Einführung in die Funktionsweise von Computersystemen

6. Kapitel

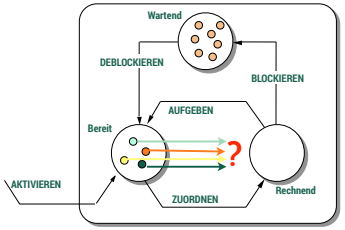
Ressource Prozessor: Scheduling

Prof. Matthias Werner

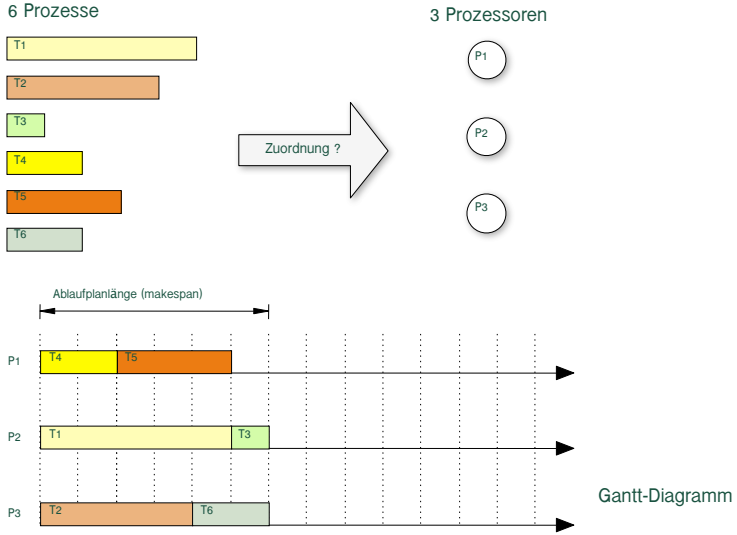
Professur Betriebssysteme

Einführung

- Bisher: Wenn ein Prozesses den Prozessor aufgibt (Zustand „laufend“ verlässt), wird ein anderer Prozess ausgewählt.
  - Dieser muss im Zustand „bereit“ sein.
  - Möglicherweise sind mehrere Prozesse bereit
- Problem: Welcher Prozess wird ausgewählt?
- Allgemeineres Problem:
  Zuordnung von Aktivitäten zu Ressourcen in Raum und Zeit
  ➡ Scheduling (etwa: Ablaufplanung)
  - Schedulingtheorie ist Teilgebiet der Mathematik und älter als die Informatik



Klassisches Scheduling-Problem

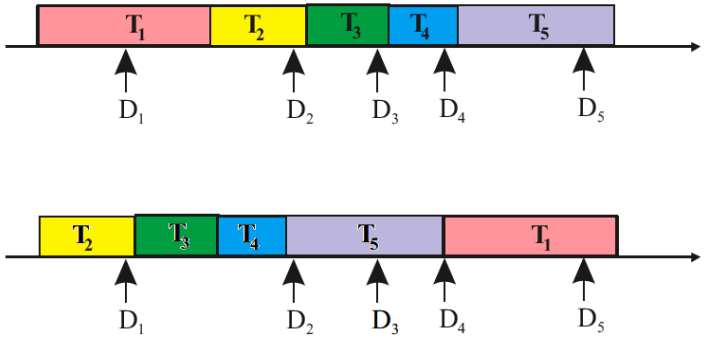


Schedulingziel

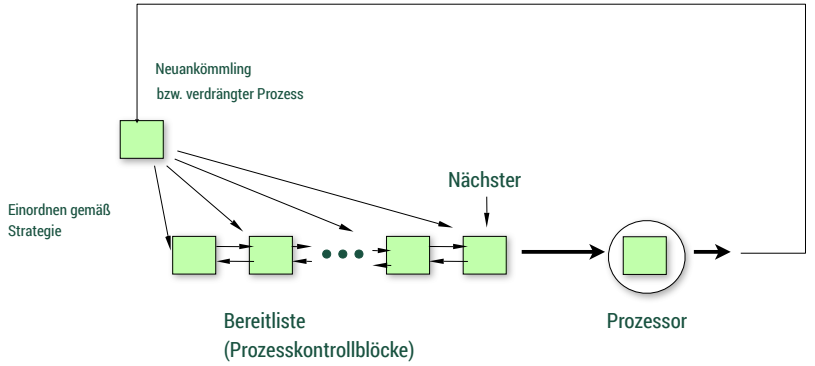
- Welches Schedulingverfahren gewählt werden soll, hängt vom Ziel ab, das erreicht werden soll
  - **Benutzerorientierte Ziele**
    - Länge des Ablaufplans ➡ minimal
    - Maximale Antwortzeit ➡ minimal oder Frist einhalten
    - Mittlere (gewichtete) Antwortzeit ➡ minimal
    - Maximale Verspätung bzgl. Frist ➡ minimal
    - Anzahl verspäteter Prozesse ➡ minimal
    - ...
  - **Systemorientierte Ziele**
    - Anzahl benötigter Prozessoren ➡ minimal
    - Durchsatz ➡ maximal
    - Prozessorauslastung ➡ maximal
    - ....
  - Ziele können einander widersprechen!

Beispiel widersprechender Ziele

- Geringste maximale Verspätung vs. minimale Anzahl verspäteter Prozesse



Zur Erläuterung der Umschaltstrategien



Standardstrategien

- Durchgängiges Beispiel:
- Gegeben seien die folgenden fünf Prozesse:

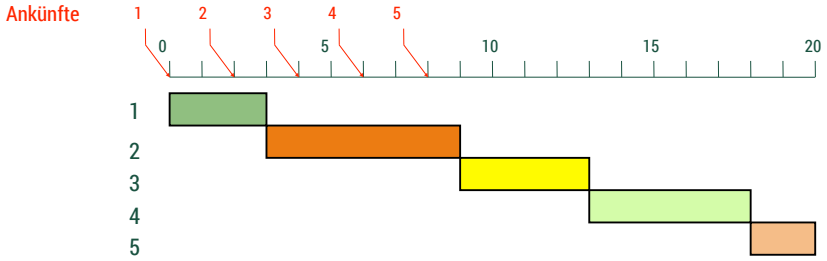
#	Ankunft	Bedienzeit	Priorität
1	0	3	2
2	2	6	4
3	4	4	1
4	6	5	5
5	8	2	3

First Come First Served

FCFS (auch FIFO (First In First Out) genannt)

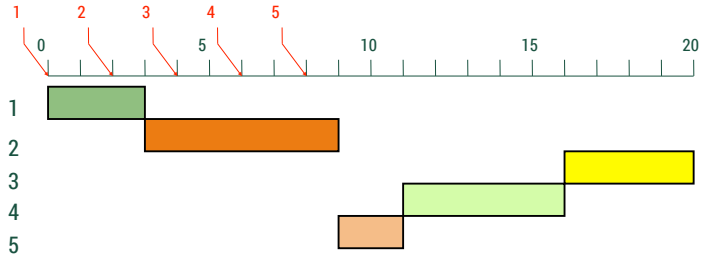
- Arbeitsweise:
  - Bearbeitung der Prozesse in der Reihenfolge ihrer Ankunft in der Bereitliste.
  - Prozessorbesitz bis zum Ende oder zur freiwilligen Aufgabe.

  Anmerkung: Entspricht der Alltagserfahrung.



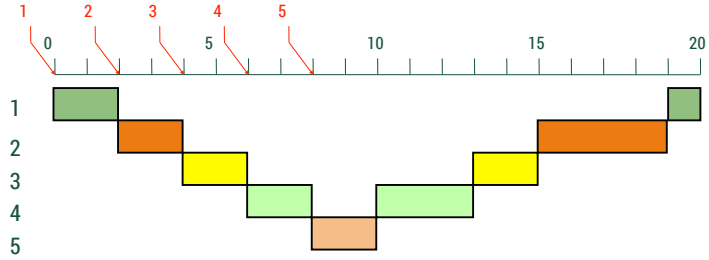
Last Come First Served

- LCFS, Arbeitsweise:
  - Bearbeitung der Prozesse in der umgekehrten Reihenfolge ihrer Ankunft in der Bereitliste.
  - Prozessorbesitz bis zum Ende oder zur freiwilligen Aufgabe.
    Anmerkung: In dieser reinen Form selten benutzt.



Last Come First Served - Preemptive Resume

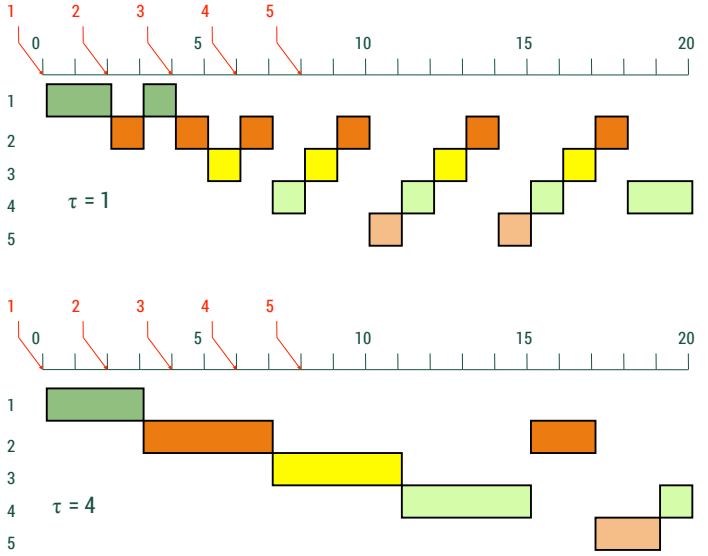
- LCFS-PR, Arbeitsweise:
  - Neuankömmling in Bereitliste verdrängt den rechnenden Prozess
  - Verdrängter Prozess wird hinter dem verdrängenden eingereiht
  - Falls keine Ankünfte ➡ Abarbeitung der Liste ohne Verdrängung
- **Anmerkung**:
  - Ziel ist die Bevorzugung kurzer Prozesse.
  - Kurzer Prozess hat die Chance, noch vor nächster Ankunft fertig zu werden.
  - Langer Prozess wird u.U. mehrfach verdrängt.



Round Robin (Zeitscheibenverfahren)

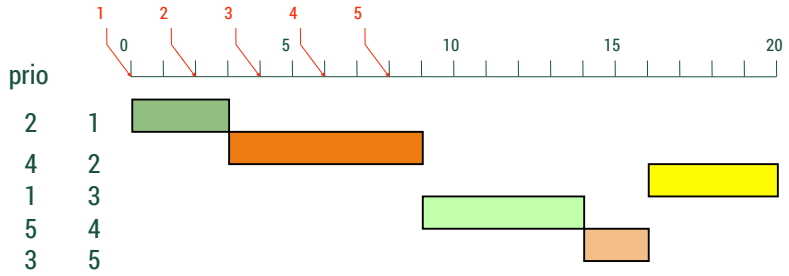
- RR, Arbeitsweise:
  - Bearbeitung der Prozesse in Ankunftsreihenfolge.
  - Nach Ablauf einer vorher festgesetzten Frist (Zeitscheibe, time slice, CPU-quantum) findet eine Verdrängung statt und es wird auf den nächsten Prozess umgeschaltet.
- **Anmerkung**:
  - Ziel des Verfahrens ist die gleichmäßige Verteilung der Prozessorkapazität und der Wartezeit auf die Prozesse.
  - Wahl der Zeitscheibenlänge τ ist Optimierungsproblem:
  - Für großes τ nähert sich RR der Reihenfolgestrategie FCFS.
  - Für kleines τ schlägt der Aufwand für das häufige Umschalten negativ zu Buche.
  - Üblich sind Zeiten im msec-Bereich.

RR Round Robin



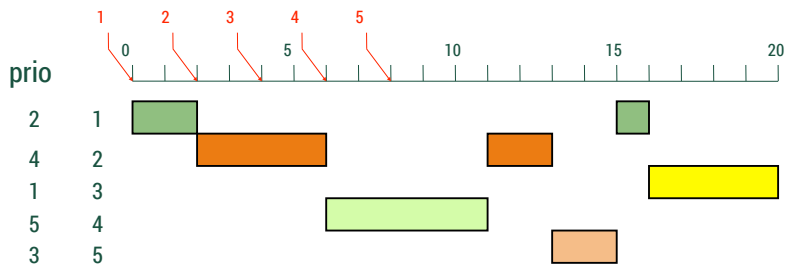
Priorities - Nonpreemptive

- PRIO-NP, Arbeitsweise:
  - Neuankömmlinge werden nach ihrer Priorität in die Bereitliste eingeordnet.
  - Prozessorbesitz bis zum Ende oder zur freiwilligen Aufgabe.



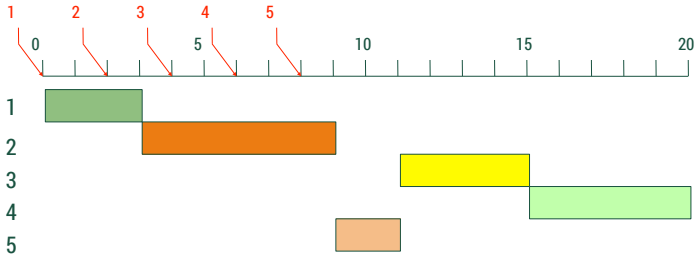
Priorities - Preemptive

- PRIO-P, Arbeitsweise:
  - Wie PRIO-NP, jedoch findet Verdrängungsprüfung statt, d.h. der rechnende Prozess wird verdrängt, wenn er eine geringere Priorität hat als der Neuankömmling.



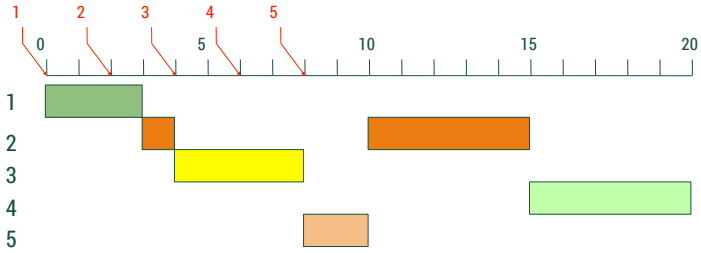
Shortest Job Next

- SJN, Arbeitsweise:
  - Prozess mit der kürzesten Bedienzeit wird als nächster bis zum Ende oder zur freiwilligen Aufgabe bearbeitet.
  - Wie PRIO-NP, wenn man die Bedienzeit als Prioritätskriterium verwendet.
- **Anmerkung**:
  - Bevorzugt kurze Prozesse und führt daher zu kürzeren mittleren Antwortzeiten als FCFS.



Shortest Remaining Time Next

- SRTN, Arbeitsweise:
  - Prozess mit der kürzesten Restbedienzeit wird als nächster bearbeitet.
  - Rechnender Prozess kann verdrängt werden.
- **Anmerkung**:
  - Beide Algorithmen (SJN und SRTN) haben den Nachteil, dass sie Kenntnis der Bedienzeit benötigen, die nur vom Benutzer in Form einer Schätzung stammen kann. Längere Prozesse können „verhungern“, wenn immer kürzere vorhanden sind.



Highest Response Ratio Next

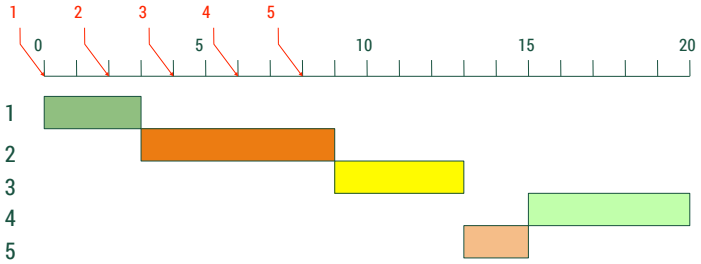
- HRN, Arbeitsweise:
  - Die Response Ratio rr ist definiert als

$$rr = \frac{\text{Wartezeit} + \text{Bedienzeit}}{\text{Bedienzeit}}$$

  - rr wird dynamisch berechnet und als Priorität verwendet:
  - Der Prozess mit dem größten rr-Wert wird als nächster ausgewählt.
  - Die Strategie ist nicht verdrängend.

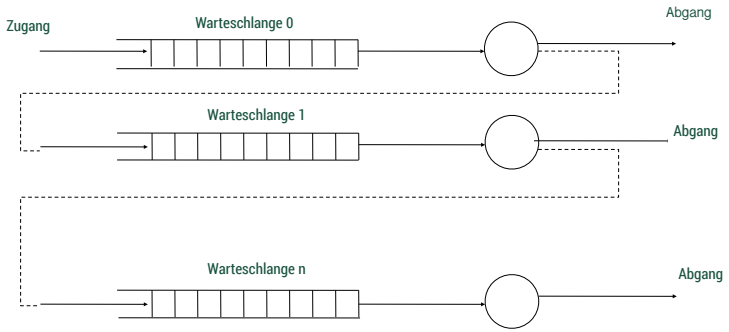
- **Anmerkung**:
  - Wie bei SJN werden kurze Prozesse bevorzugt, lange Prozesse müssen aber nicht ewig warten, sondern können durch Warten „Punkte sammeln“.

HRN Highest Response Time Ratio Next

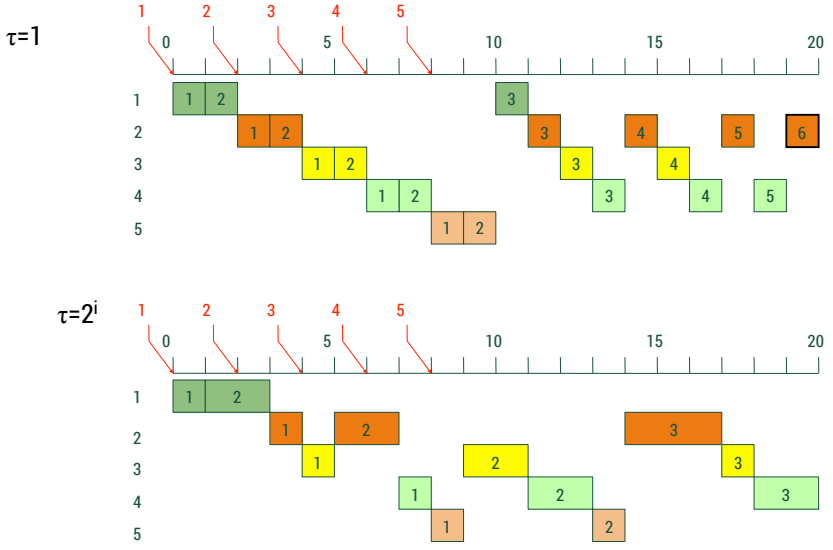


(Multilevel) Feedback

- FB, Arbeitsweise:
  - Kennt man die Bedienzeit a priori nicht, möchte aber trotzdem lang laufende Prozesse benachteiligen, so kann man den Prozess nach jeder CPU-Benutzung in seiner „Priorität“ schrittweise herabsetzen.
  - Die einzelnen Warteschlangen können nach Round Robin verwaltet werden.
  - Unterschiedliche Zeitscheibenlängen τ sind möglich.



Feedback



Standard-Umschaltstrategien (Übersicht)

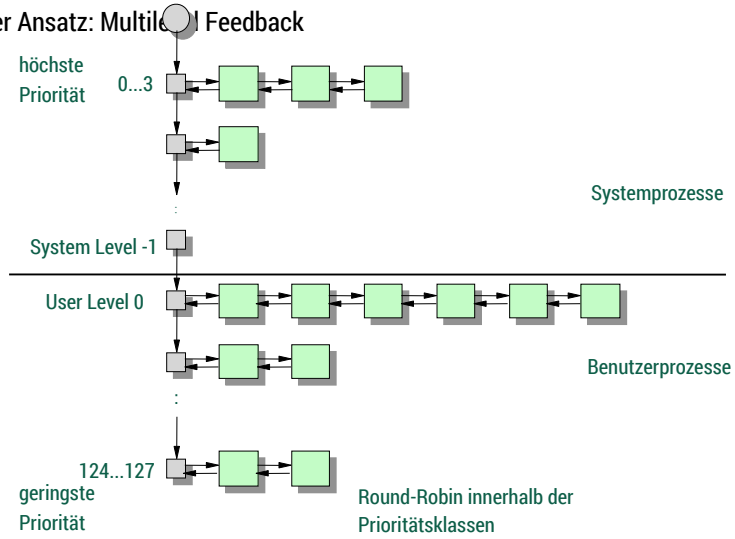
	ohne Verdrängung		mit Verdrängung	
	ohne Prioritäten	mit Prioritäten	ohne Prioritäten	mit Prioritäten
BZ-unabhängig	**FCFS, LCFS**	**PRIO-NP**	**LCFS-PR, RR, FB**	**PRIO-P**
BZ-abhängig	**SJN, HRN**		**SRTN**	

Fallbeispiel UNIX

- Wir betrachten als reales Beispiel das Scheduling im „klassischen“ UNIX (BSD, System V)
- Prinzipieller Ansatz: **Multilevel feedback queues**
- Ziel: Bevorzugung interaktiver Jobs
- Darstellung von Prioritäten durch verschiedene Queues
  - Prioritäten von 0 bis 127
    - kleine Werte ➡ hohe Priorität, große Werte ➡ niedrige Priorität
  - Aufteilung der bereiten Prozesse auf 32 Run-Queues (Prioritätsklassen)
    - 4 Prioritäten = eine Prioritätsklasse pro Queue
- Timer-Tick: typisch alle 10ms (VAX)

Feedback Queues in UNIX

- Prinzipieller Ansatz: Multilevel Feedback



Berechnung der Priorität

- Bei jedem Tick: $L_{CPU} = L_{CPU} + 1$
- Berechnung der Prozesspriorität P_{Proc} bei **jedem vierten** Tick (alle 40 ms)

$$P_{Proc} = P_{basis} + \left\lceil \frac{L_{CPU}}{4} \right\rceil + 2 \cdot P_{nice}$$

  - L_{CPU}: Abschätzung der CPU-Zeit dieses Prozesses
  - P_{nice}: nutzersetzbare „Wichtigkeit“ -20 ... 19
  - P_{Basis}: Basispriorität, hängt vom Art und Zustand des Prozesses ab
    - Normale Nutzerprozesse starten mit dem höchsten (niedrigster Wert!) Nutzerprozess-Level (typisch 50...60)
    - Prozesse im Kernel haben (je nach Aufgabe) höhere Prioritäten
- Werte von P_{Proc} > 127 werden auf 127 festgelegt

Alterung

- Starke Prozessorbenutzung führt zu einer schlechten Priorität!
  ➡ **Alterung**
- Rechenintensive Prozesse werden „benachteiligt“
- I/O-intensive Prozesse werden bevorzugt
  - Sie belegen den Prozessor nur kurz, um einen E/A-Auftrag abzusetzen.
  - Man erreicht dadurch eine **hohe Parallelität** zwischen den aktiven Rechnerkomponenten (CPU und Peripherie).

Glättung

- Mit der Zeit wächst L_{CPU} und damit die Priorität
  ➡ alle (hinreichend lange laufende) Prozesse hätten nach einiger Zeit hohe (schlechte) Prioritätswerte
- Glättung des Wertes der Prozessornutzung L_{CPU} **einmal pro Sekunde**
  - Reduzierung des Einflusses der letzten Rechenzeiten (Dämpfungsfilter)
  - l_{RQ} - Durchschnitt der Länge der Ready-Queue während der letzten Minute

$$L'_{CPU} = \frac{2 \cdot l_{RQ}}{2 \cdot l_{RQ} + 1} \cdot L_{CPU} + P_{nice}$$

Beispiel Dämpfungsfilter

- Auswirkung der Systemlastanpassung
  - Einzelner Prozess, sammelt T_i Timerticks in der i. Sekunde an, nice=0

$$\begin{aligned}
L^{(1)}_{CPU} &= \frac{2}{3}T_0 \\
L^{(2)}_{CPU} &= \frac{2}{3}\left(T_1 + \frac{2}{3}T_0\right) = \frac{2}{3}T_1 + \frac{4}{9}T_0 \\
L^{(3)}_{CPU} &= \frac{2}{3}T_2 + \frac{4}{9}T_1 + \frac{8}{27}T_0 \\
L^{(4)}_{CPU} &= \frac{2}{3}T_3 + \cdots + \frac{16}{81}T_0 \\
L^{(5)}_{CPU} &= \frac{2}{3}T_4 + \cdots + \frac{32}{243}T_0
\end{aligned}$$

$$\frac{32}{243} \approx 0,13$$

➡ nach fünf Sekunden gehen nur noch etwa 13 % der „Altlast“ ein

Zusammenfassung

- schnelle Ermittlung des nächsten Prozesses bei vielen Prozesse mit unterschiedlicher Priorität
- aufwendige Neuberechnung der Priorität
- u. U. Entnahme und Neueinreihen in Queue anderer Priorität
- Neuberechnung der Priorität nur für den aktuellen Prozess
- Berücksichtigung der aktuellen Last
- keine Echtzeit-Priorisierung