System Software Zusammenfassung

Kapitel 1 - 9

UNIVERSITÄT ZÜRICH

27 Dezember 2010

Verfasst von: Patrick

System Software Zusammenfassung

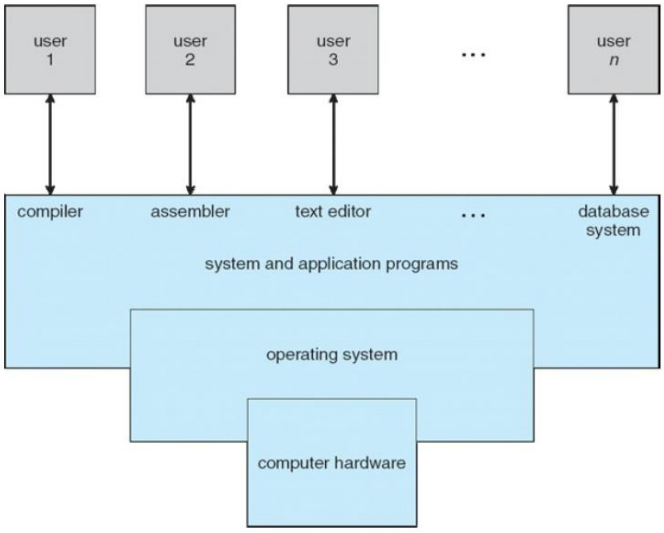
Kapitel 1 - 9

Kapitel 1 – Einführung

Operating System Definition

Computer System besteht aus vier aufeinander bauenden Komponenten:

- Computer Hardware
- Operating System
- System- und Anwendungsprogramme
- User



Operating System ist ein „*Ressource Allocator*“:

- Managed alle Ressourcen
- Entscheidet bei Request Konflikten und stellt effiziente und faire Ressourcenbenutzung sicher

Operating System ist ein „*Control Program*“:

- Kontrolliert Ausführung von Programmen um Fehler und falsche Benutzung zu verhindern/aufzufangen

„das Programm, das immer auf dem Computer läuft“ ist der ***Kernel:***

- Alle anderen sind entweder System Programme (wird mit dem OS mitgeliefert) oder Anwendungsprogramme

Bootstrap program wird beim Starten des Computers geladen:

- Typischerweise in ROM oder EPROM gespeichert, allgemein bekannt als ***firmware***
- Initialisiert alle Aspekte des Systems
- Ladet Operating System Kernel und startet Ausführung

Computer System Operation:

- Einer oder mehrere CPUs, device controllers sind durch einen gemeinsamen Bus miteinander verbunden und haben Zugriff auf shared Memory
- Konkurrierende Ausführungen von Prozessoren und Devices „kämpfen“ um Memory-Zyklen

I/O Devices und Prozessor können gleichzeitig ausführen

Jeder Device Controller hat Verantwortung für einen bestimmten Device Typ

- Zbsp Festplatten, Network Controller, Grafikkarte

Jeder Device Controller hat lokalen Memory Buffer

- CPU schiebt Daten von/zu Main Memory zu/von lokalen Buffer
- I/O schiebt vom Device zum lokalen Buffer des Controllers

Device Controller informiert CPU das es fertig ist mit der Ausführung in dem es einen *INTERRUPT* erzeugt

Computer System Architektur

Die meisten Systeme haben einen single general-purpose Prozessor

- Die meisten Systeme haben auch einen special-purpose Prozessor

Multiprozessor Systeme gewinnen immer mehr an Wichtigkeit

- Auch bekannt als parallel Systems, tightly-coupled systems
- Vorteile:
  1. Erhöhter Durchsatz
  2. Economy of scale
  3. Erhöhte Zuverlässigkeit
- Zwei Typen:
  1. Asymmetric Multiprocessing
  2. Symmetric Multiprocessing

Clustered Systems

Wie Multiprozessor Systeme mit dem Unterschied, dass mehrere Systeme (ganze Computer) zusammen arbeiten
Teilen sich normalerweise einen Speicherplatz via *STORAGE-AREA-NETWORK* (SAN)

Bieten *HOHE VERFÜGBARKEIT* – überlebt Fehler

- Asymmetric clustering hat eine Maschine in hot-standby Modus
- Symmetric clustering hat mehrere Knoten die Anwendungen ausführen und sich gegenseitig überwachen

Einige Cluster sind für high-performance computing (HPC) entwickelt worden

- Anwendungen müssen für Parallelization geschrieben worden sein

Multiprogramming ist für Effizienz wichtig

- Einzelner Benutzer kann CPU und I/O Devices nicht ständig auslasten
- Multiprogramming organisiert Jobs (Code und Daten), so dass CPU immer etwas zum Ausführen hat
- Eine Teilmenge aller Jobs im System werden im Memory behalten
- Wenn OS warten muss (zbsp für I/O), wechselt zu anderem Job

Timesharing (Multitasking) ist eine logische Erweiterung, in welcher CPU so schnell zwischen den Jobs wechselt, dass ein Benutze mit jeden Job interagieren kann

- Reaktionszeit sollte <1 Sekunde sein
- Jeder Benutzer hat mindestens ein Programm ausführend im Memory → *PROCESS*
- Wenn mehrere Jobs gleichzeitig bereit für die Ausführung sind → *CPU SCHEDULING*
- Wenn ein Prozess nicht in den Memory passt, wird dieser via *SWAPPING* rein und raus bewegt
- *VIRTUAL MEMORY* erlaubt Ausführung von Prozessen nicht komplett im Memory

Process Management

Ein Prozess ist ein Programm in Ausführung. Es ist eine Einheit von Arbeit innerhalb des Systems. Ein Programm ist ein passiver Gegenstand, ein Prozess ein aktiver.

Prozesse brauchen für die Ausführung Ressourcen

- CPU, memory, I/O, files
- Initialization data

Der Prozess Termination folgt, dass wiederverwendbare Ressourcen freigegeben werden

Single-threaded Prozesse haben einen *Program Counter*, der jeweils die nächste Instruktion im Code anzeigt (zbsp falls Prozess aus dem Memory raus-geswapped und nachher wieder reingeholt wird)

Multi-threaded Prozesse haben einen program counter *pro Thread.*

Prozess Management Aktivitäten

Das OS ist verantwortlich für folgende Aktivitäten im Zusammenhang mit process Management:

- Erzeugen und Löschen von User und System Prozessen
- Pausieren und Weiterführen von Prozessen
- Mechanismen für Prozess Synchronisation anbieten
- Mechanismen für Prozess Kommunikation anbieten
- Mechanismen für Deadlock Handling anbieten

Memory Management

Alle Daten im Memory vor und nach der Ausführung
Alle Instruktionen im Memory, welche ausgeführt werden
Memory Management entscheidet was im Memory ist, damit:

- CPU Benutzung optimiert wird

Memory Management Aktivitäten

- Merken sich, welche Teile des Memorys in Verwendung sind und von wem diese verwendet werden
- Entscheiden, welche Prozesse und Daten als nächstes in oder aus dem Memory geschoben werden
- Allozieren oder deallozieren von Memory Space falls nötig

Storage Management

OS bietet einheitliche und logische Ansicht vom Speicher
Abstrakte physische Eigenschaften werden zu einer logischen Speichereinheit zusammengefasst → *File*

File System Mangement

Daten werden normalerweise in Ordner organisiert
Zugriffskontrolle um zu entscheiden, wer auf Daten Zugriff haben darf
OS Aktivitäten beinhalten:

- Erzeugen und Löschen von Daten und Ordnern
- Manipulation von Daten und Ordner
- Mapping Daten in sekundären Speicher
- Backup von Daten auf anderes Storage Medium

Kapitel 2 – Operating System Strukturen

OS bietet eine Umgebung für die Ausführung von Anwendungen an. Es stellt gewisse Services bereit. Diese werden nachfolgend erklärt:

User Interface:

- Praktisch jedes OS hat ein UI – einige haben Command-Line (CLI), andere Graphics User Interface (GUI) oder Batch

Program Execution:

- Das System muss in der Lage sein, ein Programm ins Memory zu laden und dieses auszuführen. Es muss auch in der Lage sein, es zu beenden (normal oder anormal).

I/O Operations:

- Ein laufendes Programm kann I/O anfordern, was eine Datei oder ein I/O Device involviert

File System Manipulation:

- File System ist von besonderem Interesse. Programme müssen Daten lesen und schreiben können, wie auch neue Daten erzeugen, bearbeiten oder auch Daten suchen. File System muss auch Zugriffsmanagement anbieten.

Communication:

- Prozess muss eventuell Informationen austauschen – entweder lokal oder mit anderen Computern via Netzwerk
- Kommunikation erfolgt entweder über shared memory oder durch Message Passing

Error Detection:

- OS muss ständig für mögliche Fehler gewappnet sein
- Fehler können im CPU oder im Memory entstehen, aber auch bei I/O Devices oder in der Netzwerkverbindung. Auf Benutzerebene können Anwendungen zbsp durch zu hohe Verwendung von CPU Zeit Fehler erzeugen.
- Für jeden Fehlertyp sollte OS entsprechende Reaktion anbieten

Es gibt noch eine weitere Anzahl an Services und Funktionen, die nicht dem User direkt helfen, sondern vielmehr einen effizienten Betrieb des Systems selber sicherstellen:

Ressource Allocation:

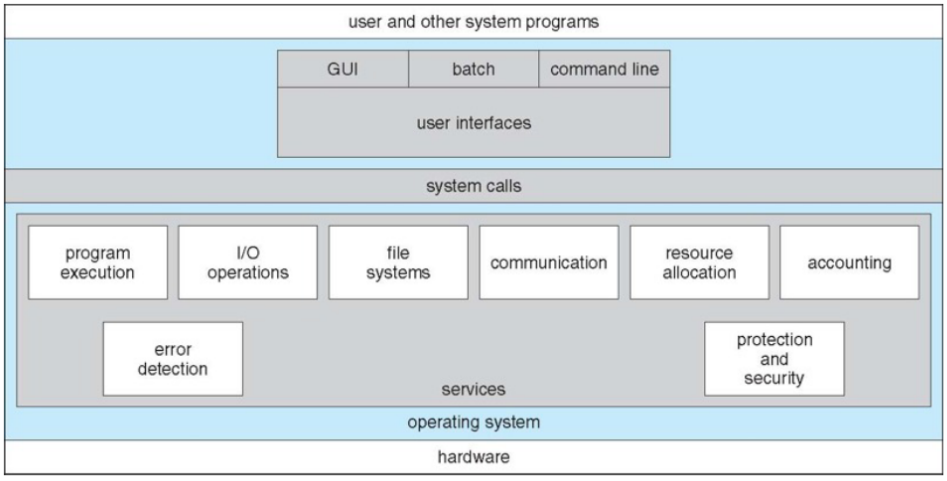
- Wenn mehrere User oder mehrere Jobs gleichzeitig aktiv sind, müssen die Ressourcen unter diesen aufgeteilt werden

Accounting:

- Accounting oder Buchhaltung führ Buch darüber, welche User wieviel und welche Art von Ressourcen verwenden

Protection & Security:

- Protection muss sicherstellen, dass der Zugriff auf System Ressourcen kontrolliert geschieht. Security dagegen muss dafür sorgen, dass der Zugriff von Aussen korrekt erfolgt. Also dass sich Benutzer mit einem Passwort anmelden, zudem schützt es externe Devices wie Netzwerk Adapter vor unerlaubten Zugriffen



User Operating System Interface

Command Line Interface (CLI) oder Command Interpreter:

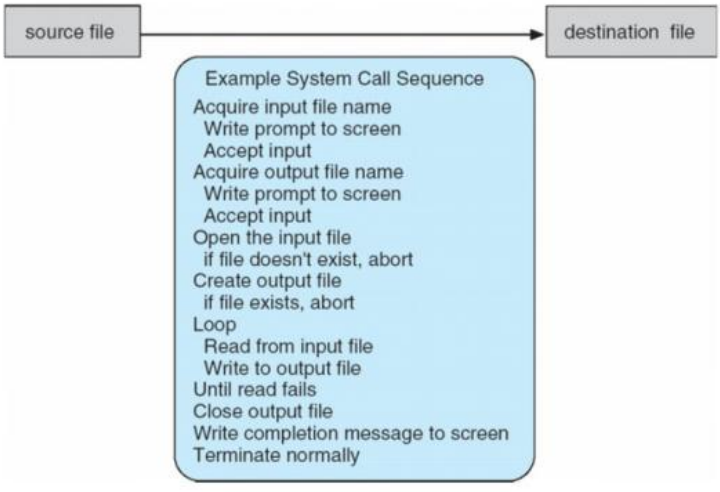
- Erlaubt direkte Kommandoeingabe
- Manchmal im Kernel implementiert
- zT gibt es unterschiedliche Command Interpreter, sogenannte Shells – die sich in der Funktionalität kaum unterscheiden (eher Aufmachung und grafische Erscheinung)

Graphical User Interface:

- Oft bedient durch Maus, Tastatur und Monitor
- Icons repräsentieren Daten, Programme und Aktionen
- Unterschiedliche Maustasten erzeugen unterschiedliche Aktionen im Interface
- Viele OS bieten CLI und GUI zusammen an (Microsoft, OS X, Solaris)

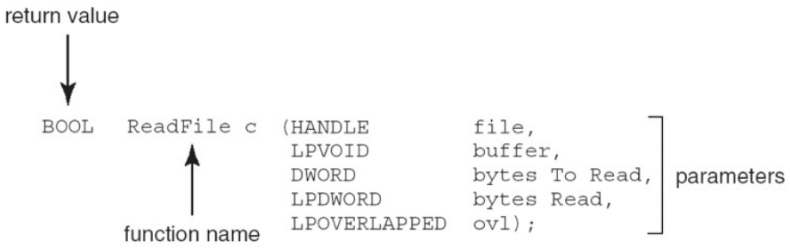
System Calls

- System calls bieten ein Interface (Zugriff) zu den genannten Services, welche das OS bereitstellt.
- Normalerweise in einer high-level System Sprachewie C oder C++ geschrieben
- Programme greifen meistens über ein high-level Application Program Interface (API) auf die Services zu, anstatt direkte System Calls aufzurufen
- Die drei bekanntesten APIs sind Win32 API (Windows), POSIX API (UNIX, Mac OS X) und Java API (JVM)
- Alleine ein simples Programm, dass Daten von einer Datei einliest und diese in eine andere Datei kopiert, muss eine Vielzahl von Calls aufrufen (man bedenkt, dass eine Datei evtl. gar nicht vorhanden ist oder schreibgeschützt ist, oder dass die zu erzeugende Datei bereits vorhanden ist, etc) →



Beispiel für Standard API:

- Consider the ReadFile() function in the
- Win32 API—a function for reading from a file



- A description of the parameters passed to ReadFile()
  - HANDLE file—the file to be read
  - LPVOID buffer—a buffer where the data will be read into and written from
  - DWORD bytesToRead—the number of bytes to be read into the buffer
  - LPDWORD bytesRead—the number of bytes read during the last read
  - LPOVERLAPPED ovl—indicates if overlapped I/O is being used

Arten von System Calls

Process Control

- End, abort
- Load, execute
- Create process, terminate process
- Get process attributes, set process attributes
- Wait for time
- Wait event, signal event
- Allocate and free memory

File management

- Create file, delete file
- Open, close
- Read, write, reposition
- Get file attributes, set file attributes

Device Management

- Request device, release device
- Read, write, reposition
- Get device attributes, set device attributes
- Logically attach oder detach devices

Information maintenance

- Get time or date, set time or date
- Get system data, set system data
- Get process, file or device attributes
- Set process, file or device attributes

Communications

- Create, delete communication connection
- Send, receive messages
- Transfer status information
- Attach or detach remote devices

Protection

Get file security, set file security

Beispiel für Windows und Unix System Calls :

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Programs

Ein weiterer Aspekt eines modernen Systems sind System Programme. Wenn man sich an logische Computer Hierarchie aus Kapitel 1 erinnert, waren da auf dem tiefsten Level die Hardware, gefolgt vom OS und dann den System Programmen. System Programme, auch System Utilities genannt, bieten eine Umgebung für Programmentwicklung und deren Ausführung an. Einige davon sind einfach nur User Interfaces um System Calls darzustellen, andere sind ein wenig komplexer. Sie können in folgende Kategorien aufgeteilt werden:

File Management

- Erzeugen, löschen, kopieren, umbenennen, drucken, auflisten und generell manipulieren Daten und Ordnerstrukturen

Status Information

- Abfragen von Datum, Zeit, verfügbare Menge von freiem Memory, Festplattenspeicher, Anzahl Benutzer

File Modification

- Text Editoren sind verfügbar mit denen man Daten erzeugen und deren Inhalt bearbeiten kann

Programming-Language Support

- Das OS liefert oft Compilers, Assembler, Debugger und Interpreter für gängige Programmierungssprachen mit

Program loading & execution

- Sobald ein Programm assembled oder kompiliert ist, muss es für die Ausführung ins Memory geladen werden

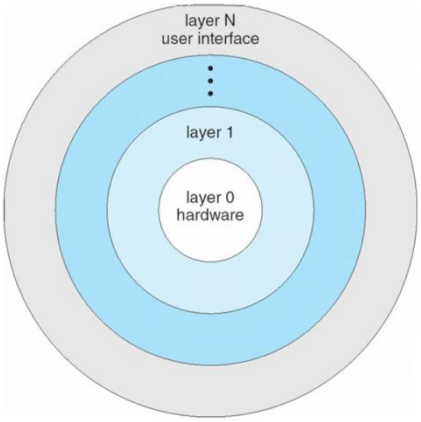
Communication

- Diese Programme bieten Mechanismen zur Erzeugung von virtuellen Connections zwischen Prozessen, Benutzern und Computer Systeme. Sie erlauben das Versenden von Nachrichten zu anderen Bildschirmen, das Besuchen von Web Pages oder das Senden von e-Mails und das transferieren von Daten von einem Computer zu einem anderen

Operating System Design und Implementation

- Bestes Design und Implementation eines OS ist nicht lösbar, aber es gibt einige Ansätze, die sich als erfolgreich bewiesen haben
- Interne Struktur der verschiedenen OS' können sich stark unterscheiden
- Zuerst müssen Ziele und Spezifikationen festgelegt werden – bei OS Entwicklung sehr wichtiger aber auch enorm schwieriger Teil
- Benutzerziele und Systemziele müssen in Einklang gebracht werden:
    - Benutzerziele: OS sollte einfach erlern- und benutzbar sein, stabil, sicher und schnell
    - Systemziele: einfach zu designen und zu implementieren, einfach zu warten sowie flexibel, fehlerfrei und effizient sein

Operating System Struktur



Simple Structure

- Viele OS haben gar keine gut definierten Strukturen, da man zu ihrer Entstehungszeit nicht ahnte, dass diese OS später so bekannt und wichtig wurden. Zbsp MS-DOS

Layered Approach

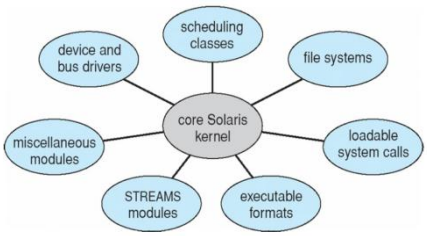
- OS ist geteilt in eine Anzahl von Layern (Levels), jeder aufgebaut auf den darunterliegenden Layern. Der unterste Layer (Layer 0) ist die Hardware, der höchste Layer (Layer N) ist das User Interface

Microkernel System Structure

- Soviel wie möglich vom Kernel in den User Space verschieben
- Kommunikation zwischen User Modulen via message passing
- Vorteile:
    - Microkernel kann einfacher erweitert werden
    - OS kann einfacher auf neue Architekturen portiert werden
    - Stabiler
    - Sicherer
- Nachteile:
    - Perfomanceverbrauch wegen User space und Kernel space Kommunikation

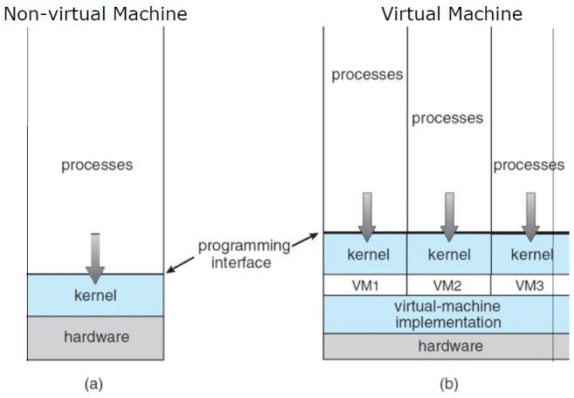
Module

- Die vielleicht momentan beste Methode ist ein object-orientierter Ansatz, wobei man einen modularen Kernel erstellt. Zusätzliche Services werden als Module entweder beim Booten oder während der Laufzeit dazu geschalten, falls notwendig.

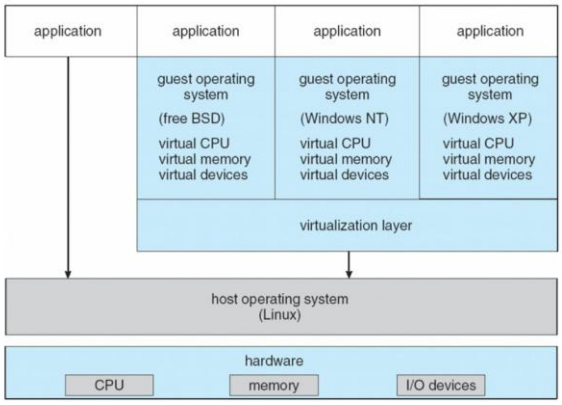


Virtual Machines

Virtuelle Maschinen erzeugen die Illusion, dass ein Prozess seinen eigenen Prozessor hat und sein eigenes (virtuelles) Memory. Die Virtual Machine bietet ein Interface an, das ***IDENTISCH*** zur darunterliegenden Hardware ist, so dass jeder Prozess eine Kopie des darunterliegenden Computers erhält

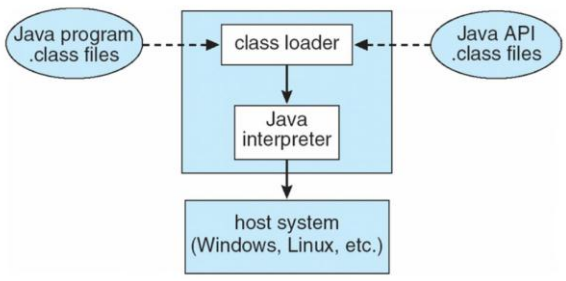


Nebst Virtual Machine Implementationen, die direkt auf der Hardware ausgeführt werden, gibt es auch Lösungen wie ***VMWARE***, das als User Anwendung auf dem OS läuft und von dort aus diese Illusion mehrerer einzelner Computer erzeugt:



Java Virtual Machine

- Compiler erzeugt neutralen Bytecode, der dann von der jeweiligen JVM auf dem entsprechenden System (es gibt für alle gängigen OS ein JVM) ausgeführt werden kann.

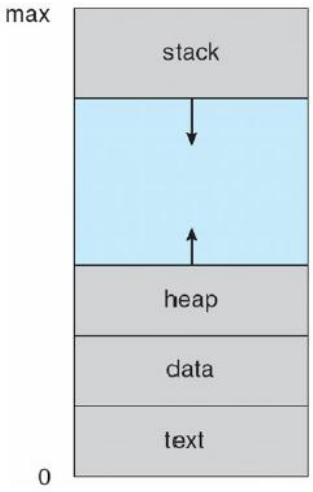


System Boot

- OS muss für Hardware verfügbar gemacht werden, so dass Hardware OS starten kann
- Kleines Stück Code – *BOOTSTRAP LOADER*, im Kernel, ladet sich in Memory und startet sich

Kapitel 3 – Processes

Prozess Konzept

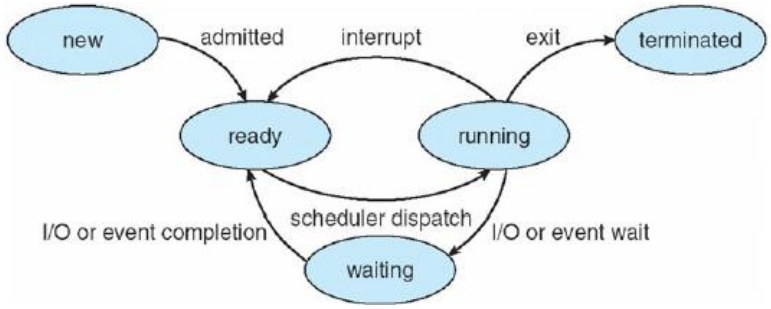


Ein OS führt eine Vielzahl an Programmen aus:

- Batch System – jobs
- Time-shared Systeme – Benutzerprogramme oder Tasks
- Fachbücher verwenden den Ausdruck Job und Prozess gleichbedeutend
- Prozess: ein Programm in Ausführung
- Ein Prozess beinhaltet:
  - Program counter
  - Stack
  - Data section

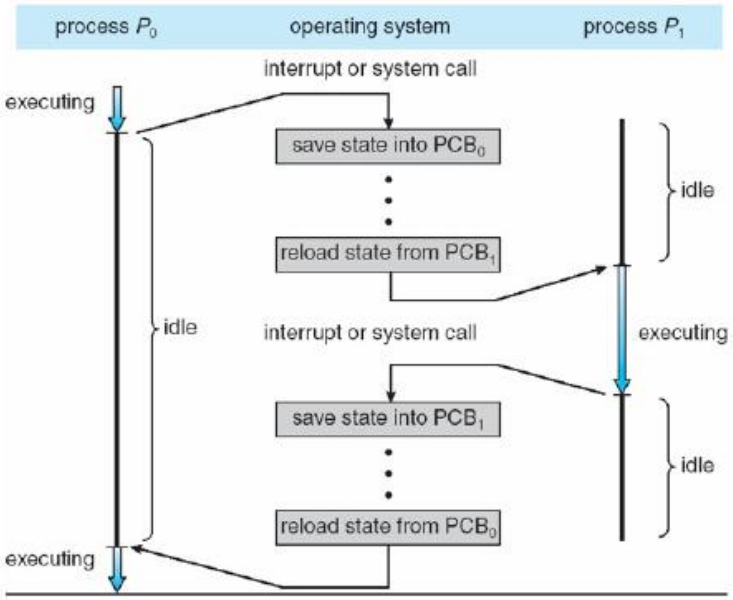
Prozess Zustände

Ein Prozess kann während seiner Ausführung verschiedene Zustände durchlaufen:
New: der Prozess ist erstellt worden
Running: Instruktionen werden ausgeführt
Waiting: Prozess wartet auf ein Ereignis
Ready: Prozess wartet darauf, einem Prozessor zugewiesen zu werden, damit er wieder ausgeführt werden kann
Terminated: Prozess ist mit der Ausführung fertig



Process Control Block

Jeder Prozess wird im OS durch ein Process Control Block (PCB) dargestellt. Dieser beinhaltet jeweils eine Vielzahl an Informationen über den Prozess, wie diese:
Process State: new, ready, running, waiting, halted, etc
Program counter: zeigt die Adresse der nächsten Instruktion an, die ausgeführt warden muss in diesem Prozess
CPU registers: beinhalten index registers, stack pointers, etc – muss zusammen mit dem Program Counter gesichert werden, falls beim Prozess ein Fehler auftritt, damit an dieser Stelle danach weitergefahren werden kann
CPU Scheduling information: beinhaltet Prozessprioritäten und zeigt auf Queues
Memory Management Information: Informationen über Register, Page Tables, etc
Accouting Information: Informationen zu CPU used Time, time limits, job oder process numbers, etc
I/O Status: Informationen über I/O Devices, die dem Prozess zugewiesen sind, offene Files, etc

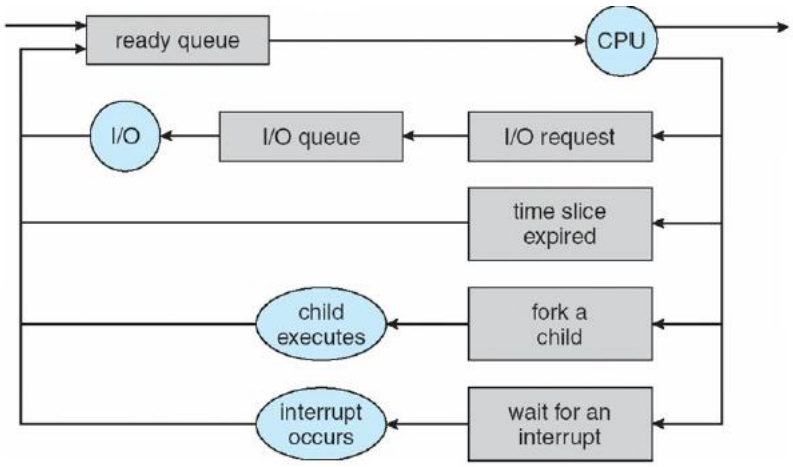


Wenn die CPU zwischen Prozessen switched, muss das System den alten Zustand des Prozesses im PCB abspeichern und den neuen Prozess via *CONTEXT SWITCH* hereinladen – in dieser Zeit macht das System keine nützliche Arbeit

Process Scheduling

Das Ziel von Multiprogramming ist es, dass immer irgendein Prozess ausgeführt wird und die CPU Verwendung maximiert wird. Dazu werden Prozesse so oft geswitched, damit der User mit allen Programmen interagieren kann, während diese ausgeführt werden. Damit dies erreicht wird, muss der *PROCESS SCHEDULER* aus einer Menge von verfügbaren Prozessen einen Prozess auswählen und der CPU zuweisen

Scheduling Queues
Job queue: alle Prozesse im System
Ready queue: alle Prozesse die im Memory bereit sind und auf die Ausführung warten
Device queue: alle Prozesse die für ein I/O Device warten



Schedulers
Long-term Scheduler: entscheidet welcher Prozess in die ready queue gebracht werden soll
Short-term Scheduler: entscheidet welcher Prozess als nächstes der CPU zugewiesen und ausgeführt werden soll

- ➔ Short-term Scheduler wird sehr oft aufgerufen (alle paar Millisekunden), muss demnach sehr schnell sein mit Entscheiden
- ➔ Long-term Scheduler wird unterschiedlich oft aufgerufen (Sekunden, Minuten), kann demnach auch etwas mehr Zeit für eine Entscheidung in Anspruch nehmen

Der Long-term Scheduler muss jedoch seine Entscheidungen sinnvoll treffen. In der Regel gibt es I/O-bound Prozesse und CPU-bound Prozesse – eine optimale Mischung führt zu einer effizienten Auslastung.

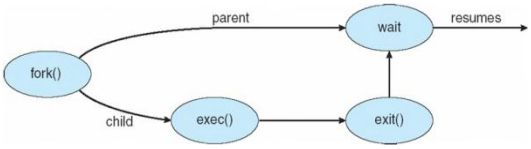
Operations on Processes

OS müssen gewisse Mechanismen für Prozess Erzeugung und Termination anbieten

Process Creation

- Parent erzeugt children Prozesse, welche wiederum andere Prozesse erzeugen können
- Generell werden Prozesse via einem process identifier (pid) identifiziert
- Ressource sharing options: es gibt mehrer Option bezüglich dem Zugriff auf Ressourcen
  - Parent und children teilen sich alle Ressourcen
  - Children teilt sich eine Untermenge der Parent Ressourcen
  - Parent und child teilen sich gar keine Ressourcen
- Es gibt unterschiedliche Ausführungsstrategien:
  - Parents und children werden konkurrierend ausgeführt
  - Parent wartet auf die Termination der children
- Address Spaces:
  - Child dupliziert den Adressraum des Parents
  - Child ladet Programm rein

In Unix wird mit dem Aufruf **fork()** ein child Process erzeugt, welcher nach dem Aufruf die Instruktionen im Code ausführt. Der Return Code für fork() ist Null für einen neuen (child)Prozess. So können also mit if-else Statement zwischen den Prozessen unterschieden werden.
Der Aufruf **exec()** (nach einem fork() Aufruf) erwirkt, dass das Memory des Prozesses mit einem neuen Program ersetzt wird




```c
int main()
{
pid_t pid;

  /* fork a child process */
  pid = fork();

  if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed\n");
    exit(-1);
  }
  else if (pid == 0) { /* child process */
    printf("I am the child %d\n",pid);
    execlp("/bin/ls","ls",NULL);
  }
  else { /* parent process */
    /* parent will wait for the child to complete */
    printf("I am the parent %d\n",pid);
    wait(NULL);

    printf("Child Complete\n");
    exit(0);
  }
}
```

Prozess Termination
Es gibt unterschiedliche Gründe, weshalb ein Process gelöscht werden sollte:

- Prozess führt letztes Statement aus und fragt OS an, es danach zu löschen (**exit**)
  - Output Daten werden vom Child an Parent weitergegeben (via **wait**)
  - Prozess Ressourcen werden vom OS wieder freigegeben
- Parent bricht die Ausführung des children Prozesses ab (abort)
  - Child hat zugewiesene Ressourcen überstrapaziert
  - Dem Child zugewiesene Aufgabe wird nicht mehr benötigt
  - Wenn Parent **exit** ausführt

Interprocess Communication

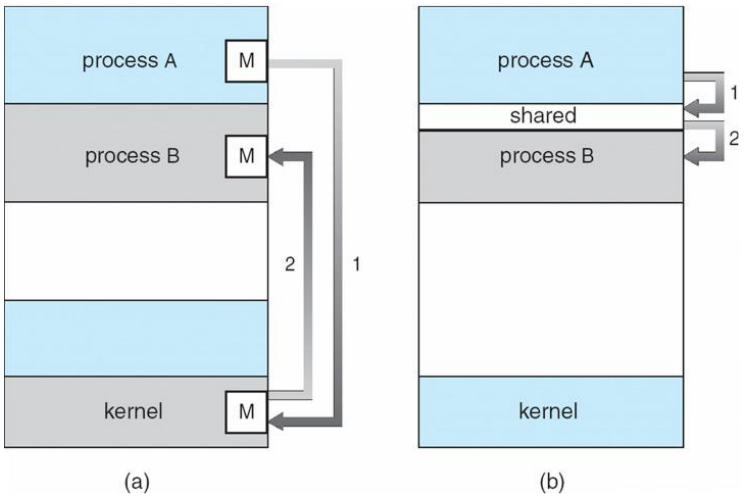
Prozesse können entweder unabhängige oder kooperierende Prozesse sein. Ein unabhängiger Prozess kann von anderen Prozessen nicht beeinflusst werden und auch keine anderen beeinflussen. Jeder Prozess, der keine Daten mit einem anderen Prozess teilt, ist ein unabhängiger Prozess. Ein Prozess kooperiert, wenn er beeinflusst werden oder beeinflussen kann. Logischerweise ist jeder Prozess, der Daten mit anderen Prozessen teilt, ein kooperierender Prozess.

Gründe für Prozess Kooperation:

- Information sharing
- Erhöhung der Geschwindigkeit von Berechnungen
- Modularität

Kooperierende Prozesse benötigen interprocess communication (IPC)
Es gibt zwei Modelle:

- Shared Memory
- Message passing



Producer-Consumer Problem

Paradigma für kooperierende Prozesse: producer process produziert Information die von einem consumer process konsumiert wird

- Unbounded-buffer hat praktisch keine Limite bezüglich der Buffergrösse
- Bounded-buffer nehmen eine fixe Buffergrösse an

```
#define BUFFER_SIZE 10
class item {
        . . .
        };
        item buffer[BUFFER_SIZE];
        int in = 0; // points to empty slot
        int out = 0; // next available item

        while (true) {
        // Produce an item
        while (((in + 1) % BUFFER SIZE count) == out)
        ; // do nothing -- no free buffers
        buffer[in] = item;
        in = (in + 1) % BUFFER SIZE;
        }
        while (true) {
        while (in == out)
        ; // do nothing -- nothing to consume
        // remove an item from the buffer
        item = buffer[out];
        out = (out + 1) % BUFFER SIZE;
        return item;
        }
```

Message Passing

- Mechanismus für Prozesse um miteinander zu kommunizieren und ihre Aktionen zu synchronisieren
- IPC bietet zwei Operationen:
  - Send(**message**) – message Grösse fix oder variabel
  - Receive(**message**)
- Wenn P und Q miteinander kommunizieren wollen, brauchen sie:
  - Einen communication link zwischen ihnen
  - Nachrichten via send/receive austauschen
- Implementation des communication Links:
  - Physikalisch (shared memory, hardware bus, Netzwerk)
  - Logisch (logial properties)

Direkte Kommunikation

- Prozesse müssen sich explizit mit dem Namen aufrufen:
  - Send(P, message) – sendet Nachricht an Prozess P
  - Receive(Q, message) – erhalte eine Nachricht von Prozess Q
- Eigenschaften des communication links:
  - Links werden automatisch erzeugt
  - Ein Link gehört zu genau einem Paar Prozessen
  - Zwischen jedem Paar existiert genau ein Link
  - Ein Link kann unidirektional, meistens aber bi-direktional sein

Indirekte Kommunikation
Kommunikation erfolgt über eine Mailbox

- Nachrichten werden direkt von einer Mailbox empfangen und an diese gesendet
- Jede Mailbox hat eine einzigartige ID
- Prozesse können nur miteinander kommunizieren, wenn sie eine gemeinsame Mailbox haben
- Aktionen sind wie folgt definiert:
  - Send(A, message) – sendet Nachricht an Mailbox A
  - Receive(a, message) – empfängt Nachricht von Mailbox A
- Eigenschaften des communication links:
  - Link wird nur erzeugt, wenn Prozesse eine gemeinsame Mailbox teilen
  - Ein Link kann mehr als nur zwei Prozessen zugewiesen werden
  - Jedes Paar kann auch mehr als nur einen communication link haben
  - Link kann unidirektional aber auch bi-direktional sein
- Mailbox Sharing
  - P1, P2 und P3 teilen sich Mailbox A
  - P1 sendet, P2 und P3 erhalten
  - Wer bekommt die Nachricht?
- Lösung
  - Einem Link erlauben, nur mit höchstens zwei Prozessen zu interagieren
  - Nur einem Prozess zu einer Zeit erlauben, die receive Operation auszuführen
  - Das System soll den Empfänger betimmen – Sender wird informiert, wer der Empfänger ist

Synchronisation

- Message passing kann entweder blockierend oder nicht-blockierend sein (blocking / non-blocking)
- Blocking bedeutet synchron:
  - **Blocking send**: der Sender blockiert bis die Nachricht empfangen worden ist
  - **Blocking receive**: der Empfänger blockiert, bis er eine Nachricht erhält
- Non-Blocking bedeutet asynchron:
  - **Non-blocking send**: der Sender sendet eine Nachricht und macht weiter
  - **Non-blocking receive**: der Empfänger erhält entweder eine Nachricht oder NULL

Buffering
Nachrichten-Queue, welche dem Link zugewiesen ist, kann auf drei Arten implementiert werden:

- Zero capacity: 0 messages, Sender muss auf den Receiver warten (Rendez-vous)
- Bounded capacity: bestimmte Länge n, Sender muss warten, falls Link voll ist
- Unbounded Capacity: unendliche Länge, Sender muss nie warten

Beispiel eines IPC Sytems – POSIX

- Process first creates shared memory segment

```
segment id = shmget(IPC PRIVATE, size, S IRUSR | S IWUSR);
```

- Process wanting access to that shared memory must attach to it

```
shared memory = (char *) shmat(id, NULL, 0);
```

- Now the process could write to the shared memory

```
sprintf(shared memory, "Writing to shared memory");
```

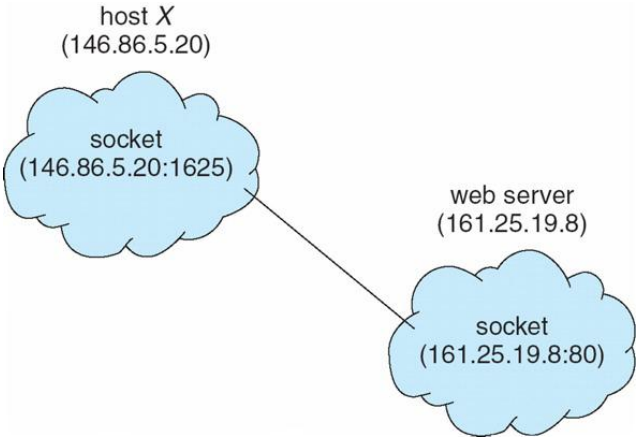
- When done a process can detach the shared memory from its address space

```
shmdt(shared memory);
```

Kommunikation in Client-Server Systeme

Socket

- Ein Socket ist als Endpunkt definiert
- Besteht aus IP Adresse und Port



Pipes

Pipes erlauben zwei Prozessen zu kommunizieren. Wenn Zwei-Weg Kommunikation erlaubt wird, kann die Pipe entweder half duplex sein (Daten fliessen nur in eine Richtung in bestimmter Zeit) oder full duplex (Daten können in einer Zeit in beide Richtungen versendet werden).

Aufbau:

pipe(int fd[])

wobei fd[0] ist das read-end der Pipe und fd[1] ist das write-end.

```
int main (int argc, char * const argv[]) {
    // insert code here...
    char write_msg[BUFFER_SIZE] = "Greetings";
    char read_msg[BUFFER_SIZE];
    int fd[2];
    pid_t pid;

    //create pipe
    if (pipe(fd) == -1) {
        cout << "pipe failed" << endl;
        return 1;
    }

    //fork a child prozess
    pid = fork();

    if (pid < 0 ) {
        cout << "Fork failed" << endl;
        return 1;
    }

    if (pid > 0) {
        //parent
        close(fd[READ_END]);

        //write
        write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

        //close
        close(fd[WRITE_END]);

    }

    else {
        //close
        close(fd[WRITE_END]);

        //read
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        cout << "here it comes: " << read_msg << endl;

        //close
        close(fd[READ_END]);
    }
```

Kapitel 4 – Multithreaded Programming

Thread Grundlagen

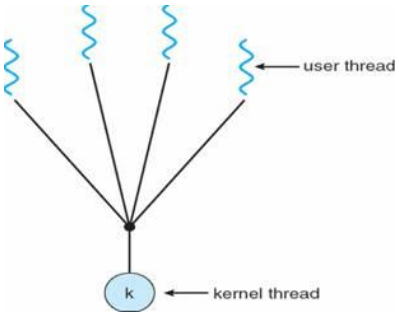
- Threads bieten Softwareportabilität
- Ausbalancierung der Auslastung
- Einfach zu programmieren, weitläufige Verwendung

Schwierigkeiten bei Programmierung für ein Multicore System:

- Aktivitäten in separate Task aufteilen
- Balance – Task sollten gleichwertige Aufgaben erledigen
- Data Splitting – Datenzugriff und Manipulation muss korrekt erfolgen

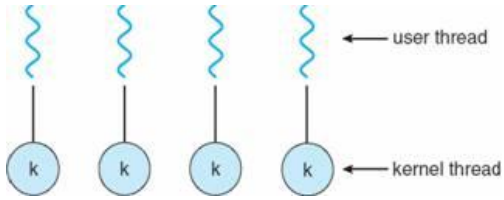
Multithreading Modelle

Many-to-one Model



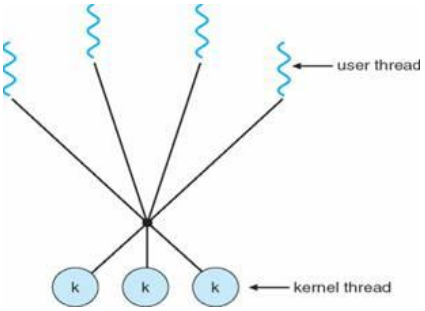
Beim Many-to-one Modell werden viele User-level Threads zu einem einzigen Kernel-Thread zusammengemappt. Dies ist zwar effizient, aber der gesamte Prozess wird blockiert, falls ein Thread einen blockierenden System Call macht.

One-to-One Model



Bei diesem Modell wird jeder User Thread einem Kernel Thread zugewiesen. Das bietet mehr Konkurrenz, da ein blockierender System Call nun nicht mehr den gesamten Prozess lahm legt. Der einzige Nachteil bei diesem Modell ist der Aufwand, welcher betrieben werden muss, damit jeder User Thread einem Kernel Thread zugewiesen wird.

Many-to-Many Model



Dieses Modell verbindet viele User Threads mit einer kleineren oder gleich grossen Anzahl an Kernel Threads.

Pthreads - Creation and Termination

Pthreads: eine POSIX Standard API für die Thread Erzeugung Synchronisation

```
1. #include <pthread.h>
2.
3. int pthread_create (
4. pthread_t *thread_handle, const pthread_attr_t *attribute,
5. void * (*thread_function)(void *), void *arg);
6.
7. int pthread_join ( pthread_t thread,void **ptr);
```

Die Funktion *pthread_create* erzeugt die Funktion *thread_function* als neuen Thread.

Thread Cancellation
Einen Thread beenden, bevor er fertig ist
Zwei Möglichkeiten:

- **Asynchronous cancellation**: terminiert den Zielthread sofort
- **Deferred Cancellation**: erlaubt einem Zielthread periodisch zu überprüfen, ob er beendet werden muss

Signal Handling
Signale werden in UNIX Systeme dazu verwendet, einen Prozess über ein bestimmtes Ereignis zu informieren.
Mögliche Optionen:

- Sende das Signal an jenen Thread, der dafür bestimmt ist
- Sende das Signal an jeden Thread im Prozess
- Sende das Signal an bestimmte Threads im Prozess
- Bestimme einen Thread, der alle Signale für diesen Prozess erhält

Thread Pools
Da das Erzeugen von Threads nicht kostenlos ist, kann man auch bereits beim Starten der Maschine/Applikation eine gewisse Anzahl an Threads vorerzeugen, die dann auf Ihre Verwendung warten – sogenannter Thread Pool.
Vorteile:

- Können einen Request schneller bedienen
- Man kann die maximale Anzahl an Threads bestimmen (durch Poolgrösse)

BEACHTE FOLIE ZU KAPITEL 4 – DORT GEHT ES HAUPTSÄCHLICH UM openMP!

Kapitel 5 – CPU Scheduling

CPU Scheduling bietet die Grundlage für Multiprogramm Betriebssysteme. Indem der Prozessor zwischen Prozessen hin- und hergeswitched wird, wird der Computer produktiver. Es gibt mehrere Wege, wie man das „hin-und herswitchen" umsetzen soll. Diese Algorithmen werden nachfolgend vorgestellt.

Der Grund, weshalb man solche Algorithmen entwickelt hat, ist jener, dass der Prozessor nicht voll ausgelastet ist, wenn man ihm nur einen Prozess zuweist und dieser vielleicht auf einen I/O Input warten muss. In dieser Wartezeit wäre es effizient, wenn ein anderer Prozess bearbeitet wird.

CPU Scheduler

Der CPU Scheduler wählt von einer Menge an Prozessen, die bereit sind und im Memory warten, einen aus und weist diesem dem Prozessor zu.
Der CPU Scheduler muss eine solche Wahl in folgenden 4 Fällen treffen:

1. Ein anderer Prozess wechselt vom „running" Zustand in den „waiting" Zustand (zbsp wegen I/O Request)
2. Ein Prozess wechselt von „running" Zustand in den „ready" Zustand (zbsp wenn ein Interrupt auftritt)
3. Wenn ein Prozess vom „waiting" Zustand in den „ready" Zustand wechselt (sofern preemptive scheduling, wird später erklärt)
4. Terminiert (dann muss ein neuer Prozess vom CPU Scheduler ausgewählt werden=

Die Fälle 1 und 4 sind *NONPREEMPTIVE*
Dies bedeutet, dass die Prozesse ausgeführt werden, bis diese in den „waiting" Zustand gehen oder terminieren (also fertig ist mit seiner Aufgabe) – der Prozess gibt den Prozessor also freiwillig auf.

Die anderen Fälle sind *PREEMPTIVE*
Preemptive kann mit „bevorrechtigt" übersetzt werden. Bedeutet also, dass es Gründe git, dass ein Prozess „preemptive" ist, also ein Vorrecht hat und der aktuelle Prozess vom CPU entfernt wird, damit dieser preemptiver Prozess zum Zuge kommt. Der Fall 3 muss man sich so vorstellen, dass es Scheduling Algorithmen gibt, die sich die Überlegung machen, dass ein Zustand, der von „waiting" zu „ready" wechselt, wichtig sein muss (zbsp weil er auf eine Keyboard Eingabe gewartet hat) und wird dann sofort bearbeitet.

Dispatcher

Dispatcher = Umschalter.
Der Dispatcher schaltet zwischen den einzelnen Prozessen um

Scheduling Kritierien

Es gibt einige Kriterien, mit denen man die Scheduling Algorithmen einteilen kann – einige Algorithmen konzentrieren sich zbsp mehr auf eine kurze Wartezeit zwischen Prozessen und andere haben ein Augenmerk auf einen besonders hohen Durchsatz. Hier die passende Auflistung:

CPU Utilization: Prozessor so gut auslasten wie nur möglich
Troughput (Durchsatz): Anzahl an Prozessen, die pro Zeiteinheit fertig bearbeitet werden
Turnaround time: Zeit, von der ein Prozess zur Berechnung weitergegeben wird, bis er fertig ist inkl. Warten im Memory, in der Ready Queue, Ausführen auf der CPU und I/O Bearbeitung
Waiting time: Zeit, die ein Prozess in der Ready Queue warten muss
Response time (Reaktionszeit): Zeit, die es braucht, bis ein Prozess auf eine Anfrage reagiert (nicht Output)

Scheduling Algorithmen

First Come, First Served (FCFS) Scheduling

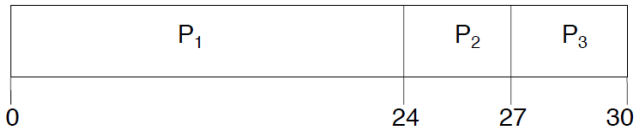
Bei weitem die einfachste Implementierung – die Prozesse, die gerne auf die CPU wollen, müssen in einer Reihe anstehen und werden nach dem first come, first served Prinzip bedient.

Nachteil hierbei ist, dass wenn ein Prozess, der eine lange CPU Burst Zeit hat (CPU Burst: Zeit, in dem der Prozessor für den Prozess am rechnen ist) vielleicht viele andere Prozesse, die nur eine kurze CPU Burst Zeit bräuchten, lange warten müssen. Besser wäre, wenn diese „kleinen" Prozesse vor dem grossen Prozess rasch bearbeitet werden könnten. Nachfolgend eine Verdeutlichung:

Prozess	Burst Zeit
P1	24
P2	3
P3	3

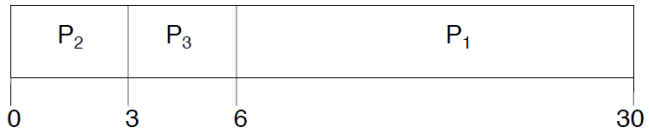
Angenommen, die Prozesse kommen in dieser Reihenfolge an: P1, P2, P3

Dann sieht die Zeitverteilung so aus:



Das ist relativ ungünstig, da die Wartezeiten für P1 = 0, P2 = 24 und P3 = 27 ist. Das ergibt eine durchschnittliche Wartezeit von (0+24+27)/ 3 = 17

Würden die Prozesse in der Reihenfolge P2, P3, P1 eintreffen, sehe es so aus:



Die durchschnittliche Wartezeit würde dann nur noch 3 betragen.

In diesem Zusammenhang kann es auch zum „Conoy Effect" kommen. Das bedeutet, dass kleine Prozesse lange auf das Beenden eines grossen Prozesses warten und wie bei einem Konvoi diesem grossen Prozess „folgen".

Die genannte Problematik und die Missachtung von Prozess-Burst-Längen führt zu einem anderen Scheduling Algorithmus, der sich Shortest-Job-First Scheduling nennt.

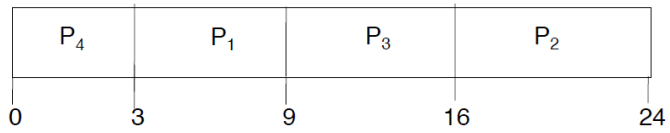
Shortest-Job-First (SJF) Scheduling

Das Optimum bei der SJF Vorgehensweise liegt darin, dass die durchschnittliche Wartezeit minimal ist, weil der kleinste (kürzeste Burst Zeit) Prozess zuerst „drankommt" und die längeren bzw. längsten Prozesse am Schluss behandelt werden.
Das Problem ist jedoch, dass man die Länge der CPU-Bursts wissen muss, damit man die Prozesse korrekt priorisieren kann.

Ein Beispiel zur Verdeutlichung:

Prozess	Burst Zeit
P1	6
P2	8
P3	7
P4	3

Der Idee von SJF folgend, sollte die Prozesse in dieser Reihenfolge (von kurz nach lang) zum Zuge kommen: P4, P1, P3, P2



Dies führt zu folgenden Wartezeiten: 0, 3, 9, 16. Die durchschnittliche Wartezeit ist also (0 + 3 + 9 + 16) / 4 = 7

Man kann die SJF Idee nun noch so erweitern, dass die ganze Sache preemptive wird. Dass also gerade ausführende Prozesse für kürzere Prozesse unterbrochen werden.
Diese Vorgehensweise nennt sich Shortest-Remaining-Time (SRT) Scheduling.

Shortest-Remaining-Time (SRT) Scheduling

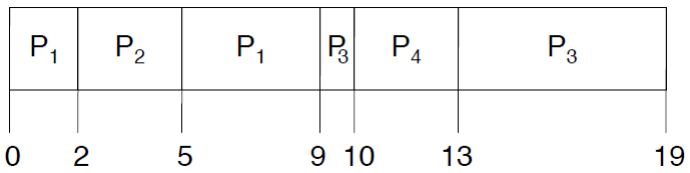
Diese Taktik erlaubt es, dass ein Prozess, der gerade in der Ausführung ist, unterbrochen wird, falls es einen anderen Prozess gibt, der weniger lange für seine Ausführung braucht, als der Prozess, der gerade aktiv ist.

Stellt man sich zbsp einen Prozess vor, der eine CPU Burst Time von insgesamt 8 hat. Nun beginnt die CPU mit der Ausführung dieses Prozesses beim Zeitpunkt 0. Nun kommt nach 1 Zeiteinheit ein neuer Prozess in die Queue, der insgesamt nur 4 Zeiteinheiten benötigt. Da der erste Prozess beim Zeitpunkt 1 noch 7 weitere Einheiten benötigt (braucht ja insgesamt 8), ist er länger als der neue Prozess, der nur 4 für seine komplette Aufgabe braucht. Der „alte" Prozess wird also unterbrochen und es wird der neue Prozess komplett ausgeführt. Danach wird mit dem alten Prozess weitergemacht, falls es keinen weiteren Prozess gibt, der kürzer als diese 7 Einheiten ist.

Wichtig bei dieser Vorgehensweise ist neben der absoluten Burst Zeit, die ein Prozess braucht, auch die Ankunftszeit, wann er in die Queue kommt.

Hier eine Verdeutlichung:

Prozess	Ankunftszeit	Burst Zeit
P1	0	6
P2	2	3
P3	4	7
P4	10	3



Man sieht nun, dass der erste Prozess P1 insgesamt 6 Einheiten braucht, für seine Abfertigung. Da aber nach 2 Zeiteinheiten der Prozess P2 dazukommt, der insgesamt nur 3 Einheiten benötigt, wird P1 unterbrochen, da dieser noch 4 Einheiten bräuchte, was mehr ist, als die komplette Burst Zeit von P2.

Die durchschnittliche Wartezeit ist 2.75. Dies ergibt sich durch (3 + 0 + 5 + 3 + 0) / 4. Man muss beachten, dass die Ankunftszeit bei dieser Berechnung berücksichtigt wird sowie bereits verbrauchte Zeit für einen Prozess abgezogen wird.

P1: 5 – 2 (bereits verbrauchte Zeit für P1 von 0 bis 2) = 3
P2: 2 – 2 (Ankunftszeit) = 0
P3: 9 – 4 (Ankunftszeit) = 5
P3: 13 – 10 (unklar, weshalb…) = 3
P4: 10 – 10 (Ankunftszeit) 0

Berücksichtige das Beispiel im Buch auf Seite 192. Ein wenig verständlicher…

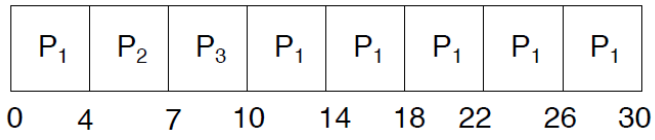
Round Robin

Diese Verfahrensweise möchte sicherstellen, dass jeder Prozess einen gleich grossen Anteil an CPU Zeit erhält. Dieser Anteil ist in der Regel 10-100 Millisekunden gross.
Ein Prozess bekommt also bei jedem Durchgang einen Zugang von 10-100 Millisekunden (je nach Implementation) zum Prozessor. Danach muss sich der Prozess wieder „hinten“ in der Queue anstellen und auf den nächsten Zugang warten.

Wenn also n Prozesse in der ready queue warten und die Zeiteinteilung q ist, dann bekommt jeder Prozess 1/n der CPU Zeit in Einteilungen von q Zeiteinheiten.
Ein Prozess mus also nie länger als (n-1) * q Zeiteinheiten warten (angenommen, es gibt 8 Prozesse und die Zeiteinteilung ist 10ms, dann wird der Prozess nach 70ms wieder „dran“ sein).

Prozess	Burst Zeit
P1	24
P2	3
P3	3

Zeiteinteilung q = 4ms



Priority Scheduling

Hier wird jedem Prozess eine Nummer zugewiesen, die die Priorität ausdrücken soll. Der Prozessor bearbeitet demnach immer jenen Prozess, der die höchste Priorität hat.

Die Zuweisung der Zahlen bzw. deren Bedeutung ist nicht einheitlich abgemacht. Es gibt Implementation, wo eine grosse Zahl eine hohe Priorität ausdrückt und andere Umsetzungen, wo eine kleine Zahl eine hohe Priorität bedeutet.

FCFS ist eigentlich auch eine Priority Scheduling Idee. Die Ankunftszeit ist zugleich die Priorität (frühe Ankunftszeit = höhere Priorität).

Bei diesen Priorisierungsverfahren kann es zu einem Problem kommen, dass sich *STARVATION* nennt. Das bedeutet, dass ein Prozess mit tiefer Priorität gar nie ausgeführt wird, da immer andere Prozesse mit höherer Priorität ihm zuvorkommen. Dieses Problem kann man aber durch *AGING* lösen. Man kann zbsp festlegen, dass alle 15 Minuten lang-wartende Prozesse um eine Prioritätseinheit erhöht werden. Angenommen die Prioritätsskala reicht von 0 bis 127 (0 = hoch), dann dauert es maximal 32 Stunden, bis der 127-prioritisierte Prozess die Priorität 0 hat und ausgeführt wird.

Multilevel Queue

Hier ist die Idee, dass man Prozesse in Kategorien einteilt, welche eigene Queues haben (die nach unterschiedlichen Scheduling Verfahren funktionieren können). Man kann also zbsp Prozesse in die beiden Kategorien

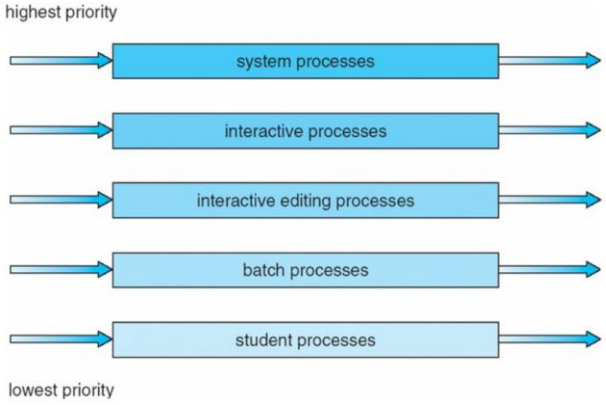
- Foreground
- Background

Einteilen. Diese Kategorien haben unterschiedliche Scheduling Algorithmen:

- Foreground: Round Robin
- Background: FCFS

Zudem kann man zbsp festlegen, dass die Foreground-Queue ein absolutes Vorrecht gegenüber der Background-Queue hat. Sprich: Background-Prozesse werden erst ausgeführt, wenn die Foreground-Queue leer ist.

Man kann selbstverständlich noch weiter aufteilen. Nehmen wir ein Beispiel an, wo 5 Queues angewendet werden:



Jede Queue hat ein absolutes Vorrecht gegenüber tiefer priorisierten Queues. Kein Prozess in der Batch-Queue kann ausgeführt werden, wenn nicht die 3 darüber liegenden Queues leer sind.

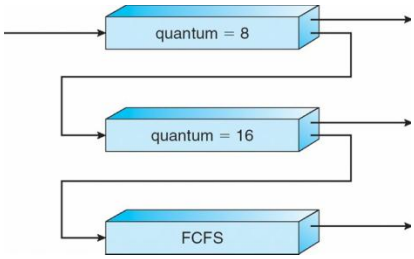
Um eine etwas gerechtere Aufteilung sicherzustellen, kann man die Zeit einteilen, die die Queues erhalten. Zbsp 80% für Foreground Prozesse und 20% für Background Prozesse.

Multilevel Feedback Queue

Beim vorher genannten Multilevel Verfahren können Prozesse nicht zwischen den einzelnen Queues wechseln, da sie fix einer Queue zugewiesen werden. Beim Multilevel Feedback Verfahren können Prozesse zwischen den Warteschlangen wechseln.
Wenn ein Prozess zu lange in einer tief-priorisierten Queue warten muss, kann der Scheduler ihn aufsteigen lassen – dies verhindert Starvation.

Beispiel:
3 Queues:
Q0 – Round Robin mit 8ms
Q1 – Round Robin mit 16ms
Q2 – FCFS

Ein neuer Prozess trifft in Q0 ein und erhält nach dem Round Robin Verfahren 8ms Burst-Zeit. Falls das nicht ausreicht, wird der Prozess in Q1 verschoben, wo er 16ms erhalten wird. Falls auch das nicht ausreicht, kommt er danach in Q2, wo er nach dem FCFS Verfahren bearbeitet wird – beachte jedoch, dass Q2 nur zum Zuge kommt, wenn Q0 und Q1 leer sind.



Multiprozessor Scheduling

Die bisherigen Verfahren basieren auf der Annahme, dass das System nur mit einem Prozessor ausgestattet ist. Moderne Systeme haben aber oft mehrere Prozessoren oder einen Prozessor mit mehreren Kernen, die dem OS als einzelne Prozessoren erscheinen. Scheduling auf solchen System ist komplexer.

Homogeneous: alle Prozessoren sind identisch – es können alle die gleichen Aufgaben erledigen
Asymmetric Multiprocessing: nur ein Prozessor greift auf die Systemdaten zu und verteilt diese dann an die anderen Prozessoren
Symmetric Multiprocessing (SMP): jeder Prozessor hat auf alles Zugriff und entscheidet selber, welcher Prozess als nächstes dran kommt. Entweder haben die Prozessoren eine gemeinsame Queue oder jeder Prozessor hat seine eigene private Queue (Windows XP, Mac OS X, Linux, Solaris)
Prozessor Affinität (processor affinity): die meisten SMP Systeme versuchen, dass ein Prozess immer auf dem gleichen Prozessor ausgeführt wird, da sonst viele Daten aus dem Cache verschoben werden müssen. Ein Prozess hat also eine gewisse Vorliebe (Affinität) für jenen Prozessor, auf dem er schon ausgeführt wurde

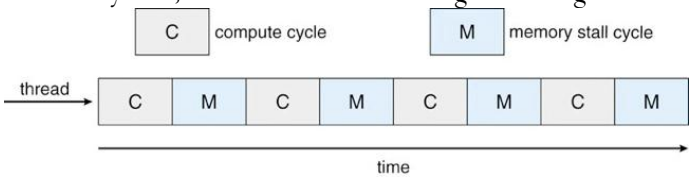
- **Soft affinity**: Das OS versucht, einen Prozess immer auf der gleichen CPU auszuführen, garantiert das aber nicht (Migration zu anderer CPU möglich)
- **Hard affinity**: dem Prozess wird eine Migration nicht mehr ermöglicht – er wird gezwungen, immer auf der gleichen CPU sich auszuführen

Load Balancing
Auf SMP Systemen ist es wichtig, dass die Auslastung gleichmässig verteilt ist und nicht etwa nur ein Prozessor ständig unter Volllast läuft und die anderen im Idle-Modus sind.
Für die Überwachung gibt es zwei Möglichkeiten:
Push Migration: ein spezieller Task überprüft regelmässig die Auslastung der Prozessoren und falls er eine „Ungereimtheit" findet, verschiebt (pushed) er Prozesse von diesem überladenen Prozessor zu einem anderen
Pull Migration: ein freier Prozessor (Idle Mode) zieht (pull) sich einen neuen Prozess von einer anderen Queue.

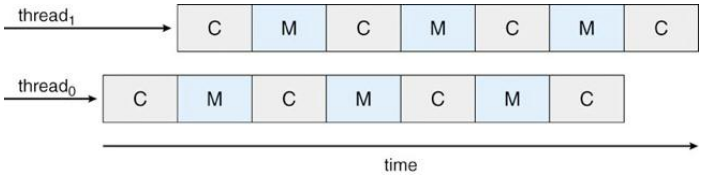
Multicore Prozessor
Wie bereits erwähnt gibt es Prozessoren, die mehrere Kerne verbaut haben, sogenannte Multicore (Multikerne) Prozessoren. Das OS erkennt die einzelnen Kerne als eigenständige Prozessoren. Solche Prozessoren sind schneller als Systeme mit echten mehreren physischen Prozessoren, da die Speicheranbindung schneller ist.

Man muss bei solchen Systemen jedoch den „**Memory Stall**“ Effekt berücksichtigen. Forscher haben festgestellt, dass ein solcher Prozessor eine nicht unerhebliche Zeit auf Memory-Zugriff warten muss (zbsp wenn er auf Daten zugreifen will, die noch gar nicht vorhanden sind). In solchen Fällen kann der Prozessor bis zu 50% der Zeit warten, bis Daten verfügbar werden. Um diese Zeit sinnvoll zu verwenden, hat man eine weitere Technik entwickelt, die es erlaubt, jedem Kern zwei oder mehrere Hardware Threads zuzuweisen. Wenn also ein Thread „stalls“ (wartet), kann der Kern zum anderen Thread wechseln und dort weiter machen.

Hier ein System, das keine Multi-Threading Technologie hat:



Und hier eines, welches die soeben beschriebene Technik verwendet und zwischen zwei Threads wechseln kann:

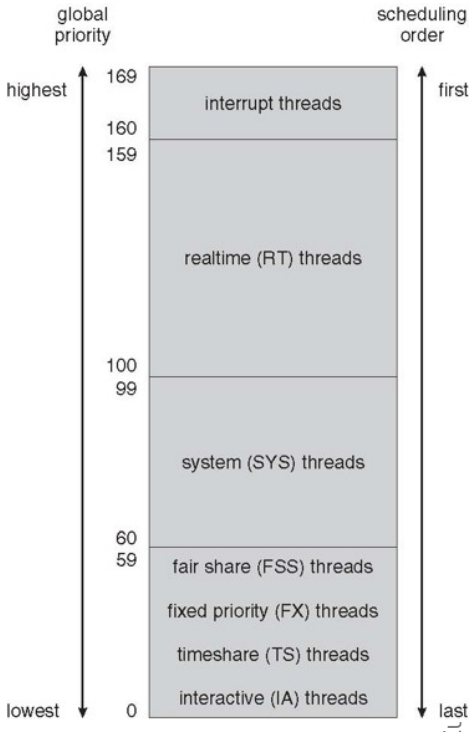


Operating System Beispiele

Solaris
Solaris verwendet ein preemptive priority Scheduling Verfahren mit sechs Prioritätsklassen. Standardklasse für einen Prozess ist „time-sharing“.
Jede Klasse hat ihre eigene Scheduling-Taktik.

Desweiteren kann die Priorität eines Prozesses je nach Situation geändert werden. Nachfolgende Tabelle gibt einen Überblick:

priority	time quantum	time quantum expired	return from sleep
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59



Hierbei muss beachtet werden, dass eine grosse Zahl auch eine hohe Priorität ausdrückt. Ein Prozess mit hoher Priorität hat nur noch eine kleine „time quantum“ (Zeiteinheit, für die Ausführung auf der CPU). Wenn ein

Prozess „returns from sleep“, wird seine Priorität stark erhöht, da Solaris ausgeht, dass zbsp ein Keyboard Input gemacht wurde und dieser Prozess nun besonders wichtig ist.

Windows XP

Windows XP verwendet auch ein priority based preemptive Scheduling Verfahren, wobei der Prozess mit der höchsten Priorität ständig aktiv ist, solange nicht er nicht:

- Von einem noch höheren priorisierten Prozess „preempted“ wird
- Terminiert
- Seine Zeiteinheit aufgebraucht ist (time quantum)
- Oder er einen blockierenden System Call macht und in den „Wait“ Zustand geht

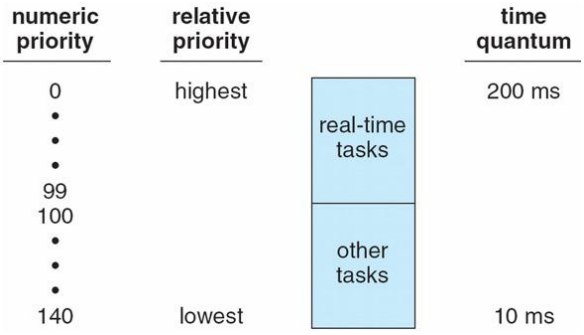
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Es gibt 32 Prioritätslevels. Level 1-15 sind „variabel“ und 16-31 sind für „real-time“ Prozesse. Die einzelnen Prioritätsklassen (real-time, high, above normal, normal, below normal, idle priority) sind innerhalb ihrer Klassen nochmals eingeteilt in time-critical, highest, above normal, normal, below normal, lowest und idle.

Standardmässig ist ein Prozess auf „normal“. Die Priorität wird gesenkt, wenn ein Prozess seine Zeiteinheit verbraucht (kann aber niemals unter die anfänglich zugewiesene Prioritätsklasse sinken). Wenn ein Prozess vom Warte-Zustand zurückkommt, wird seine Priorität „geboosted“.

Linux

Ebenfalls preemtive priority-based Verfahren. Es gibt zwei separate Prioritäts-Bereiche. Einen **Real-Time** Bereich von 0 bis 99 und einen **nice**-value Bereich von 100 bis 140. Tiefe Zahlen geben hohe Prioritäten an. Prozesse mit hoher Priorität haben einen grösseren „time quantum“.



Java

Verwendet eine preemptive, priority-based Verfahren. FIFO Queue, wenn es mehrere Threads mit derselben Priorität gibt.

Kapitel 6 – Prozess Synchronisation

Problematik: es kann zu Problemen kommen, wenn kooperierende Prozesse gemeinsame Daten bearbeiten bzw. diese überschreiben. Zur Verdeutlichung nehmen wir das Consumer-Producer Problem aus Kapitel 3 an und legen zudem fest, dass die Variabel „count“ anzeigen soll, wieviele Buffer schon voll sind. Wenn der Producer einen neuen Buffer erzeugt, steigt count um 1 und wenn der Consumer einen Buffer konsumiert, senkt dieser count um 1.

Hier weitere Angaben zu diesem Problem:

- count++ could be implemented as

```
register1 = count
register1 = register1 + 1
count = register1
```

- count-- could be implemented as

```
register2 = count
register2 = register2 - 1
count = register2
```

- Consider this execution interleaving with “count = 5” initially:
  - S0: producer execute register1 = count {register1 = 5}
    S1: producer execute register1 = register1 + 1 {register1 = 6}
    S2: consumer execute register2 = count {register2 = 5}
    S3: consumer execute register2 = register2 - 1 {register2 = 4}
    S4: producer execute count = register1 {count = 6 }
    S5: consumer execute count = register2 {count = 4}
- Major problem!

Wie man sieht, hat count schlussendlich den Wert 4, was falsch ist. Der Wert sollte bei 5 sein, da der Producer count (als Startwert wird im Beispiel 5 genannt) um 1 auf 6 erhöht und der Producer danach einen Buffer konsumiert und den count auf den Wert 5 wieder zurücksetzen sollte.
In diesem Beispiel hängt es also stark davon ab, in welcher Reihenfolge die Prozesse auf count zugreifen. Diese Problematik nennt sich ***RACE CONDITION***.

Critical-Section Problem

Nehme ein System mit n Prozessen an. Jeder Prozess hat ein Segment an Code, ***CRITICAL SECTION*** genannt, in welchem der Prozess Variablen, Arrays, Files etc updatet/bearbeitet. Das System hat eine wichtige Funktion, nämlich, dass wenn ein Prozess in seiner Critical Section ist, kein anderer Prozess auf seine jeweils eigene Critial Section zugreifen darf. So können niemals zwei Prozesse gleichzeitig auf ihre Critical Sections zugreifen. Damit die ganze Sache auch geordnet abläuft, muss ein Prozess zuerst eine Anfrage für den Zugriff in seine Critical Section starten, entry section genannt.

```
while (TRUE) {

  entry section
    critical section    // prone to race conditions
  exit section

  remainder section
};
```

“anfällig für Race Condition“

Eine Lösung für dieses Critical-Section Problem muss folgende drei Kriterien erfüllen:

1. **Mutual exclusion**: wenn ein Prozess in seiner Critical Section ist, darf kein anderer Prozess in seiner eigenen Critical Section sein
2. **Progress**: wenn kein Prozess in seiner Critical Section ist und es Prozesse gibt, die gerne auf ihre Critical Section zugreifen möchten, dann darf die Entscheidung, welcher Prozess als nächstes auf seine Critical Section zugreifen darf, nicht unbestimmt lange verschoben werden
3. **Bounded waiting**: eine Limite legt fest, wie oft andere Prozesse (noch) auf ihre Critical Section zugreifen dürfen, wenn ein anderer Prozess bereits eine Anfrage für den eigenen Zugriff gemacht hat

Synchroniziation Hardware

Viele Systeme bieten Hardware Unterstützung für Critical Section Code in dem Prozessor Interrupts „missachtet" bzw. deaktiviert. Heisst: gerade ausgeführter Code wird ohne Preemption ausgeführt.
Auf Multiprozessor Systemen ist das aber zu ineffizient. Daher bieten moderne Systeme spezielle „atomic" Hardware Instruktionen (atomic = nicht unterbrechbar) an. Diese kann mit einem simplen Tool „Lock" angeboten werden. Ein Prozess muss bevor er seine Critical Section betreten darf, zuerst einen Lock anfordern und wenn er fertig ist mit seiner Critical Section diesen Lock (Schloss) wieder aufheben.

```
while (TRUE) {

  acquire lock
    critical section
  release lock

  remainder section
};
```

TestAndSet Instruktion

TestAndSet Function gibt an (wenn auf mehreren CPUs simultan ausgeführt wird), ob gerade eine Critical Section gelocked ist.

```
boolean TestAndSet (boolean *locked)  // modifiable parameter
{
  boolean answer = *locked;
  *locked = TRUE;

  return answer;
}
```

Gibt als Antwort den Wert von Locked zurück, also ob es locked ist oder nicht.

Der Wert „locked" wird mit allen Prozessen geteilt, so dass diese wissen, ob irgendwo ein Lock aktiv ist:

```
while (TRUE) {
  while (TestAndSet(&locked))
    ;   // do nothing, busy wait

  //    critical section

  locked = FALSE;

  //      remainder section
};
```

Wie man sieht, wird nichts gemacht, solange „while (TestAndSet(&locked)" wahr ist. Nur wenn kein Lock aktiv ist, wird die Critical Section betreten. Nach dem der Prozess damit fertig ist, setzt er locked auf False, damit ein anderer Prozess, der seinerseits überprüft, ob irgendwo ein Lock aktiv ist, grünes Licht bekommt.

Swap Instruction

```
void Swap (boolean *lock, boolean *key)// modifiable parameters
{
  boolean temp = *lock;
  *lock = *key;
  *key = temp:
}
```

Tauscht die Werte von Key und Lock gegeneinander aus.

```
while (TRUE) {
  key = TRUE;
  while (key == TRUE)
    Swap(&lock, &key);  // busy wait

  //    critical section

  lock = FALSE;

  //      remainder section
};
```

Lock ist nach dem Swap Vorgang auf true und somit kann kein anderer Prozess zugreifen.

```
while (TRUE) {
  waiting[i] = TRUE;          // process i waiting for access to CS
  key = TRUE;
  while (waiting[i] && key)  // wait for lock from other process
    key = TestAndSet(&lock); // busy wait
  waiting[i] = FALSE;

  // critical section

  j = (i+1) % n;
  while ((j != i) && !waiting[j])  // check for waiting processes
    j = (j+1) % n;
  if (j == i)
    lock = FALSE;           // no one waiting, release lock
  else
    waiting[j] = FALSE;     // transfer lock

  // remainder section
};
```

UNKLAR WAS HIER GENAU GESCHIEHT – Siehe Buch Seite 232, 233 und 234

Semaphores

Eine Semaphore S ist eine integer Variable, die nur durch die atomaren (nicht unterbrechbaren) Funktionen wait() und signal() beeinflusst werden kann:

```
wait (S) {
  while S <= 0
    ; // no-op
  S--;
}

signal (S) {
  S++;
}
```

Binary Semaphore:

- integer Wert kann nur 0 oder 1 sein
- auch bekannt als „Mutex Lock“

kann verwendet für das Critical Section Problem verwendet werden:

```
Semaphore mutex;     //  initialized to 1
while (TRUE) {
  wait (mutex);
  // critical section
  signal (mutex);
  // remainder section
};
```

Counting Semaphore:

- Wert der Integer Variable ist nicht beschränkt

Kann verwendet werden, um den Zugang zu einer Ressource einzuschränken. Die Semaphore wird dabei mit dem Wert der vorhandenen Ressourcen initialisiert. Angenommen es gibt 3 CD-Laufwerke, dann wird die Semaphore mit 3 initialisiert und jeder Prozess der diese Ressource verwenden will, führt eine wait() Operation aus, so dass die Semaphore um 1 sinkt. Sobald der Prozess fertig ist mit dieser Ressource signalisiert er dies mit einer signal() Operation und die Semaphore erhöht sich um 1.

Busy Waiting

Busy Waiting ist ein Problem in multiprogramming Systemen, denn solange ein Prozess in seiner Critical Section ist, müssen alle anderen Prozesse in ihrer Entry Section in einem Loop warten und machen in dieser Zeit nichts Sinnvolles (ausser ständig zu checken, ob der Wert jetzt endlich passt).

Man kann daher die wait() und signal() Funktion anpassen: wenn ein Prozess die wait() Operation ausführt und dann feststellt, dass der Semaphore Wert negativ ist, wird er warten. Aber stattdessen dass er busy waiting (einfach im Loop der Entry Section hängt und ständig überprüft, ob der Wert nun positiv ist) macht, kann er sich selber **blocken**. Die Block-Operation legt den Prozess in eine Queue, die zu dieser Semaphore gehört, ab und der Zustand des Prozesses geht in den „waiting“ Zustand über.
Sobald ein anderer Prozess eine signal() Operation ausführt und somit eine Ressource freigibt, sollte der wartende Prozess geweckt werden mit einer wakeup() Operation, so dass er wieder in den „ready“ Zustand kommt.

- Implementation of wait:

```
wait(semaphore *S) {
  S->value--;
  if (S->value < 0) {
    add this process to S->list;
    block();
  }
}
```

- Implementation of signal:

```
signal(semaphore *S) {
  S->value++;
  if (S->value <= 0) {
    remove a process P from S->list;
    wakeup(P);
  }
}
```

Deadlock and Starvation

Deadlock
Zwei oder mehr Prozesse warten unbestimmt lange für ein Ereignis, dass nur von einem dieser wartenden Prozesse hervorgerufen werden kann.

P_0	P_1
wait (S);	wait (Q);
wait (Q);	wait (S);
.	.
.	.
.	.
signal (S);	signal (Q);
signal (Q);	signal (S);

P0 ruft wait(S) auf und P1 ruft wait(Q) auf. Wenn P0 nun wait(Q) aufrufen will, muss er warten bis P1 signal(Q) aufruft, da diese sginal-Funktionen nicht aufgerufen werden können (unklar, wieso das nicht geht, aber diese Behauptung ist grundlegend, für die ganze Aussage hier…), sind die Prozesse deadlocked.

Starvation
Unbestimmtes langed blocking. Ein Prozess könnte nie mehr aus der Semaphore Queue entfernt werden

Priority Inversion
Ein Scheduling Problem, nämlich wenn ein tief-priorisierter Prozess einen Lock auf eine Ressource hält, die von einem höher-priorisierten Prozess benötigt wird

Klassische Probleme der Synchronisation

Bounded-Buffer Problem

- N buffers, each can hold one item
  - Semaphore `mutex` initialized to the value 1
    - Used to exclude others from modifying the buffer state concurrently
  - Semaphore `full` initialized to the value 0
    - Used to indicate available items in buffer
  - Semaphore `empty` initialized to the value N
    - Used to indicate available storage slots
- The structure of the producer process

```
while (TRUE)  {
  //   produce an item in nextp

  wait(empty);
  wait(mutex);

  //  add the item to the  buffer

  signal(mutex);
  signal(full);
};
```

- The structure of the consumer process

```
while (TRUE) {
  wait(full);
  wait(mutex);

  //  remove an item from buffer

  signal(mutex);
  signal(empty);

  //  consume the item
};
```

Readers-Writers Problem

Angenommen es gibt einen Datensetz das unter einer Anzahl Prozessen geteilt wird. Einige dieser Prozesse wollen den Datensatz nur lesen und andere wollen den Datensatz lesen und beschreiben. Wenn zwei Reader-Prozesse auf den Datensatz gleichzeitig zugreifen, sollte logischerweise nichts passieren. Wenn aber ein Writer-Prozess zu dieser Zeit ebenfalls zugreift, kann es zu Problemen kommen. Um dieses Problem zu verhindern, kann man voraussetzen, dass ein Writer-Prozess exklusiven Zugang bekommt und keine anderen Prozesse in dieser Zeit auf den Datensatz zugreifen dürfen.

- Shared Data
  - Data set
  - Semaphore `mutex` initialized to 1
    - Used to exclude others from modifying the buffer state concurrently
  - Semaphore `wrt` initialized to 1
    - Used to serialize exclusive write operations
  - Integer `readcount` initialized to 0
    - Used to indicate concurrent readers
- The structure of a writer process

```
while (TRUE) {
  wait(wrt);

  //    writing is performed

  signal(wrt);
};
```

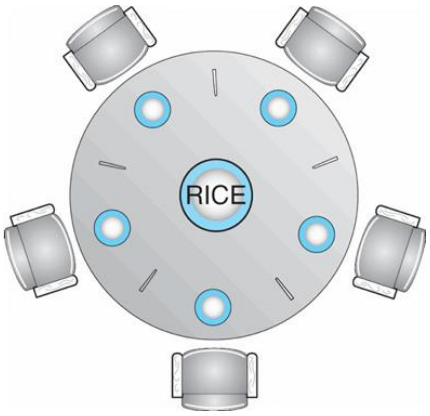
- The structure of a reader process

```
while (TRUE) {
   wait(mutex);          // operate safely on read count
   readcount++;
   if (readcount == 1)
     wait(wrt);          // prevent any writers at this time
   signal(mutex)

   // reading is performed

   wait(mutex);
   readcount--;
   if (readcount == 0)
     signal(wrt);        // make sure writers can update data
   signal(mutex);
};
```

Dining-Philosophers Problem



Man muss sich hier 5 Philosophen vorstellen, die gemeinsam an einem runden Tisch sitzen, in dessen Mitte es eine Schüssel Reis hat und neben jedem Teller hat es ein Essstäbchen (also 5 insgesamt, beachte das Bild). Sie können zwei Aktionen ausführen: Denken und Essen. Wenn Sie am Denken sind, interagieren Sie mit niemandem. Wenn Sie Essen wollen, so müssen sie die zwei Essstäbchen nehmen, die am nächsten sind. Also jene zwei zwischen ihm und dem Philosophen links und rechts von ihm. Solange dieser Philosoph am Essen ist, legt er die Stäbchen nicht ab. Mit Semaphoren kann man dieses Problem ebenfalls bis zu einem gewissen Grad lösen:

- The structure of Philosopher ***i***:

```
while (TRUE)  {
  wait(chopstick[i] );              // get left chopstick
  wait(chopStick[ (i + 1) % 5] );   // get right chopstick

  //  eat

  signal(chopstick[i] );
  signal(chopstick[ (i + 1) % 5] );

  //  think
};
```

-

Monitors

Monitors bietet einen Mechanismus für eine effektive Prozess Synchronisation an, in dem immer nur ein Prozess pro Zeit im Monitor aktiv sein darf und Funktionen aufrufen kann.

Man kann sich diesen Mechanismus anhand des Dining Philosophers Problem anschauen, wobei *DiningPhilosophers* der Monitor ist. Jeder Philosoph *i* muss die Operationen pickup() und putdown() aufrufen in dieser Reihenfolge:

DiningPhilosophers.pickup(i);

// EAT

DiningPhilosophers.putdown(i);

```
monitor DiningPhilosophters
{
  enum {THINKING; HUNGRY, EATING} state[5];
  condition self[5];

  void pickup(int i) {
    state[i] = HUNGRY;
    test(i);
    if (state[i] != EATING)
      self[i].wait;
  }

  void putdown(int i) {
    state[i] = THINKING;
    // test left and right neighbors
    test((i+4) % 5);
    test((i+1) % 5);
  }
```

```
  void test(int i) {
    if ((state[i] == HUNGRY) &&
        (state[(i+4)%5] != EATING) && (state[(i+1)%5] != EATING)){
      state[i] = EATING;
      self[i].signal();
    }
  }

  initialization_code() {
    for (int i = 0; i < 5; i++)
      state[i] = THINKING;
  }
}
```

Wenn also ein Philosoph hungrig ist und die Stäbchen aufheben will, ruft er pickup(), wobei dieses Funktion via test() überprüft, ob die beiden Nachbarn des hungrigen Philosophen nicht gerade am Essen sind und somit die Stäbchen in Benutzung wären. Dieser Monitor-Mechanismus **verhindert** zudem einen **Deadlock**, welcher passieren könnte, wenn alle Philosophen gleichzeitig ein Stäbchen aufheben und erst dann das zweite (welches ja dann nicht mehr verfügbar ist, da alles gleichzeitig eins genommen haben) aufheben möchten und unendliche lange warten.

A Monitor to Allocate Single Resource

Man stellt sich einen Monitor vor, der eine Resource (zbsp CD Laufwerk) managed. Wenn diese Ressource gerade in Verwendung ist, müssen alle anderen Prozesse warten. Die Frage ist nun, welche der wartenden Prozesse als nächstes den Zugriff erhalten soll. Die FCFS Methode wäre eine Möglichkeit, aus Effizienzgründen sollte man aber die SJF (shortest job first) Methode verwenden. Dies kann man so umsetzen, dass wenn ein Prozess wait() ausführt, diesem wait() einen Integer übergibt, der die Zeit repräsentiert, die er für diese Ressource braucht: wait(time).

```
monitor ResourceAllocator
{
  boolean busy;
  condition x;

  void acquire(int time) {
    if (busy)
      x.wait(time);
    busy = TRUE;
  }

  void release() {
    busy = FALSE;
    x.signal();
  }

  initialization code() {
    busy = FALSE;
  }
}
```

Synchronisations-Beispiele

Solaris

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses adaptive mutexes for efficiency when protecting data from short code segments
- Uses condition variables and readers-writers locks when longer sections of code need access to data
- Uses turnstiles to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock

Turnstile ist eine Queue für Prozesse, die aufgrund eines Lockes am Warten sind. Sobald der Lock entfernt wird, entscheidet der Kernel, welcher Prozess aus dem Turnstile als nächstes dran ist.

Windows XP

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

Linux

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - semaphores
  - spin locks

Pthread

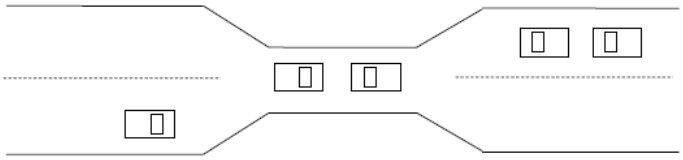
- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spin locks

Kapitel 7 – Deadlocks

Das Problem:
Eine Gruppe von Prozessen haben gleichzeitig Zugriff auf je eine Ressource und warten, um ein Ressource eines der anderen Prozesse zu erhalten (und umgekehrt).

Beispiel: System hat 2 Festplatten
P1 und P2 haben je exklusiven Zugriff auf eine der Festplatten, brauchen aber nun Zugriff auf die jeweils andere
→ Deadlock

Bridge Crossing Example



- Traffic only in one direction at any time
- Each section of a bridge can be viewed as a resource
- If a deadlock occurs, it can be resolved if one car backs up
  - preempt resources and rollback
- Several cars may have to be backed up if a deadlock occurs
- Starvation is possible
- **Note** – Most OSes do not prevent or deal with deadlocks

Deadlock Kriterien

Ein Deadlock kann auftreten, wenn folgende 4 Konditionen erfüllt sind:
Mutual Exclusion: eine Ressource kann nur von einem Prozess gleichzeitig verwendet werden
Hold and Wait: es gibt einen Prozess, der bereits eine Ressource verwendet und auf eine andere zugreifen will, die bereits verwendet wird
No Preemption: eine Ressource kann nur freiwillig vom Prozess freigegeben werden, wenn er fertig ist. Niemand kann ihn zwingen
Circular wait: es gibt eine Menge an wartenden Prozessen (P0, P1, …, Pn), die jeweils auf die Ressource des nächsten Prozesses warten: P0 wartet auf Ressource, die von P1 verwendet wird, P1 wartet auf Ressource die von P2 verwendet wird und Pn wartet auf die Ressource von P0.

Resource Allocation Graph

Deadlocks können mit Graphen präziser beschrieben werden. Die Graphen bestehen aus einem Set Knoten V und einen Set Kanten E.

Die Knoten werden in zwei Gruppen geteilt:
P (P1, P2, …, Pn) = alle Prozesse im System
R (R1, R2, …, Rn) = alle Ressourcen im System.

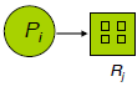
Request Edge (Anfrage Kante) = gerichtete Kante Pi ⟶ Rj
Assignment Edge (Zuweisungs Kante) = gerichtete Kante Rj ⟶ Pi

Pi wird als Kreis dargestellt und Rj als Rechteck:

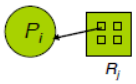
- Process

- Resource Type with 4 instances

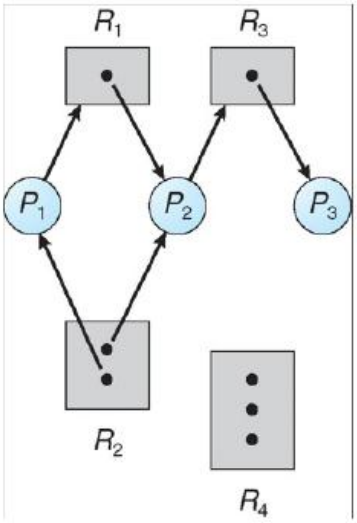
- P_i requests instance of R_j



- P_i is holding an instance of R_j



Beispiel eines Ressource Allocation Graphs:



Die Sets P, R und E:

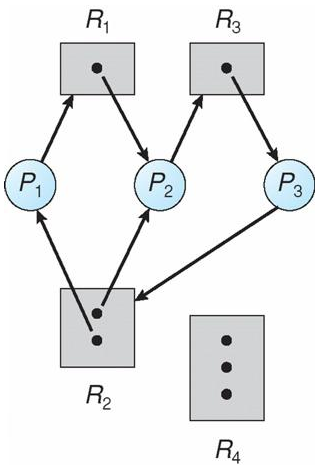
P = {P1, P2, P3}
R = {R1, R2, R3, R4}
E = { P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3}

Ressource Instanzen:

- Eine Instanz der Ressource R1
- Zwei Instanzen der Ressource R2
- Eine Instanz der Ressource R3
- Drei Instanzen der Ressource R4

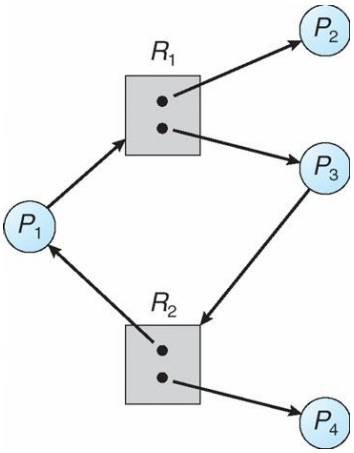
Zustände:
Prozess P1 hält eine Instanz von R2 und wartet auf den Zugriff für R1.
Prozess P2 hält eine Instanz von R2 und R1 und möchte auf R3 zugreifen.
Prozess P3 hält eine Instanz von R3.

Wenn der Graph keinen Kreislauf (Cycle) aufweist, dann ist kein Prozess deadlocked.
Hier ein Beispiel, wo der Graph einen Kreislauf aufweist:



Man sieht, dass
P1 → R1 → P2 → R3 → P3 → R2 → P1
und
P2 → R3 → P3 → R2 → P2
je einen Kreislauf (siehe die gerichteten Pfeile verlaufen im Uhrzeigersinn) bilden, wenn nun jede der involvierten Ressource **nur eine Instanz** hat (also nur eine Festplatte, oder nur 1 CD Laufwerk), dann gibt es hier einen **Deadlock**. Falls es mehrere Instanzen gibt, ist ein Deadlock nicht garantiert.

Hier noch ein Beispiel, wo es auch einen Cycle gibt, aber kein Deadlock auftreten wird, nämlich weil P4 irgendwann die Ressource R2 freigibt und diese dann P3 zugewiesen werden kann.



Zusammenfassend:

Wenn ein Graph keine Cycles aufzeigt → kein Deadlock
Wenn ein Graph einen Cycle aufzeigt →

- wenn es nur eine Instanz pro Ressource gibt, dann Deadlock
- wenn es mehrere Instanzen gibt, ist ein Deadlock nicht garantiert

Methoden um mit Deadlock umzugehen

- Sicherstellen, dass ein System niemals in einen Deadlock Zustand gelangt (zbsp durch ein Protokoll)
- Dem System erlauben, in einen Deadlock Zustand zu gelangen, es zu erkennen und dann zu beheben
- Das Problem ignorieren und so tun, als ob Deadlocks niemals auftreten werden, wird von den meisten OS so umgesetzt (war ja klar… :P)

Deadlock Prävention

Wie bereits erwähnt, müssen 4 Kriterien erfüllt werden, dass ein Deadlock auftreten kann. Man kann nun versuchen, dass eine dieser Kriterien gar nie erfüllt werden kann.

Mutual Exclusion
Diese Kondition muss für nicht teilbare Ressourcen sichergestellt werden. Bei teilbaren Ressourcen wäre die Aufhebung dieser Kondition natürlich kein Problem. Trotzdem, die Kondition „Mutual Exclusion" zu missachten, ist keine gute Idee.

Hold and Wait
Wenn man sicherstellen will, dass der „Hold and Wait" Zustand niemals auftritt, könnte man mithilfe eines Protokolls dafür sorgen, dass ein Prozess keine andere Ressource halten darf, wenn er Zugriff für eine Ressource anfragt. Oder eine andere Idee wäre, dass ein Protokoll dafür sorgt, dass alle notwendigen Ressourcen bereits zu Beginn zugewiesen werden und nicht erst während der Ausführung auf eine zusätzliche Ressource zugegriffen werden will. Somit kann man sich, sobald die Ausführung startet, sicher sein, dass alle notwendigen Ressourcen verfügbar sind. Problematisch ist hier die schlechte Auslastung (zbsp wenn ein Drucker gebraucht wird, der erst am Schluss der Ausführung verwendet wird und daher über die ganze Zeit hinweg zwar zugewiesen aber gar nicht gross verwendet wird), zudem kann es zu Starvation führen.

No Preemption
Die dritte Regel „no Preemption" besagt, dass eine Ressource, die bereits zugewiesen wurde, nicht mehr mit „Gewalt" entfernt werden kann. Man muss also warten, bis der Prozess die Ressource freiwillig aufgibt. Man kann nun ein Protokoll einführen, damit diese Regel aufgehoben wird. Das Protokoll funktioniert so, dass wenn ein Prozess (der bereits Ressourcen in seiner „Gewalt" halt) auf eine andere Ressource zugreifen will, welche aber schon besetzt ist und er deshalb warten muss, all seine Ressourcen freigeben muss. Diese freigewordenen Ressourcen werden auf eine Liste gesetzt, welche festhält, auf welche Ressourcen dieser Prozess wartet. Der Prozess wird erst fortfahren, wenn er alle alten und die neue gewünschte Ressource erhalten wird.

Circular Wait
Wir teilen jeder Ressource eine feste Zahl zu und setzen diese somit in eine totale Ordnung. Jeder Prozess darf nun nur in aufsteigender Ordnung auf die Ressourcen zugreifen. Beispiel: CD Laufwerk = 1, Festplatte = 5,

Drucker = 12. Wenn jetzt ein Prozess die Festplatte und den Drucker braucht, muss er zuerst die Festplatte sich zuweisen lassen und erst danach darf er den Drucker anfordern.

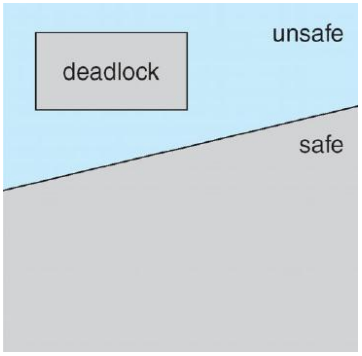
Deadlock Vermeidung (Avoidance)

Man könnte versuchen, bereits im Voraus zu entscheiden, wie Ressourcen zugeteilt werden, indem das System bereits von Beginn an genau weiss, wie und wann ein Prozess Ressourcen anfordern wird. Es kann somit im Voraus planen und eine passende Taktik zur Verteilung der Ressourcen wählen.

Die einfachste Lösung ist, dass jeder Prozess von Beginn an mitteilt, wie viele Instanzen er von einer Ressource maximal brauchen wird. Ein Deadlock-Avoidance Algorithmus entscheidet dann, in welcher Reihenfolge die Ressourcen verteilt werden sollen.

Safe State

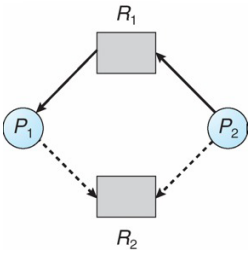
Ein Zustand (state, bestimmt durch die Anzahl freier Ressourcen) ist „**safe**“, wenn das System Ressourcen jedem Prozess zuweisen kann und immer noch einen Deadlock vermeiden kann. Zudem existiert eine Sequenz aller Prozesse <P1, P2, …Pn>, wobei für jeden Pi, die Anfragen die er noch maximal machen kann, durch die verfügbaren Ressourcen plus jene, die von Pj (j < i) gehalten werden, bedient werden können.



Wenn nun Ressourcen, die Prozess Pi braucht, nicht gerade verfügbar sind, kann Prozess Pi warten bis alle Pj fertig sind und Pi kann diese danach verwenden und seine Aufgabe erledigen. Wenn Pi fertig ist, dann kann Pi+1 die Ressourcen verwenden und diese danach für den nächsten Prozess freigeben (und so weiter).
Wenn keine solche Sequenz vorhanden ist, ist das System „**unsafe**“. Ein „unsafe“ Zustand könnte zu einem Deadlock führen. Wohingegen ein „safe“ Zustand garantiert, dass niemals ein Deadlock auftreten wird.

Zwei Algorithmen

Resource-Allocation Graph



Wenn jede Ressource nur eine einzelne Instanz hat, kann man einen Resource-Allocation Graph verwenden, welcher um die Kante „**claim**“ erweitert wird. Eine Kante „claim“ soll aufzeigen, dass ein Prozess irgendwann in der Zukunft diese Ressource benötigen wird – er erhebt also im Voraus Anspruch auf diese Ressource. Claim-Kanten werden als gestrichelte Pfeile dargestellt. Sobald der Prozess die Ressource benötigt, verwandelt sich die claim Kante in eine Request Kante und sobald der Prozess die Ressource erhalten hat und verwenden kann, wird aus der Request Kante eine Assignment Kante. Nach der Freigabe der Ressource wird die Kante wieder zu einer Claim Kante.
Agenommen der Prozess Pi fordert die Ressource Rj an. Dies wird nur dann stattgegeben, wenn die Zuweisung zu keinem Cycle führt. Ein Cycle-Detection-Graph checkt daher stets, ob dies nicht geschehen wird (er kann sozusagen ja vorausschauen, da er dank den Claim Kanten weiss, welche Kombinationen auftreten können).

Banker's Algorithm

Der eben beschriebene Resource-Allocation Graph kann nicht angewendet werden, wenn es mehrere Instanzen einer Ressource gibt.
Mit diesem Algorithmus hier können mehrere Instanzen betreut werden. Wenn ein neuer Prozess ins System kommt, muss er angeben, was die maximale Anzahl an Instanzen jeder Ressource ist, die er benötigt. Diese Nummer darf nicht grösser sein, als die totale Anzahl Ressourcen im System.Wenn also eine gewisse Anzahl Ressourcen angefordert werden, muss das System überprüfen, ob es damit nicht in einen „unsafe“ Zustand gerät. Falls alles okay ist, kann es die Ressourcen zuweisen, ansonsten muss der Prozess warten, bis andere Prozesse ihre Ressourcen wieder freigeben.

Sei n = Anzahl an Prozessen und m = Anzahl an Ressourcen Typen

Available: Vector der Länge m
Wenn Available[j] = k ist, dann sind k Instanzen des Ressource-Typs Rj verfügbar
Max: n x m Matrix
Max[i,j] = k sagt aus, dass der Prozess Pi höchstens k Instanzen der des Ressource-Typs Rj braucht
Allocation: n x m Matrix
Allocation[i,j] = k sagt aus, dass im moment gerade k Instanzen des Ressource-Typs Rj dem Prozess Pj zugewiesen sind
Need: n x m Matrix
Need[i,j] = k sagt aus, wieviele Instanzen noch zusätzlich vom Ressource-Typ Rj gebraucht werden von Prozess Pi, damit er seine Aufgabe beenden kann.

Need[i,j] = Max[i,j] - Allocation[i,j]

Nun kann man diese Vorgaben dazu verwenden, herauszufinden, ob ein System im Safe Zustand ist oder nicht. Der Algorithmus ist wie folgt aufgebaut:

Safety Algorithmus

1. Seien *Work* und *Finish* Vektoren der Länge *m* und *n*. Initialisiere *Work* mit *Available* und *Finish[i] = false* für i = 0, 1, …, n -1.

2. Finde einen Index i, so dass
   a. *Finish[i] == false*
   b. $Need_i \leq Work$
   Falls kein solches i existiert, gehe zu Schritt 4.

3. $Work = Work + Allocation_i$
   Finish[i] = true
   Geh zu Schritt 2.

4. If *Finish[i] == true* für alle i, dann ist das System in einem safe Zustand.

Nun beschreiben wir noch den Algorithmus für die Zuweisung eines Requests.
$Request_i[j] = k$ sagt aus, dass Prozess Pi k Instanzen des Ressource Typs Rj möchte.

1. Wenn $Request_i \leq Need_i$ ist, geh zu Schritt 2. Ansonsten ist sowieso schon etwas „faul“, denn eigentlich dürfte der Request nie grösser sein als der Need…

2. Wenn $Request_i \leq$ Available ist, geht zu Schritt 3. Ansonsten muss Pi warten, bis Ressourcen wieder frei sind.

3. Das System muss nun die angeforderten Ressourcen nach folgendem Schema zuteilen:

   $Available = Available - Request_i$;
   $Allocation_i = Allocation_i + Request_i$;
   $Need_i = Need_i - Request_i$;

   ➔ Wenn **Safe**, dann können die Ressourcen Pi zugewiesen werden
   ➔ Wenn **Unsafe**, Pi muss warten

Beispiel zu Banker's Algorithm

5 Prozesse P0 bis P4;
3 Ressourcen Typen: A (10 Instanzen), B (5 Instanzen), C (7 Instanzen)

Zum Zeitpunkt T_0 sieht es wie folgt aus:

	Allocation	*Max*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Die Matrix *Need* ist definiert als *Max – Allocation*

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Das System ist im Safe Zustand, da die Sequenz < P1, P3, P4, P2, P0> die „Safety" Kriterien erfüllt.

Beispiel 1: P1 fordert 1 Instanz von A und 2 Instanzen von C an: $Request_1 = (1,0,2)$

Überprüfe, ob *Request* ≤ *Available* stimmt, also ob (1,0,2) ≤ (3,3,2) → true
Falls der Request stattgegeben wird, wird die Liste nachher so aussehen:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2 |	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Nun müssen wir wieder schauen, ob unser System im Safe Zustand ist. Wir führen wieder den Safety Algorithmus (rotgefärbter Algorithmus oben) aus und finden heraus, dass die Sequenz <P1, P3, P4, P0, P2> die Anforderungen erfüllt. Daher können wir sofort den Request von P1 stattgeben.

Recovery from Deadlock (Wiederherstellung nach Deadlock)

Prozess Termination

Um Deadlocks zu eliminieren, kann man einen Prozess abbrechen. Es gibt dazu zwei Methoden, wobei bei beiden das System dem Prozess die Ressourcen wegnimmt.

Abort all deadlocked processes (alle deadlocked Prozesse abbrechen): Diese Methode wird offensichtlich jeden Deadlock aufheben, da alle Prozesse, die im Deadlock verwickelt sind, abgebrochen werden – ist jedoch sehr rabiat, da vielleicht diese Prozesse schon sehr lange am Rechnen waren und so alle Daten verloren gehen

Abort one process at a time until the deadlock cycle is eliminated (breche einen Prozess nach dem anderen ab, bis der Deadlock Cycle eliminiert wurde) : Man bricht einen der involvierten Prozesse ab und schaut dann wieder, ob der Deadlock weiterhin existiert, falls ja, bricht man einen weiteren ab und so fort. Braucht viel Zeit, da nach jedem Abbruch wieder überprüft werden muss, ob es noch einen Deadlock Cycle hat. Hier eine Empfehlung, wie der nächste Prozess, der abgebrochen werden muss, ermittelt werden kann:

- Priorität des Prozesses
- Wie lange war der Prozess schon am Rechnen und wie lange braucht er noch für seine Beendigung
- Ressourcen die der Prozess brauchte
- Ressourcen, die der Prozess für die Beendigung benötigt
- Wie viele Prozesse müssen beendet werden
- Ist es ein interaktiver Prozess oder ein batch?

Resource Preemption

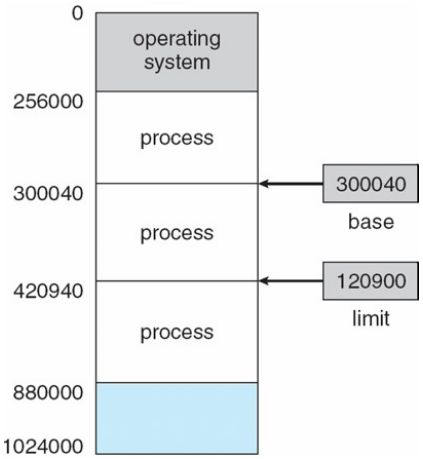
Um Deadlocks zu eliminieren, kann man Resource Preemption verwenden. Wir nehmen also schrittweise Prozessen Ihre Ressourcen weg und geben Sie anderen Prozessen, solange bis der Deadlock aufgehoben wurde.

Drei Probleme müssen berücksichtigt werden:

1. **Select a victim (wähle ein Opfer):** welche Ressourcen und Prozesse sollen preempted werden? Man muss die Kosten minimieren in dem man Faktoren wie bereits verwendete Rechenzeit, voraussichtliche Zeit bis zur Termination, bereits verwendete Ressourcen etc berücksichtigt.
2. **Rollback**. Wenn wir einem Prozess eine Ressource wegnehmen kann er logischerweise nicht mehr normal fortfahren. Wie müssen ihn in einen safe Zustand zurückrollen (roll back), von wo aus er neu starten kann. Es ist jedoch schwierig festzustellen, was ein safe Zustand ist.
3. **Starvation**. Wie können wir sicherstellen, dass Starvation nicht auftritt? Wenn wir immer die Kosten (siehe Punkt 1) als Kriterium nehmen, kann es sein, dass immer der gleiche Prozess ausgewählt wird und somit es zu Starvation kommt. Wir müssen also sicherstellen, dass ein Victim nur eine begrenzte Anzahl lang aufgerufen werden kann, die Anzahl Rollbacks muss also in den Kostenfaktoren miteinbezogen werden

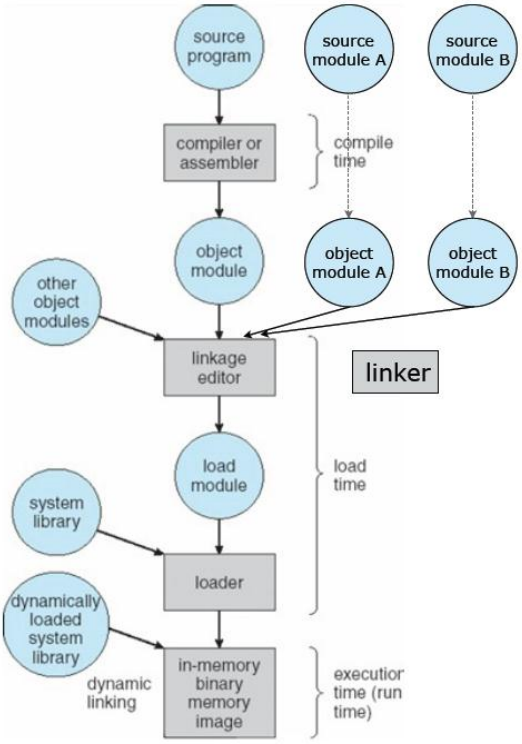
Hintergrund
Programme müssen von der Harddisk ins Memory gebracht werden, da CPU nur auf Memory (Arbeitsspeicher, RAM) und seine eigene Register (sind in der CPU eingebaut) direkt zugreifen kann. Den Zugriff auf die Register schafft die CPU in einem CPU Zyklus (oder weniger), für den Zugriff aufs Memory benötigt es mehr als einen Zyklus (denke hier an den bereits erklären „Memory Stall" Effekt auf Seit 26!). Da dies nicht tolerierbar ist, wird zwischen den Memory und der CPU ein Zwischenspeicher eingebaut, **Cache** genannt.
Zudem muss sichergestellt werden, dass nur erlaubte Operationen ausgeführt werden und sich Prozesse nicht gegenseitig in ihre Speicherbereiche auf dem Memory reinschreiben. Diese Sicherstellungen müssen von der Hardware angeboten werden, in dem jeder Prozess einen separaten Memorybereich hat.

Base und Limit Registers
Diesen Memorybereich können wir mit einem base register und einem limit register festlegen. Der base register merkt sich die kleinste legale physische Memoryadresse und der limit register gibt die Grösse des Bereichs an. Angenommen der base register ist 300040 und der limit register 120900, dann kann das Programm alle Adressen von 300040 bis 42939 (inklusive) legal verwenden.



Multistep Processing of a User Program

- Translation
  - Source written in symbolic programming language
  - Conversion of symbolic instructions into machine specific form
- Linking
  - Complex systems made up by multiple modules
  - Resolving of external references to other modules
  - Conversion from logical addresses to physical memory locations
- Loading
  - Transfer of code from disk to main memory
  - Resolving of external references to system libraries



Das Binden von Instruktionen und Daten an Memoryadressen auf drei verschiedenen Ebenen geschehen:

Compile time: wenn die Memorylocation bereits zu Kompilierzeiten bekannt ist, kann absoluter Code (**absolute code**) generiert werden. Falls die Location ändert, muss der Code neu kompiliert werden.

Load time: wenn zu Kompilierzeiten die Location noch unbekannt ist, muss „relocatable code" generiert werden. In diesem Fall wird das finale Binding erst zur Load Time geschehen.

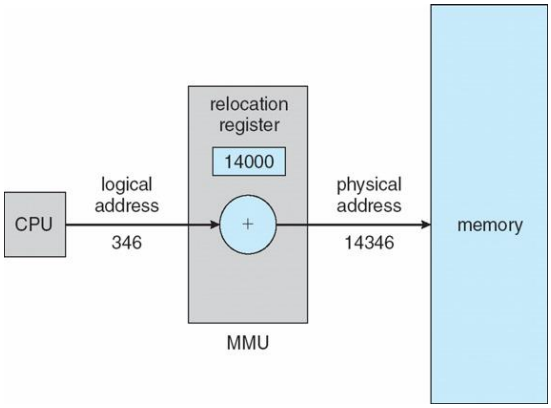
Execution time: wenn der Prozess während seiner Ausführung von einem Memorybereich zu einem anderen verschoben werden kann, muss das Binding bis zur run time verschoben werden. Dafür ist Hardwareunterstützung notwendig.

Logical vs. Physical Adresse Space

Logical Address: wird von der CPU erzeugt, auch virtual address genannt
Physical Address: Adresse, welche die Memoryeinheitsieht

Logical und physical Adressen sind in compile time und load time die gleichen. In execution time unterscheiden sich die logical (virtual) und physical Adressen.

Relocation Register
Das run time Mapping von der virtual Address zur physical Adress wird von einer memory-management Unit (MMU) erledigt (siehe Bild). Der base register wird somit zum relocation register. Somit wird das Programm niemals die echte physische Adresse sehen. Wenn es also zbsp einen Pointer zur Location 346 erzeugt, wird zwar physisch auf 14346 zugegriffen, aber für das Programm sieht es immer nach 346 aus.



Dynamic Loading
Dynamic Loading bietet einen Mechanismus, dass nicht das ganze Programm mit seinem gesamten Code ins Memory geladen wird. Erst wenn eine gewisse Routine (zbsp Error Routine) wirklich benötigt wird, wird diese ins Memory geladen. Die totale Programmgrösse kann also zwar gross sein, die aber tatsächlich ausgeführte Grösse kann viel kleiner sein. Somit werden gewisse Routinen gar nie geladen (zbsp wenn es nie einen Error gibt, wird auch nie die Error Routine aufgerufen). Damit diese ganze Sache aber auch funktioniert, muss der der Code relocatable sein, sprich wenn eine Routine neu hinzugeladen wird, muss der Loader die Adresstabelle des Programms updaten (damit das Programm auch weiss, wo nun diese Routine gespeichert ist).

Dynamic Linking
Programme müssen manchmal auf System-Libraries zugreifen. Damit man nicht jedem Programm die gesamte Library mitgeben muss (was Festplatten- und Memoryspeicher verschwenden würde), sollte man dafür sorgen, dass im Programm jeweils eine Referenz zur Library gemacht wird (die ja sowieso mit dem OS mitgeliefert wird). Dies kann man sicherstellen, in dem man ein kleines Stück Code, **stub** genannt, implementiert, dass mitteilt, wie man die benötigte Library-Routine aufruft. Dieser Stub ersetzt sich dann selber mit der Adresse der Routine und führt die Routine aus. Das OS muss lediglich schauen, ob die Routine im Adressenraum des Prozesses ist. Dank dieser Vorgehensweise braucht es nur eine Kopie der System-Library auf dem System. Somit muss man sich auch nicht sorgen, dass Updates der System-Libraries „missachtet" werden. Da alle Programme immer stets auf die aktuellste Library zugreifen. Es kann aber sein, dass gewisse Programme mit der neuen Version der System-Library nicht klar kommen, daher werden jeweils Versionsnummern verteilt. Falls ein Programm eine ältere Version braucht, lädt es sich extra für sich diese Version ins Memory. Dies wird **shared Libraries** genannt.

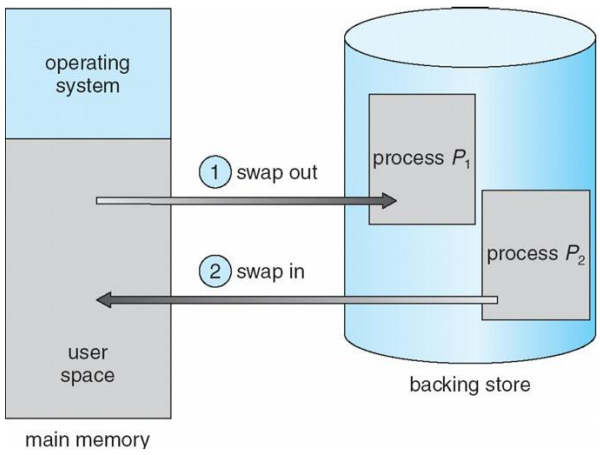
Swapping

Ein Prozess kann temporär aus dem Memory zu einem anderen Speicher (zbsp Festplatte) geswapped werden und dann später für die weitere Ausführung wieder zurückgeholt werden.

Backing store (Speicher, der den swapped Memoryinhalt aufnimmt): muss gross genug sein, um alle Memory-Abbilder für alle Benutzer aufnehmen zu können.

Roll out, roll in: Swapping Variante, die für priority-based Scheduling Verfahren verwendet wird: tief-priorisierter Prozess wird raus-geswapped damit höher-priorisierter Prozess geladen und ausgeführt werden kann (der tiefer priorisierte Prozess wird danach wieder reingeholt, „roll in").

Der grösse Faktor beim Swappen ist die Zeit, welche direkt proportional zur Grösse des Speicherinhalts, der geswapped wird, ist.

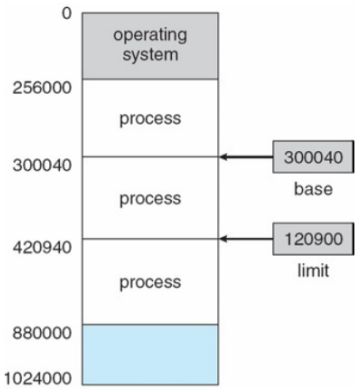


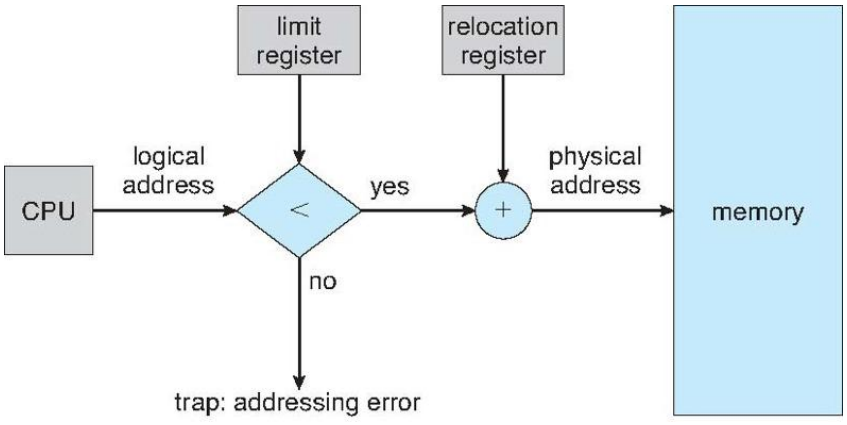
Contiguous Allocation (zusammenhängende Allokation)

Memory wird normalerweise in zwei Bereiche eingeteilt:

Das **Betriebssystem** wird normalerweise in **den tieferen Memoryadressbereichen** abgelegt zusammen mit dem Interrupt Vector und **User Prozesse** werden in den **höheren Memoryadressbereichen** gehalten.

- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - *Base register* contains value of smallest physical address
  - *Limit register* contains range of logical addresses – each logical address must be less than the limit register
  - MMU maps logical address dynamically

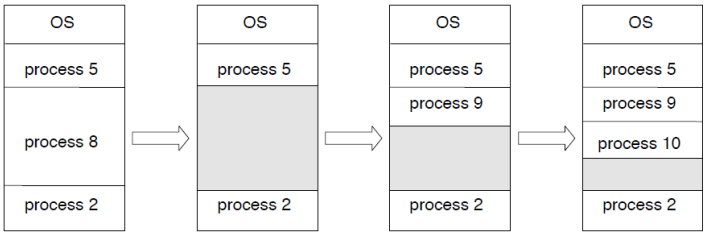




Memory Allocation

Nun können wir uns mit der Memory-Allocation befassen. Eine der einfachsten Methoden ist es, wenn man das Memory in mehrere fix-sized Teile aufteilt. Jeder Teil kann genau einen Prozess beherbergen. Der Grad an Multiprogramming ist daher an die Anzahl Teile gebunden. Dies wird **multiple-partition method** genannt. Das OS merkt sich, welche Teile des Memory bereits besetzt sind und welche noch fei sind. Ein Prozess wird in ein Loch (Hole) geladen, dass gross genug ist. Es gibt unterschiedlich grosse Löcher. Siehe nachfolgende Erklärung:

- Multiple-partition allocation
  - Hole – block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about:
    a) allocated partitions b) free partitions (holes)



Nun wäre es gut, wenn man mit etwas Taktik ein passendes Loch aussucht. Es gibt dazu drei bekannte Vorgehensweisen:

First Fit: nimm das erste Loch, dass gross genug ist
Best Fit: nimm das kleinste Loch, das gerade noch gross genug ist, dazu muss aber die ganze Liste freier Löcher durchsucht werden
Worst Fit: nimm das grösste Loch, auch hier muss die gesamte Liste durchsucht werden

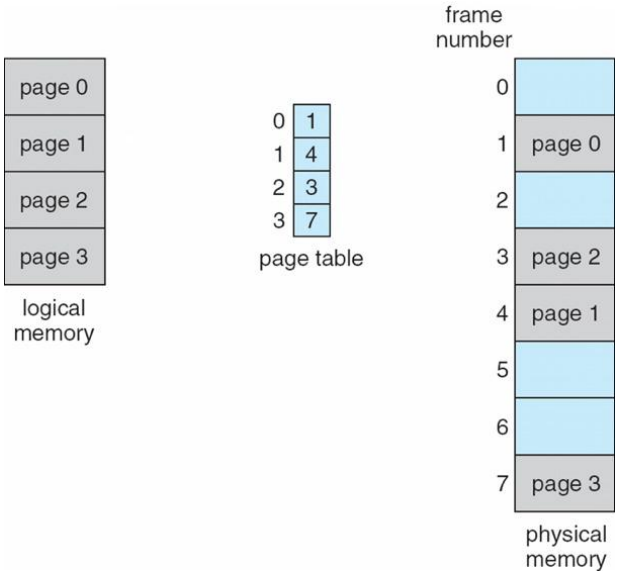
➔ First Fit und Best Fit sind besser als Worst Fit in Bezug auf Geschwindigkeit und Speicherverwendung

Fragmentation

Es kann nun sein, dass es zwar viele kleine Löcher gibt, die alle zu klein sind, die aber – wenn sie zusammengefasst werden – wieder genügend gross wären für einen Prozess. Diese Fragmentierung (zerstückelt) wird durch die Best Fit, First Fit und Worst Fit Strategien verursacht.
Dieses Fragmentierungssproblem kann jedoch gelöst werden. Dies heisst **compaction**.
Man schiebt die freien Memorylöcher zu einem grossen Block zusammen. Dies geht aber nur, wenn der Code relocatable ist, also auch an einer anderen Stelle im Memory ausgeführt werden kann.

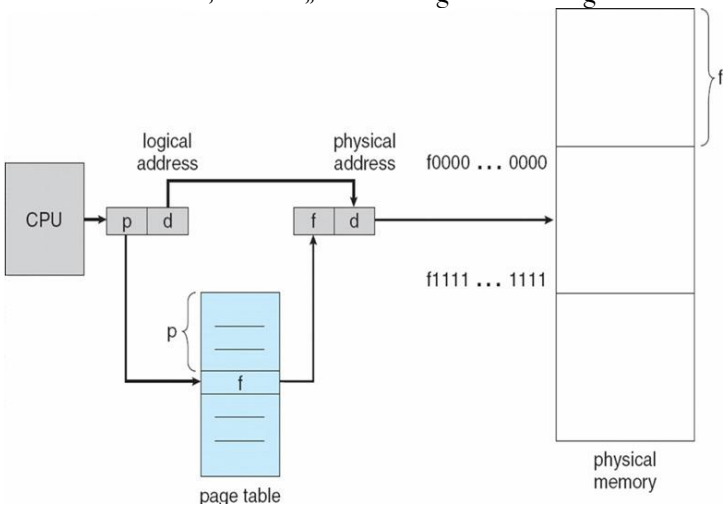
Paging

Paging verhindert die externe Fragmentierung, weil es erlaubt, dass die physischen Adressen eines Prozesses nicht zusammenhängend sein müssen. Dabei wird das physische Memory in gleich grosse Stücke, Frames genannt, zerteilt und das logische Memory ebenfalls in Stücke dieser gleichen Grösse, Pages genannt. Wenn ein Prozess nun ausgeführt wird, werden seine Pages in verfügbare Frames geladen.
Jede Adresse, die von der CPU generiert wird, ist in zwei Teile geteilt: eine page Nummer (p) und ein page offset (d). Die Page Nummer wird als Index für die Page Table verwendet. Die Page Table beinhaltet die Base Adresse jeder Page im physischen Memory. Diese Base Adresse, kombiniert mit dem page offset, ergibt die physische Adresse. Nachfolgendes Bild illustriert dies:



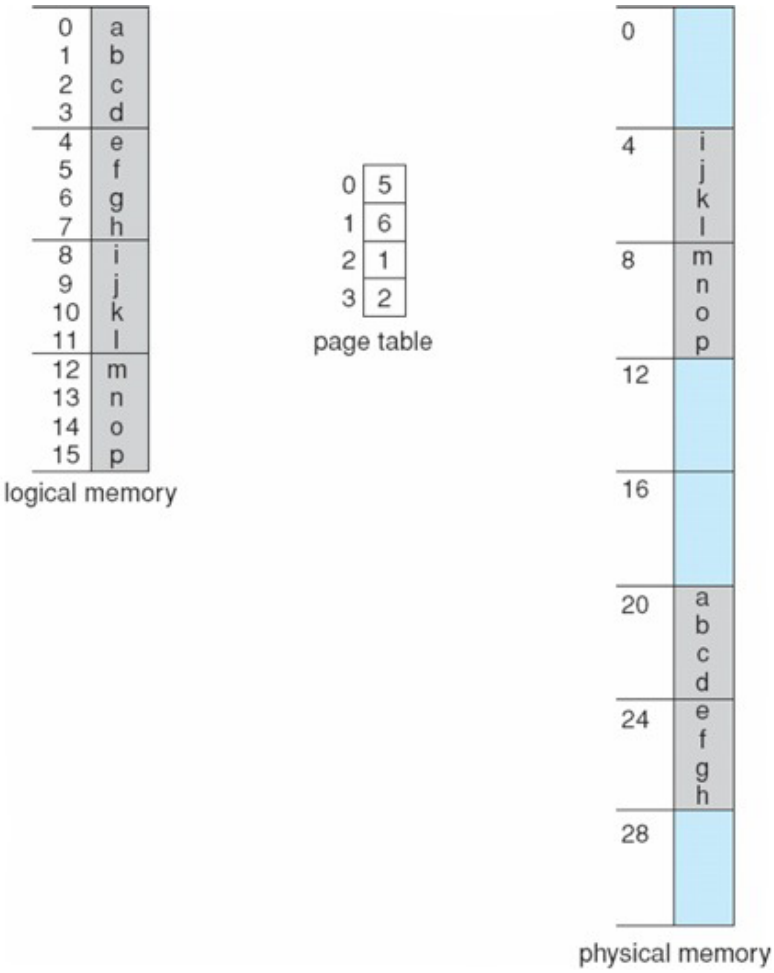
Wie man sieht, ist Page 3 laut Page Table im 7. Frame des physischen Memorys.

Und hier sieht man, wie die „Übersetzung“ bildlich dargestellt werden kann:



Die Grösse einer Page ist normalerweise ein Vielfaches von 2, oft zwischen 512 Bytes und 16MB. Die Tatsache, dass es ein Vielfachtes von 2 ist, macht die Übersetzung einer logischen Adresse in eine physische relativ einfach. Wenn die Grösse des logischen Adressraums 2^m ist und eine Pagegrösse 2^n Adresseneinheiten (zbsp Bytes) ist, dann können m-n Bits für die Page Nummer verwendet werden und n Bits geben den offset innerhalb der Page an.

Auf der nächsten Seite dazu ein Beispiel:



Wie man hier sieht, wird eine Pagegrösse von 4 Bytes und ein physisches Memory von 32 Bytes (8 pages) verwendet.

Nun kann man mithilfe der Page Table errechnen, wie sich ein Wert aus dem logischen Memory uf das physische Memory abbildet. Als Beispiel nehmen wir die logische Adresse 0. Diese ist in Page 0 mit Offset 0. Gemäss Page Table befindet sich diese Page in Frame 5 des physischen Memorys, die logische Adresse wird demnach also auf die physische Adresse 20 abgebildet [= (5 x 4) + 0]. Die logische Adresse 3 (page 0, offset 3) wird auf Adresse 23 abgebildet [= (5 x 4) + 3]. Und die logische Adresse 13 zbsp wird auf die physische Adresse 9 gemapped [= (2 x 4) + 1].

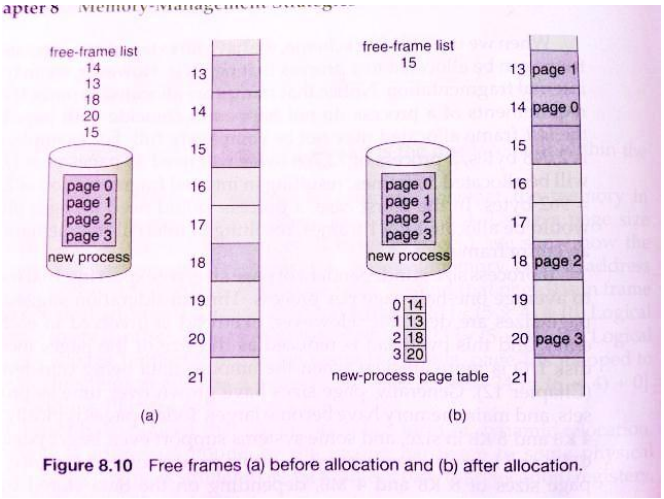


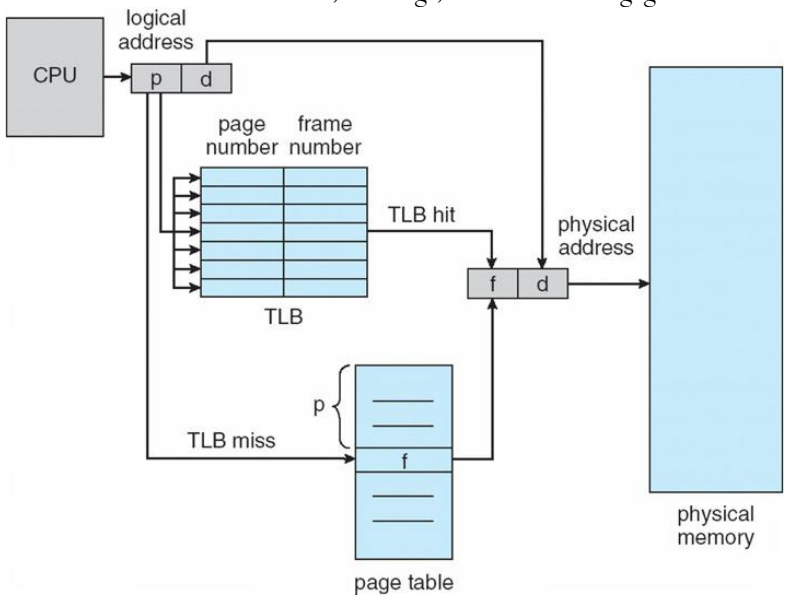
Figure 8.10 Free frames (a) before allocation and (b) after allocation.

OS führt Buchhaltung darüber, welche Frames noch frei sind und welche von welchem Prozess besetzt sind

Page Table Implementierung

Jeder Prozess hat seine eigene Page Table (kann sehr gross sein), die im Memory gehalten wird. Ein page-table base register (**PTBR**) zeigt auf die Page Table. Hierbei entsteht aber das Problem, dass wenn man nun auf eine User Memory Location Zugriff haben will, zuerst einmal auf das Memory zugreifen muss (via PTBR) um in der Page Table nachschauen zu können, wo nun diese Memory Location ist und dann mit dieser Information erneut auf die richtige Stelle im Memory zugreifen muss. Um also auf ein Byte zugreifen zu können, braucht man zwei Memoryzugriffe, was nicht tolerierbar ist (verdoppelt Aufwand/Zeit).
Dieses Memory-Access-Problem kann man mit einer speziellen, kleinen, schnellen Such-hardware Cache (fast-lookup hardware cache), **translation look-aside buffer** (TLB) genannt, lösen. Das ist ein high-speed Memory. Jeder Eintrag im TLB besteht aus zwei Teilen: einem Key (oder tag) und einer value. Wenn man dem TLB ein Item gibt, vergleicht er das simultan mit allen Keys und falls einen passenden Eintrag gibt, gibt er den dazugehörigen Value zurück.
TLB wird zusammen mit Page Tables wie folgt verwendet: das TLB beinhaltet nur einige Page Table Einträge. Wenn nun von der CPU eine logische Adresse erzeugt wurde, wird die Page Nummer den TLB gezeigt und der checkt ab, ob er einen passenden Eintrag hat und falls ja, gibt er die dazugehörige Frame Nummer zurück (somit muss nicht extra aufs Memory zugegriffen werden und dort in der Page Table nachgeschaut werden). Falls die Page Nummer nun aber nicht im TLB eingetragen ist (diese Situation wird auch **TLB miss** genannt), muss der Extra-Zugriff aufs Memory gemacht werden und dort in der Page Table nachgeschaut werden, zusätzlich wird man diese extrahierte Information nun ins TLB ablegen, dass es bei der nächsten Abfrage gespeichert ist. Falls das TLB voll ist (TLB ist relativ klein, kann zbsp nur 1024 Einträge halten), muss das OS einen Eintrag finden, den es entfernen kann um Platz zu machen. Nach welcher Vorgehensweise dies geschieht ist, ist unterschiedlich. Kann von LRU (least recently used) bis zu random sein.
Für zusätzliche Sicherheit speichern einige TLB zusätzlich adress-space identifiers (ASIDs) zu jedem Eintrag. Mit dieser ASID kann ein Prozess identifiziert werden und so sicherstellen, dass nicht im falschen Adressraum operiert wird.

Hier nun noch eine Grafik dazu, die zeigt, wie das TLB eingegliedert ist.

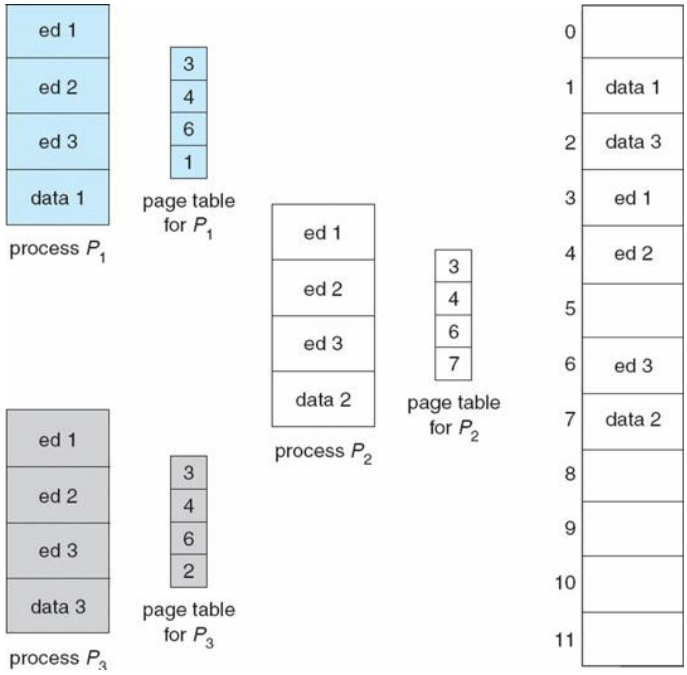


Nun kann man zusätzlich noch die effective memory-access time berechnen, um zu sehen wie effektiv der Einsatz eines TLB ist. Nehmen wir an, dass in 80% (wird auch 80-percent **hit ratio** genannt) der Fälle ein Eintrag im TLB gefunden wird und man dafür 20 Nanosekunden aufwendet und weitere 100ns um nachher aufs Memory zuzugreifen. Dann muss man in den anderen 20%, wo es keinen Eintrag im TLB gibt, 220ns aufwenden, weil man 20ns für das Nachschauen im TLB brauchte und zweimal aufs Memory zugreifen muss (einmal um den Page Table Eintrag zu finden und ein weiteres Mal um das eigentliche Byte zu lesen).
Somit lasst sich die effective access time wie folgt berechnen:
0.80 x 120 + 0.20 x 220 = 140 Nanosekunden.

Shared Pages

Ein Vorteil von Paging ist, dann man gleichen Code „sharen“ kann. Stellt man sich nun die Situation vor, dass 40 User gleichzeitig einen Text Editor ausführen, wäre es von Vorteil, wenn alle User den gleichen Code im Memory verwenden könnten, anstatt dass 40mal der gleichen Text-Editor-Code ins Memory geladen wird.
Das kann natürlich nur gemacht werden, wenn es sich um non-self-modifying-Code handelt, sich also niemals ändert während der Ausführung. Natürlich haben alle 40 User bzw. ihre Prozesse noch private Daten (wie zbsp der Text, der in den Editor eingetippt wird), diese private Daten werden nicht geteilt und können irgendwo im logischen Adressraum gespeichert sein. Die nachfolgende Grafik verdeutlicht die Idee:

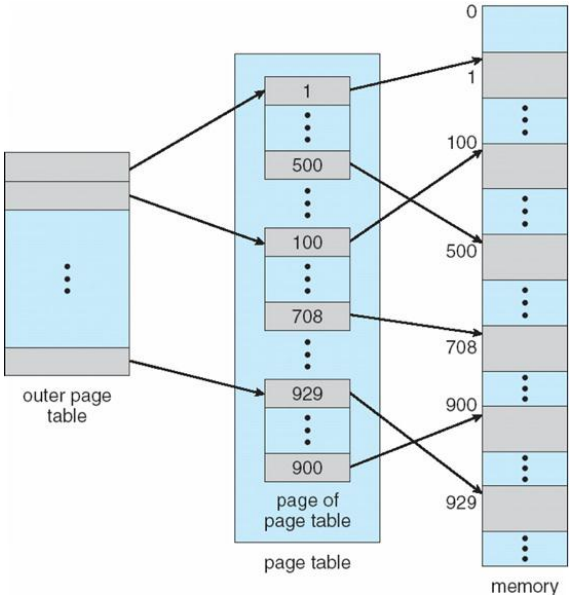
Man sieht hier drei Prozesse, die alle drei Pages haben (ed1, ed2, ed3), die sich auf diesen non-self-modifying-Code beziehen. Die Page Tables zeigen nun alle auf die gleichen Frames (3,4,6). Nur die jeweiligen Data-Pages (data1, data2, data3) sind jeweils an unterschiedlichen Orten (1, 7, 2). Auf diese Art kann man viele viel-benutzte Programme „sharen“, sofern ihr Code „reentrant“ (englisch: ablaufinvariant, also sich nicht verändert) ist.



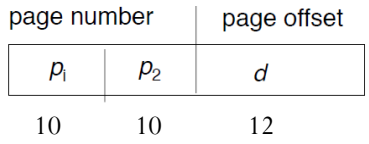
Page Table Struktur

Hierarchische Struktur

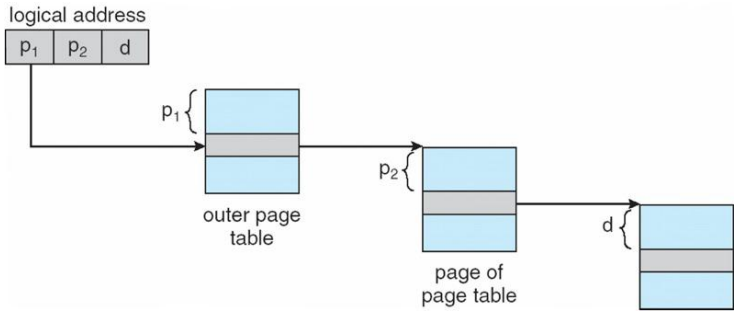
Aufgrund dessen, dass moderne Computersysteme einen sehr grossen logischen Speicherraum haben (2^{32} bis 2^{64}), wird auch die Page Table selber sehr gross. In einem 32-bit System mit 4KB (2^{12}) grossen Pages, gäbe es über 1 Million Einträge in der Page Table ($2^{32}/2^{12}$). Wenn nun jeder Eintrag 4 Bytes beinhaltet, bräuchte jeder Prozess nur alleine für die Page Table 4MB im Memory. Da wir die Page Table nicht zusammenhängend im Memory allozieren wollen (Kommentar: weiss nicht, wieso man das nicht will, gemäss Buch ist das aber so… evtl. zu aufwändig), kann man die Page Table in kleinere Stücke teilen. Ein Weg wäre, einen two-level paging Algorithmus zu verwenden, welcher die Page Table selber nochmals paged (siehe Grafik).



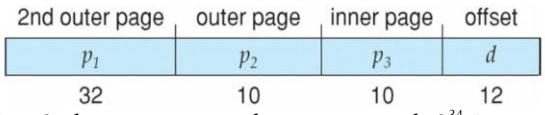
Beispiel: nimm ein 32-bit System an und eine Page-Grösse von 4KB. Eine logische Adresse wird dann in eine Page Nummer, bestehend aus 22 bits und einem page offset, bestehend aus 12 bits, geteilt. Da die Page Table selber gepaged ist, wird die Page Nummer noch weiter geteilt in eine 10-bit page Nummer und einem 10-bit Page offset, was dann so aussieht:



Wobei Pi der Index der äusseren Page Table ist und P2 zeigt auf die Page innerhalb der äusseren Page Table. Hier sieht man den Ablauf:



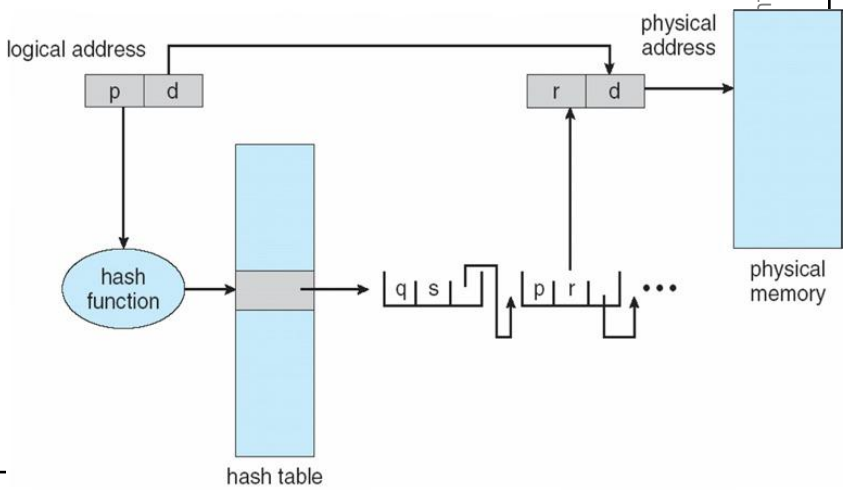
In einem 64-bit System wäre ein two-level paging Schema jedoch nicht mehr angebracht. Bedenkt man, dass in einem solchen System, bei 4KB (2^{12}) grossen Pages, 2^{52} Einträge vorhanden wären. Daher muss man die äussere Page Table noch weiter aufteilen, zbsp so:



Die 2nd outer page ist aber immer noch 2^{34} Bytes gross, was noch zu gross ist. Der nächste Schritt wäre also ein four-level-paging Schema. Das ist irgendwie dann auch nicht mehr angemessen, weshalb man sagt, dass solche hierarchische page Tables für 64-bit Systeme nicht angebracht sind.

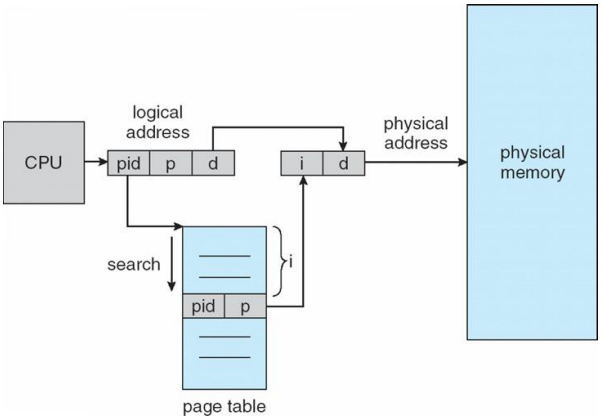
Hash Page Tables

Eine andere Lösung für das Handhaben von Adressräumen, grösser als 32 bit, ist das Verwenden von **hashed page tables**, wobei der Hashwert die virtuelle Page Nummer ist. Jeder Eintrag in der Hashtabelle beinhaltet eine Linked List von Elementen, die zur gleichen Location gehören. Jedes Element beinhaltet 3 Felder: (1) die virtuelle Page Nummer, (2) der Wert des gemappten Page Frames und (3) einen Pointer zum nächsten Element in der Liste. Der Algorithmus funktioniert wie folgt: Die virtuelle Page Nummer in der virtuellen Adresse wird in die Hashtabelle gehashed. Nun wird die virtuelle Page Nummer mit dem Feld 1 des ersten Elements verglichen, wenn es einen Treffer gibt, wird das Feld 2 (das passende Page Frame) zurückgegeben. Wenn es keinen Treffer gibt, wird dank dem Pointer jeweils im nächsten Element weitergesucht und so weiter bis es einen Treffer gibt. Dies wird in folgender Grafik verdeutlicht:



Inverted Page Tables

Eine inverted Page Table hat einen Eintrag für jede echte Page (oder Frame) des Memorys. Jeder Eintrag beinhaltet die virtuelle Adresse der Page, wo diese im Memory gespeichert ist zusammen mit Informationen über den Prozess, der diese Page besitzt.

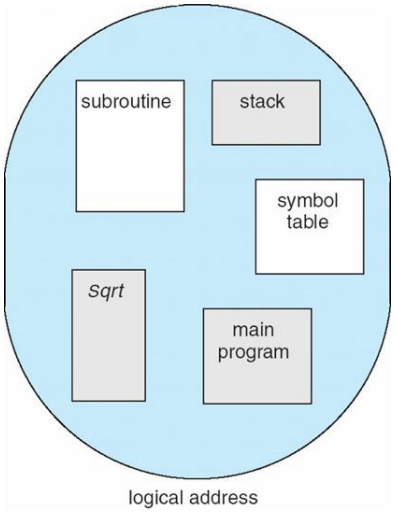


Segmentation

Aufgrund des Pagings ist es unvermeidbar, dass sich die Sicht des Benutzers auf das Memory von der eigentlichen physischen Adressverteilung unterscheidet (der User meint, dass er auf die Adresse 17 zugreift, in Wahrheit wird dies aber gemapped/paged auf die echte Adresse und ist dann vielleicht die Adresse 1928).

Benutzer stellen sich die Ansicht eines Programmes so vor:

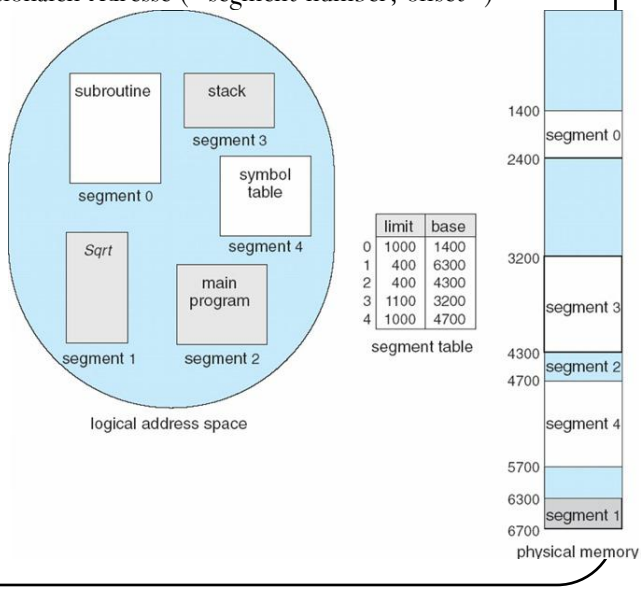
Ein Segment ist eine logische Einheit wie zbsp: Main Program, Prozedur, Funktion, Methode, Objekt, lokale Variable, Stack, Array, etc.
Segmentation ist ein Memory-management Schema, das diese Ansicht unterstützt. Ein logischer Adressraum ist somit eine Sammlung von Segmenten. Jedes Segment hat einen Namen und eine Länge. Die Adressen wird durch zwei Teile spezifiziert: ein Segmentname und ein Offset (Vergleich dazu die Paging Vorgehensweise, wo der Benutzer nur eine einzige Adresse nennt und die Hardware dies dann in eine Page Nummer und ein Offset übersetzt, alles unsichtbar für den Programmierer). Zur Vereinfachung werden statt Segmentnamen Segmentnummer verwendet, so dass eine logische Adresse wie folgt aufgebaut ist:
<segment-number, offset>



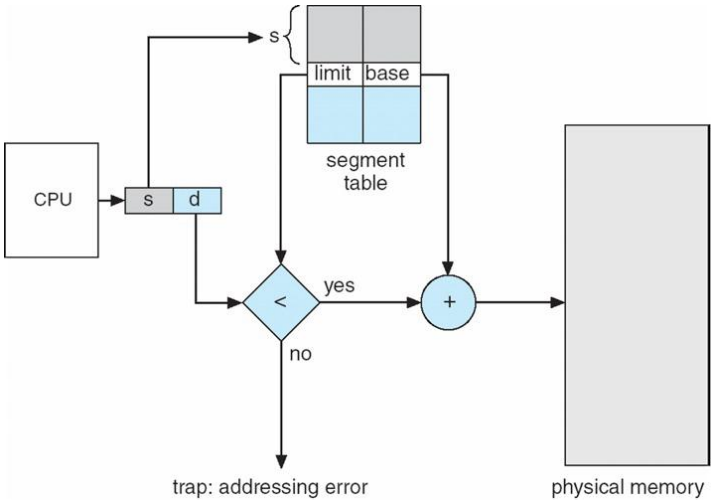
Normalerweise konstruiert der Compiler automatisch diese Segmente. Ein C Compiler könnte zbsp diese separaten Segmente erzeugen:

1. Der Code
2. Globale Variablen
3. Heap
4. Stacks
5. Standard C Library

Obwohl der User jetzt Objekte im Program bei einer zwei-dimensionalen Adresse (<segment-number, offset>) aufrufen kann, ist die physische Adresse im Memory natürlich immer noch eine ein-dimensionale Sequenz an Bytes. Daher muss man die zwei-dimensionale Adresse irgendwie in eine ein-dimensionale Adresse ummappen. Dieses Mapping wird von der **segment table** gemacht. Jeder Eintrag in der segment table hat **eine segment base** und ein **segment limit**. Segment base beinhaltet die startende physische Adresse (also wo das Segment im Memory ist bzw. beginnt) und segment limit spezifiert die Länge des Segments.
Beispiel in der Grafik: will man auf Segment 3, Byte 852 zugreifen, wird das auf 3200 (base, siehe segment table) + 852 gemapped.

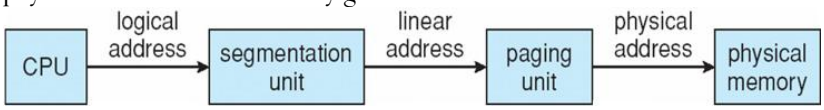


Diese Berechnung sieht man hier grafisch:



Beispiel: Intel Pentium

Der Intel Pentium bietet Segmentation und Segmentation mit Paging an. In Pentium Systeme generiert die CPU logische Adressen, welche an die Segmentation Unit weitergegeben werden. Diese Unit produziert eine lineare Adresse, diese lineare Adresse wird dann an die Paging Unit weitergeleitet, welche aus dieser Adresse die physische Adresse fürs Memory generiert:



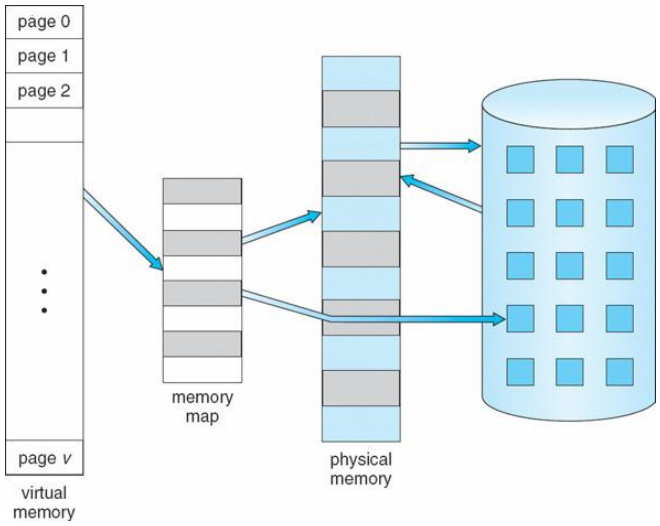
Der logische Adressenraum eines Prozesses wird in zwei Stücke geteilt. Das erste Stück beinhaltet 8K Segmente, die privat dem Prozess gehören und das andere Stück beinhaltet ebenfalls 8K an Segemente, die mit anderen Prozessen geteilt werden. Informationen über den ersten Teil werden in der local descriptor table (LDT) gespeichert, die anderen in der global descriptor table (GDT).

Schaue im Buch Seite 345 – 349 für genauere Ausführungen.

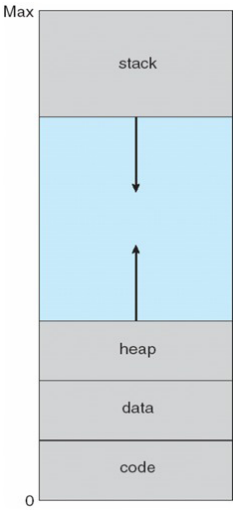
Kapitel 9 – Virtual Memory Management

Virtual Memory ist eine Technik, die es erlaubt, dass Prozesse ausgeführt werden können, die nicht komplett im Memory sind. Ein grosser Vorteil dabei ist, dass Programme grösser als das physische Memory sein können.

Es müssen also nur jene Instruktionen und Daten im Memory sein, die auch wirklich gebraucht werden (zbsp Error Routinen für Fehler, die nur selten auftreten, müssen nicht unbedingt im Memory aktiv sein). Somit kann ein Programmierer auch Programme schreiben, die einen viel grösseren logischen Adressraum haben als die eigentliche physische Adressgrösse.

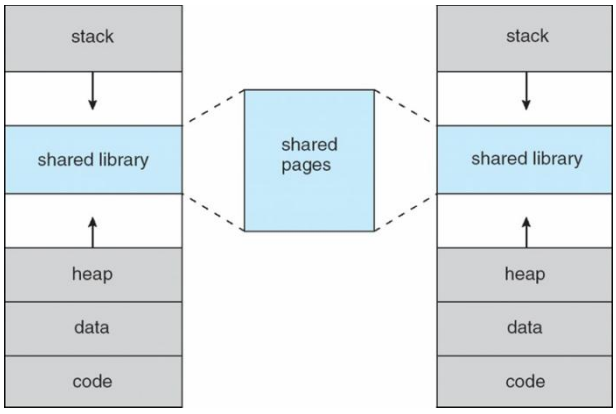


Betrachten wir nun diese Grafik:



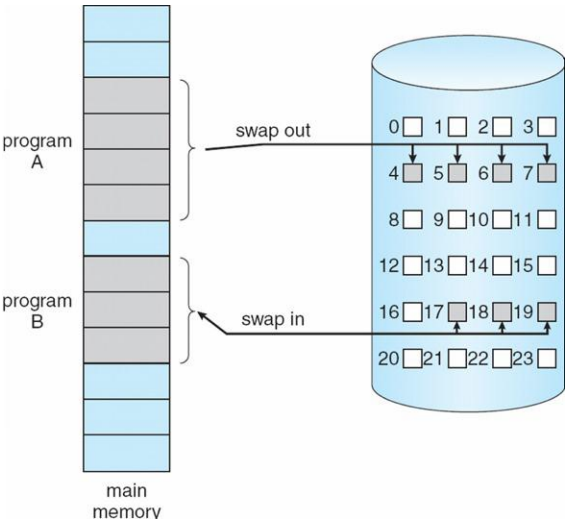
Wie man sieht, kann der Stack abwärts in die freie Fläche wachsen und der Heap aufwärts ebenfalls in diese freie Fläche. Diese Fläche ist solange im virtuellen Memory abgelegt, bis sie auch wirklich vom Stack oder Heap gebraucht wird.

Virtual Memory erlaubt es zudem, dass Files und Memory von zwei oder mehreren Prozessen geshared werden können, in dem das Objekt einfach in den virtuellen Adressraum gemapped wird.

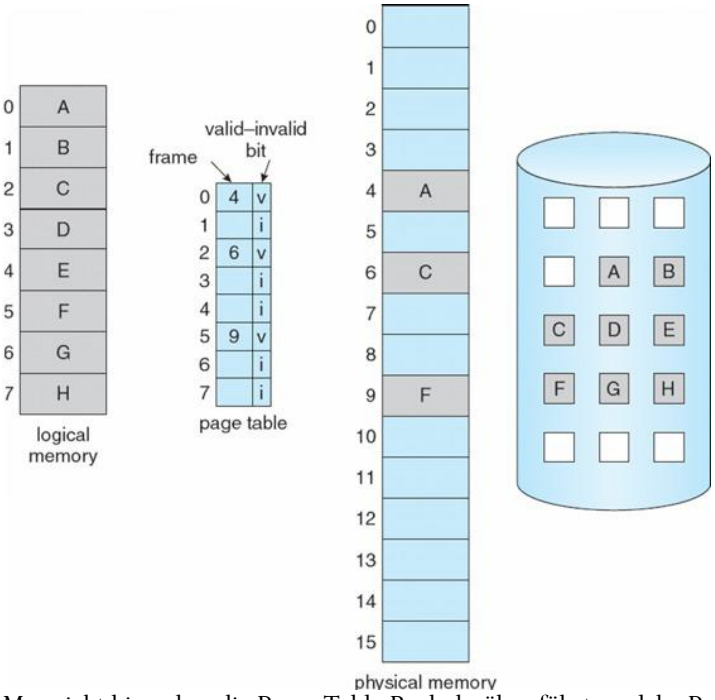


Demand Paging

Demand Paging bringt eine Page nur ins Memory, wenn sie gebraucht wird, dadurch wird weniger I/O gebraucht, weniger Memory, schnellere Reaktion und mehr Programme können ausgeführt werden. Eine Page wird gebraucht, wenn die CPU auf eine Variable, Funktion, Adresse etc referenziert. Das ganze wird von einem „**Lazy Swapper**“ erledigt, der niemals eine Page ins Memory swapped, ausser die Page wird benötigt. „Swapper“ ist zwar ein nicht ganz korrekter Name, da ein Swapper mit ganzen Prozessen „dealt“, daher sollte man hier, wo nur Pages reingeladen werden, besser von einem „**Pager**“ gesprochen werden.



Um die Übersicht zu behalten, welche Pages nun schon im Memory sind und welche noch auf der Disk liegen, kann man valid-invalid Bit jedem Table Eintrag zuweisen (v → in Memory, i → not-in-memory). Standardmässig sind alle Einträge auf i (not-in-memory) gesetzt.



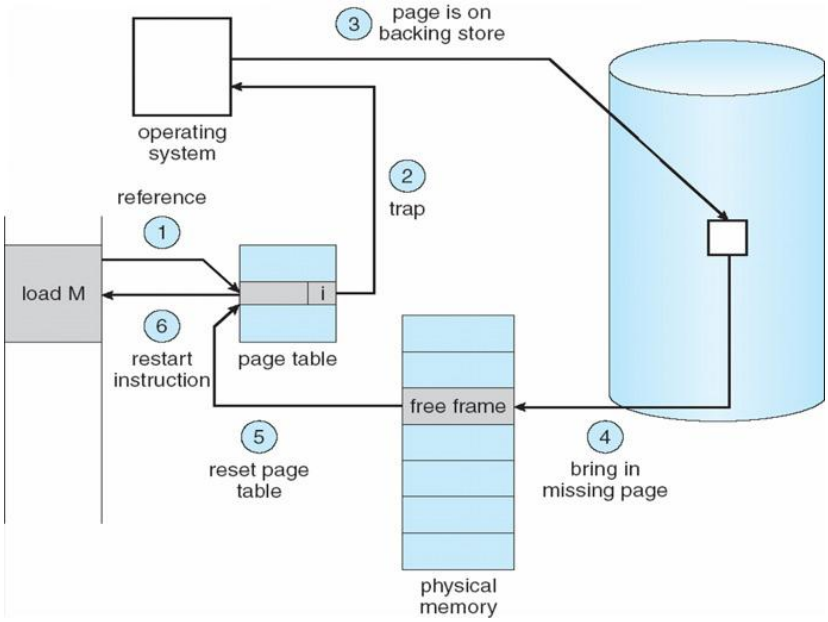
Man sieht hier, dass die Pager Table Buch darüber führt, welche Pages bereits im Memory sind (4, 6, 9) und diese mit „v“ kennzeichnet.

Was passiert aber, wenn ein Prozess auf eine Page zugreifen will, die nicht ins Memory gebracht wurde? Zugriff auf eine Page, die als invalid markiert ist, erzeugt einen **Page Fault**. Dann liegt es am OS, diesen Fehler zu überprüfen und diesen Page Fault angemessen zu verarbeiten. Dies geschieht diesem Protokoll folgend:

1. Wir überprüfen in einer internen Tabelle (normalerweise im Process Control Block PCB gehalten), ob der Aufruf nach dieser Page überhaupt zulässig war oder auf einen „verbotenen“ Speicherbereich zugegriffen werden wollte
2. Wenn die Referenz nicht zulässig war, wird der Prozess terminiert. Falls es zulässig war, aber die Page eben noch nicht im Memory war, pagen wir sie jetzt (siehe nachfolgende Punkte) rein.
3. Wir suchen & finden ein freies Frame
4. Wir führen eine Disk-Operation aus, damit die gewünschte Page in das gefundene Frame reingelesen wird

5. Wenn das Lesen der Disk beendet ist, modifizieren wir die interne Tabelle (im PCB) und die Page Table um festzuhalten, dass die Page nun im Memory ist.
6. Wir starten die Instruktion, welche den Page Fault ausgelöst hat, erneut. Der Prozess kann nun auf die Page zugreifen, als ob nie etwas gewesen wäre.

Hier ein grafischer Ablauf dieses Protokolls:



Performance of Demand Paging

Man kann nun erneut die **effective access time** berechnen.

Effective access time = (1 – p) x memory access (Geschwindigkeit in Nanosekunden) + p x page fault time

Page Fault Rate $0 \leq p \leq 1.0$
Wenn p = 0, keine Page Faults
Wenn p = 1, jede Referenz ist fault

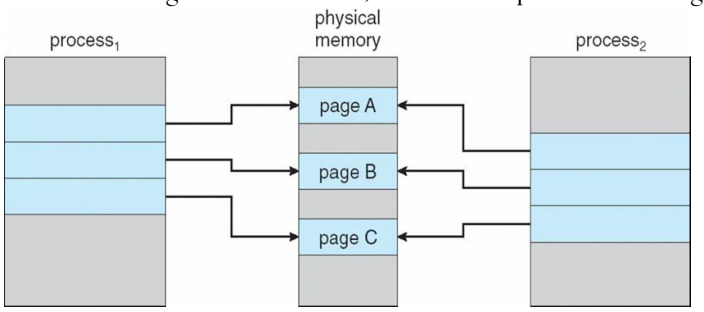
Beispiel:
Memory Access Time = 200 Nanosekunden
Durchschnittliche Page-Fault Service Time = 8 Millisekunden

EAT = (1 – p) x 200 + p (8 Millisekunden)
= (1 – p) x 200 + p x 8‘000‘000 // -200p
= 200 + p x 7‘999‘800

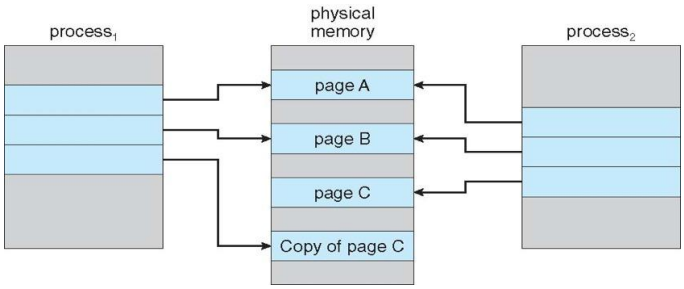
Wenn nun 1 Zugriff von 1000 einen Pauge Fault erzeugt, ist die EAT 8.2 Mikrosekunden
(200 + 0.0001 x 7‘999‘800 = 8199.8 Nanosekunden)

Copy on Write

Diese Technik erlaubt es Parents und Children Prozessen die gleichen Pages im Memory zu teilen. Wenn ein Prozess eine Page verändern muss, wird eine Kopie für ihn erzeugt auf dieser er dann arbeitet:



Hier teilen sich zwei Prozesse gewisse Pages



Und hier wurde eine Kopie von Page C erzeugt, damit Prozess P1 auf dieser Daten verändern kann.

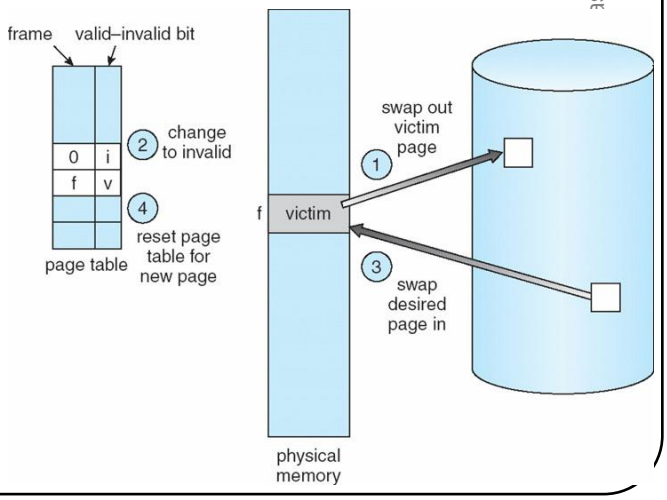
Page Replacement

Was passiert, wenn es kein freies Frame gibt? Dann muss man eine Page im Memory finden, die nicht gerade aktiv verwenden wird und diese raus-swappen. Die Wahl einer geeigneten Page wird vom einem Page Replacement Algorithmus getroffen. Der Algorithmus sollte natürlich eine gewisse Performance bieten, so dass möglichst wenige Page Faults erzeugt werden. Wenn nun eine Page aus dem Memory geswapped wird, muss diese im virtuellen Speicher abgelegt werden und die neue Page, die an den freigewordenen Platz kommt, muss reingeswapped werden. Nun wäre es ideal, dass man vielleicht eine Page rausswapped, die noch gar nicht bearbeitet wurde und somit ja eine identische Kopie davon bereits im virtuellen Speicher ist. Somit kann man sich das rausswappen sparen und die Page einfach löschen. Um solche Pages zu kennzeichnen, kann man ein **modify (dirty) bit** verwenden. Es kennzeichnet Pages, die bereits bearbeitet wurden und somit nicht mehr identisch sind zu ihrer Kopie im virtuellen Speicher. Solche Pages müssen dann gezwungenermassen auf die Disk kopiert werden.

Basic Page Replacement

Wenn kein Frame frei ist, suchen wir eines das gerade nicht verwendet wird und machen es frei. Freimachen kann man ein Frame in dem man seinen Inhalt in einen Swap Space schreibt und danach die Page Tables updatet, um anzuzeigen, dass diese Page nicht länger mehr im Memory ist. Nun kann man die Page-Fault Routine um die Page Replacement Idee erweitern:

1. Finde den Ort der gewünschten Page auf der Disk
2. Finde ein freies Frame
   a. Wenn es eins hat, benutz es
   b. Wenn es keins hat, benutze einen page-replacement Algorithmus um ein **victim frame** (Opferframe) zu bestimmen
   c. Schreibe das victim frame auf die Disk und passe die Page und Frame Tables an.
3. Lese die gewünschte Page in das neue freie Frame und passe die Page und Frame Tables an
4. Starte den Prozess neu



Page Replacement Algorithmen
Nun untersuchen wir einige Page Replacement Algorithmen. Hauptkriterium ist eine tiefe page-fault Rate. Die Algorithmen evaluieren wir in dem wir einen String an Memory Referenzen (reference string) verwenden und schauen, wieviele Page Faults dabei entstehen.

First-In-First-Out (FIFO) Algorithmus
Dies ist der einfachste Algorithmus. Der Algorithmus merkt sich zu jeder Page, wann diese ins Memory geholt wurde und kickt die älteste Page raus, wenn Platz gemacht werden muss.

Schauen wir uns dies anhand eines Beispiels an:

Der Referenzstring soll in diesem Beispiel **1,2,3,4,1,2,5,1,2,3,4,5** sein

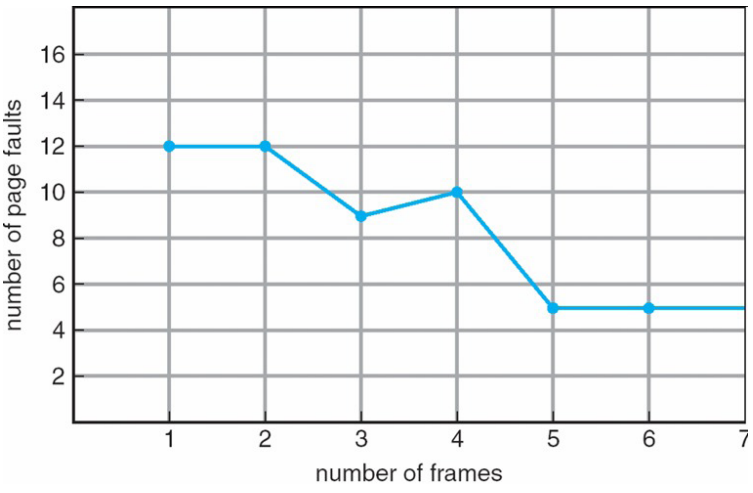
Nun stellen wir uns vor, dass es 3 Frames gibt, die zu Beginn noch leer sind. Dadurch entstehen ingesamt 9 Page Faults!

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

3 Faults entstehen schon, um die ersten 3 Referenzen reinzuladen (1,2,3). Nun will man laut String auf die Referenz 4 zugreifen. Da die nicht drin ist, muss Referenz 1 raus und 4 rückt nach. Das ergibt wieder einen Page Fault. So kann man nun den ganzen String durchgehen und sich zbsp mit einer Strichliste merken, wie viele Page Faults entstehen.

Aufgabe: Wieviele Page Faults gibt es beim String 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 und einem 3er-Frame?[1]

Wenn nun 4 Frames vorhanden sind und man wieder den String **1,2,3,4,1,2,5,1,2,3,4,5** verwendet, wird man feststellen, dass es zu 10 Page Faults kommen wird. Diese Anomalie wird auch **Belady's Anomaly** genannt.



[1] Lösung: 15

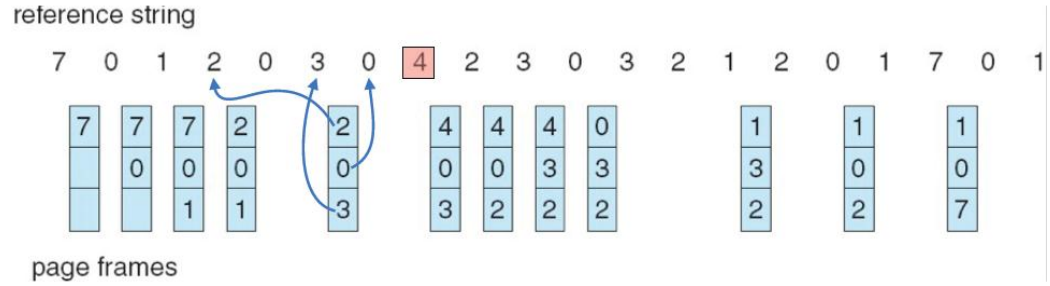
Optimal Page Replacement
Es gibt einen optimalen Algorithmus, der die tiefste Page-Fault Rate hat, und manchmal auch OPT oder MIN genannt wird. Er ist sehr einfach:
Ersetze jene Page, die am längsten nicht mehr verwendet wird.

Nimmt man wieder den String 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 und setzt dies um, wird man feststellen, dass 9 Page Faults erzeugt werden (im Gegensatz zu 15 vorher).
3 gibt's fürs Füllen der leeren Felder. Referenz 2 wird mit 7 ersetzt, da 7 erst sehr spät (18.Stelle) wieder gebraucht wird. Referenz 3 wird mit Page 1 ersetzt, weil 1 von allen drei Pages als letztes wieder aufgerufen wird.
Der Algorithmus ist aber nur soweit gut, wie man auch den Referenzstring kennt.

LRU Page Replacement
Anstatt wie bei den vorherigen Vorgehensweisen nur in die Vergangenheit bzw. Zukunft zu schauen, kann man versuchen auch en Mittelmass zu finden. Zbsp in dem man jene Page ersetzt, die für die längste Periode nicht genutzt wurde. Dies nennt man **least-recently-used (LRU) algorithm**.

Der LRU Algorithmus produziert für den String 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 zwar mehr Faults als der OPT aber weniger als der FIFO: nämlich 12 Page Faults.

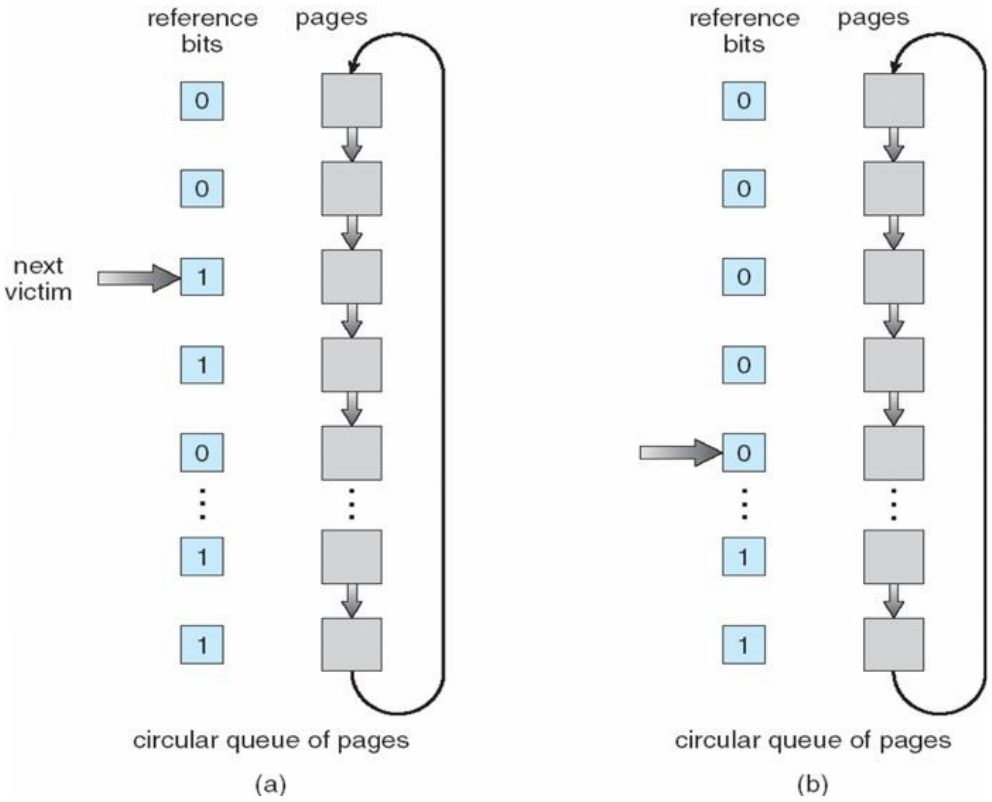
Das sieht dann so aus:



Um die Übersicht behalten zu können, kann man auch einen Stack verwenden, in dem man einen Stack mit allen Page Nummern hat. Wenn nun eine Page referenziert wird, nimmt man diese aus dem Stack raus und legt sie wieder zuoberst ab. So ist die neueste-verwendete Page zuoberst und die der am längsten nicht gebrauchte Eintrag befindet sich immer am Boden des Stacks.

LRU-Approximation Page Replacement
Wir führen ein Reference Bit ein, das jede Page markiert, die mindestens einmal referenziert wurde. Zu Beginn sind alle Pages auf das Referenzbit 0 gestellt. Nach einiger Zeit sieht man aber, welche Pages noch nie verwendet wurden (jene die immer noch 0 als Referenzbit haben). Wir wählen also einer dieser Pages aus um ein Frame frei zu machen.

Second-Chance Algorithm
Die Basis bei diesem Algorithmus bietet der FIFO Replacement Algorithmus. Wir wählen eine Page, die wir gerne rausschmeissen möchten und schauen ihr Referenzbit an. Wenn dieses auf 0 ist, wird die Page rausgenommen. Wenn Sie auf 1 ist, geben wir der Page eine zweite Chance in dem wir sie im Memory lassen, jedoch setzen wir ihr Referenzbit auf 0. Wir gehen dann zur nächsten Page und gehen gleich vor. Falls es keine 0-Referenzbit Page gibt, wird eine der Pages, die eine zweite Chance bekommen hat, dann dran glauben müssen und wird rausgeworfen.



Counting-Based Page Replacement

Least frequently used (LFU) Algorithm
Wir können auch einen Zähler verwenden, der schaut, wie oft eine Page referenziert wurde. Daraus leiten wir dann den **least frequently used (LFU) Algorithmus** ab, also wird jene Page ersetzt, die am wenigstens oft aufgerufen wurde. So kann man sicherstellen, dass eine Page, die oft verwendet wird und somit einen hohen Zähler hat, nicht rausgeworfen wird. Es könnte aber auch sein, dass eine Page zbsp nur zu Beginn sehr oft verwendet wird und danach nicht mehr, der Zähler aber trotzdem sehr hoch ist. Dies kann man berücksichtigen, in dem man den Zähler in regelmässigen Abständen um 1 verringert.

Most frequently used (MFU) Algorithm
Hier hat man die Ansicht, dass jene Page mit dem kleinsten Zähler wohl gerade erst ins Memory geholt wurde und noch gar nicht gross verwendet werden konnte.