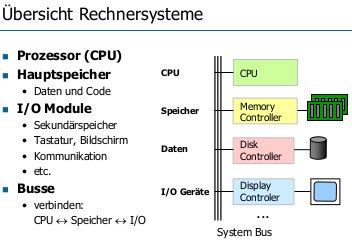
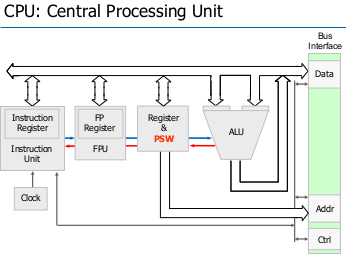
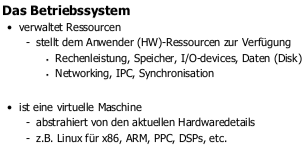
**Zusammenfassung BSY**

*René Bernhardsgrütter, 27.03.2013*

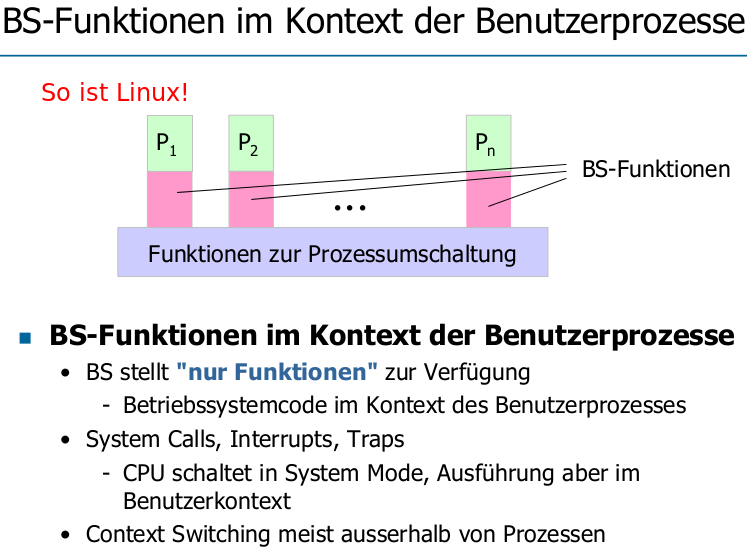
Betirebssystem / Computer

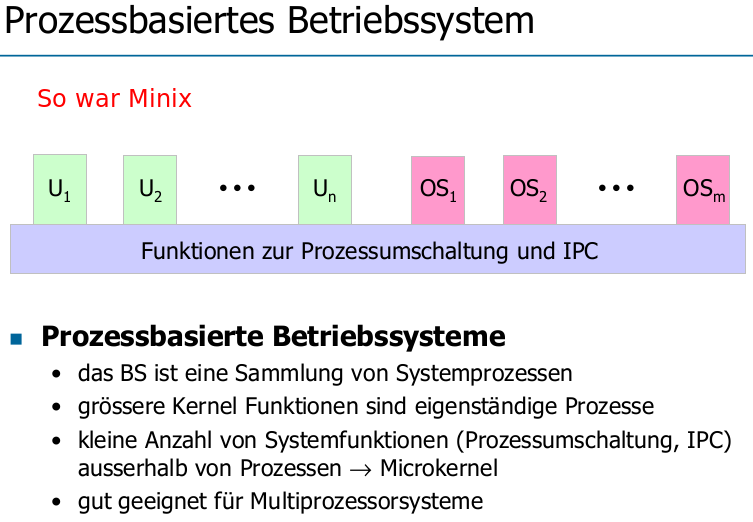


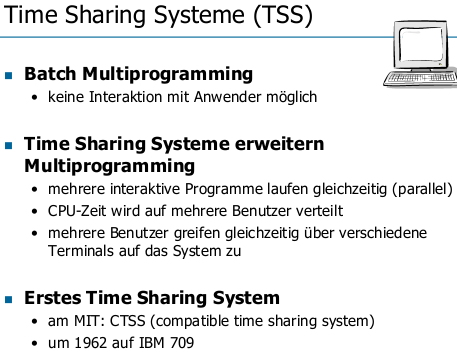




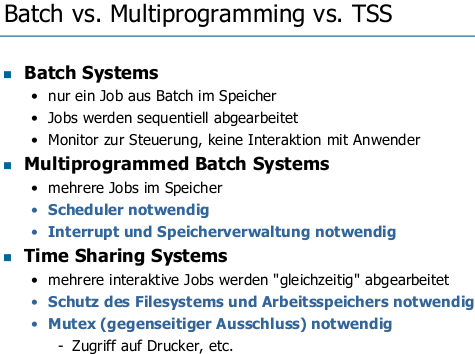
Dienste: Programmausführung (laden, init IO, Fileaccess), File-IO, Systemzugriff (kontrolliert Zugriffe auf Ressourcen und Daten), Interprozesskommunikation, Fehlererkennung (interne und externe Hardware, Software, Zugriff auf nicht vorhandene Dienste), Fehlerbehandlung (Fehlermeldungen an Applikation, Operation nochmals starten, Applikation killen), Accounting (Userkonten, Statistiken zur Ressourcennutzung, Uberwachung).







Monitor (Software): Ist ein Programm, das andere Programme, Speicher/IO, Cache, MMU, Disk-Controller, etc initialisiert. Im Batchbetrieb sagt er, welches Programm als nächstes läuft. Bei Komplexen Systemen könnte das das BIOS sein.



**Grundkonzepte für BS-Design:**

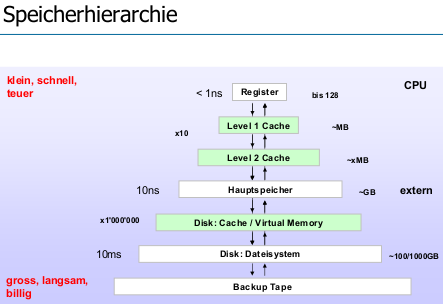
1. Prozesse
2. Scheduling und Ressourcenverwaltung
3. Speicherverwaltung (MemoryMgmt + FS)
4. Schutz und Sicherheit von Informationen
5. Systemarchitektur

**IO:**

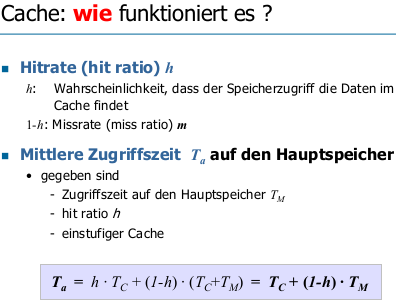
- Programmed: pollt das IO-Modul, Status Flags geben Verfügbarkeit von IO-Modul an => BusyWait, 100% Last

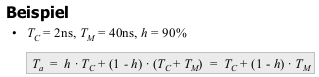
- Interruptgesteuert: Kein BusyWait, benötigt Rechenzeit, weil CPU alles lesen / schreiben muss

- DMA: CPU richtet Anfrage an DMU-Modul => diese transferiert Datenblock direkt in Speicher, async zu CPU. CPU nur zu Beginn und Ende des Transfers nötig.

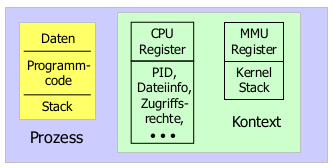


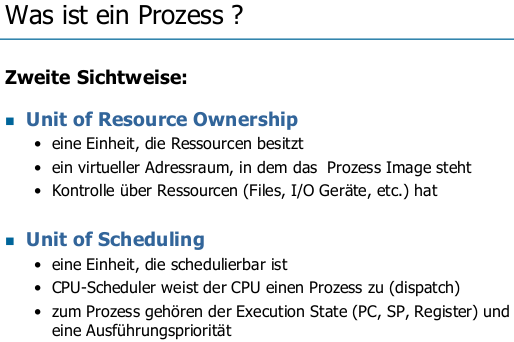
Lokalitätsprinzip: Räumliche Lokalität (Grosse Wahrscheinlichkeit, dass nächster Zugriff auf ‚nahe‘ Daten stattfindet). Zeitliche Lokalität (Grosse Wahrscheinlichkeit, dass Speicherzugriff auf gleiches Datum nochmals stattfindet).





Prozesse

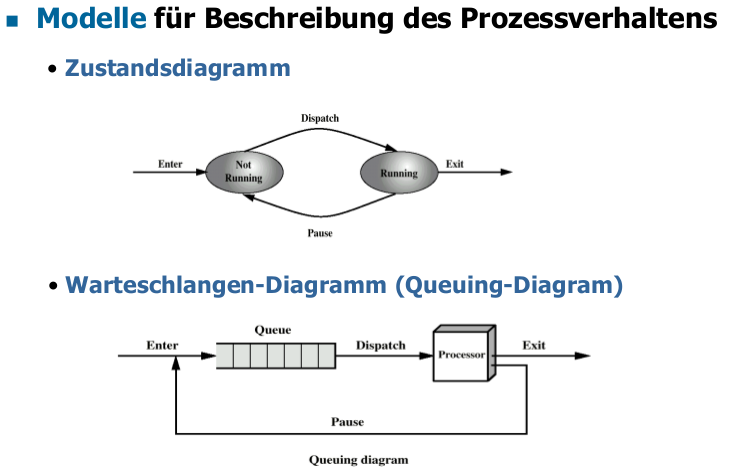


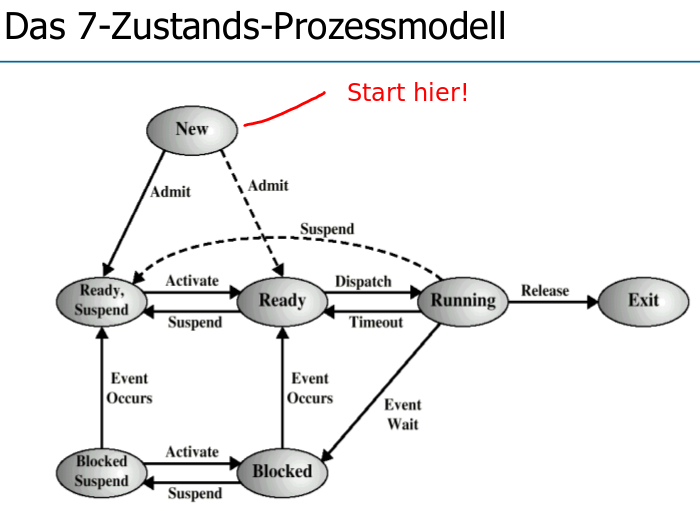


Was muss bei Umschaltung ausgetauscht / gespeichert werden? a) Registerinhalte b) Informationen zum Zustand des Prozesses c) die drei Speicherbereiche/-segmente.

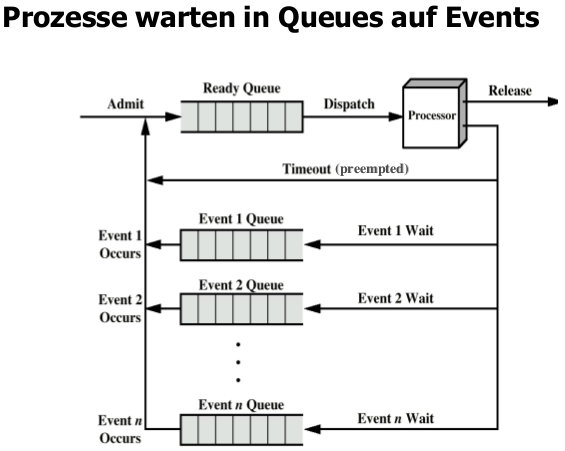
Punkt a)+b) = **Prozesskontext**

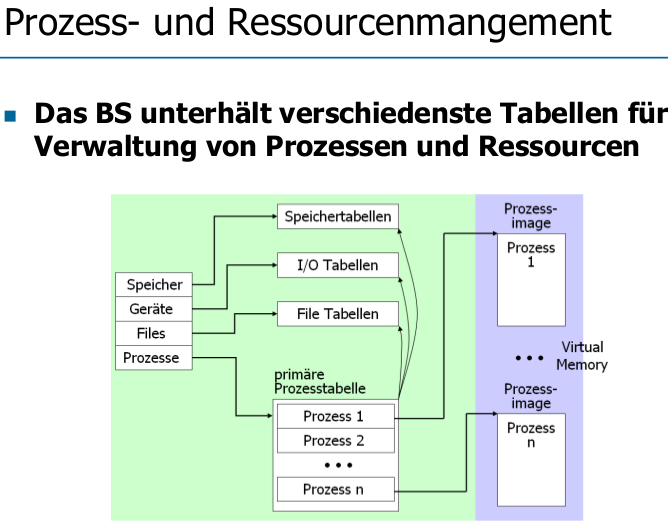
Zu einem Prozess gehören alle Daten, die benötigt werden, damit ein Programm auf einem Rechner ausgeführt werden kann, zu einem beliebigen Zeitpunkt unterbrochen werden kann und wieder dort gestartet werden kann, wo es unterbrochen wurde.





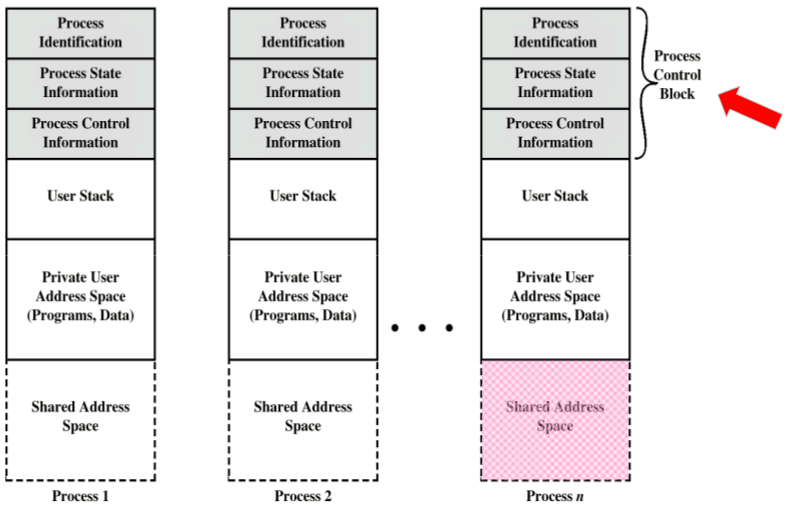
New: erzeugt, aber noch nicht gestartet. Ready: Kann ausgeführt werden. Running: ist aktiv. Blocked: Wartet auf Ereignis, z. B. IO. Suspend Blocked: Wurde ausgelagert, wartet auf Ereignis. Suspend Ready: ist ausgelagert, aber bereit für Aktivierung.

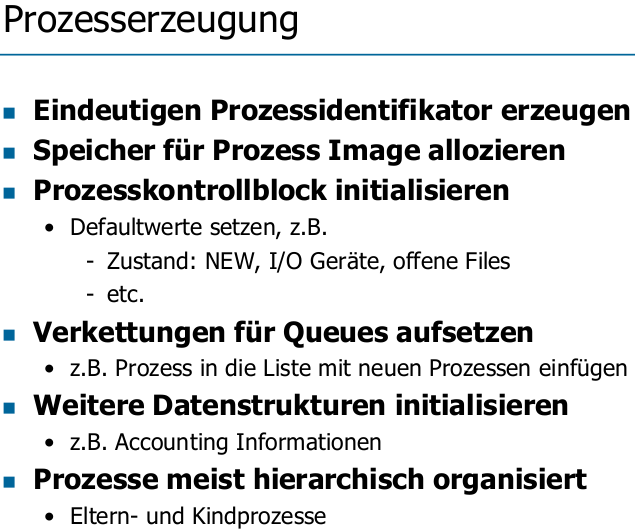




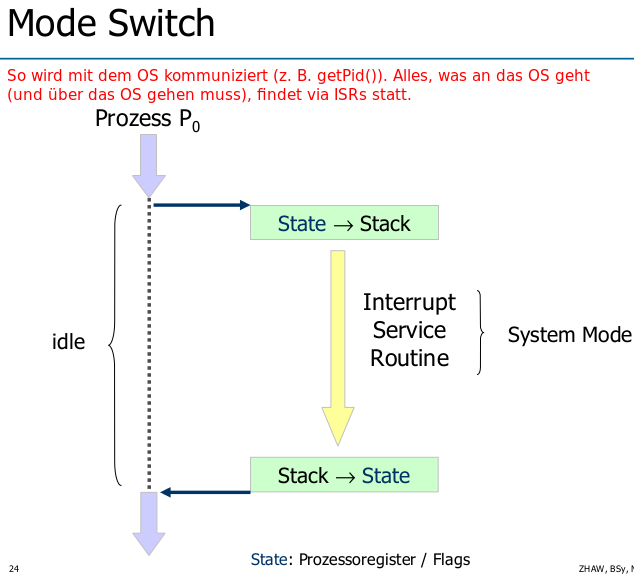
**PCB (Process Control Block) ist sehr wichtige Datenstruktur im BS:** Speichert Prozesskontext, ist Teil des Prozessimages. Prozessinformationen: PID, Process State Information, Process Control Information.

Prozess Image im virtuellen Memory:

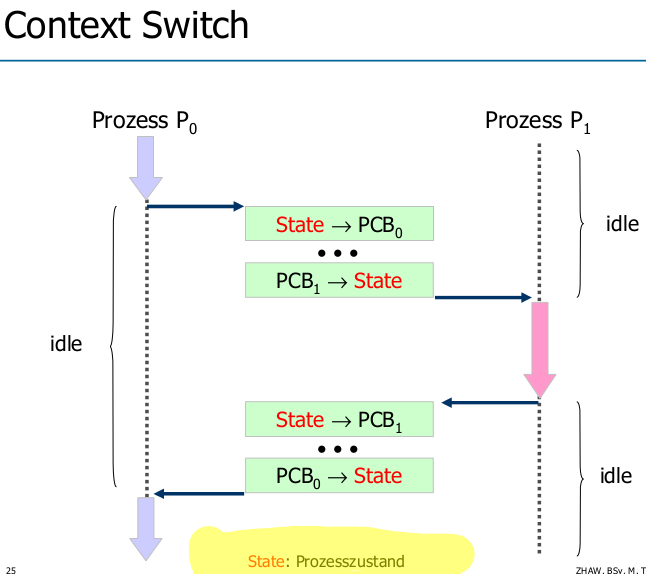




Zwei Modi: UserMode (weniger priviligiert für Anwenderprogramme), SystemMode (priviligierter für OS-Funktionen).



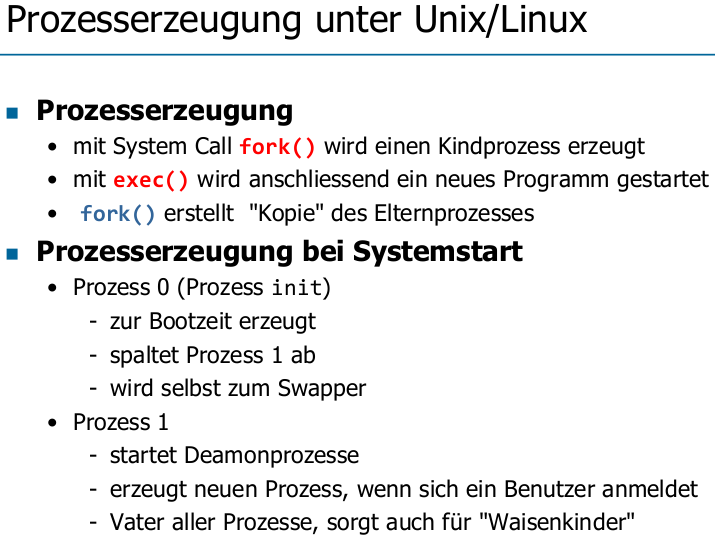
Man kommt vom UserSpace nur via Interrupt Service Routine an den System Mode. Dort ‚beantworet‘ das OS die Anfrage, die der User hatte. Der User kann nie selbst dort Code ausführen. Das ist effizienter als ein Contextswitch, denn es werden nur die Flags gesichert.



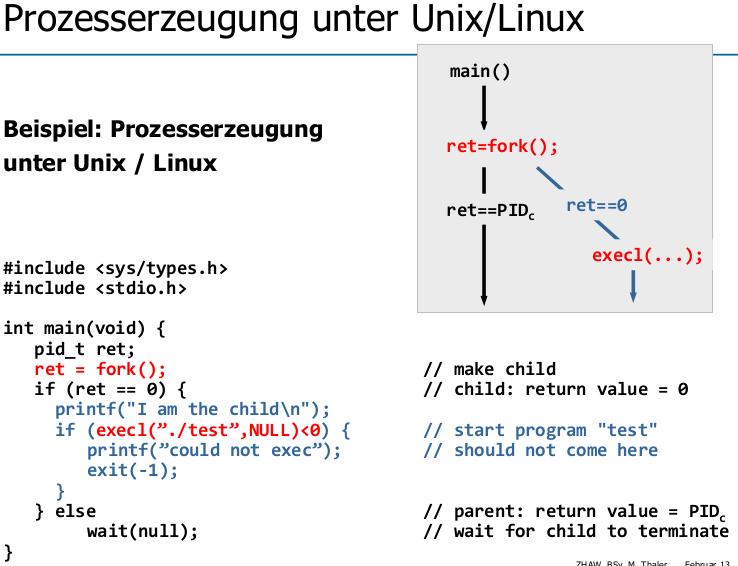
Prozessumschaltung: Gesamter Prozesszustand muss gespeichert werden.

Dies kann durch Programm (IO blocking, etc), OS oder äussere Interrupts passieren.

Prozesserzeugung



Elternprozess kann mit den Kindern arbeiten, z. B. auf die Terminierung warten. Kinder, die terminieren und auf die der Elternprozess nicht wartet, werden zu Zombies. Zombies belegen Betriebssystemressourcen und müssen verwaltet werden: während diesem Zustand können z. B. Accountinginformationen abgefragt werden.



Exec(..) überlagert das Programm und den Datenbereich mit neuem Programm und neuem Datenbereich. Prozesskontext wird geerbt (kann z. B. weiterhin auf geöffnete Files zugreifen).

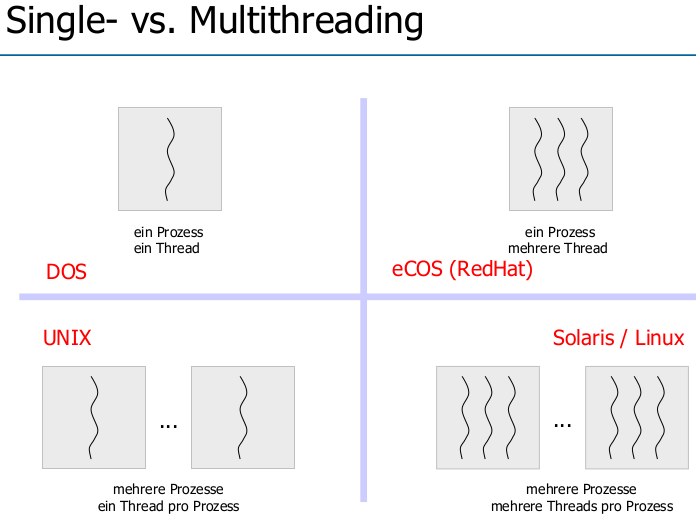
fork() < 0 => Fehlgeschlagen

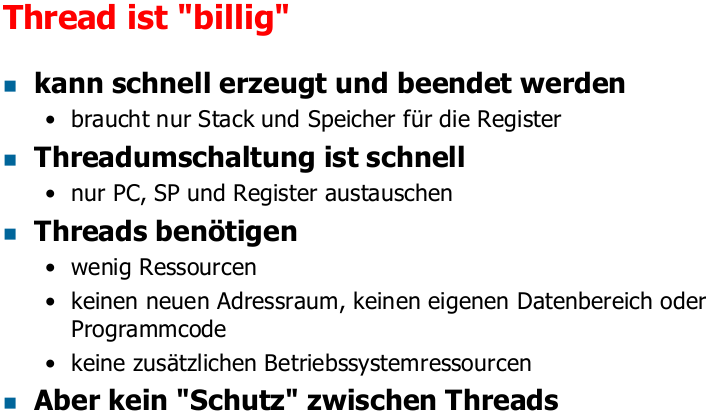
fork() > 0 => Mutterprozess

fork() == 0 => Kindprozess

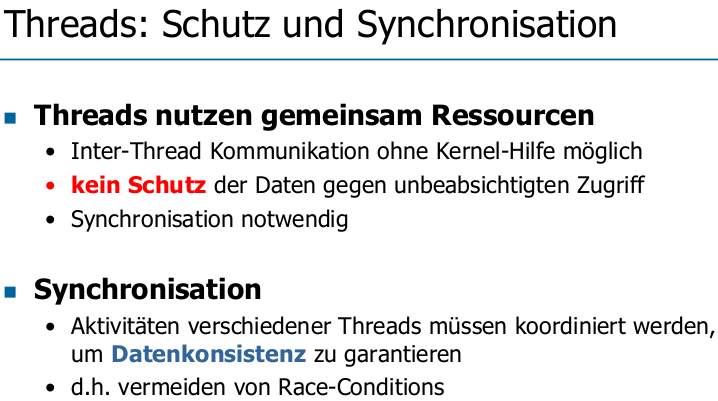
**FORKS FARBIG ZEICHNEN!**

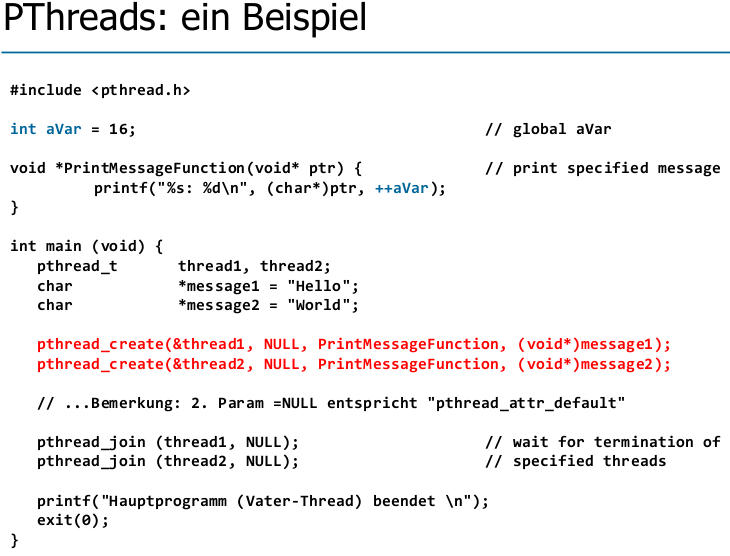
Threads

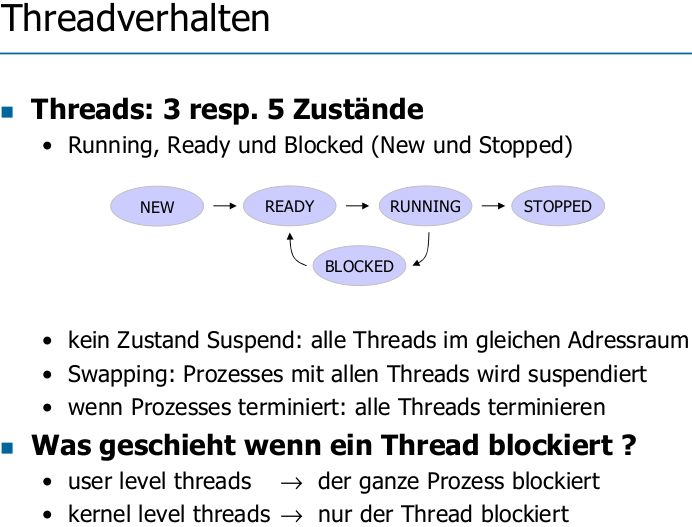


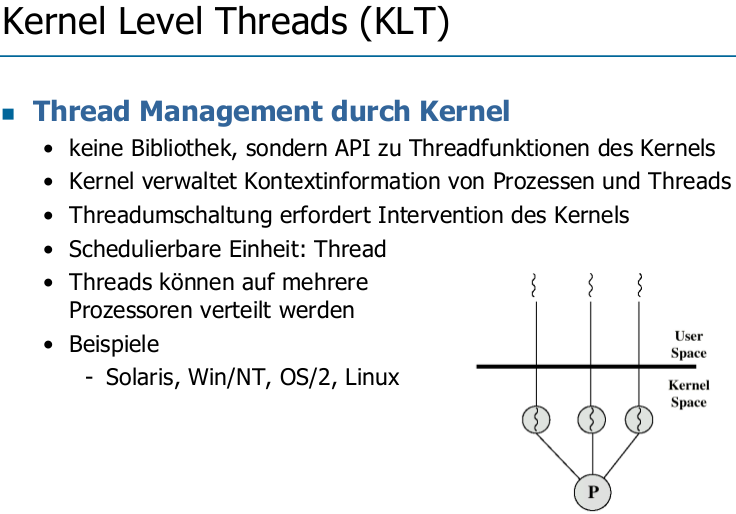


Wenn ein Thread (z. B. wg. IO) blockiert, muss nur dieser warten (gilt nur für Kernelthreads).

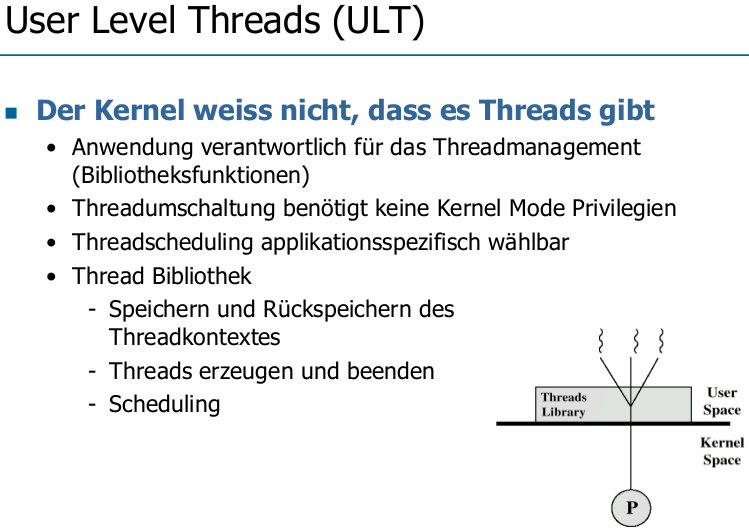




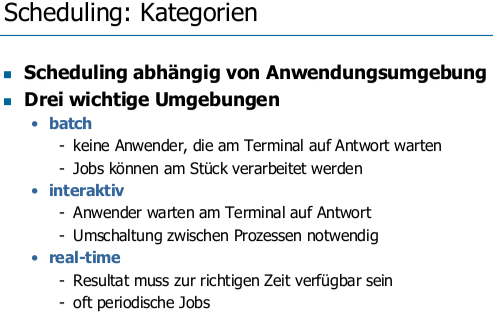




Scheduling auf Threadbasis; Geeignet für oft blockierende Anwendungen (z. B. Server)



Prozess-Scheduling



Batch: non-preemitve, also ohne Unterbrechung druch Scheduler, nur, wenn Prozess blockiert

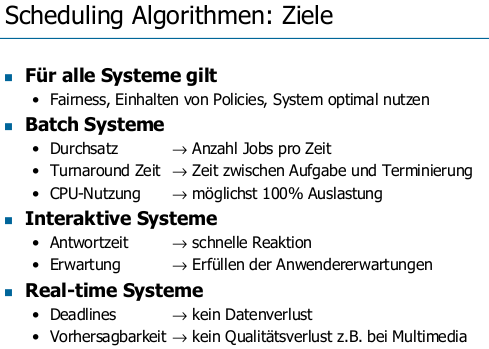
**Definitionen**:

Tb = Rechenzeit (Bedienzeit)

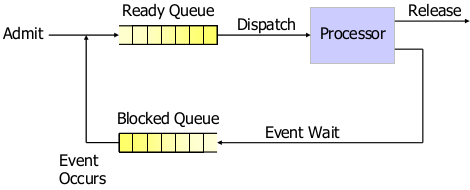
Tw = Wartezeit, bis das erste Mal ausgeführt wird

Ta = Turnaroundzeit (Zeit zwischen Prozessaufgabe und Prozessterminierung)

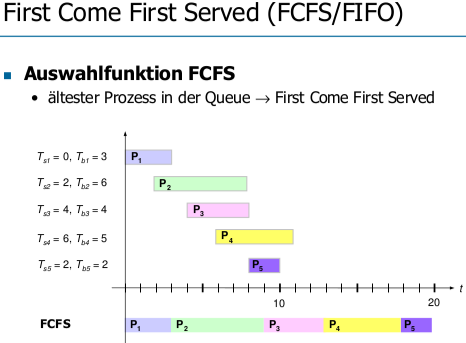
Ts = Starzeit



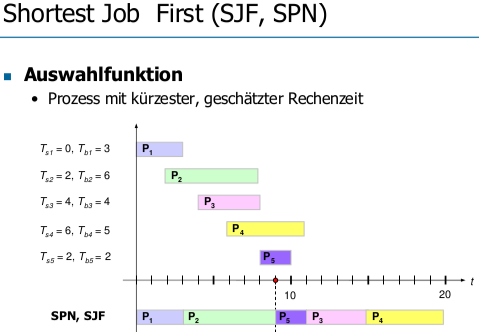
Non-Preemptive Scheduling



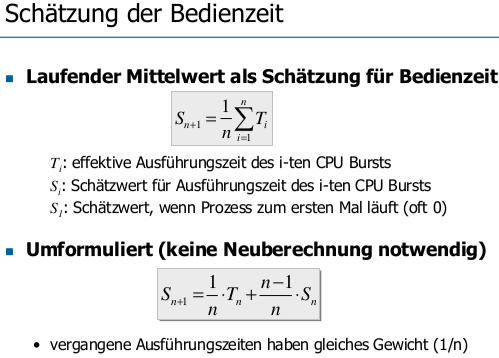
Nichtblockierende Prozesse am Stück abgearbeitet. Blockierende Prozesse in Blocked queue, warten dort, bis Ready Queue leer.

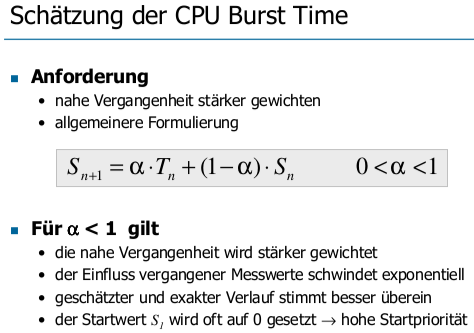


Prozess läuft bis er terminiert oder blockiert. Lange Antwortzeit, hoher Durchsatz, schlechte Ausnutzung von IO, verhungern nicht möglich.

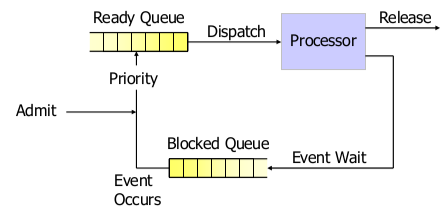


Prozess mit kürzesterwarteten Rechenzeiten zuerst. Gut für kurze Prozesse, schlecht für lange. Hoher Durchsatz, bestraft lange Prozesse, lange Prozesse können verhungern.



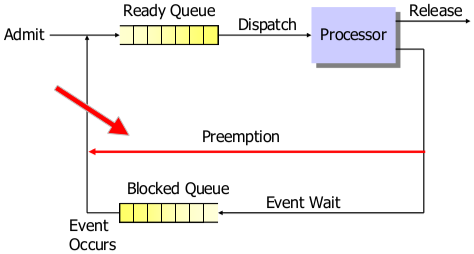


Priority Scheduling

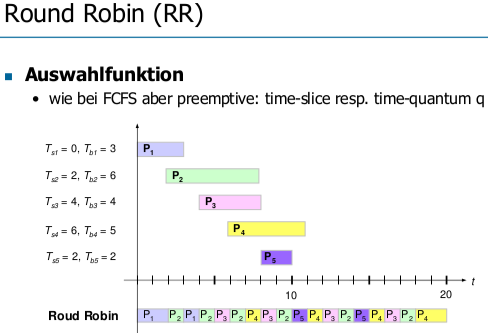


Non-preemptive, nach Priorität, Prozesse werden anhand der Priorität in die ReadyQueue eingefügt. Prozesse mit tiefer Priorität können verhungern.

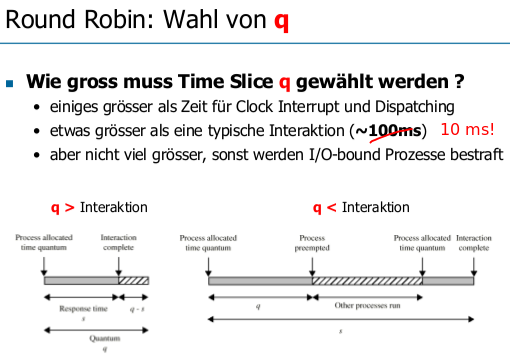
Preemptive Scheduling

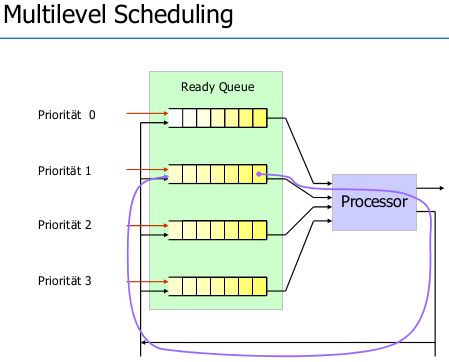


Prozesse können vom Scheduler unterbrochen und wieder in die ReadyQueue eingereiht werden. Unterbruch i.d.R. nach time-slice Ablauf oder bei Blocking.



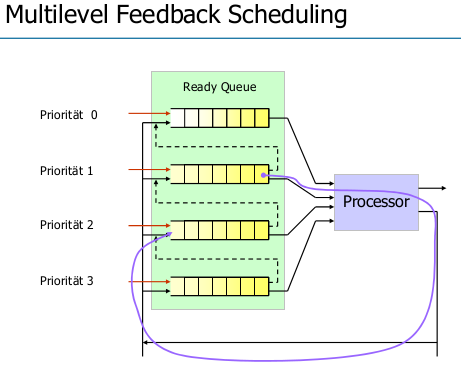
Jobs können nicht verhungern, Response für kurze Prozesse gut. „q“ = time quantuum = time slice. Die Länge dessen sagt, wie lange ein Prozess am Stück laufen darf.Unfair gegenüber Prozessen, die viel IO machen, denn die können oft nicht das ganze q ausnutzen. Round Robin ineffizient, wenn q zu klein (zuviele Context Switches).





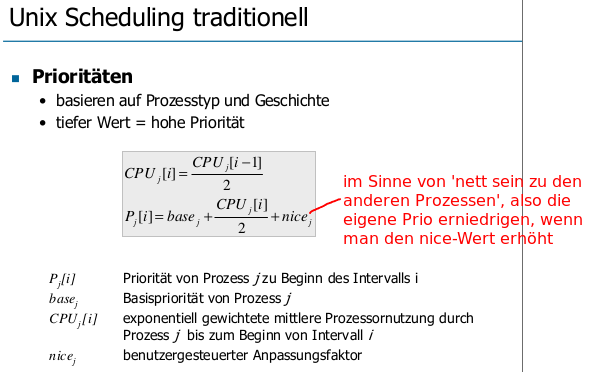
Jobs erhalten je nach Art verschiedene Prios. Scheduler wählt immer Job mit höchster Prio. ReadyQueues: Round Robin -> FIFO in jeder Queue.

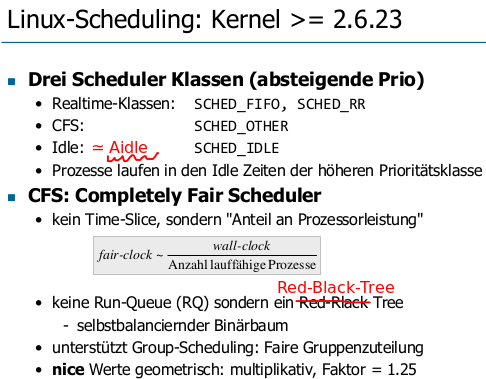
Jobs mit tiefen Prios können verhungern. Abhilfe: Dynamische Prios.

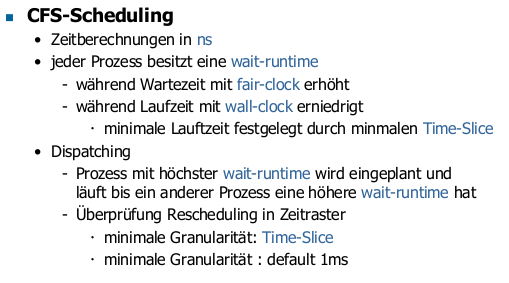


Wie Multilevel Scheduling, aber:CPU-lastige Prozesse sinken von Queue zu Queue. Unterste Queue ist Round Robin. (Lange) Prozesse können immernoch verhungern. Lösung: Dynamische Prioritäten (Prio bei zu langer Wartezeit erhöhen -> Prozesse steigen von Queue zu Queue).

Scheduling in der Praxis



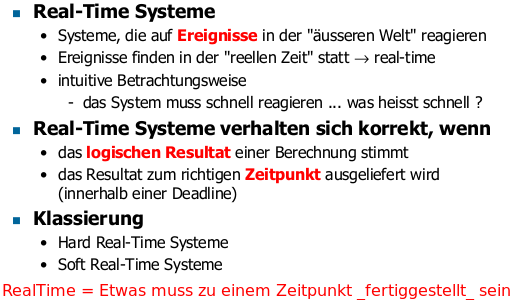


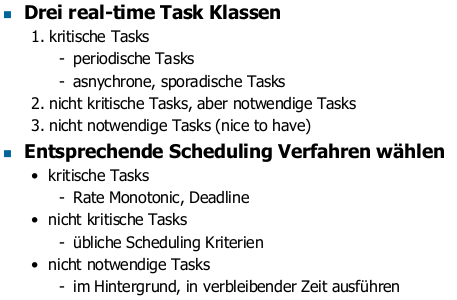


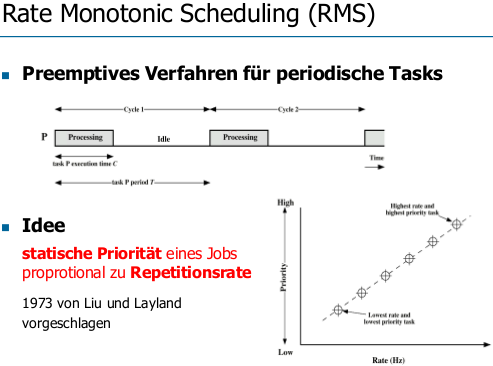
Real-Time Systeme

**RT-Linux:**

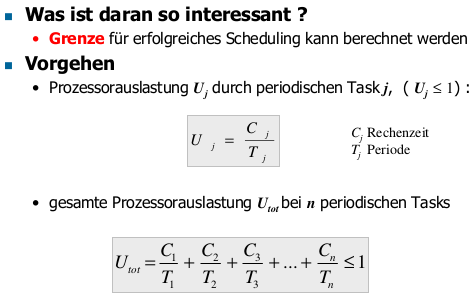
Antwortzeit in Mrikosekunden. Kritische RT-Tasks sind Foreground-Tasks. Nichtkritische RT-Tasks sind Background-Tasks.u



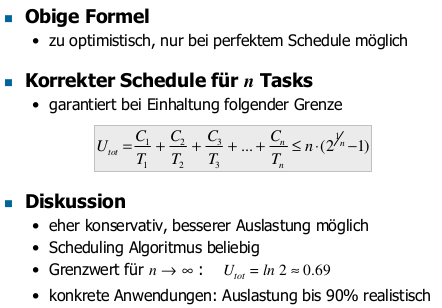




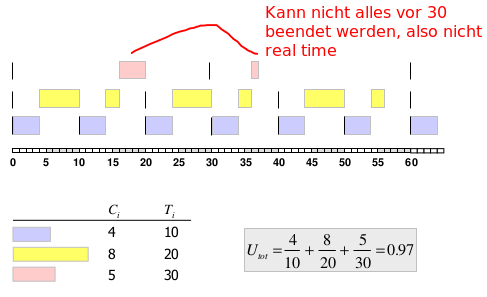
Tasks müssen periodisch und unterbrechbar (preemptive) sein. Priorität proportional zur Repetitionsrate. Statistische Zuweisung von Priorität: Tasks müssen unabhängig sein, sie dürfen nicht aufeinander warten müssen (z. B. wg. Resultat).



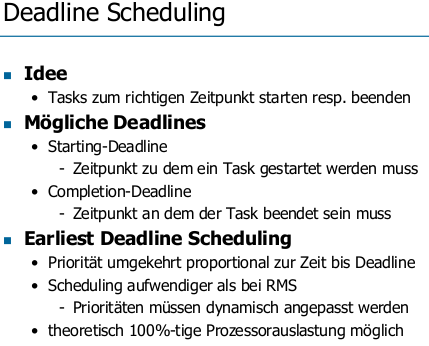
Utot muss <= 1 sein

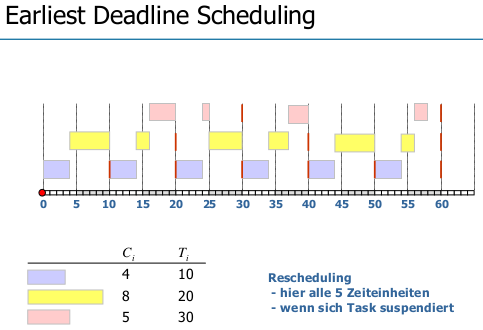


Negativ: für grosse n nur 70% CPU Auslastung.

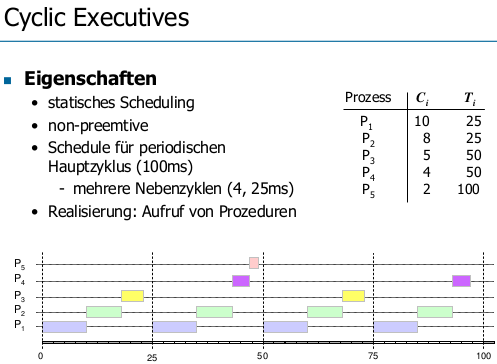


Die Repetitionsrate des Schedulers ist das kgV.





Rescheduling alle 5 Zeiteinheiten.



Manchmal schwierig, einen passenden Schedule zu finden.