesse*.

- Pro Thread werden folgende Elemente verwaltet: Program counter, Register, Stack und Zustand. Ein Thread hat die gleichen Ausführungszustände wie ein Prozess.

- Da alle Threads auf denselben Adressraum zugreifen, können sie gegenseitig ihre Stacks verändern usw., da kein Schutz zwischen den Threads besteht.

- Threads können sich selbst schlafen legen um andere Threads arbeiten zu lassen. Dies ist wichtig, da es auf Threadebene keine Time-Interrupts gibt, die Timesharing erzwingen.
- Vorteile von Threads: Schneller zu Erzeugen als Prozesse, Beschleunigung von Anwendungen, z.B. durch eigene Threads für GUI und I/O usw.
- Threads können entweder im Benutzeradressraum durch eine Laufzeitumgebung innerhalb der Prozesse verwaltet werden oder direkt im Kernel des Betriebssystems (oder in einer Mischform). Beide Varianten haben Vor- und Nachteile.

2.3 Interprozesskommunikation

- Interprozesskommunikation hat drei Unterpunkte:
  1. Es muss geklärt werden, wie ein Prozess Informationen an einen anderen weiterreichen kann.
  2. Es muss sichergestellt werden, dass sich zwei Prozesse nicht in die Quere kommen, wenn kritische Aktivitäten durchgeführt werden.
  3. Es müssen Abhängigkeiten unter den Prozessen so gelöst werden, dass ein sauberer Ablauf möglich ist.
- *Race Conditions* sind Situationen, in denen mehrere Prozesse einen gemeinsamen Speicherbereich lesen bzw. beschreiben und das Ergebnis davon abhängt, welcher Prozess wann an die Reihe kommt. (Beispiel: Erster Prozess liest Variable `x`, wird dann zufällig schlafen gelegt. Währenddessen liest zweiter Prozess auch `x`, berechnet damit irgendwas und schreibt den Wert neu. Der erste Prozess kommt wieder an die Reihe, arbeitet mit dem alten Wert weiter, schreibt ebenfalls und das Ergebnis des zweiten Prozesses ist auf immer verloren.)
- Zur Vermeidung von Race Conditions muss *wechselseitiger Ausschluss* beim Lesen bzw. Schreiben von gemeinsamen Speicherbereichen gewährleistet werden. Die Programmteile, in denen auf gemeinsamen Speicher zugegriffen wird, nennt man *kritische Abschnitte*. Es muss verhindert werden, dass zwei Prozesse nie gleichzeitig in kritischen Abschnitten sind.
- Um eine gute Lösung für dieses Problem zu bekommen, müssen folgende Bedingungen eingehalten werden:
  1. Keine zwei Prozesse dürfen gleichzeitig in kritischen Abschnitten arbeiten.
  2. Keine Annahmen über Geschwindigkeit / Anzahl der CPUs dürfen getroffen werden.
  3. Kein Prozess außerhalb eines kritischen Abschnitts, darf andere Prozesse blockieren.
  4. Kein Prozess soll ewig warten, um in seine kritische Region einzutreten.

  (Steffens super Merksatz: Kritische Geschwindigkeiten blockieren ewig :-))

- Ansätze für wechselseitigen Ausschluss:

  Unterbrechungen ausschalten Die Idee ist, dass ein Prozess sämtliche Unterbrechungen ausschaltet, nachdem er in seine kritische Region eintritt und erst dann wieder aktiviert, wenn er den kritischen Bereich verlässt. Unterbrechungen abschalten ist aber ein Privileg des Betriebssystems und daher für Benutzerprogramme nicht geeignet. Außerdem funktioniert dies nicht mit mehreren CPUs, da nur die von einem Prozess benutzte CPU von Unterbrechungen geschützt ist, nicht etwa die anderen CPUs auf denen Prozesse laufen, die ebenfalls gemeinsame Speicherbereiche manipulieren könnten.

  Variablen sperren Man benutzt eine gemeinsame Sperrvariable, die mit `0` initialisiert wird. Bevor ein Prozess in seinen kritischen Abschnitt geht, wird diese Variable abgefragt, ob sie `0` ist, wenn ja, darf er weitermachen und setzt sie auf `1`. Ansonsten wird solange gewartet, bis sie auf `0` gesetzt wird. Dies ist aber auch keine gute Lösung, da hier bei der Sperrvariable Race Conditions auftreten und somit beide Prozesse gleichzeitig in die kritische Region eintreten können.

  ```
  if(lock == 0) {
    lock = 1;
    doCriticalStuff();
  }
  else {
    warten ...
  }
  ```

  Strikter Wechsel Die Variable `turn` zeigt an, welcher Prozess gerade an der Reihe ist (bzw. als nächstes ran darf). Solange der Prozess nicht an der Reihe ist, wird durch *busy waiting* die Variable überprüft, bis sie sich auf den gewünschten Wert ändert. Dieses Sperren durch aktives Warten wird auch als *Spinlock* bezeichnet. Diese Lösung ist allerdings nicht ideal, da die dritte genannte Bedingung verletzt wird, da es auftreten kann, dass ein Prozess auf einen anderen warten muss, obwohl dieser in einem nicht kritischen Bereich ist. Dies passiert, wenn z.B. Prozess A so viel schneller seine Berechnungen durchführt als Prozess B, sodass A wieder oben in der while-Schleife hängt und B derweilen immer noch in seiner nicht kritischen Berechnungsphase rechnet.

  Prozess A
  ```
  while(TRUE) {
    while(turn != 0) ;
    doCriticalStuff();
    turn = 1;
    nonCriticalStuff();
  }
  ```

  Prozess B
  ```
  while(TRUE) {
    while(turn != 1) ;
    doCriticalStuff();
    turn = 0;
    nonCriticalStuff();
  }
  ```

  Petersons Lösung Petersons Lösung für zwei Prozesse verbindet den Mechanismus der Sperrvariablen mit dem strikten Wechsel. Bevor ein Prozess ge-

meinsam genutzten Speicher verwendet, ruft er `enter_region` mit seiner Prozessnummer (0 oder 1) auf. In der Funktion bleibt er dann so lange hängen, bis er sicher die Speicherbereiche benutzen kann. Wenn er fertig ist, ruft er `leave_region` auf.

```
void enter_region(int process) {
  int other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process && interested[other] == TRUE) ; // busy waiting
}

void leave_region(int process) {
  interested[process] = FALSE;
}
```

Es kann hier zusätzlich zu dem ungünstigen busy waiting zum Problem der *Prioritätsumkehr* kommen. Dabei hat man zwei Prozesse, der eine mit hoher Priorität H und der andere mit niedriger Priorität L, sodass immer H ausgeführt wird, sobald er bereit ist. Sei L nun im kritischen Abschnitt und H werde in diesem Moment ausführungsbereit. Dann wird L schlafen gelegt, aber H wird endlos in einer Schleife hängen, da L den kritischen Bereich noch nicht verlassen hat und somit die Sperre nicht abgegeben und auch nicht mehr rankommen wird. H kann also nicht in seinen kritischen Bereich eintreten.

Test and Set Lock (TSL) Die Prozessoranweisung `TSL RX,LOCK` ist ein atomarer Befehl, der den Inhalt des Speicherworts `LOCK` ins Register `RX` einliest und zudem den Speicherbereich `LOCK` mit einem Wert ungleich null überschreibt. Während der Operation wird der Speicherbus gesperrt, sodass auch andere CPUs nicht währenddessen auf den Speicherbereich zugreifen können. Auch hier müssen Prozesse vor der Verwendung von gemeinsamen Speicherbereichen `enter_region` aufrufen und danach `leave_region`.

```
enter_region:
  TSL RX,LOCK  // kopiere LOCK in Register RX und setzte LOCK auf 1
  CMP RX, #0   // vergleiche RX mit 0
  JNE enter_region // wenn RX != 0, springe
  RET

leave_region:
  MOVE LOCK, #0 // schreibe in LOCK den Wert 0
  RET
```

Auch hier hat man das Problem des busy waiting und der Prioritätsumkehr.

Sleep and Wakeup Anstatt aktiv zu warten ist es sinnvoller, dass der Prozess schlafen gelegt wird und so keine CPU Zeit beim Sperren verschwendet. Durch den Systemaufruf `sleep` (oder `int pause()`) wird ein Pro-

zess in solcher Weise gesperrt. Da sich der Prozess nicht selbst wieder aufwecken kann um die Sperre aufzuheben, bedarf es eines anderen Prozesses, der das Aufwachen signalisiert. Dies geschieht durch `wakeup` (oder `void signal(int sig, int *sighand)`), das auf den schlafenden Prozess aufgerufen wird.

Das Erzeuger-Verbraucher-Problem (s.u.) kann so gelöst werden, dass der entsprechende Prozess schlafen gelegt wird und erst wieder aufgeweckt, sobald der Prozess weiterarbeiten kann. Dazu wird die Belegung des Puffers über eine Variable `count` verfolgt. Es kann hier aber auch zu Race Conditions kommen, da z.B. der Erzeuger `count` ausliest mit z.B. dem Wert N und sich daraufhin schlafen legen will. Bevor aber dies geschieht, wählt nun der Scheduler den Verbraucher aus. Dieser verarbeitet einen Eintrag und schickt das Signal an den Erzeuger. Da der Erzeuger aber noch nicht schläft, geht das Signal verloren. Wenn nun der Erzeuger das nächste Mal durch den Scheduler ausgewählt wird, legt er sich schlafen und wird nie mehr aufwachen. Ebenso wird auch irgendwann der Verbraucher schlafen gehen, wenn der Buffer leer wird und auch nicht mehr aufwachen.

Eine schnelle Lösung wäre die Einführung eines *Weckruf-Warte-Bit*, aber bei mehr als zwei Prozessen funktioniert dies auch nicht mehr.

```
void producer() {
  item = produce_item();
  if(count == N)
    sleep();
  insert_item(item);
  count++;
  if(count == 1)
    wakeup(consumer);
}
```

```
void consumer() {
  if(count == 0)
    sleep();
  item = remove_item();
  count--;
  if(count == N-1)
    wakeup(producer);
  consume_item(item);
}
```

Semaphoren Ein Semaphor ist eine Datenstruktur, die neben einer Zählvariable und einer Warteschlange der Prozesse zwei Operationen `up` (oder `release`, `V`) und `down` (oder `aquire`, `P`) enthält, um diese Variable zu erhöhen bzw. zu erniedrigen. Der Zweck von Semaphoren ist es Weckrufe zu speichern. Sollte ein Prozess auf einem Semaphor `down` ausführen und der Wert des Semaphor 0 sein, so wird der Prozess sofort schlafen gelegt. Ansonsten wird ein gespeicherter Weckruf verbraucht und der Zähler erniedrigt. Bei einer `up` Operation wird der Wert um eins erhöht und sollten Prozesse in einer `down` Operation hängen, zufällig einer von den wartenden aufgeweckt. Die Operationen des Semaphors sind *atomar*, sodass sie nicht unterbrochen werden können. Semaphoren werden meistens als Systemaufrufe durchs Betriebssystem bereitgestellt, in dessen Operationen die Unterbrechungen kurz ausgeschaltet werden (bei mehreren CPUs brauch man noch Sperrvariable mit TSL).

Das Erzeuger-Verbraucher-Problem kann mit Semaphoren gelöst werden unter Benutzung von drei Semaphoren `mutex` (steuert Zugriff auf kritische Bereiche, initialisiert mit 1), `empty` (zählt leere Pufferplätze, initialisiert mit

N) und `full` (zählt volle Pufferplätze, initialisiert mit 0).

```
void producer() {
  int item;
  while(TRUE) {
    item = produce_item();
    down(&empty);
    down(&mutex);
    insert_item(item);
    up(&mutex);
    up(&full);
  }
}
```

```
void consumer() {
  int item;
  while(TRUE) {
    down(&full);
    down(&mutex);
    item = remove_item();
    up(&mutex);
    up(&empty);
    consume_item(item);
  }
}
```

Semaphoren, die mit 1 initialisiert sind und dazu gebraucht werden gegenseitigen Ausschluss von mehreren Prozessen zu gewährleisten, werden *binäres Semaphor* genannt.

Mutex Ein Mutex ist eine vereinfachte Version des Semaphors, in der die Variable nur zwei Zustände einnehmen kann, nämlich gesperrt oder nicht gesperrt. Dementsprechend nennt man die Operationen hier `mutex_lock` und `mutex_unlock`. Mit Mutexen kann man gegenseitigen Ausschluss auf kritischen Abschnitten realisieren. Mutexe werden meistens für Threads verwendet.

```
mutex_lock:
    TSL RX, MUTEX     // kopiere MUTEX in RX und setze MUTEX auf 1
    CMP RX, #0        // vergleiche RX mit 0
    JZE ok            // wenn RX == 0, dann ist Mutex nicht gesperrt
    CALL thread_yield // Mutex aktuell gesperrt -> schlafen legen
    JMP mutex_lock    // versuche erneut die Sperre zu bekommen
ok: RET

mutex_unlock:
    MOVE MUTEX #0     // schreibe 0 in MUTEX (gebe Lock ab)
    RET
```

Monitore Ein Monitor ist ein *abstrakter Datentyp*, der Prozeduren, Datenstrukturen und Variablen bereitstellt, und zwar in der Form, dass interne Variablen des Monitors nicht von außerhalb verwendet werden können sondern nur über die Monitoreigenen Prozeduren. Zusätzlich kann nur ein Prozess gleichzeitig eine Prozedur eines Monitors verwenden. Will ein Prozess eine Prozedur ausführen, so wird zuerst durch den Monitor geprüft, ob ein anderer Prozess im Monitor aktiv ist, wenn ja, wird der anfragende Prozess schlafen gelegt, bis der aktive Prozess den Monitor verlässt. Der Compiler ist in dem Fall dafür zuständig den wechselseitigen Ausschluss zu gewährleisten; hier werden meistens Mutexe verwendet.
Zusätzlich sind noch *Bedingungsvariablen* (oder auch Zustandsvariablen, condition variable) nötig, die einen Prozess sperren können, wenn dieser

aktuell nicht weiterarbeiten kann. Auf einer Bedingungsvariable können die Funktionen `wait` und `signal` aufgerufen werden. Ersteres schickt dabei einen Prozess schlafen, wenn dieser nicht weitermachen kann und letzteres weckt einen auf, der auf eine Bedingungsvariable wartet. Ähnlich wie bei `wakeup` geht ein Signal verloren, wenn niemand auf es wartet.
Es gibt drei Varianten, wie nach einem `signal` verfahren werden kann:

1. Der aufgeweckte Prozess wird laufen gelassen und der Signalgeber wird eingestellt.
2. Der Signalgeber muss sofort den Monitor verlassen, d.h. dass ein `signal` nur am Ende einer Monitor-Prozedur stehen kann.
3. Der Signalgeber darf noch so lange weiterarbeiten, bis er den Monitor verlässt. Erst dann darf der aufgeweckte Prozess den Monitor betreten.

Es kann hier übrigens nicht zu den Race Conditions wie bei Sleep-and-Wakeup kommen, da durch den wechselseitigen Ausschluss durch den Monitor Signale nicht verlorengehen können.
In Java kann man über das Schlüsselwort `synchronized` wechselseitigen Ausschluss unter Methoden einer Klasse realisieren. Innerhalb von `synchronized`-Blocks kann man die Bedingungsmethoden `wait` und `notify` aufrufen, die auf jedem Objekt definiert sind.
Monitore haben allgemein das Problem, dass die Sprache sie schon unterstützen muss; und das haben nicht viele.

Message passing Message passing benutzt nicht gemeinsame Speicherbereiche um Daten auszutauschen, sondern zwei Methoden `send(destination, &message)` und `receive(source, &message)` die *Nachrichten* verschicken. Die Adressierung der Nachrichten kann man auf verschiedene Arten realisieren:

- Jeder Prozess hat eindeutige Adresse, sodass Nachrichten direkt an den Prozess adressiert werden.
- Mailboxen werden eingeführt und Nachrichten werden direkt an die Mailboxen adressiert.
- Keine Benutzung von Zwischenspeichern. Der sendende Prozess wird so lange blockiert, bis der `receive`-Aufruf erfolgt. Dann kann die Nachricht direkt vom Sender in den Empfänger ohne Zwischenspeichern kopiert werden. Dieses Vorgehen bezeichnet man auch als *Rendezvous*.

Das Erzeuger-Verbraucher-Problem kann so gelöst werden, dass der Verbraucher N leere Nachrichten an den Erzeuger schickt, der wiederum die gefüllten Nachrichten zurücksendet. Sollte der Erzeuger schneller arbeiten und es keine leeren Nachrichten geben, so wird er blockiert; im anderen Fall, wenn es keine vollen Nachrichten zum Verarbeiten durch den Verbraucher gibt, so wird der Verbraucher blockiert.

Barrieren Eine Barriere ist zum Synchronisieren von einer Gruppe von Prozessen geeignet. Sobald ein Prozess die Barriere erreicht, blockiert er, bis alle Pro-

zesse die Barriere erreicht haben. Dann werden alle wieder auf rechenbereit geschaltet.

- Gängige Probleme der Interprozesskommunikation

  Erzeuger-Verbraucher Problem Auch als beschränktes Pufferproblem bekannt. Zwei Prozesse benutzen einen gemeinsamen Puffer begrenzter Speicherkapazität. Der eine Prozess (*Erzeuger*) schreibt nur Daten in den Puffer, der andere Prozess (*Verbraucher*) liest Daten aus dem Puffer. Es kommt hierbei zu einem Problem, wenn einer der Prozesse schneller arbeitet als der andere. Wenn z.B. der Verbraucher ein Datum aus dem Puffer verarbeiten möchte, dieser aber leer ist bzw. wenn der Erzeuger neue Daten eintragen will, aber die Kapazität des Puffers erschöpft ist.

  Problem der speisenden Philosophen Fünf Philosophen sitzen um einen runden Tisch. Auf dem Tisch liegen fünf Essstäbchen. Die Philosophen versuchen nun abwechselnd zu denken und zu essen. Um zu Essen versuchen sie die beiden Stäbchen links und rechts von ihnen zu bekommen. Wenn sie fertig sind, legen sie diese wieder hin und denken nach. Das Problem ist nun ein Programm zu konstruieren, dass richtig funktioniert und nicht hängenbleibt.
  Die einfachste Lösung, in der beim Nehmen des Stäbchens blockiert wird, bis es verfügbar ist, funktioniert nicht, da es hier zu einem *Deadlock* kommt, wenn alle gleichzeitig ihr linkes Stäbchen nehmen und alle dann beim Nehmen des rechten Stäbchens hängenbleiben.
  Wenn man die Lösung so verändert, dass vor dem Nehmen des rechten Stäbchens überprüft wird, ob es frei ist und wenn nicht das linke Stäbchen wieder hingelegt wird und dann eine feste Zeit gewartet wird um es erneut zu probieren, hat man nicht mehr das Problem des Deadlocks aber des *Aushungerns*. Dies entsteht, wenn schon wieder alle gleichzeitig das linke Stäbchen nehmen, gleichzeitig feststellen, dass das rechte nicht verfügbar ist, das linke wieder hinlegen, gleichlange warten um wieder von vorne zu beginne. Das Problem kann umschifft werden, wenn alle Philosophen eine zufällige Zeit warten, bevor sie wieder anfangen (siehe Ethernet).
  Eine andere Möglichkeit ist, dass das Nehmen des Stäbchens, das Essen und das Hinlegen der Stäbchen durch einen Mutex geschützt werden, dabei kann allerdings nur ein Philosoph gleichzeitig essen, obwohl es theoretisch zweien möglich wäre.
  Die richtige Lösung benutz ein Feld um den Status der Philosophen zu verfolgen (essend, denkend, will essen). Ein Philosoph kann nur in den Zustand essend gehen, wenn keiner der Nachbarn isst. Gleichzeitig wird ein Feld von Semaphoren verwaltet, die pro Philosoph den Philosoph sperrt, wenn er was essen will und die Stäbchen nicht verfügbar sind.

  Leser-Schreiber-Problem Das Problem modelliert den Zugriff auf eine Datenbank, in der es akzeptabel ist, dass mehrere Prozesse gleichzeitig Daten lesen, aber sobald ein Prozess die Datenbank beschreiben will, muss der

Schreiber exklusiven Zugriff bekommen.
Eine Lösung zählt die Anzahl der lesenden Prozesse, Semaphoren werden zum Schützen dieser Variable und zum Gewähren des Datenbankzugriffs benutzt. Der erste lesende Prozess ruft `down` auf dem DB-Semaphor auf, alle weiteren Leser erhöhen einfach nur die Zählvariable. Beim Beenden des Lesevorgangs gibt der letzte Leser den DB-Semaphor wieder mit einem `up` frei. Ein Schreiber versucht ebenfalls den DB-Semaphor zu bekommen.
Ein Problem ist hier, dass solange ein Leser beschäftigt ist, alle weiteren Leser bevorzugt werden, auch wenn ein Schreiber wartet. So kann es sein, dass der Schreiber nie an die Reihe kommt. Daher sollte es so konzipiert werden, dass der Schreiber nur auf die aktiv laufenden Leser warten muss, und nicht auf die nach ihm kommenden.

3 Scheduling

- Scheduling ist dafür verantwortlich zu entscheiden, welcher von mehreren rechenbereiten Prozessen als nächstes an die Reihe kommt und die CPU nutzen darf. Der Teil des Betriebssystems, der diese Entscheidungen trifft, wird *Scheduler* genannt.
- Scheduling-Entscheidung muss getroffen werden, wenn
  - ein neuer Prozess erzeugt wird (soll Vater- oder Kindprozess rechnen?),
  - ein Prozess beendet wird,
  - ein Prozess wegen I/O, oder aus einem anderen Grund blockiert wird oder
  - eine I/O Unterbrechung auftritt.
- Man unterscheidet zwischen *non preemtive* (nicht unterbrechend) und *preemtive* (unterbrechend) Scheduling-Algorithmen. Bei ersteren wird ein Prozess so lange ausgeführt, bis er von selbst blockiert oder von selbst die CPU freigibt. Im anderen Modus lässt der Scheduling-Algorithmus den Prozess nicht länger als eine bestimmte Zeit laufen. Sollte der zugeteilte Zeitschlitz abgelaufen sein, wird ein anderer rechenbereiter Prozess ausgewählt. Die preemtive Variante benötigt einen Taktgeber mit entsprechender Unterbrechung.
- Man unterscheidet *long-term Scheduling*, bei dem Prozesse zur weiteren Ausführung aus dem Hintergrundspeicher in den Hauptspeicher geladen werden (*swapping*) und *short-term Scheduling*, bei dem aus der Liste der rechenbereiten Prozesse einer zum Weiterlaufen ausgewählt wird.
- Mögliche Zielsetzungen bei Schedulingverfahren:

  Gerechtigkeit Vergleichbare Prozesse sollen gleich behandelt werden

  Balance Alle Systemteile sind ausgelastet

  Durchsatz Anzahl der erledigten Aufgaben pro Zeiteinheit (soll meist maximiert werden)

Turnaroundzeit Durchschnittliche Zeit, die ein Prozess braucht um bearbeitet zu werden (soll meist minimiert werden)

Antwortzeit Zeit zwischen Aufruf eines Befehl und Ausgabe des Ergebnisses (soll meist minimiert werden)

Proportionalität Benutzererwartungen erfüllen

Deadlines einhalten Termine müssen eingehalten werden

Vorhersagbarkeit

- Bei Kernelthreads (SCS - system contention scope) konkurrieren alle Threads des Systems, bei Benutzerthreads (PCS - process contention scope) kann die Laufzeitumgebung des aktiven Prozesses den laufenden Thread bestimmen.

- Bei mehreren CPUs wird das Scheduling komplexer. Man unterscheidet das *asymmetrische Multiprozessor-Scheduling*, bei dem eine CPU als Master ausgewählt wird, die dann nur für Systemaktivitäten und I/O zuständig ist und alle anderen Prozessoren für Benutzerprozesse, und das *symmetrische Multiprozessor-Scheduling* in dem alle Prozessoren gleich behandelt werden und jeder Prozessor seine eigene Scheduling-Queue führt.
  Kriterien für CPU-Zuteilung:

  Affinität Prozesse werden entweder nur jeweils einem speziellen Prozessor zugeordnet (*hart*) oder werden bevorzugt auf der CPU laufen gelassen, auf der sie zuletzt liefen (*weich*).

  Lastverteilung Ein spezieller Task prüft periodisch die Lasten der Prozessoren und teilt die Prozesse neu auf (*push migration*) oder unbelastete CPUs holen sich selbst Prozesse (*pull migration*).

3.1 Stapelverarbeitungsscheduling

Ziele: Durchsatz maximieren, Turnaroundzeit minimieren, CPU-Belegung konstant auf hohem Niveau halten.

First-Come First-Serverd Non preemtive Algorithmus, der die Prozesse in der Reihenfolge ihres Auftretens abarbeitet, also eine simple Warteschlange.
Vorteile: Einfach, kein Overhead
Nachteile: I/O Geräte werden nicht sinnvoll genutzt, ebenso wird die CPU nicht gut ausgelastet

Shortest Job First Non preemtive Algorithmus, bei dem die Laufzeiten der Prozesse vorher bekannt sein müssen, damit immer der kürzeste als nächstes zur Ausführung ausgewählt werden kann. SJF minimiert die durchschnittliche Wartezeit und ist dabei optimal, wenn alle Aufgaben gleichzeitig verfügbar sind.
CPU Burst (also benötigte CPU-Rechenzeit) kann approximiert werden über die letzte Voraussage τ_n und den zuletzt benötigten tatsächlichen CPU Burst t_n: $\tau_{n+1} = at_n + (1-a)\tau_n$.

Shortest Remaining Time Next Dies ist eine preemtive Version von SJF, bei dem immer der Prozess mit der kürzesten Restlaufzeit ausgewählt wird.

3.2 Scheduling in interaktiven Systemen

Ziele: Antwortzeit minimieren, Proportionalität wahren.

Round-Robin Preemtive Algorithmus, in dem jedem Prozess ein Quantum (Zeitabschnitt) zugewiesen wird. Hat der Prozess sein Quantum aufgebraucht oder blockiert er, reiht der Prozess sich hinten in der Warteschlange wieder ein um ein neues Quantum zu erhalten. Wichtig ist die Größe des Quantums, da bei zu kleinen Quanten der Overhead des Kontextwechsels zu groß wird und bei zu großen Quanten die Interaktivität verloren geht. Gewöhnlich liegt die Größe des Quantums zwischen $10ms$ und $100ms$.

Prioritätsbasiertes Scheduling Jeder Prozess bekommt eine Priorität zugewiesen und der Prozess mit der höchsten Priorität bekommt den Zuschlag. Prioritäten können dabei dynamisch oder statisch vergeben werden.
Prozesse niedriger Priorität können verhungern, da sie nie an die Reihe kommen. Eine Lösung wäre hier die Priorität der unwichtigen Prozesse mit der Zeit anzuheben bzw. die der hochpriorisierten nach und nach herabzusetzen. Des weiteren kann es zur *Prioritätsumkehr* kommen, wenn ein unwichtiger Prozess eine gemeinsame Ressource mit einem wichtigen Prozess verwendet. Der unwichtige Prozess kann dann den wichtigen blockieren. Eine Lösung ist hier die Prioritätsvererbung (also das der unwichtige Prozess im Zuge des Ressourcenzugriffs die hohe Priorität bekommt).

Highest Response Ratio Next Die Response Ratio berechnet sich über $R = \frac{w+s}{s}$ wobei w die bisherige Wartezeit und s die erwartete Bearbeitungszeit insgesamt des Prozesses ist. Der Prozess mit dem höchsten Wert für R wird als nächstes ausgeführt. Insgesamt werden hier kurze Prozesse bevorzugt, aber lange Prozesse können auch nicht verhungern.

Multilevel Queue Scheduling Man benutzt für verschiedene Prozessgruppen unterschiedliche Schedulingverfahren, z.B. für Systemprozesse Prioritäten, für interaktive Prozesse Round Robin usw.

Multilevel Feedback Queue Scheduling Man hat vier Prioritätsklassen, pro Prioritätsklasse wird eine Warteschlange von Prozessen verwaltet. Je höher die Priorität, umso kleinere Quanten bekommt der Prozess am Stück zugesprochen (höchste 4, dann 8, dann 16 und ganz unten geht es non preemtive nach FCFS).

Lotterie-Scheduling Die Prozesse bekommen Lotterielose für die verschiedenen Ressourcen (z.B. CPU Zeit), der Scheduler zieht zufällig Lose und der Gewinner bekommt die Ressource. Prioritäten können dadurch simuliert werden, wer wie viele Lose bekommt. Generell ist es so, dass ein Prozess, der den Bruchteil f der

Lose hat, auch den gleichen Bruchteil f der Ressource durchschnittlich abbekommt. Eine interessante Möglichkeit ist, dass kooperierende Prozesse ihre Lose austauschen können. Damit auch kein Prozess verhungert, bekommen alle eine Mindestanzahl an Losen.

O(1)-Scheduler Der O(1)-Scheduler ist der alte Scheduler unter Linux. Prinzipiell werden zwei Runqueues verwaltet, die der aktiven Tasks und die der inaktiven. Beide dieser Runqueues bestehen wiederum aus einem festen Array von Listen, nach Priorität sortiert. Ein Task, der unterbrochen wird oder dessen Quantum abgelaufen ist, wird in die Runqueue der inaktiven Tasks eingetragen und dabei noch seine Priorität neu berechnet. Wenn die aktive Runqueue leer ist, werden die Zeiger auf die Listen vertauscht und es geht von vorne los.

CFS Der *Completely Fair Scheduler* ist der neue Linux Scheduler. Die Idee ist hierbei, dass alle Prozesse (auch unterschiedlicher Priorität), die im rechenbereiten Zustand sind, in einem einzigen Rot-Schwarz-Baum verwaltet werden. Jede CPU hat dabei seinen eigenen Scheduler mit R-S-Baum. Die Prozesse sind innerhalb des Baums nach der sogenannten virtuellen Zeit einsortiert, die jedes Mal vom Scheduler neu berechnet wird. Es wird dann immer der Task ausgeführt, der die kleinste virtuelle Zeit besitzt (und im R-S-Baum ganz unten links sitzt).

3.3 Echtzeitscheduling

- Scheduling wird Echtzeitscheduling genannt, wenn mehrere oder manche der Prozesse Deadlines haben, die sie erfüllen müssen.
- Ziele: Deadlines einhalten, Vorhersagbarkeit zur Vermeidung von Qualitätseinbußen z.B. im Multimediabereich.
- Man unterscheidet *harte Echtzeitsysteme*, in denen eine verspätete Antwort eine falsche Antwort ist und *weiche Echtzeitsysteme* in denen Verspätungen toleriert werden, aber zu verminderter Qualität führen.
- Man unterscheidet *statische* und *dynamische* Schedulingalgorithmen. Bei ersteren wird die Abfolge bzw. die Prioritäten vor dem Systemstart festgelegt, was voraussetzt, dass alle Informationen über Periode, Laufzeit, Deadlines usw. vorliegen. Bei den dynamischen Varianten wird zur Laufzeit entschieden.
- Ein Echtzeitsystem ist *schedulable*, wenn $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$ gilt, mit C_i CPU-Zeit in Sekunden eines Prozesses und P_i die Periode unter der der Task wiederholt wird. Das Verhältnis $\frac{P_i}{C_i}$ gibt dabei den Anteil der CPU-Zeit an, den der Prozess i verwendet. Wenn diese Bedingung verletzt wird, dann ist es unmöglich ein richtiges Scheduling hinzubekommen.

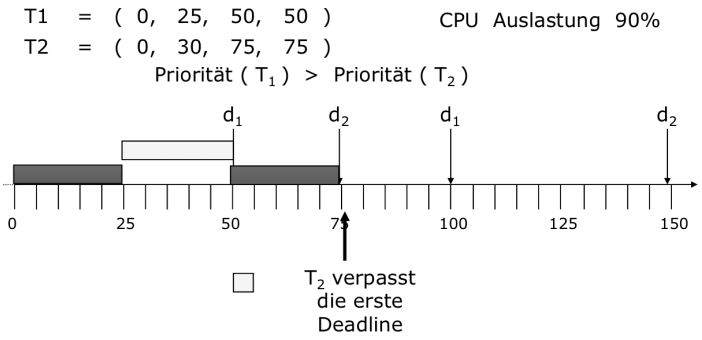
Rate Monotonic Scheduling Dies ist ein statischer Algorithmus für unterbrechbare, periodische Prozesse, die eine feste Priorität zugeordnet bekommen, die linear mit der Anzahl der Ausführungen pro Sekunde (Rate) ist. Der Scheduler wählt

als nächstes immer den rechenbereiten Prozess mit der höchsten Priorität aus und unterbricht falls nötig den aktuell laufenden Prozess.
RMS ist optimal in der Klasse der statischen Verfahren.
Folgende Bedingungen müssen hier eingehalten werden:

- Jeder Prozess muss in seiner Periode fertig werden.
- Keine Abhängigkeiten der Prozesse untereinander.
- Prozesse benötigen pro Periode die gleiche CPU-Zeit.
- Kein nicht-periodischer Prozess hat eine Deadline.
- Unterbrechungen und Kontextwechsel haben keinen Overhead (unrealistisch).

Es kann gezeigt werden, dass RMS nur funktioniert, wenn $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m}-1)$ gilt, für $m \to \infty$ geht das gegen $\ln 2$.

Deadline Monotonic Scheduling Ist auch ein statischer Algorithmus mit festen Prioritäten. Hier sind die Prioritäten umgekehrt Proportional zu Länge der Deadline, also die Anzahl der Deadlines pro Sekunde. Wenn bei DMS die Deadline und die Periode identisch ist, dann sind wir bei RMS.



Earliest Deadline First Dies ist ein dynamischer Algorithmus, in dem die Prioritäten zur Laufzeit angepasst werden. Die Priorität berechnet sich über $P_i(t) = D_i - t$, also dem Abstand zur Deadline. Es wird immer der Prozess ausgewählt, dessen Abstand zur Deadline am geringsten ist. Im Gegensatz zu RMS werden hier keine periodischen Prozesse benötigt und pro Periode müssen auch nicht immer dieselben CPU-Zeiten benutzt werden.
EDF funktioniert für jede ausführbare Menge an Prozessen die schedulable sind, d.h. es kann tatsächlich eine Auslastung von 100% erreicht werden.

Least Laxity First Dies ist ebenfalls ein dynamischer Algorithmus, der die Prioritäten zur Laufzeit anpasst. Hier werden die Prozesse bevorzugt, die den geringste zeitlichen Spielraum haben, also $L_i(t) = D_i - t - E_i$ wobei D_i die Deadline ist und E_i die verbleibende Ausführungszeit.

4 Deadlocks

- Ein *Deadlock* ist ein Zustand, in dem mehrere Prozesse sich gegenseitig blockieren, da sie Ressourcen verwenden wollen, die aber von den anderen bereits genutzt werden und so alle Prozesse in einem ewigen Blockierungszustand festhängen.

- Eine *Ressource* ist dabei etwas, was nur von genau einem Prozess gleichzeitig verwendet werden kann. Die Ressourcen können dabei entweder Hardware (CD-Laufwerk, Festplatte etc.) oder Software (Datei, Datensätze auf Datenbank usw.) sein.
  Man unterscheidet *unterbrechbare Ressourcen*, bei dem einem Prozess, der die Ressource verwendet, ohne ungewünschte Nebenwirkungen die Ressource wieder entzogen werden kann (wie z.B. Arbeitsspeicher) und *ununterbrechbare Ressourcen*, bei denen ein von außen kommendes Entziehen zu einem Fehler führt (z.B. CD-Brenner). Deadlocks haben nur mit ununterbrechbaren Ressourcen zu tun.

- Es gibt vier Voraussetzungen, die alle zusammenkommen müssen, damit ein Deadlock entsteht. Sobald eine der Voraussetzungen nicht erfüllt wird, ist ein Deadlock unmöglich.

  Wechselseitiger Ausschluss Jede Ressource ist entweder verfügbar, oder genau einem Prozess zugeordnet.

  Hold-and-wait Prozesse können mehrere Ressourcen anfordern, auch wenn sie schon welche reserviert haben.

  Ununterbrechbarkeit Ressourcen können nicht gewaltsam dem reservierenden Prozess entzogen werden. Der Prozess muss die Ressource freiwillig wieder abgeben.

  Zyklische Wartebedingungen Prozesse müssen zyklisch auf eine Ressource warten, die dem nachfolgenden Prozess gehört.

- Deadlocks können über einen gerichteten Graphen modelliert werden, den sogenannten *resource allocation graph*. In dem Graph gibt es zwei Arten von Knoten: Die Prozesse P ($\circ$) und die Ressourcen R ($\square$). Eine Kante $p \to r$ drückt aus, dass der Prozess die Ressource anfordert und auf sie wartet, wohingegen die Kante $r \to p$ angibt, dass die Ressource dem Prozess zugeordnet ist, also der Prozess die Ressource belegt.
  Wenn der Graph keinen Zyklus enthält, so ist kein Deadlock vorhanden, wenn er einen Zyklus enthält, und jede Ressource nur einmal existiert, so entsteht ein Deadlock.

- Es gibt vier Varianten Deadlocks zu behandeln:
  - Deadlocks ignorieren, d.h. weder vorbeugen, noch erkennen und beheben. Hier gliedert sich der *Vogel-Strauß-Algorithmus* ein, der besagt, dass man den Kopf in den Sand stecken soll, denn wenn man das Problem nicht sieht, so gibt es das auch nicht. (Wird in Linux / Windows so gemacht, da andere

Verfahren meist zu aufwendig/ große Einschränkungen mit sich bringen und so häufig Deadlocks nun auch nicht unbedingt sind.)

- Deadlocks nicht vorbeugen, aber erkennen und beheben.
- Dynamische Verhinderung durch vorsichtiges Ressourcenmanagement.
- Deadlocks vorbeugen durch Ausschalten einer der notwendigen Voraussetzungen.

4.1 Deadlocks Erkennen und Beheben

Erkennung bei einer Ressource pro Typ Der resource allocation graph wird konstruiert, wenn dieser einen oder mehrere Zyklen enthält, so besteht ein Deadlock.
Der Algorithmus zum Erkennen eines Deadlocks nimmt jeden Knoten als Wurzel eines Baums und durchläuft dann mit Tiefensuche den Graphen. Es wird dabei eine Liste der schon besuchten Knoten geführt und sollte in dieser dann ein Knoten doppelt vorkommen, liegt ein Deadlock vor.

Erkennung bei mehreren Ressourcen pro Typ Hier wird ein Matrix-basierter Algorithmus verwendet. Es werden vier Datenstrukturen geführt:

- Ressourcenvektor E, der in Eintrag i angibt, wie viele Ressourcen von Typ i insgesamt verfügbar sind.
- Ressourcenrestvektor A, der in Eintrag i angibt, wie viele Ressourcen von Typ i noch frei sind.
- Belegungsmatrix C, die in Zelle c_{ij} angibt, wie viele Ressourcen von j der Prozess i belegt.
- Anforderungsmatrix R, die in Zelle r_{ij} angibt, wie viele Ressourcen von j der Prozess i anfordert.

Folgende Invariante gilt dabei $\sum_{i=1}^{n} c_{ij} + A_i = E_i$.
In dem Algorithmus werden Prozesse markiert, die ohne Probleme zu Ende laufen können. Sollten am Ende des Algorithmus noch unmarkierte Prozesse vorliegen, so sind diese an Deadlocks beteiligt. Anfangs sind alle Prozesse unmarkiert.

1. Suche unmarkierten Prozess P_i, für den die i-te Zeile von R komponentenweise kleinergleich als A ist.
2. Wenn ein solcher Prozess existiert, addiere i-te Zeile von C zu A und markiere den Prozess i. Mache weiter mit Schritt 1.
3. Andernfalls beende den Algorithmus.

Beheben

Unterbrechung Einem Prozess kann zeitweise eine Ressource entzogen und diese dann einem anderen Prozess zugewiesen werden. Diese Variante ist sehr schwierig bis unmöglich.

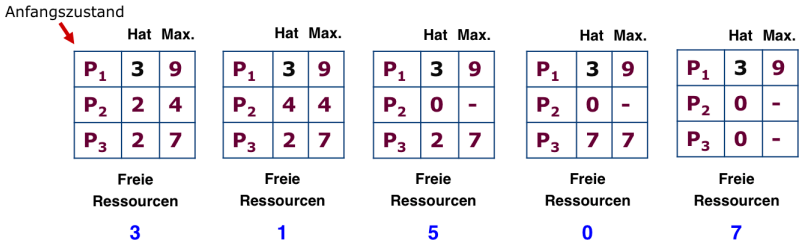
Rollback Es werden regelmäßig Checkpoints der Prozesse angelegt. Sollte ein Deadlock eintreten, kann der Prozess zu einem Checkpoint zurückgesetzt werden, bevor er die kritische Ressource angefordert hat. Durch das Rollback werden natürlich entsprechende Teile dann erneut ausgeführt.

Prozessabbruch Ein Prozess, der an einem Deadlock beteiligt ist, kann einfach abgebrochen werden. Dabei sollte der abzubrechende Prozess sorgsam ausgewählt werden.

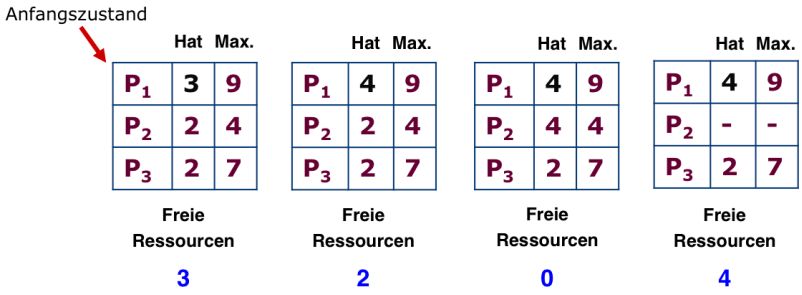
Man muss sich natürlich auch noch überlegen, wann man diesen Erkennungs- und Behebungsprozess anstößt. Bei jeder Ressourcenanforderung wäre dies wohl zu viel Overhead, besser wäre nach k Minuten zu überprüfen bzw. wenn z.B. die CPU-Auslastung unter einen bestimmten Schwellwert fällt.

4.2 Deadlock-Verhinderung

- Ein Zustand heißt *sicher*, wenn kein Deadlock vorliegt und es eine Scheduling-Reihenfolge gibt, die nicht zu einem Deadlock führt. Ein sicherer Zustand garantiert einem, dass alle Prozesse ungehindert zu Ende laufen können.



Ein Zustand ist *unsicher*, wenn diese Garantie nicht gegeben werden kann (es muss allerdings nicht zwangsweise zu einem Deadlock kommen).



Bankier-Algorithmus für einzelne Ressource Es wird der Deadlock-Erkennungsalgorithmus erweitert. Der Algorithmus funktioniert der Idee nach wie ein Bankier, der die Kreditwünsche seiner Kunden behandeln will. Bei jedem Kreditantrag (= Ressourcenanforderung) wird überprüft, ob die Bewilligung zu einem sicheren Zustand führen würde. Wenn ja, wird der Kredit bewilligt, ansonsten wird

sie auf später verschoben. Die Überprüfung wird so vorgenommen, dass geprüft wird, ob noch genügend Ressourcen frei sind. Wenn ja, wird imaginär der Kredit bewilligt und zurückgezahlt und der Kunde, der am nächsten am Limit ist in gleicher weise überprüft usw. Wenn es aufgeht weiß man, dass man in einem sicheren Zustand ist und die Ressource kann tatsächlich bewilligt werden.

Bankier-Algorithmus für mehrere Ressourcen Folgender Algorithmus überprüft, ob ein Zustand sicher ist:

1. Suche Zeile aus R (Anforderungsmatrix) die komponentenweise kleinergleich A (Ressourcenrestvektor) ist. Wenn keine solche Zeile vorhanden ist, dann wird das System in einen Deadlock laufen, da kein Prozess beendet werden kann.
2. Der Prozess, der die Bedingung erfüllt, wird ausgewählt und kann beendet werden. Die von ihm belegten Ressourcen werden freigegeben und auf A aufaddiert. Markiere den Prozess als beendet.
3. Wiederhole 1. und 2. bis alle Prozesse markiert sind (Zustand ist sicher) oder ein Deadlock auftritt (Zustand ist unsicher).

Problem: Theoretisch gute Lösung, in der Praxis quasi nicht umsetzbar, da nur selten im Voraus bekannt ist, wie viele Ressourcen von einem Prozess benötigt werden. Zudem ändert sich die Anzahl der Prozesse ständig und Ressourcen können plötzlich ausfallen oder verschwinden.

4.3 Deadlock-Vermeidung

Aufheben des wechselseitigen Ausschlusses Eine Möglichkeit wäre Zugriff durch Spooler auf einen exklusiven Deamon (z.B. Druckerdeamon) einzuschränken. Da der Deamon nie auf andere Ressourcen wartet, kann es hier zu keinem Deadlock kommen. Spooling geht allerdings nicht immer und auch hier kann es zum Deadlock kommen (z.B. wenn zwei Prozesse um den Spooling-Ordner konkurrieren).

Aufheben von Hold-and-Wait Um zu Verhindern, dass Prozesse auf Ressourcen warten während sie andere belegt halten, lässt sich damit umgehen, dass ein Prozess alle Ressourcen auf einmal zu Beginn anfordert. Bekommt er alle, ist es super, bekommt er nicht alle, reserviert er keine und muss warten. Problem ist hier, dass meistens nicht bekannt ist, wer wie viele Ressourcen benötigt, zudem werden die Ressourcen schlecht ausgenutzt.

Aufheben der Ununterbrechbarkeit Gewaltsame Entzug einer Ressource ist problematisch und meistens unmöglich.

Aufheben von zyklischen Wartebedingungen Die erste Variante wäre, dass ein Prozess nur eine Ressource gleichzeitig halten kann. Bevor er eine zweite bekommt, muss die erste freigegeben werden. Dies ist allerdings nicht annehmbar, wenn man z.B. von einer Festplatte auf die nächste Kopieren will. Die zweite Variante

sieht vor, dass die Ressourcen durchnummeriert werden, und man nur Ressourcen in aufsteigender Reihenfolge anfordern darf. Bei dieser Variante bleibt der resource allocation graph azyklisch, aber es ist nicht sichergestellt, dass man eine für alle passende Nummerierung der Ressourcen findet.

5 Speicherverwaltung

- Die Aufgabe der *Speicherverwaltung* ist es die Speicherhierarchie aus den unterschiedlich schnellen Speicherbausteinen zu verwalten und zu koordinieren. Es wird dabei verfolgt, welche Speicherbereiche benutzt bzw. belegt sind, den Prozessen werden entsprechende Speicherbereiche zugeteilt oder freigegeben. Zudem muss das Swapping verwaltet werden, sollte der Hauptspeicher nicht alle aktiven Prozesse fassen können.

- Es gibt folgende grundlegende Strategien

  Direkte Speicherverwaltung In dem Szenario teilt sich ein einziges laufendes Programm den Speicher mit dem Betriebssystem, dies ist die simpelste Version.

  Partitionierung Um Multiprogrammierung zu erreichen, kann der verfügbare Speicher einfach in n Partitionen aufgeteilt werden, wobei jede Partition dann einem Prozess zugeordnet wird. Bei zu vielen / zu großen Prozessen, muss *Swapping* eingesetzt werden (siehe Abschnitt 5.1).
  Folgende neue Probleme treten hier auf:

  – Speicherschutz: Speicherbereich eines Prozesses soll vor Zugriff von anderen Prozessen geschützt sein (Betriebssystem vor Zugriff aller Benutzerprozesse).

  – Relokation: Adressen, die im Benutzerprogramm angegeben werden, sind nicht absolut, sondern müssen auf den tatsächlichen Speicherbereich angepasst werden (dies kann zur Übersetzungszeit, zur Ladezeit oder zur Ausführungszeit geschehen).

  Beides kann mit der Verwendung von *Basisregister* BR und *Grenzregister* GR gelöst werden. Im Basisregister wird die Startadresse der Partition eingetragen und der Grenzregister mit der Länge der Partition belegt. Bei jedem Speicherzugriff eines Benutzerprozesses wird zur geforderten Adresse das Basisregister addiert und überprüft, ob die generierte Adresse A zwischen $BR \leq A \leq BR + GR$ liegt.

  Segmentierung Segementierung ist ein Speicherverwaltungsschema, das die logische Sichtweise des Benutzers unterstützt. Ein Programm wird so beispielsweise in mehrere getrennte logische Adressräume, die *Segmente* aufgeteilt. Segmente können verschiedenen Größe haben und auch unabhängig voneinander wachsen. Die logische Adresse würde dann aus zwei Teilen bestehen: der Segmentnummer und der Adresse innerhalb des Segments. Segmente

werden dann in einer Segmenttabelle ähnlich wie den Seitentabellen verwaltet. Segmente können unabhängig voneinander verschiedene Rechte besitzen (Schutzbits). Durch Segmente ist auch die Verwendung von gemeinsamen Code o.ä. erheblich vereinfacht (z.B. *shared libraries*). Bei Segmenten hat man das Problem der *externen Fragmentierung*.

Segmentierung und Paging Segmente können hier in einzelne Seiten aufgeteilt werden, um so noch mehr Flexibilität zu erhalten.

Virtueller Speicher Die Grundidee ist, dass der Programmcode und die Daten zusammen größer als der verfügbare Arbeitsspeicher sein können. Das Betriebssystem hält dann nur die Teile des Programms im Hauptspeicher, die aktuell benötigt werden, der Rest kann auf die Festplatte ausgelagert werden. Virtueller Speicher wird oft mittels *Paging* (siehe Abschnitt 5.2) umgesetzt. (Vor virtuellem Speicher gab es *Overlays*, bei denen vom Programmierer das Programm händisch in mehrere Teile aufgesplittet werden musste.)

- Mit *Memory-Mapped-Dateien* können Prozesse Dateien in ihren virtuellen Adressraum einbinden, dadurch werden die zeitaufwendigen Lese- und Schreibbefehle auf der Datei vermieden und die Datei kann so behandelt werden wie ein großen Zeichenfeld im Speicher. Dateien können so auch gemeinsam von mehreren Prozessen benutzt werden, indem sie die entsprechenden Seiten in ihren Adressraum einblenden. Zum Einbinden wird der Systemaufruf `mmap` zu Verfügung gestellt, zum Ausblenden (und Zurückschreiben der Änderungen in die Datei) gibt es den Befehl `munmap`.

- Speicherverwaltung für Kernelprozesse wird oft unabhängig von der Speicherverwaltung für Benutzerprozesse getätigt. Dies hat den Grund, dass oft bestimmte Hardware nicht mit virtuellen Adressen umgehen kann und daher zusammenhängede physikalische Speicherbereiche benötigt. Ein zweiter Grund ist noch, dass die verwendeten Datenstrukturen oft so klein sind, dass sie keine ganze Seite füllen würde und man so enorm viel Platz verschwenden würde.
  Zwei Ansätze gibt es hierfür:

  Buddy System Zu Beginn besteht der Speicher aus einem einzigen kontinuierlichen Segment konstanter Seitenzahl (Zweierpotenz). Wenn eine Speicheranforderung hereinkommt, wird die Anforderung auf eine Segmentgröße in Größe einer Zweierpotenz aufgerundet. Der jeweils untere Teil eines großen Segments wird dabei so lange halbiert, bis man die gewünschte Größe erreicht. Werden zwei benachbarte Segmente freigegeben, so werden sie wieder verschmolzen. Das Problem ist hier, dass interne und externe Fragmentierung entstehen kann. (Linux macht das)

  Slab Allocation Die Idee ist hier, dass man für bekannte Datenstrukturen (z.B. Semaphoren, file descriptors etc.) Platz bereithält. Dazu werden pro Strukturtyp *Caches* verwaltet, die Platz führ mehrere solcher Objekte des gleichen Typs bieten. Die Caches zeigen auf den eigentlichen Speicherbereiche,

die in Form von mehrere hintereinander liegenden Seiten in *Slabs* zusammengefasst sind.

5.1 Swapping

- Swapping wird nötig, wenn mehr Prozesse aktiv sind, als in den Hauptspeicher passen.
- Idee beim Swapping: Lagere den kompletten Speicherbereich eines zur Zeit inaktiven Prozesses auf den Hintergrundspeicher aus und lagere ihn bei Bedarf wieder ein.
- Wichtige Überlegungen im weiteren Sinne: Wie viel Speicher soll für einen Prozess reserviert werden? Z.B. ist immer etwas mehr Speicher sinnvoll, da Prozesse dynamisch wachsen.
- Speicherverwaltungsmethoden

  mit Bitmaps Speicher wird in *Allokationseinheiten* unterteilt. Pro Allokationseinheit wird in der Bitmap ein Bit unterhalten, dessen Wert angibt, ob die Einheit belegt ist (1) oder nicht (0). Wenn ein Prozess k Allokationseinheiten benötigt, muss die gesamte Bitmap nach k 0-Bits hintereinander durchsucht werden. Es ist zu beachten, dass die Größe der Bitmap unmittelbar von der Allokationseinheitsgröße abhängt.
  Es kommt bei großen Einheiten übrigens zur *internen Fragmentierung*, d.h. dass Speicher innerhalb einer Allokationseinheit verschwendet wird, wenn ein Prozess diese nicht ganz ausfüllt.

  mit verketteten Listen Der Speicher wird als verkettete Liste von Segmenten dargestellt. Jedes Segment ist dabei entweder ein durch einen Prozess belegter Bereich (P) oder ein Loch (H). Jedes Segment enthält die Startadresse, die Länge des Segments und einen Zeiger auf den nächsten Eintrag. Wenn ein Prozess k Allokationseinheiten benötigt, dann muss die ganze Liste durchsucht werden und nach einem H mit Mindestgröße k gesucht werden.

- Wie wählt man nun sinnvoll freie Speicherbereiche aus?

  First Fit Die Liste wird einfach linear nach der ersten passenden Lücke durchsucht. Das Segment wird dann in zwei Teile aufgespalten, einen Bereich, den der Prozess belegt, und ein neues leeres Segment mit der Restkapazität.

  Best Fit Die ganze Liste wird durchsucht und es wird das Segment ausgewählt, welches das kleinste ist, in den der Prozess gerade noch reinpasst. Best Fit ist langsamer als First Fit und verschwendet mehr Speicherplatz.

  Worst Fit Quasi das Gegenteil von Best Fit. Es wird immer der größte Speicherbereich ausgewählt, um so möglichst große Löcher zurückzulassen. Dies ist aber die schlechteste von den bisher genannten Varianten.

Quick Fit Es wird hier eine andere Datenstruktur verwendet. Es werden verschiedene Listen der Löcher mit gebräuchlichen Segmentgrößen geführt. So kann sehr schnell ein passendes Loch gefunden werden, aber das Finden und Verschmelzen von Löchern (z.B. bei Prozessauslagerung) ist hier sehr aufwendig.

First Fit und Best Fit haben beide das Problem der *externen Fragmentierung*, da hier jeweils immer kleinere Löcher entstehen, die nicht genutzt werden können. Maßnahmen wie der Speicherverdichtung, die alle Löcher zu einem großen Loch zusammenfasst, sind sehr kostspielig.

5.2 Paging

- Das Programm nutzt *virtuelle Adressen*, die den gesamten *virtuellen Adressraum* bilden. Dieser kann wesentlich größer sein als der tatsächlich physikalisch vorhandene Speicher (oft z.B. 2^{32} oder 2^{64} Bit).

- Die virtuelle Adresse kann nicht vom Speicher direkt behandelt werden, sondern muss erst durch die *MMU* (memory management unit) auf die entsprechende physikalische Adresse abgebildet werden. Für diese Zuordnung wird der virtuelle Speicher in sogenannte *Pages* unterteilt, die ein gleich großes Äquivalent im physikalischen Speicher besitzen (die *Frames*). Es können von Prozessen nur ganze Pages benutzt werden, sodass auch hier interne Fragmentierung auftreten kann. Es wird hardwareseitig ein *present/absent Bit* unterhalten, was angibt, ob die Seite aktuell im Speicher liegt oder nicht. Wenn die Seite nicht im Speicher liegt und angefordert wird, wird ein *Seitenfehler* ausgelöst, dadurch wird das Betriebssystem angewiesen die Seite nachzuladen mittels Seitenersetzungsstrategie. Die Zuordnung von Page zu Frame wird in der *Seitentabelle* verwaltet.

- Die beste Seitengröße p ist abhängig von der Prozessgröße s, und der Größe des Seitentabelleneintrags e. Der Prozess belegt s/p Seiten und dementsprechend $s/p * e$ Platz mit den Seitentabelleneinträgen. Der gesamte Speicherverbrauch von Seitentabelle und interner Fragmentierung ergibt sich als $V = s/p * e + p/2$. Dies muss minimiert werden, um dann die optimale Seitengröße für den Prozess zu finden. Dies ergibt sich als $p = \sqrt{2es}$.

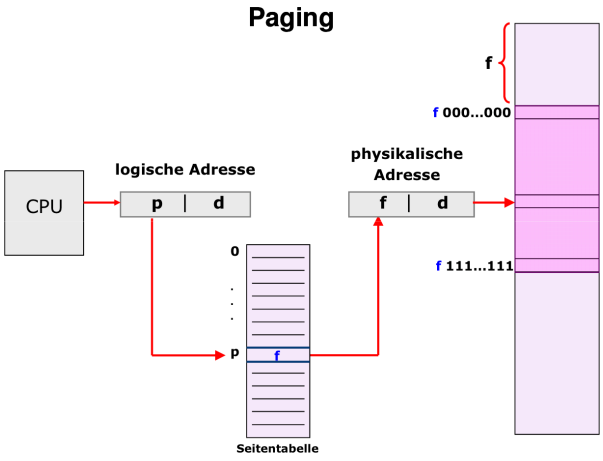
- Bei Timesharing-Systemen o.ä. kann man darüber nachdenken, dass man bei mehreren offenen Instanzen eines Programms die entsprechenden Seiten gemeinsam verwendet (*shared pages*), wie z.B. die Programcode-Teile, die nur lesend von allen Prozessen genutzt werden. Bei `fork` in UNIX müssen Vater- und Kindprozess gemeinsamen Code und gemeinsame Daten anfangs haben. Bei Paging wird dies so gelöst, dass beide Prozesse in ihren Seitentabellen dieselben Frames verwalten. Die Zugriffsrechte werden dabei für beide Prozesse beim `fork` auf readonly gesetzt. Wenn ein Prozess auf eine Seite schreibend zugreifen will, so wird die Seite kopiert, sodass beide Prozesse nun eine eigene Kopie haben (beide Kopien bekommen dann Lese-/Schreibrecht). Diese Methode, bei der nur Seiten kopiert werden, die geschrieben werden, wird *copy-on-write* genannt. Bei `vfork`

wird nicht copy-on-write verwendet, sondern hier hat der Kindprozess denselben Adressraum wie der Elternprozess. Beim Aufruf des Befehls wird der Elternprozess suspendiert. Wenn nun also der Kindprozess etwas schreibt, so ist die Änderung auch für den Elternprozess sichtbar. Wird oft verwendet, wenn danach sofort ein `exec` aufgerufen wird um z.B. so effizient Shells zu programmieren.

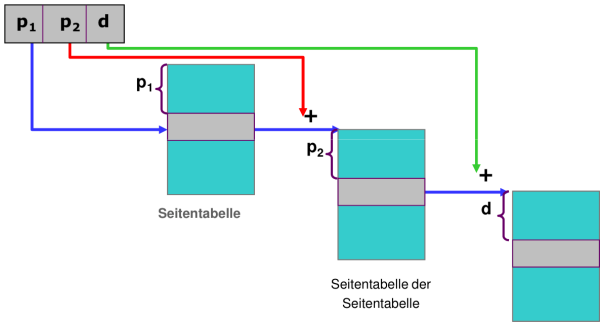
- Paging läuft am besten, wenn jederzeit freie Frames bei einem Seitenfehler zu Verfügung stehen. Durch einen im Hintergrund laufenden *Paging-Deamon* wird regelmäßig der Zustand überprüft und sollten nicht genug Frames frei sein, so werden mit dem Seitenersetzungsalgorithmus Seiten ausgewählt, die ausgelagert werden können. Veränderte Seiten werden so auf die Festplatte zurückgeschrieben. Die Seiten bleiben allerdings noch im Speicher, können so aber bedenkenlos überschrieben werden.

5.2.1 Seitentabelle

- Pro Prozess wird eine Seitentabelle verwaltet.
- Eine virtuelle Adresse teilt sich in zwei Unterbereiche: die *Seitennummer* und den *Offset*. Mit der Seitennummer wird die Seitentabelle indexiert, an der entsprechenden Stelle ist dann die physikalische Anfangsadresse des Frames gespeichert, an der dann unverändert der Offset rangehangen wird. Dies zusammen ergibt dann die korrekte physikalische Adresse.

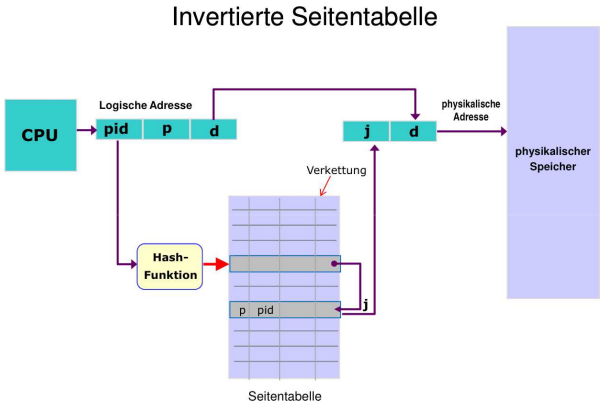


- Seitentabellen können auch mehrstufig angelegt werden. So muss dann bei großen Seitentabellen nicht mehr die gesamte Tabelle vorgehalten werden sondern nur noch der entsprechende Teil. Wenn man z.B. zwei Ebenen hat, dann ist die virtuelle Adresse nun dreigeteilt. Der erste Eintrag indexiert den Speicherort der entsprechenden zweiten Tabelle, der zweite Teil dann den gesuchten Eintrag in der zweiten Tabelle. Mit dem Wert und dem Offset kann dann auf die richtige physikalische Adresse zugegriffen werden.



Mit dem selben Prinzip könnten auch noch tiefere Hierarchien umgesetzt werden, aber mehr als drei Stufen sind nicht wirklich sinnvoll.

- Für 32 Bit große Adressräume ist eine Seitentabelle mit $4kB$ pro Seite schon ca. $4MB$ groß, bei modernen 64 Bit Systemen käme man bei entsprechender Seitengröße auf eine Seitentabelle der Größe von $30*10^6GB$, was nicht realisierbar ist. Daher muss man sich hier eine andere Paging-Variante überlegen.
  Die Lösung ist die sogenannte *invertierte Seitentabelle*. Die Idee ist hier, dass man pro physikalischen Frame einen Eintrag in der Tabelle unterhält und nicht wie vorher pro Seite des virtuellen Adressraums. Die Größe der Seitentabelle ist hier also durch die tatsächliche Größe des Arbeitsspeichers festgelegt, sodass bei wesentlich größerem virtuellen Adressraum enorm viel Speicherplatz eingespart werden kann. Die Abbildung einer virtuellen Adresse auf eine physikalische ist hier aber wesentlich komplizierter. Wenn ein Prozess *pid* auf Seite p zugreifen will, muss die gesamte Tabelle nach einem Eintrag mit dem Tupel (pid, p) durchsucht werden. Um diese Suche zu beschleunigen, wird eine Hashtable verwendet, die die virtuelle Adresse als Key hat. Alle virtuellen Adressen im Speicher werden als verkettete Liste mit dem entsprechenden Verweis auf den Frame gehalten (oder es wird ein zusätzliches Verkettungsfeld in die Tabelle eingefügt).



- Ein Seitentabelleneintrag ist meistens 32 Bit groß und folgendermaßen aufgebaut:

	Caching	R-Bit	M-Bit	Protection-Bits	Present/ Absent	Frame

Caching gibt an, ob die Seite gecacht werden darf oder nicht (wichtig, wenn Pages auf Geräteregister abgebildet werden sollen). Das R-Bit (referenced bit) gibt an, ob auf eine Seite zugegriffen wurde, das M-Bit (modified bit, auch *dirty bit*) wird gesetzt, wenn sie zusätzlich verändert wurde. Wenn es gesetzt ist, muss beim Auslagern auf alle Fälle die Seite zurück in den Hintergrundspeicher geschrieben werden. Die Protection-Bits geben Lese-/ Schreib bzw. Ausführungserlaubnis an. Present/Absent (valid/invalid) wird gesetzt, je nach dem ob die Seite im Speicher liegt oder nicht, falls nicht, wird beim Zugriff ein Seitenfehler ausgelöst. Frame enthält schließlich die Adresse des Frames.

- Seitentabellen können unterschiedlich gehalten werden:

  Registern innerhalb CPU Beim Prozessstart wird die Seitentabelle in die Hardwareregister der CPU geladen und während der Prozess läuft, müssen für die Umrechnung in die physikalische Adresse keine teuren Speicherzugriffe durchgeführt werden, was schnell und einfach ist. Allerdings ist dies unpraktikabel für große Seitentabellen, da diese bei jedem Kontextwechsel komplett geladen werden müssen, was dann wieder teuer ist.

  Im Hautspeicher Die Seitentabelle liegt komplett im Hauptspeicher und im Register wird nur der *Page Table Base Register* und der *Page Table Lenth Register* geführt, die den Speicherbereich der Seitentabelle im Speicher angeben. Der Vorteil ist, dass ein Kontextwechsel schnell geht, da nur diese zwei Register neu geschrieben werden müssen, der Nachteil, dass sich die Speicherzugriffe verdoppeln.

  TLB Der *Translation-Lookaside-Buffer* ist ein schneller Assoziativspeicher innerhalb der MMU, in dem eine kleine Anzahl an Tabelleneinträgen (ca. 64) zwischengespeichert wird, sodass oft verwendete Seiten nicht jedes Mal mit einem Speicherzugriff behandelt werden müssen. Bei jeder virtuellen Adresse wird also zunächst im TLB nachgeschaut, ob dort der Eintrag vorhanden ist (dies geschieht parallel), wenn ja kann direkt der Wert ausgelesen werden, ansonsten muss halt tatsächlich auf die Seitentabelle zugegriffen werden. In diesem Fall wird für spätere Zugriffe die Zeile auch in den TLB gespeichert. Bei jedem Kontextwechsel muss der TLB komplett gelöscht werden.

5.2.2 Seitenersetzungsalgorithmen

- Ein Seitenersetzungsalgorithmus ist dafür zuständig bei einem Seitenfehler die Seite auszuwählen, die durch die neue Seite ersetzt werden soll. Wurde die auszulagernde Seite geändert, so muss der Algorithmus diese zusätzlich auf den Hintergrundspeicher schreiben. Grundsätzlich möchte man die Seiten auslagern, die nur selten benutzt werden.

- *Demand paging* wird die Strategie genannt, nach der Seiten nur nach Bedarf in den Speicher eingelagert werden.

- Das Prinzip der *Lokalität der Referenzen* besagt, dass Prozesse dazu neigen innerhalb einer zeitlich begrenzten Ausführungsphase auf nur einen relativ kleinen Teil der Seiten zuzugreifen.
- Intuitiv würde man meinen, dass man mit mehr Frames weniger Seitenfehler erzeugt. Das ist allerdings nicht unbedingt der Fall, dies wird als *Beladys Anomalie* bezeichnet. Bei der Seitenfolge $0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4$ gibt es bei vier Frames 10 Seitenfehler und bei drei nur 9.
- Seiten müssen im Speicher zum Auslagern gesperrt werden, falls der entsprechende Prozess eine I/O-Anfrage stellt (*I/O interlock*). Die Idee ist hierbei, dass beim Warten auf I/O Ergebnisse nicht versehentlich die Seite ausgelagert wird, in der der Buffer liegt, die die I/O Ergebnisse entgegennehmen sollen.
- Folgende Schritte müssen durchgegangen werden beim Seitenfehler:
  1. Zunächst muss das Betriebssystem rausfinden, welche virtuelle Seite den Fehler erzeugt hat (z.B. durch Nachgucken in entsprechenden Registern).
  2. Dann muss überprüft werden, ob die Adresse gültig ist oder eine Schutzverletzung vorliegt. Wenn ja, dann wird dem Prozess ein entsprechendes Signal geschickt oder es abgebrochen. Wenn alles in Ordnung ist, muss nach einem freien Rahmen gesucht werden und ggf. der Seitenersetzungsalgorithmus angeschmissen werden.
  3. Bei einer modifizierten zu ersetzenden Seite muss diese zunächst zurückgeschrieben werden.
  4. Nun findet das System die Festplattenadresse der zu ladenenden Seite und löst die entsprechenden I/O-Operationen aus, die die Seite in den Frame laden.
  5. Durch ein Interrupt wird das fertige Laden angezeigt und die Seitentabelle des Prozesses wird entsprechend aktualisiert.
  6. Der Prozess wird an der Stelle neu gestartet, an der durch den *trap* die Ausführung unterbrochen wurde.

Optimaler Algorithmus Der optimale Algorithmus müsste wissen, wann auf welche Seite in der Zukunft zugegriffen wird. Man ersetzt in dem Szenario dann einfach die Seite, deren nächster Zugriff am weitesten entfernt ist. Dies ist aber natürlich unmöglich zu implementieren, dient aber als Vergleichsmöglichkeit mit anderen Algorithmen.

NRU Not-Recently-Used basiert auf den R- und M-Bits der Seitentabelle. Aus den Kombinationsmöglichkeiten ergeben sich vier Klassen: Klasse 0 - nicht referenziert und nicht geändert, Klasse 1 - nicht referenziert, modifiziert, Klasse 2 - referenziert und nicht modifiziert und Klasse 3 referenziert und modifiziert. Der Algorithmus entfernt dann einfach eine Seite aus der niedrigsten nichtleeren Klasse.

FIFO Bei First-In-First-Out wird wird eine Liste der Seiten im Speicher verwaltet. Eine neue Seite wird immer ganz hinten angehangen und die vorderste wird ausgelagert. Das Problem ist hier, dass Seiten ausgelagert werden, obwohl sie häufig benutzt werden.

Second-Chance Second-Chance versucht FIFO so zu erweitern, dass häufig besuchte Seiten nicht ausgelagert werden. Dazu wird auch eine Liste verwaltet der im Speicher liegenden Seiten. Von der vordersten Seite wird zunächst das R-Bit geprüft, ist dies nicht gesetzt so wurde sie im aktuellen Zyklus nicht verwendet und wird ausgelagert. Ist es gesetzt, so wird die Seite ans Ende verschoben und das R-Bit gelöscht und der Vorgang so lange wiederholt, bis man eine auszulagernde Seite findet. Beim Einlagern wird das R-Bit der neuen Seite zunächst auf 0 gesetzt und ans Ende der Liste gehangen. Durch das ständige Verschieben der Elemente ist das recht ineffizient.

Clock Um die Second-Chance effizienter zu gestalten, wurde die Liste zu einem Kreis geschlossen und es wird nur noch ein Zeiger verwaltet, der auf die älteste Seite zeigt (entspricht Anfang der Liste bei Second-Chance). Wenn das R-Bit 0 ist, wird die entsprechende Seite ersetzt und der Zeiger eins weiter gerückt, im anderen Fall wird das R-Bit gelöscht und der Zeiger eins weiter gerückt und die Prozedur wiederholt.

LRU Least-Recently-Used ersetzt die Seite, die am längsten nicht verwendet wurde, da man davon ausgeht, dass die letzten Seiten, die benutzt wurden, auch für die nächsten Zugriffe interessant sein werden. Es wird eine Liste über alle Seiten im Speicher geführt, wobei ganz vorne die Seite ist, die zuletzt benutzt wurde und die letzte die, die am längsten nicht mehr gebraucht wurde. Das Problem ist hierbei, dass die Liste bei jedem Speicherzugriff aktualisiert werden muss und dabei die gesamte Liste durchgegangen. Man kann dies auch in Hardware mit einer $n \times n$-Bitmatrix implementieren, wobei n die Anzahl der Seiten ist. Für jeden Zugriff auf Seite i setzt man die i-te Zeile auf 1 und dann die i-te Spalte auf 0 (initialisiert wird die Matrix mit 0en). Zu jedem Zeitpunkt entspricht dann die Zeile mit dem niedrigsten Binärwert der am längsten nicht benutzen Seite.

NFU Not-Frequently-Used unterhält für jede Seite eine Zähler, der zu Beginn 0 ist. Bei jeder Timerunterbrechung wird zu dem Zähler das aktuelle R-Bit addiert, damit wird versucht mitzuzählen, wie oft auf eine Seite zugegriffen wurde. Die Seite mit dem niedrigsten Zählerstand wird dann ersetzt. Das Problem ist hier, dass niemals Zugriffe vergessen werden. Wenn z.B. zu Beginn auf Seiten sehr oft zugegriffen wird aber später gar nicht mehr, so kann es sein, dass der Zähler trotzdem hier so hoch ist, dass immer andere Seiten ersetzt werden.

Aging Eine Verbesserung von NFU, bei der das Problem des Nicht-Vergessens behoben werden soll. Dabei wird der Zähler vor der Addition des R-Bits nach rechts geschifftet und dann das R-Bit nicht wie üblich an das niedrigwertigste Bit addiert, sondern an das höchstwertigste. Hierbei ist dann aber das Problem, dass

die Anzahl der vergangenen Zugriffe über die Zählergröße begrenzt sind. Sollten zwei Zähler beide 0 sein, so kann hier nicht unterschieden werden, welcher tatsächlich der ältere bzw. weniger benutzte ist.

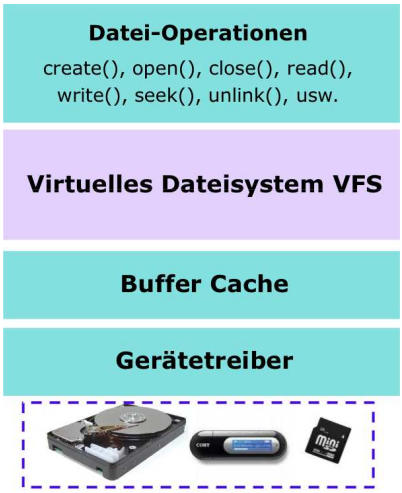
Working-Set Die Menge der Seiten, die zu einem bestimmten Zeitpunkt durch einen Prozess benutzt werden, werden als *Working-Set* (Arbeitsbereich) bezeichnet. Liegt der gesamte Arbeitsbereich einer Phase im Speicher, so treten bis zum Übergang zur nächsten Phase nur wenige Seitenfehler auf. Ist der Speicher zu klein um den kompletten Arbeitsbereich zu fassen (oder die Arbeitsbereichgröße k zu klein gewählt) so werden viele Seitenfehler erzeugt und der Prozess läuft wesentlich langsamer ab. Dieses Phänomen wird als *Thrashing* (Seitenflattern) bezeichnet.
Die Idee beim *Working-Set-Modell* ist nun, dass man sich beim Auslagern eines Prozesses den aktuellen Arbeitsbereich merkt, damit dieser beim Wiedereinlagern komplett geladen werden kann, bevor der Prozess weiterarbeitet. Das im Voraus laden von Seiten nennt man auch *Prepaging*.
Die Idee des WS-Algorithmus ist dann die, dass man bei einem Seitenfehler die Seite auslagert, die nicht zum Arbeitsbereich gehört. Der Algorithmus läuft dann wie folgt: Zunächst muss ein festes k gewählt werden, welches das Zeitintervall des Arbeitsbereichs beschreibt. Dann wird bei einem Seitenfehler die Tabelle durchsucht und bei jedem Eintrag, wo das R-Bit gesetzt ist, in das Feld, was den Zeitpunkt des letzten Zugriffs speichert, die aktuelle Prozesslaufzeit eingetragen (d.h. Seite gehört zum WS). Wenn das R-Bit 0 ist, so wird die Differenz von aktueller Prozesszeit und gespeicherter letzter Zugriffszeit genommen und mit dem festen k verglichen um zu entscheiden, ob die Seite zum WS gehört. Wenn die Differenz größer ist, gehört sie offensichtlich nicht dazu und wird ausgelagert. Alle Einträge müssen aber weiter durchgegangen werden um die Zugriffszeiten überall korrekt zu setzen. Wenn keine Seite gefunden wurde, die nicht zum WS gehört, wird die älteste Seite, deren R-Bit 0 ist, ersetzt, ansonsten wird eine zufällige Seite ausgelagert, deren M-Bit möglichst nicht gesetzt ist. Hier ist wieder der Nachteil, dass jedes Mal die gesamte Tabelle durchlaufen werden muss.

WSClock Dies ist ein Algorithmus, bei dem der Clock-Algorithmus um zusätzliche Zeitstempel erweitert wird. Auch hier wird also eine ringförmige Liste verwendet, die zu Beginn leer ist. Bei einem Seitenfehler wird zunächst die Seite unter dem Zeiger betrachtet. Ist das R-Bit gesetzt, so wird es auf null gesetzt, die letzte Zugriffszeit auf den aktuellen Wert gesetzt und der Zeiger weitergerückt und weitergesucht. Ist es 0 und das Zeitintervall seit dem letzten Zugriff zu groß, so wird das M-Bit betrachtet. Wurde es gesetzt, so wird noch die Seite verschont, aber zum Rückspeichern vorgemerkt. Der Zeiger rückt weiter. Ist es nicht gesetzt, so wird die Seite ersetzt. Ist das R-Bit 0 und das Zeitintervall in Ordnung, so wird auch der Zeiger weitergerückt. Ist der Zeiger einmal rumgelaufen und gibt es markierte Seiten, so wurde vermutlich bis dahin eine Zurückgeschrieben (und hat nun also W-Bit 0) und kann ersetzt werden. Wenn keine markiert wurde, so gehören alle zum Arbeitsbereich und eine zufällige Seite wird ersetzt.

Die besten Algorithmen sind Aging und WSClock.

6 Dateisysteme

- Die Aufgabe des Betriebssystems ist es eine Schnittstelle zu Verfügung zu stellen, die für verschiedene Massenspeichermedien funktioniert.
- Es gibt folgende Anforderungen an das Dateisystem:
  - Dateien dürfen flexibel (mit möglicherweise mehreren Namen) benannt werden
  - Dateien sollen persistent sein, insbesondere nach Abschalten / Absturz des Systems
  - Es muss exklusiver Zugriff als auch gemeinsame Benutzung von Dateien möglich sein
  - Zugriff muss möglichst effizient erfolgen
  - Overhead durch benötigte Metadaten soll minimal sein
- Grundlegende Struktur des Dateisystems



6.1 Datei

- Dies geschieht in Form von *Dateien*. Eine Datei ist eine abstrakte Datenstruktur, die die kleinste logische Informationseinheit darstellt, die durch Prozesse erzeugt und benutzt werden können und die persistent gespeichert werden können.
- Die Informationen einer Datei ist in Bytes gespeichert (bei Disk in Blocks). Dateien besitzen zur Identifizierung einen symbolischen Namen (auf Disks hat man

Blocknummer). Man kann auf Dateien Zugriffsschutz definieren (auf der Disk nicht) und die Konsistenz der Daten wird garantiert (bei Disks hat man dies nicht).

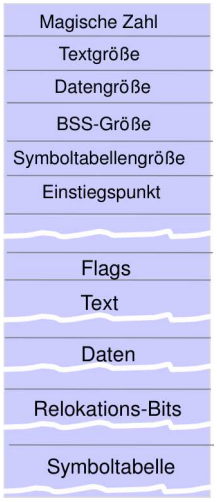
- Dateien können unterschiedlich strukturiert sein:

  Byte-Sequenz Hier liegen die Informationen unstrukturiert als Folge von Bytes vor. Das Betriebssystem weiß nichts über den Inhalt der Datei, die entsprechenden Benutzerprogramme müssen den Daten Bedeutung beimessen. Dies gewährt ein maximales Maß an Flexibilität, was mit Dateien alles angestellt werden kann. Diese Ansatz wird von Windows und UNIX verwendet.

  Record-Sequenz Dateien sind eine Sequenz von Records, wobei Records feste Größe haben. Beim Lesen und Schreiben kann immer nur auf ganzen Records agiert werden.

  Baum Die Datei besteht aus einer baumartig angeordneten Menge von Records. Jeder Record enthält einen Schlüssel, nach dem der Baum sortiert ist.

  Beispielstruktur einer ausführbaren Datei:



- Dateitypen können auf verschiedene Arten durch das Betriebssystem unterstützt werden. Möglichkeiten sind den Dateityp als Teil des Namens zu verwalten (Dateiendung), den Dateityp innerhalb der Datei (als Attribut?) zu verwalten. Jedes Betriebssystem muss mindestens einen Dateityp erkennen und unterstützen und zwar die der eigenen ausführbaren Dateien.
  Ganz grob kann man vier unterschiedliche Dateiarten unterscheiden: *reguläre Dateien*, das sind ganz normale Dateien mit Benutzerinformationen. Daneben gibt es die *Verzeichnisse* zur Strukturierung, *spezielle Zeichendateien* die sich auf serielle I/O-Geräte beziehen und *spezielle Blockdateien*, die blockweise lesbare I/O-Geräte modellieren.

- Neben dem Namen hat eine Datei noch weitere Attribute. Dies sind z.B. der Identifier zur systemweiten Identifikation, der Dateityp, der Ort an dem die Datei auf dem Datenträger zu finden ist, Größe der Datei in Bytes, Zugriffsschutz (z.B. Password, Eigentümer usw.), diverse Zeitstempel (Erstellungsdatum, Änderungsdatum etc.) ...

- Um mit Dateien zu arbeiten, werden eine Menge von Operationen zum Dateizugriff bereitgehalten:

  Create Erstellt eine neue (leere) Datei. Das Betriebssystem muss Platz auf dem Datenspeicher dafür finden und einen Eintrag im Verzeichnisbaum vornehmen.

  Write Die zu beschreibende Datei muss unter Angabe des Namens gesucht werden. Es muss ein Zeiger auf die aktuell zu beschreibende Stelle innerhalb der Datei unterhalten werden.

  Read Auch hier muss der Dateiname angegeben werden. Zusätzlich wird noch ein Buffer verlangt, in denen die gelesenen Daten zwischengespeichert werden können. Auch hier muss wieder ein Zeiger unterhalten werden, der auf die nächste zu lesende Position zeigt.

  Seek Der Zeiger wird innerhalb der Datei auf eine gewünschte Position gerückt.

  Delete Die zu löschende Datei muss zunächst gesucht werden. Dann wird der durch die Datei belegte Platz freigegeben und am Ende muss der Eintrag für die Datei aus dem Verzeichnis gelöscht werden.

  Truncate Erlaubt es nur den Inhalt einer Datei zu löschen.

  Append Anhängen ans Ende der Datei

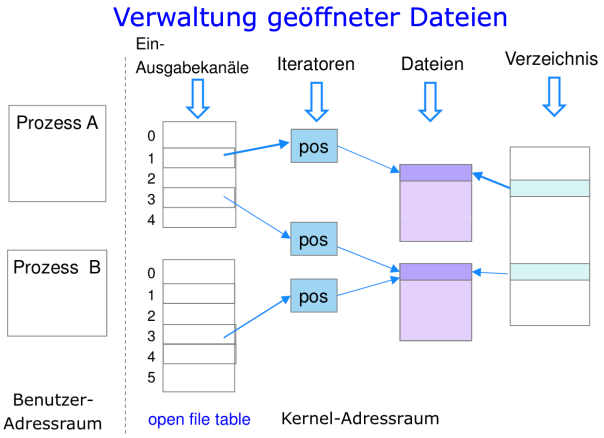
  Rename Umbenennen der Datei

  Set/Get Attribute Setzen und Lesen von Dateiattributen

  Open Ein Prozess muss eine Datei zunächst öffnen, bevor er mit ihr arbeiten kann. Dadurch bekommt er das Zugriffsrecht auf die Datei genehmigt.
  Zunächst muss beim Öffnen das Speichergerät gesucht werden, ist dies gefunden verbleibt noch das Suchen der Datei in seiner Verzeichnisstruktur. Dann müssen die Zugriffsrechte überprüft werden. Ist alle gut, so müssen die Verwaltungsstrukturen der Datei im Hauptspeicher angelegt werden. Schlussendlich wird ein Eintrag in der *open-file table* des Prozesses eingetragen.

  Close Die Operation löscht den Dateieintrag aus der open-file table des Prozesses und gibt so die gehaltenen Ressourcen frei.

- Zur Verwaltung von geöffneten Dateien unterhält das Betriebssystem zwei Tabellen: *per-process table* und *system-wide table*. In der ersten Tabelle werden alle Dateien eingetragen, die ein Prozess gerade verwendet inkl. Interatur und Zugriffsmodus und ein Link auf die system-wide table. In der letzteren werden prozessunabhängige Informationen über die Dateien gehalten. Über diese Tabelle kann auch kontrolliert werden, wie viele Prozesse aktuell auf einer Datei arbeiten.



- Das Betriebssystem muss kontrollierte Zugriffsmöglichkeiten anbieten. Beim *mandatory* Zugriff wird durch das Betriebssystem der Zugriff auf eine Datei komplett verweigert, wenn z.B. ein andere Prozess ein Lock der Datei besitzt. Beim *advisory* Zugriff wird zwar der Prozess in diesem Fall gewarnt, darf aber selber entscheiden, ob er auf die Datei zugreift oder nicht.

- Es werden zwei verschiedene Locktypen unterstützt: Das *Standard Lock* verhindert, dass eine bereits geöffnete Datei erneut geöffnet werden kann. Das *Opportunistic Lock* lässt Clients auf lokalen Kopien arbeiten. Sofern ein andere Client das Lock haben möchte, so wird dem aktuell arbeitendem Client das *oplock* unterbrochen, die Änderungen zurückübertragen und das Lock dem neuen Client gewährt.

- Man unterscheidet zwei Zugriffsarten auf Dateien, den *sequentiellen Zugriff* und den *random access*. Bei ersteren können Dateien nur von Beginn an in sequentieller Weise gelesen werden, bei dem letzteren können Dateien in beliebiger Reihenfolge gelesen werden (siehe `seek`).

6.2 Verzeichnisse

- Dateien werden meist in verschiedenen *Verzeichnissen* zur Strukturierung verwaltet. Meist sind Verzeichnisse dabei selbst spezielle Dateien.

- Man kann drei grundlegende Verzeichnisstrukturen unterscheiden:

  Eine Ebene Es gibt ein einziges Verzeichnis (*Wurzelverzeichnis*), dass alle Dateien beinhaltet. Dies ist natürlich sehr einfach zu implementieren, problematisch wird er hier aber z.B. wenn verschiedene Benutzer Zugriff haben und zwei Benutzer Dateien den selben Namen vergeben wollen.

  Zwei Ebenen Um das Problem der Namenskonflikte zu umgehen, kann man zwei Ebenen einführen in dem man pro Benutzer ein eigenes Verzeichnis anlegt und alle Dateien des Benutzers in dem Verzeichnis verwaltet werden.

Hierarchische Verzeichnissysteme Hier wird das Konzept verallgemeinert und die Idee ist Dateien in logischen Gruppen zu arrangieren, d.h. pro Gruppe hat man ein Verzeichnis bzw. eine Hierarchie von Verzeichnissen. Die hierarchische Anordnung entspricht einem Baum oder allgemeiner einen azyklischen gerichteten Graphen, da man beliebige Verlinkungen erzeugen kann (durch `link` und `unlink` Befehle).

- Die *Verzeichnisstruktur* kann als eine Symbol-Tabelle betrachtet werden, mit deren Hilfe die Dateinamen in Verzeichniseinträge übersetzt werden können. So ist es auch möglich, dass ein Dateieintrag in mehreren Verzeichnissen gelistet werden kann. Per `link` wird ein Eintrag in der Tabelle von einer Datei in einem Dateiverzeichnis angelegt, mit `unlink` kann ein solcher Eintrag entfernt werden. Wird der letzte Verweis auf eine Datei gelöscht, so wird auch diese gelöscht.
  In Linux unterscheidet man den *hard links* (alle Veränderungen des Links wirken auf die ursprüngliche Datei) und den *soft links* (symbolische Verknüpfung). Ein Problem bei Links ist, dass beim Löschen der ursprünglichen Datei die Verweise unnütz rumliegen. Eine Lösung wäre Rückwärts-Zeiger von der Datei auf alle seine Links u.ä. Genauso muss man darauf achten, dass keine Kreise entstehen, da es sonst zu Problemen beim Suchen nach Dateien oder beim Löschen kommt. Dies kann man z.B. dadurch machen, dass man nur Links auf Dateien zulässt (blöd) oder in dem man beim Erzeugen überprüft, ob ein Kreis entstanden ist.

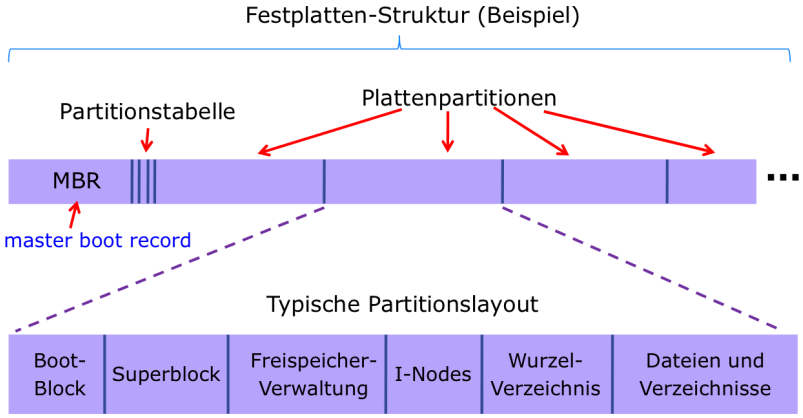
- In UNIX-ähnlichen Systemen können ganze Dateisysteme wie Verzeichnisse eingebunden werden, in dem man sie in die aktuelle Verzeichnisstruktur *mountet*.

6.3 Implementierung von Dateisystemen

- Eine moderne Festplatte besteht aus mehreren *Zylindern*, die durch die *Schreib/Leseköpfe* abgetastet werden. Pro Zylinder gibt es mehrere *Spuren* (tracks), die wiederum in *Sektoren* eingeteilt sind. Die Sektoren haben jeweils feste Größe. Die Gesamtkapazität einer Festplatte ergibt sich als $cylinder * heads * sectors * secsize$

- Die Struktur auf der Festplatte ist allgemein folgendermaßen: In Sektor 0 ist der *MBR* (master boot record) angesiedelt, der beim Booten benötigt wird. Am Ende des MBR liegt die Partitionstabelle, in der für jede Partition (können unterschiedliche Dateisysteme verwalten!) Anfangs- und Endadresse angegeben ist. Eine der Partitionen ist als aktiv gekennzeichnet. Am Anfang jeder Partition befindet sich der *Bootblock*, der bei der aktiven Partition dann geladen wird. Durch die Ausführung des Bootblocks wird das Betriebssystem gestartet. Neben dem Bootblock haben die meisten Dateisysteme noch den *Superblock* (volume control block), der Schlüsselparameter (magische Nummer um Dateisystem zu identifizieren, Anzahl der Blöcke usw.) des Dateisystems bereithält und auch ganz zu Anfang geladen wird. Anschließend kommen dann die Informationen über die freien Blöcke (z.B. in Form von Bitmap oder Liste), dann die *I-Nodes* (Array von Datenstrukturen, wobei jede Struktur pro Datei alle relevanten Informatio-

nen der Datei enthält), dann das Wurzelverzeichnis und alles restliche (hier kann aber die Anordnung von Dateisystem zu Dateisystem variieren).



- Ein Dateisystem besitzt Schichtenarchitektur. Dabei benutzen die Prozesse das *logische Dateisystem* (virtuelles Dateisystem, VFS), in dem die Metadaten der Dateisystemstrukturen verwaltet werden, die file control blocks (FCB) (in Linux I-Nodes) und Schutzmechanismen implementiert sind. Darunter liegen die Dateinverwaltungsmodule, die logische Adressen auf physikalische Blöcke übersetzen und das Freiplatzmanagement übernehmen. Darunter liegt dann das *primitive Dateisystem*, das generische Lese-/Schreibbefehle zu Verfügung stellt und Caches für Verzeichnisse und Dateiblöcke verwalten. Ganz unten liegt dann der *Gerätetreiber*, der die verschiedenen Geräte ansprechen kann.

- Das *VFS* in Linux hat folgende wichtige Datenstrukturen: I-Node-Objekte, file-Objekte für geöffnete Dateien, superblock-Objekte für ein komplettes Dateisystem und dentry-Objekte für Verzeichniseinträge. Für all diese Objekte müssen spezielle Schnittstellen implementiert werden, um unterschiedliche Dateisystem zu unterstützen (z.B. `int open(...)`, `ssize read(...)` usw.).

- Ein *file control block* (FCB) beinhaltet folgende Informationen: Permissions, Zeitstempel, Eigentümer / Gruppe, Größe und Datenblöcke bzw. Zeiger auf Datenblöcke. Bei der Erzeugung einer neuen Datei generiert das logische Dateisystem einen neuen FCB, teilweise wird sogar eine Liste von freien FCBs gehalten, die im Voraus erzeugt wurden.

- Verzeichnisse können auf mehrere Arten implementiert werden:

  Liste Ein Verzeichnis wird als lineare Liste von den Dateinamen mit ihren entsprechenden Verweisen (auf z.B. I-Nodes) implementiert. Dies ist sehr einfach umzusetzen, aber die Suche nach einer Datei benötigt $O(n)$.

  Hashtable Zur Beschleunigung kann man eine Hashtable verwalten. Jeder Dateiname wird gehasht und in die entsprechende Zeile eingetragen (bzw. an

die schon vorhandene Liste von Dateien desselben Hashwerts angehangen). Das Problem ist hier, dass man die Größe der Tabelle festlegen muss.

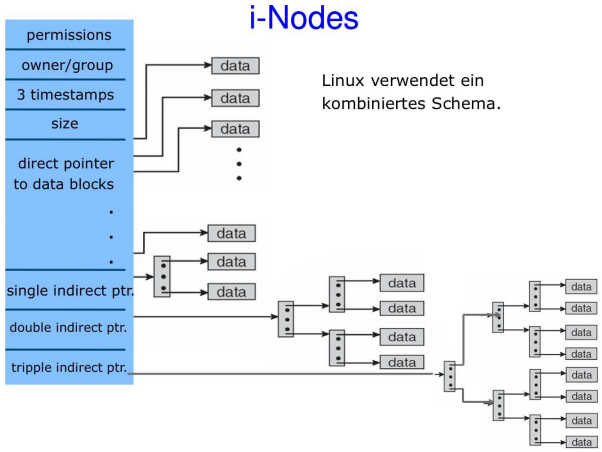
- Wie assoziiert man Plattenblöcke mit Dateien?

  Zusammenhängende Belegung Dateien werden als zusammenhängende Menge von Plattenblöcken abgespeichert. Dies ist sehr einfach zu implementieren (Dateiblock durch Adresse und Länge definiert) und gewährt sehr schnellen Zugriff auf die Daten, da sequentiell gelesen werden kann und nicht mittels `seek` gesprungen werden muss. Problematisch wird es mit dynamisch wachsenden Dateien. Zudem wird mit der Zeit die Platte fragmentiert (Kompaktifizierung notwendig, ist aber aufwendig) und man muss die freien Speicherblöcke verwalten. Diese Variante wird z.B. bei CD-ROMs verwendet, da dort nur lesend zugegriffen wird.

  Verkettete Blöcke Jede Datei ist eine verkettete Liste von Plattenblöcken. Dabei ist das erste Wort jedes Blocks ein Zeiger auf den nächsten Block. Im Verzeichniseintrag wird nur die Adresse des ersten und letzten Blocks gehalten. Diese Methode hat den Vorteil, dass man kein Problem mit externer Fragmentierung hat. Allerdings ist der wahlfreie Zugriff innerhalb der Datei langsam, da jedes Mal die gesamte Verkettung von Anfang an durchlaufen werden muss. Jeder Verweis verursacht zudem einen neuen Plattenzugriff und ist auch als Overhead in der Speicherung anzusehen.

  FAT (File Allocation Table) Die Zeiger eines jeden Plattenblocks werden in einer Tabelle im Hauptspeicher abgelegt. So geht kein Platz innerhalb der Blöcke für das Speichern der Verlinkung verloren. Zudem ist der wahlfreie Zugriff so viel schneller, da, obwohl immer noch die Verkettung abgegangen werden muss, dies nun im Hauptspeicher getan werden kann und so keine teuren Zugriffe auf die Festplatte nötig sind.

  I-Nodes Jede Datei hat einen Indexblock, sodass wahlfreier Zugriff möglich ist. Zudem tritt keine externe Fragmentierung auf. Außerdem muss im Gegensatz zu FAT nur für die offenen Dateien diese Indextabelle im Speicher gehalten werden. Allerdings muss man den Overhead durch die zusätzliche Indextabelle in Kauf nehmen. Man kann die Indextabelle auch mehrstufig anlegen.



- Verwaltung von freiem Blöcken

  Verkettete Liste Diese Methode verschwendet keinen Speicherplatz, da die Verkettung innerhalb der leeren Blöcke stattfindet. Allerdings ist es schwierig zusammenhängende Speicherbereiche zu finden.

  Bitmap In der Bitmap muss für jeden Block ein Bit verwaltet werden. Je nach dem ob das Bit gesetzt ist oder nicht ist der Block belegt. Es können einfach zusammenhängende Bereiche genutzt werden. Nachteil hier ist, dass man zusätzlich Platz für die Bitmap benötigt.

  Grouping Mehrere freie Blöcke werden in Indexblöcken (pro Block werden n Adressen von freien Blöcken gespeichert) zusammengefasst. Der letzte Block hat jeweils einen Zeiger auf den nächsten Indexblock.

  Counting Zeiger auf zusammenhängende Blöcke und die Anzahl der Blöcke wird gespeichert. Dies kann statt als Liste auch über einen B-Baum verwaltet werden.

  Space Maps In Sun's ZFS System verwendet. Es werden *metaslabs* verwendet, denen jeweils eine Space Map zugeordnet wird. Das ist eine Log-Struktur, die die gesamte Aktivität in zeitlicher Reihenfolge speichert. Freie Speicherbereiche werden mit Counting verwaltet.

- Da Festplattenzugriffe wesentlich langsamer von statten gehen als Speicherzugriffe, muss überlegt werden, wie man die Anzahl der Zugriffe minimieren bzw. beschleunigen kann.
  Eine Möglichkeit ist das *Caching* von Blöcken im Speicher. Hierbei hat man dann ähnliche Problematiken wie beim Paging. Da Cache-Referenzen allerdings relativ unstetig sind, ist es sinnvoll die Blöcke in ihrer LRU Reihenfolge zu verwalten. Zudem kann man durch *Vorausladen* (read ahead) von Blöcken eventuell folgende Zugriffe auf die Festplatte vermeiden (sinnvoll allerdings nur bei sequentiellem Lesen).

Bei der Verwendung von Caches muss allerdings überlegt werden, wie man Schreibzugriffe umsetzt. Beim *Write-Through-Caches* wird sofort die Änderung auch auf die Festplatte übertragen (synchron), im Gegensatz dazu kann das Schreiben asynchron erfolgen, das Problem hierbei ist, dass bei einem Systemausfall Inkonsistenzen entstehen können, wenn Änderungen nur im Cache sind. Daher sollte man regelmäßig den Cache rausschreiben (z.B. wie in UNIX mit `sync` Systemaufruf). Dies kann zusätzlich natürlich noch durch geeignete Schedulingverfahren optimiert werden, sodass z.B. die Plattenkopfbewegung minimiert wird.

- Bei Systemabstürzen soll es möglich sein mögliche Inkonsistenzen aufzuspüren. Um eine Inkonsistenz festzustellen, kann z.B. bei einer Veränderung der Metatdaten des Dateisystems ein spezielles Bit gesetzt werden, das zurückgesetzt wird, wenn die Veränderung erfolgreich beendet wurde. In den Systemen gibt es Prüfungsprogramme, die die Konsistenz checken (z.B. `chkdsk` in Windows oder `fsck` in UNIX) in dem sie die Daten der Verzeichnisstrukturen mit der Blockbelegung überprüfen (und eventuell Fehler beheben). Folgende Szenarien können auftreten:

  vermisster Block Dies hat nur Speicherplatzverschwendung als Folge. Zur Behebung muss die Liste der freien Blöcke aktualisiert werden.

  Datenblock doppelt als frei ausgewiesen Auch hier muss die Liste der freien Blöcke aktualisiert werden.

  Block kommt in zwei Dateien vor Dies ist der problematischste Fall, da das Löschen einer Datei unweigerlich den Block auf die Freiliste setzen würde. Eine Lösung ist den Blockinhalt in einen anderen Block zu kopieren und die Kopie einer der Dateien zuzuweisen. Es ist sinnvoll den Benutzer zu benachrichtigen, da es hier zu unerwünschten Effekten innerhalb der Dateien kommen kann.

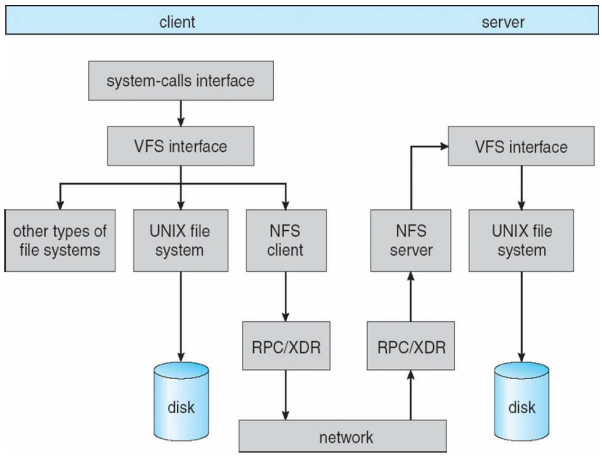
- Mit Backups kann man sich gegen Katastrophen oder dumme Benutzer absichern. Es gibt zwei grundlegende Möglichkeiten: Die *vollständige Sicherung*, bei der alle Daten auf das Backupmedium kopiert werden und die *inkrementelle Sicherung*, bei der nur die Daten gesichert haben, die sich seit der letzten Sicherung verändert haben.

- Da Festplattenzugriffszeit ein Flaschenhals darstellt und durch kleine zerstückelte Schreibzugriffe die Effizienz der Platte auf 1% sinken kann, hat man die Idee der *Log-basierten Dateisysteme* (LFS) entwickelt. Die Idee ist hierbei, dass die gesamte Platte als Log strukturiert wird. Dabei werden alle Schreibzugriffe im Speicher gepuffert und zu gegebener Zeit in einem einzigen Segment zusammengefasst und unter Ausnutzung der Festplattenbandbreite geschrieben, und zwar ans Ende des Logs. Ein solches Segment kann so I-Nodes, Datenblöcke oder Verzeichnisblöcke wild gemischt enthalten, was natürlich die Verwaltung komplizierter macht. Um etwas mehr Überblick zu behalten steht am Anfang jedes Segments eine *Segmentzusammenfassung*. Da I-Nodes über die gesamte Platte

verstreut sind, wird eine *I-Node-Map* verwaltet um schneller die I-Nodes zu finden. Da das Log nicht beliebig wachsen kann, gibt es einen *Clearer Thread*, der regelmäßig das gesamte Logbuch durchgeht und neue Segmente erzeugt, die nur noch die benötigten Daten beinhaltet. Es wird also jeweils das vorderste Segment freigegeben und ein neues Segment mit den aus dem Segment benötigten Daten am Ende des Logs angelegt. So gesehen ist die Platte ein zirkulärer Puffer. Das LFS liefert gute Ergebnisse, ist allerdings inkompatibel zu existierenden Dateisystemen und daher nicht sehr verbreitet.

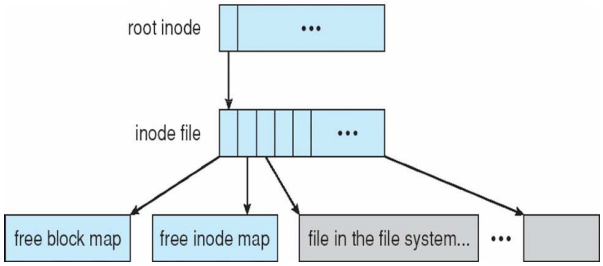
- Bei einem *Journaling Dateisystem* werden alle auszuführenden Aktionen (z.B. Löschen einer Datei etc.) als Transaktion in einen Log-Eintrag zusammengefasst, die in die Log-Datei geschrieben wird. Die Transaktion ist abgeschlossen, wenn die Log-Datei geschrieben wurde. Zu dem Zeitpunkt ist das Dateisystem noch nicht aktualisiert, sondern dieses wird asynchron geändert. Wenn alle Aktualisierungen vorgenommen wurden, so wird die Transaktion aus dem Log gelöscht. Bei einem Systemabsturz müssen so nur alle im Log befindlichen Transaktionen durchgeführt werden. Dazu ist es notwendig, dass die Operationen im Log *idempotent* sind. Beispiele für Journaling Systeme sind NTFS, welches Metadaten Journaling mit automatischer Fehlerkorrektur durchführt und ext3 in Linux, welches drei Modi hat: Jounal, ordered, wo nur Metadaten-Veränderungen gespeichert werden aber die Aktualisierung erst nach Veränderung der Dateiinhalte abgeschlossen ist und writeback, welches im Gegensatz zu ordered auch nach der Aktualisierung des Journals Veränderungen an Dateiinhalten erlaubt.

- Bei *verteilten Dateisystemen* (NFS) erfolgt der Zugriff auf nicht lokale Festplatten völlig transparent. Dies kann über RPCs realisiert werden. Per *Mount Protokoll* werden Dateisysteme übers Netz zu Verfügung gestellt. Über dieses wird eine logische Verbindung zwischen Client und Server erstellt, die nur die Benutzersicht auf das System verändert. Die Dateioperationen werden dann als RPCs zu Verfügung gestellt. Beim Modifizieren von Daten muss zunächst der Server beschickt werden, bevor das Ergebnis an den Client kommt. Das NFS-System hat zwei Schichten, eine virtual file system-Schicht (VFS), die zwischen lokalen und remote Dateien unterscheidet und entsprechende NFS-Protokoll-Prozeduren aufruft und die untere Schicht, die das NSF-Protokoll implementiert.



Bildquelle: Silberschatz, Galvin, Gagne

- *WAFL* (write anywhere file layout) ist ein verteiltes Dateisystem, dass nur in Server-Systemen verwendet wird. Dateien werden über verschiedene Protokolle zu Verfügung gestellt, z.B. NFS, FTP oder HTTP. Jedes Dateisystem hat hier einen Root-I-Node, alle I-Nodes selbst sind in Dateien, freie Blöcke und freie I-Nodes werden auch in I-Nodes verwaltet.



- Die Festplatte wird als einfacher eindimensionales Array von Blöcken adressiert, ein Block ist dabei die kleinste Übertragungseinheit. Die Abbildung der Sektoren in das Array erfolgt sequentiell, beginnend von Sektor 0 (erster Sektor, des ersten Tracks des äußersten Zylinders) über alle Tracks vom äußersten zum innersten Zylinder.

- Disk-Scheduling

  FCFS First-Come-First-Served ist sehr einfach und unvermeidlich, wenn die Festplatte unterbelastet ist. Der Plattentreiber verwaltet eine Tabelle, die über die Zylindernummer indiziert wird und in der alle Operationen pro Zylinder in einer verketteten Liste gespeichert sind. Schlechte Suchzeit, da der Kopf sehr viel hin und herbewegt werden muss.

  SSTF Shortest-Seek-Time-First wird stets die Anfrage ausgeführt, die am nächsten zur aktuellen Position des Kopfs liegt. Dies ist ähnlich zu SJF beim

Prozess-Scheduling und auch hier können Anfragen verhungern. Die Suchzeiten sind hier im Gegensatz zu FCFS besser, aber nicht optimal.

SCAN-Scheduling Der Plattenkopf startet am Ende einer Platte, geht durch zum anderen Ende und bearbeitet alle auf dem Weg liegenden Aufträge, dann geht es in anderer Richtung weiter. Es wird ein Bit benötigt, um die Suchrichtung anzugeben. Anfragen am anderen Ende der Platte müssen länger warten, daher kann man mit dem Suchprozess auch immer von der gleichen Seite beginnen. Dies wird *C-SCAN* genannt. Beide Verfahren sind gut für Systeme mit hoher Belastung der Platte.

C-LOOK verbessert C-SCAN, sodass der Kopf nicht ganz bis zum Rand bewegt wird, sondern nur bis zur Position der letzten Anfrage.

SSTF und C-LOOK sind oft die default-Algorithmen.

- *Swap-Space-Management* muss auch von dem Betriebssystem durchgeführt werden. Der Swap-Space kann entweder als normale Datei innerhalb des Dateisystems implementiert werden, was sehr einfach und flexibel ist, allerdings auch ineffizient. Eine weitere Möglichkeit besteht daher darin, eine extra Partition für das Swapping vorzuhalten. Innerhalb dieser Partition geht es nicht um die Ausnutzung der Speicherbereichs, sondern nur um die Optimierung der Geschwindigkeit. Man muss sich gut überlegen, wie viel Platz man für Swapping bereithält. Solaris empfiehlt als Größe den virtuellen Speicher minus den Hauptspeicher. In Linux bestand die frühere Schätzung in zwei Mal der virtuelle Speicher als Swap-Größe.

- *RAID* (redundant array of independent disks) wird verwendet um Effizient und Zuverlässigkeit zu gewinnen. Die Idee hierbei ist mehrere Platten mit einem RAID-Controller zu einer einzigen Platte zusammenzufassen. Man unterscheidet mehrere Klassen:

  RAID 0 Daten werden in aufeinanderfolgenden Streifen (Stripes) nach einer Round-Robin-Strategie auf alle Platten verteilt. Dies ist eine sehr einfache Variante und die Effizient kann leicht erhöht werden. Allerdings wird das System unzuverlässiger als mit nur einer großen Platte, da die Ausfallwahrscheinlichkeit mit der Anzahl der Platten steigt, denn der Ausfall einer Platte zerstört das gesamte System.

  RAID 1 Dies ist ein echtes redundantes System, alle Daten werden in gleicher Weise auf allen Platten geschrieben, Schreiben benötigt so doppelten Aufwand, aber Lesen kann beschleunigt werden.

  RAID 2 Auf Wortbasis wird der Hamming-Code berechnet und auf allen Platten parallel gespeichert. D.h. die ersten vier Platten erhalten z.B. ein Bit des Wortes und die letzen drei ein Bit des Hamming-Codes.

  RAID 3 Ist eine vereinfachte Version von RAID-2, die nur ein Paritätsbit benutzt, welches für jedes Datenwort gespeichert wird. Das Paritätsbit wird auf eine extra Paritätsplatte geschrieben. Auch hier müssen die Platten synchron arbeiten.

RAID 4 Es wird mit Stripes gearbeitet und eine Stripe-zu-Stripe-Parität auf einer Paritätsplatte gesichert. Wenn ein Sektor geändert wird, müssen alle Laufwerke eingelesen werden und die Parität neu berechnet.

RAID 5 Die Paritätsbit wird Round-Robin mäßig auf alle Festplatten verteilt. Die Wiederherstellung ist aufwendig, aber die Leistung kann gut gesteigert werden und die Redundanz einfach und billig bekommen werden. Daher ist dies die beliebteste RAID-Variante.

7 I/O

- Man kann I/O Hardware grob in drei Kategorien einteilen:

  Blockorientierte Geräte Die Informationen sind in Blöcken fester Größe gespeichert. Jeder Block hat eine eindeutige Adresse, sodass Blöcke unabhängig voneinander adressiert und benutzt werden können. Beispiele sind Festplatten, CD-ROMS, USB-Sticks usw.

  Zeichenorientierte Geräte Diese Geräte arbeiten nur mit linearen Zeichenströmen, insbesondere können Daten nicht adressiert werden und keine Suche durchgeführt werden. Beispiele sind Zeigergeräte, Netzwerkkarten usw.

  Weder noch/ Beides Uhren erzeugen Unterbrechungen in festen Zeitintervallen, und z.B. bei Bildschirmen verschwimmen die Grenzen zwischen Blocks und Zeichen.

- I/O-Geräte bestehen meistens aus zwei Komponenten, dem *Controller* und dem Gerät an sich. Der Controller ist dafür zuständig den seriellen Bit-Strom des Geräts in Byte-Blöcke zusammen zufassen und Fehlerkorrekturen durchzuführen. Die CPU kommuniziert mit dem Controller über spezielle Register innerhalb des Controllers, die die CPU lesen bzw. beschreiben kann. Über diese Register kann die CPU Befehle an das Gerät erteilen (wie Lesen/Schreiben, Ein/Ausschalten usw.). Über Datenpuffer im Controller kann das Betriebssystem zusätzlich Daten an den Controller schicken oder von ihm lesen. Die Kommunikation kann über verschiedene Arten erfolgen.

  I/O Port Bei dieser Methode wird jedem Kontrollregister eine I/O Portnummer zugewiesen. Durch spezielle Ein/Ausgabe-Anweisungen `IN REG,PORT` und `OUT PORT,REG` kann von einem Port in ein Register der CPU gelesen werden bzw. von dem Register in den Port geschrieben werden. Hier unterscheiden sich die Adressräume von Hauptspeicher und I/O-Speicher.

  Memory mapped I/O Bei dieser Methode wird jedem Kontrollregister eine Hauptspeicheradresse zugewiesen, an der kein Hauptspeicher vorhanden ist (meistens im oberen Speicheradressraum), d.h. die Kontrollregister werden in den Hauptspeicher eingeblendet. Es gibt auch hybride Versionen, in denen es memory mapped Datenpuffer gibt und I/O-Ports für Kontrollregister.
  Der Vorteil von memory mapped I/O ist z.B., dass mit den ganz normal bekannten Routinen Speichervariablen geschrieben/ gelesen werden können.

Es sind auch keine speziellen Schutzmechanismen notwendig um Benutzern die Ausführung von I/O zu verbieten. Das Betriebssystem muss lediglich einem Benutzerprozess die Teile des Adressbereichs mit den Kontrollregistern eines Geräts in den Adressbereich des Prozesses einblenden, damit dieser das Gerät nutzen kann. Problem bei memory mapped I/O ist, dass man aufpassen muss, dass die Speicherbereiche der Geräte nicht gecachet werden. Dazu setzt man meistens ein spezielles Bit.

- Der Zugriff auf die Geräte kann verschieden gestaltet werden

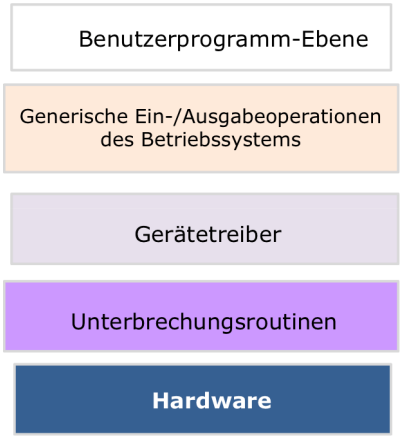
  Polling Zwei Bits werden benötigt, um eine Erzeuger/Verbraucher-Beziehung zwischen Host und Controller zu steuern. Der Host liest in der Schleife das `ready`-Bit, bis dieses auf 0 gesetzt wird. Dann wird das `write`-Bit in das Befehlsregister des Controllers geschrieben und der Host schreibt ein Byte in das `data-out-register`. Ist dies geschehen signalisiert der Host über das Setzten des `command-ready` Bit, dass der Controller arbeiten kann. Sobald dieses Bit gesetzt ist, setzt der Controller das `ready`-Bit auf 1, was für beschäftigt steht. Er liest den Befehlsregister, sieht den Schreibbefehl, und liest die Daten aus dem `data-out-register`. Danach wird das `command-ready` bit auf 0 gesetzt und weitere Statusregister gesetzt. Dann wird noch das `ready`-Bit auf 0 gesetzt und der Host kann mit dem nächsten Byte weitermachen.

  Interrupts Ein I/O Gerät erzeugt einen Interrupt indem ein entsprechendes Signal auf dem Bus abgesetzt wird. Der Interrupt-Controller erkennt dieses und entscheidet das weitere Vorgehen. Im Allgemeinen legt der Controller eine Nummer auf die Adressleitung, welche mitteilt, welches Gerät bearbeitet werden kann und signalisiert dies durch eine Unterbrechungsleitung zur CPU. Wenn die CPU ein Unterbrechungssignal bekommt, unterbricht sie die aktuelle Arbeit und benutzt die Nummer auf der Adressleitung als Index für den *interrupt vector*. Aus diesem wird dann der neue PC geholt, der auf den Beginn der entsprechenden Unterbrechungsroutine zeigt.
  Wenn mehrere Interrupts am Controller anliegen, so muss nach Priorität entschieden werden und ein Interrupt vorerst ignoriert werden. Interrupts werden nicht nur für I/O Ergebnisse verwendet, sondern auch für Ausnahmefehler.
  Die Behandlung von Interrupts wird kompliziert mit Pipelines bzw. superskalaren Prozessoren, da hier der PC keine genaue Abgrenzung von fertigen Anweisungen und noch nicht ausgeführten Befehlen leisten kann. Man kann höchstens die Adresse der nächsten Anweisung, die in die Pipeline gelegt werden soll. Daher muss kompliziert die letzte ausgeführte Anweisung herausgefunden werden.
  Man unterscheidet deshalb zwei Gruppen von Interrupts: Den *präzisen Interrupt*, der eine Maschine in einem wohldefinierten Zustand hinterlässt (d.h. genauer: PC an einer bekannten Stelle gesichert, Anweisungen vor der auf die der PC zeigt wurden vollständig abgearbeitet, kein Befehl nach dem PC wurde bearbeitet und der Ausführungszustand des Befehls auf den der PC

zeigt ist bekannt) und den *unpräzisen Interrupt*, der diese Eigenschaften nicht erfüllt. In letzterem Fall muss eine große Menge von internen Zuständen etc. gespeichert werden, damit das Betriebssystem rausfinden kann, welche Aktionen bereits statt gefunden haben und welche noch nicht. Dies ist natürlich sehr kompliziert und verlangsamt die Interruptbehandlung. Dies führt paradoxerweise dazu, dass schnelle superskalare Prozessoren für den Echtzeitbetrieb ungeeignet werden, da die Interruptbehandlung zu lange dauert.
Manche CPUs erlauben beide arten von Interrupts (z.B. ist ein unpräziser Interrupt möglich, wenn ein Trap aufgrund eines fehlerhaften Programms aufkam, und dieser Prozess so oder so abgebrochen wird).

DMA (direct memory access) Ein *DMA-Controller* steuert den Zugriff zwischen den Geräten und dem Speicher. Dazu hat der DMA-Controller immer Zugriff auf den Systembus, unabhängig von der CPU. Der DMA-Controller besitzt mehrere Register, die vom Prozessor gelesen und geschrieben werden können, wie Speicheradressregister, Byzezähler und mehrere Kontrollregister (bestimmen I/-Port, und die Richtung der Datenübertragung usw.). Zunächst programmiert die CPU die Register des DMA-Controllers, damit dieser weiß, was zu tun ist. Der DMA-Controller leitet den Datentransfer an den Bus zum Plattencontroller, der dann die gewünschten Daten an den DMA-Controller zurückschickt, welcher dann wiederum die Daten direkt in den gewünschten Puffer im Speicher legt. Am Ende der Übertragung wird vom DMA-Controller ein Interrupt abgesetzt um das Ende der Operation zu signalisieren.
Statt den Umweg über den DMA-Controller zu gehen, gibt es auch die Möglichkeit, dass der Plattencontroller direkt in den Speicherpuffer schreibt, diese Variante heißt *fly-by*.

- I/O Schnittstelle wird über ein Schichtenmodell bereitgestellt. Die Schichten erlauben es von den eigentlichen Geräten zu abstrahieren und generische I/O-Operationen zu definieren. Die Gerätetreiber kapseln dann die Unterschiede zwischen den verschiedenen Gerätecontrollern und verbergen so z.B. die Komplexität der Registerbelegungen. Jedes Gerät benötigt seinen eigenen Treiber, die als Teil des Kerns implementiert sind (auch wenn sie vom Hersteller kommen!). Die Schichten haben wohldefinierte Schnittstellen als auch wohldefinierte Funktionalitäten.



- Man kann Geräte unterschiedlich klassifizieren. Einmal z.B. nach Datentrafer-Modus (Zeichenweise oder Blockweise), nach Zugriffsmodus (Sequentiell oder direkt), man kann Unterscheidungen treffen auf Grund von exklusivem oder gemeinsamen Zugriff, Übertragungsgeschwindigkeiten, Ein/Ausgabe-Richtung usw.

- Mit der `ioctl`-Funktion (`int ioctl(int channel, int command, ...)` kann in UNIX ein beliebiges Kommando aus einer Anwendung an einen Gerätetreiber geschickt werden.

- *Netzwerk-Geräte* unterscheiden sich stark von Datei-I/O-Geräten, daher bieten die meisten Betriebssysteme für diese Geräteklasse eine spezielle Schnittstelle an wie den *Socket*. Der Socket abstrahiert vom Kommunikationsprotokoll.

- *Hardwareuhren* haben die Aufgabe die aktuelle Zeit anzugeben oder die vergangene Zeit. Zudem kann man mit *Timern* eine Operation X nach Ablauf einer bestimmten Zeitspanne t durchführen. Es gibt dabei zwei Modi, den Einmalmodus und den Wiederholungsmodus. Die aktuelle Zeit wird immer in einer batteriebetriebenen Sicherungsuhr gespeichert, damit die Zeit auch nach einer Systemabschaltung erhalten bleibt. Hardwareuhren lösen Interrupts in vorgegebenen Intervallen aus.

- Eine *Softwareuhr* hat als Aufgabe die Verwaltung der Echtzeit, Steuerung der Time-Slides eines Prozesses, Buchhaltung über die Prozessornutzung, Behandlung der `alarm`-Systemaufrufe, Überwachung von Betriebssytemaktivitäten und Führung von Statistiken.
  Bei Verwaltung der Echzeit mit 32 Bit bei 60Hz hat man bald einen Überlauf, besser ist daher 64Bit zu verwenden.
  Wenn in sehr kurzen Abständen Timer-Interrupts ausgelöst werden sollen, kommt es zu Problemen. Daher ist eine Lösung immer wenn der Kern im Einsatz ist zu prüfen, ob ein Soft-Timer abgelaufen ist. So ist kein extra Kontextwechsel notwendig. Bei dieser Variante werden Timer immer dann aktualisiert, wenn

Systemaufrufe, Seitenfehler, TLB-Fehler, I/O-Interrupts auftreten oder die CPU im Leerlauf ist. Damit sind $12\mu s$ Intervalle möglich. Sollte es mal länger keine solchen Ereignisse geben, so kann der Soft-Time mit einem Hardware-Timer kombiniert werden.

- Eine I/O-Operation kann *synchron* (blockierend oder nicht blockieren) oder *asynchron* sein. Bei der synchronen blockierenden Variante wartet der Prozess so lange, bis die Operation beendet ist. Diese Variante wird meist in der Benutzerschnittstelle eingesetzt, da es einfach zu verstehen ist. Allerdings sind die meisten Operationen eigentlich asynchron, sodass es die Aufgabe des Betriebssystems ist es nach außenhin blockierend erscheinen zu lassen. Bei nicht blockierenden Operationen liefert der I/O-Aufruf sofort Informationen zurück über das, was übertragen werden konnte, ohne den Prozess zu blockieren. Beim asynchronen läuft der Prozess weiter, während die I/O-Operation durchgeführt wird. Das I/O-Subsystem meldet dann am Ende der I/O-Operation per Signal/ Interrupt/ Call-Back dies an den Prozess.

- Folgende Dienste muss das I/O-Subsystem des Kernels anbieten:

  I/O Scheduling Das Scheduling optimiert die Abarbeitungsreihenfolge von I/O Anforderungen mit dem Ziel den Durchsatz zu erhöhen und die durchschnittliche Wartezeit zu verkürzen. Es werden hierzu Warteschlangen pro Gerät verwendet, die Reihenfolge innerhalb der Warteschlange kann dabei durch den Scheduler geändert werden. Zusätzlich wird die *device-status table* verwaltet, die einen Eintrag pro Gerät enthält, und von denen aus die Warteschlangen verlinkt sind.

  Buffering Es werden Puffer verwendet um Daten zu speichern, die zwischen Programm und Gerät transferiert werden mit dem Ziel die Geschwindigkeitsunterschiede auszugleichen oder unterschiedliche Größen an Datentransfereinheiten anzugleichen. Zudem kann mit Buffering Nebenläufigkeit von Verbraucher und Erzeuger der Daten erzeugt werden. Eine Technik ist z.B. das *double buffering* bei dem der Schreiber zwei Puffer hat, die abwechselnd beschrieben werden, sodass währenddessen aus dem anderen (gerade nicht beschriebenen) gelesen werden kann.

  Caching

  Spooling Beim Spooling werden Geräte-Ausgaben zwischengespeichert, wenn Anfragen nur sequentiell abgearbeitet werden können (z.B. beim Drucker).

  Geräte Reservierung Exklusiver Zugriff muss auf die Geräte gewährleistet werden, zudem sollen natürlich Verklemmungen verhindert werden.

  Fehlerbehandlung Fehler sollen über eine eindeutige Nummer identifiziert werden können, da die Fehler Gerätespezifisch sind sollen sie vom Treiber behandelt werden. Andere Fehler werden weiter oben im I/O System behandelt. Loggen von Fehlern wäre auch wünschenswert.

I/O Schutzmechanismen I/O-Operationen sind nur als priviligierter Nutzer durchführbar, d.h. man muss Systemaufrufe verwenden. Memory-mapped Speicherbereiche und I/O-Ports müssen vor direktem Benutzerzugriff geschützt werden.

- *Streams* sind bidirektionale Kanalverbindungen zwischen einem Benutzerprozess udn einem Gerätetreiber. Der Stream besteht aus dem *stream head*, der die Schnittstelle zum Prozess stellt, dem *driver end*, welches die Schnittstelle zum Treiber ist und einem oder mehreren *stream modules* die sich dazwischen eingliedern. Jedes module hat eine *read-* und eine *write queue*, die die Kommunikation mittels Nachrichtenverkehr zwischen den Warteschlangen realisiert. Die Warteschlangen können zusätzlich *flow control* unterstützen.
- Bei Linux sind Geräte übers Dateisystem ansprechbar und lassen sich mit denselben Systemaufrufen ansprechen wie Dateien. So gibt es aus Anwendersicht keinen Unterschied zwischen I/O-Gerät und Dateien.

8 Schutz und Sicherheit

- Alle *Objekte* sind über eindeutige Namen identifiziert und es gibt eine endliche Menge von *Operationen*, die von Prozessen auf den Objekten ausgeführt werden können. Der *Schutzmechanismus* muss nun dafür sorgen, dass auf die Objekte nur von den Prozessen zugegriffen werden kann, die die entsprechenden Berechtigung haben. Es gibt eine Reihe von Schutzmechanismen, die diese Policys durchsetzen.

  Schutzdomäne Eine *Domäne* ist eine Menge von *Zugriffsrechten*. Zugriffsrechte sind dabei Tupel von Objektname und Rechtemenge. Oft ist die Domäne benutzerspezifisch und enthält alle Rechte eines Benutzers. Es ist zu beachten, dass jeder Prozess zu jedem Zeitpunkt in einer Schutzdomäne läuft.
  In UNIX wird die Domäne über die Angabe von UID und GID definiert, d.h. alle Objekte mit der selben (UID, GID)-Kombination, liegen in derselben Domäne. Bei UNIX besteht ein Prozess aus zwei Hälften, dem User-Teil und dem Kernel-Teil. Wenn ein Systemaufruf durchgeführt wird, dann wechselt der Prozess in den Kernel-Teil, wobei implizit auch ein Domänenwechsel stattfindet. Ein Prozess kann die Domäne wechseln, wenn er z.B. eine `exec` auf einer Datei aufruft, bei der das `SETUID`- oder `SETGID`-Bit gesetzt ist. Dabei erhält der Prozess nämlich UID und GID der Datei und wird mit diesen Rechten ausgeführt.
  In MULTICS werden Domänen über 8 Ringe definiert, und jeder Prozess wird einer aktuellen Ringnummer zugeordnet. Auch hier ist ein Domänenwechsel möglich, wenn ein Prozess eine Prozedur aus einem anderen Ring aufruft.
  Die Domänen können konzeptionell in einer großen Matrix verwaltet werden, in der in den Zeilen die Domänen angegeben sind und in den Spalten die Objekte. Innerhalb der Zellen sind dann die entsprechenden Rechte gelistet, die die Domäne auf das Objekt hat. Der Domänenwechsel kann auch

repräsentiert werden, in dem man die Domänen selbst als Objekte hinzufügt. Wenn von Domäne i in Domäne j gewechselt werden kann, muss in der Zelle (i, j) ein `switch`-Recht eingetragen sein. Um Zugriffsrechte zu ändern, benötigt man Zugriffsrechte auf das Zugriffsmatrix-Objekt.

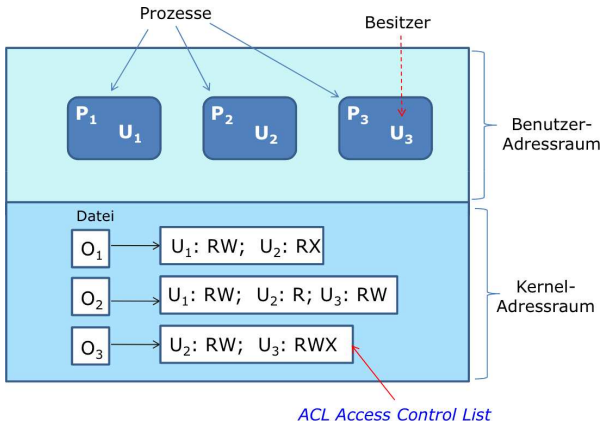
copy Kann Zugriffsreche von einer Domäne in eine andere Domäne innerhalb derselben Spalte kopieren (man unterscheidet transfer, bei dem das Recht transferiert wird und vom ursprünglichen Halter entfernt, limited copy, bei dem Zugriffsrechte von dem Empfänger nicht weiter kopiert werden können und copy, was die uneingeschränkte Nutzung zulässt).

owner Zugriffsrechte können addiert, gelöscht oder verändert werden d.h. ein Prozess kann ein beliebiges Zugriffsrecht in der entsprechenden Spalte bewirken.

control Das Recht ist nur auf Domänen-Objekte anwendbar. Wenn es gesetzt ist, kann ein Prozess in der Domäne ein beliebiges Zugriffsrecht in der Zeile der Domäne verändern.

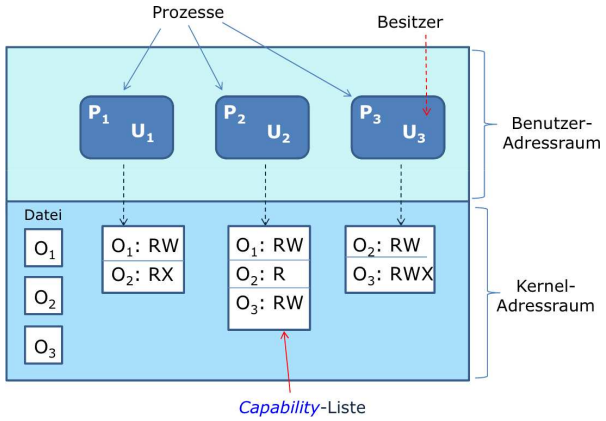
Mit copy und owner kann man Zugriffsrechte kontrolliert propagieren lassen, aber dies ist kein geeignetes Werkzeug dafür, dass keine Informationen, die in den Objekten gespeichert sind, außerhalb der Ausführungsumgebung des Objekts migrieren können. Dies ist das *confinement problem*.

Zugriffskontrolllisten Zugriffskontrolllisten (ACL) sind eine spaltenweise Implementierungsart der Zugriffsmatrix. Hier wird pro Objekt eine Liste geführt, die alle Domänen enthält, die auf das Objekt zugreifen können (mit entsprechenden Rechten).



Capabilities Dies ist die zeilenweise Darstellungsform der Zugriffsmatrix, d.h. man gibt die Fähigkeiten einer Domäne an. C-Lists sind selber wieder Objekte, sodass auch auf diesen prinzipiell Rechte gewährt werden können. Dies muss aber in geeigneter Form passieren. Z.B. kann man mittels *tagged*-Bit hardwareseitig nur das Verändern im Kernelmodus erlaubt werden. Oder

die C-Listen können komplett innerhalb des Betriebssystems gehalten werden. Die letzte Möglichkeit ist es die C-Listen zwar im Benutzeradressraum zu lagern, aber diese kryptographisch zu schützen.



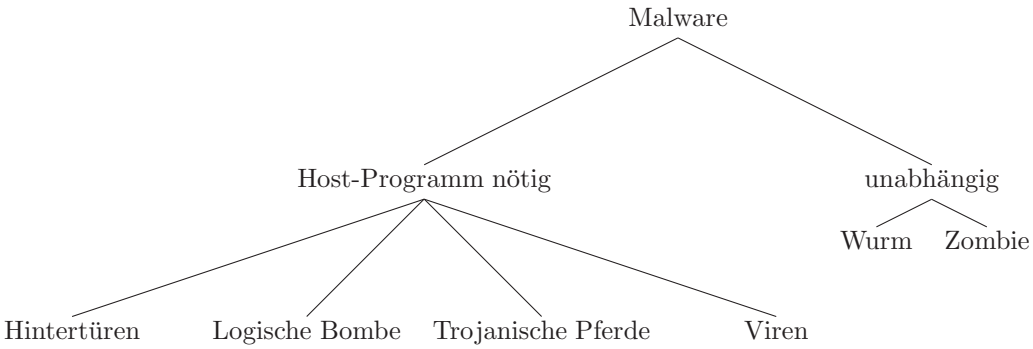
Mit Capabilities ist es schwierig einzelne Rechte selektiv zu entziehen. Das Hydra-System ist z.B. Capabilities-basiert.

Sprachbasierter Schutz Auf Programmiersprachen-Ebene können Schutzkontrollen definiert werden. Damit ist es möglich Sicherheitsanforderung sehr flexibel und passend auf eine Anwendung zuzuschneiden. Der Schutz wird deklarativ spezifiziert, sodass die Anforderungen unabhängig vom Betriebssystem beschrieben werden können. Die Durchsetzung der Schutzmechanismen übernimmt das Betriebssystem, der Softwareentwickler muss nichts tun. Java bieten z.B. solch einen Schutz, der durch die VM gewährleistet wird.

- Schutzmechanismen beziehen sich auf die spezifischen Mechanismen und Methoden des Betriebssystems, die verwendet werden um Sicherheit zu erzielen. Folgende Sicherheitsanforderungen gibt es, die ans Betriebssystem gestellt werden: Vertraulichkeit der Daten, Datenintegrität, Systemverfügbarkeit und Authentizität. Auftretende Sicherheitsverletzungen können dementsprechend den verschiedenen Anforderungen zugeordnet werden. Zusätzlich kann noch Diebstahl von Diensten auftretend oder Störung der Funktionalität, in dem Dienste arbeitsunfähig gemacht werden.

- Standard Angriffe sind die *Unterbrechung* der Kommunikation, das *Mithören*, das *Verändern der Daten* und das *Erzeugen*.

- Taxonomie der Malware



- Angriffe auf ein System können entweder von Außen erfolgen (durch Ausnutzung von Programmfehlern, Brechen des Sicherheitssystems usw.) oder von Innen in Form eines *Insider-Angriffs* (über Hintertüren, logische Bomben usw.).

  Logische Bombe Hierbei handelt es sich um eingebetteten Programmcode, der nur unter bestimmten Bedingungen ausgeführt wird (oft benutzt um Arbeitgeber zu erpressen).

  Hintertüren Hierbei benutzt man ein Programms ohne die üblichen Sicherheitsüberprüfungen zu durchlaufen. Diese Hintertüren sind schwierig durch das Betriebssystem zu kontrollieren, daher müssen sich Sicherheitsmaßnahmen auf Softwareentwicklungsebene konzentrieren.

  Ausnutzen von Bufferoverflow Der Bufferoverflow ist einer der meist ausgenutzten Programmierfehlern. Bufferoverflow geht natürlich nur in Sprachen wie C, wo Indexgrenzen nicht überprüft werden und man somit fremde Speicherbereiche überschreiben kann. Die Idee bei einem solchen Angriff ist, dass man den Puffer mit kompiliertem Schadcode überschreibt, der dann möglichst die Rücksprungadresse mit erwischt und dann als nächstes den Schadcode ausführt. Eine simple Gegenmaßnahme in C wäre z.B. stets die Länge der in den Puffer einzulesenden Daten zu überprüfen (`sizeof`). Mittlerweile gibt es auch Compiler-Unterstützung mit *Canary-Zahlen* (Überprüfungszahl vor der Rücksprungadresse eingefügt) oder einer Sicherheitskopie der Rücksprungadresse.

  Ausnutzung von Format-String Wenn ungefilterte Benutzereingaben z.B. in `printf` verwendet werden, kann es auch zu Problemen kommen.

  Trojanische Pferde Dies sind scheinbar nützliche Programme, die verborgenen Progrogrammcode enthalten, der bei Aktivierung schädigende Funktionen ausführt.

  Viren Ein Virus ist ein Programm, das sich vervielfältigt in dem es sich an den Code anderer Programme anheftet. In UNIX sind Viren nicht so verbreitet, da ausführbare Programme mit Schreibschutz vom Betriebssystem gespeichert werden. Ein *virus dropper* ist ein Trojaner, der einen Virus ins System einfügt.
  Es gibt eine Reihe von verschiedenen Virentypen:

Datei-Viren Der Virus nistet sich am Ende der Datei ein und ändert den Start des Programm in derart ab, dass der Virus-Code zunächst ausgeführt wird. Erst danach wird das eigentliche Programm gestartet.

Bootsektor-Viren Der Bootsektor des Systems wird infiziert und bei jedem Systemstart ausgeführt bevor noch das Betriebssystem gestartet wird. Die funktioniert so, dass zunächst das Virus den Bootsektor an einen sicheren Platz kopiert und sich dann selbst in den Bootsektor schreibt. Beim Systemstart kopiert sich dann der Virus an eine sichere Speicherstelle (z.B. an unbenutzte Interrupt-Vektor-Plätze). Wenn dies getan ist, startet der Virus das Betriebssystem und bleibt im Speicher.

Makroviren Ein Makro ist ein Programm, das in einem Dokument eingebettet ist. Dies sind meistens in interpretierbaren Sprachen geschrieben (Visual Basic etc.), Beispiel ist hier z.B. Lovelettter.

Sourcecode-Viren Verändert Quellcode und ist somit plattformunabhängig.

Gerätetreiber-Viren Ein Gerätetreiber wird hierbei infiziert, der dann automatisch beim Systemstart geladen wird. Zudem laufen Treiber im Kernelmodus und haben so also alle Möglichkeiten das System zu beschädigen.

Polymorphe Viren Diese Viren verändern sich um ihre Erkennung noch schwieriger zu gestalten. So kann z.B. für ein und dieselbe Funktionalität verschiedenen Implementierungen gewählt werden.

Würmer Würmer sind ähnlich wie Viren, können sich aber selbst replizieren. Ein berühmtes Beispiel ist der Morris-Internet-Wurm (von 1988), der aus zwei Teilen besteht: Dem Loader und dem Wurm. Der Loader wird auf einem angegriffenen System kompiliert und lädt dann den Wurm übers Internet nach.

Spyware Sind versteckte „Schnüffelprogramme“, die versuchen Daten (besuchte Websites, Passwörter usw.) auszuspionieren und an eine zentrale Sammelstelle zu senden. Es gibt drei Anwendungsfälle für Spyware: zu Marketingzwecken um gezielt Werbung zu erzeugen, zu Überwachungszwecken (z.B. in einer Firma) und halt als Malware um Benutzerinformationen missbrauchen zu können. Spyware wird häufig über „Unterhaltung für Erwachsene“, Warez-Seiten u.ä. eingefangen.

Rootkit Dies bezeichnet eine Sammlung von Programmen und Dateien, die versucht ihre Existenz zu verbergen. Das Rootkit enthält Schadsoftware oder Hintertüren, die ebenfalls versteckt werden. Firmen verwenden Rootkits teilweise auch dazu um Kopierschutz von Produkten zu erzwingen (Beispiele Sind Sony, und EA u.ä.).

- Mittels *Port Scanning* können Angriffsstellen gefunden werden.

- Gegenmaßnahmen gibt es auch auf Hardware-Ebene: Es gibt z.B. das sogenannte *No eXecute-Bit* (NX-Bit), dass Speichersegmente mit Daten von Speichersegmenten mit Programmcode unterscheiden lassen kann. Das Betriebssystem muss natürlich in der Lage sein, dieses Bit auszunutzen.

- Im Gegensatz zu benutzerbestimmten Zugriffskontrollstrategien (DAC) wird z.B. bei MAC (Mandatory Access Control) die Zugriffsregeln von der Organisation o.ä. festgelegt. Also eine feste Strategie, die durch das System umgesetzt werden muss. Dieser MAC-Mechanismus kontrolliert den Informationsfluss um sicherzustellen, dass Informationen nicht nach außen dringen.

- Multilevel-Sicherheitsmodelle

  Bell-La Padula Modell Das Modell wurde für militärische Sicherheit entwickelt und Objekte unterscheiden sich hier durch ihre Sicherheitsstufe (nicht klassifiziert, vertraulich, geheim, streng geheim), genauso bekommen die Benutzer eine entsprechende Sicherheitsstufe zugesprochen. Es gibt nun zwei Regeln, die den Zugriff auf die Objekte erlauben bzw. verbieten:

  Simple-Security-Regel Ein Prozess mit Sicherheitsstufe k kann nur Objekte mit Sicherheitsstufe $\leq k$ lesen.

  ***-Regel** Ein Prozess mit Sicherheitsstufe k kann nur Objekte mit Sicherheitsstufe $\geq k$ schreiben.

  Biba-Modell Das Bell-La Padula Modell wurde zur Geheimniswahrung entwickelt, nicht um die Integrität der Daten sicherzustellen. Dafür benötigt man quasi die komplementären Regeln:

  Simple-Integrity-Regel Ein Prozess mit Sicherheitsstufe k kann nur auf Objekte mit Sicherheitsstufe $\leq k$ schreiben.

  Integrity-*-Regel Ein Prozess mit Sicherheitsstufe k kann nur Objekte mit Sicherheitsstufe $\geq k$ lesen.

- *Authentifizierung* wird meistens mittel Login-Mechanismus umgesetzt. Da die meisten Leute aber sehr leicht zu ratende Passwörter benutzen, ist dies eigentlich eine blöde Lösung.

- *Firewalls* sind ein Abwehrmechanismus. Bei Hardware-Firewalls (also eigenen Geräten für diesen Zweck) wird entweder der Header jedes eintreffenden Pakets untersucht und entsprechend nach Regeln Pakete verworfen (zustandslos) oder sogar Verbindungen verfolgt und durch Angriffserkennungssysteme gecheckt (zustandsbehaltend). Bei Software-Firewalls wird das Filtern innerhalb des Betriebssystemkerns vorgenommen. Firewalls können z.B. nichts gegen *Tunneling* von verbotenen Protokollen durch erlaubte und auch nichts gegen gefälschte Absender-IPs (*IP-Spoofing*) unternehmen.

- Um Nachrichten sicher durch unsichere Medien zu senden, müssen sie verschlüsselt werden. Bei der *symetrischen Kryptographie* kann aus dem Verschlüsselungsschlüssel der Entschlüsselungsschlüssel bestimmt werden und umgekehrt.

Ein Beispiel ist hier DES (data encryption standard) und dessen Nachfolger wie AES usw. Das Problem ist hier, dass Sender und Empfänger der verschlüsselten Nachricht im Besitz des gemeinsamen geheimen Schlüssels sein müssen. Außerdem könnte durch Brute-Force alle möglichen Schlüssel durchprobiert werden.
Bei der *Public Key Kryptografie* werden unterschiedliche Schlüssel zum Ver- und Entschlüsseln verwendet. Jeder Benutzer hat dabei einen öffentlichen Schlüssel und einen privaten Schlüssel, wobei der öffentliche Schlüssel zum Verschlüsseln und der private zum Entschlüsseln der Nachricht verwendet wird. Ein Beispiel ist hier RSA, der auf der Tatsache beruht, dass es einfach ist festzustellen, ob eine Zahl eine Primzahl ist aber schwierig ist die Primfaktorzerlegung einer Zahl durchzuführen.

- *Intrusion-Detection-Systeme* gehen in drei Schritten vor: Der Wahrnehmung durch Sensoren, die Logdaten oder Netzwerkverkehrdaten sammeln. Diese Daten werden dann im zweiten Schritt mittels Mustererkennung überprüfen und mit einer Musterdatenbank abgleichen und entsprechend im letzten Schritt dann entsprechend drauf reagiert.