Betriebssysteme VO

Zusammenfassung 2020

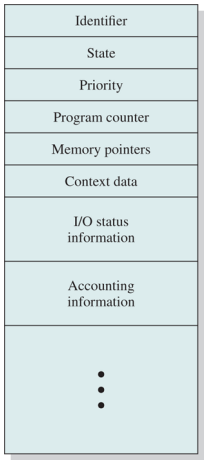
Quelle:

Stallings, W. (2018). *Operating systems: Internals and design principles*. Harlow: Pearson Education Limited.

Prozesse

Definition

Ein Prozess ist ein Programm in der Ausführung. Während ein Programm an sich statisch ist (d.h. der Code der geschrieben wurde) ist ein Prozess dynamisch, da sich dieser von "einem Punkt zum anderen bewegt". Die zwei Grundelemente des Prozesses sind der **Programmcode** und ein damit assoziierter **Datensatz**. Zu jedem Zeitpunkt während der Programmausführung kann der Prozess anhand einiger Elemente, die als **Process Control Block (PCB)** bezeichnet werden, charakterisiert werden. Dementsprechend besteht ein Prozess aus Programmcode, assoziierten Daten und einem PCB. Unterscheidet sich der PCB je nach Implementierung und System sind üblicherweise zumindest Daten zur **Identifizierung**, **Zustand** und **Prozesskontrolle** zu finden.

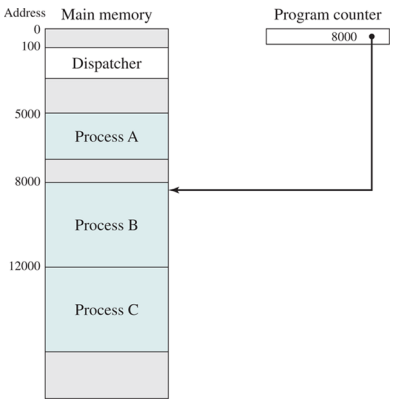


Prozesszustände

Trace

Das Verhalten eines einzelnen Prozesses kann mittels einer Auflistung der Instruktionssequenz dargestellt werden. Hierbei spricht man von einem **Trace**. Durch mehrere Traces kann das Verhalten der CPU und das Zusammenspiel der verschiedenen Prozesse dargestellt werden.

Als Beispiel werden drei Prozesse (A, B und C) hergenommen sowie ein *Dispatcher* der sich um den Prozesswechsel kümmert.



Darüberhinaus besitzen die drei Prozesse folgende *Traces*:

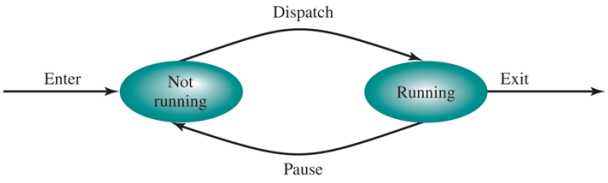
Trace A	**Trace B**	**Trace C**
5000	8000	12000
5001	8001	12001
5002	8002	12002
....	8003	....
5011	*I/O Operation*	12011

Wird beispielsweise angenommen, dass das OS maximal 6 *instruction cycles* zulässt (um eine Monopolisierung der Prozessorleistung zu verhindern) ergibt sich folgendes Bild:

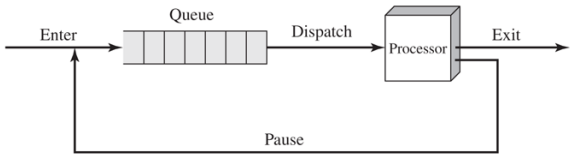
1	5000	27	12004
2	5001	28	12005
3	5002	----------	Time-out
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------	Time-out	32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----------	Time-out
16	8003	41	100
----------	I/O request	42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		----------	Time-out

Nach jeweils 6 *instruction cycles* wird der aktuelle Prozess unterbrochen und der *Dispatcher* übernimmt die Kontrolle. Bei Prozess B wird aufgrund des I/O requests, welcher tendenziell sehr lange dauert, ebenfalls ein Wechsel vorgenommen.

Im einfachsten Modell kann man davon ausgehen, dass ein Prozess entweder ausgeführt wird oder nicht. Die Prozesse die zur Ausführung in Frage kommen müssen in irgendeiner Form gespeichert werden. Die PCBs werden hierfür in einer Queue gespeichert in der sie auf ihre Ausführung warten.



(a) State transition diagram



(b) Queueing diagram

Prozesserzeugung

Wird ein neuer Prozess zu den aktuell verwalteten Prozessen hinzugefügt baut das OS die Datenstrukturen auf und alloziert den benötigten Speicher. Vier Gründe sind üblicherweise für die Prozesserzeugung verantwortlich:

- Login eines interaktiven Benutzers
- Ausführung eines Services durch das Betriebssystem
- Ein Prozess wird durch einen übergeordneten Prozess erzeugt (*process spawning*). Im Normalfall müssen *parent process* und *child process* miteinander kommunizieren können.
- Absetzen eines neuen Jobs

Prozessbeendigung

Jeder Computer muss einem Prozess die Möglichkeit geben eine Beendigung anzeigen zu können. Abgesehen von einer gewollten Beendigung eines Prozesses kann dieser auch aufgrund von Fehlern abgebrochen werden. Unter anderem gehören hierzu:

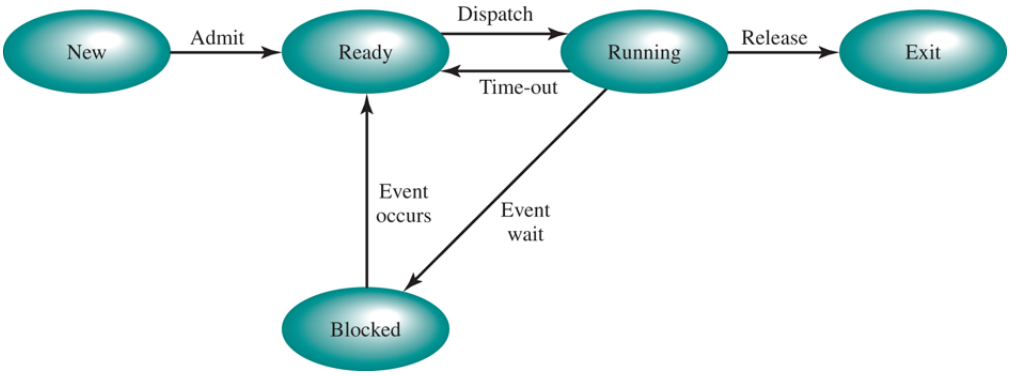
- Logout durch den Benutzer
- Service Request an das OS
- *Halt*-Instruktion eines Jobs
- Auftreten eines Fehlers (*Arithmetic error, protection error, I/O failure...*)

- Beendigung durch einen *parent* Prozess

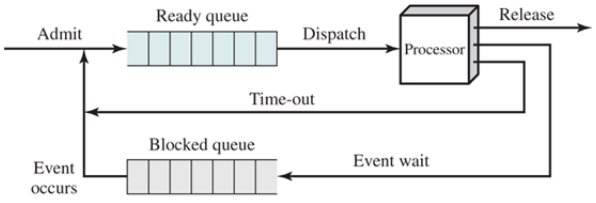
Prozesszustände

Prozesszustand	Beschreibung
RUNNING	Prozess wird aktuell ausgeführt und ist im Besitz der CPU.
READY	Prozess ist bereit zur Ausführung und wartet auf die CPU.
BLOCKED (WAITING)	Prozess ist noch nicht laufbereit und wartet auf ein bestimmtes Ereignis (bspw. Beendigung einer I/O Operation).
NEW	Prozess (d.h. PCB) wurde soeben kreiert ist aber noch nicht zur Ausführung bereit. Im Normalfall ist ein solcher Prozess noch nicht im Hauptspeicher. Die Gründe weswegen ein Prozess in diesem Zustand verweilt können bspw. bei der Speicherkapazität oder der Systemperformance liegen.
EXIT	Prozess wurde beendet oder abgebrochen und ist nicht mehr zur Ausführung auswählbar. Informationen die mit dem Prozess in Verbindung stehen werden temporär erhalten falls diese für andere Programme (*accounting programs*, *utility programs*, ...) notwendig sind.

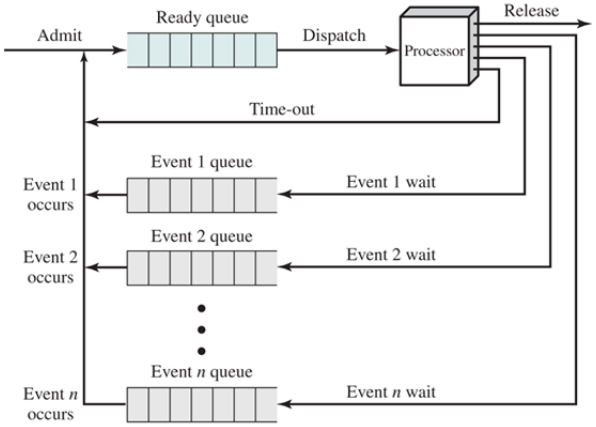
Diese Prozesszustände können im **Five-State Prozessmodell** dargestellt werden:



Anders dargestellt ergibt sich ein **Queuing Modell** in dem für jedes Event eine Queue bereitgestellt wird in der Prozesse je nach Zustand gespeichert werden können.



(a) Single blocked queue



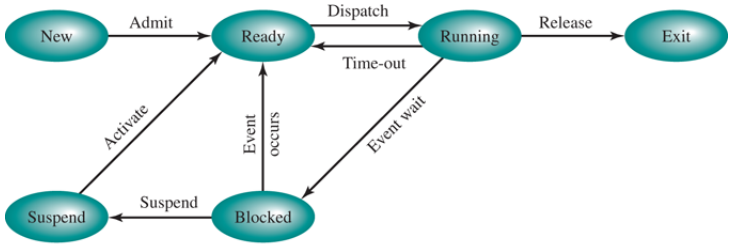
(b) Multiple blocked queues

Swapping

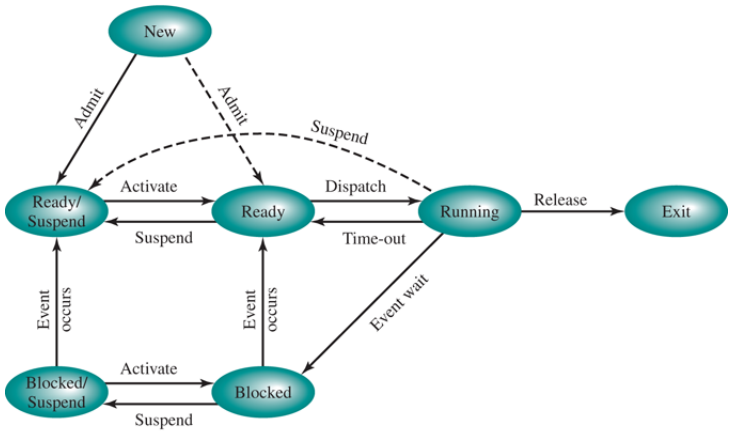
Da der Hauptspeicher nicht unlimitiert groß ist kann es durch zuviele Prozesse sehr schnell zu einem Performanceverlust kommen. Der Ansatz die Speichergröße zu erhöhen führt nicht sehr weit, da zwar über die Jahre Speicherkosten dramatisch gefallen sind wodurch sich der Hauptspeicher vergrößert hat, jedoch damit auch die Prozessgröße.

Eine andere Lösung ist das **Swapping**. Hierbei wird ein teilweiser oder gesamter Prozess vom Hauptspeicher in den Sekundärspeicher gelegt. Ist dann keiner der Prozesse im Hauptspeicher *ready* wird der Prozess mit einem aus dem Sekundärspeicher getauscht und in eine *suspende queue* gelegt. Zu Beachten ist jedoch, dass Swapping eine I/O Operation ist wodurch viel Zeit verloren geht und die Möglichkeit besteht die Performance noch weiter zu verschlechtern.

Zur Umsetzung des Swappings ist ein neuer Prozesszustand, der *suspend* Zustand, notwendig. Sind also alle Prozesse im Hauptspeicher *blocked* kann einer der Prozesse vom OS auf *suspend* gestellt und auf den Sekundärspeicher übertragen werden. Der freigewordene Platz wird danach entweder mit einem Prozess aus der *suspend queue* oder mit einem neu erstellten Prozess gefüllt.



(a) With one Suspend state



(b) With two Suspend states

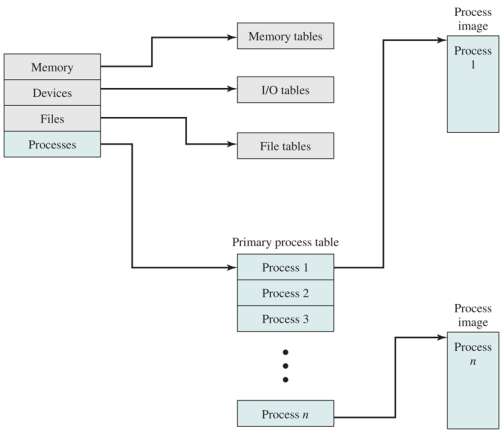
Die Ursachen für einen *suspend* Zustand gehen jedoch über das Swapping hinaus. Weitere Gründe sind:

- OS kann einen Hintergrundprozess bzw. einen problembehafteten Prozess suspenden
- Ein interaktiver Request (bspw. Debugging)
- Ein periodisch ausgelöster Prozess kann während der Wartezeit suspended werden
- Ein *parent* Prozess kann aus verschiedenen Gründen (Untersuchung, Modifikation oder Koordination) den *child* Prozess suspenden.

Kontrollstrukturen für die Prozessverwaltung

Im OS werden verschiedene Tabellen für Prozesse und Ressourcen verwaltet. Diese haben zwar unterschiedliche Aufgaben müssen jedoch in irgendeiner Weise von und zueinander referenziert werden. Die vier Tabellen sind:

Tabelle	Beschreibung
Memory Table	Werden dazu genutzt um sowohl Haupt als auch Sekundärspeicher zu verfolgen. Zwingende Informationen die in einer solchen Table gespeichert werden sind die Allokation von Haupt- bzw. Sekundärspeicher an Prozesse, welche Prozesse auf welche Speicherbereiche zugreifen dürfen sowie jegliche Information die notwendig für die Verwaltung vom virtuellen Speicher ist.
I/O Table	Wir dazu genutzt die aktuellen I/O Geräte (sowie deren Status) und Kanäle zu managen.
File Table	Informationen über die Existenz von Dateien und deren Position bzw. Status im Speicher.
Process Table	Notwendig für das Management von Prozessen.



Process Image

Ein Prozess lässt sich demnach beschreiben als die Kombination von einem Programm, Daten, dem Stack und Attributen. Diese Sammlung wird auch als **process image** bezeichnet. Das *process image* ist eine ausführbare Datei die zur Programmausführung benötigt wird und befindet sich grundsätzlich im Sekundärspeicher. Sie muss zur Ausführung in den Hauptspeicher (oder zumindest in den virtuellen Speicher) geladen werden. Der Unterschied zwischen einem Prozess und dem dazugehörigen Process Image ist das letzterer **read-only** ist, da ansonsten bei Prozessveränderungen unerwünschte Nebeneffekte auftreten.

Typischerweise besteht ein *process image* aus

- **User Data** - modifizierbarer Bereich des User Space wie Daten bzw. modifizierbare Programme
- **User Program** - das auszuführende Programm
- **Systemstack** - Die mit dem Prozess assoziierten Stacks, Paramter und Calling Adressen von System Calls
- **Process Control Block** - im Execution Context bestehend aus
    - **Process Identification**
        - Eindeutige Prozessnummer (*process identifier*)
        - Benutzerkennung *(user identifier)*
        - *parent process identifier*
    - **Processor State Information**
        - Registerinhalte
        - Kontroll- und Statusregister
        - Stack Pointer
    - **Process Control Information**
        - Scheduling und Zustandsinformation des Prozesses
        - Querverweise auf andere Prozesse

Der **process control block** ist die wichtigste Datenstruktur in einem OS da in jedem **PCB** alle Informationen enthalten sind die ein OS über einen Prozess benötigt.

Execution Modes

Die meisten Prozessoren unterstützen zumindest zwei Modi, *privileged (bzw. kernel oder system) mode* und *user mode* zur Ausführung von Prozessen. Diese dienen dem Schutz der Datenstrukturen des OS. Gewisse Instruktionen (oder Speicherzugriffe), wie beispielsweise Prozesse die auf Kontrollregister zugreifen, sind nur im *privileged mode* ausführbar.

Der Name *kernel mode* weist bereits auf den *kernel* des OS hin, welcher den Teil des OS mit den wichtigsten Systemfunktionen darstellt. Diese betreffen vor allem:

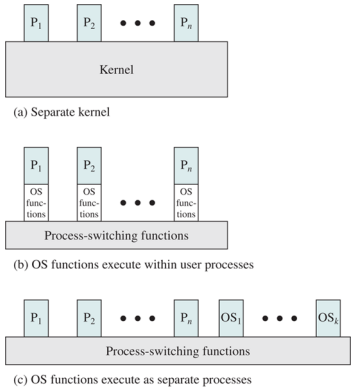
- **Prozessmanagement** - Prozesse kreieren, beenden, switchen, synchronisieren,...
- **Speichermanagement** - Speicherallokation, Swapping
- **I/O Management** - Buffer, Allokation von I/O Kanälen und Geräten
- **Supportfunktionen** - Accounting, Monitoring

OS und Prozesse

Da das OS selbst lediglich eine Sammlung von Programmen ist gibt sich die Frage auf, ob das OS selbst auch ein Prozess ist. Hierfür gibt es verschiedene Designansätze. Beispielsweise wurde früher der Kernel von allen Prozessen strikt getrennt und besaß seinen eigenen Speicherbereich bzw Stack (**Nonprocess Kernel**). Der Prozessbegriff existiert hier nur für Benutzerprogramme und der Prozesskontext wird bei OS-Aktivität verlassen.

Bei der **OS-Ausführung in User-Prozessen** ist das OS eine Sammlung von Routinen, die vom Benutzerprogramm aufgerufen werden. Fast alle OS-Routinen laufen im Prozesskontext und dieser wird nur beim Process Switch verlassen.

Das **Prozessbasierte OS** ist eine Sammlung von Systemprozessen bei der lediglich Basisservices (Process Switching, Interrupts, ...) nicht als Prozesse realisiert sind.



Threads

Bisher repräsentierte der Prozess zwei voneinander unabhängige Charakteristiken: die Ressourcenverwaltung und das Dispatching. Teilt man diese zwei Konzepte auf erhält man **Tasks** (Ressourcenverwaltung) und **Threads** (Dispatching). Ein Thread ist jener Teil eines Prozess der unabhängig von anderen Prozessteilen ausführbar ist. Dementsprechend werden Prozesse in Threads aufgeteilt. Da sich alle Threads dieselben Variablen und einen gemeinsamen Prozessadressraum teilen ist die Threaderzeugung um einiges schneller als ein *process switch*. Die Kommunikation zwischen den Threads funktioniert auch ohne Einschaltung des Kernels, daher muss die Synchronisation der Daten vom Programmierer gewährleistet werden.

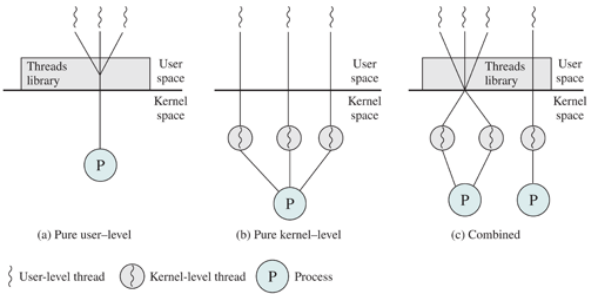
Ein Thread kann ähnlich wie ein Prozess **blocked, ready** oder **running** sein. Ein Suspendieren macht wenig sinn, da ohnehin keine Ressourcen freigegeben werden. Zudem hat jeder Thread einen eigenen **User Stack** und einen **Kernel Stack**. Ein Thread terminiert sobald der dazughörige Prozess terminiert.

Es wird zwischen zwei verschiedenen Thread-Arten unterschieden:

User-Level Threads (ULT) - Diese sind für den Kernel unsichtbar und vollständig auf Benutzerebene implementiert. Das Thread Management geschieht mittels Thread Libraries in denen Code für Erzeugung und Terminierung von Threads, Scheduling usw. zu finden ist. Der große Vorteil bei **ULT** ist, dass kein Einschalten des Kernels notwendig ist und das Scheduling sehr genau auf die Applikation selbst angepasst werden kann. Der Nachteil hier ist, dass die Threads nicht auf mehrere Prozessoren aufgeteilt werden können.

Kernel-Level Threads (KLT) - Das Thread Management wird vom Kernel übernommen wodurch ein Zugriff auf mehrere Prozessoren ermöglicht wird. Der Nachteil bei diesen Threads ist der größere Zeitverlust beim *thread-switching*, da hier zwei *mode switches* notwendig sind.

Es ist jedoch möglich einen hybriden ULT/KLT Ansatz zu wählen um die Vorteile von beiden Methoden zu nutzen.



Wiederholungsfragen

Was versteht man unter einem Process Image? Erklären sie woraus ein Process Image besteht.

Als Process Image bezeichnet man den Prozess im Computersystem der aus

- Programm,
- Daten,
- Stack und
- PCB

  besteht und zur Ausführung in den Hauptspeicher geladen wird.

Was versteht man unter einem Process Control Block? Beschreiben Sie, aus welchen Teilen der PCB besteht und welche Informationen in diesen Teilen jeweils verwaltet werden.

Der PCB ist Teil des Process Images (für den Execution Context) und besteht aus:

- **Process Identification**
  - Eindeutige Prozessnummer (*process identifier*)
  - Benutzerkennung *(user identifier)*
  - *parent process identifier*
- **Processor State Information**
  - Registerinhalte
  - Kontroll- und Statusregister
  - Stack Pointer
- **Process Control Information**
  - Scheduling und Zustandsinformation des Prozesses
  - Querverweise auf andere Prozesse

Worin liegt der grundlegende Unterschied zwischen Prozessen und Threads? Welcher Vorteil ergibt sich aus der Einführung von Threads für den Benutzer und worauf muss der Benutzer achten?

Während ein Prozess die Programmausführung beschreibt und im Normallfall einen eigenen Speicherbereich zur Verfügung hat ist ein Thread **Teil eines Prozesses** und mehrere parallel laufende Threads untereinander **teilen sich denselben Speicherplatz**.

Unterschieden werden kann zwischen **Ressourcenverwaltung** (Prozess, Task) und **Dispatching** (Thread). Unterschieden wird zwischen **User Level Threads** die für den Kernel unsichtbar sind (und daher nicht auf mehreren Prozessoren laufen können) und **Kernel Level Threads** die vom Kernel verwaltet werden (daher auf mehreren Prozessoren laufen) aber langsamer sind. Ein großer Vorteil von Threads ist das leichtere Erzeugen, Terminieren und Umschalten zwischen den Threads jedoch muss der Benutzer bei **ULT** auf eine Synchronisation achten.

Einsatzbereiche für Threads sind Anwendungen die unterschiedliche Aufgaben mit denselben Daten haben. Als Beispiel dient ein Spreadsheet Programm bei dem sich ein Thread um die Menüanzeige kümmert und ein anderer die Berechnungen durchführt.

Erklären Sie die Aktionen, die vom Betriebssystem bei einem Process Switch durchzuführen sind. Welche Arten von Ereignissen können zu einem Process Switch führen?

1. Umschalten des aktiven Prozesses (Speicherung des PCB, Register, Stack um Fortsetzen des Prozesses zu ermöglichen)
2. Alten Prozess entweder auf Blocked-Queue setzen oder terminieren
3. Neuen Prozess aus Ready-Queue holen
4. Neuen Prozess der CPU zuteilen

Ereignisse die zu einem Process Switch führen können:

- **Supervisor Call** - OS wird aktiviert aufgrund eines expliziten Aufrufs durchdas Programm (bspw. I/O Operationsaufruf resultiert in einer Routine des OS)
- **Trap** - Tritt ein Fehler oder eine Exception auf bestimmt das OS mittels Trap ob der Prozess beendet werden muss (je nach Fehler)
- **Interrupt** - Ursache außerhalb des Prozesses, Kontrolle wird an Interrupt Handler abgegeben

Wodurch unterscheiden sich die Prozesszustände READY und BLOCKED?

Im **BLOCKED** Zustand wartet ein Prozess auf das Eintreten eines Events bevor dieser einsetzbar ist. Ist der Prozess im Zustand **READY** kann dieser ausgeführt werden und wartet darauf der CPU zugewiesen zu werden.

Was versteht man unter einem Kernel Level Thread und unter einem User Level Thread? Beschreiben Sie die beiden Arten der Threadimplementierung und charakterisieren Sie deren Unterschiede.

User Level Threads - Diese Threads sind für den Kernel unsichtbar und deren Management wird über eine **Thread Library** vorgenommen, die dem Programmierer eine API zur Erzeugung und zum Management von Threads zur Verfügung stellt.

Der Vorteil von **ULT** ist ein einfaches switchen unter den Threads (kein Mode Switch notwendig). Der Nachteil ist, dass die **ULT** nicht auf mehrere Prozessoren verteilt werden können und, dass ein System Call alle Threads blockiert.

Kernel Level Threads - Hier werden die Threads gänzlich vom Kernel gemanaged und es gibt lediglich eine API zur Kernel Thread Facility.

Vorteil dieses Ansatzes ist das einzelne Threads blockiert werden können und die Möglichkeit Multithreading für Kernelroutinen zu nutzen. Der Nachteil ist, dass ein Switch 2 Mode Switches benötigt und daher deutlich langsamer als ein ULT ist.

Erklären Sie die Begriffe Process Switch und Mode Switch, sowie die Beziehung, in der diese beiden Konzepte stehen. Zählen Sie weiters die drei Klassen von Ereignissen auf, die einen Mode Switch nach sich ziehen.

- **Process Switch** - Sichern des **Kontextes** (Programm-Counter, Prozessor-Register, PSW) des alten Prozesses, ändern seines Zustandes je nach Ursache der Process Switches (in Ready, Blocked, Blocked-Suspend), Einfügen in die jeweilige Queue (Blocked-Queue, Ready-Queue,...). Finden einer Scheduling Entscheidung, d.h. Auswählen eines neuen Prozesses aus der Ready-Queue und seinen Zustand auf Running setzen. Laden des Kontextes des neuen Prozesses in den Prozessor. Hierfür ist ein Mode Switch auf jeden Fall notwendig
- **Mode Switch** - Es gibt zwei Modi zur Ausführung eines Programms: *user mode* und *kernel mode* die unterschiedliche Zugriffsrechte besitzen. Ein Mode Switch ist der Wechsel zwischen diesen beiden Modi. Der Mode Switch stellt die Basis für den Process Switch dar, jedoch bewirkt nicht jeder Mode Switch einen Process Switch.

Mutual Exclusion & Synchronisation

Da es bei parallel laufenden Prozessen zur Situation kommen kann, dass dieselben Daten benötigt werden muss ein Mechanismus dafür sorgen, dass beim Zugriff die Konsistenz der Daten bzw. die geordnete Abarbeitung von Befehlen gesichert werden. Die Daten die von mehreren Prozessen benötigt werden nennt man **kritischer Abschnitt**.

Die zwei voneinander unabhängigen Ziele die erreicht werden wollen sind zum einen der **wechselseitige Ausschluss (mutual exclusion)** der den gleichzeitigen Ressourcenzugriff verhindert und die **Atomizität** einer Aktionsfolge sicherstellt um Daten konsistent zu halten und zum anderen die **Bedingungssynchronisation (condition synchronization)** durch die eine definierte Abfolge von Operationen erreicht werden soll.

Die Kontrollaufgaben des BS umfassen zusätzlich:

- **Verhinderung von Deadlocks** - Die Situation in der Prozesse nicht fortgeführt werden können weil jeder einzelne auf den anderen wartet.
- **Verhinderung von Starvation** - Eine Situation in der ein Prozess der prinzipiell bereit zur Ausführung ist nicht zum Zug kommt und "verhungert"

Softwarebasierte Lösungen

Wartet ein Prozess auf das Eintreten einer Bedingung indem diese durchgehend abgefragt wird spricht man von **busy waiting**.

Dekker Algorithmus

Der Dekker Algorithmus für zwei Prozesse benötigt drei Variablen: zwei `flag` und eine `turn` Variable. Jeder Prozess bekommt genau ein `flag` zugewiesen und wird diese gesetzt bedeutet dies, dass der zugehörige Prozess sich im kritischen Abschnitt befindet. Im weiteren Verlauf entscheidet `turn` über den Eintritt in den kritischen Zustand. Ist in `turn` die Nummer des anderen Prozesses gespeichert, wird das eigene `flag` zurückgesetzt und gewartet bis `turn` die Nummer des eigenen Prozesses enthält.

```
boolean flag [2];
int turn;

void P0() {
      while (true) {
            flag [0] = true;
            while (flag [1]) {
                  if (turn == 1) flag [0] = false;
```

```
                        while (turn == 1) /* do nothing */;
                        flag [0] = true;
                }
        }
        /* critical section */;
  “turn = 1;
        flag [0] = false;
        /* remainder */;
    }
}

void P1() {
    while (true) {
        flag [1] = true;
        while (flag [0]) {
                if (turn == 0) {
                    flag [1] = false;
                    while (turn == 0) /* do nothing */;
                    flag [1] = true;
                }
        }
        /* critical section */;
        turn = 0;
        flag [1] = false;
        /* remainder */;
    }
}

void main ()
{
    flag [0] = false;
    flag [1] = false;
    turn = 1;
    parbegin (P0, P1);
}
```

Peterson Algorithmus

Eine einfacher zu verstehende Alternative zum Dekker Algorithmus die um einiges eleganter ist. Die Variablen sind dieselben vie im Dekker Algorithmus jedoch wird die Abfrage deutlich vereinfacht um gleichzeitige Zugriffe zu verhindern.

```
boolean flag [2];
int turn;

void P0() {
      while (true) {
              flag [0] = true;
              turn = 1;
               while (flag [1] && turn == 1) /* do nothing */;
               /* critical section */
               flag[0] = false;
               /* remainder */
      }
}

void P1() {
      while (true) {
              flag [1] = true;
              turn = 0;
               while (flag [0] && turn == 0) /* do nothing */;
               /* critical section */
               flag[1] = false;
               /* remainder */
      }
}

void main() {
  flag[0] = false;
  flag[1] = false;
  parbegin (P1, P1);
}
```

Hardwarebasierte Lösungen

Interrupt Disabling

In Uniprozessorsystemen können Prozesse keine Überlappungen bei der Ausführung haben. Daher laufen Prozesse so lange, bis sie vom OS unterbrochen werden. Damit *mutual exclusion* garantiert wird reicht es also lediglich die Unterbrechung eines Prozesses zu unterbinden. Die Kosten in Bezug auf Effizienz sind jedoch enorm, und für Multiprozessorarchitekturen funktioniert dieser Ansatz nicht.

```
while (true) {
  /* disable interrupts */
  /* critical section */
  /* enable interrupts */
  /* remainder */
}
```

Besondere (mächtigere) Maschineninstruktionen

Test and Set

Im Prinzip wird eine Variable (bspw. ein Speicherregister) überprüft ob dieses belegt ist oder nicht. Sollte es auf 0 sein wird auf den kritischen Bereich zugegriffen und der Wert der Variable auf 1 gesetzt. Während der Abarbeitung retourniert die funktion `testandset` **false**. `testandset` funktioniert als atomare Prozessorinstruktion und wird dadurch nicht unterbrochen.

exchange

```
void exchange (int *register, int *memory) {
  int temp;
  temp = *memory;
  *memory = *register;
  *register = temp;
}
```

Die *exchange* Instruktion tauscht die Inhalte eines Registers mit denen der Memory Location. Diese XCHG Instruktion wird von Prozessoren mitsamt einer globalen Variable `bolt` genutzt, die mit 0 initialisiert wird. Jedem Prozess wird eine lokale Variable `key` zugewiesen mit dem Wert 1. Ein Prozess darf demnach nur in den kritischen Abschnitt wenn `bolt == 0` und verhindert einen Zugriff von anderen Prozessen indem die Variable für die Dauer der Abarbeitung auf 1 gesetzt wird. Ist also `bolt == 1` befindet sich exakt 1

Prozess im kritischen Abschnitt.

```
/* program mutualexclusion */
int const n = /* number of processes */
int bolt;

void P(int i) {
    while (true) {
      int key_i = 1;
        do exchange (&key_i, &bolt)
        while(key_i != 0);
      /* critical section */
      bolt = 0;
      /* remainder */
    }
  }

void main() {
  bolt = 0;
  parbegin (P(1), P(2), ..., P(n));
}
```

Vorteile von speziellen Maschineninstruktionen sind deren Einfachheit und breite Anwendbarkeit (sowohl Uni- als auch Multiprozessorarchitekturen) jedoch sind ernstzunehmende **Nachteile** wie die Nutzung von *busy waiting*, die Möglichkeit von *deadlocks* als auch *starvation*.

Semaphoren

Semaphoren sind ein Synchronisationsmechanismus der es ermöglicht auf *busy waiting* zu verzichten. Die Datenstruktur eines Semaphors beinhaltet einen Integer `count` auf den lediglich drei (atomare) Operationen zugelassen sind:

- Ein Semaphor kann auf einen nichtnegativen `integer` initialisiert werden.
- `semWait` - *Prozess möchte auf Ressource zugreifen* - Die `semWait` Operation verringert den `count` Wert. Sollte dieser negativ werden, wird der aufrufende Prozess blockiert. (in Literatur P für passeren/proberen)
- `semSignal` - *Prozess ist fertig und benötigt Ressource nicht mehr* - Die `semSignal` Operation erhöht den `count`Wert. Ist der Wert kleiner oder gleich 0, wird ein durch `semWait` blockierter Prozess wieder entblockiert. (in Literatur V für vrijgave/verhogen)

```
struct semaphore {
  int count;
  queueType queue;
};

void semWait(semaphore s) {
  s.count--;
  if (s.count < 0) {
    /* place this process in s.queue */
    /* block this process */
  }
}

void semSignal(semaphore s) {
  s.count++;
  if (s.count <= 0) {
    /* remove a process P from s.queue */
    /* place process P on ready list*/
  }
}
```

Zusätzlich besitzt ein Semaphor eine `queue` in der die Prozesse nacheinander angehängt und nach dem FIFO-Prinzip abgearbeitet werden. Dadurch wird Fairness sichergestellt und, dass die Prozesse auch drankommen.

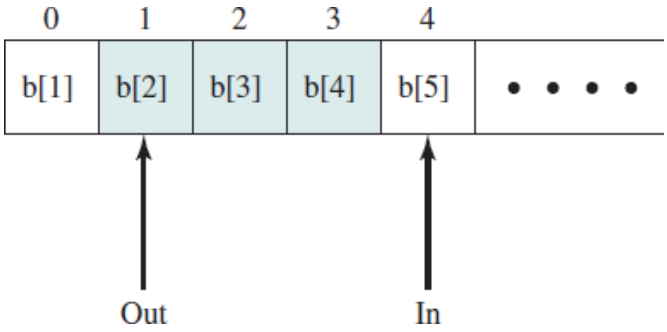
Bei der Implementierung von Semaphoren kommt zusätzlich noch eine `flag` Komponente dazu. Vor dem kritischen Abschnitt wird mittels *busy waiting* die `flag` überprüft. Da `semWait` und `semSignal`sehr kurze Operationen sind kommt es kaum zum *busy waiting* wodurch es in diesem Fall vertretbar ist.

Bedingungssynchronisation

Producer-Consumer Problem

In diesem Szenario geht es um den Datenaustausch zwischen einem **Producer** der Daten generiert sowie einem **Consumer** der Daten auslesen möchte. Jeder Produzent produziert in einem Arbeitsschritt einene Datensatz der in ein Array geschrieben wird und über einen Puffer an den Konsumenten gegeben wird, der ebenfalls in jedem Arbeitsschritt nur einen Datensatz auslesen kann.

Zu Beginn wird von einem unlimitiert großen Puffer `b[1], b[2], ....` ausgegangen. Ein `in` zeigt auf den nächsten freien Index auf den geschrieben werden kann, und ein `out` auf den nächsten nicht-leeren Index.



Die beiden Funktionen können wie folgt definiert werden:

```
producer:
while(true) {
  /* produce item v */
  b[in] = v;
  in++;
}
```

```
consumer:
while(true) {
  while(in <= out) {
    /* do nothing */
  }
  w = b[out];
  out++;
  /* consume item w */
}
```

```
/* hilfsfunktionen */

void append(v) {
  b[in] = v;
  in++;
}

int take() {
  w = b[out];
  out++
  return w;
}
```

Producer-Consumer Implementierung mit Infinite Buffer und Semaphoren

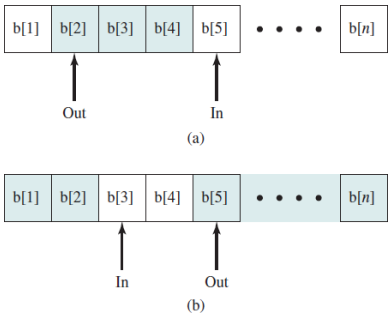
```
/* program producerconsumer */
semaphore n = 0, s = 1;
in = out = 0;

void producer() {
  while(true) {
    produce();
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}

void consumer() {
  while(true) {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    consume();
  }
}

void main() {
  parbegin (producer, consumer);
}
```

Producer-Consumer mit Ringpuffer

Da ein unendlich großer Puffer noch nicht existiert werden die Indizes der Pointer mit `% n` berechnet. Durch diesen Mechanismus lockert sich die "Verzahnung" zwischen dem Producer und dem Consumer.

```
producer:
while(true) {
  /* produce item v */
  while((in++) % n == out) {
    /* do nothing */
  }
  b[in] = v;
  in = (in++) % n;
}
```

```
consumer:
while(true) {
  while(in == out) {
    /* do nothing */
  }
  w = b[out];
  out = (out++) % n;
  /* consume item w */
}
```



Der Unterschied zum unbegrenzten Puffer liegt in der Bedingungssynchronisation. Ein Produzent darf nur dann schreiben, wenn mindestens ein leerer Speicherbereich vorhanden ist. Für diesen Zweck wird ein weiterer Semaphor `e (empty)` hinzugefügt der mit der Buffergröße initialisiert wird.

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffersize */
semaphore s = 1, n = 0, e = sizeofbuffer;
in = out = 0;

void producer() {
  while(true) {
    produce();
```

```
    semWait(e); /*  ist noch Platz im Puffer? */
    semWait(s);
    append();
    semSignal(s);
    semSignal(n);
  }
}

void consumer() {
  while(true) {
    semWait(n);
    semWait(s);
    take();
    semSignal(s);
    semSignal(e); /* neuer Platz frei im Puffer */
    consume();
  }
}

void main() {
  parbegin (producer, consumer);
}
```

Reihenfolge von Semaphorenoperationen

Die `semSignal` Operationen können in beliebiger Reihenfolge positioniert werden. Die Abfolge der `semWait` Operationen ist aber **relevant** und kann Deadlocks auslösen!

Reader-Writer Problem

In diesem Szenario gibt es wieder um einen geteilten Datensatz von mehreren Prozessen. Es gibt *readers* die den Datensatz nur auslesen und *writers* die nur in den Datensatz schreiben.

Gleichzeitige Lesezugriffe von mehreren *readern* auf die Ressource sind zugelassen. Wird jedoch der Datensatz von einem *writer* verändert, muss der Zugriff für alle anderen (auch *reader*) Prozesse blockiert werden. Demnach benötigen **Schreibzugriffe** exklusiven Zugriff auf die Ressource aber **Lesezugriffe** nicht. Der Unterschied zum **Producer-Consumer Problem** besteht darin, dass der *producer* im Gegensatz zum *writer* nicht nur in den Datensatz schreibt sondern auch Queuepointer auslesen muss und darauf achten wohin der nächste Datensatz zu schreiben ist.

Readers have priority

```
int readcount = 0; /* anzahl der reader */
semaphore x = 1, wsem = 1 /* wsem is used to enforce mutual exclusion */

 void reader() {
    while(true) {
      semWait(x);
      readcount++;

      if (readcount == 1) { /* first reader covers mutual exclusion */
        semWait(wsem);
      }

      semSignal(x);
      READUNIT();
      semWait(x);
      readcount--;

      if (readcount == 0) { /* no readers left, writers gonna write */
        semSignal(wsem);
      }

      semSignal(x);
    }
  }

void writer() {
  while(true) {
    semWait(wsem);
    WRITEUNIT();
    semSignal(wsem);
  }
}
[...]
```

Der erste *reader* stellt sicher, dass kein *writer* während des kritischen Abschnitts auf den geteilten Speicher zugreift. Alle nachfolgendenen *reader* können direkt auf den Datensatz zugreifen.

Writers have priority

In der vorherigen Lösung ergibt sich ein Problem durch die Bevorzugung von *readern*. Insofern ein *reader* auf den Datensatz zugreift ist es möglich für andere *reader* die Kontrolle über den Datensatz zu behalten und es kann zur Starvation für *writer* Prozesse kommen.

```
/* program readersandwriters */
int readcount, writecount = 0; /* anzahl der reader und writer */
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1

 void reader() {
    while(true) {
      semWait(z) /* all readers after the first blocked here */
      semWait(rsem) /* first reader gets blocked here */
      semWait(x);
      readcount++;

      if (readcount == 1) { /* first reader covers mutual exclusion */
        semWait(wsem);
      }
      semSignal(x);
      semSignal(rsem);
      semSignal(z);
      READUNIT();
      semWait(x);
      readcount--;

      if (readcount == 0) { /* no readers left, writers gonna write */
        semSignal(wsem);
      }
      semSignal(x);
    }
  }

void writer() {
    while(true) {
      semWait(y)
      writecount++;

      if (writecount == 1) {
        semWait(rsem);
      }
      semSignal(y);
      semWait(wsem);
      WRITEUNIT();
      semSignal(wsem);
      semWait(y);
      writecount--;

      if (writecount == 0) {
        semSignal(rsem);
      }
      semSignal(y);
    }
```

```
    }
  [...]
```

Wird ein Zugriff für einen *writer* benötigt kann dieser die Queue "überspringen" und es kommt somit nicht zu Starvation. Die Idee hier ist, dass im Semaphor `rsem` immer genau ein *reader* in der Queue gespeichert wird, und alle anderen *reader* im Semaphor `z` warten. Dadurch kann die Wartezeit für einen *writer* maximal eine Leseoperation lang dauern, da der kritische Abschnitt hier vom Semaphor `rsem` geschützt wird.

Monitore

Ein Problem mit der Nutzung von Semaphoren kann sein, dass es aufgrund der vielen verteilten `semWait` und `semSignal` Operationen nicht einfach ist die Auswirkung dieser Operationen nachzuvollziehen. Der **Monitor** ist ein Programmiersprachenkonstrukt, bestehend aus **Prozeduren, lokalen Daten und Initialisierungscode**, welches die gleiche Funktionalität wie Semaphoren bietet, jedoch einfacher zu kontrollieren ist. Der Monitor sorgt für *mutual exclusion* und es ist kein explizites Programmieren notwendig. Zu jedem Zeitpunkt kann sich nur ein Prozess im Monitor befinden.

Die Synchronisation wird über einen besonderen Datentyp, den *condition variables*, sichergestellt welche sich im Monitor befinden und nur in diesem sichtbar sind. Zwei Funktionen greifen auf diese Variablen zu:

- `cwait(c)` - Aufrufender Prozess wird blockiert bis Variable `c` den Wert `true` annimmt.
- `csignal(c)`- Prozess, der auf Bedingung `c` wartet kann fortgesetzt werden. Sollte kein Prozess warten, passiert nichts.

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];  /* space for N items */
int nextin, nextout;  /* buffer pointers */
int count;  /* number of items in buffer */
cond notfull, notempty; /* condition variables for synchronization */

void append (char x) {
    if (count == N) { /* buffer is full; avoid overflow */
      cwait(notfull);
    }

    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++; /* one more item in buffer */
    csignal (notempty); /*resume any waiting consumer */
}
void take (char x) {

    if (count == 0) { /* buffer is empty; avoid underflow */
      cwait(notempty);
    }

    x = buffer[nextout];
    nextout = (nextout + 1) % N);
    count--;  /* one fewer item in buffer */
    csignal (notfull);  /* resume any waiting producer */
}



{   /* monitor body */
    nextin = 0; nextout = 0; count = 0; /* buffer initially empty */
}
```

Message Passing

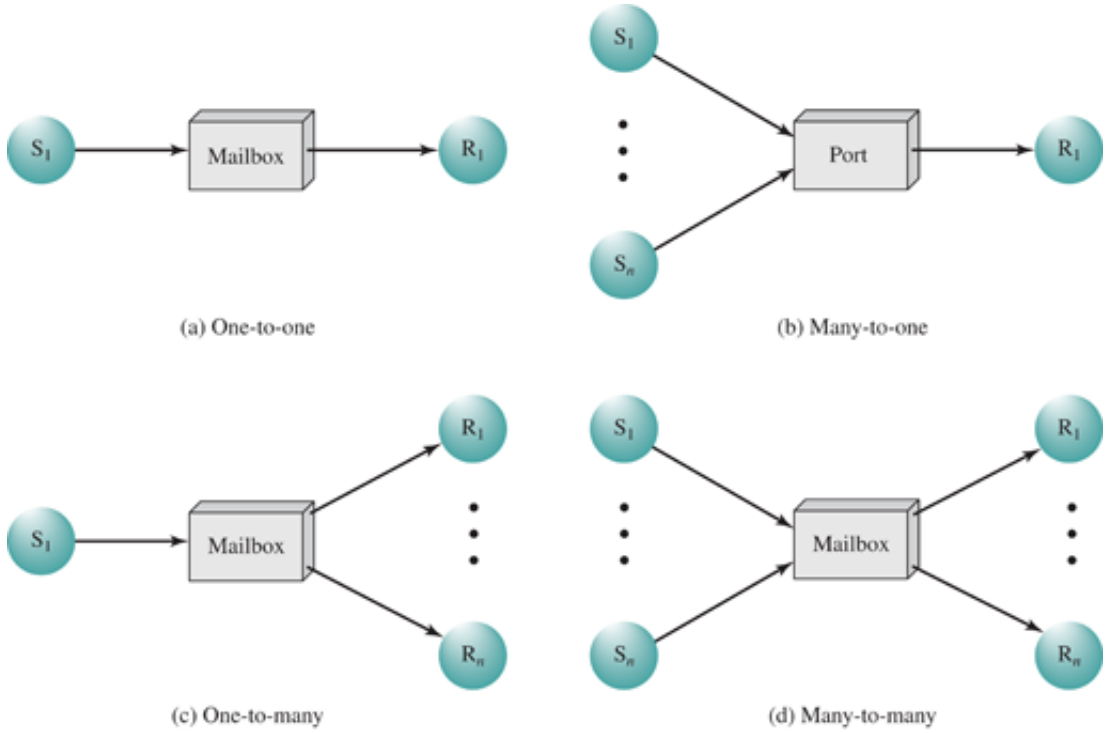
Mit Message Passing kann die Kommunikation zwischen Prozessen in Multi- und Uniprozessorsystemen aber auch in verteilten Systemen sichergestellt werden. Eine *message* in diesem Kontext ist eine atomare Datenstruktur. Die Implementierungen dieses Systems sind vielseitig, beinhalten jedoch zumindest die primitiven Funktionen zum Senden und Empfangen von nachrichten:

- `send (destination, message)`
- `receive (source, message)`

Die Synchronisation wird über die Attribute *blocking* und *non-blocking* sichergestellt. Bei der Kombination **blocking send, blocking receive** sind sowohl Sender als auch Empfänger blockiert bis die Message zugestellt ist. Diese Konstellation wird auch als Rendezvous bezeichnet und ist üblicherweise für Prozesse die sehr eng synchronisiert werden müssen gedacht. Sind die Operationen *non-blocking* können andere Prozesse unabhängig vom Versand und Erhalt weitergeführt werden.

- **Blocking send, blocking receive** - Absender und Empfänger sind blockiert bis die Nachricht ankommt (*Rendezvous*). Geeignet für enge Prozesssynchronisation.
- **Nonblocking send, blocking receive** - Absender kann mit Prozessabarbeitung fortfahren, jedoch ist der Empfänger bis zum Empfang blockiert. Dies ist die nützlichste Kombination da sie einem Prozess erlaubt eine Nachricht an mehrere Ziele gleichzeitig zu versenden und ein Prozess der eine Nachricht erhalten muss bevor dieser fortfahren kann ist so lange blockiert bis diese angekommen ist. Ein Beispiel für diese Art der Kommunikation ist ein Serverprozess für andere Prozesse einen Service bereitstellt.
- **Nonblocking send, nonblocking receive** - Keiner der beiden muss warten.

Adressierung kann direkt oder indirekt unternommen werden. Wird bei der **direkten** Adressierung ein exakter Identifier des Zielprozesses verwendet wohingegen bei der **indirekten** Adressierung nicht an einen Prozess gesendet wird sondern an eine geteilte Datenstruktur in Form einer Queue die üblicherweise als **Mailbox** bezeichnet wird. Der Vorteil bei der Nutzung der indirekten Zustellung ist eine größere Flexibilität und die möglichkeit **one-to-many** Beziehungen abzubilden. Beim **many-to-one** Verhältnis, welches meistens für Server/Client Kommunikation genutzt wird, spricht man hingegen von einem **Port**.



Sequencer und Eventcount

Mechanismen zur Steuerung der Abfolge von Aktionen und sind gut für **Bedingungssynchronisation** geeignet.

Der **Eventcount** ist ein Ereigniszähler der mit 0 initialisiert wird. Dieser hat zwei Operationen:

- `advance(E)` - erhöht Ereigniszähler `E` um 1.
- `await(E, v)` - Blockiert den Prozess bis `E >= v` erreicht wird.

Der **Sequencer** ist eine Art "Nummernvergabemaschine" die eine (atomare) Operation zulässt:

- `ticket(S)` - liefert den aktuellen Wert des Sequencers S und erhöht diesen anschließend um 1.

Das Zusammenspiel von Sequencer und Eventcount lässt sich mit einem Beispiel illustrieren:

```
sequencer S;
eventcount E;

while(true) {
  await(E, ticket(S)) /* waiting until ticketnumber equals E */
    /* critical section */
  advance(E)
}
```

Wiederholungsfragen

Für die Lösung des Problems des geregelten Eintritts in einen kritischen Abschnitt werden drei Eigenschaften gefordert.

(a) Nennen Sie diese drei Eigenschaften und erklären Sie deren Bedeutung.

- **Mutual Exclusion** - Nur ein Prozess darf auf den kritischen Abschnitt zugreifen
- **Progress** - Entweder befindet sich ein Prozess aktuell im kritischen Abschnitt oder sollte sich keiner darin befinden und es existiert ein Prozess der in diesen eintreten will wird das ermöglicht
- **Bounded Waiting** - Sollte ein Prozess auf den kritischen Abschnitt zugreifen wollen muss die Wartezeit darauf beschränkt sein

(b) Wodurch werden die drei Eigenschaften gewährleistet, wenn Semaphore zum Schutz eines kritischen Abschnitts verwendet werden?

Durch die Mechanismen von Semaphoren die vorgeben, dass ein Prozess nur eintreten darf wenn der `count` Wert nicht 0 ist und ansonsten in eine Blocked Queue eingereiht wird, wird für **Mutual Exclusion** gesorgt. Da nach jeder Verringerung der Semaphore eine Überprüfung stattfindet kommt, sollte kein Deadlock vorliegen, jeder Prozess irgendwann dran (**Progress**). Durch die Queue die zumeist im FIFO-Verfahren abgearbeitet wird kann die Wartezeit nur maximal der Dauer der vorliegenden Prozesse betragen wodurch **Bounded Waiting** erreicht wird.

Was versteht man unter einem Monitor zur Prozesssynchronisation? Nennen Sie die wichtigsten Komponenten und Eigenschaften des Monitors.

Ein Monitor ist ein Softwaremodul bestehend aus:

- **Lokale Daten**, die nur im Monitor zugänglich sind
- **Konditionsvariablen**, besondere Datentypen innerhalb des Monitors auf welche die `cwait` und `csignal` Operationen ausgeführt werden.
- **Prozeduren**, die aufgerufen werden damit ein Prozess in den Monitor eintritt
- **Initialisierungscode**

Zu jedem Zeitpunkt kann sich im Monitor nur ein Prozess befinden. Der Monitor sorgt darüber hinaus für Mutual Exclusion ohne explizites Programmieren und die Bedingungssynchronisation wird über die Monitorvariable erreicht. Die zwei Operationen innerhalb des Monitors sind `cwait(cond)`, bei der gewartet wird bis `cond == true` und `csignal(cond)`, die den Prozess fortsetzt der auf `cond` wartet.

Gegeben sei ein Computersystem, in dem Ihnen zur Synchronisation bzw. Kommunikation von Prozessen nur Nachrichten zur Verfügung stehen (d.h., es gibt keine Semaphore oder andere Synchronisationskonstrukte). Nennen Sie zwei verschiedene Möglichkeiten, wie Sie in diesem Computersystem einen konsistenten Datenaustausch zwischen parallelen Prozessen realisieren können.

- **Blocking send, blocking receive** - Absender und Empfänger sind blockiert bis die Nachricht ankommt (*Rendezvous*). Geeignet für enge Prozesssynchronisation.
- **Nonblocking send, blocking receive** - Absender kann mit Prozessabarbeitung fortfahren, jedoch ist der Empfänger bis zum Empfang blockiert.
- **Nonblocking send, nonblocking receive** - Keiner der beiden muss warten.

Wichtige Begriffe

Race Condition - Entsteht wenn mehrere Prozesse oder Threads gleichzeitig Daten lesen oder schreiben möchten und das Endresultat von der Instruktionsabfolge abhängig ist.

Atomare Operation - Instruktionsabfolge die nicht unterbrochen werden kann und daher unteilbar ist.

Kritischer Abschnitt - Codeabschnitt innerhalb eines Prozesses der Zugriff auf geteilte Daten benötigt und nicht gleichzeitig mit anderen Prozessen ausgeführt werden darf.

Deadlock - Die Situation in der zwei oder mehr Prozesse nicht fortfahren können da beide darauf warten, dass der andere etwas macht

Livelock - Die Situation in der zwei oder mehr Prozesse duchgehend ihre Zustände ändern ohne "sinnvolle Arbeit" zu verrichten

Mutual Exclusion - Die Anforderung, dass kein Prozess Zugriff auf geteilte Ressourcen hat sobald ein anderer darauf zugreift

Starvation - Ein ausführbarer Prozess wird vom Scheduler übersehen, nie der CPU zugewiesen und "verhungert".

Deadlock

Ein Deadlock ist das permanente Blockieren einer Menge von Prozessen die um Ressourcen konkurrieren bzw. miteinander kommunizieren. Dies entsteht durch einen **zyklischen Ressourcenkonflikt** in dem jeder Prozess eine Ressource hält und auf eine andere wartet die gerade vom anderen Prozess gehalten wird.

Es wird zwischen **erneuerbaren** und **konsumierbaren** Ressourcen unterschieden. Erneuerbar sind jene Ressourcen die sich nicht verbrauchen lassen, wie beispielsweise Prozessorleistung, I/O Kanäle, Speicher und Geräte. Konsumierbare Ressourcen hingengen sind jene die produziert und konsumiert (bzw. zerstört) werden können. Beispiele hierfür sind *interrupts*, Signale, Messages und Informationen in I/O Buffern.

Damti ein Deadlock möglich ist müssen vier Voraussetzungen erfüllt sein:

1. **Mutual Exclusion** - Ein Prozess hat exklusiven Zugriff auf Ressourcen
2. **Hold and wait** - Prozesse können Ressourcen halten während sie auf andere warten
3. **No preemption** - Eine zugewiesene Ressource kann einem Prozess nicht mehr weggenommen werden
4. **Circular wait** - Eine geschlossene Kette von Prozessen existiert, sodass jeder Prozess zumindest eine Ressource hält die der nächste benötigt.

Behandlung von Deadlocks

Deadlock Prevention

Die Strategie der Deadlock Prevention ist im Prinzip ein Systemdesign welches die Möglichkeiten eines Deadlocks ausschließt. Es wird zwischen **direkten** und **indirekten** Methoden unterschieden wobei sich die direkten Methoden um die ersten drei Voraussetzungen kümmern und indirekte das Aufkommen von Circular waits verhindern.

- **Mutual Exclusion** - kann nicht unterbunden werden
- **Hold and Wait** - Prozess muss alle benötigten Ressourcen auf einmal anfordern und wird blockiert bis dies möglich ist.
- **No Preemption** - Hält ein Prozess gewisse Ressourcen und kann keine weiteren anfordern muss die Ressource freigegeben und neu angefordert werden. Alternativ kann der Prozess der die Ressource gerade hält zur Freigabe gezwungen werden. Anwendbar für Ressourcen, deren Zustand leicht gespeichert und wiederhergestellt werden kann (z.B. der Prozessor dank Processor State Information)
- **Circular Wait** - Mit einer strikten linearen Ordnung für Ressourcenklassen kann die Reihenfolge der Ressourcenvergabe eingeschränkt werden.

Deadlock Avoidance

Wird in der Deadlock Prevention eine Ressourcenanfrage blockiert um eine der vier Voraussetzungen zu verhindern, werden bei der Deadlock Avoidance die ersten drei Voraussetzungen gewährt aber Maßnahmen getroffen damit der Deadlock nie erreicht wird. Erreicht wird dieses Ziel indem dynamisch entschieden wird ob die Ressourcenanfrage vom Prozess potentiell zu einem Deadlock führen kann.

Die zwei Mechanismen die dazu führen sind **Process Initiation Denial** und **Resource Allocation Denial**. Der Vorteil der Deadlock Avoidance ist die höhere Parallelität gegenüber der Deadlock Prevention, da keine Prozesse umgekehrt werden müssen. Damit diese Strategie funktioniert muss jedoch der Ressourcenbedarf der Prozesse bekannt sein.

Process Initiation Denial

Ein Prozess wird nicht gestartet, wenn seine Anforderungen zu einem Deadlock führen könnten.

In einem System mit n Prozessen und m verschiedenen Ressourcenarten gibt es die Vektoren $R = (R_1, R_2, \ldots, R_m)$, für die Menge der im System verfügbaren Ressourcen, und $V = (V_1, V_2, \ldots, V_m)$, für die keinem Prozess zugewiesene Menge jeder Ressource. Zusätzlich wird eine Claim und eine Allokationsmatrix generiert wobei gilt C_{ij} = Anforderung von Prozess i für Ressource j und A_{ij} = Aktuelle Allokation der Ressource j an Prozess i.

$$C = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix} \quad A = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$$

Die Claim-Matrix gibt die Maximalen Anforderungen für jeden Prozess und jede Ressource an, wobei eine Reihe jedem Prozess gewidmet ist. Diese Information muss im Vorhinein von einem Prozess deklariert werden damit Deadlock Avoidance funktioniert. Entsprechend kann der Allocation-Matrix die aktuelle Allokation für jeden Prozess entnommen werden.

Ein Prozess P_{n+1} wird nur gestartet, wenn seine Ressourcenanforderungen keinen Deadlock hervorrufen können. Also nur wenn gilt:

$$R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$$

D.h. ein Prozess wird nur gestartet wenn der maximale Claim für alle aktuellen Prozesse inklusive dem neuen Prozess abgedeckt werden kann.

Resource Allocation Denial - Banker's Algorithm

Eine Ressourcenanforderung wird nicht gewährt, wenn ein Deadlock entstehen könnte.

Die Strategie wird auch als **Banker's Algorithm** bezeichnet und beinhaltet zunächst das Konzept des **State** der die aktuelle Allokation von Ressourcen an Prozesse beschreibt. Es wird zwischen einem **safe state**, einem Zustand in dem *zumindest eine Abfolge* an Allokationen möglich ist um keinen Deadlock zu erzeugen, und einem **unsafe state** unterschieden.

Hierfür wird eine zusätzliche Need-Matrix N definiert die sich aus $C_{ij} - A_{ij}$ ergibt. Anders ausgedrückt: $Maximum - Allocation = Need$.

$$N = \begin{pmatrix} N_{11} & N_{12} & \cdots & N_{1m} \\ N_{21} & N_{22} & \cdots & N_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ N_{n1} & N_{n2} & \cdots & N_{nm} \end{pmatrix}$$

Über den Vergleich dieser Matrix mit den Verfügbaren Ressourcen kann herausgefunden werden ob einer der Prozesse vollendet werden kann. Sei beispielsweise folgende N

$$\begin{array}{c} \\ P1 \\ P2 \\ P3 \\ P4 \end{array} \begin{array}{c} \begin{array}{ccc} R1 & R2 & R3 \end{array} \\ \begin{bmatrix} 2 & 2 & 2 \\ 0 & 0 & 1 \\ 1 & 0 & 3 \\ 4 & 2 & 0 \end{bmatrix} \end{array}$$

sowie ein Verfügbarkeitsvektor V

$$V = \begin{array}{ccc} R1 & R2 & R3 \\ 0 & 1 & 1 \end{array}$$

Ist ersichtlich, dass der Zustand ein **safe state** ist, da der Prozess $P2$ vollständig ablaufen kann, da noch eine Ressource von $R3$ benötigt wird und diese auch verfügbar ist.

Deadlock detection

Da Deadlock Präventionsstrategien sehr konservativ sind und das Problem mit dem Ansatz lösen den Zugang zu Ressourcen einzuschränken gibt es als alternative Lösung die Deadlock Detection. Hier werden weder Ressourcenzugriffe noch Prozessausführungen limitiert, sondern lediglich ein Algorithmus der in periodischen Abständen überprüft ob irgendwo eine **Circular Wait** Bedingung existiert.

Wird ein Deadlock gefunden benötigt man eine **Recovery** Strategie um den erkannten Deadlock zu verhindern. Die einfachste Lösung ist der Abbruch aller beteiligten Prozesse (eine häufig angewandte Strategie). Alternativ können alle beteiligten Prozesse mittels eines **Rollbacks** zu einem definierten Checkpoint zurückgesetzt, und neu gestartet

werden, was natürlich zu einem wiederholten Deadlock führen kann.

Eine etwas subtilere Strategie wäre das Abbrechen einzelner beteiligter Prozesse bis der Deadlock beseitigt ist. Nach jedem Abbruch muss der Algorithmus erneut aufgerufen werden um dies sicherzustellen. Abgesehen davon, können den Prozessen schrittweise die Resourcen entzogen werden bis kein Deadlock mehr vorliegt.

In der Praxis kommt es selten vor, dass nur einer der Ansätze gewählt und ausgeführt wird. Im Rahmen der **integrierten Deadlock-Strategie** werden verschiedene Ansätze kombiniert und für unterschiedliche Situationen unterschiedliche Strategien angewandt.

Wiederholungsfragen

Was versteht man unter Deadlock Avoidance? Geben Sie zwei Strategien für Deadlock Avoidance an und beschreiben Sie diese.

Bei der Deadlock Avoidance wird versucht Deadlocks gar nicht erst auftreten zu lassen um Prozesse nicht rückgängig machen zu müssen. Die zwei Strategien sind:

- **Process Initiation Denial** - Prozess wird nicht gestartet, falls die Ressourcenanforderungen einen Deadlock auslösen könnten. Dafür müssen die Ressourcen bereits im Vorhinein bekannt sein.
- **Resource Allocation Denial** - Ressourcenanforderungen werden verweigert falls Deadlock entstehen könnte (Verwaltung über Matrizen und Safe/Unsafe State)

Nennen Sie die Bedingungen für das Eintreten eines Deadlocks und erklären Sie diese.

Mutual Exclusion - Nur ein Prozess darf auf einen kritischen Abschnitt zugreifen

Hold and wait - Prozesse dürfen Ressourcen halten, während sie auf neue warten

No preemption - Hat ein Prozess eine Ressource übernommen kann diese nicht mehr weggenommen werden und wird erst wieder freigegeben wenn sie nicht mehr benötigt wird

Circular Wait - Jeder Prozess wartet auf eine Ressource die ein anderer Prozess hält, und hält eine Ressource auf die ein anderer Prozess wartet

Bei der Deadlock Prevention spricht man von einem Safe State bzw. einem Unsafe State. Erklären Sie die Bedeutung dieser Begriffe.

- **Safe State** - Es gibt zumindest eine Prozessabarbeitungsreihenfolge die nicht zu einem Deadlock führt
- **Unsafe State** - Deadlock kann auftreten

Jeder Prozess benötig für seine Abarbeitung CPU und Arbeitsspeicher. Welche Strategien werden angewandt, um das Auftreten eines Deadlocks durch den Wettbewerb um diese Ressourcen auszuschließen? Erklären Sie, wie diese Verfahren auf einem Computersystem konkret realisiert werden.

- **Deadlock Prevention** - Deadlock Bedingungen werden versucht aufzuheben.
    - **Hold and Wait** - Prozess muss alle benötigten Ressourcen auf einmal anfordern und wird blockiert bis dies möglich ist.
    - **No Preemption** - Hält ein Prozess gewisse Ressourcen und kann keine weiteren anfordern muss die Ressource freigegeben und neu angefordert werden.
    - **Circular Wait** - Mit einer strikten linearen Ordnung für Ressourcenklassen kann die Reihenfolge der Ressourcenvergabe eingeschränkt werden.
- **Deadlock Avoidance** - Entweder wird ein Prozess gar nicht erst gestartet oder die angeforderten Ressourcen nicht freigegeben.
- **Deadlock Detection** - Nach jeder Ressourcenzuweisung wird gecheckt ob ein Deadlock aufgetreten ist. Sollte dies der Fall sein werden die Prozesse entweder abgebrochen, zurückgesetzt oder Ressourcen schrittweise entzogen.

Memory Management

Die Anforderungen die vom Memory Management erfüllt werden müssen sind:

- **Partitioning** - Speicheraufteilung auf Prozesse
- **Relocation** - Positionierung von Code und Daten im Speicher
- **Protection** - Speicherschutz
- **Sharing** - Gemeinsamer Zugriff auf Speicher
- **Performance** - effektive logische/physische Organisation

Partitionierung des Speichers

Fixed Partitioning

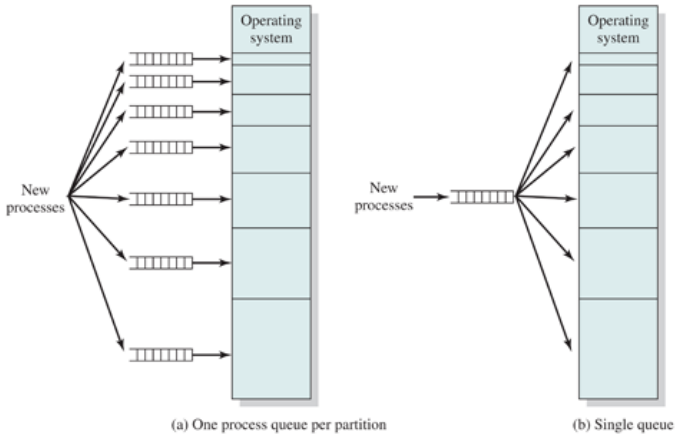
Mit diesem System wird der Speicher in fixe Partitionsgrößen aufgeteilt denen Prozesse zugewiesen werden können. Diese Partitionierungen können alle von der gleichen Größe sein, oder unterschiedlich viel Speicherplatz übernehmen.

Es geben sich jedoch zwei gravierende Probleme mit *equal size partitioning*:

- Ist ein Programm zu groß um in eine Partition zu passen muss der/die ProgrammiererIn das Programm entsprechend designen.
- Speichernutzung ist extrem ineffizient, da ein Programm welches noch so klein sein kann die gesamte Partition blockiert. Diese Situation wird auch als **Interne Fragmentierung** bezeichnet.

Im nächsten Schritt muss überlegt werden welches System gewählt wrid um Prozesse einer Partition zuzuweisen. Bei unterschiedlich großen Partitionen lassen sich zwei Strategien definieren.

Die einfachste Variante ist eine Queue pro Partition anzulegen, in der jeweils ein Prozess zur kleinstmöglichen Partitionsqueue zugewiesen wird. Das offensichtliche Problem mit diesem Ansatz ist, dass im Fall von einer großen Anzahl an Prozessen gleicher Größe eine Partition benutzt wird während die restlichen leer bleiben. Daher ist eine Variante, in der eine einzige Queue für alle Partition genutzt wird besser geeignet. Wird ein Prozess in den Speicher geladen, wird die kleinstmögliche Partition dafür genutzt. Sind alle Partitionen besetzt, wird geswappt. Die Nutzung von Fixed Partitioning ist heutzutage nicht mehr existent.



Dynamic Partitioning

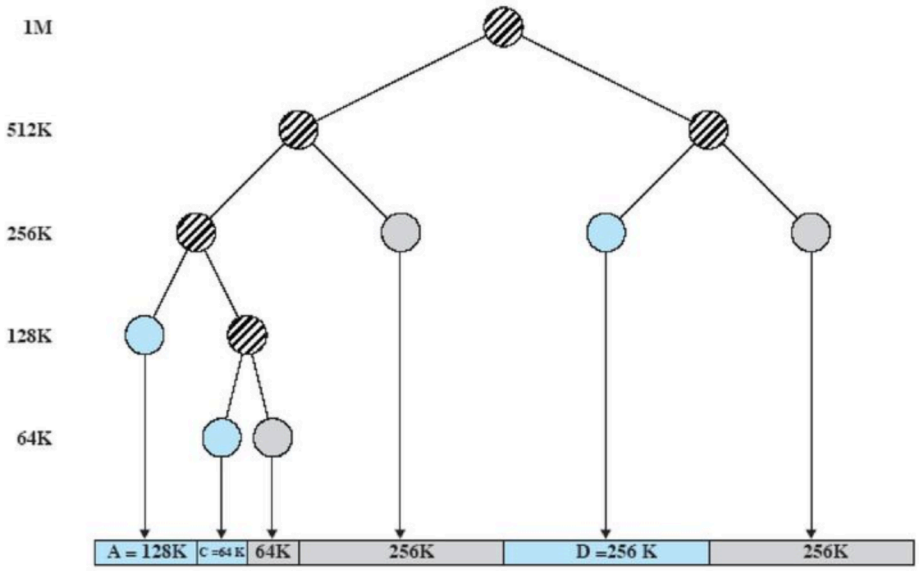
Eine alternative zum Fixed Partitioning, die ebenfalls nicht mehr in Benutzung ist, ist das Dynamic Partitioning. Der Ansatz hier ist, dass jedem Prozess exakt soviel Speicher zugewiesen wird wie dieser benötigt. Diese Methode fängt zwar gut an, weist jedoch im weiteren Verlauf eklatante Schwächen auf, da durch die beendeten Prozesse zwischen den zugewiesenen Speicherpartitionen "Löcher" entstehen die zu klein sind um neue Prozesse aufzunehmen wodurch die gesamte Speicherverwaltung stark ineffizient ist. Dieses Phänomen wird auch als **Externe Fragmentierung** bezeichnet. Eine (kostenintensive) Lösung für dieses Problem ist die sogenannte **Compaction** bei der die besetzten Speicherbereiche "zusammengeschoben" werden um einen zusammenhängenden Speicher für neue Prozesse zu öffnen.

Die Platzierungsstrategien für Dynamic Partitioning:

- **best fit** - der Speicherblock mit der ähnlichsten Größe wird gewählt
- **first-fit** - nachdem der Speicher von oben angefangen gescannt wird kommt der Prozess in den ersten passenden Bereich
- **next-fit** - der Speicher wird von der letzten erfolgreichen Platzierung an gescannt und der Prozess in den nächsten passenden Bereich gegeben

Buddy System

Um die schwächen der beiden Systeme auszugleichen wird mit dem Buddy System ein hybrider Ansatz vorgeschlagen. Im Prinzip wird hier der gesamte Speicherplatz aufgebrochen mit 2er-Potenzen bis die kleinstmögliche Partitionierung für den Prozess erreicht wurde. Wird beispielsweise ein 100MB großer Prozess in einen 1GB großen Speicher geladen, wird dieser rekursiv bis zur 128MB Partition halbiert in die dann der Prozess geladen wird. Folgt auf diesen ein 240MB großer Prozess, kommt dieser in die bereits existierende 256MB Partition. Dieser Prozess lässt sich als Binärbaum gut illustrieren. Sobald es zwei Blattknoten mit demselben Elterknoten gibt muss einer der beiden belegt sein.



Relocation

Es wird eine Möglichkeit benötigt Prozesse dynamisch zu positionieren, daher müssen Referenzen auf physikalischen Speicher veränderbar sein. Hier muss unterschieden werden zwischen **logischen** und **physikalischen** Adressen.

Eine physikalische Adresse ist die tatsächliche Position im Hauptspeicher. Die logische Adresse hingegen ist im einfachsten Fall eine Variablenzuweisung. Im Allgemeinen ist es eine Referenz auf eine Position im Speicher die komplett unabhängig von der tatsächlichen pyhsikalischen Position ist. Eine **relative Adresse** ist eine Position die relativ zu einem bekannten Punkt (einer logischen Adresse) im Programm ist. Übersetzter Code enthält logische, meist relative Adressen die zur Laufzeit aufgelöst werden.

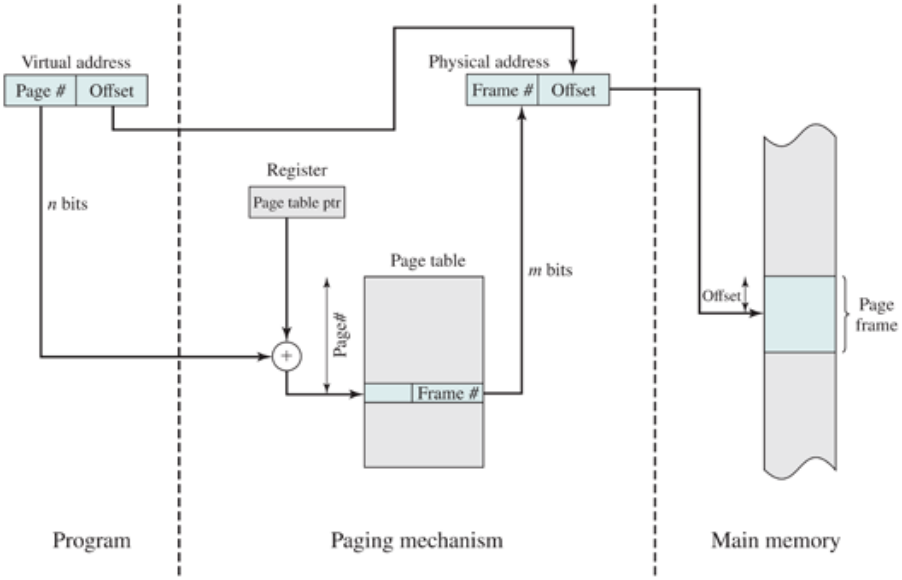
Segmentierung

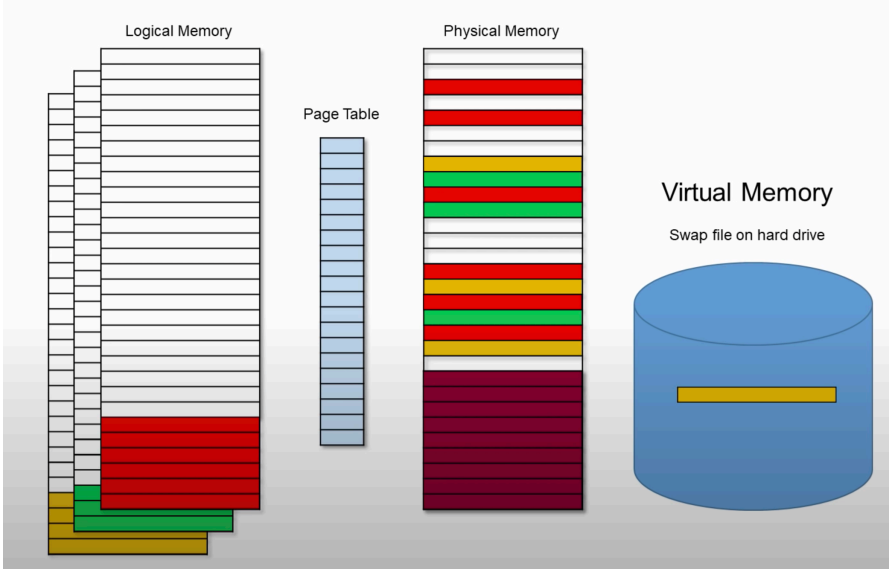
Ein Programm kann in Blöcke unterschiedlicher Länge, sogenannter **Segmente**, unterteilt werden um zu verhindern, dass ein großer zusammenhängender Speicherbereich reserviert werden muss. Um ein Programm weiterhin verwalten zu können benötigt man eine Segmenttabelle in der für alle Segmente ein Eintrag existiert in dem der Start und die Länge eines Segments notiert wird.

Paging

Beim Paging wird der Hauptspeicher in kleine, gleichgroße Bereiche unterteilt die als **Page Frames** bezeichnet werden. Prozesse werden ebenfalls unterteilt, in sogenannte **Pages** die der Größe der *page frames* entsprechen. Für die Prozesse selber existiert eine eigene Ansicht auf den Speicher, die als **logischer Speicher** bezeichnet wird. Ein Prozess sieht den gesamten Speicher als verfügbar und belegt dessen *page frames* nach Bedarf. Das OS wiederum verwaltet die verschiedenen pages und übersetzt zwischen dem logischen und dem physikalischen Speicher. Die Übersetzung funktioniert über **Page**

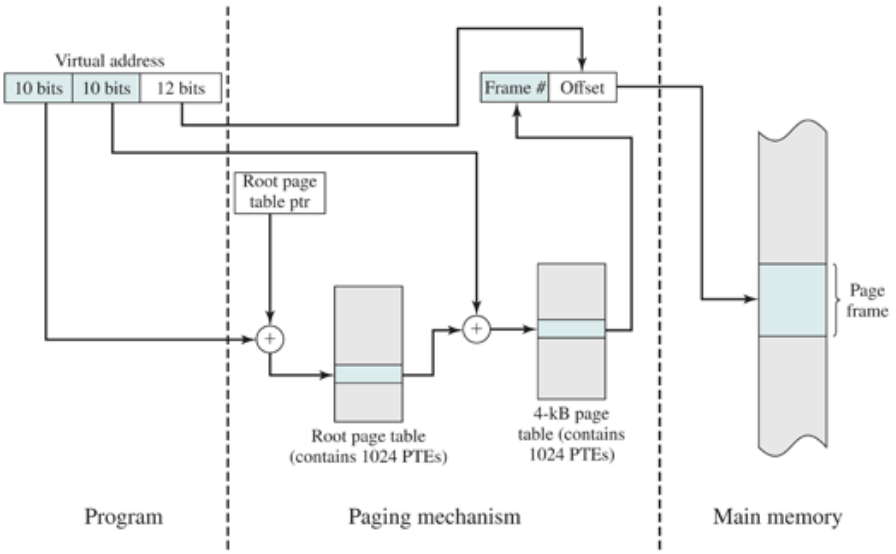
Tables, welche jeder Prozess besitzt, in denen die Übersetzung steht. Ist der physikalische Speicher belegt kann über *swapping* eine **Page** auf den Sekundärspeicher ausgelagert werden der als **Virtual Memory** bezeichnet wird.



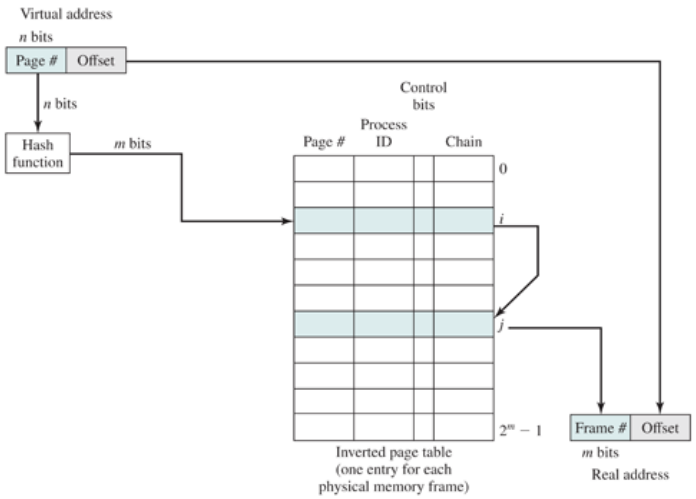


Versucht der Prozessor auf eine Page zuzugreifen die sich aktuell nicht im Hauptspeicher befindet kommt es zu einem **Page Fault** und der Prozess muss erst aus der Virtual Memory geladen werden. Verbringt der Prozessor zuviel Zeit damit Pages von der Virtual Memory zu laden kann es zum **Thrashing** kommen, bei dem ein drastischer Einbruch der Effektivität und eine starke Verlangsamung der Abarbeitungsgeschwindigkeit eintritt.

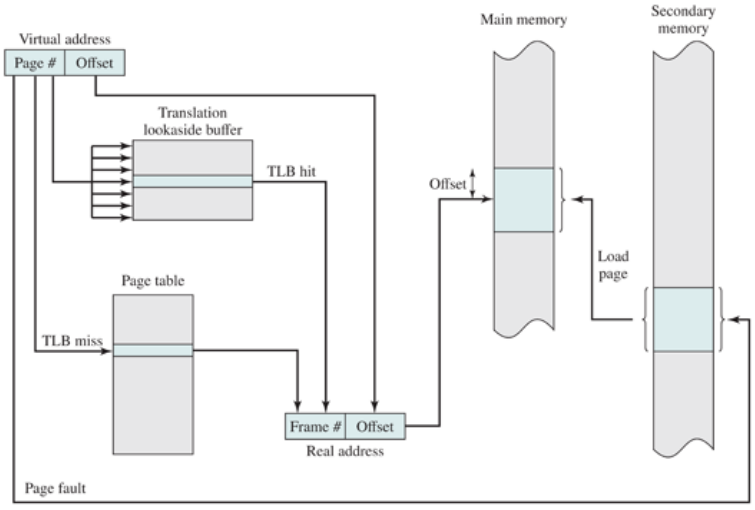
Die Page Table kann mit der Zeit sehr groß werden und selbst in Pages aufgeteilt werden. Eine solche Tabelle wird als **Multilevel Page Table** bezeichnet. Für eine solche gibt es ein **Outer (bzw. Root) Page Table** welche als Verzeichnis dient für die untergeordneten Pages in der Tabelle.



Alternativ kann auch auf eine **Inverted Page Table** zugegriffen werden. Anstatt einer Page Table pro Prozess wird dieser Ansatz "invertiert" und es existiert eine einzige Page Table für den gesamten Arbeitsspeicher. Zusätzlich gibt es einen IPT-Eintrag pro physikalischem Frame. Der Zugriff wird mittels Hashing gemacht.



Im Prinzip passieren bei jedem Zugriff über eine logische Speicheradresse zwei physikalische Speicherzugriffe. Eine um den Eintrag im Page Table zu laden, und ein weiterer um die Daten zu laden. Um eine verdopplung der Speicherzugriffszeit zu verhindern wird meist ein spezieller high-speed cache für Page Table Einträge genutzt welcher als **Translation Lookaside Buffer (TLB)** bezeichnet wird. Von der Funktionalität her arbeitet dieser Cache wie der Memory Cache und beinhaltet die kürzlich geladenen Page Table Einträge.



Der Aufbau des TLB ermöglicht es aufgrund einer assoziativen Suche innerhalb eines Zyklus die Frame Nummer zu erhalten. Bei jedem Context Switch wird der TLB zurückgesetzt.

Beim Laden der Pages in den Hauptspeicher gibt es verschiedene Strategien:

- **Demand Paging** ladet eine Page genau dann wenn sie referenziert wird. Führt beim Starten eines Prozesses zu vielen Page Faults.
- **Prepaging** ladet im Voraus mehr Pages als eigentlich benötigt. Der Nachteil hier ist, dass auch nicht benötigte Pages unnötig in den Hauptspeicher geladen werden.

Replacement Policy

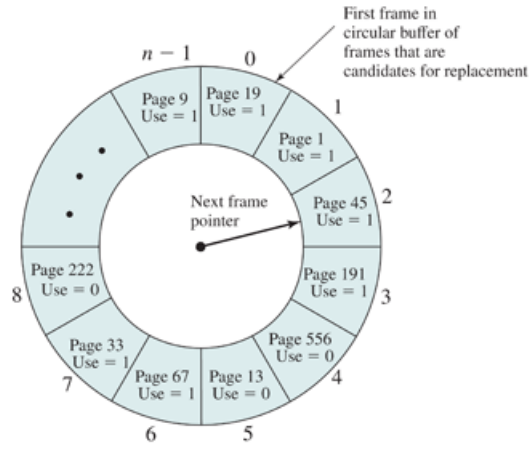
Die Replacement Policy bestimmt, welche Page beim Laden einer neuen Page ersetzt wird und stattdessen in den Virtual Memory geschrieben wird. Hierfür stehen verschiedene Strategien zur Auswahl:

- **OPT** Policy - Hier wird die Page ersetzt deren Referenz am weitesten in der Zukunft ist. Ein Vorteil dieser Strategie ist, dass es hier tendenziell zu den wenigsten **page faults** kommt. Sie ist jedoch nicht wirklich implementierbar, da eine Voraussetzung dafür ein OS wäre, welches ein exaktes Wissen zukünftiger Events hätte. Sie wird aber hergenommen um "echte" Strategien zu bewerten (da sie rückblickend funktioniert).
- **LRU** Policy - Die **Least Recently Used** Strategie ersetzt die Page, die am längsten nicht benutzt worden ist. Aufgrund des Lokalitätsprinzips sollte diese Page die geringste Wahrscheinlichkeit haben in Zukunft referenziert zu werden. Problematisch ist hier jedoch die Implementierung da das Speichern, Suchen und Aufrufen der Zugriffszeiten für Pages einen riesigen Overhead kreiert.
- **FIFO** Policy - Eine einfach zu implementierende Strategie bei der einfach die älteste (am längsten im Speicher befindliche) Page ersetzt wird ohne Rücksicht auf andere Parameter zu nehmen. Der Ansatz stößt jedoch schnell an seine Grenzen, da Programme die stark und häufig genutzt werden hier auch am häufigsten in den Speicher geladen werden müssten.

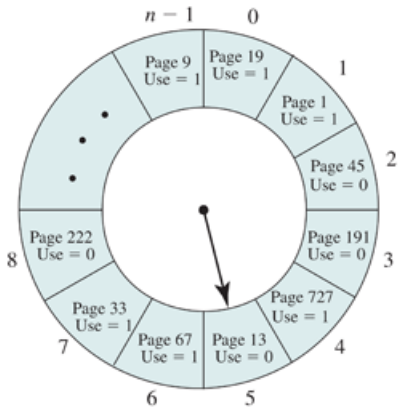
- **Clock Policy** - Ein Versuch an die Performance der LRU heranzukommen jedoch den Overhead zu reduzieren stellt die Familie der Clock Policies dar. In diesem kommt es zu kaum mehr Page Faults als bei der LRU.

  In der einfachsten Implementierung kommt zu jedem Frame ein Bit hinzu welches als **Use Bit** bezeichnet wird. Dieses wird auf 1 gesetzt sobald die Page in den Speicher geladen oder von einem Programm referenziert wird. Wird es nun Zeit eine Page zu ersetzen geht der **Frame Pointer** so lange "im Kreis" bis er auf eine Page zeigt deren Use Bit den Wert 0 hat. Bei allen Prozessen die im Verlauf überprüft werden, und dementsprechend als Use Bit den Wert 1 haben, wird das Use Bit auf 0 gesetzt. Die erste Page die mit einem Use Bit 0 aufgerufen wird, wird durch die neue Page ersetzt, und der Pointer zeigt auf den darauffolgenden Prozess.

  In der Illustration startet der Pointer bei der Page an Position 2 und geht bis zu Position 4 (die erste Position mit Use Bit 0). Die Positionen 2 und 3 haben nun ebenfalls ein Use Bit 0. An Position 4 wird nun die Page ersetzt, das Use Bit auf 1 gesetzt und der Pointer um 1 erhöht.



(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

Resident Set

Das Resident Set beschreibt die Pages eines Prozesses, die sich aktuell im Hauptspeicher befinden. Die Größe des Resident Set muss ebenfalls gemanaged werden indem entschieden werden muss wieviele Page Frames einem Prozess zugeteilt werden sollen. Je weniger Speicher einem Prozess zugewiesen wird, desto mehr Prozesse finden im

Hauptspeicher Platz und die Wahrscheinlichkeit, dass das OS zumindest einen einsatzbereiten Prozess findet ist höher. Ist eine kleine Anzahl von Pages eines Prozesses im Hauptspeicher würde es zu vielen Page Faults kommen.

Unterschieden wird zwischen einer **fixed-allocation** Policy, bei der einem Prozess eine fixe Anzahl an Frames im Speicher zugewiesen wird die bei der Prozesserstellung festgelegt wird, und einer **variable-allocation** Policy, bei der sich die allozierten Page Frames sich während der Prozesslebensdauer noch verändern können.

Eine beliebte Strategie zur Allokation von Pages ist die **Working Set Strategie**. Hierbei werden Frames aufgrund der Lokalitätsannahme variabel alloziert. Das Working Set $W(D, t)$ beschreibt die Menge der Pages eines Prozesses die in den letzten D (virtuellen) Zeiteinheiten referenziert worden sind. Während der Prozessausführung wird dieses Working Set beobachtet, und Pages die sich nicht in diesem befinden werden periodisch gelöscht. Der Verlauf eines Working Sets zeichnet sich durch ein starkes Wachstum beim Start eines Prozesses aus, welches sich während der Ausführung stabilisiert. Die Implementierung einer "echten" Working Set Strategie ist jedoch mit vielen Schwierigkeiten verbunden, da Seitenreferenzen mitgeloggt und geordnet werden müssen. In der Praxis werden einfach die Anzahl der Page Faults pro Zeitintervall und Prozess beobachtet anstatt eines Working Sets.

Wichtige Begriffe

Lokalitätsprinzip - Nach einem Zugriff auf einen Adressbereich wird der nächste Zugriff mit hoher Wahrscheinlichkeit in der Nähe erfolgen.

Wiederholungsfragen

Was versteht man unter dem Begriff Relocation? Wofür ist Relocation von Bedeutung?

Prozesse werden beim aus- und einlagern unterschiedlichen Adressen im Hauptspeicher zugewiesen. Um dieses Problem zu lösen bedient man sich verschiedener Arten von Adressen:

- **Logische Adresse** - Logische Speicherreferenz die erst auf die physische Adresse übersetzt werden muss
- **Relative Adresse** - Spezielle Art einer logischen Adresse die in Bezug auf einen bekannten Punkt ausgedrückt wird
- **Physische Adresse** - Die tatsächliche Adresse im Hauptspeicher.

Beschreiben Sie, wozu und wie eine Page Table verwendet wird. Geben Sie weiters an, welche Informationen in den Tabelleneinträgen einer Page Table gespeichert werden.

Beim Paging wird der Hauptspeicher in gleich große Speicherblöcke zerlegt die man als **Page Frames** bezeichnet. Prozesse werden ebenfalls in Blöcke zerlegt die von der Größe her den Frames entsprechen und als **Pages** bezeichnet werden. Da in einem solchen System ein Prozess den gesamten Speicher als verfügbar sieht muss das OS das Laden der benötigten Pages managen. Die **Page Table** steht zwischen dem logischen und dem physischen Adressraum und kümmert sich um die Übersetzung. Die logische Adresse eines Programms beinhaltet die Page Number und einen Offset. Über die Page Table kann die zugehörige Frame Number gefunden werden die durch das zusammenfügen mit dem Offset die tatsächliche physische Adresse im Hauptspeicher liefert.

Wir betrachten ein System mit Virtual Memory Management. Diskutieren Sie, welche der folgenden Situationen beim Referenzieren einer virtuellen Adresse auftreten bzw. nicht auftreten kann:

(a) TLB Miss ohne Page Fault

Ja ist möglich wenn sich die Page im Hauptspeicher befindet.

(b) TLB Miss mit Page Fault

Ja ist möglich, da in diesem Fall die CPU die fehlende Page vom Sekundärspeicher ladet.

(c) TLB Hit ohne Page Fault

Ist immer der Fall bei einem TLB Hit.

(d) TLB Hit mit Page Fault

Nein nicht möglich, da eine Page wenn sie bereits im TLB ist nicht mehr geladen werden muss.

Wozu wird die Clock Policy verwendet? Beschreiben Sie deren Funktionsweise.

Mittels Clock Policy wird versucht eine möglichst optimale Page Replacement Strategie zu finden. Hier wird jedem Page ein zusätzliches Bit, das sogenannte **Use Bite**, zugewiesen welches dazu dient um zwischen benötigten und nicht benötigten Pages zu unterscheiden. Wird es Zeit eine Page auszutauschen geht ein Pointer reihum durch die Page Frames und setzt bei jedem das Use Bit auf 0. Trifft es auf eine Page bei der das Use Bit bereits auf 0 gesetzt ist wird diese ersetzt und der Pointer um eins erhöht.

Was versteht man unter der Working-Set Strategie? Beschreiben Sie deren Funktionsweise und erklären Sie, wie diese Strategie zur Optimierung eines Paging-Systems eingesetzt werden kann.

Das Working Set ist das Set an Pages die aktuell von einem Prozess genutzt werden. Es beschreibt eine Funktion in der die Anzahl der aufgerufenen Frames D in einem virtuellen Zeitraum t überwacht werden. In periodischen Abständen wird überprüft ob ein Frame noch benötigt wird (d.h. im Working Set ist). Befindet sich ein Frame nicht mehr im Working Set kann dieses entfernt werden.

Beschreiben Sie Aufgabe und Funktion eines Translation Lookaside Buffers? Worauf hat man bei der Betriebssystemimplementierung bei einem Process Switch zu achten, wenn man einen Translation Lookaside Buffer verwendet?

Der TLB wird genutzt um Page Faults zu reduzieren indem es als Hochgeschwindigkeitscache vor die Page Table gestellt wird. Benötigt nun ein Prozess eine Page wird zunächst im TLB nachgesehen und sollte diese dort gespeichert sein (*TLB hit*) kann sie sofort aufgerufen werden. Bei einem TLB miss muss in der Page Table nachgesehen werden und es kann zu einem Page Fault kommen sollte sich die gesuchte Page nicht im Hauptspeicher befinden.

Bei einem Process Switch muss darauf geachtet werden, dass der TLB komplett zurückgesetzt wird.

Was ist Swapping? Wann wird es angewandt?

Mit Swapping bezeichnet man das Auslagern eines Prozesses in den Sekundärspeicher wenn Platz im Hauptspeicher geschaffen werden muss. Meist werden Prozesse die beispielsweise auf eine I/O Operation warten ausgelagert bis diese fertiggestellt wird.

Was ist Thrashing, wodurch kommt es dazu? Wie erkennt das Betriebssystem Thrashing? Wie kann dieses Problem beseitigt werden?

Thrashing tritt auf wenn es zu häufig zu Page Faults kommt und der Prozessor übermäßig damit beschäftigt ist fehlende Pages aus dem Sekundärspeicher zu laden wodurch die Prozessorleistung dramatisch abfallen kann. Erkannt werden **Erkannt** wird Thrashing vom System da die CPU Ausnutzung abfällt (aufgrund häufiger Page Faults) und dadurch vom Scheduler weitere Prozesse geladen werden und die CPU Ausnutzung weiter fällt. **Lösen** kann das OS dieses Problem beispielsweise mit der Working Set Strategie.

Was versteht man unter interner Fragmentierung und externer Fragmentierung? Beschreiben Sie die Begriffe und geben Sie je ein Beispiel an.

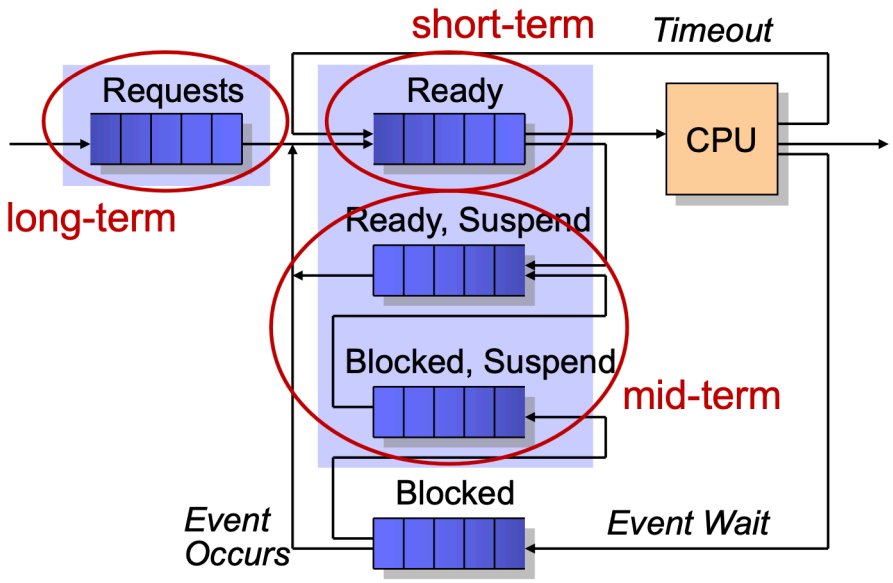
- **Externe Fragmentierung** entsteht bei der dynamischen Vergabe von Speicherplatz. Werden Prozesse ausgelagert können zwischen den Speicherräumen "Lücken" entstehen die zu klein sind um neue Prozesse aufzunehmen und es kommt zu einer sehr ineffizienten Speichernutzung.
- **Interne Fragmentierung** passiert bei fixen Partitionen wenn ein kleiner Prozess einen sehr großen Speicherbereich einnimmt und daher innerhalb der Partition ein großer Bereich ungenutzten Speichers entsteht.

Beschreiben Sie den Aufbau und Verwendung einer Multilevel Page Table (eventuell mit Skizze). Warum werden Multilevel Page Tables verwendet?

Da Page Tables in modernen Systemen viel Speicher benötigen (und nicht rausgeswappt werden können) und aufgrund der Tatsache, dass jedes Programm eine eigene Page Table benötigt kann die Verwaltung dieser Page Tables wiederrum durch eine eigene Page Table gemacht werden. Diese erste Page Table wird auch als "Root Page Table" bezeichnet. Durch diesen Mechanismus können die weiteren Levels an Page Tables in den Sekundärspeicher ausgelagert werden. Innerhalb der Root Page Table befinden sich jetzt nur noch Verweise auf weitere Page Tables.

Scheduling

Das Ziel des Scheduling ist die Festlegung der Abarbeitungsreihenfolge der Prozesse auf dem Prozessor auf eine Art und Weise in der Optimierungsziele wie Durchsatz, Prozessorauslastung, Response Time, uvm. gewährleistet werden. In den meisten Systemen wird das Scheduling auf Basis vom Planungshorizont in lang-, mittel-, und kurzfristiges Scheduling sowie dem I/O Scheduling unterschieden.



- **Long Term Scheduling** - Der LTS Scheduler entscheided welche Programmanfragen in das System gelassen werden um verarbeitet zu werden. Wird eine solche Anfrage gestattet, gelangt der Prozess in die Ready Queue und wird vom **Short Term Scheduler** verwaltet.
- **Medium Term Scheduling** - Das MTS ist Teil der Swapping Funktionalität und verwaltet die Ein- und Auslagerung von Prozessen.
- **Short Term Scheduling** - Der STS (auch **Dispatcher** genannt) wird am häufigsten eingesetzt da hier schlussendlich bestimmt wird welcher Prozess tatsächlich als nächster ausgeführt wird.

Short Term Scheduling Kriterien

	User Oriented	System-Oriented
Performance	Response Time, Turnaround Time, Deadlines (Fertigstellungszeitpunkt)	Throughput, Processor Utilization
Other	Predictability	Fairness, Resource Balance, Priorities

- **Turnaround Time** - Die vergangene Zeit zwischen dem Zulassen und der Beendigung eines Prozesses.
- **Response Time** - Die Zeit zwischen dem Zulassen eines Prozesses und dem ersten Output.

Verwendung von Priorities

In den meisten Systemen bekommen Prozesse zusätzlich einen Prioritätswert zugewiesen um dem Scheduler eine Zuweisung auf Basis der Dringlichkeit von Prozessen zu ermöglichen. Eine Gefahr hier ist jedoch, dass es im Falle einer großen Anzahl von hochpriorisierten Prozessen zu Starvation für Prozesse mit niedrigerer Priorität kommt. Dem kann entgegengewirkt werden indem die Priorität mit der Verweildauer in der Queue verändert wird.

Scheduling Strategien

Es gibt eine große Menge an unterschiedlichen Scheduling Strategien die sich in ihrer Herangehensweise anhand von 2 Parametern unterscheiden lassen:

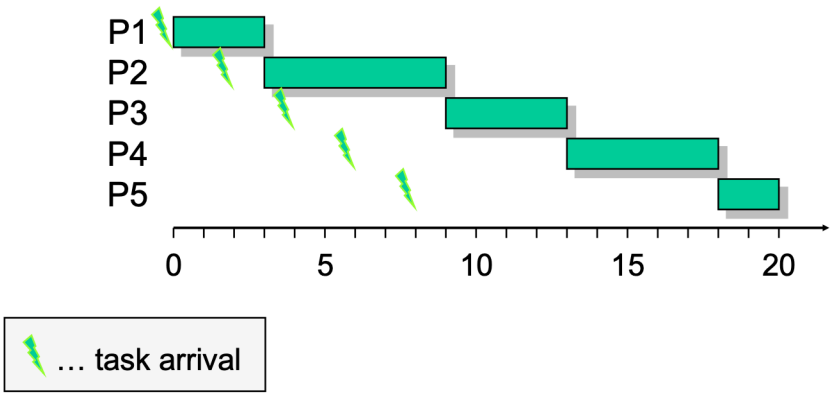
- **Selection Function** - Auswahl des nächsten auszuführenden Prozesses
- **Decision Mode** - Beschreibt die Zustände in denen die Selection Function ausgeführt wird. Es wird unterschieden zwischen **non-preemptive** (keine externe Unterbrechung von Prozessen durch OS) und **preemptive** (Prozesse können durch das OS unterbrochen werden) Decision Modes.

First Come First Served (FCFS)

SF: Auswahl des Prozesses, der bereits am längsten in der Ready Queue wartet

DM: non-preemptive

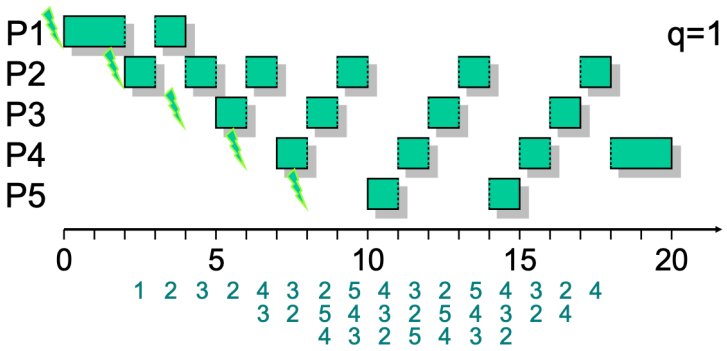
Der einfache FCFS Ansatz begünstigt lange und CPU-intensive Prozesse und führt insgesamt zu Prozessmonopolisierung und einer schlechten Auslastung von CPU und I/O. Jedoch kann die Effektivität oft in Kombination mit einem Prioritätssystem erhöht werden.



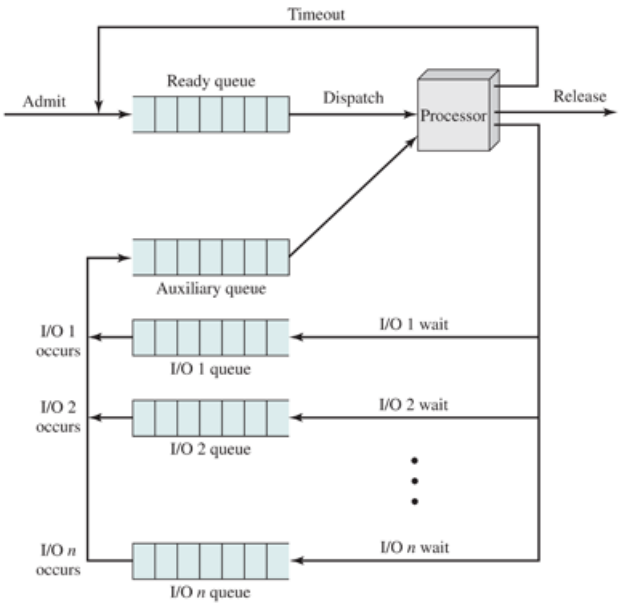
Round Robin (Time Slicing)

SF: wie FCFS | **DM:** preemptive

Im RR System werden Prozesse ebenfalls nach dem FCFS System zugelassen aber es werden Zeitscheiben in gleicher Länge vergeben. Jeder Prozess wird nun eine solche Zeiteinheit lang verarbeitet und kommt danach zurück in die Ready Queue. Wichtig ist eine sinnvolle Zeitscheibenlänge um Overhead zu vermeiden. Allgemein werden I/O Prozesse benachteiligt, da diese die Zeitscheiben nicht voll ausschöpfen und von CPU-intensiven Prozessen "überholt" werden.



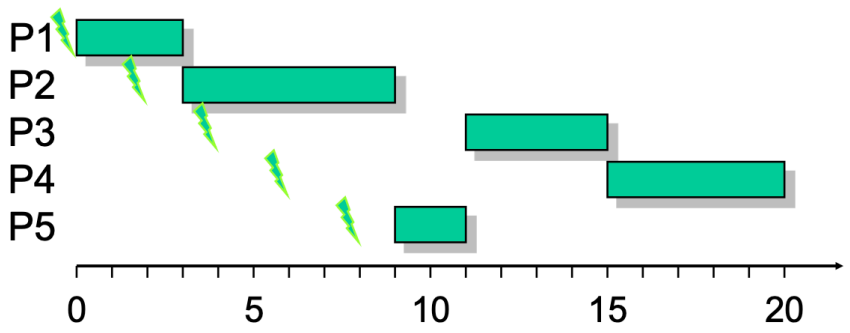
Eine Möglichkeit diese Problematik zu umgehen liefert das sogenannte **Virtual Round Robin** welches eine zusätzliche Queue, die **Auxilliary Queue**, implementiert in der blockierte I/O Prozesse geladen und bei der Ausführung prioritär behandelt werden.



Shortest Process Next (SPN)

SF: Prozess mit kürzestem erwarteten CPU-Burst zuerst | **DM:** non-preemptive

In dieser Strategie werden Prozesse mit der vorraussichtlich kürzesten Dauer priorisiert und bearbeitet. Der SPN-Ansatz führt zu besseren Response Times als FCFS, da kurze, häufig auftretende Prozesse priorisiert werden und dadurch schneller abgearbeitet werden. Ein großer Nachteil dieses Ansatzes ist die Notwendigkeit die Dauer von Prozessen zumindest ungefähr zu kennen und die Möglichkeit, dass es zu Starvation für lange Prozesse kommen kann.



Shortest Remaining Time (SRT)

SF: wie SPN | **DM:** preemptive

Die SRT Strategie ist prinzipiell nur die preemptive Version der SPN Strategie. Hier werden jene Prozesse vom Prozessor ausgewählt, die die kürzeste verbleibende Abarbeitungsdauer haben. Kürzere Prozesse werden hier fair behandelt und es kommt zu weniger Interrupts als beim RR, jedoch ist weiterhin Starvation möglich und Service Time muss protokolliert werden, wodurch es wiederum zu Overhead kommt.

Highest Response Ratio Next (HRRN)

SF: $RR = \frac{w+s}{s}$ | **DM:** non-preemptive

Der Prozess der als nächstes herangezogen wird, wird anhand einer Formel bestimmt in der w die gesamte bisherige Wartezeit und s die geschätzte Service Time darstellt. Der nächste Prozess ist jener mit dem größten RR Wert. Starvation kommt nicht mehr vor, da ein Prozess mit wachsender Wartezeit eher drankommt.

Feedback Scheduling (FS)

SF: basierend auf bisheriger Aufführungszeit | **DM:** preemptive

Ist es nicht möglich im Vorhinein zu wissen wie lange ein Prozess laufen wird, kann mit FS Abhilfe geschafft werden indem länger laufende Prozesse "bestraft" werden. Je mehr CPU-Zeit ein Prozess konsumiert, desto niedriger wird seine Priorität. Um Starvation zu verhindern muss nach einer bestimmten Zeit die Priorität eines Prozesses wieder angehoben werden.

Real Time Scheduling

Real-Time Computing lässt sich am einfachsten als ein System beschreiben welches nicht nur vom logischen Resultat einer Berechnung, sondern auch von dem Zeitpunkt in dem dieses produziert wird abhängt. In einem Real-Time System gibt es **Real-Time Tasks** die zumeist Reaktionen auf externe Events darstellen. Daher ist es möglich einen

Task mit einer entsprechenden Deadline zu assoziieren die eingehalten werden muss. Unterschieden wird zusätzlich zwischen **hard** und **soft** real-time, wobei beim ersteren die Deadline eingehalten werden muss und beim letzteren nicht.

Earliest Deadline First (EDF)

SF: Task mit frühester Completion Deadline zuerst | **DM:** preemptive

Die Tasks mit der frühesten Deadline werden ausgeführt weswegen ein Sortieren notwendig ist.

Wiederholungsfragen

Was versteht man unter Long-Term Scheduling, Mid-Term Scheduling und Short-Term Scheduling? Erklären Sie jeden der Begriffe.

- **LTS** - Entscheidet welche Programmanfragen ins System gelassen werden
- **MTS** - Swapping (Welche Prozesse werden ein- bzw. ausgelagert)
- **STS** - Dispatching, welcher Prozess aus der Ready Queue kommt zum Einsatz

Nennen Sie die Arten von Optimierungszielen, die ein Scheduler beim Prozess-Scheduling verfolgen kann und geben Sie jeweils Beispiele an.

- **Durchsatz** - Möglichst viele Prozesse sollen abgearbeitet werden - SRT Strategie
- **Prozessorauslastung** - Leerlaufzeiten des Prozessors müssen minimiert werden - VRR
- **Fairness** - Jeder Prozess soll irgendwann drankommen und beendet werden - RR
- **Response Time** - Der Output eines jeden Prozesses soll möglichst schnell verfügbar sein - SPN
- **Einhaltung zeitlicher Vorgaben** - Prozessende soll zu gewissem Zeitpunkt eintreten - EDN

Bei welchen der folgenden Scheduling-Strategien kann es zur Starvation kommen? Begründen Sie jeweils Ihre Antwort:

(a) FCFS - Nein, da die maximale Wartezeit den zuvor eingereihten Prozessen entspricht.

(b) Shortest Job First - Ja, da kurze Prozesse potentiell unendlich lange den längeren gegenüber bevorzugt werden können.

(c) Round Robin - Nein, da jeder Prozess reihum eine Zeitscheibe lang zum Einsatz kommt.

(d) Priority Scheduling? - Ja, wenn die Priority sich nicht mit der Wartezeit ändert können höher priorisierte Prozesse durchgehend bevorzugt werden.

Wie funktioniert das CPU-Scheduling nach dem Highest Response Ration Next Verfahren? Welche Vorteile bzw. Nachteile hat diese Schedulingstrategie?

Die Selection function für den nächsten Prozess lautet $RR = \frac{w+s}{s}$ wobei s die geschätzte Service Time und w die bereits gewartete Zeit darstellt. Der Prozess mit dem größten RR Wert, also der der am längsten wartet, wird als nächstes gewählt. Der Vorteil ist, dass es hier nicht zu Starvation kommen kann, jedoch ist eine aufwändige Schätzung der Service Time notwendig damit diese Strategie eingesetzt werden kann.

Wie funktioniert Round Robin Scheduling? Welchen wichtigen Parameter gibt es bei diesem Verfahren? Wie wird man diesen Parameter günstiger Weise wählen?

Jedem Prozess der bereit zum Einsatz ist wird nacheinander eine Zeitscheibe zugewiesen in der die Instruktionen ausgeführt werden können. Der wichtigste Parameter hier ist die Dauer dieser Zeitscheibe die so lang gewählt werden muss, dass es nicht zu zu häufigen Process Switches kommt aber auch nicht zu lang da I/O lastige Operationen sonst den Prozessor zu wenig auslasten würden. Üblicherweise entspricht die Länge einer Zeitscheibe etwas mehr als die Dauer einer typischen Prozessorinteraktion.

I/O Management und Disk Scheduling

Es wird zwischen 3 Arten von externen I/O Geräten unterschieden:

- **Mensch <-> Interface** - Anzeige, Terminal, Tastatur
- **Maschinen <-> Interface** - Datenlaufwerke, USB Sticks, Sensoren
- **Kommunikation** - für den Datenaustausch mit anderen Geräten, z.B. Netzwerk

Zusätzlich gibt es innerhalb der Klassen zum Teil gravierende Unterschiede:

- **Datenrate** - Die Unterschiede zwischen den verschiedenen Geräten und ihrer Datenrate können sich um einige Potenzen unterscheiden
- **Anwendung** - Die Nutzung eines Gerätes setzt unterschiedlichste Software beim OS voraus.
- **Ansteuerungskomplexität** - Braucht ein Drucker ein sehr einfaches Kontrollinterface ist eine Speicherplatte um einiges komplexer.
- **Transfereinheit** - Daten können zeichenweise oder blockweise übertragen werden.
- **Datenrepräsentation** - Unterschiedliche Encodingschemata von unterschiedlichen Geräten.
- **Fehlerbehandlung** - Art der Fehler sowie die Auftrittshäufigkeit

Im Allgemeinen lassen sich I/O Funktionen mit 3 unterschiedlichen Strategien realisieren:

- **Programmed I/O** - Die einfachste zu Wählende Strategie. Der Prozessor setzt einen I/O Command für einen Prozess ab der dann mittels *busy waiting* auf den Abschluss der Operation wartet bevor er fortfährt.
- **Interrupt-Driven I/O** - Der Task setzt einen I/O Command für einen Prozess ab und hat danach zwei Möglichkeiten. Falls die I/O Instruktion vom Prozess *non-blocking* ist, werden weitere Befehle des ersten Prozesses ausgeführt. Ist die Instruktion aber *blocking* wird der Prozess blockiert und ein anderer vom Scheduler gewählt.
- **Direct Memory Acces (DMA)** - Ein sogenannter **DMA Controller** kontrolliert den Datenaustausch zwischen Hauptspeicher und einem I/O Modul. Der Prozess schickt einen Befehl an das I/O Modul woraufhin dieses autonom Daten zwischen Speicher und I/O Modul kopiert und der Prozess erst nach der Fertigstellung unterbrochen wird. Da kein Kontextwechsle der CPU notwendig ist und Daten quasi parallel mit der CPU-Aktivität transferiert werden können ist dieser Ansatz sehr effizient.

Kriterien für das OS-Design

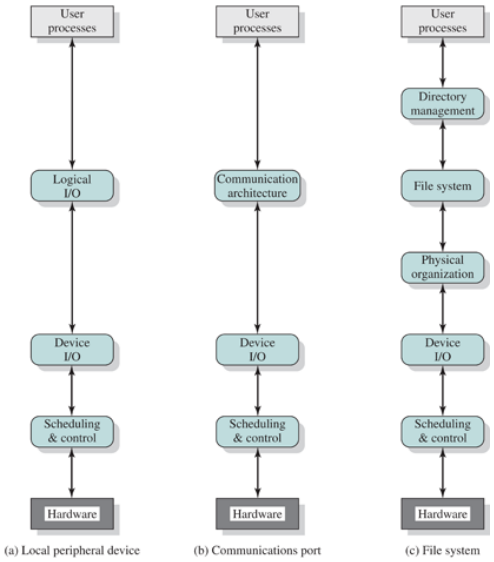
Zwei grundlegende sich widersprechende Ziele die beim Entwerfen eines OS verfolgt werden sind Effizienz und Flexibilität (*Generality*). **Effizienz** ist wichtig da I/O Operationen oft den Bottleneck eines Systems darstellen. Trotz der Größe des Hauptspeichers in modernen Systemen kommt es noch oft vor, dass I/O Operationen nicht mit dem Prozessor mitkommen. Swapping ermöglicht zwar mehr Prozesse bereitzustellen ist jedoch selbst auch eine I/O Operation.

Flexibilität soll für Einfachheit und dadurch geringere Fehlerhäufigkeit sorgen. Dies kann erreicht werden indem einheitliche Schnittstellen für alle Geräte erstellt werden, was sich in der Praxis jedoch als immens schwer herausstellt. Da Effizienz für stark gerätespezifische Lösungen sprich ergibt sich hier der zuvor erwähnte Widerspruch.

Logische Struktur von I/O Funktionen

Die Lösung um diese widersprüchlichen Ziele unter einen Hut zu bekommen ist ein Schichtenmodell mit 3 Ebenen die es ermöglichen ein einheitliches Interface mit gerätespezifischer Ansteuerung zu vereinen.

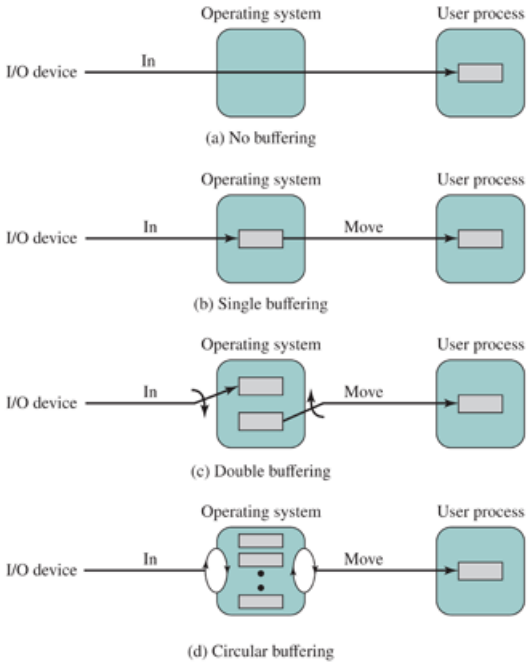
- **Logical I/O** - Die oberste Ebene stellt das Interface dar, welches für alle verbundenen Geräte dieselben Funktionen (`open(), close(), read(), ...`) zur Verfügung stellt. Diese Schicht besteht aus 3 Komponenten:
  - **Directory Management** - Übersetzung von symbolischen Dateinamen in Dateireferenzen.
  - **File System** - Behandelt die logische Struktur und Zugriffsrechte von Daten.
  - **Physical Organisation** - Übersetzung von logischen Referenzen in physische.
- **Device I/O** - Hier werden Operationen in eine Sequenz von I/O-Kontrollkommandos übersetzt werden.
- **Scheduling and Control** - Die gerätespezfische Ebene. Hier werden schlussendlich die I/O Operationen verarbeitet und verwaltet. Auf diesem Layer interagiert die Software mit dem I/O Module, also steuert sie die Gerätehardware an .



Puffern von I/O Anfragen

Der Puffer ist ein Zwischenspeicher zwischen der Applikationsdatenstruktur und der Hardware für Daten beim I/O Transfer. Um zu verhindern, dass I/O Operationen einzeln übertragen werden und bspw. ein Deadlock riskiert wird, kann durch ein Zusammenfassen von low-level Operationen der Overhead reduziert werden. Ein Beispiel hierführ wären einzelne Character die anstatt direkt in den Speicher in den Puffer geschrieben und dann zusammengefasst werden wodurch sie eine einzelne I/O Operation darstellen.

Es gibt auch unterschiedliche Möglichkeiten Buffer einzusetzen. Beim Single Buffering gibt es lediglich einen einzigen Buffer der beschrieben und ausgewertet wird. Beim **Double Buffering** hingegen kommen zwei Buffer zum Einsatz, sodass der eine ausgelesen werden kann während der andere befüllt wird. Die logische Konsequenz hier ist das **Circular Buffering** bei dem noch weitere Buffer genutzt werden.



Disk I/O - Scheduling

Der rasante Geschwindigkeitsanstieg von Prozessor und Hauptspeicher hat dazu geführt, dass diese in der Regel um mehrere Größenordnungen schneller sind als Speicherplattenzugriffe. Da sich dieser Unterschied in der nächsten Zeit nur noch weiter vergrößern wird, benötigt man Ansätze um die Performance zu verbessern. Bei einer rotierenden Speicherplatte gibt es verschiedene Zeiten die beachtet werden müssen:

- **Seek Time** T_S - Mittlere benötigte Zeit um den Lesearm auf die gewünschte Spur zu bewegen
- **Rotational Delay** T_{RD} - Mittlere Zeitverzögerung, bis der Anfang des gesuchten Sektors gefunden wird
- **Transfer Time** T_{TF} - Benötigte Zeit zur Übertragung der Daten
- **Average Acces Time** T_A - Mittelwert der benötigten Zeit für den Datenzugriff

$$T_A = T_S + T_{RD} + T_{TF}$$

Disk Scheduling Strategien

Shortest-Service-Time-First (SSTF) - Hier wird die I/O Anfrage gewählt die die geringste Bewegung des Lesearms von seiner aktuellen Position erfordert. Damit soll also der Prozess mit der geringsten T_S gewählt werden.

SCAN (Elevator Algorithm) - Da SSTF für einige I/O Operationen zu Starvation führen kann, kann stattdessn die SCAN Policy angewandt werden. Wie der Name schon sagt, bewegt sich der Lesearm hier nur in eine Richtung bis die letzte Spur erreicht wurde, und führt auf dem Weg alle aufkommenden Anfragen durch bis dann in die andere Richtung dieselbe Prozedur wiederholt wird.

C-SCAN - Eine Schwäche des Scan Algorithmus ist die hohe Wartezeit für Anfragen die auf den äußersten Spuren zu finden sind. Der C-Scan wirkt dem entgegen indem nur in eine Richtung gearbeitet, und dann auf dem Rückweg eine "Leerfahrt" eingelegt wird bei der keine Anfragen bearbeitet werden.

N-step-SCAN und **FSCAN** - Hier wird die Request Queue in kleinere Subqueues mit der Länge N unterteilt. Diese werden dann mit dem SCAN Algorithmus abgearbeitet. Der **FSCAN** Algorithmus hingegen arbeitet mit zwei Subqueues. Wird ein Scan begonnen ist eine der beiden Queues leer wobei die andere alle Requests beinhaltet. Während dem Scan wird die eine Queue abgearbeitet während in die andere neue aufkommende Requests geschrieben werden.

Disk Cache

Ähnlich zum Memory Cache kann dieses Prinzip auch auf der Speicherplatte eingesetzt werden indem ein Puffer im Hauptspeicher für die Sektoren einer Disk erstellt wird. Will ein Programm auf neue Daten zugreifen sucht das OS zuerst im Cache nach diesen Daten um einen eventuellen I/O Request zu vermeiden. Durch die Geschwindigkeit des Hauptspeichers lässt sich hierdurch die Performance stark verbessern.

Blockersetzungsstrategien hier sind **LRU** oder **LFU**.

Asynchrone und Synchrone I/O

Stellt eine Anwendung eine asynchrone I/O Anfrage kann der Prozess bis zur Erfüllung weitergeführt werden. Diese Form ist um einiges effizienter, jedoch wird ein Mechanismus benötigt um eine Fertigstellung der I/O Anfrage zu signalisieren. Bei der synchronen I/O ist der Prozess blockiert bis die I/O Operation fertiggestellt wird.

Wiederholungsfragen

Was versteht man unter Buffering? Welche Vorteile bietet es, wo liegen seine Grenzen und worauf hat man bei der Verwendung von Puffern bei der Betriebssystemimplementierung zu achten?

Mit Buffering wird der Ansatz beschrieben Daten eines I/O Transfers vor der Übertragung an den Prozessor in einen Buffer zu laden und dann gesammelt an diesen zu übergeben.

Erhöhte Effizienz aufgrund von Entkopplung diverser Anfragen. Als Beispiel dient ein Prozess der Daten vom Sekundärspeicher lesen möchte mit einer bestimmten Größe. Ohne Buffering würde der Prozess den Daten einen virtuellen Speicherraum zuweisen die dann im weiteren Verlauf mit Swapping Entscheidungen kollidieren können und im Hauptspeicher bleiben müssen bis die I/O Übertragung abgeschlossen wird.

Dennoch wird keine Form des Bufferings ermöglichen das I/O Geräte mit Prozessen "mithalten" können.

Beschreiben Sie das Ziel von Disk Scheduling. Nennen Sie drei „intelligente" Disk-Scheduling Algorithmen und beschreiben Sie diese kurz.

Disk Scheduling Strategien helfen dabei Anfragen an die Disk in möglichst geringer Zeit abzuarbeiten, da die Lesegeschwindigkeit stark hinter der Prozessorgeschwindigkeit hinterherhinkt.

SSTF - *Shortest Service Time First* - Der Prozess der die geringste Bewegung des Lesearms verlangt wird gewählt. Starvation ist möglich.

SCAN - Der Lesearm bewegt sich in eine Richtung bis zur letzten Spur und arbeitet unterwegs alle Anfragen ab. Dasselbe wird danach in die andere Richtung gemacht.

C-SCAN - Um die Schwächen des SCAN auszugleichen werden hier nur noch in einer Richtung Anfragen abgearbeitet und in die andere wird eine "Leerfahrt" eingelegt.

N-Step-SCAN - Nur N aufeinanderfolgende Anfragen werden abgearbeitet, damit der Lesearm nicht in einer stark benötigten Region hängen bleibt.

FSCAN - Zwei Queues werden angelegt bei der eine abgearbeitet wird und die andere mit nicht bearbeiteten Anfragen gefüllt wird die in der nächsten Runde abgearbeitet werden.

Mit welcher logischen Struktur des I/O Systems versucht man bei der Realisierung von I/O Funktionen sowohl ein einheitliches Programmierinterface, als auch eine möglichst gerätespezifische Ansteuerung zu erreichen?

Mit einem Schichtenmodell bestehend aus:

- **Logical I/O** - Interface für User Input mit klassischen Funktionen. Kann je nach I/O Gerät in weitere Schichten aufgeteilt werden (Beispielsweise File System)
- **Device I/O** - DIe angeforderten Daten (characters, records) werden in I/O Befehlssequenzen umgewandelt
- **Scheduling & Control** - Auf diesem Layer wird mit dem I/O Gerät direkt interagiert und I/O Operationen werden verwaltet

Was versteht man unter Synchronous bzw. Asynchonous I/O? Beschreiben Sie die beiden Arten, I/O-Operationen durchzuführen.

Bei der synchronen I/O wird der Prozess bis zur Fertigstellung blockiert wohingegen bei der asynchronen I/O der Prozess weiterlaufen kann.

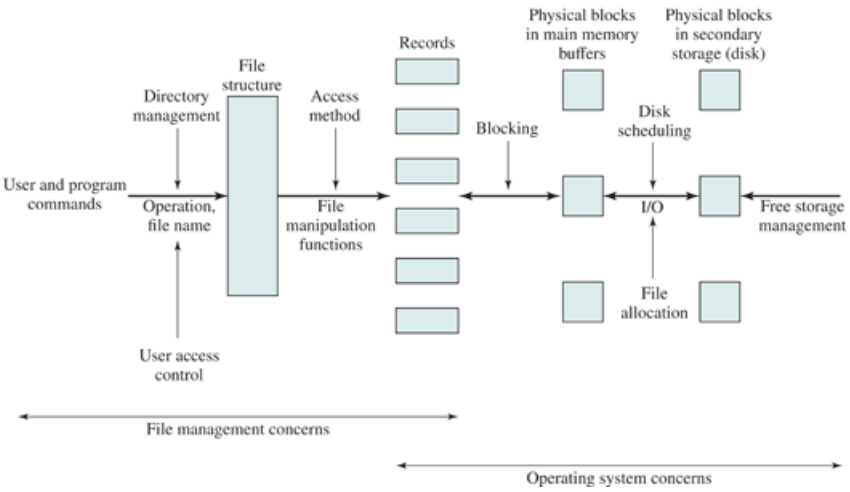
Wie berechnet sich die mittlere Zugriffszeit beim Lesen von Daten von einer mechanischen Festplatte? Geben Sie die charakteristischen Zeitparameter einer Festplatte an. Durch welche Strategien kann das Betriebssystem dazu beitragen, die mittleren Zugriffszeiten auf die Festplatte zu reduzieren?

$$T_A = T_S + T_{RD} + T_{TF}$$

T_S (Seek Time) bezeichnet die benötigte Zeit um den Lesearm zum Ziel zu bewegen. T_{RD} (Rotational Delay) hingegen die Zeitverzögerung um beim gefundenen Sektor zum Anfang zu gelangen. T_{TF} ist die Dauer des Datentransfers. Die Reduktion kann durch intelligente Strategien (C/F-SCAN oder SSTF) oder durch Disk Caching erreicht werden.

File Management

Die Datei bzw. das **File** ist ein zentrales Element vieler Prozesse und zuständig dafür große Informationsmengen zu Speichern. Das besondere an einer File ist die Persistenz, d.h. sie existieren unabhängig von Prozessstart oder Prozesstermination. Das **File System** verwaltet Files und kümmert sich um Namensgebung, Struktur, Lokalisierung und Zugriff sowie um den Schutz (Protection).



Dateiorganisation und Zugriff

Die Ziele der Dateiorganisation lassen sich zusammenfassen als:

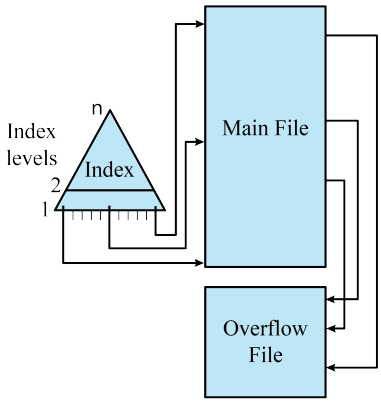
- Kurze Zugriffszeiten
- Leichte Veränderbarkeit
- Geringer Platzverbrauch
- Einfache Wartbarkeit
- Zuverlässigkeit

Die relative Wichtigkeit dieser Kriterien unterscheidet sich selbstverständlich jeweils für die Anwendungen die die Datei nutzen werden. Beispielsweise ist eine leichte Veränderbarkeit kein Kriterium für Dateien die auf einer CD gespeichert sind.

Die Dateiorganisation kann auf verschiedene Arten erfolgen:

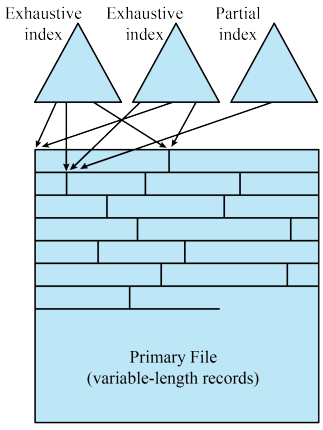
- **Unstructured sequence of bytes**
- **Pile** – Records variabler Länge werden in chronologischer Form (in der Reihenfolge in der sie ankommen) in die Datei geschrieben. Einzelne Records können unterschiedliche Felder besitzen die selbsterklärend sein sollten.

- **Sequential File** - Die häufigste Form der Dateiorganisation. In dieser Form wird ein fixes Format für Records genutzt mit fixer Länge, Reihenfolge und Anzahl an Feldern. Eine Besonderheit ist, dass das erste Feld das **Key Field** ist welches den Record eindeutig identifizierbar macht. Bei interaktiven Anwendungen in denen Anfragen und/oder Updates einzelner Records vorkommen ist die Performance einer solchen File tendenziell sehr schlecht.
- **Indexed Sequential File** - Um die Performanceprobleme von Sequential Files zu umgehen und den Suchaufwand drastisch zu reduzieren wird zusätzlich zur Sequential File (Main File) ein Index sowie eine **Overflow File** hinzugefügt. Die Index File ist im einfachsten Fall ein weiterer Record der aus zwei Feldern besteht: einem *key field* sowie einen Pointer in die **Main File**. Werden neue Records hinzugefügt kommen diese zunächst in die Overflow File.



Indexed Sequential File

- **Indexed File** - Anstatt die Suche nur nach einem einzigen Suchfeld zu ermöglichen kann hier eine Suche nach mehreren Kriterien erfolgen.



Indexed File

- **Hash File** - Die Zugriffszeit kann stark verkürzt werden indem die Keys über eine Hashfunktion gemappt werden. So kann jeder Block direkt angesteuert werden.

File Typen

Man unterscheidet zwischen **Regular Files** (ASCII, binary), Directories, Character Special Files (Repräsentation sequentieller I/O Geräte) und Block Special Files (Repräsentation von Platten). Das Betriebsystem muss zumindest ein eigenes ausführbares Format interpretieren können. Erkannt werden solche Dateien meistens indem in den Header eine besondere Bitstruktur, die sogenannte *magic number*, hineingeschrieben wird.

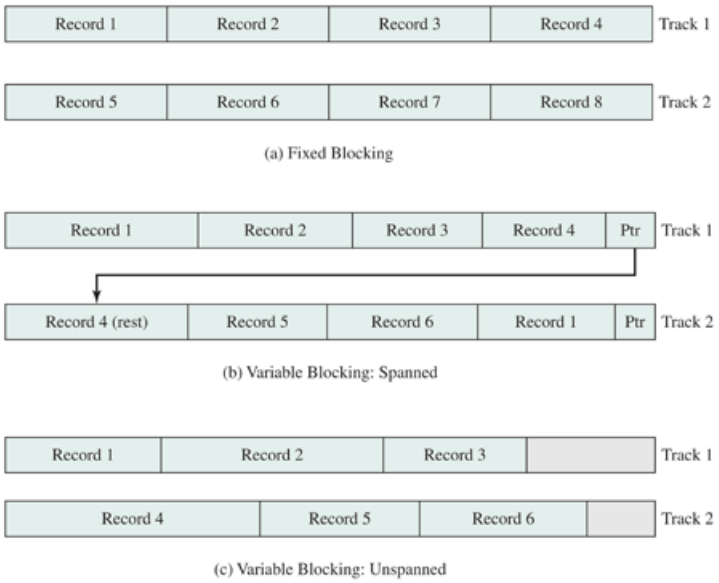
In den **File Attributen** befinden sich Metadaten die eine große Anzahl an Informationen bieten wie den Zeitpunkt der Dateierstellung, Owner, Protection, Size, etc. Organisiert sind die Daten in einer Directory die als hierarchische Baumstruktur aufgebaut ist.

Im Speicher ist eine Datei nichts weiter als eine Sammlung von Speicherblöcken. Bei der Implementierung von Dateien gibt es unterschiedliche Strategien um Blöcke zu allozieren:

- **Contiguous Allocation** - Das klassische Modell in dem eine Datei einen aneinander angrenzenden Bereich von Blöcken belegt. Dies führt zu guter Performance ist jedoch bei häufigen Änderungen nicht empfehlenswert da es zu externer Fragmentierung kommt.
- **Chained Allocation** - Hier werden einzelne Blöcke belegt die über Pointer miteinander verkettet sind und dementsprechend auf beliebige Stellen geschrieben werden können. Dadurch kommt es zwar nicht zu externer Fragmentierung aber es kann zu langsamen Zugriffen bei Random Access und verringerten Nutzdaten pro Block ($< 2^n$) kommen da der Pointer selbst Speicher benötigt.
- **Indexed Allocation** - Um das Problem der verringerten Nutzdaten pro Block zu reduzieren werden die Pointer in einer extra Tabelle im Speicher (FAT, File Allocation Table) gespeichert. Dadurch ist direkter und sequentieller Zugriff gut unterstützt, jedoch braucht die FAT viel Platz im Arbeitsspeicher.
- **I-Nodes** - Eine besondere Datenstruktur mit fixer Größe für jedes File welches Fileattribute und Referenzen auf die Blöcke des Files enthält. Abgesehen von direkten Verweisen auf die Daten kann bei entsprechender Größe eine gewisse Anzahl von indirekten Verweisen genutzt werden, welche wiederum auf Filetabellen zeigen (die selbst auf solche Tabellen zeigen können).

Record Blocking

Damit I/O Operationen durchgeführt werden können müssen Records als Blocks organisiert werden. Hier ergeben sich einige Fragestellungen in Bezug auf die Größe der Blocks. Beim **Fixed Blocking** werden Records mit fixer Länge genutzt und eine Anzahl an Records in einem solchen Block gespeichert. Der Nachteil hier ist, dass die Speichernutzung sehr ineffizient sein kann. Beim **Variable-length Spanned Blocking** hingegen sind die Recordlängen variabel und kein Speicherplatz wird verschwendet. Daher müssen manche Records über zwei Blocks gehen, die mit einem Pointer zueinander verbunden sind. Alternativ kann auch **Variable Unspanned Blocking** genutzt werden bei dem Records jedoch nicht auf mehrere Blocks aufgeteilt werden können, wodurch es wiederum zu ungenutztem Speicher kommt.



Directory Implementierung

Um Dateien zu finden wird zuerst das Root Directory lokalisiert. Danach wird der Pfadname interpretiert. Das Root Directory kann sich zum Beispiel an einer fixen Position vom Partitionsanfang aus befinden. In Unix-Systemen werden im Superblock die I-Nodes gespeichert. Die erste solche I-Node verweist auf das Root Directory.

Unter Windows enthält der Boot Sector Informationen über die Adresse der sogenannten Master File Table (MFT) wo sämtliche wichtigen Information über Files und Directories des NTFS File Systems gespeichert sind.

Wichtige Begriffe

Feld (Datei) - Das Grundelement von Daten in dem gilt, dass ein Feld exakt einen Wert beinhaltet (Beispielsweise Name, Datum, numerischer Wert, etc.).

Record - Ein Record ist eine Sammlung von verwandten Feldern die von einem Programm als Einheit behandelt werden können. Eine Datei ist dementsprechend eine Sammlung von verwandten Records.

Wiederholungsfragen

Bei der Realisierung von Dateisystemen gibt es verschiedene Möglichkeiten, um die zu einer Datei gehörenden Datenblöcke zu organisieren bzw. auffindbar zu machen (BlockAllokierung). Nennen Sie vier verschiedene Strategien zur Block-Allokierung von Dateien und beschreiben Sie diese mit ihren Vor- und Nachteilen.

- **Pile** - Daten werden aneinandergereiht in der Reihenfolge in der sie ankommen. Zum Finden eines Records muss der gesammte Pile durchgegangen werden.
- **Sequential File** - Records werden in Blocks gleicher Länge gespeichert bei dem jedes Feld eine Funktion übernimmt (erstes Feld ist **Key Field**). Bei der sequentiellen Abarbeitung gut geeignet, nicht so für Vergrößerung bzw. Verkleinerung sehr großer/kleiner Files.
- **Indexed Sequential File** - Ein Index wird abgespeichert um schnellere Zugriffe zu ermöglichen.
- **Indexed File** - Anstatt nur über einen, kann hier über mehrere Indizes zugegriffen werden.
- **Direct File** - Zugriff erfolgt über Hashen des Key Fields.

Was versteht man unter einer File Allocation Table? Wie ist diese organisiert?

Eine extra Tabelle die im Speicher angelegt wird in der die Aufteilung und Speicherorte der Files in der Disk gespeichert werden. Damit ist unabhängig von der Art des Zugriffs ein schnelles Finden möglich, benötigt aber viel Speicherplatz.

Wie ist ein I-Node aufgebaut? Welche Informationen enthält er?

In Linux Systemen enthält jede File eine I-Node in der zum einen Attribute und Informationen gespeichert werden die Auskunft über Permissions, Owner, Größe, etc. geben sowie Referenzen die auf die Daten der Datei im Speicher zeigen. Da ein I-Node eine fixe Größe hat sind die Referenzen auch begrenzt. Gelöst wird dieses Problem indem einfach, doppelt oder sogar dreifach indirekte Referenzen genutzt werden (I-Node Pointer zeigt auf Filetabelle die wiederum auf weitere Filetabelle zeigt, etc.)

Beschreiben Sie das typische Layout einer Disk bzw. eines Filesystems. Welche Rolle spielen die einzelnen Teile beim Hochfahren des Betriebssystems?

Das Filesystem wird in unabhängige Partitionen unterteilt wobei sich im ersten Sektor der Disk der Master Boot Record befindet der zum Hochfahren des Systems benötigt wird indem dieser beim Start interpretiert wird. Darauf folgt die Partition Table in der Informationen über alle in der Platte existierenden Partitionen enthält wodurch das OS weiß aus welcher Partition gestartet wird.

Security

Allgemein geht es beim Thema IT-Sicherheit um die Wahrung der Vertraulichkeit, Integrität und Verfügbarkeit von Informationen. Diese drei Bereiche werden auch als **CIA-Triad** (Confidentiality, Integrity and Availability) bezeichnet.

Sicherheitsziel	Beschreibung	Bedrohung
Confidentiality	Geheimhaltung der Daten muss gewahrt werden. *Wer darf Daten ansehen?*	**Exposure/Interception** - Nicht autorisierte Lesezugriffe
Integrity	Daten sollen nicht einfach manipuliert werden können. *Wer darf Daten verändern?*	**Modification/Fabrication** - Verletzung der Integrität durch Änderung oder Generierung von Daten
Availability	Verfügbarkeit der Daten, wenn benötigt.	**DoS Angriffe** - Unterbrechung des Services durch Überlastung

Zusätzliche Ziele sind

- **Authenticity** - Korrektheit der Identität
- **Accountability** - Nachvollziehbarkeit

Bedrohungen werden zusätzlich in **aktive** (Manipulation/Modifikation von System und Daten) und **passive** (Abhören ohne Wissen des/der Betroffenen) Bedrohungen unterteilt.

Intrusion

Dieser Angriff hat als Ziel die Verschaffung des Zugangs zu einem Systems oder die nicht zugelassene Erhöhung der Privilegien auf diesem (*privilege escalation*). Die Methoden variieren hier und reichen von der Ausnutzung einer Sicherheitslücke bis hin zur Aneignung eines Passworts (beispielsweise über *Social Engineering*).

Eindringlinge (*Intruder*) werden je nach Szenario in unterschiedliche Kategorien mit immer größer werdendem technischen Know-How unterteilt. Die unterste Kategorie sind Gelegenheitsattacken von Usern die kein tiefgründiges Wissen zu den genutzten Tools besitzen (*script kiddies*). Die nächste Stufe bezeichnet gelegentliche Attacken technisch versierter Insider, gefolgt von gezielten Versuchen der Bereicherung. Die Spitzenkategorie ist dann die Industrie- und Militärspionage in der es teilweise um

geopolitische Themen geht.

Malware

Es gibt unterschiedliche Arten von Malware:

- **Virus** - In Programm versteckter Schadcode, der sich autonom in andere Programme kopiert (Key Logger, DDoS, ...)
- **Worm** - Selbstreplizierendes Programm welches sich autonom über andere Rechner verbreitet
- **Trojan Horse** - Programm mit gewünschter Funktionalität welches versteckten Schadcode beinhaltet
- **Logic Bomb** - Programmstück, dass sich beim Auftreten einer Bedingung selbst aktiviert
- **Trapdoor (Backdoor)** - Geheimer Einstiegspunkt mit dem Zugriffskontrolle umgangen werden kann

Angriffsmethoden

- **Auslesen von Speicher** - Zugewiesener Speicher der möglicherweise sensible Daten enthält und nicht gelöscht wurde.
- **Aufruf unerlaubter System Calls** - Aufruf von Calls mit sinnlosen Parametern kann die Software crashen lassen
- **Fälschung der Login-Routine** - Beispielsweise eine gefälschte Loginmaske die zum abgreifen der Userdaten genutzt wird
- **Social Engineering** - Bestechung, Phishing, ...

Designprinzipien für Security

Einige Prinzipien müssen verfolgt werden um weniger Angriffsfläche zu bieten:

- **Open Design** - Die Sicherheit eines Systems darf nie von der Geheimhaltung ("Security by Obscurity") abhängig sein und soll möglichst offengelegt werden. So können Schwachstellen im Code schnell gefunden und gefixt werden.
- **Least Privilege** - Jedem User sollten nur soviele Berechtigungen zukommen wie wirklich notwendig sind. Per Default sollten keine Berechtigungen existieren.
- **Economy of Mechanisms** - Sicherheitsmechanismen sollten einfach und überschaubar bleiben um Fehler zu vermeiden.
- **Acceptability** - Sicherheitsmechanismen sind nur sinnvoll wenn sie auch von den Usern angenommen werden. Die sicherste Methode ist wertlos wenn sie zu umständlich in der alltäglichen Nutzung ist (bspw. unrealistische Passwortvorgaben mit Zahlen, Sonderzeichen und Hieroglyphen führen zu Post-its am Monitor).
- **Complete Mediation** - Sicherheitsmechanismen sollen in allen Betriebsmodi verfügbar sein.

Protection Domains

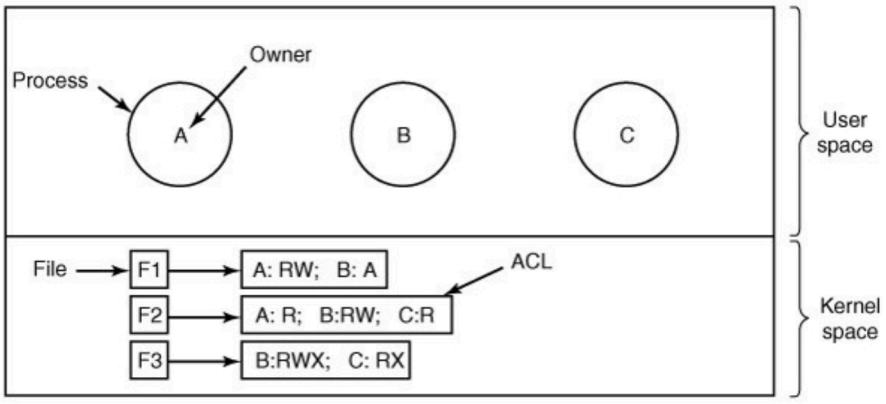
In einem System stellt ein **Objekt** alles dar wofür der Zugriff kontrolliert wird. Beispiele hierfür sind Dateien, Programme, Speichersegmente sowie Softwareobjekte (Java Objects). Eine **Protection Domain** beschreibt eine menge von Paaren aus einem Objekt und den dazugehörigen Rechten. Die Domain kann sich beispielsweise beim Wechsel vom User in den Kernel Mode ändern.

Die Zugriffsrechte können auch in einer **Access Matrix** gespeichert werden. In dieser gilt, dass ein Prozess in Domain D_i die Operationen in der Matrix auf Objekt O_j ausführen darf.

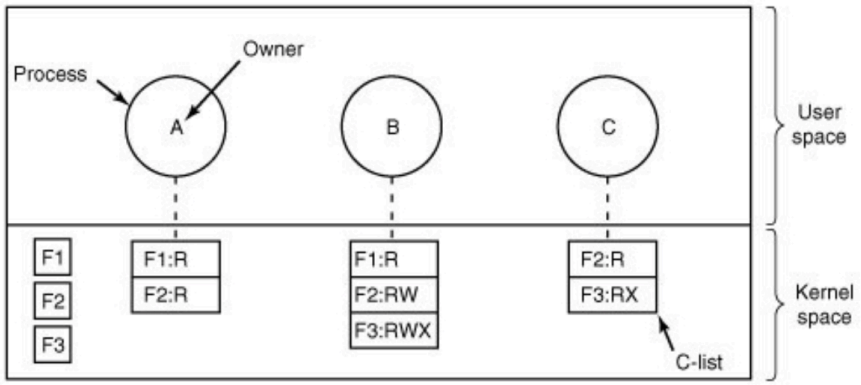
object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Eine wichtige Unterscheidung muss zwischen **Mechanismus** und **Policy** gemacht werden. Wird beim Mechanismus von den Methoden gesprochen die vom OS zur Verfügung gestellt werden (Access Matrix, Regeln, ...) spricht man bei der Policy davon, wie die Mechanismen eingesetzt werden (Wer darf welches Objekt wie verwenden?).

Realisiert werden kann die Access Matrix über **Access Control Lists** indem die Matrix entlang der Spalten zerlegt wird und die Zugriffsrechte bei den Objekten gespeichert werden. Der Vorteil hier ist ein leichtes Ändern der Zugriffsrechte von Objekten.



Alternativ kann die Access Matrix auch über **Capability Lists** realisiert werden. Pro Prozess gibt es eine Liste von Objekten und den erlaubten Zugriffsoperationen in sogenannten Tickets. Ein Vorteil ist die Möglichkeit der Weitergabe bzw. Vererbung dieser Tickets an andere Prozesse.



Bell and LaPadula's Model

Das Modell stellt Regeln für den Informationsfluss bereit. Zum einen gibt es eine Hierarchie von Security Classifications (SC) für Objects und Subjects (top secret, secret, public) und zum anderen Operationen von Subjects auf Objects (read-only, append (without reading), execute, read-write).

Dazu gehören eine Reihe von **Security Axioms** die das Verhältnis zwischen Subjekt und Objekt klarifizieren.

Für ein **Read** muss gelten $SC(S) \geq SC(O)$ - Man kann nur auf Objekte in der selben SC-Stufe oder darunter zugreifen (*"no read up"*).

Für ein **Append** muss gelten $SC(S) \leq SC(O)$ - Beim Schreiben der Information muss die Sicherheitsstufe des Subjekts kleiner der des Objekts sein um Leaks zu vermeiden (*no write down*).

Für ein **Read-Write** muss folglich gelten $SC(S) = SC(O)$ - Schreiblesezugriff darf nur bei entsprechender Sicherheitsstufe möglich sein.

Wiederholungsfragen

Nennen Sie die drei Kategorien von Security Threats und beschreiben Sie diese. Geben Sie für jede Kategorie an, welches grundlegende Security-Ziel dadurch bedroht wird.

- **Kategorie:** Exposure/Interception - **Betroffenes Security Ziel:** Confidentiality - Eine Öffentlichmachung vertraulicher Daten
- **Kategorie:** Modification/Fabrication - **Betroffenes Security Ziel:** Integrity - Daten wurden unerlaubterweise abgeändert oder gelöscht
- **Kategorie:** Denial of Service - **Betroffenes Security Ziel:** Availability - Ein Zugriff auf benötigte Daten ist nicht mehr möglich.

Was beschreibt das Modell von Bell und LaPadula? Geben Sie die vom Modell geforderten Eigenschaften an.

Das Modell beschreibt Regeln für den Informationsfluss indem es Security Classifications für Subjekte (die zugreifende Entität) und Objekte (die zugegriffene Entität) verteilt und das Verhältnis zwischen diesen beiden regelt.

No read up - Ein Subjekt kann nur Objekte der gleichen oder geringeren Sicherheitsstufe lesen.

No write down - Werden Daten an ein Objekt angehängt muss die Sicherheitsstufe des Subjektes kleiner sein als jene des Objektes um Leaks zu vermeiden.

Read Write darf dementsprechend nur bei exakt gleicher Sicherheitsstufe erfolgen.

Erklären Sie die Begriffe Access Control List und Capability List. Wozu und wie werden diese verwendet?

In der **ACL** werden die Zugriffsrechte bei den Objekten selbst gespeichert wodurch diese leicht geändert werden können. Bei der **CL** hingegen wird pro Prozess eine Liste der Objekte und der dazugehörigen erlaubten Operationen erstellt mittels Tickets. Diese können weitergegeben und vererbt werden und sind verschlüsselt.

Nennen Sie Design Prinzipien für die Konstruktion von sicheren Systemen. Geben Sie für jede Regel ein Beispiel an.

Open Design - Sicherheit darf nie von der Geheimhaltung der Systeme und Mechanismen abhängen ("Security by obscurity")

Default Settings - Per Default sollten Benutzer keine Rechte haben

Least Privilege - Werden dann Rechte zugewiesen, sollen diese so klein wie möglich ausfallen

Economy of Mechanisms - Sicherheitsmechanismen sollen einfach gehalten werden zur Vermeidung von Fehlern

Acceptability - Das sicherste System ist nutzlos wenn es nicht von den Nutzenden angenommen wird

Complete Mediation - Zugriffe auf Ressourcen soll in jeder Situation kontrolliert sein

Check Privileges - Berechtigungen sollen periodisch überprüft und nicht als fixiert angenommen werden

Beschreiben Sie das Prinzip einer Sicherheitsattacke durch Stack/Buffer Overflow. Wodurch kann man sich bei der Implementierung eines Betriebssystems vor einen solchen Angriff schützen?

Bei einem Buffer Overflow wird ein Fehler ausgenutzt der die Grenzen des Arrays nicht überprüft und es dem Angreifer ermöglicht in angrenzende Speicherbereiche zu schreiben. Beispielsweise kann in den Stack eine neue Return Adresse geschrieben werden um Schadcode ausführen zu können.

Schützen kann man sich indem sichere Funktionen genutzt werden die zu jeder Zeit die Eingabelänge überprüfen.