Vorlesung Betriebssysteme I

Thema 7: Zuteilung des Prozessors

Robert Baumgartl

11. Januar 2016

Prozessorzuteilung (*Scheduling*)

- = Beantwortung der Frage:

  „Welche Aktivität soll zu einem bestimmten Zeitpunkt abgearbeitet werden (und für wie lange)?“

- Komponente im Betriebssystem: der *Scheduler* (Planer)
- Verfahren zur Ermittlung einer Abarbeitungsplans (*Schedule*)

Typische Zielgrößen

Je nach *betrachteter Systemklasse* (z. B. Batchsysteme, Interaktive Systeme, Echtzeitsysteme) existieren verschiedene zu optimierende Parameter:

- mittlere Reaktionszeit aller Prozesse
- mittlere Verweilzeit aller Prozesse (*turnaround time*)
- maximale CPU-Ausnutzung
- maximale Anzahl gleichzeitiger Datenströme
- Garantie einer maximalen Reaktionszeit
- Fairness: n Prozesse $\rightarrow$ jeder $1/n$ der Prozessorzeit
- Quality-of-Service (QoS): „Jeder bekommt so viel, wie er bezahlt hat.“
- Ausschluss des Verhungerns einzelner Prozesse

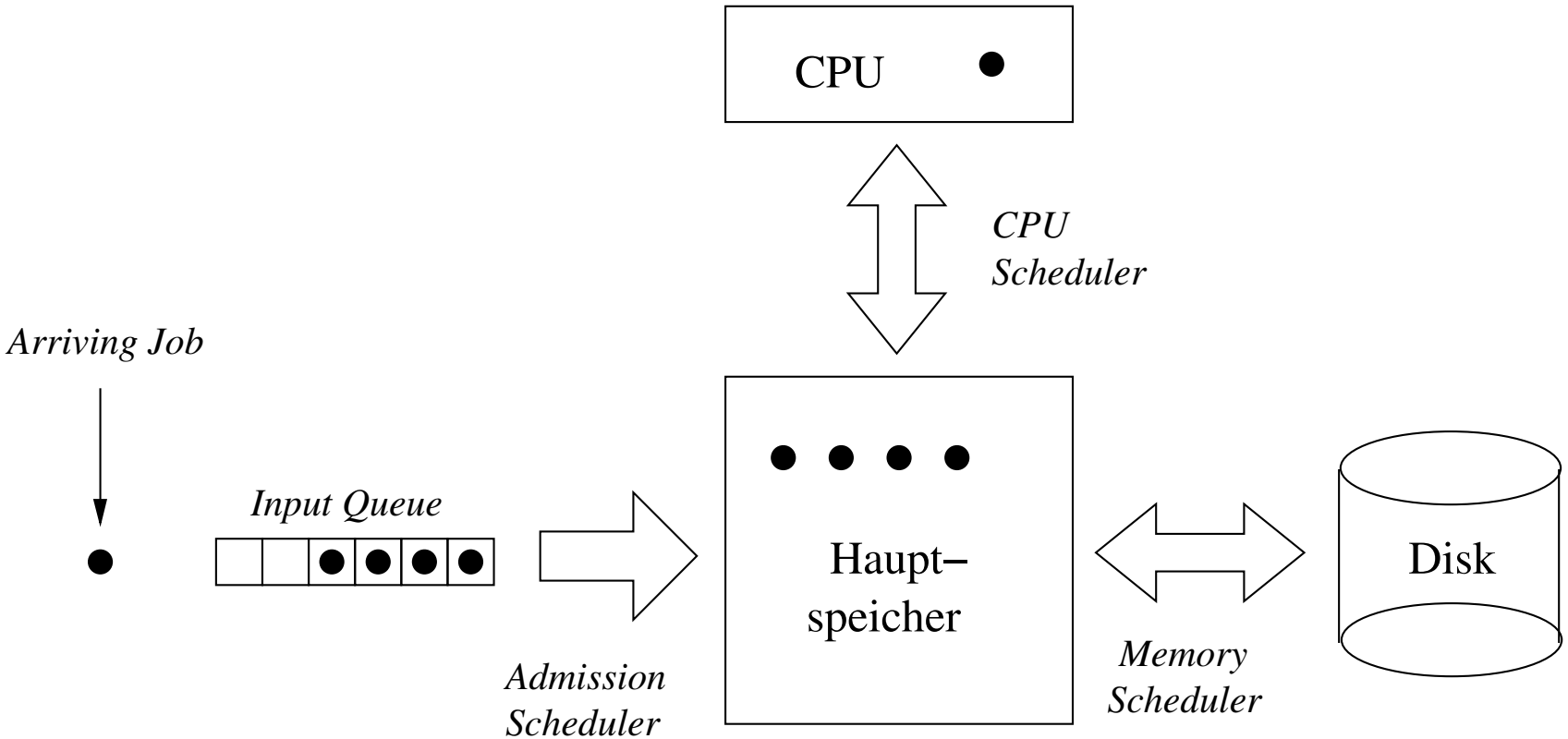
Planung anderer Ressourcen

Außer dem Prozessor können (müssen aber nicht) die folgenden Ressourcen geplant werden:

- Hauptspeicher,
- Aufträge an den Massenspeicher,
- Kommunikationsbandbreite,
- Interrupts
- ...

Beispiel: Linux besitzt einen sog. I/O-Scheduler, der Festplattenaufträge plant (d. h. , ggf. umsortiert).

Beispiel: Schedulingebenen in einem Batch-System



Quelle: Andrew Tanenbaum, *Modern Operating Systems*. 2000, S. 141

Off-Line- vs. On-Line-Verfahren

Off-Line

- komplette Ermittlung des Abarbeitungsplans *vor* Inbetriebnahme des Systems
- Zur Laufzeit des Systems wird der vorbereitete Plan abgearbeitet (keine Entscheidungen mehr notwendig).
- inflexibel
- sehr hohe Auslastung möglich
- Startzeitpunkte, Ausführungszeiten, Abhängigkeiten aller Aktivitäten müssen *a priori* bekannt sein.
- z. B. bei autonomen oder Echtzeit-Systemen
- situationsspezifische Pläne möglich, System unterscheidet mehrere *Modi*

Off-Line- vs. On-Line-Verfahren

On-Line

- Auswahl des jeweils nächsten abzuarbeitenden Prozesses erfolgt zur Laufzeit des Systems.
- Flexibel: System kann auf Änderungen verschiedener Parameter, Umwelteinflüsse, Nutzeranforderungen reagieren
- keine Zeit für langwierige Auswahlverfahren $\rightarrow$ Kompromiss zwischen Optimalität des ausgesuchten Prozesses und Dauer für die Entscheidung notwendig.

Typische interaktive Betriebssysteme wie Windows oder Linux planen on-line.

Beispiel für Off-Line-Scheduling

Ein (nicht näher spezifiziertes) Rechensystem bestehe aus 3 Prozessen, die wiederum aus den folgenden unabhängigen *Teilprozessen* bestehen (benötigte Rechenzeit in Klammern):

$P_1 : \{\ p_{11}(3),\ p_{12}(2),\ p_{13}(2),\ p_{14}(5)\ \}$
$P_2 : \{\ p_{21}(5),\ p_{22}(7)\ \}$
$P_3 : \{\ p_{31}(5),\ p_{32}(2)\ \}$

Außerdem bestehen die folgenden expliziten zeitlichen Abhängigkeiten zwischen den Teilprozessen:

p_{21} *vor* p_{12}, p_{12} *vor* p_{22}, p_{13} *vor* p_{31},
p_{14} *vor* p_{32}, p_{22} *vor* p_{32}.

Darüberhinaus müssen die Teilprozesse ein- und desselben Prozesses hintereinander liegen.

Präzedenzgraph

Die zeitlichen Abhängigkeiten veranschaulicht man am besten in einem *Präzedenzgraphen*:

- einen *Knoten* für jeden Teilprozess
- eine *Kante* zwischen zwei Knoten genau dann, wenn der erste Knoten beendet sein muss, bevor der zweite gestartet werden darf

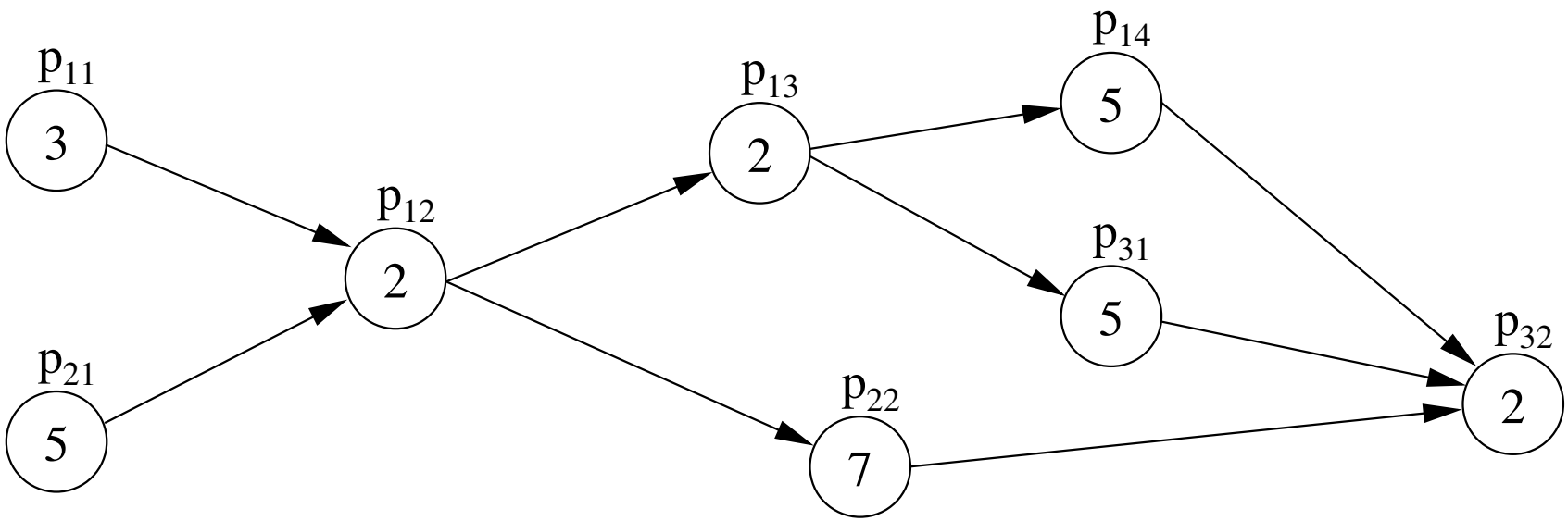


Abbildung: Präzedenzgraph des Beispielprozesssystems

Ableitung eines Schedules (off-line)

Verfahren:

1. Bildung der Bereit-Menge $\mathcal{B}$ (enthält alle Prozesse, die abgearbeitet werden können)
2. Auswahl von n Prozessen aus $\mathcal{B}$ (n ist die Prozessoranzahl, im einfachsten Falle also 1) nach vorgegebenem Kriterium (z. B. „den kürzesten Prozess zuerst“)
3. Planung der ausgewählten Prozesse für bestimmte Zeitspanne (im einfachsten Falle: für *eine* Zeiteinheit)
4. Falls noch nicht alle Prozesse geplant sind $\rightarrow$ Goto 1
5. Stop

Anwendung auf Beispieltaskmenge

- $n = 2$ (z. B. Dualcore-Prozessor)
- Auswahl des jeweils kürzesten Prozesses (*Shortest Job Next*)
- Abarbeitung ohne Unterbrechung, wenn einmal gestartet (*Run-to-Completion*)

Zeit	$\mathcal{B}$	Auswahl
0	p_{11}, p_{21}	p_{11}, p_{21}
3	(p_{21})	(p_{21})
5	p_{12}	p_{12}
7	p_{13}, p_{22}	p_{13}, p_{22}
9	p_{14}, p_{31}	p_{14}
14	p_{31}	p_{31}
19	p_{32}	p_{31}

Tabelle: Schedulingzeitpunkte für Beispiel

Resultierender Schedule

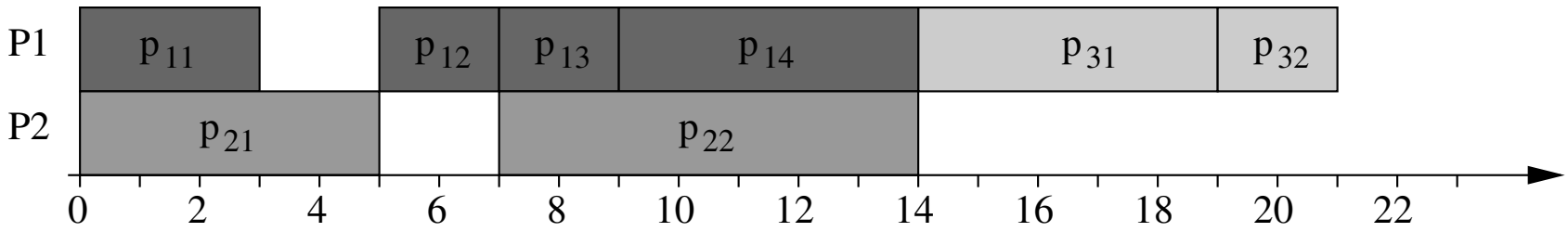


Abbildung: Off-Line Schedule für Beispieltaskmenge und ohne Unterbrechungen

- Resultat: Gantt-Diagramm (benannt nach dem Unternehmensberater (!) Henry L. Gantt)
- Komplettierung des letzten Teilprozesses zu $t = 21$
- Prozessoren nicht voll ausgelastet (*idle time*); Ursache: Präzedenzen zwischen Teilprozessen

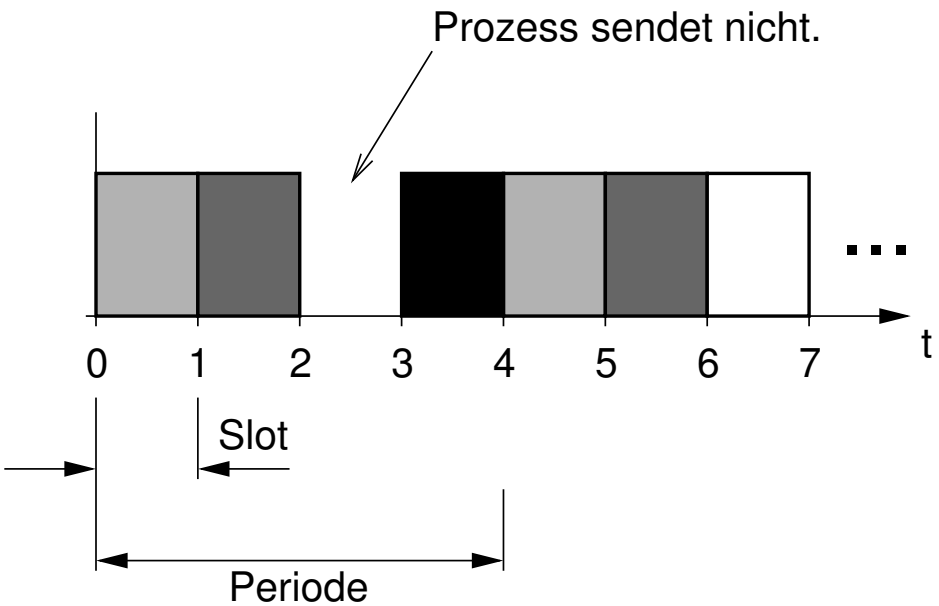
Zeitgesteuertes Scheduling

- alle Abläufe im System erfolgen in festem zeitlichen Rahmen, periodisch
- keine Interrupts $\rightarrow$ keine unvorhergesehenen Aktivitäten
- Kommunikation mit externe Komponenten: Abfragen (*Polling*)
- typisch für autonome und Echtzeitsysteme
- Nutzung von off-line ermittelten Schedules, zwischen denen umgeschaltet werden kann (Moduswechsel)
- Beispiel: Medienzugriffsverfahren *T*ime *D*ivision *M*ultiple *A*ccess (TDMA)

Time Division Multiple Access

Prinzip:

- Übertragungszeit wird in (unendlich viele) Perioden fester Länge aufgeteilt
- innerhalb jeder Periode erhält *jeder* (potentielle) Kommunikationsteilnehmer $1/n$ der Periodenlänge, einen sog. *Slot*
- in seinem Slot kann jeder senden oder nicht
- $\rightarrow$ keine Kollisionen möglich



Ereignisgesteuertes Scheduling

Prinzip:

- System reagiert auf Einflüsse von außen (Interrupts)
- Aktivitäten werden als Reaktion auf Interrupts bereit
- prinzipiell keine Garantie von Ausführungszeiten möglich, da Auftrittszeitpunkte von Interrupts nicht vorhersehbar
- typisch für interaktive Systeme
- Beispiel: Grafische Benutzeroberflächen (Ereignisse: Mausbewegung, Klick, Tastendruck, aber auch Interrupt durch die Netzwerkkarte)

Was passiert denn eigentlich beim Interrupt?

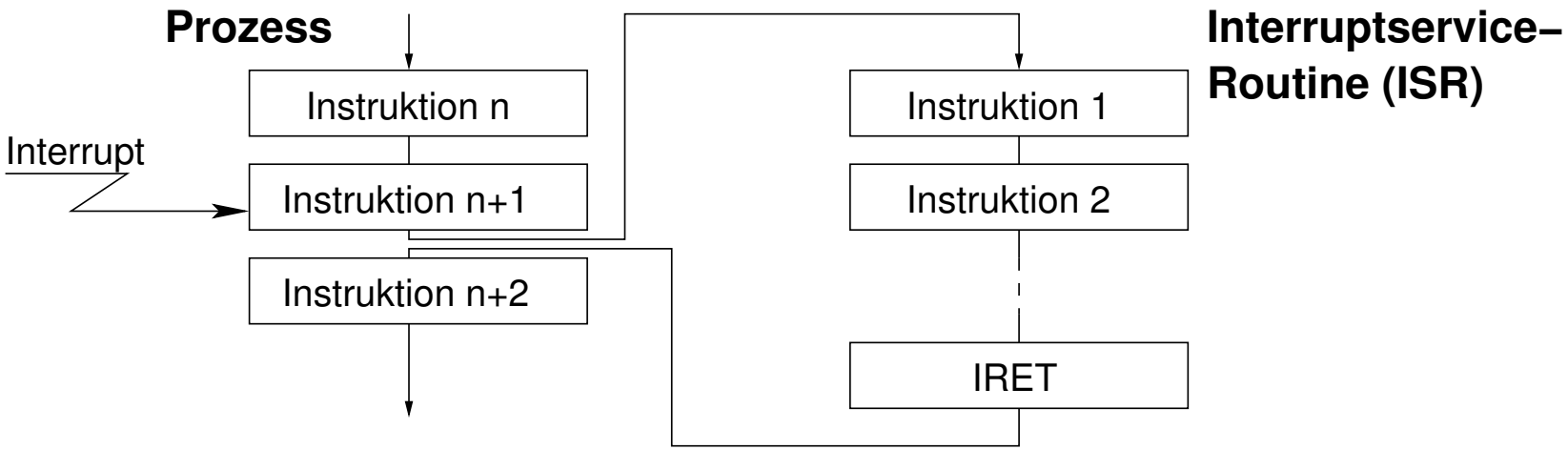
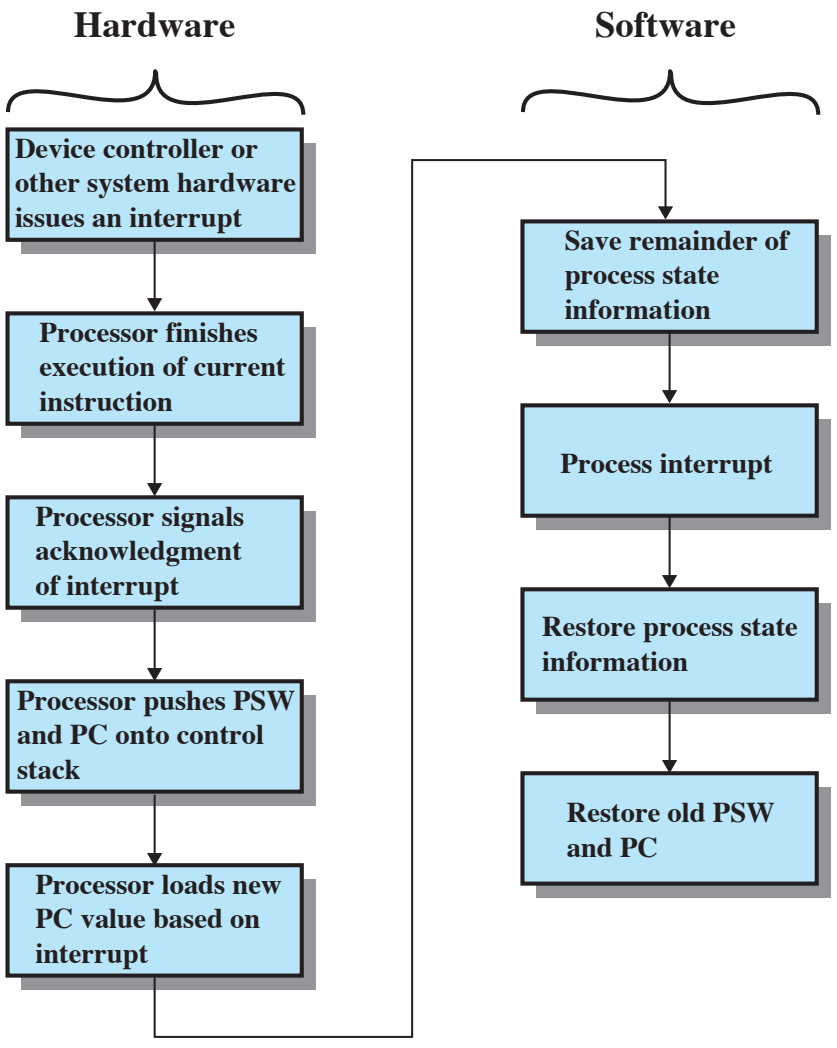


Abbildung: Ablauf einer Interruptbehandlung (vereinfacht)

- Interrupts sind *asynchron* zum Programmablauf
- Quellen: Geräte (I/O), Programm, Betriebssystem

Interrupt: Ablauf in der CPU



(William Stallings: *Operating Systems*. 6th ed., Pearson, 2009)

Schedulingzeitpunkt beim On-Line Scheduling

Unterbrechung eines aktiven Prozesses:

- durch das BS, (prinzipiell) jederzeit (**präemptives Multitasking**):
  - wenn ein Prozess blockiert (z. B. an Ressource),
  - wenn ein Prozess bereit wird (z. B. als Reaktion auf einen Interrupt oder durch eine Ressourcenfreigabe),
  - wenn ein Prozess endet.
- durch das BS, jedoch nur an bestimmten Stellen, sogenannten *Preemption Points*
- freiwillig, an bestimmten Stellen, z. B. Systemruf (**kooperatives Multitasking**)
- nach Komplettierung einer Aktivität (*run-to-completion*)

Prioritäten und Priorisierung

- (gewollt) unfair, Prozesse besitzen unterschiedliche Wichtigkeit
- einfachste Möglichkeit: **F**ixed **E**xternal **P**riorities (FEP))
- d. h. , jeder Prozess erhält vor der Laufzeit des Systems einen Parameter *fest* zugeordnet, der seine Wichtigkeit ausdrückt, seine Priorität
- zur Laufzeit wird stets der höchstpriorisierte unter allen bereiten Prozessen ausgewählt

Prioritäten zum zweiten

Implizite Prioritäten: ein bestimmter Parameter jedes Prozesses wird „zweckentfremdet“ zur Bestimmung der Priorität herangezogen.
Beispiele:

- Länge des Jobs
- verbleibende Abarbeitungszeit
- Zeit seit letzter Aktivierung
- Deadline (Zeit bis zur unbedingten Komplettierung)

Statische und dynamische Prioritäten

Statisch: Priorität eines Prozesses ist konstant.

- einfacher Scheduler
- gut zu analysieren
- nicht besonders flexibel (was ist, wenn sich die Wichtigkeit eines Prozesses ändert?)

Dynamisch: Priorität eines Prozesses ändert sich mit der Zeit.

- periodische Neuberechnung (Aufwand!)
- erlaubt situationsspezifische Anpassungen
- schwieriger zu analysieren

Uniprozessor- vs. Multiprozessor-Scheduling

- zusätzlich nötige Entscheidung, *wo* Prozess abgearbeitet wird
- Ziel: *Load Balancing*
  - zu starr: möglicherweise schlechte Ausnutzung der Prozessoren
  - zu flexibel: häufiger Wechsel des Prozessors (*Thrashing*) $\rightarrow$ sehr hoher Overhead
- ideal: auf einem unbeschäftigten Prozessor fortsetzen
- günstig: Prozessor, auf dem der Prozess unterbrochen wurde (Cache, TLB)
- Parameter *Affinität* des Prozesses zu einem bestimmten Prozessor

Round Robin – Zeitscheibenverfahren

Idee: Jeder Prozess erhält den Prozessor für eine bestimmte Zeitspanne (*Quantum* t_q), dann ist der nächste dran.

- Grundgedanke: Fairness
- t_q klein $\rightarrow$ Umschaltaufwand im Verhältnis zur Nutzarbeit groß, kleine Reaktionszeit pro Prozess
- t_q groß $\rightarrow$ relativer Umschaltaufwand klein, Reaktionszeit pro Prozess groß
- wichtiger Parameter: Umschaltzeit t_{cs} (*Context Switch Time*)
- Reaktionszeit eines Prozesses abhängig von t_{cs}, Anzahl Prozesse, t_q
- häufig Kombination mit Prioritäten (RR innerhalb einer Prioritätsklasse)

Veranschaulichung Round-Robin

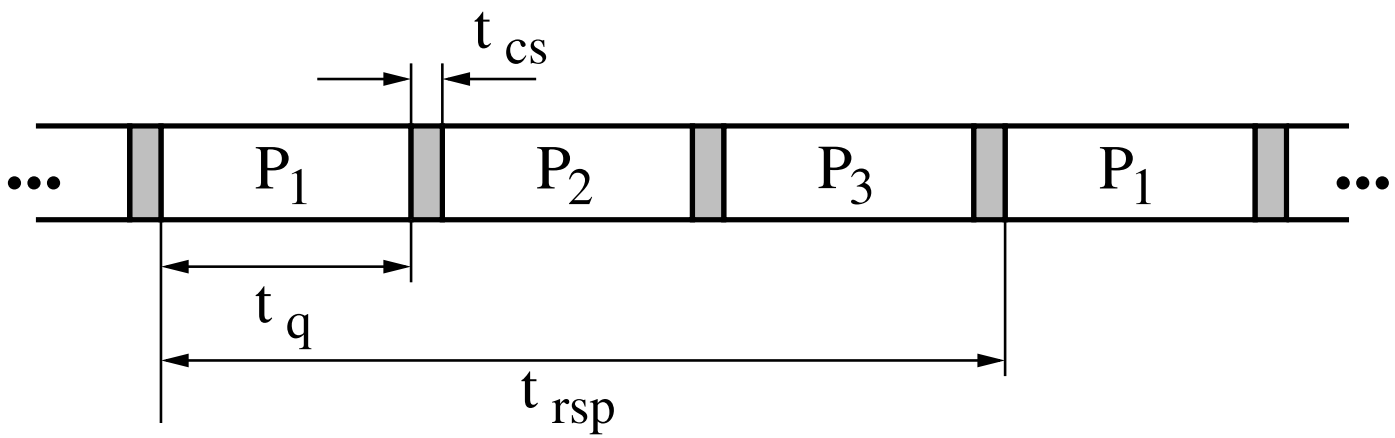


Abbildung: Parameter beim Zeitscheibenverfahren

Beispiel: Einfache Batch-Verfahren

First In First Out (FIFO, FCFS)

- Prozesse werden in der Reihenfolge ihres Eintreffens (vollständig abgearbeitet)
- fair
- leicht zu analysieren ($\rightarrow$ Warteschlangentheorie)

Shortest Job Next (SJN)

- Idee: schnell ein paar kurze Jobs fertigstellen, bevor alle auf einen langen Job warten müssen.
- Prozess mit der kürzesten Dauer wird ausgewählt, run-to-completion
- Ausführungszeit muss bekannt sein
- minimiert mittlere Verweilzeit ($\bar{t}_v$) und mittlere Wartezeit ($\bar{t}_w$)
- ungerecht, Verhungern möglich

Beispiel zu SJN

4 Beispieltasks:

Job	*Dauer*
J_1	6
J_2	8
J_3	7
J_4	3

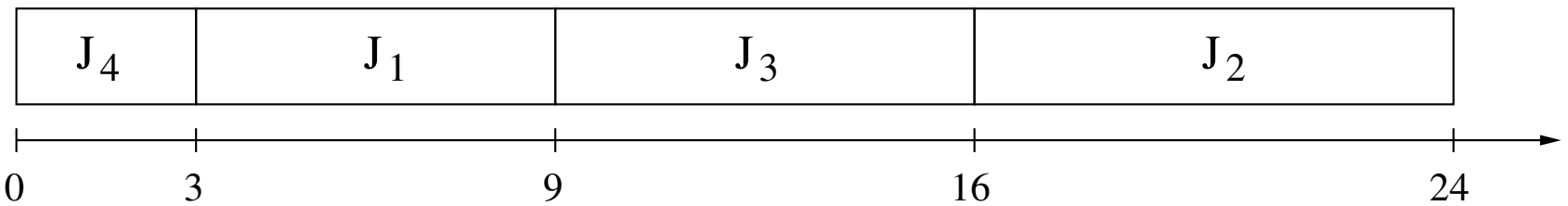


Abbildung: Resultierender SJN-Schedule

$$\bar{t}_w = \frac{0+3+9+16}{4} = 7 \qquad \bar{t}_v = \frac{3+9+16+24}{4} = 13$$

Unix

- zeitscheibengesteuert (Quantum)
- versucht, 2 Klassen von Prozessen zu unterscheiden und getrennt zu behandeln:
  1. interaktive („I/O-bound“)
  2. (vorwiegend) rechnende („compute-bound“)
- Rechnende Prozesse nutzen ihre Zeitscheibe voll aus
- Interaktive Prozesse nutzen ihre Zeitscheibe häufig nicht aus (warten auf Interaktion durch Nutzer oder Gerät; d. h. blockieren häufig)
- wenn Zeitscheibe nicht ausgenutzt, wird Priorität (leicht) erhöht $\rightarrow$ Unix bevorzugt interaktive Prozesse:
  - interaktive Prozesse reagieren besser
  - rechnende Prozesse werden etwas benachteiligt

Linux?

- dynamische Prioritäten mit Zeitscheiben
- genaues Verfahren ziemlich kompliziert
- jeder interaktive Prozess besitzt einen sog. *nice*-Value
- dieser beschreibt, wie der betreffende Prozess im Vergleich zu anderen interaktiven Prozessen priorisiert wird
  - -20: höchste Priorität
  - 0: default
  - 19: niedrigste Priorität
- Kommandos `nice` und `renice` (für bereits laufende Prozesse) ändern diesen Wert

```
~> renice 20 -p 24195
24195: old priority 0, new priority 19
```

Achtung: normale Nutzer dürfen Priorität nur monoton ändern.

Anzeige Priorität und nice-Wert mittels `top`

```
top - 14:31:23 up  3:28,  6 users,  load average: 1.68, 0.67, 0.30
Tasks:  91 total,   4 running,  87 sleeping,   0 stopped,   0 zombie
Cpu(s): 42.1%us, 44.7%sy, 13.2%ni,  0.0%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:    256396k total,   239564k used,    16832k free,    31364k buffers
Swap:  1048568k total,       84k used,  1048484k free,   100936k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 3735 root      16   0  161m  12m 4400 R 50.9  4.9   3:08.68 Xorg
24194 robge     15   0  7404 3216 2244 S 26.6  1.3   1:59.25 xterm
24195 robge     39  19  5672 2636 1412 R 15.6  1.0   2:08.29 bash
23793 robge     15   0  9640 5808 4272 S  6.0  2.3   0:01.57 WindowMaker
23928 robge     15   0  7408 3260 2264 R  0.3  1.3   0:01.82 xterm
    1 root      15   0  1948  648  552 S  0.0  0.3   0:01.14 init
    2 root      RT   0     0    0    0 S  0.0  0.0   0:00.00 migration/0
    3 root      34  19     0    0    0 S  0.0  0.0   0:00.00 ksoftirqd/0
    4 root      10  -5     0    0    0 S  0.0  0.0   0:00.08 events/0
    5 root      10  -5     0    0    0 S  0.0  0.0   0:00.00 khelper
    6 root      11  -5     0    0    0 S  0.0  0.0   0:00.00 kthread
    9 root      17  -5     0    0    0 S  0.0  0.0   0:00.00 kblockd/0
   10 root      19  -5     0    0    0 S  0.0  0.0   0:00.00 kacpid
  104 root      10  -5     0    0    0 S  0.0  0.0   0:00.02 kseriod
  138 root      15   0     0    0    0 S  0.0  0.0   0:00.01 pdflush
  139 root      15   0     0    0    0 S  0.0  0.0   0:00.08 pdflush
  140 root      10  -5     0    0    0 S  0.0  0.0   0:01.04 kswapd0
```

Scheduling in Windows 2000/XP/Vista

- prioritätsgesteuert, präemptiv

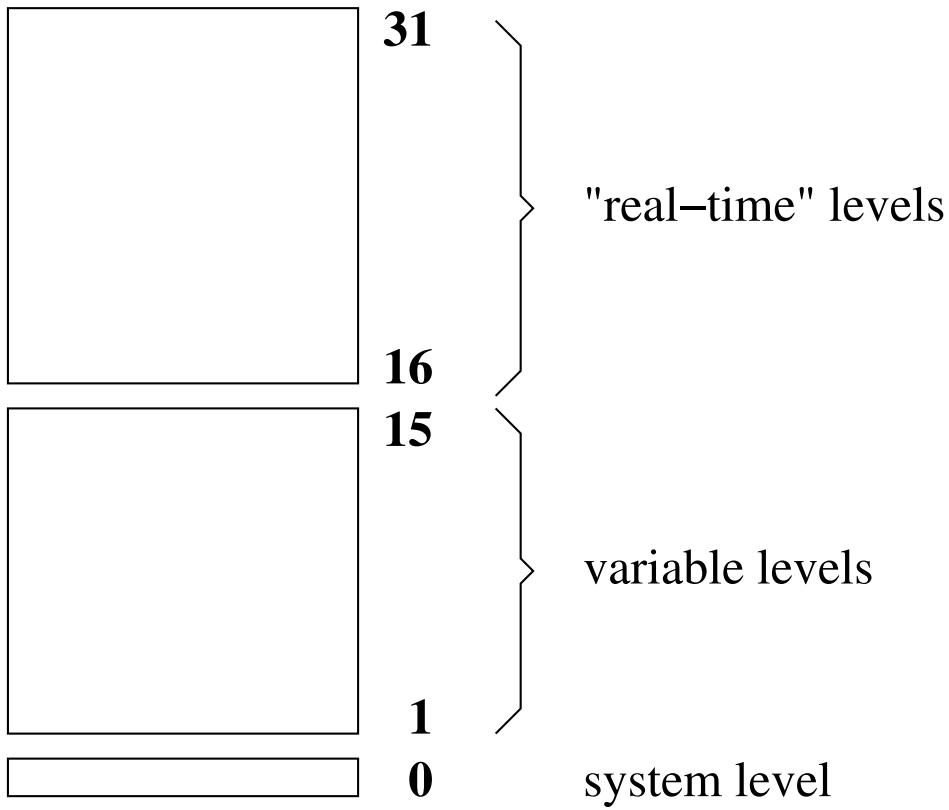
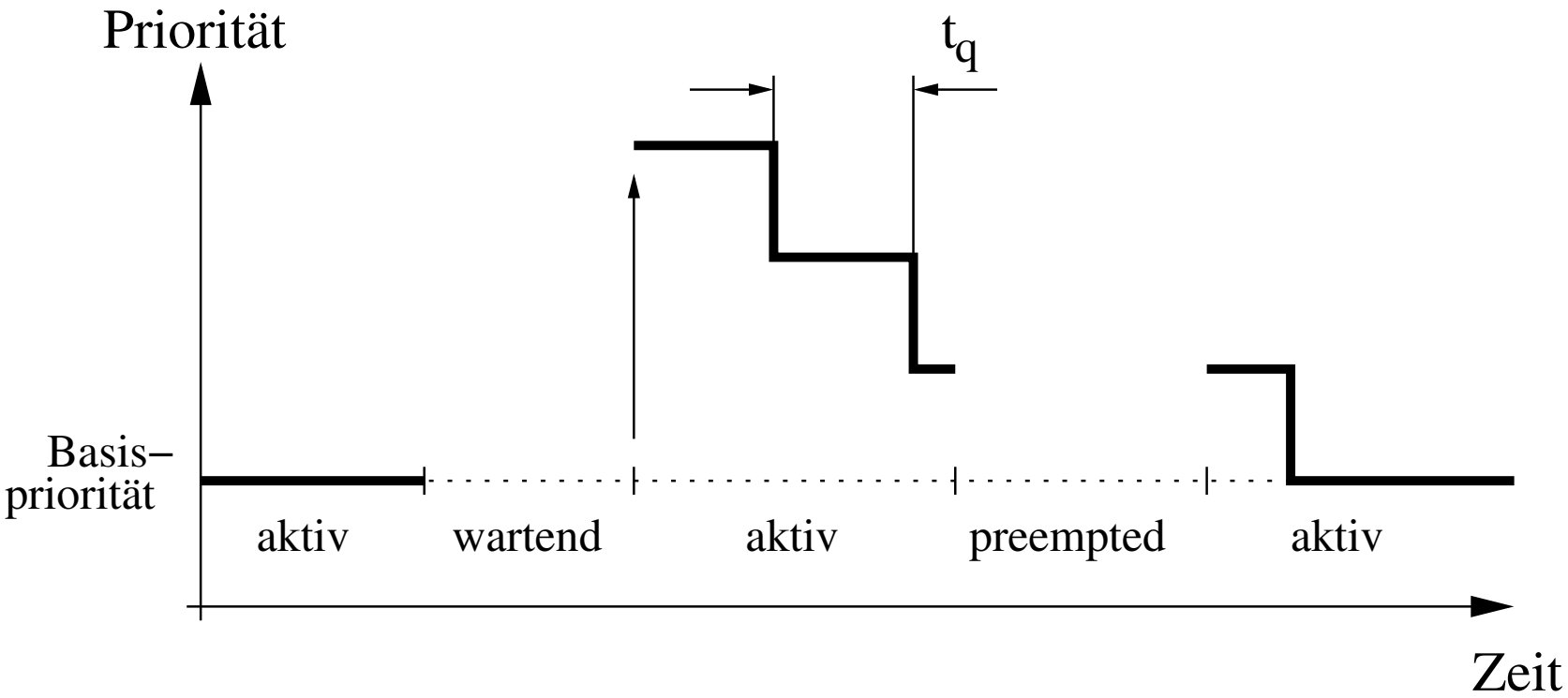


Abbildung: Prioritätsstufen im Windows 2000/XP

Scheduling in Windows 2000/XP/Vista

- Round-Robin bei Threads gleicher Priorität
- Länge des Quantums differiert für Desktop- und Server-Variante (Server: 6fach länger!)
- Quantum wird für Vordergrundthreads verdoppelt
- temporäre Prioritätsanhebung (*Priority Boost*) in den Levels 1-15 u. a. bei
  - Komplettierung einer I/O-Operation
  - Fensterthreads, die in den Vordergrund gelangen,
  - Gefahr des Verhungerns.

Prinzip des *Priority Boost*



Was haben wir gelernt?

1. on-line vs. off-line Scheduling
2. zeitgesteuertes vs. ereignisgesteuertes Scheduling
3. Interrupts
4. kooperatives vs. präemptives Multitasking
5. statische vs. dynamische Prioritäten
6. Round Robin; Einfluss der Zeitscheibenlänge
7. Priority Boost