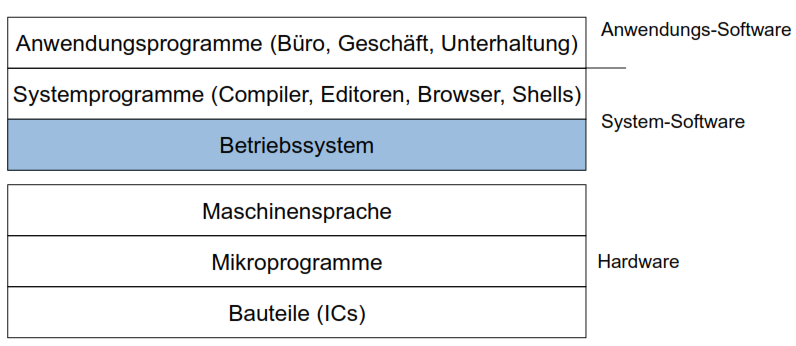
Zusammenfassung Betriebssysteme FS 17

# Entwicklung und Typen

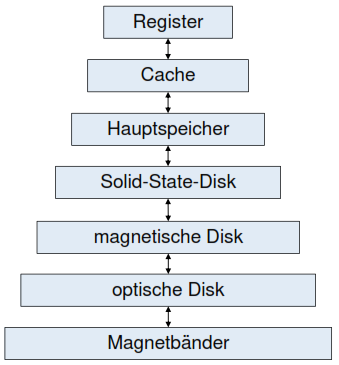
### Definition Betriebssystem

Nach dem internationalen Standard ist ein **Betriebssystem** die Programme eines digitalen Rechensystemes, die zusammen mit der Rechenanlage die Basis der Betriebsart legen und insbesondere die Abwicklung von Programmen steuern und überwachen.

Das Betriebssystem ist das Bindeglied zwischen Computer-Benutzer und Computer-Hardware. Es dient der bequemen Bedienung eines Computers und sorgt durch Vermeidung von Überlastsituationen oder Untätigkeitszeiten für eine effiziente Ausnutzung der Computer-Hardware.



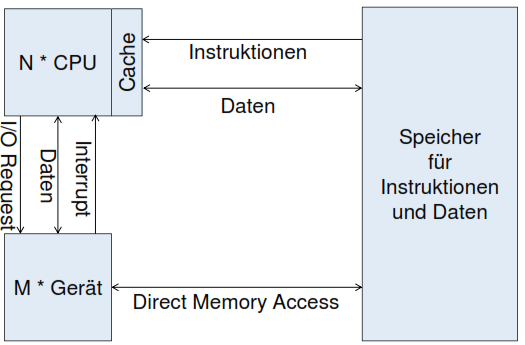
### Speicherhierarchie



Dabei nimmt der Speicherplatz nach unten zu.

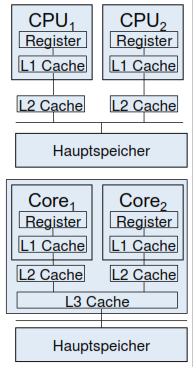
## Architektur von Rechnersystemen

### Einprozessorsysteme



Hier können Geräte autonom ohne Einfluss der CPU Daten lesen/schreiben und dann erst die CPU benachrichtigen, die derweil andere Sachen gemacht hat.

### Mehrprozessorsysteme



Ein Mehrprozessorsystem hat mehr als einen Prozessor. Durch einen gemeinsamen Speicher und einen gemeinsamen Takt geschieht eine enge Systemkopplung, beziehungsweise Synchronisation.

Dabei wird zwischen zwei Typen unterschieden. Beim **symmetrischen Multiprocessing** führen die Prozessoren eine identische Kopie des Betriebssystems aus. Ein Problem hierbei ist gleichmässige Auslastung durch Benutzung gemeinsamer Datenstrukturen.   
Beim **asymmetrischen Multiprocessing** werden gewisse Aufgaben (Tasks) den Prozessoren zugeteilt. Dabei plant der Master die Tasks für die Slave-Prozessoren (Scheduling).

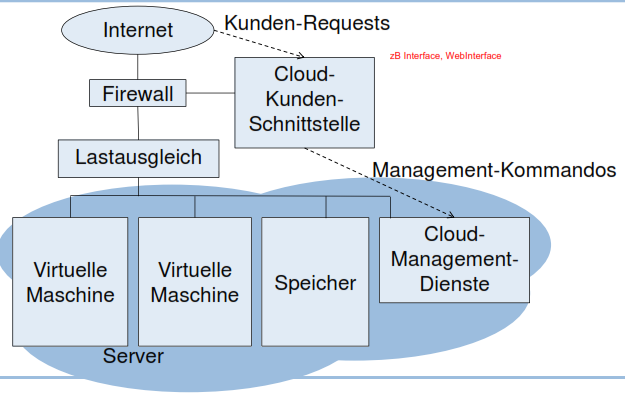
## Cluster-Systeme

Ein **Cluster-System** ist die Sammlung von eigenständingen Rechnern zur Durchführung von Berechnungen. Eine typische Definition ist das Teilen von gemeinsamem Speicher und die Verbindung über ein lokales Netz. Durch die hohe Überwachung und Übernahme von Anwendungen (Cluster-Software) entsteht eine hohe Verfügbarkeit. Dabei kann zwischen **asymmetrischem Clustering** (Ein Rechner ist im Hot-Standby-Modus, in dem er den anderen Rechner überwacht und allenfalls anspringt, und überwacht so den anderen, verarbeitenden Rechner) und **symmetrischen Clustering** (zwei oder mehr sich gegenseitig überwachende Rechner verarbeiten Anwendungen).

## Verteilte Systeme

Hierbei werden die Berechnungen auf mehrere (heterogene) Rechnersysteme verteilt, die loose über Kommunikationsnetze gekoppelt sind. Vorteile davon sind Lastausgleich und Vermeidung von Redundanz.

## Cloud Computing

Hier sind die Dienstleistungen auf einen Server ausgelagert, worüber der Kunde über eine Schnittstelle zugreifen kann.   


Dabei können diese Dienstleistungen nur im Internet verfügbar sein, nur für interne Benutzer oder eine Mischung aus diesen Varianten. Dabei kann auch zwischen Software as Service (Verfügbarkeit von Anwendungen), Platform as a Service (Verfügbarkeit von Middleware, zB Datenbank-Server) und Infrastructure as a Service (Bereitstellung von virtuellen Servern und Speicher) unterschieden werden.

## Sichtweisen auf ein Betriebssystem

Bei den Sichtweisen wird für üblich zwischen der Anwendersicht (top-down), möglichst einfaches Benutzen des Computers und Maximieren der Leistung und der Systemsicht (bottom-up), Robustes Ausführen und Sicherheit unterschieden.

### Abstrakte Maschine

Ein Betriebssystem stellt eine virtuelle Ablaufumgebung bereit. So wird die Hardware vor den Anwendungsprogrammen verdeckt, diese haben also keinen direkten Zugriff auf die Festplatte. Dabei wird eine schönere, einfachere, abstrahierte Schnittstelle angeboten.

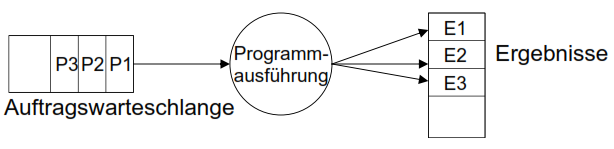
### Ressourcenverwaltung

Ein Computersystem besteht aus verschiedenen Komponenten, die vom Betriebssystem verwaltet werden.

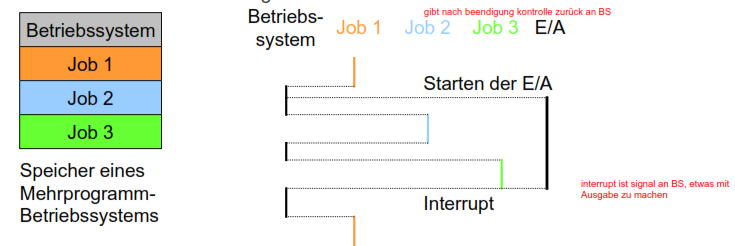
## Betriebssystemtypen

### Mainframe-Systeme

#### Stapelverarbeitung (Batch-Systeme)

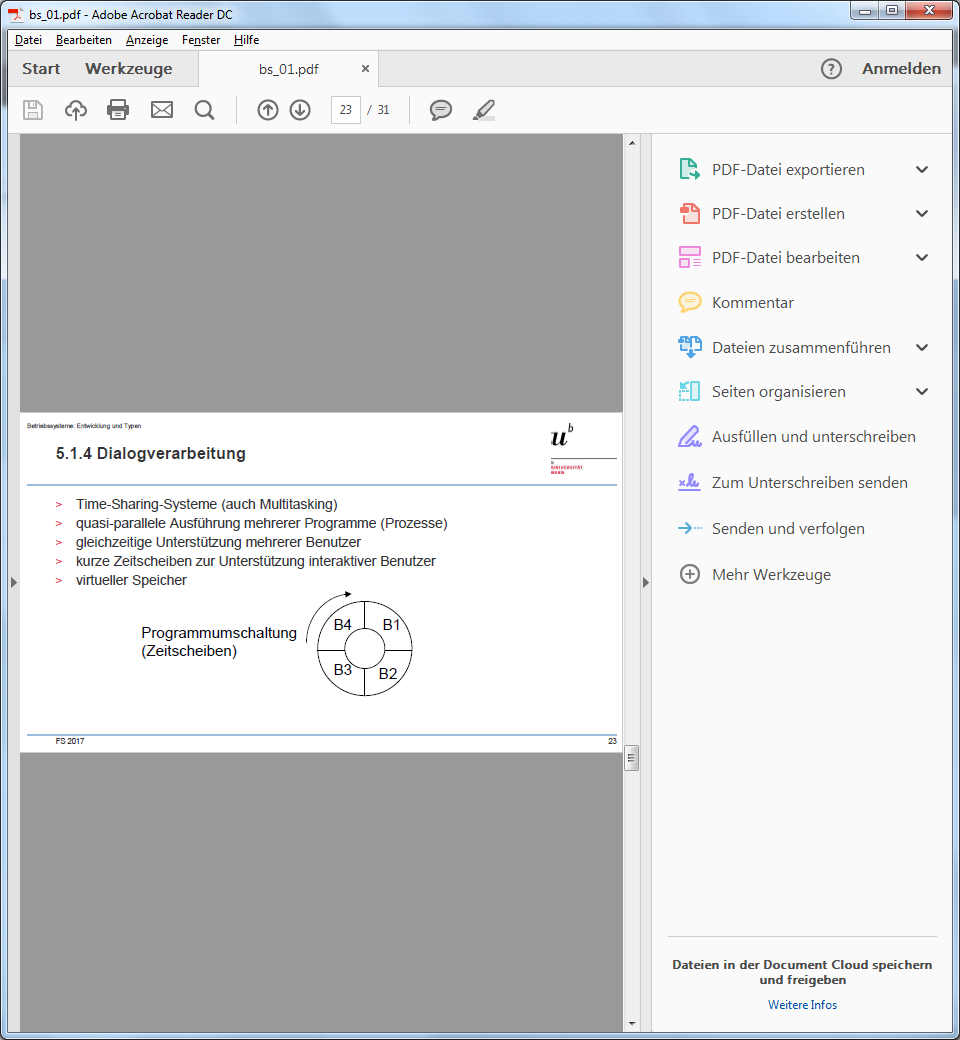
Die auszuführenden Programme werden auf ein Band geschrieben. Der Operatpr lädt dann ein spezielles Programm, das die Programme einliest und sequenziell ausführt. Die Ereignisse werden dann protokolliert.   


#### Mehrprogrammbetrieb



#### Dialogverarbeitung

Das ist ein Time-Sharing-System. Das heisst, nach einer gewissen Zeit wird die Kontrolle über die CPU abgeben. So entsteht der Eindruck von Gleichzeitigkeit.



### Server-Betriebssysteme

Das sind grössere PC, Arbeitsplatzrechner oder Mainframes, die mehrere Nutzer über ein Netz bedienen und dabei Soft- und Hardwareressourcen wie Drucker, Datei-, oder Webdienste bereitstellen.

### Multiprozessor-Betriebssysteme

Das ist eine Variante der Server-Betriebssysteme, die spezielle Funktionen für Kommunikation, Konnektivität und Konsistenz bereitstellt.

### PC-Betriebssysteme

Ein PC wird meist nur von einem einzigen Benutzer zu einem Zeitpunkt genutzt. Dabei werden viele Ein-/Ausgabe-Geräte unterstützt. Für diese Art Betriebssysteme sind einfache Benutzeroberflächen notwendig.

### Handheld-Computer-Betriebssysteme

Das sind zB Mobiltelefone. Dabei gelten spezielle Anforderungen wegen den begrenzten Ressourcen. Die Verarbeitung ist dabei ereignisgesteuert.

### Eingebettete Betriebssysteme

Das sind Betriebssysteme für kleine Geräte wie Mikrocontroller. Dabei gelten geringere Sicherheitsanforderungen, da kein Herunterladen von unzulässiger Software möglich ist.

### Sensor-Betriebssysteme

Sensoren sind kleine, batteriebetriebene Rechner.

### Echtzeit-Betriebssysteme

Die Verarbeitung geschieht innerhalb fester Zeitschranken. Durch diese Zeitschranken geschieht die Umschaltung zwischen Programmen, beziehungsweise durch das Betriebssystem. Dabei kann zwischen **strikten Echtzeitsystemen (hard real-time systems)** bei sehr harten Zeitforderungen und **weichen Echtzeitsystemen (soft real-time systems)** bei zwar Priorisierung für Realzeitaufgaben, aber seltenes Verfehlen der Zeitanforderungen akzeptabel unterschieden werden.

# Aufgaben und Strukturen

## Systemkomponenten

Die **Prozessverwaltung** ist zuständig für das Erzeugen, Löschen, Ausführen, Anhalten und Weiterführen von Prozessen. Zusätzlich kümmert es sich um die Synchronistation von diesen und behandelt Verklemmungen.

Die **Hauptspeicherverwaltung** ist zuständig für das Belegen und Freigeben von Hauptspeicher. Ausserdem ordnet es den Prozessen Hauptspeicher zu.

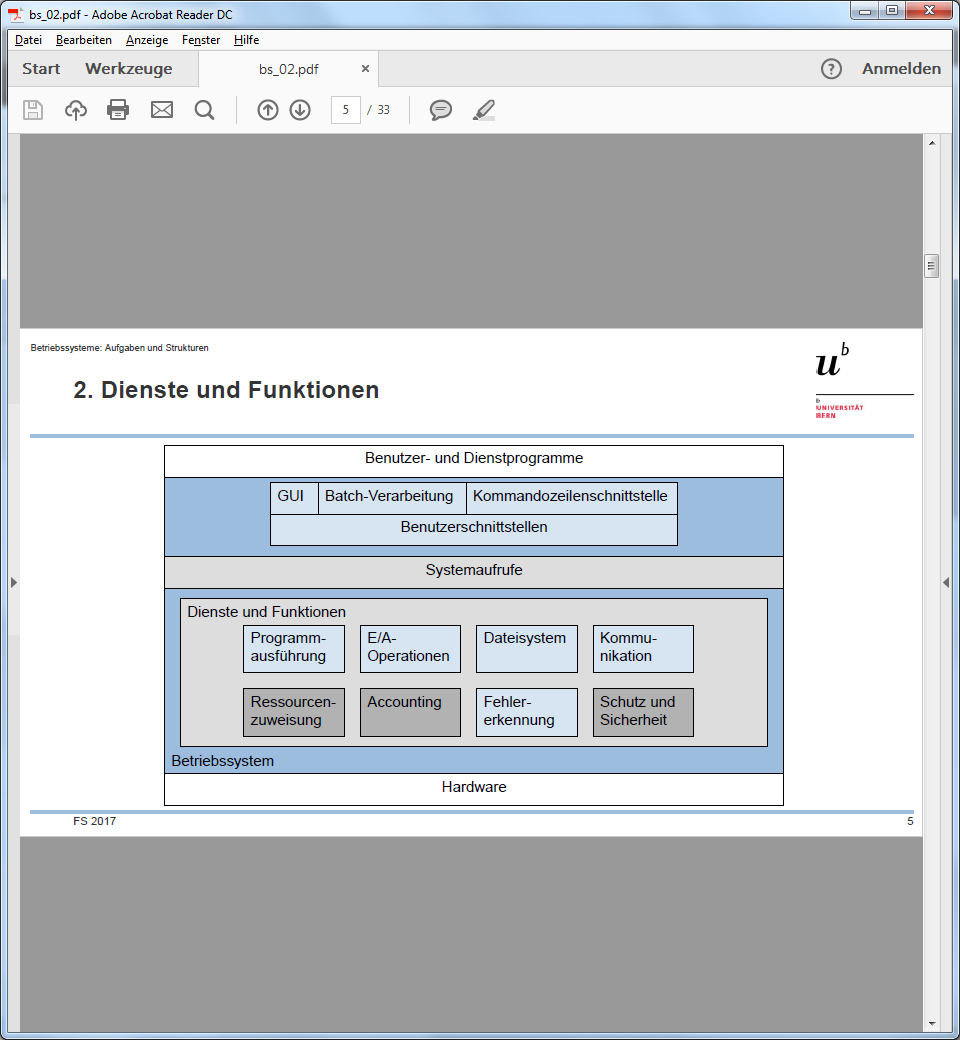
Das **Ein-/Ausgabe-System** ist zuständig für Speicherverwaltung. Ausserdem stellt es eine allgemeine Gerätetreiberschnittstelle dar und stellt für spezielle Hardware-Geräte die benötigten Treiber zur Verfügung.

Die **Sekundärspeicherverwaltung** ist eine Dateisystem-Verwaltung auf dem Sekundärspeicher und eine Massenspeicherverwaltung (Auslagern auf Sekundärspeicher, Zuweisen von Speicherplatz, Scheduling bei Festplatten). Ausserdem gehört das Caching dazu.

Die **Schutz- und Sicherheitssysteme** regeln die Zugriffskontrolle auf System- oder Benutzerressourcen.

Ebenfalls Systemkomponenten sind der **Kommando-Interpreter, grafische Benutzerschnittstellen** und **Kommunikationssysteme.**

## Dienste und Funktionen



### Dienste

Dienste dienen für Programme und Benutzer und zur Vereinfachung der Programmierung. Das sind zB die Benutzerschnittstellen, Programmausführung, Dateisystem-Manipulationen oder Fehlererkennung.

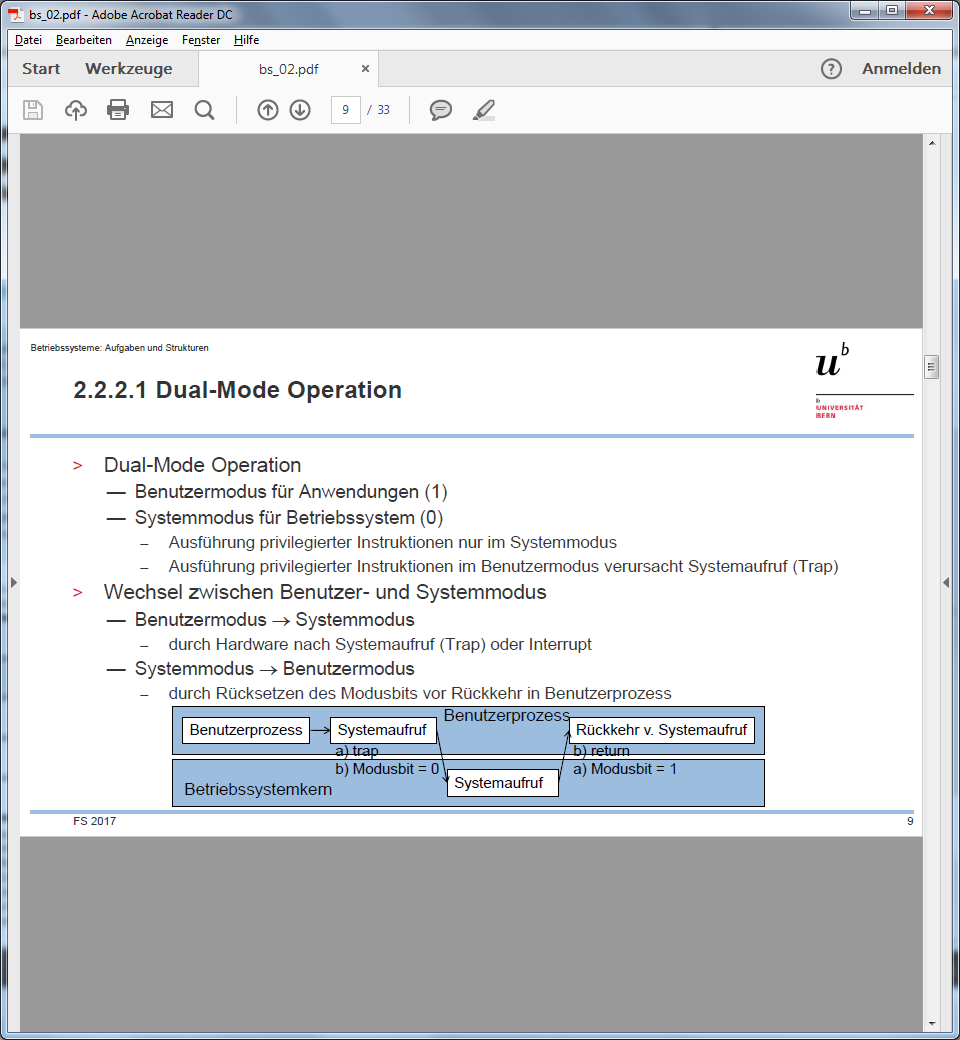
### Funktionen

Funktionen dienen zur effizienten Ausführung des Systems. Das sind zB Zuweisung von Ressourcen, Accounting, Schutzmechanismen oder Sicherheitsmechanismen.

#### Schutzmechanismen

Die Fehererkennung geschieht häufig durch die Hardware, die Fehlerbehandlung durch das Betriebssystem.

Bei **Dual-Mode Operationen** gibt es den Benutzermodus für Anwendungen und den Systemmodus für das Betriebssystem. Dabei können privilegierte Instruktionen nur im Systemmodus ausgeführt werden. Wird versucht, eine solche privilegierte Instruktion im Benutzermodus auszuführen wird ein Trap (Systemaufruf) ausgelöst. Dann wird durch das Setzen des Modusbits auf 0 in den Systemmodus gewechselt. Wenn die Instruktion ausgeführt ist, wird durch das Setzen des Modusbits auf 1 wieder zurückgewechselt. Das Gleiche gilt bei Interrupts.



Beim **E/A-Schutz** sind die E(ingabe)/A(usgabe)-Instruktionen privilegiert und können nur vom Betriebssystem aufgerufen werden. Wollen also Anwendungen E/A-Operationen ausführen müssen sie Systemaufrufe tätigen.

Der **Speicherschutz** schützt die Prozesse untereinander vor inkorrektem Speicherzugriff. Dabei wird der zulässige Speicherbereich durch ein Basis- (kleinste Adresse) und ein Limit-Register (Grösse des Speicherbereichs) beschrieben. Der Vergleich der Speicheradressen mit den Registern geschieht durch die CPU Hardware.

Der **CPU-Schutz** verhindert die Monopolisierung der CPU durch eine Anwendung. Dabei löst das Betriebssystem nach einer bestimmten Zeit ein Interrupt aus und übergibt danach die Kontrolle der CPU an eine andere Anwendung.

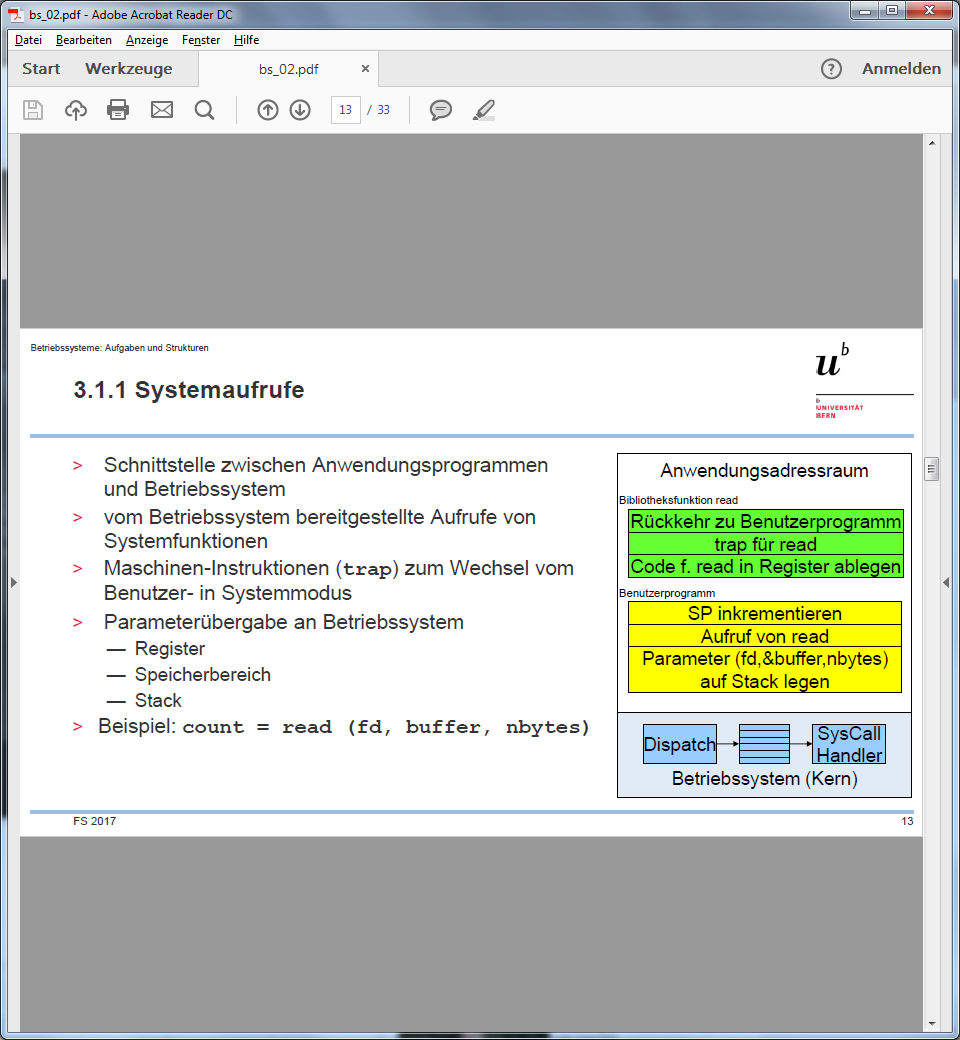
## Schnittstelle zwischen Anwendungen und Betriebssystemen

### Systemaufrufe

**Systemaufrufe** bilden die Schnittstelle zwischen Anwendungsprogrammen und Betriebssystem und werden vom Betriebssystem bereitgestellt um Systemfunktionen aufrufen zu können.

Für den Wechsel vom Benutzer- in den Systemmodus gibt es die Maschinen-Instruktion **trap**.

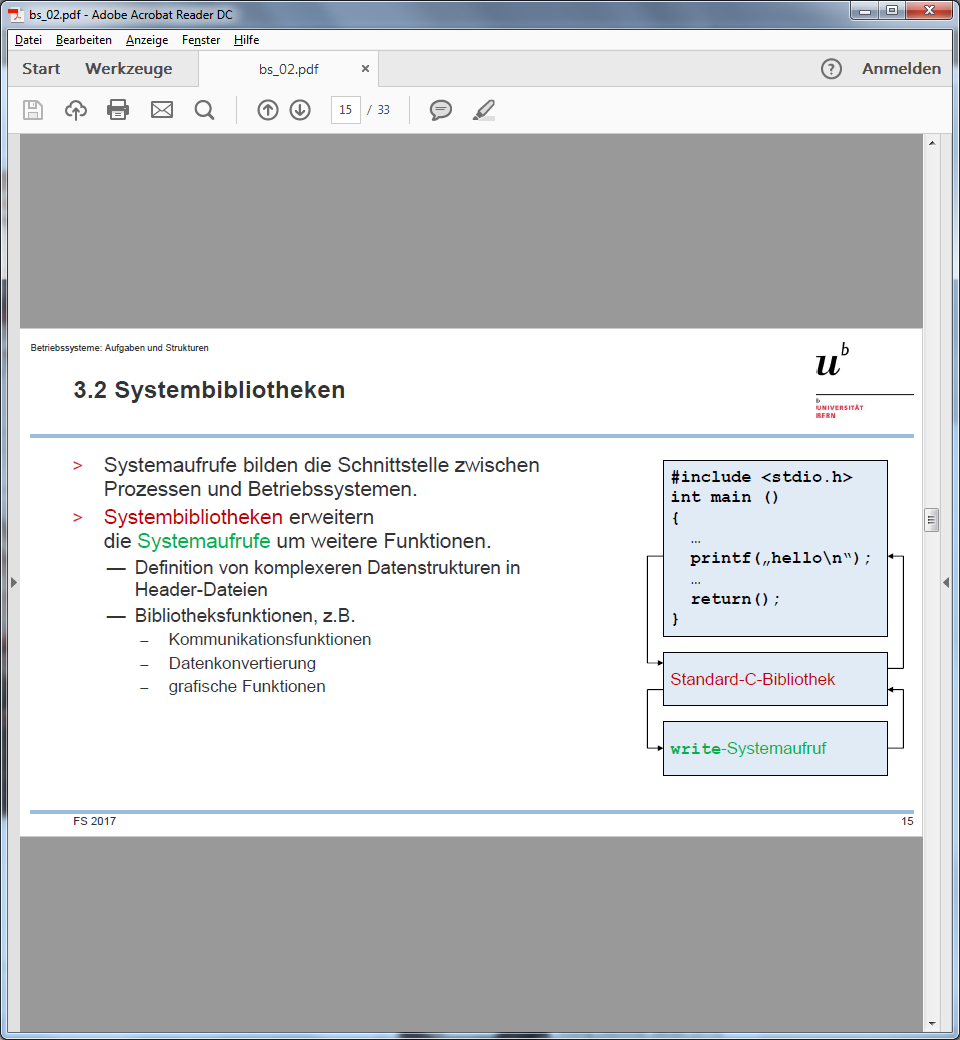
Bei einem Systemaufruf können die Parameter Register, Speicherbereich und Stack mit übergeben werden. Ein Beispiel dafür wäre *count = read (fd, buffer, nbytes)*.



Andere Beispiele für Systemaufrufe sind u.a. *fork()* (Erzeugen eines Kindprozesses), *open()* (öffnen einer Datei) oder *gettimeofday(timeval)* (Abrufen der Systemzeit).

### Systembibliotheken

Die **Systembibilotheken** erweitern die Systemaufrufe um weitere Funktionen. Dabei werden komplexere Befehle oder Instruktionen durch einfachere Aufrufe ummantelt. Beispiel *printf()* der C-Bibliothek stdio.h :



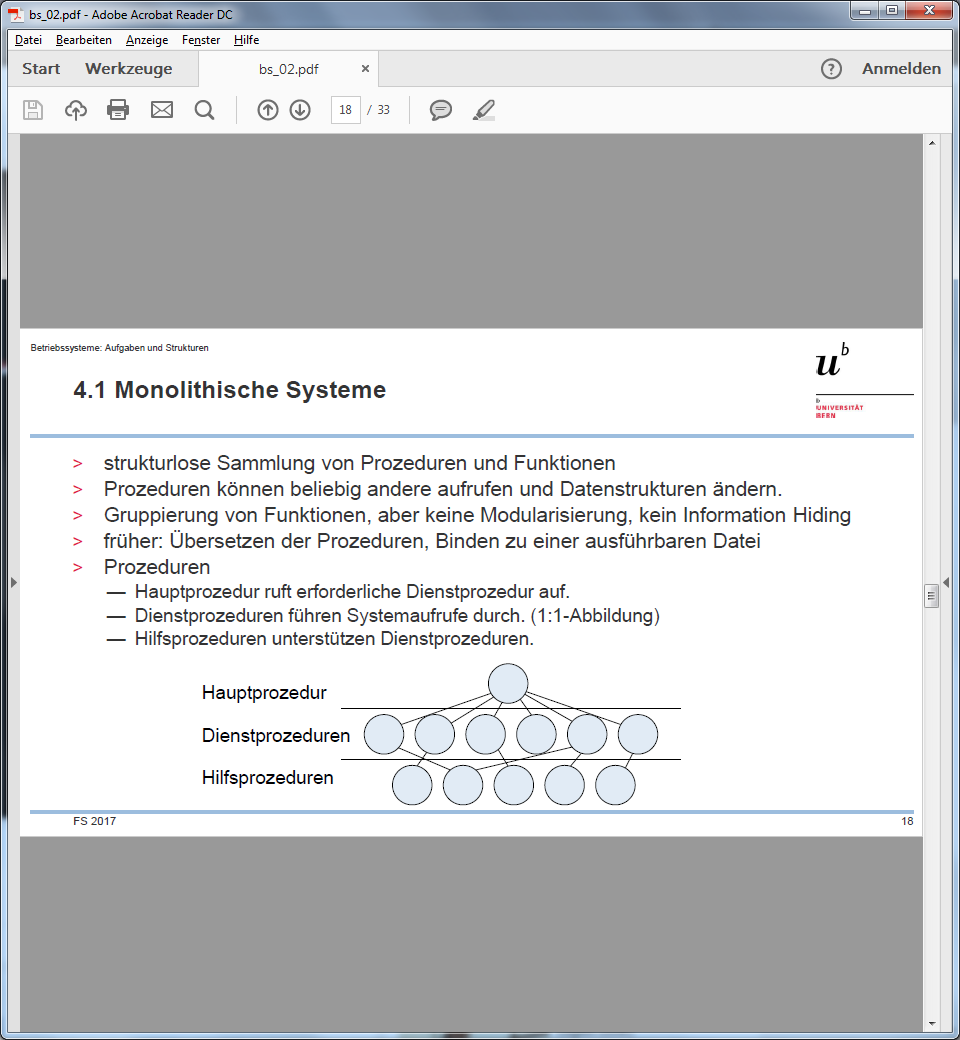
### Systemprogramme

**Systemprogramme** bieten die Grundlagen für Programmentwicklung und -ausführung. Beispiele dafür sind Programme wie der Standard-Editor vi, ftp oder ls.

## Betriebssystemarchitekturen

### Monolithische Systeme

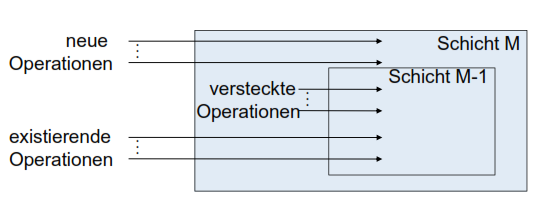
**Monolithische Systeme** sind strukturlose Sammlungen von Prozeduren und Funktionen., Die Prozeduren können dabei beliebige, andere Prozeduren aufrufen und auch Datenstrukturen ändern. Dabei ruft die Hauptprozedur die erforderlichen Dienstprozeduren auf, die jeweils einen Systemaufruf tätigen und dabei von Hilfsprozeduren unterstützt werden können.



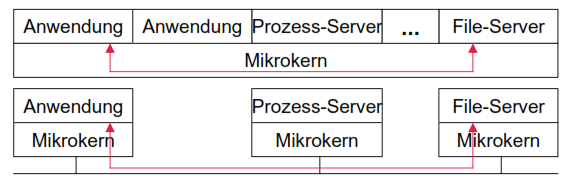
Beispiele dafür sind u.a. MS-Dos oder UNIX.

### Geschichtete Systeme

Hier besteht das Betriebssystem aus mehreren Schichten, welche aufeinander aufbauen und Funktionen der darunterliegenden Schicht brauchen. Vorteile dabei sind die Modularität (Schichten für sich funktionieren) und die Testbarkeit/das Debugging.

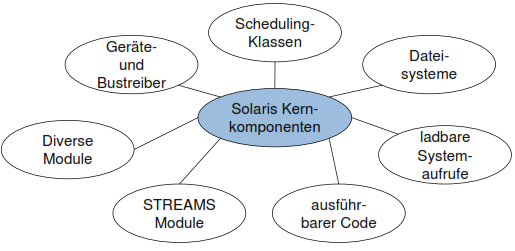


### Mikrokerne

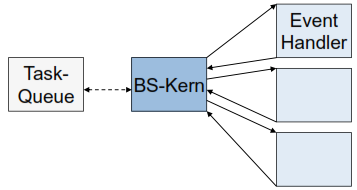
Diese Struktur sorgt für kleine Betriebssysteme, da Teilfunktionen in Server ausgelagert werden. Die Systemaufrufe werden dann an die zuständingen Server weitergeleitet.   


### Module

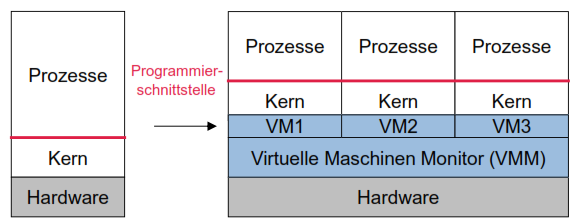
Mit der objektorientierten Programmierung werden Betriebssystemkerne modular erzeugt. Der Betriebssystemkern läuft dann mit den Kernkomponenten und bindet die zusätzlichen Dienste beim Booten oder zur Laufzeit ein. Beispiel Solaris:



### Ereignisgesteuerte Betriebssysteme

Hierbei reagiert das Betriebssystem nur auf bestimmte Ereignisse und arbeitet diese ab. Für länger andauernde Berechnungen werden unterbrechbare Tasks benutzt. Diese Art Betriebssystem wird häufig in Realzeitsystemen eingesetzt.   


### Virtuelle Maschinen

Virtuelle Maschinen abstrahieren die Hardware eines Computers und erzeugen so die Illusion, dass der Umgebung ein eigener Computer zur Verfügung hat. Durch virtuellen Speicher, CPU-Scheduling, etc können so auch verschiedene virtuelle Maschinen emuliert.   


Vorteile virtueller Maschinen sind die Sicherheit und Robustheit durch die Isolation, die Entwicklung oder das bequeme Wechseln von Betriebssystemen. Die Nachteile sind, dass der Kontextwechsel und die Emulationen Overhead erzeugen und Hardwareunterstützung erforderlich ist.

### Hybride Systeme

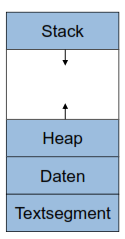
Häfuig werden in der Praxis solche Architekturkonzepte kombiniert. Linux z.B. ist monolithisch, kann aber auch (modular) dynamische Funktionen nachladen.

# Prozesse und Threads

## Prozesse

### Prozesskonzept

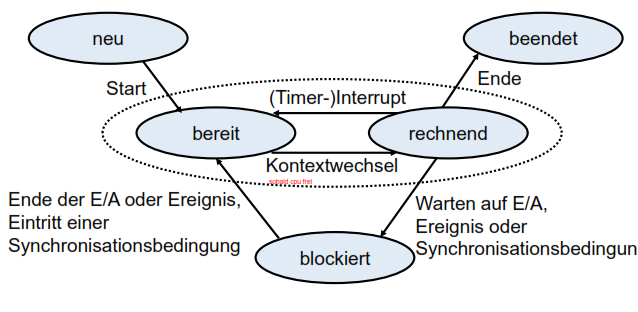
Ein **Prozess** kann als Programm in Ausführung verstanden werden. Er besteht aus Programmcode (Textsegment), einem Befehlszähler (zeigt, wo im Code wir uns befinden) + Prozessorregisterinhalt, einem Stack mit temporären Daten, gloablen Daten und einem Heap für dynamisch zugeteilten Speicher.



Dabei besitzt jedre Prozess eine virtuelle CPU.

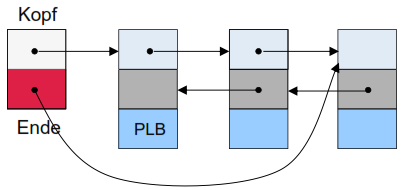
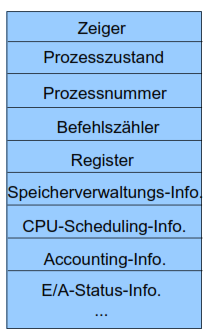
#### Prozesszustände

Im Zustand **neu** befindet sich der Prozess, wenn er neu erzeugt wurde.   
Ein Prozess ist im Zustand **rechnend**, wenn seine Instruktionen auf der CPU ausgeführt werden.   
Wenn er auf ein Ereignis warten muss (zB Ende E/A) ist er im Zustand **blockiert**.   
Wartet er auf CPU-Zuteilung ist er im Zustand **bereit**.   
Hat er die Ausführung abgeschlossen ist er im Zustand **beendet**.



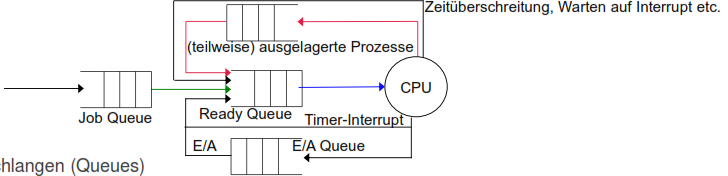
#### Prozessleitblock

Der Prozessleitblock beschreibt den Prozesszustand. Dabei repräsentiert er einen Prozess im Betriebssystem. Diese Prozessleitblöcke werden dann in den Warteschlangen verkettet.

### Prozesswechsel

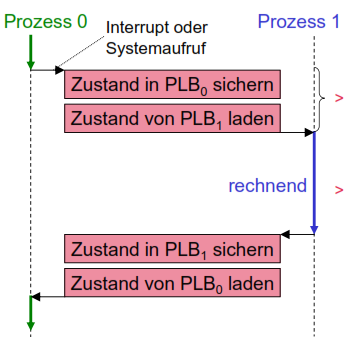
#### Scheduling

Es gibt verschiedene Warteschlangen (Queues). Die **Job Queue** sind die abgeschickten, auf Massenspeicher abgelegten Prozesse. In der **Ready Queue** sind die Prozesse, die im Zustand bereit sind. In der **E/A-Queue** sind die Prozesse, die auf ein E/A-Gerät warten. Dann gibt es noch eine Queue von ausgelagerten Prozessen.   


#### Dispatcher

Der **Dispatcher** übergibt dem beim Scheduling ausgewählten Prozess die Kontrolle über die CPU, nimmt dabei den Kontextwechsel vor, macht den Wechsel in den Benutzermodus und vollführt den Sprung an die korrekte Stelle im Anwendungsprogramm und die Fortsetzung der Ausführung. Der Dispatcher sichert also auch den aktuellen Zustand des Prozesses auf der CPU.

#### Prozessumschaltung



Der Zustand von Prozess 0 wird in den Prozessleitblock 0 gesichert. Dann wird der Zustand von Prozess 1 aus dem Prozessleitblock 1 geladen. Dann arbeitet dieser, ehe sein Zustand wieder in den Prozessleitblock 1 gesichert wird und der Zustand von Prozess 0 aus dem PLB 0 geladen wird.

### Prozesssteuerung

#### Prozesserzeugung

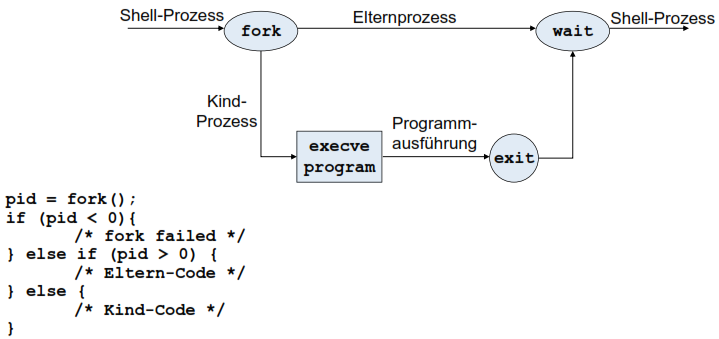
Prozesse werden bei Systeminitialisierung durch Benutzer oder laufende Prozesse erzeugt. Erzeugt dabei ein Prozess einen anderen Prozess hat er als Elternprozess einen Kindprozess erzeugt. Dabei gibt es mehrere Optionen für das Teilen von Ressourcen (Eltern und Kinder teilen alle Ressourcen, Kinder teilen nur Teilmenge der Elternressourcen, Eltern und Kinder teilen keine Ressourcen), die Ausführung (Nebenläufiges Ausführen von Eltern/Kindern, Eltern warten auf Beendigung der Kinder) und den Adressraum (Kindprozess als Duplikat des Elternprozesses, Kindprozess lädt eigenes Programm).

Beispiel: *fork()* erzeugt eine Prozesskopie mit neuer Prozessnummer

#### Beenden von Prozessen

Wenn der Prozess die letzt Anweisung ausgeführt hat beauftragt er das Betriebssystem mit der Löschung (*exit)*. Der Elternprozess kann aber auch auf die Beendigung eines Kindprozesses inklusive der Datenrückgabe warten (*wait*). Schliesslich kann der Elternprozess den Kindprozess auch beenden (*abort*), zB wenn die zugeteilten Ressourcen verbraucht wurden, der Kindprozess nicht länger benötigt wird oder der Elternprozess terminiert.

#### Beispiel UNIX

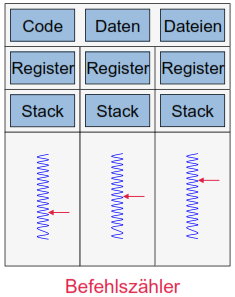


## Threads

### Threads-Konzept

Ein **Thread** kann als leichtgewichtiger Prozess angesehen werden. Er besteht aus einer ID, einem Befehlszähler, einem Register und einem Stack. Die Threads teilen untereinander den Programmcode, globale Daten und Betriebssystemressourcen (Signale, offene Dateien). Ein **Task** ist dann die Ansammlung von Threads. Ein traditioneller, schwergewichtiger Prozess besteht aus einem Task und einem Thread.

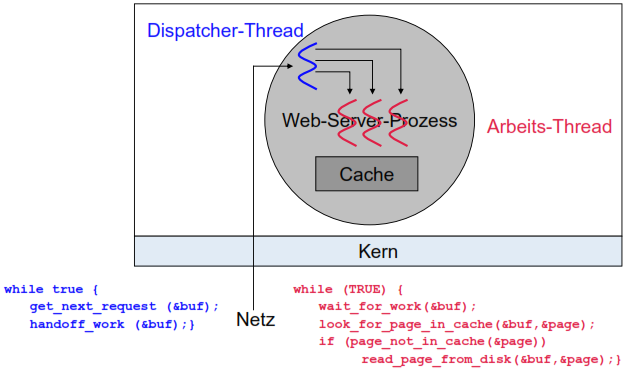
Ein Thread eines Tasks kann auch dann arbeiten, während andere Threads blockiert sind. Dazu können sie Kind-Threads erzeugen. Das erfordert Koordination und Steuerung. Durch den gemeinsamen Speicher wird die Kooperation vereinfacht und durch die gemeinsamen Ressourcen der Kontextwechsel effizient gestaltet.

Multi-Threading bezeichnet dann das abhandeln mehrerer Threads in einem Prozess. Die Zustände sind dabei gleich wie bei den Prozessen.   


#### Nutzung und Vorteile von Threads

Wenn bei Applikationen mehrere Aktivitäten gleichzeitig ablaufen, können manche davon blockieren. Solche quasiparallelen Aktivitäten können mit Threads einfacher programmiert werden. Für Threads werden auch keine **Betriebsmittel** (Für Prozesse benötigte Systemressourcen, zB Drucker, Arbeitsspeicher) benötigt, so können sie einfacher und schneller erzeugt werden. Ausserdem können sie Ressourcen teilen, sind reaktionsfreudig und können auf verschiedene Prozesse abgebildet werden.

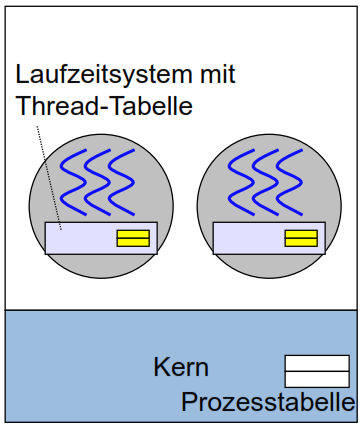
Ein Beispiel für Threads ist ein Browser mit mehreren Fenstern oder ein Web-Server mit einem Thread pro Request:



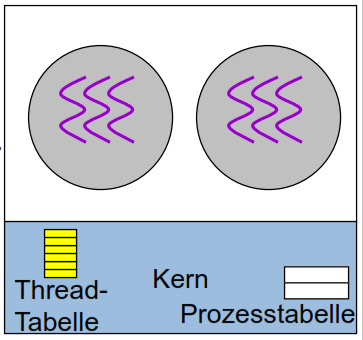
### Thread-Typen

#### User Threads

User Threads laufen unabhängig vom Betriebssystem. Es gibt auch keinen Wechsel des Adressraums bei einem Thread-Wechsel. Dazu hat jeder Prozess eine eigene Thread-Tabelle. Die Threads sollten dann die CPU nach einer Weile freiwillig aufgeben. Das Scheduling wird von den Anwendungen übernommen. Blockiert ein Thread, blockiert der ganze Prozess.

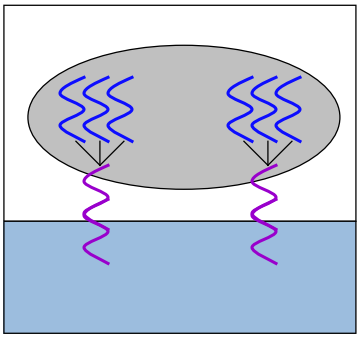


#### Kernel Threads

Hier speichert eine Thread-Tabelle die Registerinhalte und andere Zustandsinformationen. Um Threads zu erzeugen oder zu löschen werden Systemaufrufe verwendet. Blockiert ein Thread, ist ein Wechsel zu einem anderen Thread möglich. Ein Nachteil ist der höhere Overhead.   


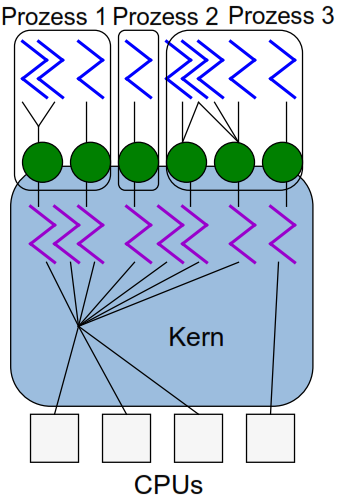
#### Hybride Threads

Die Hybriden Threads sind eine Kombination der Vorteile von User und Kernel Threads. Die User Threads werden normal verwaltet (also von den Anwendungen), dann auf Kernel Threads gemultiplext und diese werden dann vom Kern gescheduled.



Das Multiplexen kann in N:1, 1:1 oder N:M Verhältnissen stattfinden.

Ein Beispiel für hybride Threads sind **Leichtgewichtsprozessse**. Dabei ist jeder LWP mit einem Kernel Thread verbunden, welche dann vom Betriebssystem gescheduled werden. Falls ein Kernel Thread blockiert blockieren auch der LWP und die User Threads.



#### Pop-Up Threads

Im traditionellen Ansatz wird der Prozess oder der Thread bei einer ankommenden Nachricht durch d *receive* Systemaufruf blockiert und wartet so auf die Nachricht.

Bei Pop-Up Threads veranlasst das Betriebssystem bei einer ankommenden Nachricht einen neuen Thread, welcher sich um diese Nachricht kümmert.

#### Thread Pools

Die Thread-Erzeugung verursacht einen gewissen Aufwand. Eine Lösung dafür ist der Thread Pool. Dabei werden bei Task-Start Threads erzeugt und dann in einem Pool abgelegt. Wird nun ein Thread benötigt, wird er aus dem Pool entnommen um die Anfrage zu bearbeiten und dann wieder zurückgelegt. Ist kein Thread verfügbar, muss gewartet werden, bis ein Thread im Pool verfügbar wird.

#### Thread-Steuerung

Für die Steuerung von Threads werden Bibliotheksprozeduren verwendet. Beispiele dafür sind die Aufrufe *pthread\_create*, *pthread\_exit* oder *clone*. Oft gibt es auch Aufrufe zum freiwilligen Aufgaben der CPU. Diese sind wichtig, denn die User Threads sind für das Betriebssystem unsichtbar.

#### Probleme von Multi-Threaded Code

Mögliche Gefahren sind das Überschreiben globaler Variablen, das Stack-Management oder die Schwierigkeit der Zuordnung von Signalen durch Kern zu User Threads (bsp Tastatureingaben)

# Scheduling von Prozessen

Eine Prozessausführung besteht aus einer Folge von Instruktionen. Ein **CPU-Burst** bezeichnet dabei eine Sequenz von CPU-Zyklen, ein **E/A-Burst** eine Ein-/Ausgabe.

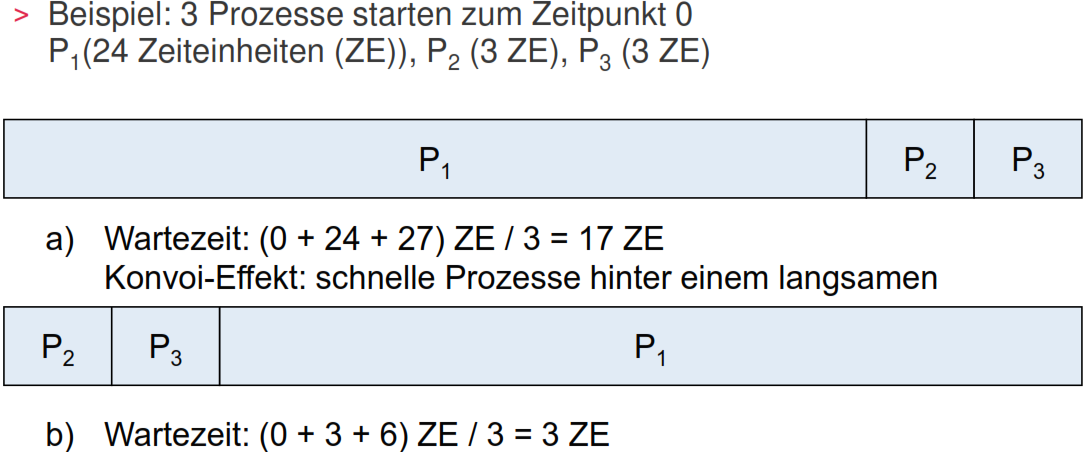
Der **CPU-Scheduler** hat die Aufgabe, den nächsten Prozess aus der Ready-Queue auszuwählen, der in den Zustand rechnend werden darf. Dabei wird zwischen nicht präemptivem Scheduling (dem rechnenden Prozess kann die CPU nicht genommen werden) und präemptivem Scheduling unterschieden.

Scheduling-Kriterien sind u.a. Fairness, CPU-Auslastung, Durchsatz, Wartezeit, Verweilzeit oder Antwortzeit.

## Scheduling-Mechanismen

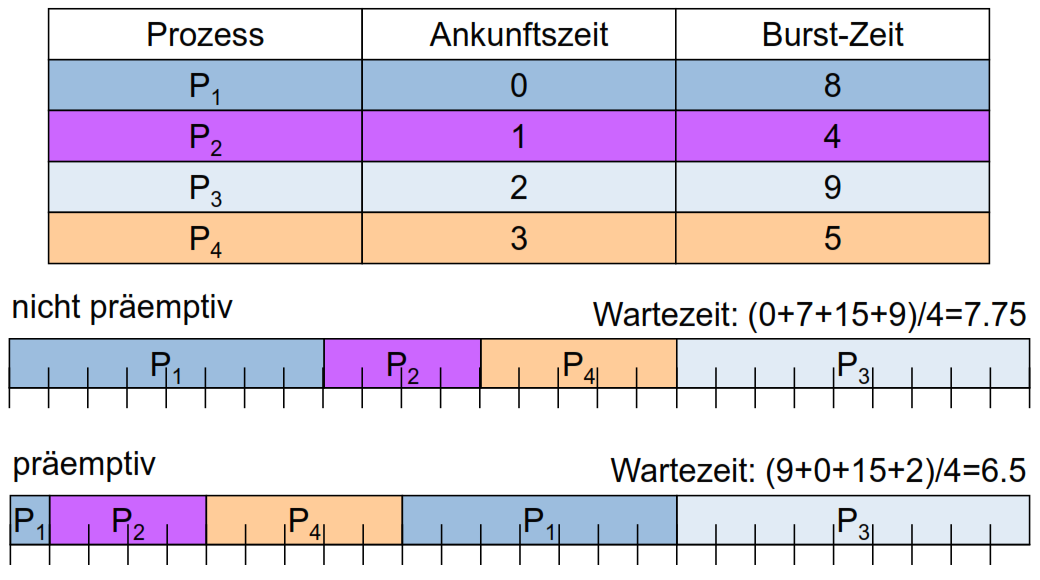
### First Come First Serve

Dieses Verfahren ist nicht präemptiv. Wie der Name sagt wird einfach der Prozess zuerst ausgeführt, der zuerst eingetroffen ist.

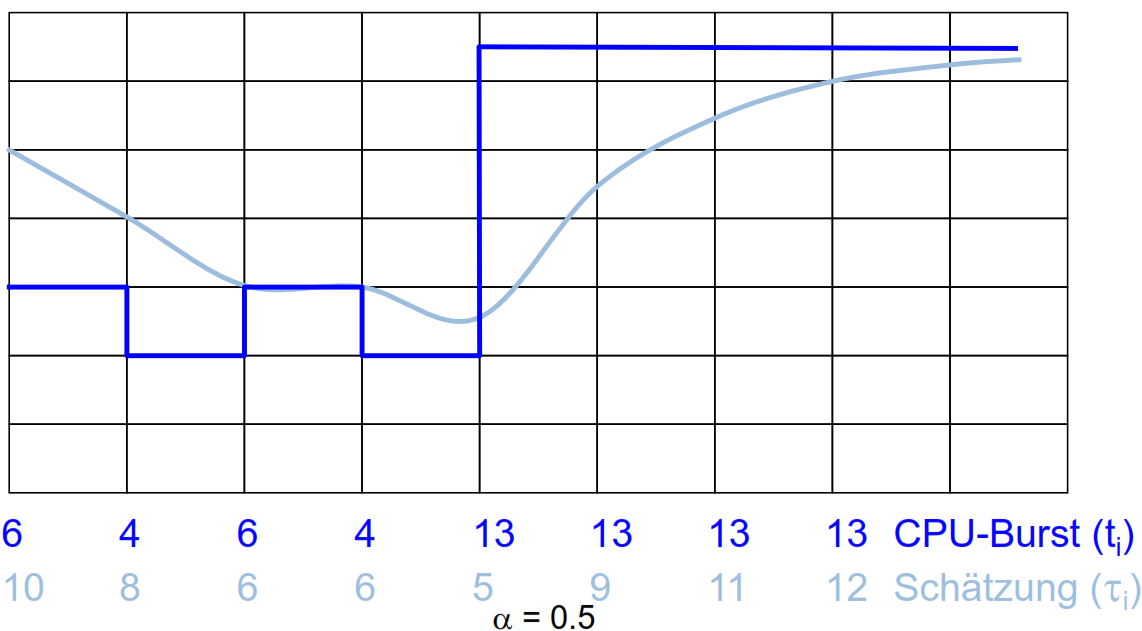
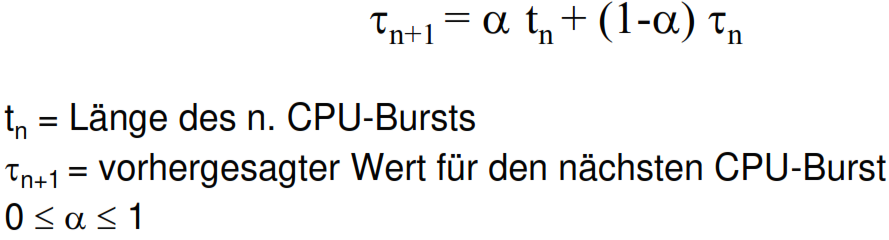


### Shortest Job First

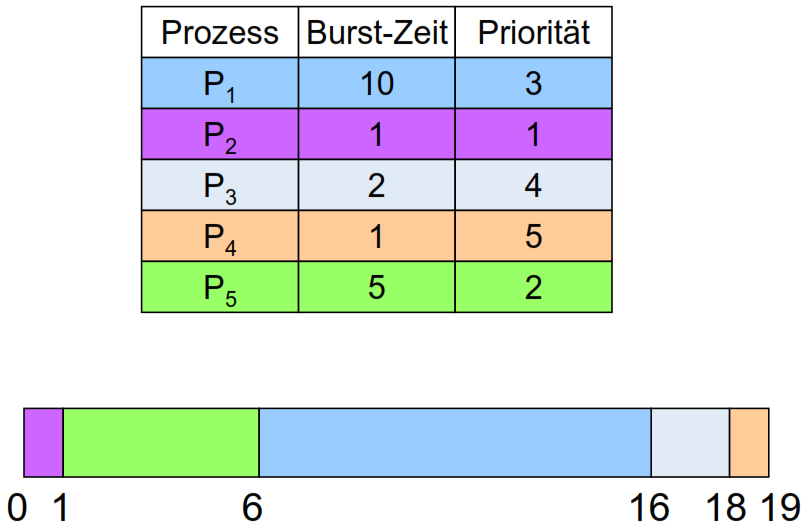
Hier wird der Prozess ausgewählt, der den nächst kürzesten CPU-Burst hat. Dies dient der Optimierung der Wartezeit. Das kann präemptiv (Ein rechnender Prozess kann sofort verdrängt werden, wenn ein neu ankommender Prozess schneller fertig wäre als der aktuell rechnende) oder nicht präemptiv (Ein rechnender Prozess kann nicht verdrängt werden, bevor sein CPU-Burst beendet ist).



#### CPU-Burst-Bestimmung

Es ist nur möglich, aufgrund vorhergegangenen Bursts eine Schätzung zu machen.   


### Prioritäts-Scheduling

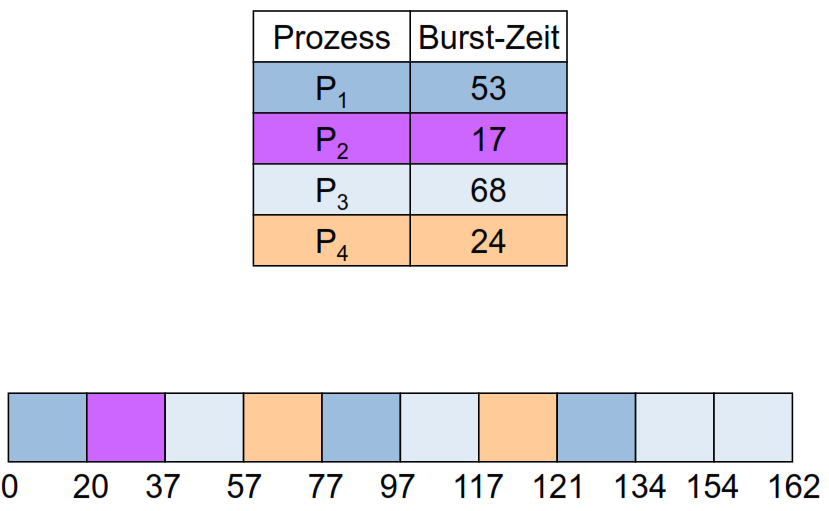
Jeder Prozess erhält eine Prioritätsnummer. Nun wird der Prozess mit der höchsten Priorität (niedrigste Prioritätsnummer) ausgewählt. Ein Problem hierbei kann das Aushungern sein. Das bedeutet, dass Prozesse mit höherer Priorität unter Umständen nie bedient werden, weil stets Prozesse mit höherer Priorität eintreffen. Eine Lösung hierfür wäre das Altern, wobei die Priorität mit der Zeit steigt. Diese Art Scheduling kann präemptiv oder nicht-präemptiv sein. 

#### Interne und externe Prioritäten

**Interne Prioritäten** basieren auf messbaren Prozesseigenschaften, wie dem Speicherbedarf oder der Zahl offener Dateien.   
**Externe Prioritäten** sind Prioritäten, die auf Kriterien von ausserhalb des Betriebssystems basieren, z.B. die Wichtigkeit des Users oder bezahlte Gebühren.

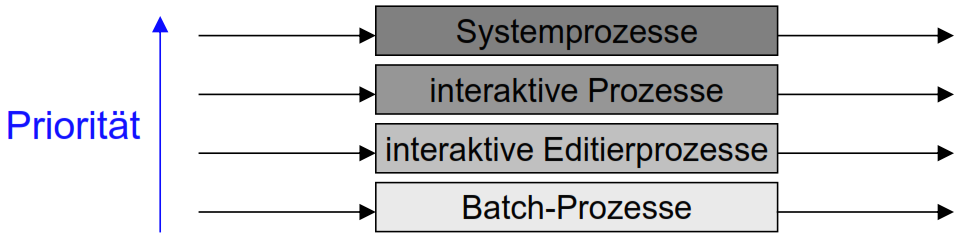
### Round Robin

Dieser präemptive Mechanismus ist geeignet für Time-Sharing. Dabei erhält jeder Prozess eine kleine Zeit die CPU. Danach gibt er sie wieder ab und reiht sich wieder in die Ready-Queue ein. Dabei ist es wichtig, einen guten Wert für die Zeit (q) zu finden. Ist der Wert zu gross, haben wir nur ein FIFO. Ist er zu gross, haben wir einen zu grossen Overhead.

q=20

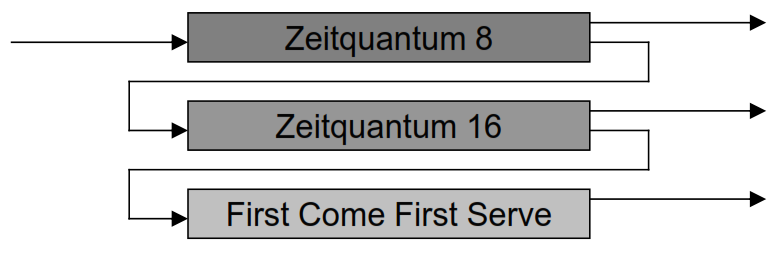
### Multilevel Queue

Die Ready-Queue wird in verschiedene Queues unterteilt, wovon jede einen eigenen Scheduling-Algorithmus verwendet. Zwischen diesen Queues wird dann ebenfalls ein Scheduling gemacht. Das kann z.B. eine feste Priorität oder Zeitscheiben sein.



### Multilevel Feedback Queue

Ein präemptives Verfahren für die Migration zwischen verschiedenen Queues (z.B. wenn ein Prozess zu viel CPU-Zeit beansprucht hat). Dabei gilt meist eine hohe Priorität für I/O-gebundene Aktivitäten und interaktive Prozesse.



### Lotterie-Scheduling

Hier hat jeder Prozess ein Los und das Betriebssystem führt die Verlosung durch. Der «Gewinner» erhält dann CPU-Zeit. Wichtige Prozesse können dabei Extra-Lose erwerben.

### Garantiertes Scheduling

Hierbei wird die vorgesehene CPU-Zeit jedes Prozesses berechnet: (aktuelle Zeit – Erzeugungszeitpunkt) / Anzahl Prozesse.   
Dann wird das Verhältnis (verbrauchte CPU-Zeit/vorgesehene CPU-Zeit) berechnet. Nun wird der Prozess mit dem tiefsten Verhältnis so lange ausgeführt, bis ein anderer Prozess das tiefste Verhältnis aufweist.

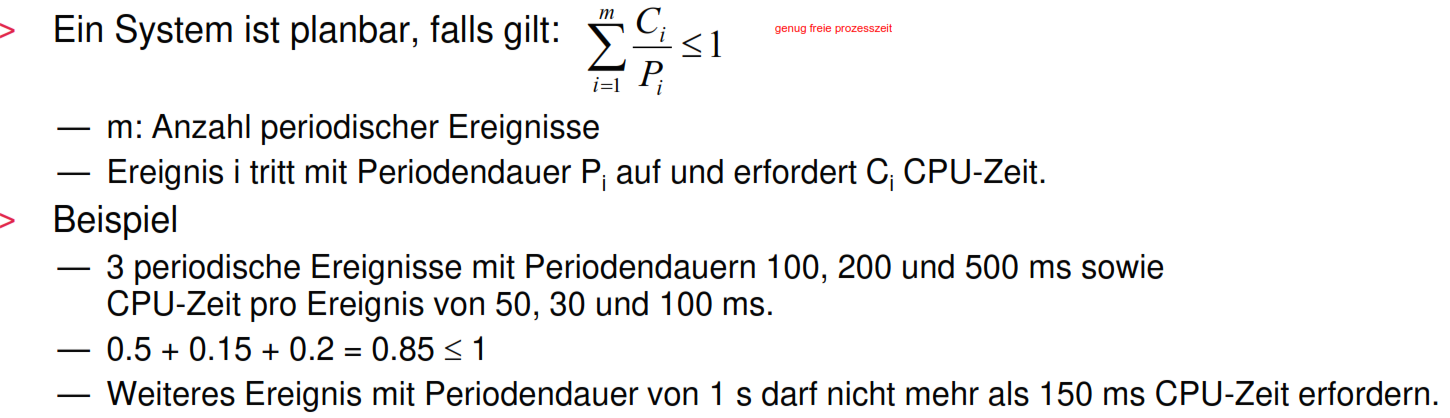
## Echtzeitsysteme

Ein **Echtzeitsystem** muss in begrenzter Zeit auf ein Ereignis reagieren. Ein **hartes Echtzeitsystem** garantiert dabei das Beenden eines Prozesses in garantierter Zeit. Dabei werden Mechanismen, wie Zugangskontrolle, Ressourcenreservierung, Scheduling und kein Sekundärspeicher (da zu langsam und schwer zu beherrschen) verwendet. Ein **weiches Echtzeitsystem** versucht zwar, Zeitüberschreittungen durch Prioritäten zu vermeiden, kann dies aber nicht garantieren.

### Implementierung von Echtzeitsystemen

Es wird ein Prioritäts-Scheduling vorgenommen, also Realzeitprozesse priorisiert. Ausserdem werden Verzögerungen minimiert. Dies geschieht durch das Vererben von Prioritäten, begrenzte Dispatch-Verzögerungen und präemptive Kerne.

### Echtzeit-Scheduling

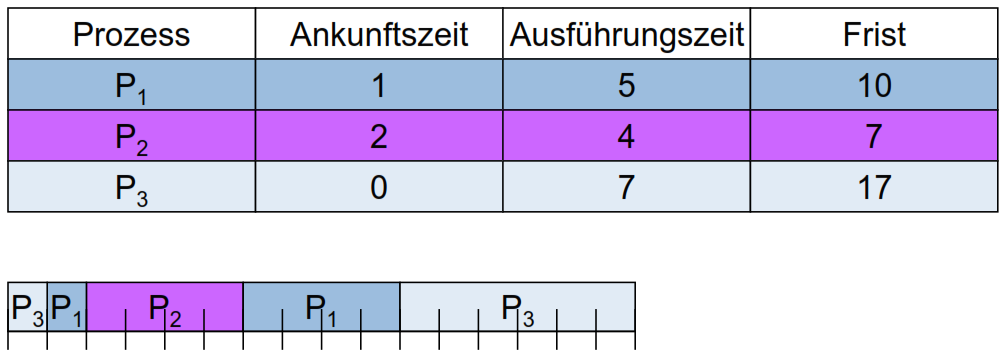


#### Offline-Scheduling

Das Scheduling wird vor der eigentlichen Programmausführung vorgenommen, um einen hohen Overhead beim Scheduling zu vermeiden. So wird ein vollständiger Ausführungsplan in Tabellenform vorberechnet. Das setzt aber periodische Aktivitäten voraus.

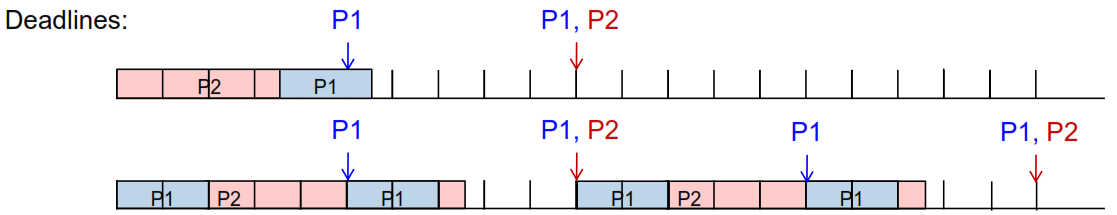
#### Earliest Deadline First

Hierbei haben Prozesse Ausführungsfristen, wobei dann der Prozess mit der engsten Frist selektiert wird. Dies ist präemptiv und nicht-präemptiv möglich.



#### Rate Monotonic Scheduling

Dieses Scheduling funktioniert nur für periodische Systeme. Dabei wird angenommen, dass Prozesse mit hoher Frequenz meist tiefe Deadlines haben. Diese erhalten dann eine hohe Priorität. Aktivitäten mit tiefer Frequenz erhalten dann auch eine tiefe Priorität. Das führt dann zu einer minimaleren Verzögerung von Aktivitäten mit hoher Frequenz und sorgt so dafür, dass deren Fristen seltener verletzt werden. Aber dadurch kann es auch zur Verstückelung von Aktivitäten mit niedriger Frequenz kommen, da eine höhere Anzahl Kontextwechsel zu Prozessen mit höherer Priorität besteht.



## Multiprozessor-Scheduling

### Asymmetrisches und symmetrisches Multiprocessing

Bei **Asymmetrischem Multiprocessing** wird durch einen Master-Prozessor gescheduled. Die anderen Prozesse führen dann den Benutzercode aus.   
Bei **Symmetrischem Multiprocessing** führt jeder Prozess sein eigenes Scheduling durch.

### Prozessor-Affinität

Unter diesem Begriff versteht man das Zuweisen der gleichen CPU für einen Thread. So kann die Verfügbarkeiten der Daten in Caches und lokalem Speicher ausgenutzt werden.

Bei der weichen Affinität versucht das System, den Prozess immer auf dem gleichen Prozessor zu belassen, kann das aber nicht garantieren.

Bei der harten Affinität kann der Prozess spezifizieren, dass er immer auf demselben Prozessor ausgeführt werden will.

### Gruppen-Scheduling

Hierbei werden Threads mehreren CPUs zugewiesen. Die gleichzeitige Zuweisung von Prozessoren erlaubt eine effiziente Interprozesskommunikation über den gemeinsamen Speicher. Die Anwendung muss dabei kennzeichnen, welche Threads kooperieren. Eine Thread-Gruppe kommt nur bei genügen vielen freien Prozessen zur Ausführung, so dass alle Threads gleichzeitig auf einem eigenen Prozessor zur Ausführung kommen (Gang-Scheduling). So wird bei eng kooperierenden Threads Blockierungen reduziert. Wenn aber nur grössere Thread-Gruppen existieren kann das problematisch sein.

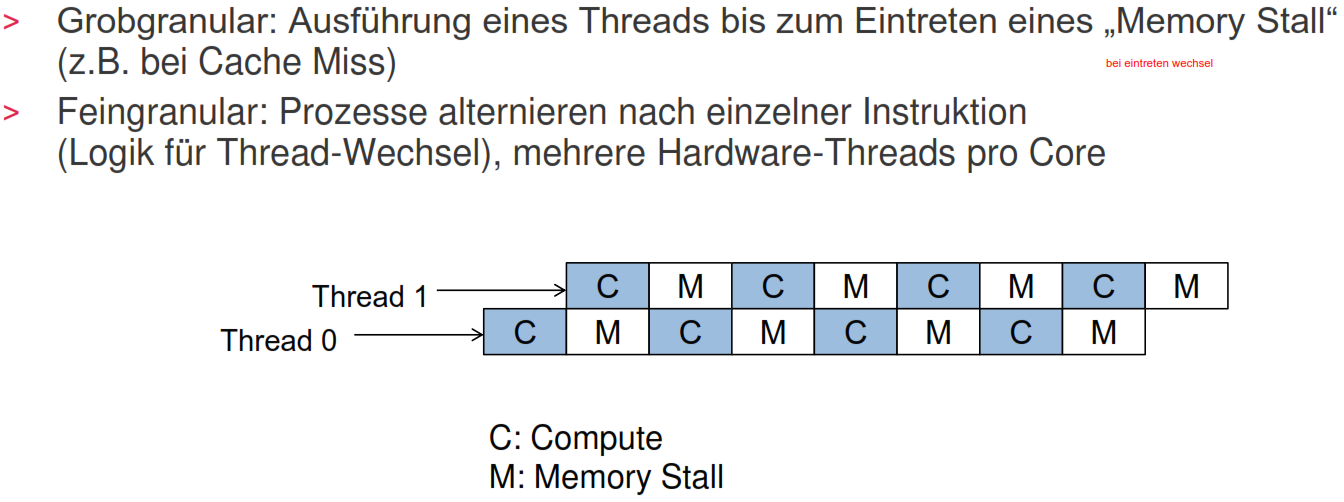
### Lastausgleich

Bei **Push Migration** prüft ein spezifischer Task periodisch die Last auf allen Prozessoren und verteilt die Prozesse darauf.

Bei **Pull Migration** fordern untätige Prozessoren Prozesse von ausgelasteten an.

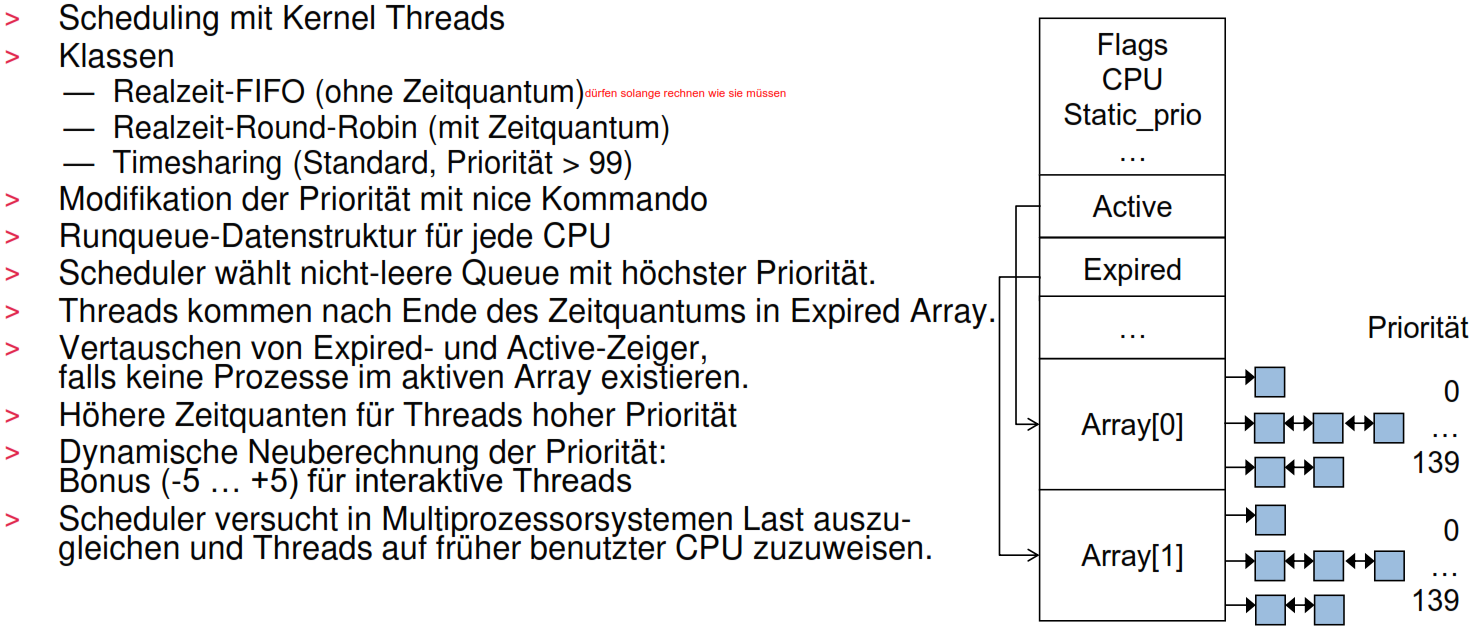
In der Praxis werden diese beiden Praktiken oft kombiniert.

### Multithreading



## Diverse Aspekte

### Beispiel Linux



### Scheduling von Threads

Es gibt zwei Scheduling-Ebenden: Prozesse und Threads. User Threads werden innerhalb eines Prozesses scheduled. Bei Kernel Threads kann auch zwischen Threads verschiedener Prozesse gewechselt werden.

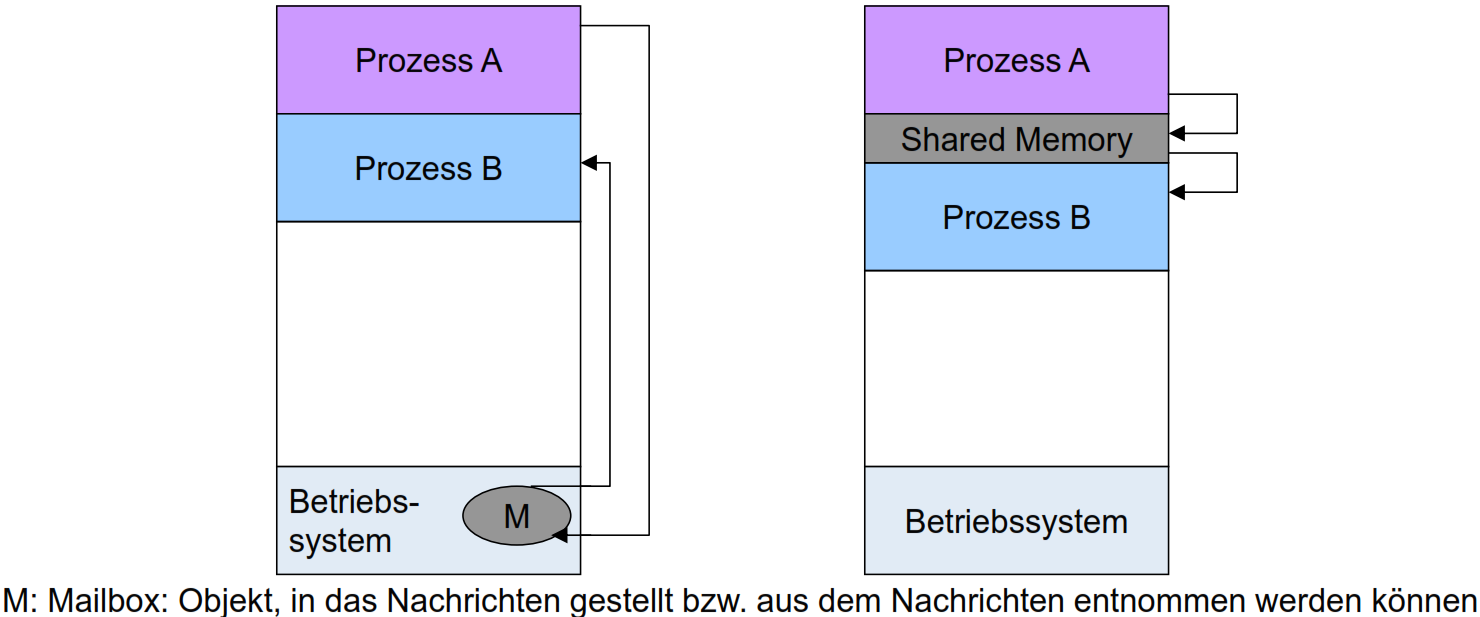
# Kommunikation und Synchronisation zwischen Prozessen

## Prozessinteraktion

Unter diesem Begriff versteht man die gegenseitige Beeinflussung kooperirender Prozesse. Der Vorteil hier liegt im Teilen von Informationen, der Modularität und der beschleunigten Verarbeitung in Multiprozessorsystemen.

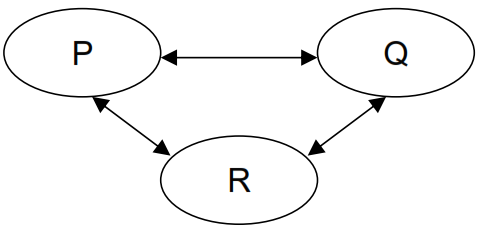
Das kann **speicherbasiert** erfolgen, nämlich über den Zugriff auf Variablen in gemeinsamem Speicher, oder **nachrichtenbasiert**, also über den Austausch von Nachrichten über *send/receive* Operationen.

### Message Passing und Shared Memory



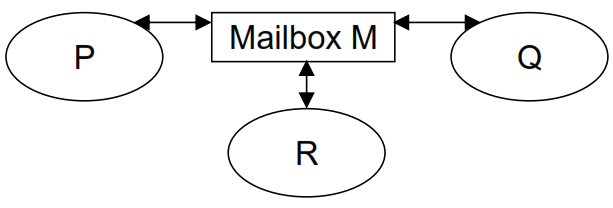
#### Direkte Kommunikation

Links zwischen Prozesspaaren werden automatisch etabliert. Dabei wird der Partnerprozess explizit adressiert.



#### Indirekte Kommunikation

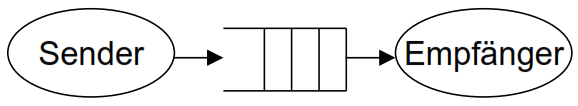
Die kommunizierenden Prozesse teilen hier eine Mailbox, die zu einem Prozess oder System gehört.



#### Puffern von Nachrichten

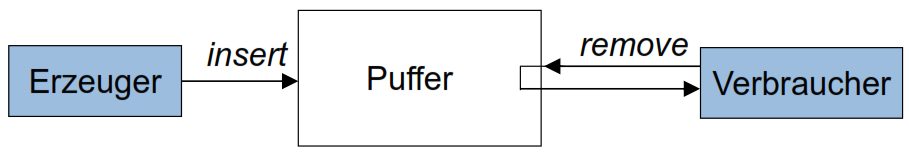
Bei der synchronen Kommunikation haben wir keine Kapazität zum Speichern von Nachrichten. Dabei müssen Sender und Empfänger synchronisiert sein.

Bei der asynchronen Kommunikation platziert der Sender die Nachrichten in einer Warteschlange mit endlicher Kapazität und fährt fort. Bei voller Warteschlange muss der Sender warten, bei leerer Warteschlange der Empfänger.

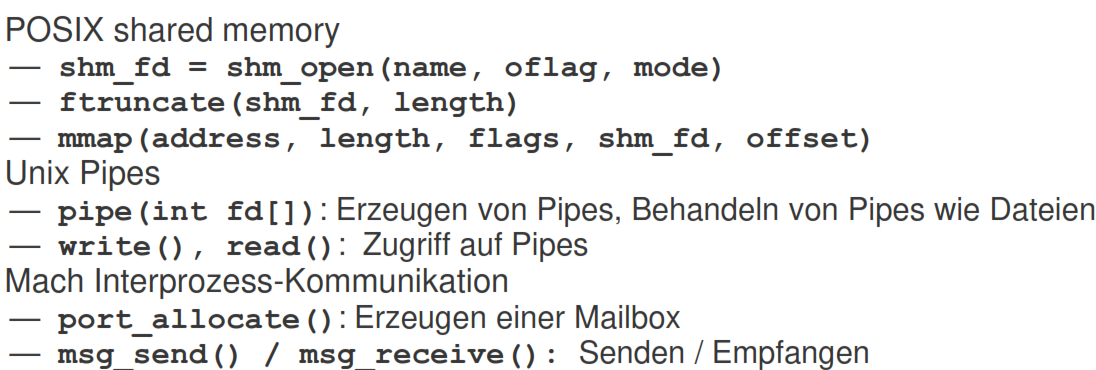


### Speicherbasierte Prozessinteraktion

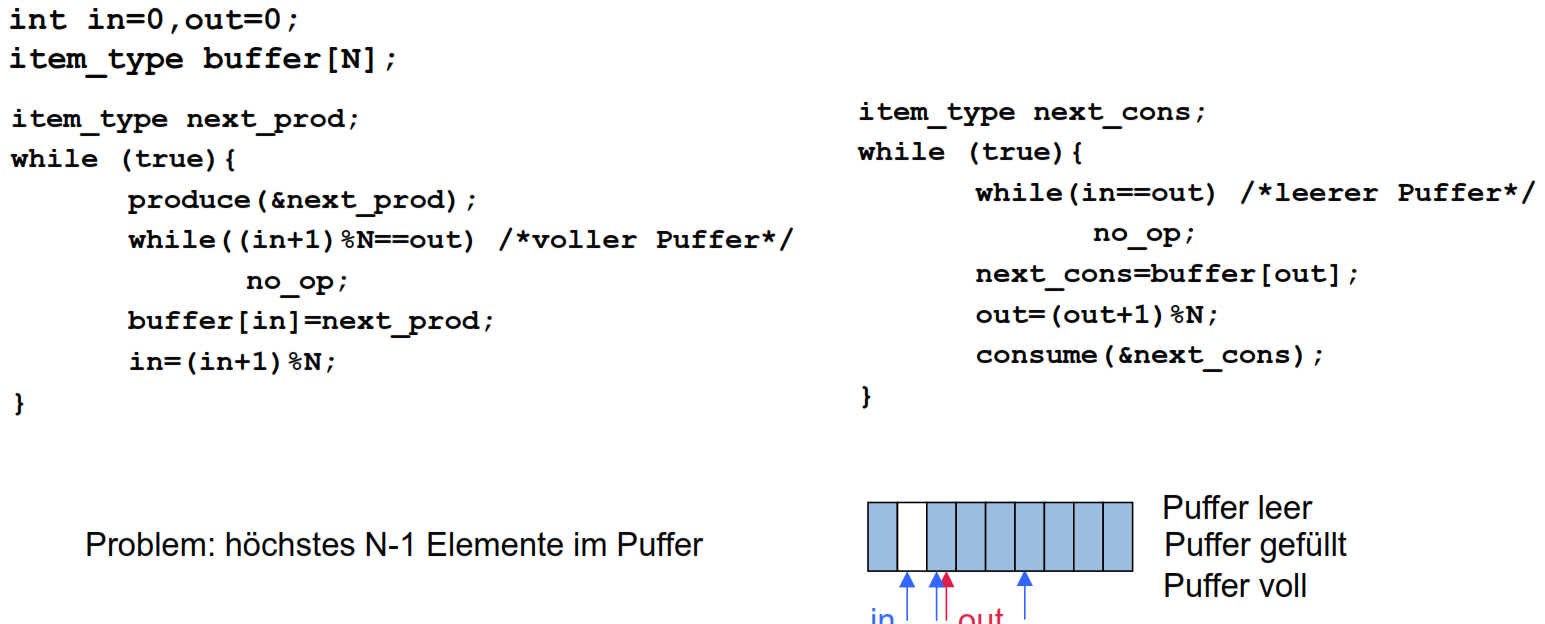
Wenn parallele Prozesse Zugriff auf gemeinsame Daten haben können dadruch Inkonsistenzen verursacht werden. Deshalb sind Mechanismen zur Synchronisation erforderlich. Ein Beispiel ist das Erzeuger/Verbraucher-Problem: Dabei kann ein Buffer N Elemente speichern. Der Erzeuger darf dann nicht in den vollen Puffer speichern, während der Verbraucher nich aus dem leeren Speicehr entfernen darf.



### Beispiele für Prozessinteraktion



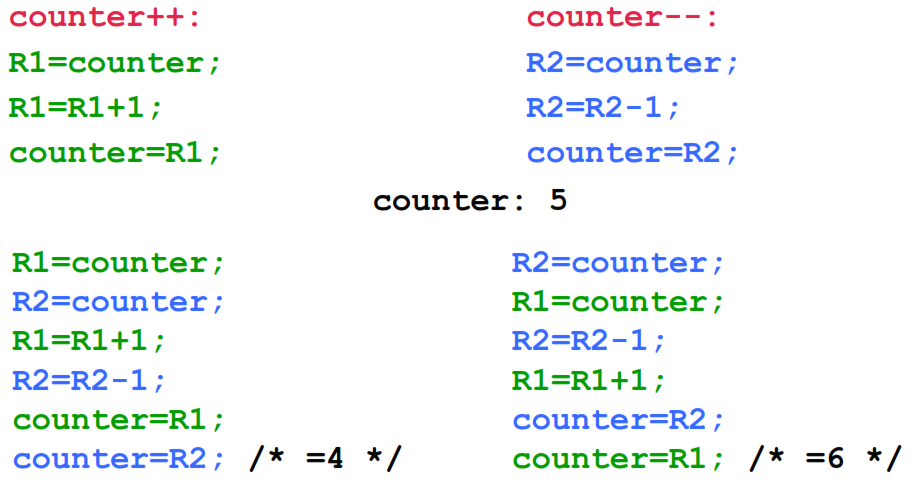
## Erzeuger/Verbraucher-Implementierung



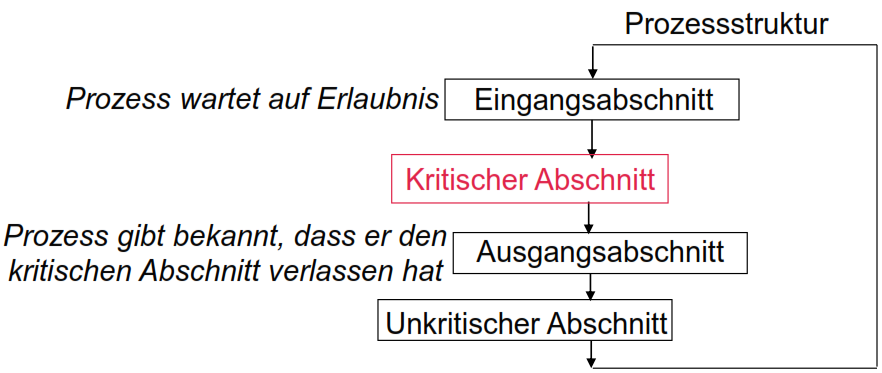
Hier wird geprüft, ob der Buffer leer/bzw. voll ist. Das Problem bei dieser Implementierung ist, dass nicht der volle Buffer genutzt werden kann. Dafür wäre eine Implementierung mit counter nötig.

### Race Condition

Eine Race Condition tritt auf, wenn zwei Prozesse gleichzeitig auf eine Variable zugreifen. Dann kann es zum Fall kommen, dass eine Operation auf die erste warten müsste, dies aber nicht tut, so zu früh ausliest und die andere Operation praktisch überschreibt. Vergleiche dazu auch die Antworten zu den praktischen Serien am Ende.

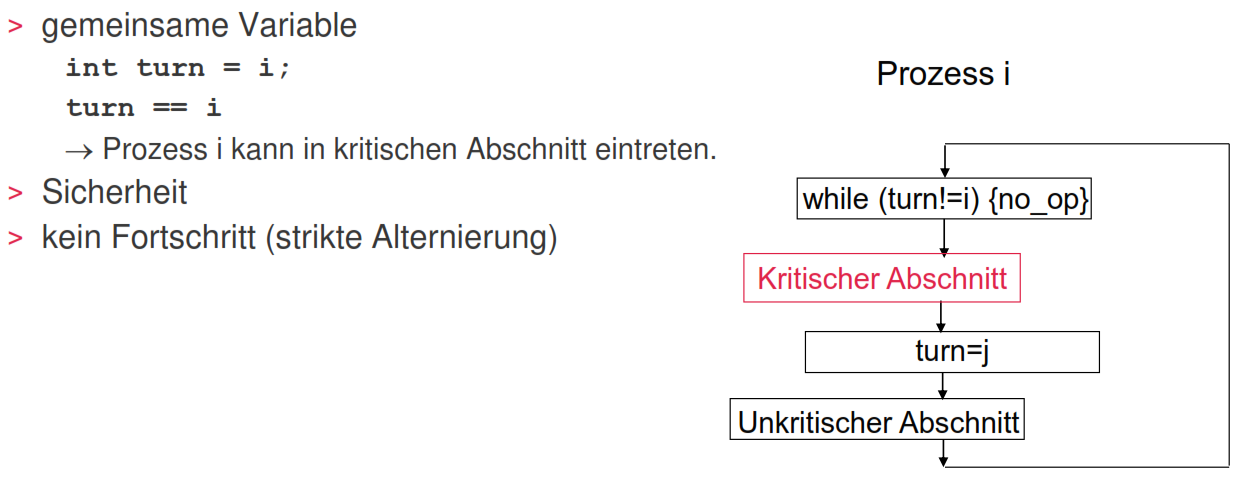


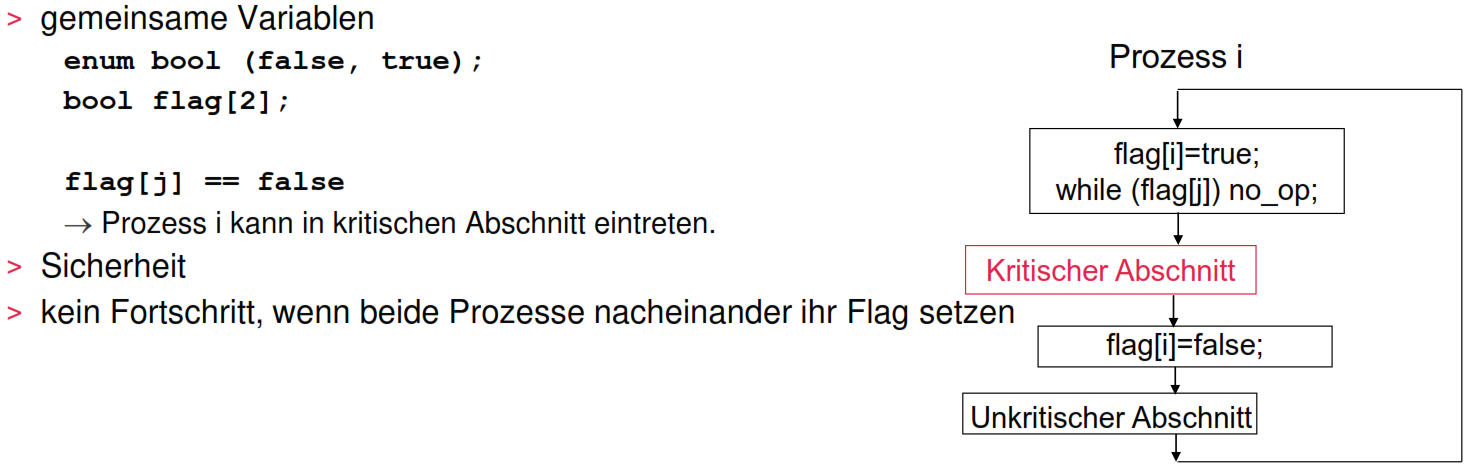
## Kritischer Abschnitt

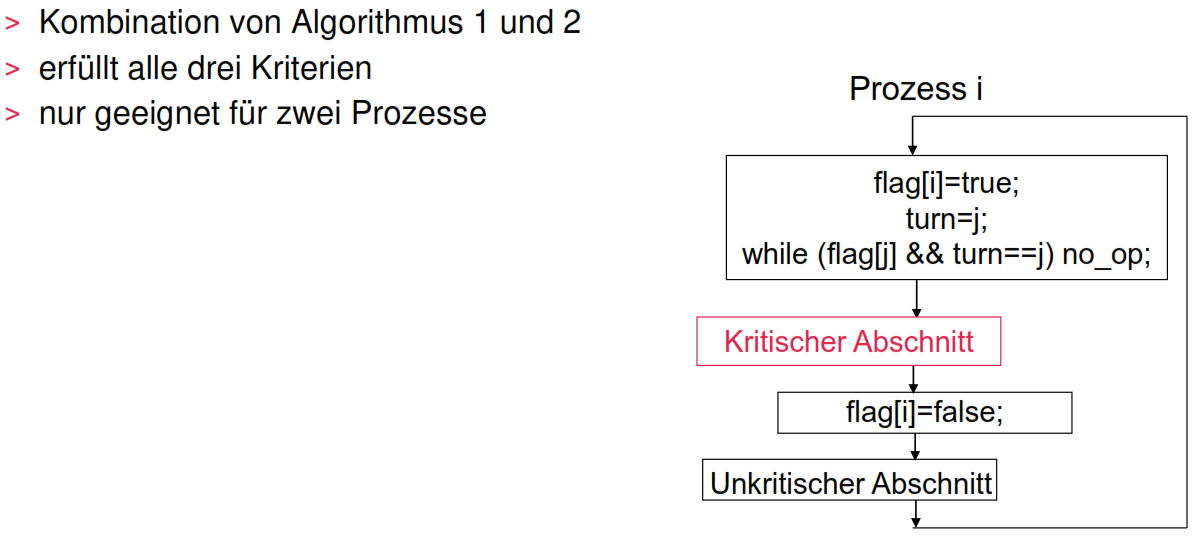
Ein **kritischer Abschnitt** ist eine Folge von Anweisungen oder ein Code-Segment mit Zugriff auf gemeinsame Daten. In einem kritischen Abschnitt darf sich höchstens ein Prozess gleichzeitig befinden.   


Die Anforderungen an Lösungsansätze für dieses Problem sind wechselseitiger Ausschluss (nur ein Prozess gleichzeitig in einem kritischen Abschnitt), Fortschritt (Ist kein Prozess in einem kritischen Abschnitt darf ein Prozess nicht davon abgehalten werden, in einen kritischen Abschnitt einzutreten) und begrenztes Warten (Die Zeit zwischen Anforderung und Gewährung des Eintretens in einen kritischen Abschnitt muss begrenzt sein).

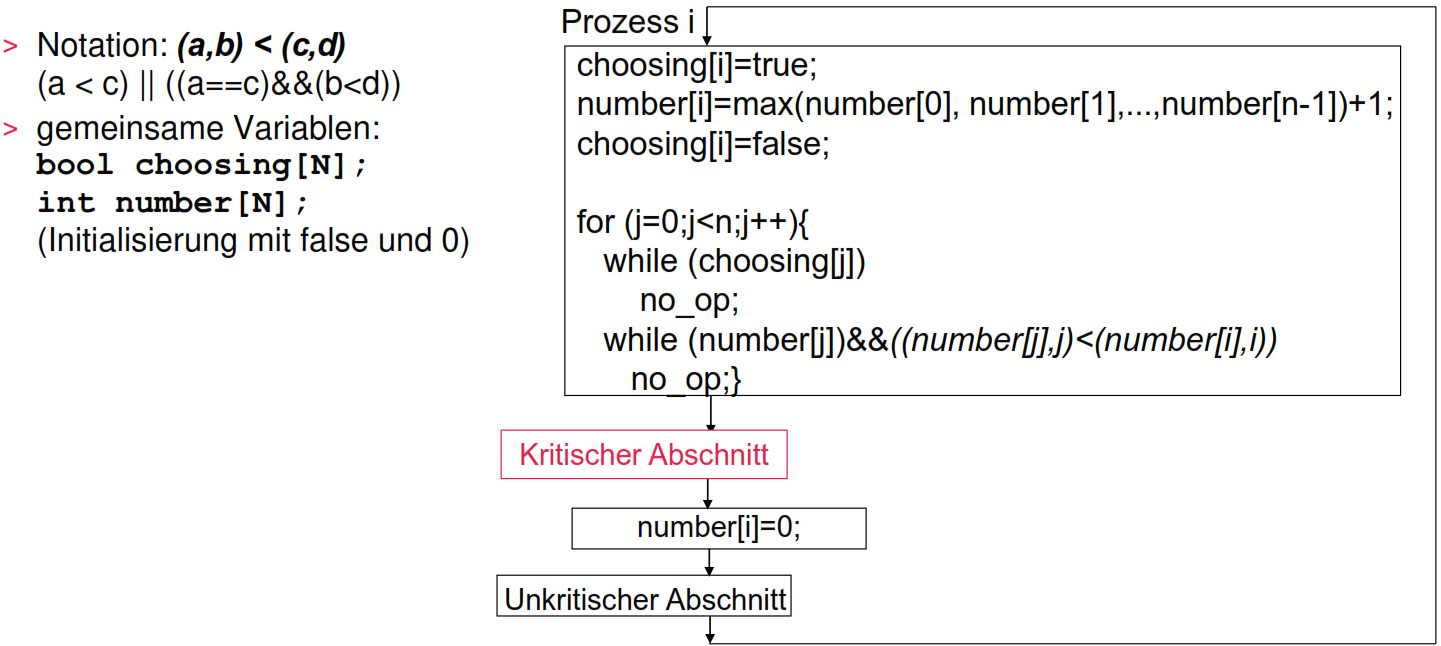
### Algorithmen





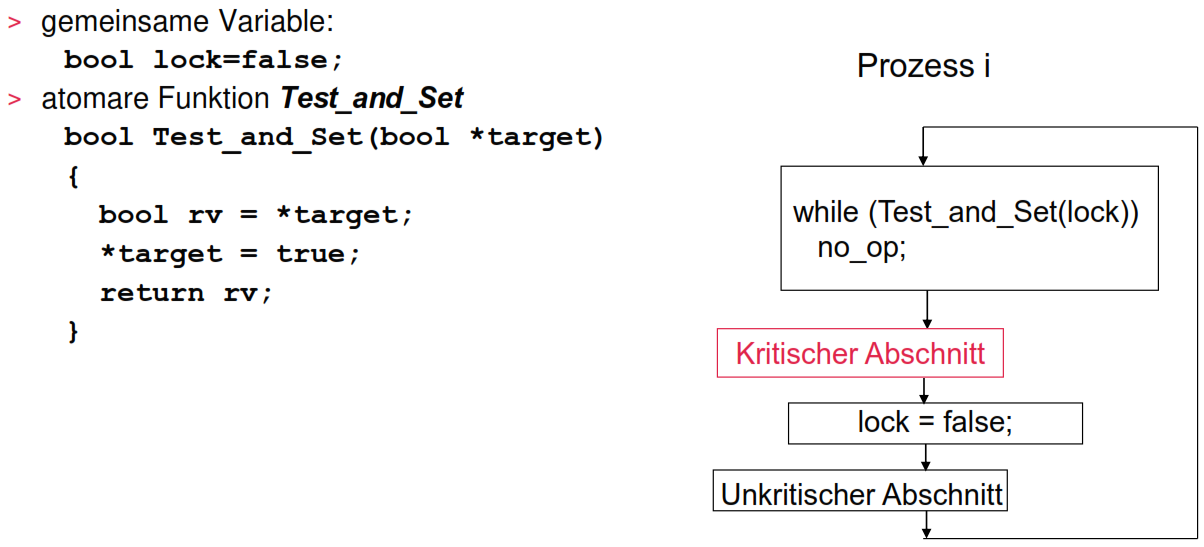


#### Bakery Algorithmus

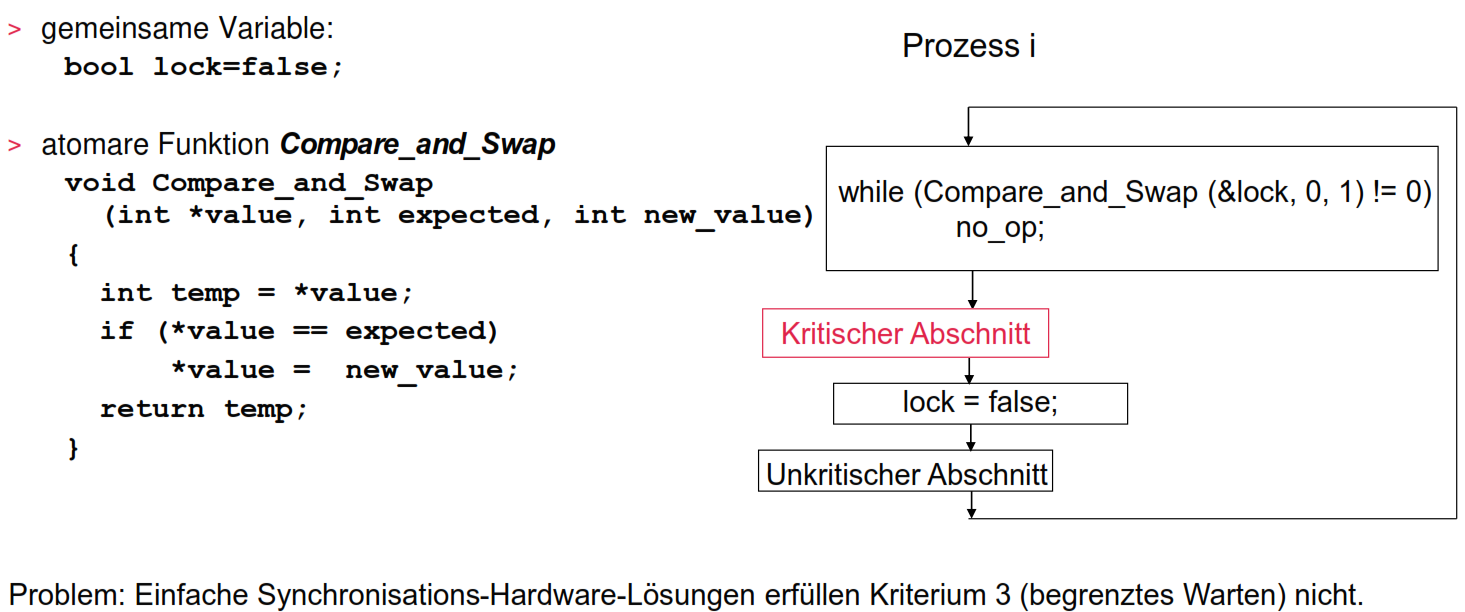
Vor dem Eintreten in einen kritischen Abschnitt erhält ein Prozess eine Nummer (aufsteigende Nummerierung). Der Prozess mit der kleinsten Nummer tritt dann in kritischen Abschnitt ein. Kommt es zu Nummerngleichheit gibt die Prozess ID den Ausschlag.   


### Synchronisations-Hardware

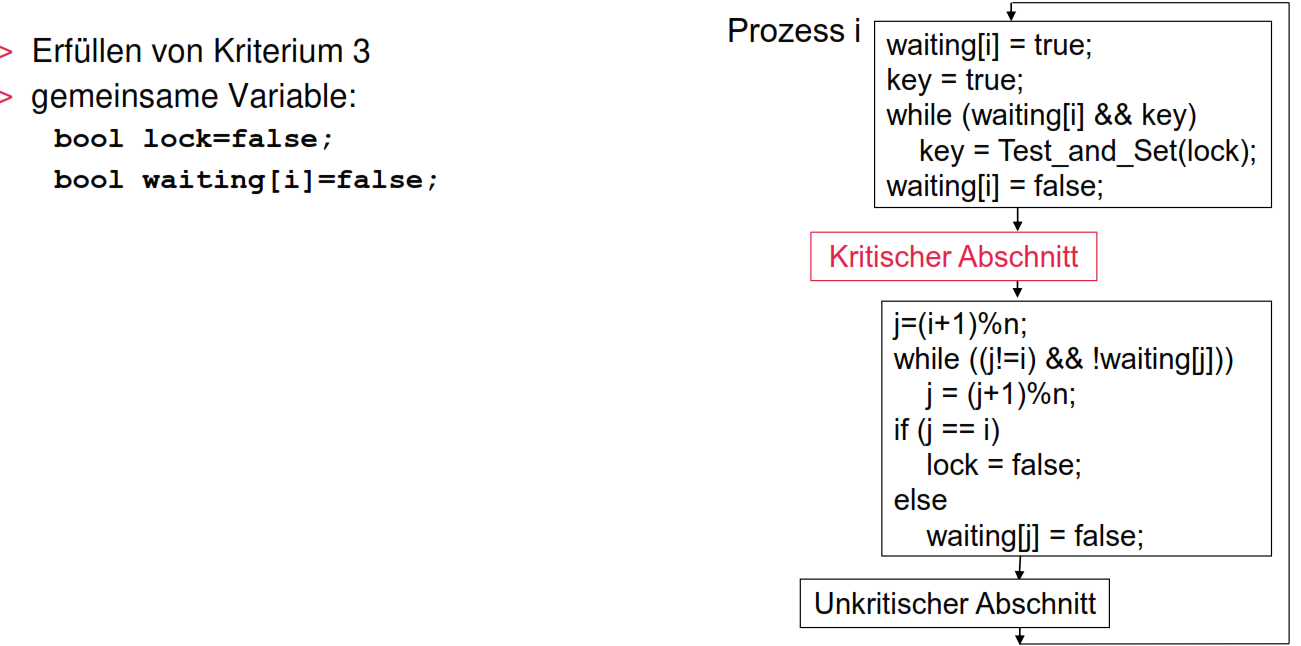
#### Test and Set



#### Compare and Swap



#### Lösung mit Synchronisations-Hardware



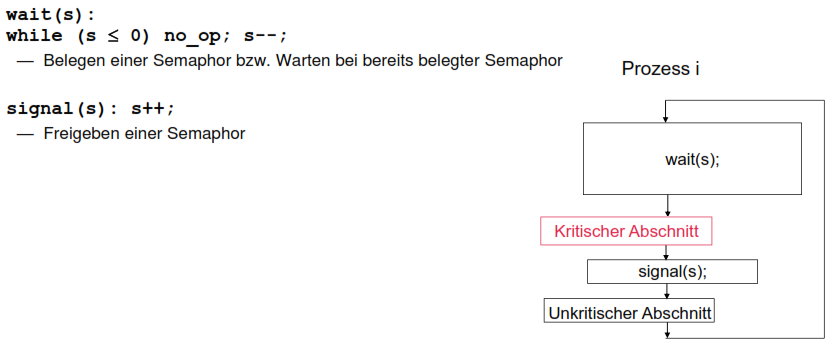
## Semaphore

Eine **Semaphore** ist eine geschützte Variable s, auf der nur die atomaren Operationen *wait(s)* und *signal(s)* ausgeführt werden können. Sie kann Werte grösser 0 (frei) und kleiner gleich 0 (belegt) annehmen.

Eine binäre Semaphore kann nur die Werte 0 und 1 annehmen, wobei sie auf 1 initialisiert wird.

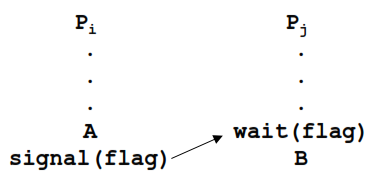
Eine zählende Semaphore kommt bei mehreren Instanzen einer verfügbaren Ressource zum Einsatz. Typischerweise wird sie dabei auf die Anzahl der verfügbaren Ressourcen initialisiert.

### Semaphoreoperationen

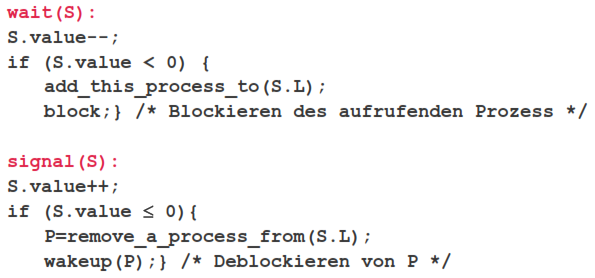


### Synchronisation mit Semaphoren

Wir haben nun ein Code Segment B von Prozess j und das soll auf jeden Fall nach Code-Segment A von Prozess i ausgeführt werden. Nun können wir eine Semaphore mit wait vor B setzten und diese Semaphore erst dann freigeben, wenn A abgearbeitet ist.



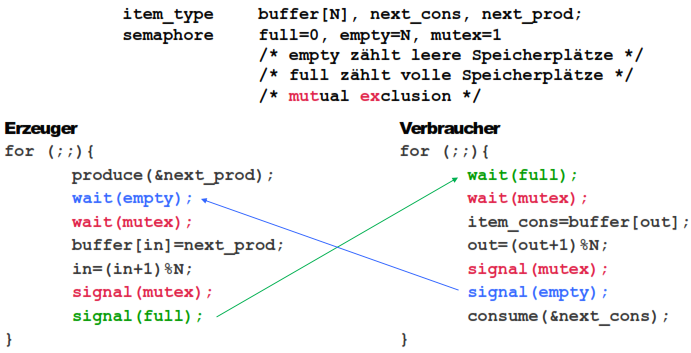
### Implementierung von Semaphoren



### Klassische Synchronisationsprobleme

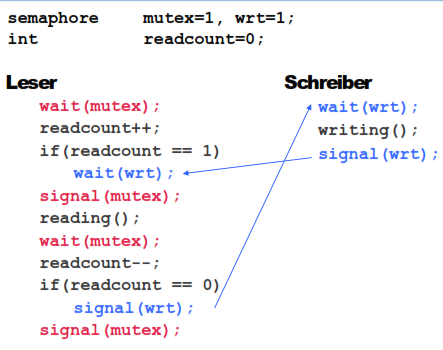
Einige klassische Synchronisationsprobleme sind das Erzeuger/Verbraucher-, Leser/Schreiber- oder Dining Philosophers-Problem.

#### Erzeuger/Verbraucher mit Semaphoren



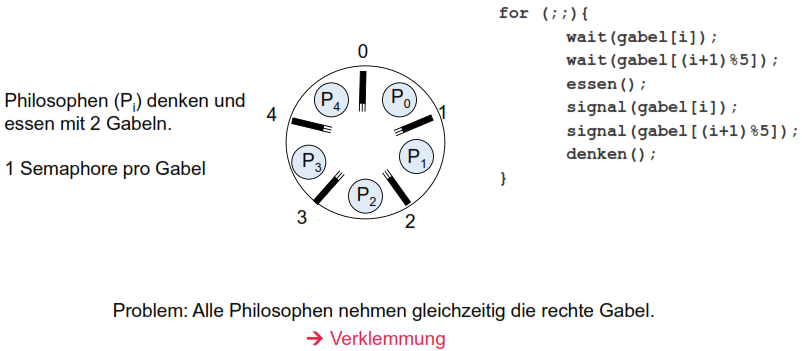
Wir haben hier zu Beginn die Semaphoren empty, die auf die Anzahl Speicherplätze gesetzt wird und full, die auf 0 gesetzt wird. Der Erzeuger wartet nun auf empty. Wenn empty also voll ist, wartet der Erzeuger, aber der Verbraucher kann arbeiten und sperrt mit mutex den Prozessor. Dann gibt er das signal auf empty, wodurch der Erzeuger dann auch wieder arbeiten kann.

#### Leser/Schreiber-Problem



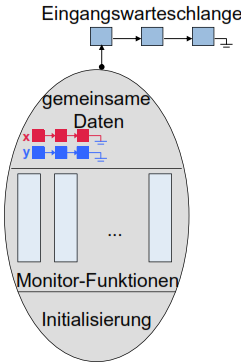
Der Leser sperrt über mutex und schaut dann ob es etwas zu lesen gibt. Wenn readcount auf 1 ist, wartet er darauf, dass der Schreiber schreibt, sperrt dann, liest, setzt readcount auf 0 und gibt wrt für den Schreiber wieder frei.

#### Dining Philosophers

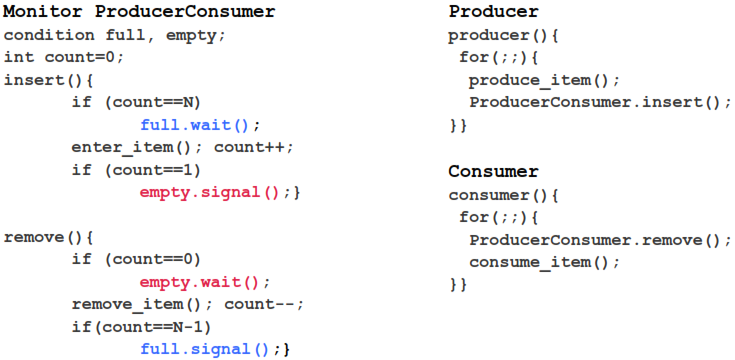


## Monitore

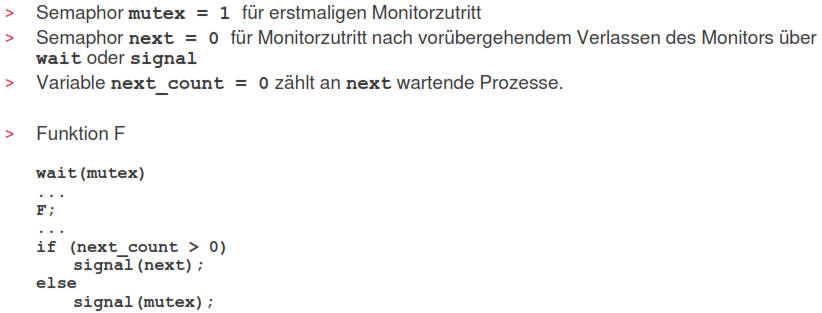
Ein **Monitor** ist eine Sammlung von Prozeduren, Variablen du Datenstrukturen in einem Modul. In einem solchen existiert zu jedem Zeitpunkt nur ein aktiver Prozess. Über Condition-Variable x kann die Behandlung kritischer Abschnitte im Monitor formuliert werden (*x.wait, x.signal*).



#### Erzeuger/Verbraucher mit Monitor

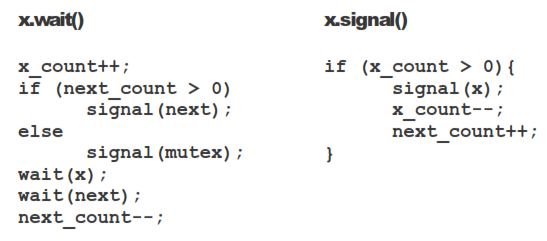


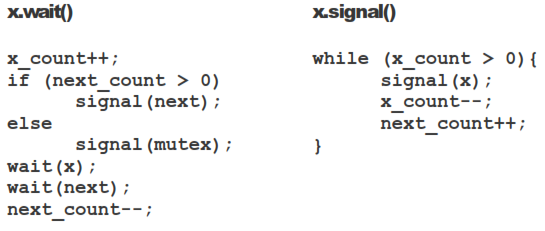
#### Implementierung von Monitoren mit Semaphoren

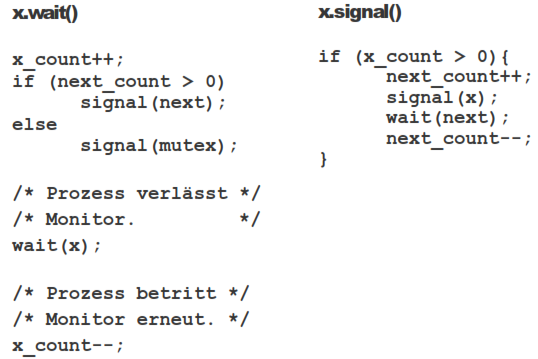


### Signal-Varianten

Für jede Condition-Variable x wird eine Semaphore x und ein zugehöriger Zähler x\_count eingeführt.







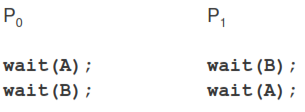
## Transactional Memory

**Memory Transaction** ist eine atomare Sequenz von Lese- und Schreiboperationen. Wenn alle Operationen erfolgreich verlaufen sind, kann die Transaktion per *commit* bestätigt werden, ansonsten kann sie per *abort* abgebrochen und per *rollback* in den Ursprungszustand zurückgebracht werden.

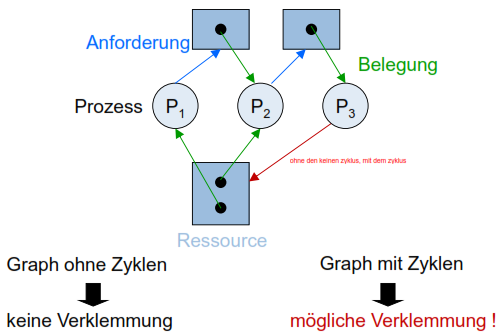
## Verklemmung von Prozessen

### Verklemmungen

Eine Menge von Prozessen ist in einer **Verklemmung (Deadlock)**, wenn jeder Prozess ein Betriebsmittel belegt und darauf wartet, ein von einem anderen Prozess belegtes Betriebsmittel zu belegen.



Eine Verklemmung tritt dann auf, wenn alle der folgenden Bedingungen erfüllt sind:  
Wechselseitiger Ausschluss (Die Ressource kann nur von einem Prozess benutzt werden)  
Halten und Warten (Prozess, der eine Ressource hält wartet auch auf eine Ressource)  
Keine Verdrängung (Die Ressource kann vom haltenden Prozess nur freiwillig abgegeben werden)  
Zirkulierendes Warten (Pi wartet auf durch Pi+1 mod n belegte Ressource.



## Behandeln von Verklemmungen

Eine Verklemmung kann entweder **verhindert** (Methoden, um Zutreffen einer der vier Bedingungen zu verhindern) oder **vermieden** (für jede Ressourcenanforderung wird entschieden, ob dadurch eine Verklemmung auftreten kann).

Unter dem **Aufheben** einer Verklemmung versteht man das Erlauben einer Verklemmung und das Behandeln davon, wenn sie erkannt wird.

### Verhindern von Verklemmungen

Der wechselseitige Ausschluss ist bei teilbaren Resspurcen nicht notwenigen, zB bei read-only Dateien.

Halten und Warten kann verhindert werden, indem ein Prozess nur dann Ressourcen anfordern darf, wenn er aktuell keine belegt.

Keine Verdrängung kann verhindert werden, indem halt doch Ressourcen entzogen werden.

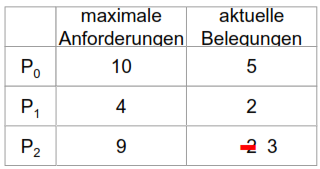
Um das Zirkulierende Warten zu verhindern, muss eine Totalordnung auf die Ressourcen gelegt werden, also die Reihenfolge von Anfragen fix gestaltet werden.

### Vermeiden von Verklemmungen

Die Prozesse Pi beschreiben vorher die maximale Menge benötigter Ressourcen. Eine Anfrage wird nur dann erfüllt, wenn das System danach in einem sicherne Zustand bleibt. Ein System befindet sich dann in einem sicheren Zustand, wenn es eine sichere Sequenz gibt. Eine Sequenz ist dann sicher wenn jeder Request von Pi durch verfügbare Ressourcen und belegte Ressourcen von vorher terminierenden Prozessen erfüllt werden kann.

#### Safe State Algorithmus

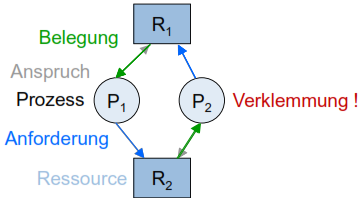
Betrachten wir diesen Algorihtmus an einem Beispiel: Wir haben 12 Bandlaufwerke. Zum Zeitpunkt t0 gelten folgende Belegungen:



Die Kette <P1, P0, P2> ist sicher, denn P1 kann nun zwei der drei freien Ressourcen belegen und dann freigeben. Dann sind 5 Ressourcen frei. Nun kann P0 5 weitere Ressourcen belegen und dann freigeben, dadurch werden 10 Ressourcen frei. Schliesslich kann P2 die benötigten 7 weiteren Ressourcen belegen. Danach haben alle Prozesse terminiert.

#### Ressourcenbelegungsgraph-Algorithmus

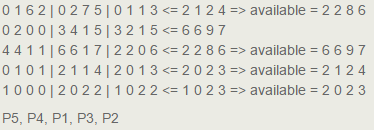
Dieser Graph funktioniert für Systeme mit genau einer Instanz für jede Ressource. Der graue Pfeil dokumentiert einen Anspruch, dass ein Prozess möglicherweise eine Ressource belegen wird. Bei einer Anforderung wird er zum blauen Anforderungs-Pfeil. Wird die Ressource dann freigegeben, wird der Anforderungs-Pfeil zum grünen Belegungs-Pfeil. Bei einer Freigabe wird der Belegungs-Pfeil zum Anspruchs-Pfeil.



#### Bankers-Algorithmus

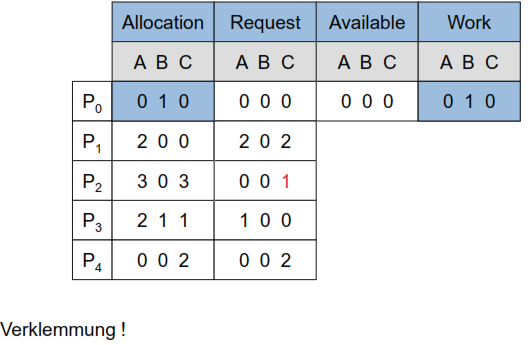
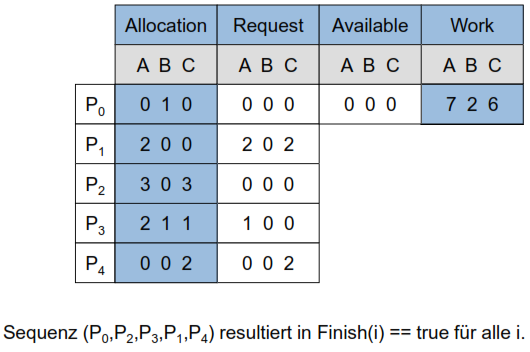
Dieser Algorithmus ist geeignet für Ressourcen mit mehrfachen Instanzen. Dabei beschreibt jeder Prozess im Voraus seinen maximalen Anspruch.

Nun haben wir die Alloc (wie viele Ressourcen jeweils belegt)-, Max (wie viele Ressourcen maximal belegt)- und Need (wie viele müssen noch belegt werden)-Matrix. Dabei gilt: Max-Alloc=Need. Dazu wissen wir, wie viele Ressourcen wir zur Verfügung haben. Das ist die available Matrix. Im untenstehenden Beispiel starten wir mit available = <1 0 2 3>. Wir vergleichen dann mit der Need-Tabelle und sehen, dass nur P5 infrage kommt, da dessen Need in allen Ressourcen kleiner als available ist. Nun wird P5 abgearbeitet und wir können Alloc[P5] zu available addieren. Nun fahren wir so mit den restlichen Prozessen weiter.



### Verklemmungsalgorithmus

Im Gegensatz zum Safety-Algorithmus vergleichen wir nun mit Request, statt mit Need. Das ist optimistischer Ansatz mit der Annahme, dass keine weiteren Ressourcenanforderungen nötig werden).



### Aufheben von Verklemmungen

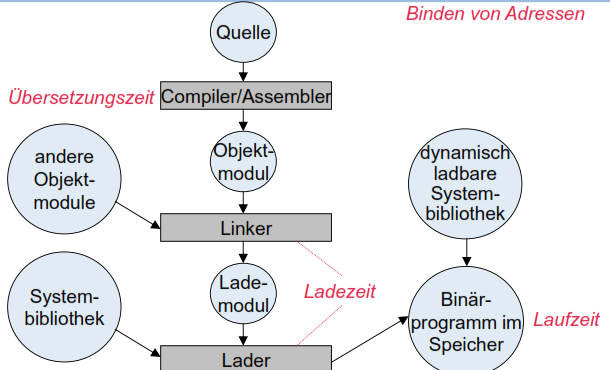
Um die Prozesse bei Verklemmungen zu beenden, können alle verklemmten Prozesse beendet werden oder damit begonnen werden, einen Prozess nach dem anderen zu beenden bis die Verklemmung aufgehoben ist.

Ansonsten können den Prozessen auch Ressourcen entzogen werden, oder ein Rollback der Prozesse in einem Deadlock vorgenommen werden. Das wäre eine Rückkehr in einen sicheren Zustand und erneutes Starten der Prozesse.

# Hauptspeicherverwaltung

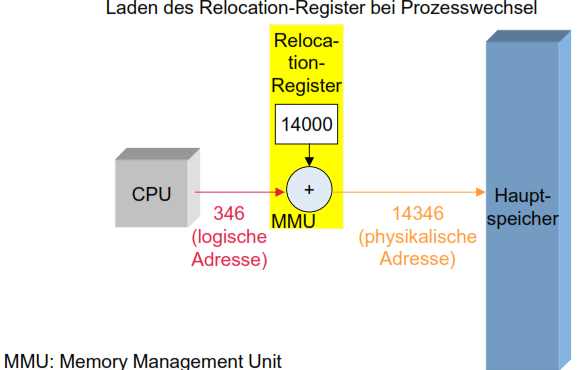
### Binden von Speicheradressen

Um ein Programm ausführen zu können, muss es in den Hauptspeicher geladen werden. Die Speicheradressen können dazu entweder zu Übersetzungszeit, Ladezeit oder Ausführungszeit (Lauzeit/Runtime) gebunden werden.



### Logische und Physische Adressen

Die **logische Adresse** ist die Adresse, die ein Programm sieht. Mit der **Memory Management Unit** werden diese logischen Adressen dann in physische Adressen umgewandelt. Damit ist es möglich, dass mehrere Prozesse die gleichen logischen, aber unterschiedlichen physischen Adressen verwenden.



### Dynamisches Laden

Eine **Routine** ist eine Folge von Anweisungen.

Nicht benötigten Routinen werden nicht geladen, erst wenn sie aufgerufen werden. Die aufrufende Routine prüft dabei, ob die aufzurufende Routine geladen ist. Falls nicht, wird der Lader damit beauftragt, die Routine in den Hauptspeicher zu laden und die Adresstabellen zu modifizieren. Das ist nützlich, wenn grosse Codesegmente nur selten benötigt werden.