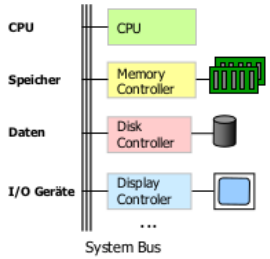
Zusammenfassung BSY

René Bernhardsgrütter, 27.03.2013

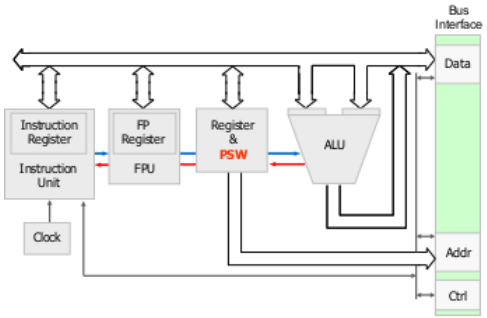
Betirebssystem / Computer

Übersicht Rechnersysteme

- **Prozessor (CPU)**
- **Hauptspeicher**
  - Daten und Code
- **I/O Module**
  - Sekundärspeicher
  - Tastatur, Bildschirm
  - Kommunikation
  - etc.
- **Busse**
  - verbinden:
    CPU ↔ Speicher ↔ I/O



CPU: Central Processing Unit



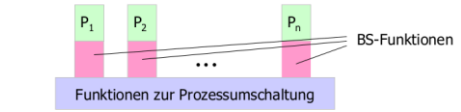
Das Betriebssystem

- verwaltet Ressourcen
  - stellt dem Anwender (HW)-Ressourcen zur Verfügung
    - Rechenleistung, Speicher, I/O-devices, Daten (Disk)
    - Networking, IPC, Synchronisation
- ist eine virtuelle Maschine
  - abstrahiert von den aktuellen Hardwaredetails
  - z.B. Linux für x86, ARM, PPC, DSPs, etc.

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, Überwachung).

BS-Funktionen im Kontext der Benutzerprozesse

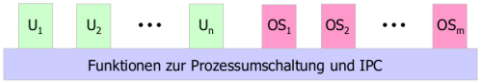
So ist Linux!



- **BS-Funktionen im Kontext der Benutzerprozesse**
  - BS stellt **"nur Funktionen"** zur Verfügung
    - Betriebssystemcode im Kontext des Benutzerprozesses
  - System Calls, Interrupts, Traps
    - CPU schaltet in System Mode, Ausführung aber im Benutzerkontext
  - Context Switching meist ausserhalb von Prozessen

Prozessbasiertes Betriebssystem

So war Minix



- **Prozessbasierte Betriebssysteme**
  - das BS ist eine Sammlung von Systemprozessen
  - grössere Kernel Funktionen sind eigenständige Prozesse
  - kleine Anzahl von Systemfunktionen (Prozessumschaltung, IPC) ausserhalb von Prozessen → Microkernel
  - gut geeignet für Multiprozessorsysteme

Time Sharing Systeme (TSS)

- **Batch Multiprogramming**
  - keine Interaktion mit Anwender möglich
- **Time Sharing Systeme erweitern Multiprogramming**
  - mehrere interaktive Programme laufen gleichzeitig (parallel)
  - CPU-Zeit wird auf mehrere Benutzer verteilt
  - mehrere Benutzer greifen gleichzeitig über verschiedene Terminals auf das System zu
- **Erstes Time Sharing System**
  - am MIT: CTSS (compatible time sharing system)
  - um 1962 auf IBM 709

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.

Batch vs. Multiprogramming vs. TSS

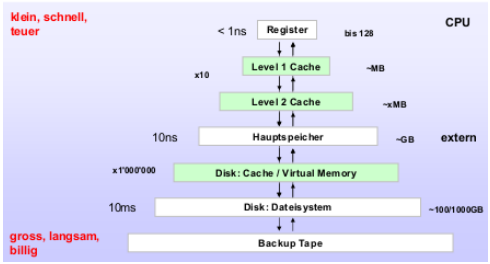
- **Batch Systems**
  - nur ein Job aus Batch im Speicher
  - Jobs werden sequentiell abgearbeitet
  - Monitor zur Steuerung, keine Interaktion mit Anwender
- **Multiprogrammed Batch Systems**
  - mehrere Jobs im Speicher
  - **Scheduler notwendig**
  - **Interrupt und Speicherverwaltung notwendig**
- **Time Sharing Systems**
  - mehrere interaktive Jobs werden "gleichzeitig" abgearbeitet
  - **Schutz des Filesystems und Arbeitsspeichers notwendig**
  - **Mutex (gegenseitiger Ausschluss) notwendig**
    - Zugriff auf Drucker, etc.

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.

Speicherhierarchie



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

Cache: **wie** funktioniert es ?

- **Hitrate (hit ratio) h**
  - h: Wahrscheinlichkeit, dass der Speicherzugriff die Daten im Cache findet
  - $1\text{-}h$: Missrate (miss ratio) m
- **Mittlere Zugriffszeit T_a auf den Hauptspeicher**
  - gegeben sind
    - Zugriffszeit auf den Hauptspeicher T_M
    - hit ratio h
    - einstufiger Cache

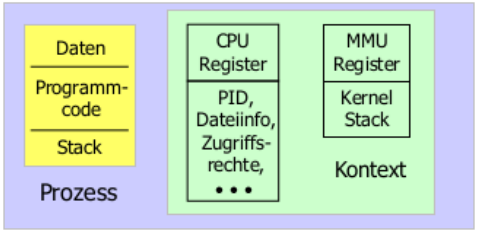
$$T_a = h \cdot T_C + (1\text{-}h) \cdot (T_C + T_M) = T_C + (1\text{-}h) \cdot T_M$$

Beispiel

- $T_C = 2\text{ns}$, $T_M = 40\text{ns}$, $h = 90\%$

$$T_a = h \cdot T_C + (1 - h) \cdot (T_C + T_M) = T_C + (1 - h) \cdot T_M$$

Prozesse



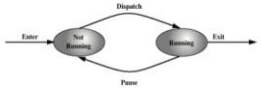
Was ist ein Prozess ?

Zweite Sichtweise:

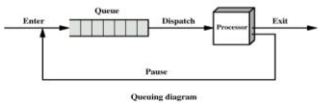
- **Unit of Resource Ownership**
  - eine Einheit, die Ressourcen besitzt
  - ein virtueller Adressraum, in dem das Prozess Image steht
  - Kontrolle über Ressourcen (Files, I/O Geräte, etc.) hat
- **Unit of Scheduling**
  - eine Einheit, die schedulierbar ist
  - CPU-Scheduler weist der CPU einen Prozess zu (dispatch)
  - zum Prozess gehören der Execution State (PC, SP, Register) und eine Ausführungspriorität

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.

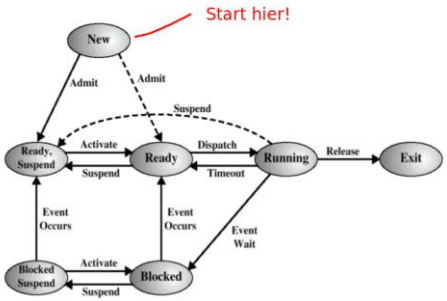
- **Modelle für Beschreibung des Prozessverhaltens**
  - **Zustandsdiagramm**



  - **Warteschlangen-Diagramm (Queuing-Diagram)**

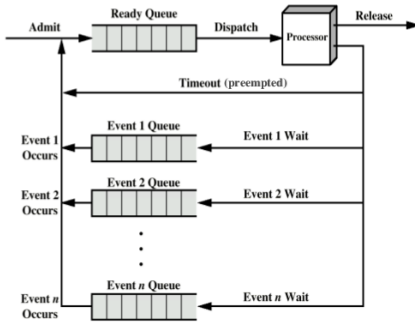


Das 7-Zustands-Prozessmodell



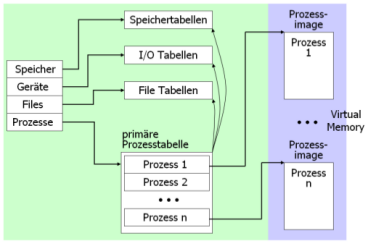
New: erzeugt, aber noch nicht gestartet. Ready: Kann ausgeführt werden. Running: ist aktiv. Blocked: Wartet auf Ereignis, z. B. IO. Suspend Blocked: Wurde ausgelagert, wartet auf Ereignis. Suspend Ready: ist ausgelagert, aber bereit für Aktivierung.

Prozesse warten in Queues auf Events



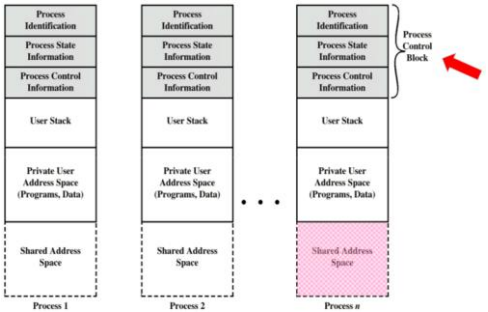
Prozess- und Ressourcenmangement

- **Das BS unterhält verschiedenste Tabellen für Verwaltung von Prozessen und Ressourcen**



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:

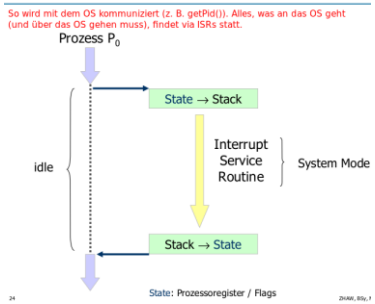


Prozesserzeugung

- **Eindeutigen Prozessidentifikator erzeugen**
- **Speicher für Prozess Image allozieren**
- **Prozesskontrollblock initialisieren**
  - Defaultwerte setzen, z.B.
    - Zustand: NEW, I/O Geräte, offene Files
    - etc.
- **Verkettungen für Queues aufsetzen**
  - z.B. Prozess in die Liste mit neuen Prozessen einfügen
- **Weitere Datenstrukturen initialisieren**
  - z.B. Accounting Informationen
- **Prozesse meist hierarchisch organisiert**
  - Eltern- und Kindprozesse

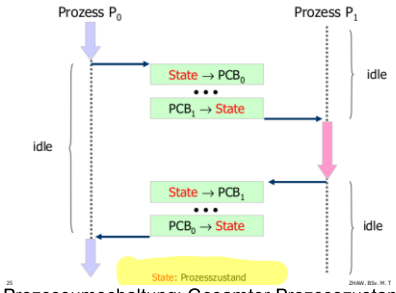
Zwei Modi: UserMode (weniger priviligiert für Anwenderprogramme), SystemMode (priviligierter für OS-Funktionen).

Mode Switch



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.

Context Switch



Prozessumschaltung: Gesamter Prozesszustand muss gespeichert werden.
Dies kann durch Programm (IO blocking, etc), OS oder äussere Interrupts passieren.

Prozesserzeugung

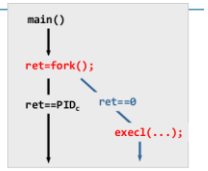
Prozesserzeugung unter Unix/Linux

- **Prozesserzeugung**
  - mit System Call `fork()` wird einen Kindprozess erzeugt
  - mit `exec()` wird anschliessend ein neues Programm gestartet
  - `fork()` erstellt "Kopie" des Elternprozesses
- **Prozesserzeugung bei Systemstart**
  - Prozess 0 (Prozess `init`)
    - zur Bootzeit erzeugt
    - spaltet Prozess 1 ab
    - wird selbst zum Swapper
  - Prozess 1
    - startet Deamonprozesse
    - erzeugt neuen Prozess, wenn sich ein Benutzer anmeldet
    - Vater aller Prozesse, sorgt auch für "Waisenkinder"

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.

Prozesserzeugung unter Unix/Linux

Beispiel: Prozesserzeugung unter Unix / Linux




```
#include <sys/types.h>
#include <stdio.h>

int main(void) {
   pid_t ret;
   ret = fork();
   if (ret == 0) {
      printf("I am the child\n");
      if (execl("./test",NULL)<0) {
         printf("could not exec");
         exit(-1);
      }
   } else
        wait(null);
}
```

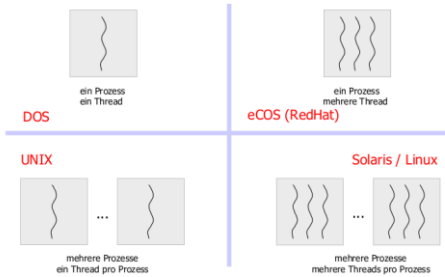
```
// make child
// child: return value = 0

// start program "test"
// should not come here

// parent: return value = PIDc
// wait for child to terminate
```

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

Single- vs. Multithreading



Thread ist "billig"

- **kann schnell erzeugt und beendet werden**
  - braucht nur Stack und Speicher für die Register
- **Threadumschaltung ist schnell**
  - nur PC, SP und Register austauschen
- **Threads benötigen**
  - wenig Ressourcen
  - keinen neuen Adressraum, keinen eigenen Datenbereich oder Programmcode
  - keine zusätzlichen Betriebssystemressourcen
- **Aber kein "Schutz" zwischen Threads**

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

Threads: Schutz und Synchronisation

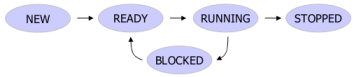
- **Threads nutzen gemeinsam Ressourcen**
  - Inter-Thread Kommunikation ohne Kernel-Hilfe möglich
  - **kein Schutz** der Daten gegen unbeabsichtigten Zugriff
  - Synchronisation notwendig
- **Synchronisation**
  - Aktivitäten verschiedener Threads müssen koordiniert werden, um **Datenkonsistenz** zu garantieren
  - d.h. vermeiden von Race-Conditions

PThreads: ein Beispiel

```
#include <pthread.h>

int aVar = 16;                                  // global aVar

void *PrintMessageFunction(void* ptr) {         // print specified message
        printf("%s: %d\n", (char*)ptr, ++aVar);
}

int main (void) {
   pthread_t       thread1, thread2;
   char            *message1 = "Hello";
   char            *message2 = "World";

   pthread_create(&thread1, NULL, PrintMessageFunction, (void*)message1);
   pthread_create(&thread2, NULL, PrintMessageFunction, (void*)message2);

   // ...Bemerkung: 2. Param =NULL entspricht "pthread_attr_default"

   pthread_join (thread1, NULL);                // wait for termination of
   pthread_join (thread2, NULL);                // specified threads

   printf("Hauptprogramm (Vater-Thread) beendet \n");
   exit(0);
}
```

Threadverhalten

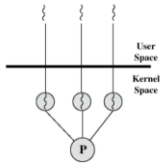
- **Threads: 3 resp. 5 Zustände**
  - Running, Ready und Blocked (New und Stopped)



  - kein Zustand Suspend: alle Threads im gleichen Adressraum
  - Swapping: Prozesses mit allen Threads wird suspendiert
  - wenn Prozesses terminiert: alle Threads terminieren
- **Was geschieht wenn ein Thread blockiert ?**
  - user level threads → der ganze Prozess blockiert
  - kernel level threads → nur der Thread blockiert

Kernel Level Threads (KLT)

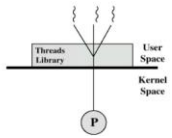
- **Thread Management durch Kernel**
  - keine Bibliothek, sondern API zu Threadfunktionen des Kernels
  - Kernel verwaltet Kontextinformation von Prozessen und Threads
  - Threadumschaltung erfordert Intervention des Kernels
  - Schedulierbare Einheit: Thread
  - Threads können auf mehrere Prozessoren verteilt werden
  - Beispiele
    - Solaris, Win/NT, OS/2, Linux



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

User Level Threads (ULT)

- **Der Kernel weiss nicht, dass es Threads gibt**
  - Anwendung verantwortlich für das Threadmanagement (Bibliotheksfunktionen)
  - Threadumschaltung benötigt keine Kernel Mode Privilegien
  - Threadscheduling applikationsspezifisch wählbar
  - Thread Bibliothek
    - Speichern und Rückspeichern des Threadkontextes
    - Threads erzeugen und beenden
    - Scheduling



Prozess-Scheduling

Scheduling: Kategorien

- **Scheduling abhängig von Anwendungsumgebung**
- **Drei wichtige Umgebungen**
  - **batch**
    - keine Anwender, die am Terminal auf Antwort warten
    - Jobs können am Stück verarbeitet werden
  - **interaktiv**
    - Anwender warten am Terminal auf Antwort
    - Umschaltung zwischen Prozessen notwendig
  - **real-time**
    - Resultat muss zur richtigen Zeit verfügbar sein
    - oft periodische Jobs

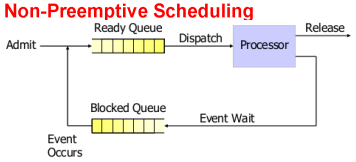
Batch: non-preemitve, also ohne Unterbrechung druch Scheduler, nur, wenn Prozess blockiert

Definitionen:
Tb = Rechenzeit (Bedienzeit)
Tw = Wartezeit, bis das erste Mal ausgeführt wird
Ta = Turnaroundzeit (Zeit zwischen Prozessaufgabe und Prozessterminierung)
Ts = Starzeit

Scheduling Algorithmen: Ziele

- **Für alle Systeme gilt**
  - Fairness, Einhalten von Policies, System optimal nutzen
- **Batch Systeme**
  - Durchsatz → Anzahl Jobs pro Zeit
  - Turnaround Zeit → Zeit zwischen Aufgabe und Terminierung
  - CPU-Nutzung → möglichst 100% Auslastung
- **Interaktive Systeme**
  - Antwortzeit → schnelle Reaktion
  - Erwartung → Erfüllen der Anwendererwartungen
- **Real-time Systeme**
  - Deadlines → kein Datenverlust
  - Vorhersagbarkeit → kein Qualitätsverlust z.B. bei Multimedia

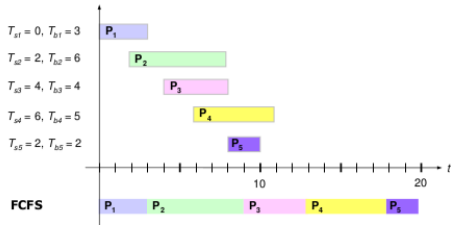
Non-Preemptive Scheduling



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

First Come First Served (FCFS/FIFO)

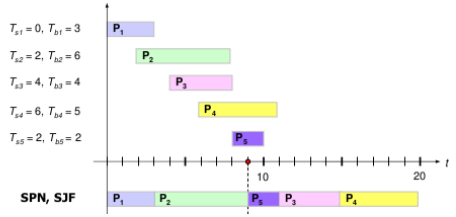
- **Auswahlfunktion FCFS**
  - ältester Prozess in der Queue → First Come First Served



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

Shortest Job First (SJF, SPN)

- **Auswahlfunktion**
  - Prozess mit kürzester, geschätzter Rechenzeit



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

Schätzung der Bedienzeit

- **Laufender Mittelwert als Schätzung für Bedienzeit**

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i$$

T_i: effektive Ausführungszeit des i-ten CPU Bursts
S_i: Schätzwert für Ausführungszeit des i-ten CPU Bursts
S_1: Schätzwert, wenn Prozess zum ersten Mal läuft (oft 0)

- **Umformuliert (keine Neuberechnung notwendig)**

$$S_{n+1} = \frac{1}{n}\cdot T_n + \frac{n-1}{n}\cdot S_n$$

  - vergangene Ausführungszeiten haben gleiches Gewicht (1/n)

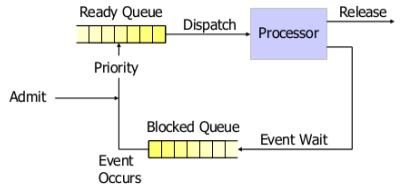
Schätzung der CPU Burst Time

- **Anforderung**
  - nahe Vergangenheit stärker gewichten
  - allgemeinere Formulierung

$$S_{n+1} = \alpha \cdot T_n + (1-\alpha)\cdot S_n \qquad 0 < \alpha < 1$$

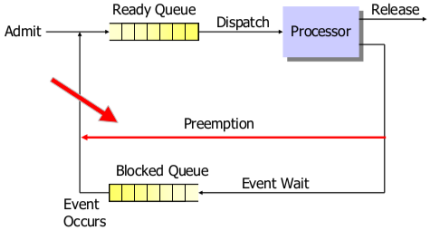
- **Für $\alpha < 1$ gilt**
  - die nahe Vergangenheit wird stärker gewichtet
  - der Einfluss vergangener Messwerte schwindet exponentiell
  - geschätzter und exakter Verlauf stimmt besser überein
  - der Startwert S_1 wird oft auf 0 gesetzt → hohe Startpriorität

Priority Scheduling



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

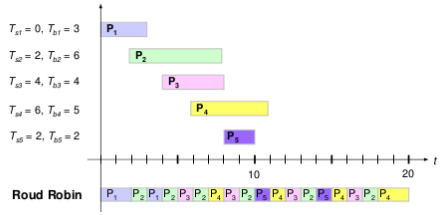
Preemptive Scheduling



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

Round Robin (RR)

- **Auswahlfunktion**
  - wie bei FCFS aber preemptive: time-slice resp. time-quantum q



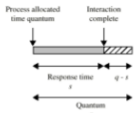
Jobs können nicht verhungern, Response für kurze Prozesse gut. „q" = time quantuum = time slice. Die Länge

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).

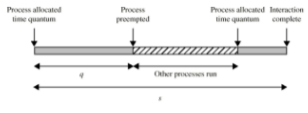
Round Robin: Wahl von q

- **Wie gross muss Time Slice q gewählt werden ?**
  - einiges grösser als Zeit für Clock Interrupt und Dispatching
  - etwas grösser als eine typische Interaktion (~~100ms~~) 10 ms!
  - aber nicht viel grösser, sonst werden I/O-bound Prozesse bestraft

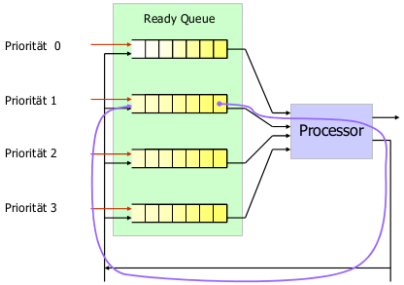
q > Interaktion



q < Interaktion

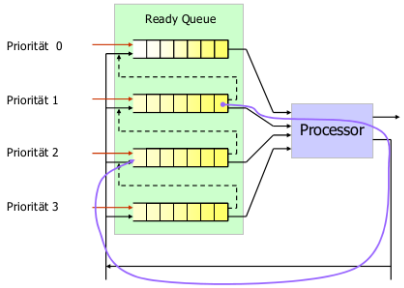


Multilevel Scheduling



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.

Multilevel Feedback Scheduling



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

Unix Scheduling traditionell

- **Prioritäten**
  - basieren auf Prozesstyp und Geschichte
  - tiefer Wert = hohe Priorität

$$CPU_j[i] = \frac{CPU_j[i-1]}{2}$$
$$P_j[i] = base_j + \frac{CPU_j[i]}{2} + nice_j$$

im Sinne von 'nett sein zu den anderen Prozessen', also die eigene Prio erniedrigen, wenn man den nice-Wert erhöht

$P_j[i]$	Priorität von Prozess j zu Beginn des Intervalls i
$base_j$	Basispriorität von Prozess j
$CPU_j[i]$	exponentiell gewichtete mittlere Prozessornutzung durch Prozess j bis zum Beginn von Intervall i
$nice_j$	benutzergesteuerter Anpassungsfaktor

Linux-Scheduling: Kernel >= 2.6.23

- **Drei Scheduler Klassen (absteigende Prio)**
  - Realtime-Klassen: `SCHED_FIFO, SCHED_RR`
  - CFS: `SCHED_OTHER`
  - Idle: ≃ Aidle `SCHED_IDLE`
  - Prozesse laufen in den Idle Zeiten der höheren Prioritätsklasse
- **CFS: Completely Fair Scheduler**
  - kein Time-Slice, sondern "Anteil an Prozessorleistung"

$$fair\text{-}clock \sim \frac{wall\text{-}clock}{\text{Anzahl lauffähige Prozesse}}$$

Red-Black-Tree

  - keine Run-Queue (RQ) sondern ein ~~Red-Black~~ Tree
    - selbstbalancierender Binärbaum
  - unterstützt Group-Scheduling: Faire Gruppenzuteilung
  - **nice** Werte geometrisch: multiplikativ, Faktor = 1.25
- **CFS-Scheduling**
  - Zeitberechnungen in ns
  - jeder Prozess besitzt eine wait-runtime
    - während Wartezeit mit fair-clock erhöht
    - während Laufzeit mit wall-clock erniedrigt
      - minimale Lauftzeit festgelegt durch minmalen Time-Slice
  - Dispatching
    - Prozess mit höchster wait-runtime wird eingeplant und läuft bis ein anderer Prozess eine höhere wait-runtime hat
    - Überprüfung Rescheduling in Zeitraster
      - minimale Granularität: Time-Slice
      - minimale Granularität : default 1ms

Real-Time Systeme

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

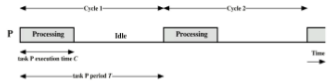
- **Real-Time Systeme**
  - Systeme, die auf **Ereignisse** in der "äusseren Welt" reagieren
  - Ereignisse finden in der "reellen Zeit" statt → real-time
  - intuitive Betrachtungsweise
    - das System muss schnell reagieren ... was heisst schnell ?
- **Real-Time Systeme verhalten sich korrekt, wenn**
  - das **logischen Resultat** einer Berechnung stimmt
  - das Resultat zum richtigen **Zeitpunkt** ausgeliefert wird (innerhalb einer Deadline)
- **Klassierung**
  - Hard Real-Time Systeme
  - Soft Real-Time Systeme

RealTime = Etwas muss zu einem Zeitpunkt _fertiggestellt_ sein

- **Drei real-time Task Klassen**
  1. kritische Tasks
     - periodische Tasks
     - asnychrone, sporadische Tasks
  2. nicht kritische Tasks, aber notwendige Tasks
  3. nicht notwendige Tasks (nice to have)
- **Entsprechende Scheduling Verfahren wählen**
  - kritische Tasks
    - Rate Monotonic, Deadline
  - nicht kritische Tasks
    - übliche Scheduling Kriterien
  - nicht notwendige Tasks
    - im Hintergrund, in verbleibender Zeit ausführen

Rate Monotonic Scheduling (RMS)

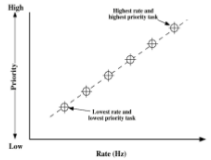
- **Preemptives Verfahren für periodische Tasks**



- **Idee**

statische Priorität eines Jobs proprotional zu **Repetitionsrate**

1973 von Liu und Layland vorgeschlagen



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).

- **Was ist daran so interessant ?**
  - **Grenze** für erfolgreiches Scheduling kann berechnet werden
- **Vorgehen**
  - Prozessorauslastung U_j durch periodischen Task j, ($U_j \le 1$) :

$$U_j = \frac{C_j}{T_j}$$

C_j Rechenzeit
T_j Periode

  - gesamte Prozessorauslastung U_{tot} bei n periodischen Tasks

$$U_{tot} = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + ... + \frac{C_n}{T_n} \le 1$$

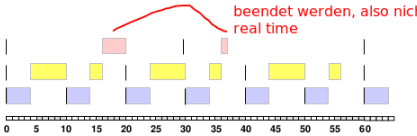
Utot muss <= 1 sein

- **Obige Formel**
  - zu optimistisch, nur bei perfektem Schedule möglich
- **Korrekter Schedule für n Tasks**
  - garantiert bei Einhaltung folgender Grenze

$$U_{tot} = \frac{C_1}{T_1} + \frac{C_2}{T_2} + \frac{C_3}{T_3} + ... + \frac{C_n}{T_n} \le n \cdot (2^{1/n} - 1)$$

- **Diskussion**
  - eher konservativ, besserer Auslastung möglich
  - Scheduling Algoritmus beliebig
  - Grenzwert für $n \to \infty$: $U_{tot} = ln\ 2 \approx 0.69$
  - konkrete Anwendungen: Auslastung bis 90% realistisch

Negativ: für grosse n nur 70% CPU Auslastung.



	C_i	T_i
	4	10
	8	20
	5	30

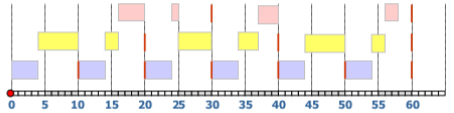
$$U_{tot} = \frac{4}{10} + \frac{8}{20} + \frac{5}{30} = 0.97$$

Die Repetitionsrate des Schedulers ist das kgV.

Deadline Scheduling

- **Idee**
  - Tasks zum richtigen Zeitpunkt starten resp. beenden
- **Mögliche Deadlines**
  - Starting-Deadline
    - Zeitpunkt zu dem ein Task gestartet werden muss
  - Completion-Deadline
    - Zeitpunkt an dem der Task beendet sein muss
- **Earliest Deadline Scheduling**
  - Priorität umgekehrt proportional zur Zeit bis Deadline
  - Scheduling aufwendiger als bei RMS
    - Prioritäten müssen dynamisch angepasst werden
  - theoretisch 100%-tige Prozessorauslastung möglich

Earliest Deadline Scheduling



	C_i	T_i
	4	10
	8	20
	5	30

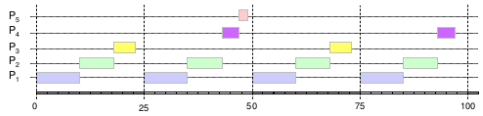
Rescheduling
- hier alle 5 Zeiteinheiten
- wenn sich Task suspendiert

Rescheduling alle 5 Zeiteinheiten.

Cyclic Executives

- **Eigenschaften**
  - statisches Scheduling
  - non-preemtive
  - Schedule für periodischen Hauptzyklus (100ms)
    - mehrere Nebenzyklen (4, 25ms)
  - Realisierung: Aufruf von Prozeduren

Prozess	C_i	T_i
P_1	10	25
P_2	8	25
P_3	5	50
P_4	4	50
P_5	2	100



Manchmal schwierig, einen passenden Schedule zu finden.