3 Prozessorzuteilungsverfahren für Echtzeitsysteme

3.1 Überblick

3.1.1 Zielstellung

Finden einer Abbildung *Ausführungseinheit→Prozessor*, so daß

- alle notwendigen Ressourcen (Rechenzeit, Speicherplatz, Geräte, ...) den Ausführungseinheiten zur Verfügung stehen und
- alle Zeitforderungen garantiert eingehalten werden.

Scheduling (Planen) = Entscheidung, welcher Prozeß zu welchem Zeitpunkt für welche Zeitspanne abgearbeitet wird.

Häufig werden in Echtzeitsystemen nicht nur der Prozessor sondern auch andere Ressourcen geplant und zugeteilt (z.B. Festplatten, Busse).

„Klassische“ Ziele der Prozessorzuteilung:

1. *Minimierung der (mittleren) Reaktionszeit*: z.B. bei interaktiven Systemen
2. *Maximierung des Durchsatzes*: z.B. bei Servern
3. *Maximierung der Prozessorauslastung*: z.B. bei Supercomputern
4. *Fairneß*: Gleichbehandlung aller Prozesse, Garantie des Fortschritts

In Echtzeitsystemen:

5. *Einhaltung der vorgegebenen Zeitschranken*

Sekundärziele: 1.-4.

3.1.2 Klassifikation von Schedulingverfahren

I. Nach dem Zeitpunkt („Wann wird geplant?“)

a) zur Laufzeit des Systems (*on-line* Scheduling, implizite Planung)

- bei Zustandsübergängen,
- bei externen Ereignissen,
- nach Verstreichen einer festen Zeitspanne,
- bei Aufruf eines Systemrufs

b) vor Laufzeit des Systems (*off-line* Scheduling, explizite Planung)

- alle Prozeßparameter vor Systemstart bekannt
- geschlossene Prozeßmengen
- Umschaltung sehr schnell, da keine Planung zur Laufzeit nötig

II. Nach der Unterbrechbarkeit („Kann einem Prozeß der Prozessor entzogen werden?“)

a) kooperatives Scheduling

- Prozeß gibt freiwillig den Prozessor ab
- Behandlung kritischen Abschnitte einfach
- geringerer Aufwand

b) präemptives Scheduling

- Prozeß kann von außen (durch das BS) unterbrochen werden
- Prioritätssteuerung möglich
- keine Beeinflussung der Prozesse untereinander

III. zeit- oder ereignisgesteuert

a) zeitgesteuertes Scheduling

- on-line und off-line möglich

- keine Interrupts, Interaktion ausschließlich durch Polling

b) ereignisgesteuertes Scheduling

- nur on-line möglich
- Aktivierung von Tasks in Folge externer Ereignisse
- Interrupts erlaubt

IV. weitere Kriterien

- Nutzung von Prioritäten (keine, fest, variabel)
- nach Anzahl der Prozessoren, Möglichkeit der Migration
- nach Taskmodell: z.B. periodisches Taskmodell
- Berücksichtigung von Prozeß-Präzedenzen (Abhängigkeiten)

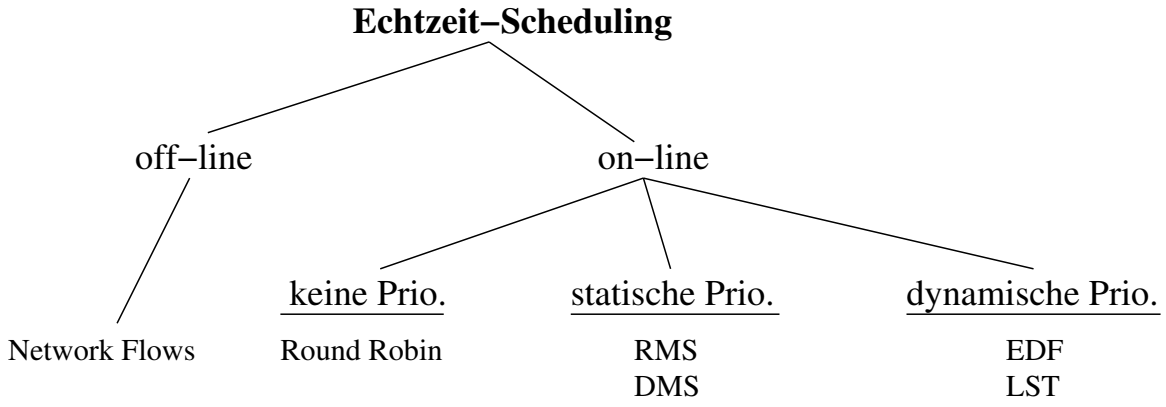


Abbildung 1: Überblick über einige Schedulingverfahren für Echtzeitsysteme

3.2 Wichtige Begriffe und Parameter

Job: Programmeinheit, die geplant und ausgeführt wird.

Task: Menge an (zusammengehörigen) Jobs, die zusammen eine Aufgabe lösen bzw. eine Funktion ausüben.

Ein Job wird durch eine aktive Ressource ausgeführt, z.B. CPU, Festplatte(-ncontroller), Netz(-karte); d.h., ein Prozessor ist erforderlich. [vgl. Liu, S. 26ff.]

Deadline: Zeitpunkt, bis zu dem Job bzw. Task ausgeführt sein muß

Jitter: (unerwünschtes) Schwanken der Ausführungszeiten der Jobs einer Task

Schedule: Plan, der die Abarbeitungsreihenfolge von Aktivitäten festlegt

3.2.1 Zeitpunkte und Zeitspannen

Jobs werden zu bestimmten Zeitpunkten (t_r) bereit, nämlich wenn sie alle angeforderten Betriebsmittel erhalten haben. Von nun an können sie potentiell abgearbeitet werden, was zu t_s beginnt. Die Abarbeitung endet zu t_c. Idealerweise ist t_c kleiner als t_d, ansonsten wird die Deadline und damit die Echtzeitbedingung dieses Jobs verletzt (vgl. Abbildung 2).

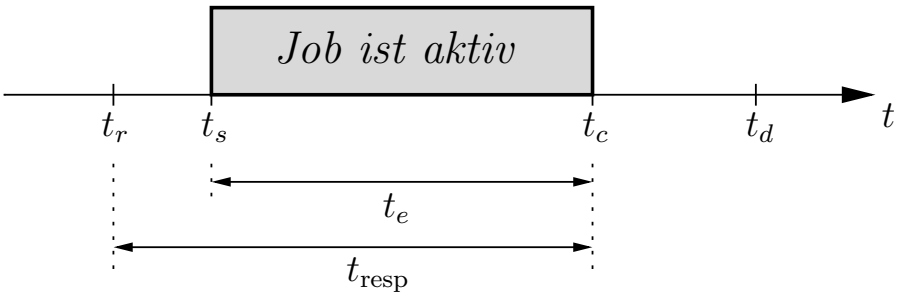


Abbildung 2: Zeiten und Zeitpunkte eines Jobs

Zeitpunkte:

release time	t_r	Zeitpunkt, ab dem ein Job ausgeführt werden kann
starting time	t_s	Zeitpunkt des Starts eines Jobs
completion time	t_c	Zeitpunkt der Komplettierung eines Jobs
deadline	t_d	Zeitpunkt, bis zu dem ein Job beendet sein muß

Zeitspannen:

execution time	Ausführungszeit	t_e	$t_c - t_s$
response time	Antwortzeit	t_{resp}	$t_c - t_r$
tardiness	Verspätung	t_{tard}	$t_c - t_d$

Anmerkungen:

- Antwortzeit = Reaktionszeit
- $t_s \geq t_r$
- $t_{\text{tard}} = 0$, wenn $t_c < t_d$ (Komplettierung vor Deadline)
- relative Deadline wird bezogen auf t_r
- absolute Deadline = t_r + relative Deadline

3.2.2 Wiederholung: Begriff des Prozesses

Prozeß: Abstraktion „Programm in Abarbeitung“

- Identifikator
- Sequenz von Anweisungen
- privater Adreßraum
- benötigt eine Menge von *Betriebsmitteln* (Ressourcen), z.B. Speicherblöcke, Geräte, Dateien usw.

- besitzt Zustand

Mikrozustand: Gesamtheit aller dem Prozeß gehörenden Betriebsmittel + der Prozessorzustand.

Makrozustand: $M \in \{\text{bereit}, \text{aktiv}, \text{wartend}\}$

Existieren mehrere Prozesse gleichzeitig → parallele oder nebenläufige Prozesse

Anzahl Prozesse >> Anzahl Prozessoren

→ Abbildung notwendig (zeitliche und/oder räumliche Teilung)

zeitliche Teilung: Prozessorzeit wird „portioniert“, jeder Prozeß erhält einen gewissen Anteil

räumliche Teilung: verschiedene Prozesse werden auf verschiedenen Prozessoren abgearbeitet

Jeder Prozeß besitzt einen *virtuellen* Prozessor:

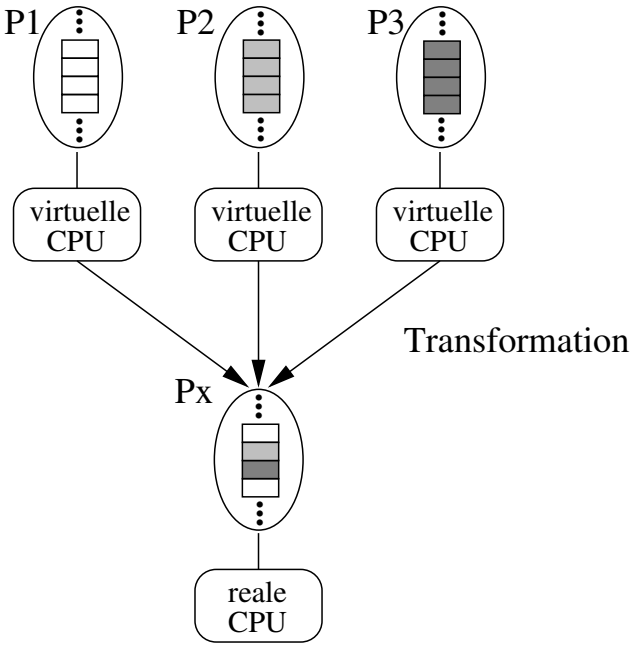


Abbildung 3: Abbildung virtueller Prozessoren auf den realen Prozessor

Transformation durch Prozeßumschalter des Betriebssystems.

Prozeßzustände

aktiv Prozeß wird abgearbeitet (besitzt alle zum Fortschritt nötigen Ressourcen)

bereit Prozeß besitzt alle zum Fortschritt nötigen Ressourcen, wird jedoch gerade nicht abgearbeitet

wartend Prozeß besitzt nicht alle zum Fortschritt nötigen Ressourcen (diese sind an andere Prozesse vergeben) oder Prozeß wartet auf ein Ereignis.

(nicht existent) Prozeß wurde beendet oder noch nicht gestartet.

(weitere Subzustände denkbar, hier nicht weiter betrachtet)

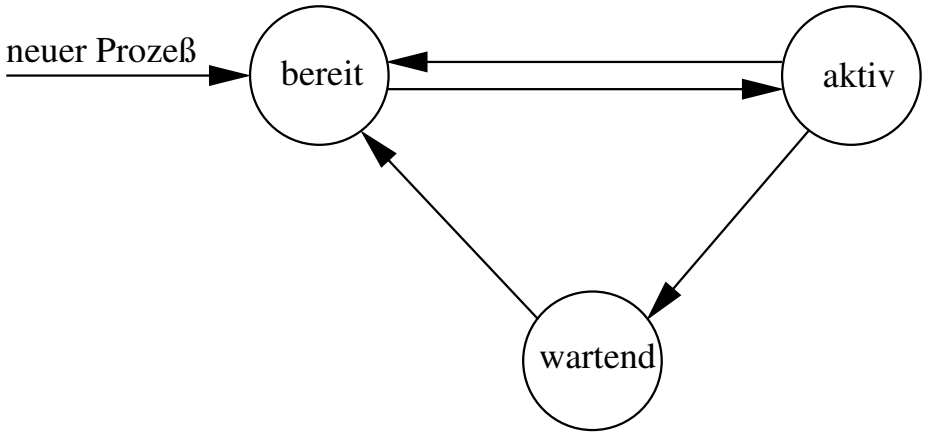


Abbildung 4: Mögliche Transitionen zwischen Prozeßzuständen

3.2.3 Periodisches Taskmodell

Sehr häufig kommen in Echtzeitsystemen periodische Taskmodelle zum Einsatz, da sie einfach zu analysieren sind und vielen Verarbeitungsaufgaben zugrunde liegen (Beispiel: multimediale Ströme).

Im periodischen Taskmodell werden (bis zu) drei verschiedene Arten von Tasks unterschieden:

periodisch: periodische Aktivierung der Jobs einer Task, i.a. als Tupel $J_i(t_{\phi,i}, t_{p,i}, t_{e,i}, t_{d,i})$ notiert mit folgender Parameterbedeutung:

Parameter		*Anmerkungen*
Phase	t_ϕ	default: 0
Periode	t_p	
Ausführungszeit	t_e	(pro Aktivierung)
(relative) Deadline	t_d	default: t_p

Da meist $t_{\phi,i} = 0$ und $t_{d,i} = t_{p,i}$, wird deren Angabe häufig fortgelassen.

aperiodisch: nichtperiodische Aktivierung der Jobs einer Task; die Zwischenankunftszeit zweier Jobs ist *nicht* nach unten beschränkt

sporadisch: nichtperiodische Aktivierung der Jobs einer Task; maximale Ankunftsrate der Jobs ist beschränkt

3.2.4 Auslastung

- nur für periodische Tasks bestimmbar
- Ermittlung für Task i:

$$u_i = \frac{t_{e,i}}{t_{p,i}}$$

- Gesamtauslastung u eines periodischen Tasksets mit n Tasks:

$$u = \sum_{i=1}^{n} u_i$$

- bei 1 Prozessor $u < 1$ nötig, sonst Überlast

3.2.5 Brauchbarkeit und Optimalität

Definition. *Ein Plan (Schedule) einer Menge Jobs heißt* ***brauchbar****, wenn jeder Job aus dieser Menge vor seiner individuellen Deadline komplettiert ist.*

Definition. *Ein Schedulingalgorithmus heißt* ***optimal****, wenn der Algorithmus zu einer gegebenen Menge an Jobs einen brauchbaren Schedule erzeugt, sofern dieser existiert.*

d.h., ein nicht-optimaler Algorithmus ist u.U. nicht in der Lage, einen brauchbaren Schedule zu erzeugen, obwohl dieser existiert.

3.2.6 Präzedenz

Häufig unterliegen Jobs einer vorgeschriebenen Reihenfolge („J_1 muß beendet sein, bevor J_2 starten kann."). Darstellung durch gerichteten Graphen, den *Präzedenzgraphen*:

- Knoten: Jobs (Tasks, Teil-Prozesse), bewertet mit Namen und Ausführungsparametern
- Kanten: Präzedenzrelationen zwischen Tasks, $J_1 \rightarrow J_2$ bedeutet, daß J_1 komplettiert sein muß, bevor J_2 gestartet werden darf

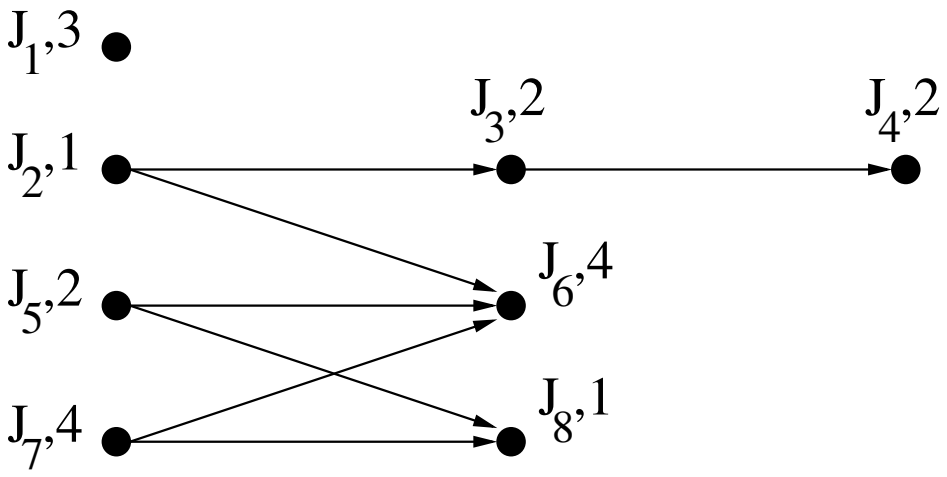


Abbildung 5: Beispiel für einen Präzedenzgraphen

zusätzlich notwendige Informationen:

- Bereit-Zeiten (t_r)
- Prioritäten, falls relevant

Aufgabe: Generieren Sie einen gültigen Schedule sowohl für präemptives als auch für nichtpräemptives Scheduling, wenn gilt:

- 2 Prozessoren, Migration möglich
- Prioritätsreihenfolge ($J_1, J_2, \ldots, J_8$) (d.h., J_1 hat höchste Priorität),
- alle Jobs sind zu t=0 bereit, außer J_5, welches erst zu t=4 bereit ist

Vorgehensweise:

1. t = 0
2. Bereitmenge $\mathcal{B}$ der Jobs bilden (Präzedenz und Bereitzeit beachten)
3. bei n Prozessoren: n höchstpriorisierte Jobs auswählen, abarbeiten
   - präemptiv: t = t + 1

- nichtpräemptiv: t bis zur Komplettierung des ersten der aktiven Jobs inkrementieren

4. Goto 2

Lösung: (präemptiv)

t	Ereignis	$\mathcal{B}$	zur Abarbeitung
0	-	J_1, J_2, J_7	J_1, J_2
1	J_2 beendet	J_1, J_3, J_7	J_1, J_3
3	J_1, J_3 beendet	J_4, J_7	J_4, J_7
4	J_5 bereit	J_4, J_5, J_7	J_4, J_5
5	J_4 beendet	J_5, J_7	J_5, J_7
6	J_5 beendet	J_7	J_7
8	J_7 beendet	J_6, J_8	J_6, J_8
9	J_8 beendet	J_6	J_6
12	J_6 beendet	-	Ende

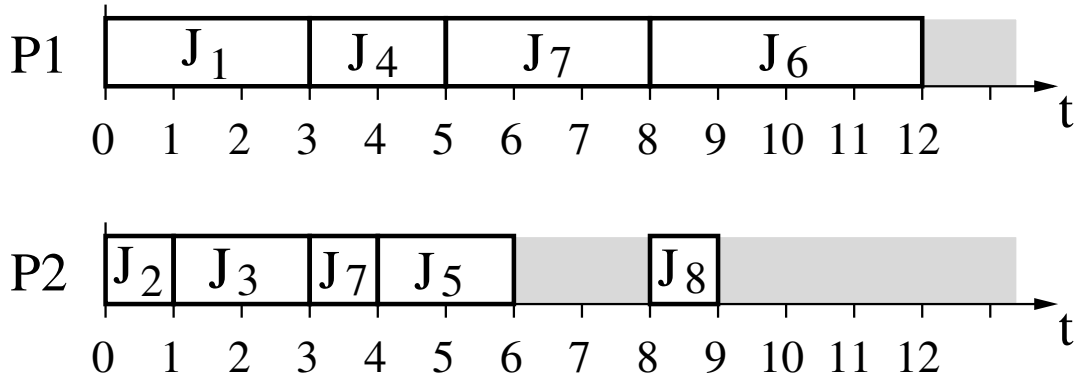


Abbildung 6: Schedule mit Prozessorentzug

Abbildung 7: Schedule ohne Prozessorentzug

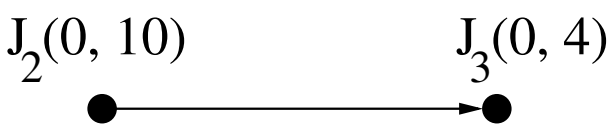
3.2.7 Effektive Deadlines und Bereitzeiten

Individuelle Deadlines und Bereitzeiten können zu den Präzedenzen inkonsistent sein:

a) Bereitzeit eines Jobs ist später als die seines Nachfolgers (⇒ Bereitzeit des Nachfolgers anpassen)

Legende: $J_x(t_r, t_d)$

b) Deadline eines Jobs ist eher als die seines Vorgängers (⇒ Deadline des Vorgängers anpassen)



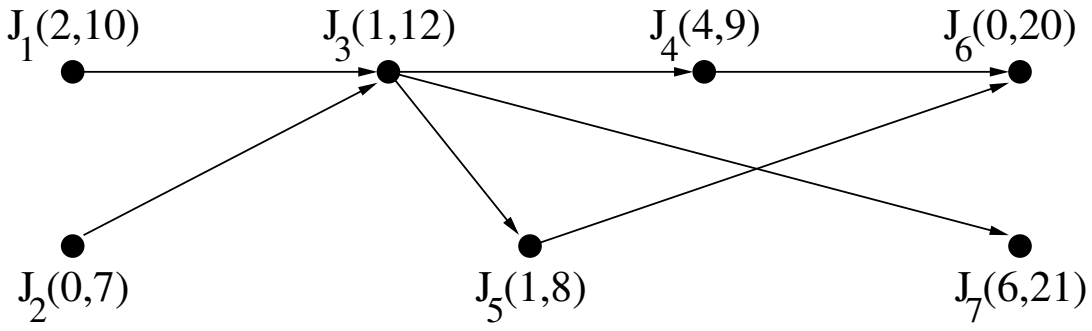
Definition. *Die **effektive Bereitzeit** eines Jobs ohne Vorgänger ist seine zugeordnete Bereitzeit. Die effektive Bereitzeit eines Jobs mit*

Vorgängern ist gleich dem Maximum aus seiner zugeordneten Bereitzeit und den effektiven Bereitzeiten aller seiner direkten Vorgänger.

Definition. *Die **effektive Deadline** eines Jobs ohne Nachfolger ist seine zugeordnete Deadline. Die effektive Deadline eines Jobs mit Nachfolgern ist gleich dem Minimum aus seiner zugeordneten Deadline und den effektiven Deadlines aller seiner direkten Nachfolger.*

Aufgabe:

Ermitteln Sie die effektiven Bereitzeiten und Deadlines der Jobs in folgendem Jobsystem $(J_x(t_r, t_d))$:



3.2.8 Der Taskgraph

- Erweiterung des Präzedenzgraphen
- erlaubt die Modellierung *verschiedener* Arten von Präzedenz:
  - „klassische“ Präzedenz (wie bisher)
  - AND/OR-Präzedenz: entweder Job muß auf Komplettierung *aller* unmittelbaren Vorgänger warten (klassisch; AND), oder er muß auf *wenigstens eine* Komplettierung warten (OR)
  - Modellierung bedingter Sprünge (*branch*, *join)*)

  - Pipeline-Beziehungen (ähnlich des Erzeuger-Verbraucher-Problems)
  - temporale Abhängigkeit (Jobs dürfen nur bestimmte Zeitspanne auseinander liegen; wichtig z.B. für Lippensynchronität)
- komplexere Abhängigkeiten einfacher modellierbar

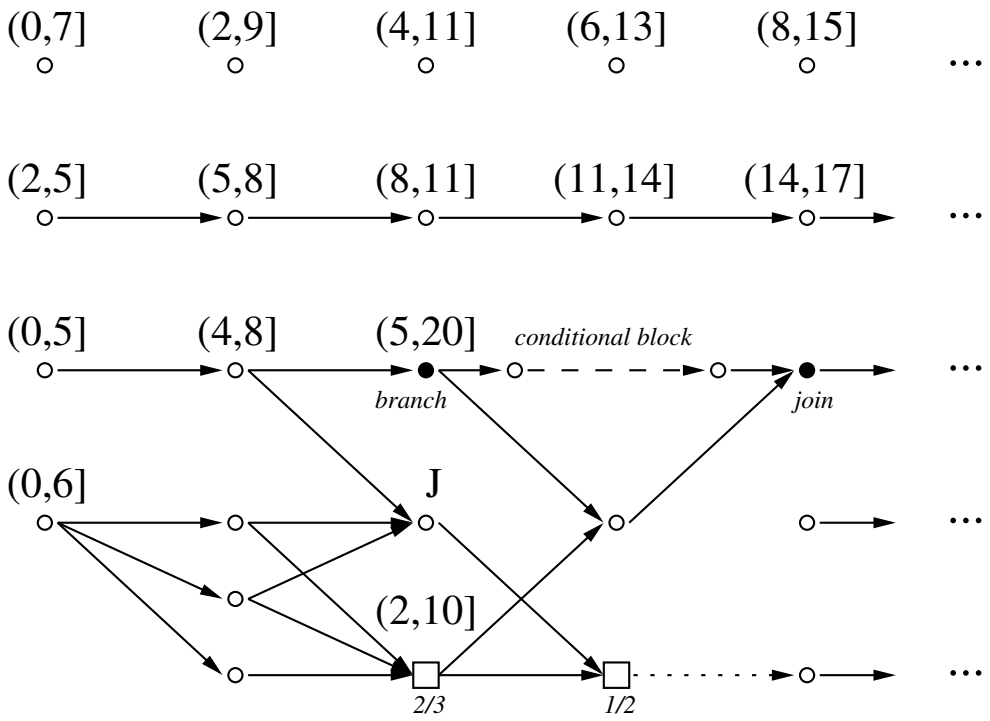


Abbildung 8: Beispiel für einen Task-Graphen

Anmerkungen:

- Knotenbewertungen sind (t_r, t_d)-Tupel, sogenannte *mögliche Intervalle*
- oberste Job-Reihe: periodische Task (Periode 2), Jobs überlappbar
- zweite Reihe: periodisch, nicht überlappbar
- dritte Reihe bedingter Sprung
- unterste Reihe: 2 OR-Präzedenzen und 1 Pipeline

3.2.9 Scheduling-Anomalien

Frage: Wie ermittelt man die Gesamtabarbeitungszeit, wenn t_c einer oder mehrerer Jobs variiert?

Beispiel: 4 Jobs, davon einer mit variabler Ausführungszeit (J_2):

$J_x(t_r, t_e, t_d)$: $J_1(0, 5, 10)$
$J_2(0, [2, 6], 10)$
$J_3(4, 8, 15)$
$J_4(0, 10, 20)$

keine Migration, 2 Prozessoren, absteigende Priorität (J_1 höchste Priorität, J_4 niedrigste), präemptive Prozessorvergabe.

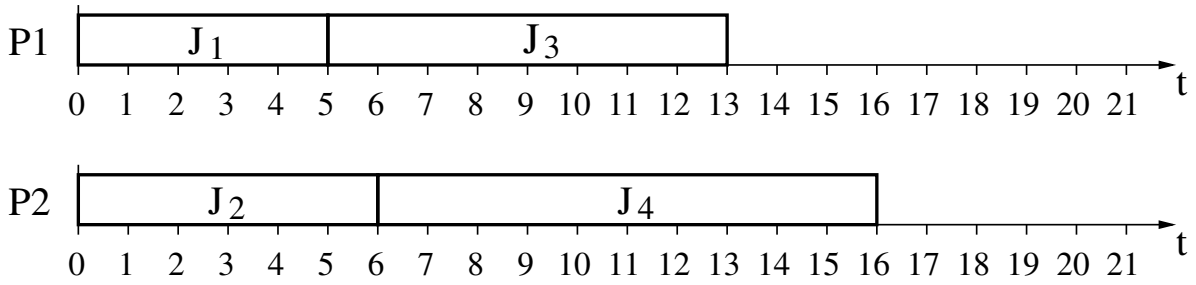


Abbildung 9: gültiger Schedule für $t_{e,2} = 6$ (Maximum)

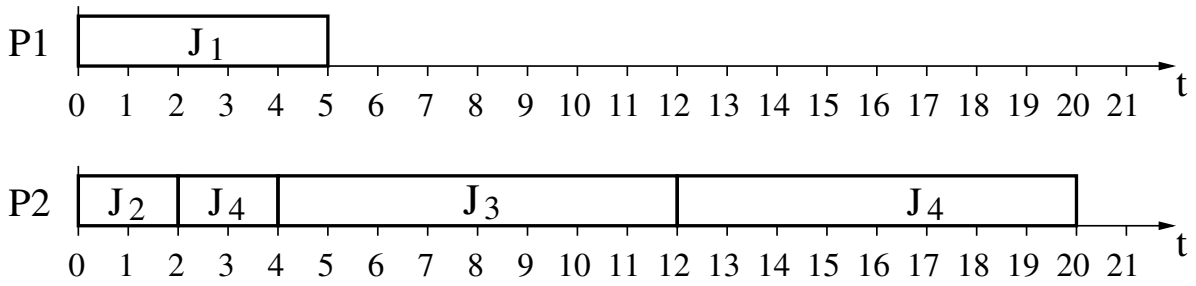


Abbildung 10: Schedule für $t_{e,2} = 2$ (Minimum)

Simulation mit Maximum und Minimum der t_e von J_2 ergibt gültige

Schedules (vgl. Abbildungen 9 und 10).

$\rightsquigarrow$ **Frage:** Ist das ausreichend?

Antwort: Nein! Gesamtausführungszeit für den Schedule ist mit $t_{e,2} = 3$ gleich 21; J_4 verpaßt seine Deadline:

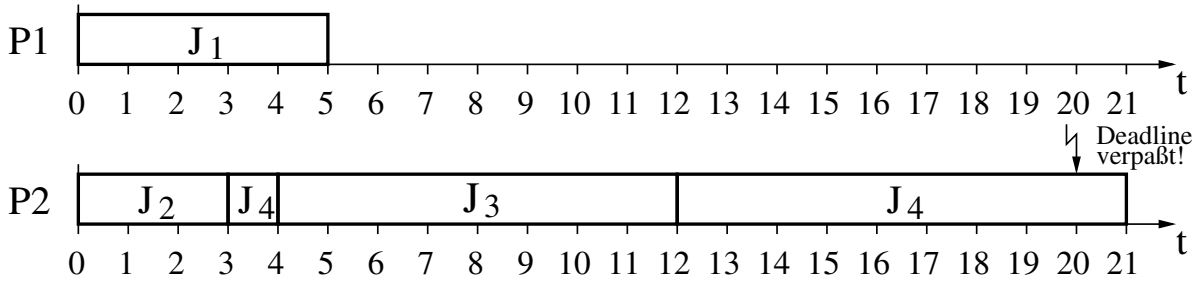


Abbildung 11: (ungültiger) Schedule für $t_{e,2} = 3$

Für $t_{e,2} = 5$ wird die Gesamtausführungszeit am kleinsten!

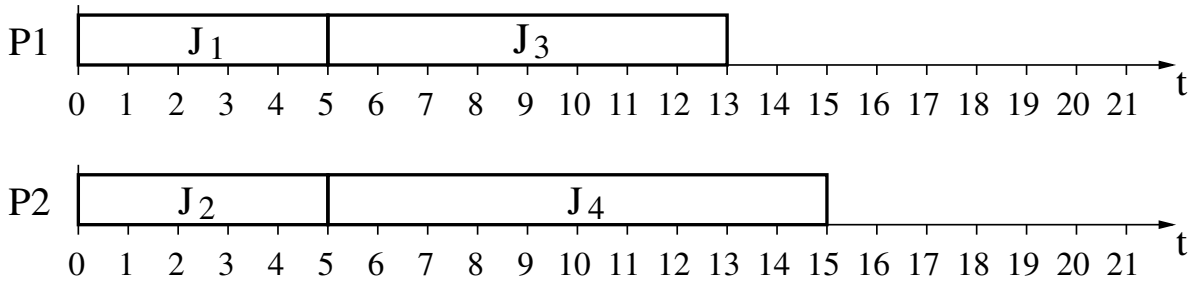


Abbildung 12: Schedule für $t_{e,2} = 5$

Sog. *Scheduling-Anomalie*, d.h. unerwartetes Zeitverhalten (eines prioritätsgesteuerten Systems)

Schlußfolgerung:

- Validierung des Zeitverhaltens von Echtzeitsystemen mit variablem Timing besonders aufwendig
- konstante Ausführungszeiten erwünscht

3.3 Einfache Zuteilungsverfahren

3.3.1 Earliest Deadline First (EDF)

Algorithmus: „Wähle unter den bereiten Job denjenigen zur Bearbeitung aus, dessen Deadline am ehesten erreicht ist.“ (prioritätsbasiertes Verfahren)

EDF ist optimal, wenn

- präemptive Prozessorvergabe,
- 1 Prozessor und
- keine Konkurrenz um Ressourcen.

Beweis:

Idee: Jeder brauchbare Schedule läßt sich systematisch in einen EDF-Schedule transformieren.

Ausgangspunkt: 2 Jobs J_i, J_k innerhalb eines brauchbaren Schedule, mit $t_{d,i}$ und $t_{d,k}$. Es soll gelten $t_{d,i} > t_{d,k}$. J_i ist im Intervall I_i geplant, J_k im Intervall I_k (siehe Abbildung 13).

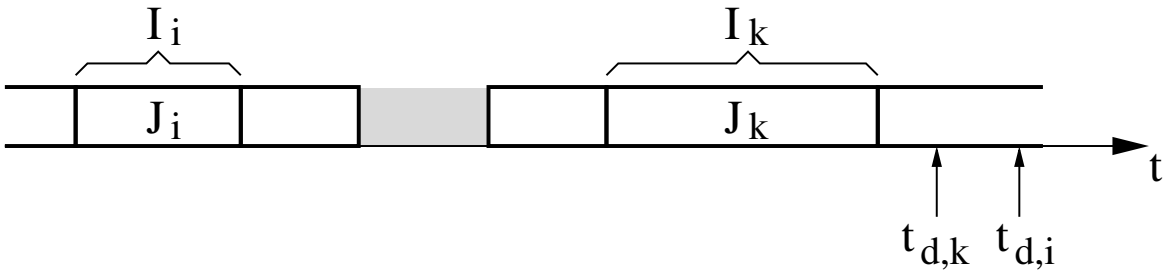


Abbildung 13: ein brauchbarer Schedule

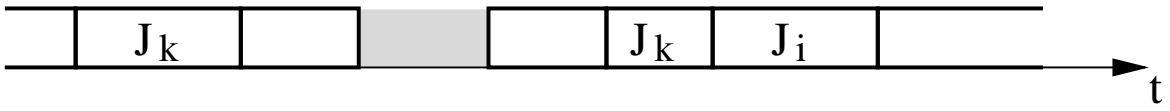
Transformation in EDF-Schedule: 2 Möglichkeiten

a) $t_{r,k} > I_i \rightarrow$ beide Jobs nach EDF geplant

b) $t_{r,k} < I_i \rightarrow$ Austausch von J_i und J_k nötig

Austausch mit Fallunterscheidung:

1. $I_i < I_k$: Teil von J_k in I_i abarbeiten, Rest in I_k, J_i danach

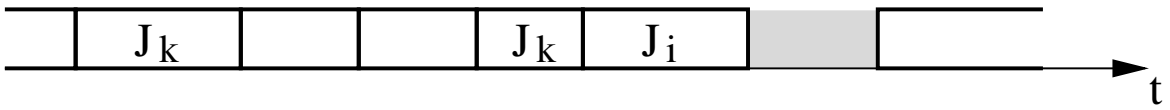


2. $I_i > I_k$: J_k und danach einen Teil von J_i in I_i abarbeiten, Rest von J_i in I_k

$\rightarrow$ Austausch ohne Verletzung der Gültigkeit des Schedule immer möglich

Verfahren auf alle Jobpaare J_x, J_y des Schedule anwenden.

Durch Vertauschung ist es möglich, daß der Prozessor nichts abarbeitet, obwohl Jobs bereit sind (vgl. Abb.) $\rightsquigarrow$ „Vorziehen“ notwendig:



Diese Verschiebung ist immer möglich.

$\Rightarrow$ Aus jedem brauchbaren Schedule kann ein EDF-Schedule generiert werden. $\Rightarrow$ EDF erzeugt brauchbaren Schedule, sofern dieser überhaupt existiert. □

EDF mit nichtpräemptiver Prozessorvergabe

EDF nicht optimal, wenn keine präemptive Prozessorvergabe. Beweis durch Angabe eines Gegenbeispiels:

3 Jobs $J_x(t_r, t_e, t_d)$: $J_1(0, 3, 10)$
$J_2(2, 6, 14)$
$J_3(4, 4, 12)$

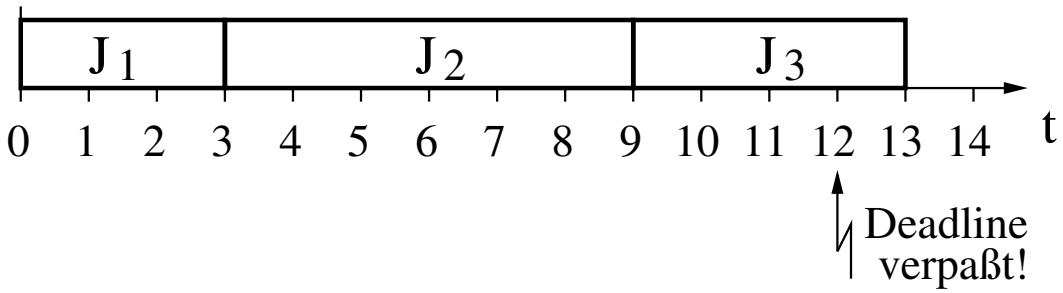


Abbildung 14: durch EDF generierter (ungültiger) Schedule

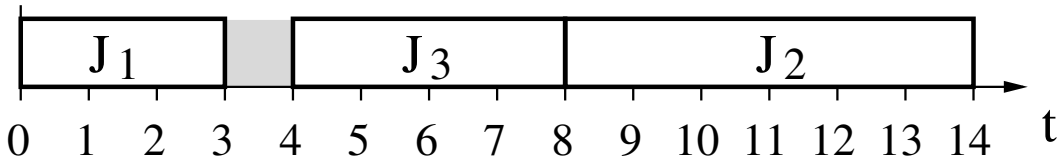


Abbildung 15: gültiger Schedule (nicht durch EDF generiert)

Kein prioritätsgesteuertes Verfahren kann den gültigen Schedule erzeugen, da diese per definitionem niemals den Prozessor leer lassen, wenn bereite Jobs existieren!

EDF auf mehreren Prozessoren

EDF nicht optimal bei mehr als einem Prozessor. Beweis durch Angabe eines Gegenbeispiels (3 Jobs, 2 Prozessoren):

3 Jobs $J_x(t_r, t_e, t_d)$: $J_1(0, 1, 1)$
$J_2(0, 1, 2)$
$J_3(0, 5, 5)$

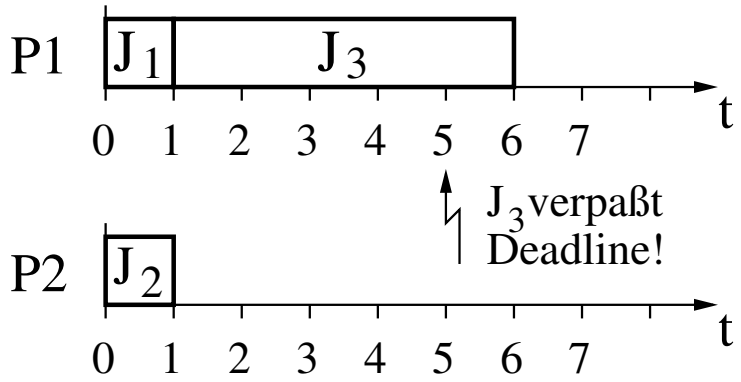


Abbildung 16: durch EDF generierter (ungültiger) Schedule

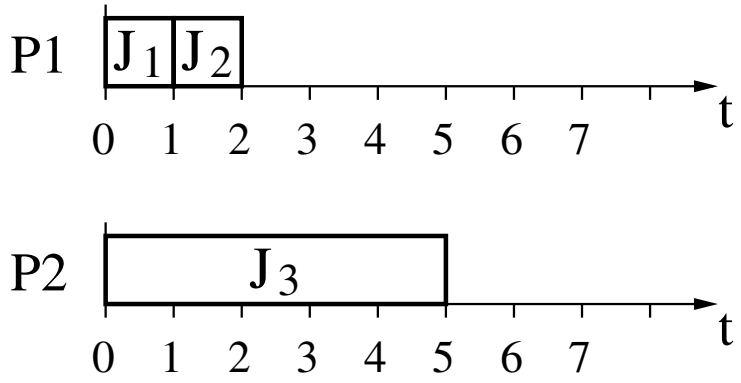


Abbildung 17: gültiger Schedule

3.3.2 Latest Release Time (LRT) oder "reverse EDF"

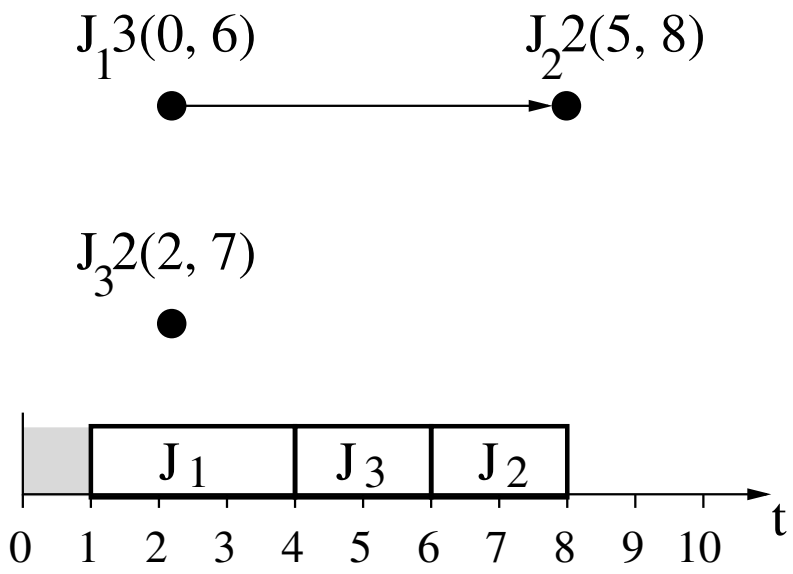
EDF: Jobs so früh wie möglich abgearbeitet; dies kann in bestimmten Situationen unerwünscht sein.

Idee: Jobs spätestmöglich bearbeiten, zuvor nach Möglichkeit andere (Nicht-Echtzeit-) Jobs einschieben (deren Reaktionszeit damit minimiert wird).

Algorithmus: Beginnend am Maximum aller Deadlines wird der Schedule „rückwärts" bis zum Startzeitpunkt aufgebaut. Wenn mehrere Prozesse zu einem Zeitpunkt bereit sind, wird stets derjenige

mit der spätesten Bereitzeit ausgewählt.

Beispiel: 3 Jobs J_x $t_e(t_r, t_d)$



LRT ist optimal unter den gleichen Bedingungen wie EDF. LRT ist nicht prioritätsbasiert, da es u.U. den Prozessor *idle* läßt, obwohl bereite Jobs existieren.

3.3.3 Least Slack Time First (LST)

Parameter *slack time* eines Jobs: $t_s(t) = t_d -$ verbleibende $t_e - t$.

(„Pufferzeit"; Wieviel Zeit ist noch, bis Job unbedingt begonnen werden muß?)

Algorithmus: „Wähle den Job zur Abarbeitung aus, dessen *slack time* am kleinsten ist."

Beispiel: J_1 $3(0, 6)$ hat zum Zeitpunkt t=0 die *slack time* $t_s(0) = 6 - 3 - 0 = 3$. Wird er abgearbeitet, so bleibt t_s konstant; wenn nicht, so reduziert sich t_s (z.B. $t_s(2) = 6 - 3 - 2 = 1$. Erreicht die *slack time* 0, so muß der Job abgearbeitet werden, sonst verletzt er seine Deadline.

- optimal für 1 Prozessor und präemptive Prozessorvergabe
- prioritätsbasiertes Verfahren
- Wissen über t_e notwendig → Nachteil! (Warum?)
- aka *Minimum Laxity First (MLF)*

3.4 Zeitgesteuertes Scheduling

3.4.1 Prinzip

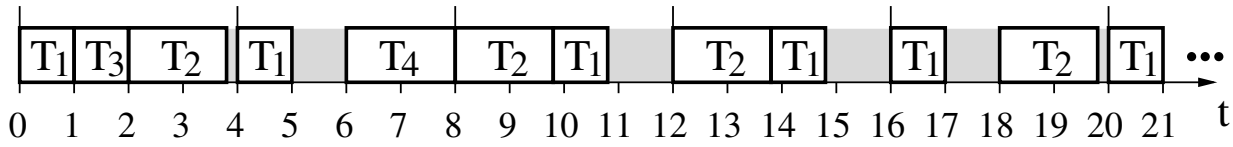
- Grundidee: Ermittlung des Schedule off-line, zur Laufzeit des Systems wird der Schedule nur Position für Position abgearbeitet
- periodisches Taskmodell nötig (sonst unendlicher Schedule, nicht speicherbar)
- Ermittlung des Schedule off-line → komplexe Algorithmen nutzbar (hier: Netzwerkflüsse)
- konstante Anzahl periodischer Echtzeit-Tasks $T_i(t_{\phi,i}, t_{p,i}, t_{e,i}, t_{d,i})$, 1 Prozessor
- Schedule einer Menge aus n Tasks besteht aus aneinandergereihten Segmenten der Länge $H = \text{kgV}(t_{p,i}) \quad i = 1, 2, \ldots, n$
- H wird *Hyperperiode* genannt
- Tasks ohne Deadline konsumieren bei Bedarf verbleibende Zeit

Beispiel: 4 periodische unabhängige Tasks $T_i(t_{p,i}, t_{e,i})$

$T_1(4, 1)$ $T_2(5, 1.8)$ $T_3(20, 1)$ $T_4(20, 2)$.

$$u = \frac{1}{4} + \frac{1.8}{5} + \frac{1}{20} + \frac{2}{20} = 0.76 \qquad H = 20$$

brauchbarer Schedule (ad hoc):



- Intervalle, in denen keine periodische EZ-Task abgearbeitet wird (z.B. [3.8, 4] oder [5, 6]) → für aperiodische Jobs
- aperiodische Jobs beim Eintreffen in Warteschlange (FIFO bzw. priorisiert) eingeordnet

Datenstruktur zur Beschreibung des Schedule: Tabelle über 1 Hyperperiode mit Einträgen $(t_k, T(t_k))$ $\quad k = 0, 1, \ldots, n-1$

t_k Schedulingzeitpunkte

$T(t_k)$ assoziierte Tasks, bzw. „I“ für *idle*

für das Beispiel:
(0, T_1) (1, T_3) (2, T_2) (3.8, I) (4, T_1) (5, I) (6, T_4) ... (19.8, I)

3.4.2 Implementierung eines zeitgesteuerten Schedulers (Cyclic Executive)

Voraussetzungen:

- Schedule als Tabelle abgelegt
- freier Timer

---

aktueller Tabelleneintrag k = 0
Timer konfigurieren, daß Interrupt zu t_0 ausgelöst wird
while (1) {
    Timerinterrupt bestätigen
    sichere Kontext des abgearbeiteten aperiodischen Jobs (falls nötig)
    aktuelle Task T := T[k]
    nächster Tabelleneintrag k = ++k mod(N)
    Timer auf nächsten Taskstart konfigurieren

```
    if (T=='I')
        aperiodischen Job ausführen (falls vorhanden)
    else
        T ausführen
    sleep
}
```

- ungünstig: Aktivierung des Schedulers nach *jedem* Job (hoher Overhead; besonders bei sehr kurzen Jobs)

3.4.3 Struktur zyklischer Schedules

- günstig, die Hyperperiode (Länge H) in gleichlange *Frames* zu strukturieren
- Verringerung der Anzahl der Aktivierungen des Schedulers: nur noch am Anfang jedes Frames (timergesteuert)
- Vereinfachung: Timer besitzt konstante Periode (keine Reprogrammierung mehr nötig)
- innerhalb des Frames werden Jobs einfach nacheinander aufgerufen
- minimiert Anzahl vorzeitiger Beendigungen aperiodischer Tasks

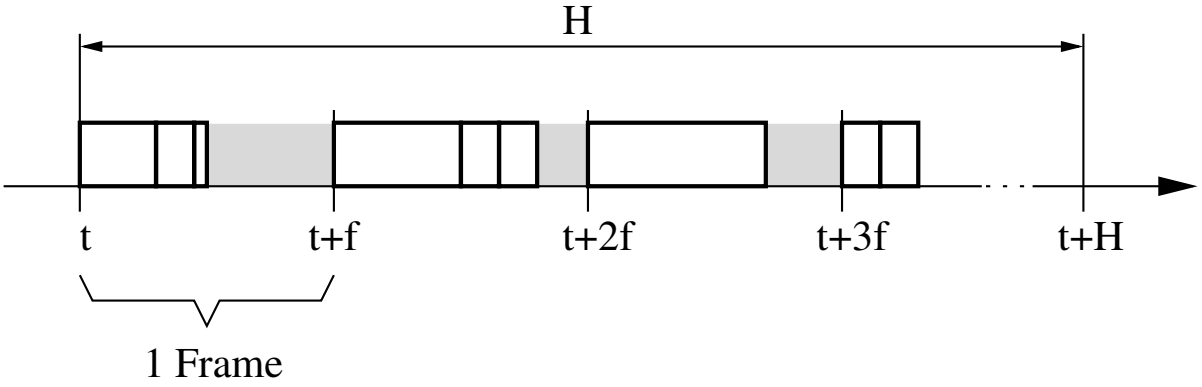


Abbildung 18: Struktur eines zyklischen Schedule

- vgl. [4] und [9, S. 88–92]

Überlegungen zur Framegröße ($\star$)

1.) keine vorzeitige Beendigung (*preemption*) an Framegrenze $\rightsquigarrow$ Framegröße f mindestens so groß wie größte Ausführungszeit eines Jobs

$$f \geq \max(t_{e,i}) \qquad \forall i : 1 \leq i \leq n \tag{1}$$

2.) Framegröße f ganzzahliger Teiler der Hyperperiode H, sonst Schedule u. U. sehr lang (kgV(f, H)).

$$f \mid H \tag{2}$$

3.) Framegröße f so klein, daß zwischen t_r und t_d jedes Jobs mindestens ein kompletter Frame liegt (erleichtert Scheduler die Entscheidung, ob jeder Job seine Deadline einhält; Plazierung des Jobs in diesem Frame garantiert Einhaltung aller Zeitbedingungen $\rightarrow$ erleichtertes Scheduling).

Ein Job J werde in einem Frame k der Größe f zu t_r bereit (mit $t \leq t_r \leq t + f$), besitze eine relative Deadline von t_d und eine Periode von t_p (vgl. Abbildung 19).

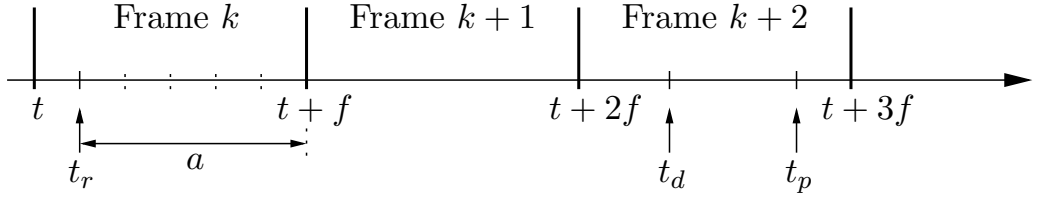


Abbildung 19: Veranschaulichung der dritten Vorschrift zur Framegröße

Damit wenigstens ein vollständiger Frame (nämlich Nr. $k+1$) zwischen t_r und t_d liegt, muß der Abstand a zwischen t_r und der Framegrenze $(t + f)$ plus die Länge des (dazwischenliegenden) Frames $k + 1$ kleiner oder wenigstens gleich der Deadline des Jobs sein:

$$a + f \leq t_d \tag{3}$$

Für $t_p \neq f$ variiert der Abstand a. Er wird genau dann maximal, wenn t_r so nahe wie möglich hinter (aber nicht genau auf, denn dann ist er 0) dem Framestart t liegt. Der minimale Abstand zwischen Framestart t und t_r wird durch das Verhältnis zwischen t_p und f bestimmt. Wenn t_r irgendwann einmal synchron zu einem Framestart lag, dann gilt:

$$t_r - t \geq \mathrm{ggT}(f, t_p)$$

Damit können wir für a formulieren:

$$a \leq f - \mathrm{ggT}(f, t_p)$$

Dies setzen wir in Gleichung 3 ein und erhalten für $t \neq t_r$:

$$f - \mathrm{ggT}(f, t_p) + f \leq t_d \tag{4}$$

Falls Framestart t und Bereitzeit t_r synchron liegen ($t = t_r$), dann ist offensichtlich $f \leq t_d$ hinreichend. Da stets $1 \leq \mathrm{ggT}(f, t_p) \leq f$ gilt, ist diese Bedingung automatisch erfüllt, wenn Gleichung 4 erfüllt ist; dieser Fall muß also nicht gesondert betrachtet werden.

Somit kann die dritte Vorschrift für die Framegröße formuliert werden zu:

$$2f - \mathrm{ggT}\left(f, t_{p,i}\right) \leq t_{d,i} \qquad \forall i \mid 1 \leq i \leq n \tag{5}$$

Die Gleichungen 1, 2 und 5 werden *Frame Size Constraints* genannt ([4]). Ihre Einhaltung gestattet die Verwendung leistungsfähiger Schedulingverfahren und einer verbesserten Cyclic Executive.

Beispiel: 3 periodische Tasks T_i mit

i	$t_{p,i}$	$t_{d,i}$	$t_{e,i}$
1	15	14	1
2	20	26	2
3	22	22	3

Ermitteln Sie H und günstige Framegrößen f !

Lösung:

$H = \text{kgV}(15, 20, 22) = 660$
1.) $f \geq 3$
2.) $f \in \{2, 3, 4, 5, 6, 10, 11, 12, 15, 20, 22, \ldots\}$
3.) $f \leq 6$

$\Rightarrow f \in \{3, 4, 5, 6\}$

3.4.4 Eine bessere Cyclic Executive

Prinzip:

- zyklische Aktivierung an Framegrenze durch Timer
- Schedule besteht aus k Frames (1 Hyperperiode) mit entsprechenden Jobs (jeweils unterschiedliche Anzahl und Länge)
- Nacheinanderausführung aller Jobs des Frames ohne zwischenzeitliche Aktivierung des Schedulers
- Vorteil: geringere Aufrufruffrequenz der CE als in Variante 1 (Overhead ↓)
- etwaiges Überlaufen eines Jobs an einer Framegrenze muß detektiert und behandelt werden

3.4.5 Job Slicing

Beispiel: $T_1(4,1) \quad T_2(5,2,7) \quad T_3(20,5)$.

H = 20, aber 1.) $f \geq 5$ und 3.) $f \leq 4$. Was tun?

Idee: Partitionierung langer Tasks in Subtasks, um Kriterium 1 leichter erfüllen zu können.

Nachteil: mehr Kontextwechsel nötig, da dekomponierte Task vorzeitige Beendigungen benötigt

$\rightsquigarrow$ Dekomposition von $T_3(20,5)$ in $T_{3,1}(20,1)$, $T_{3,2}(20,3)$ und $T_{3,3}(20,1)$; damit:

1.) $f \geq 3$
2.) $f \in \{2,4,5,10,20\}$
3.) $f \leq 4$
$\Rightarrow f = 4$.

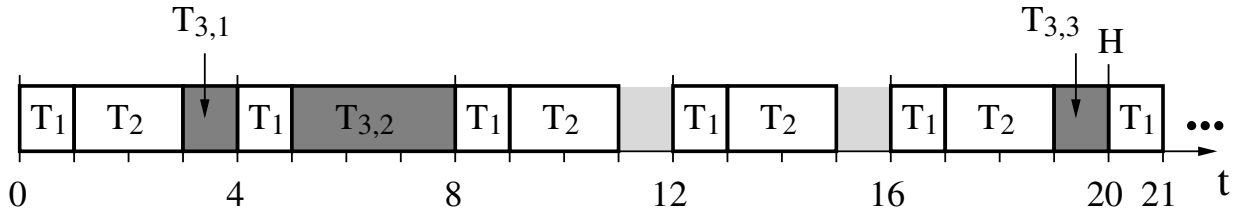


Abbildung 20: Struktur eines Schedule mit Jobslicing

3.4.6 Graphentheorie: Flüsse in Netzwerken

- Eingabe ist Problembeschreibung in Form eines gerichteten Graphen $G = (V, E)$; V: Knotenmenge, E: Kantenmenge
- jeweils ein Knoten für Quelle Q und Senke S
- jede Kante $e \in E$ besitzt eine *Kapazität* $c(e)$ und einen aktuellen *Fluß* $\varphi(e)$.

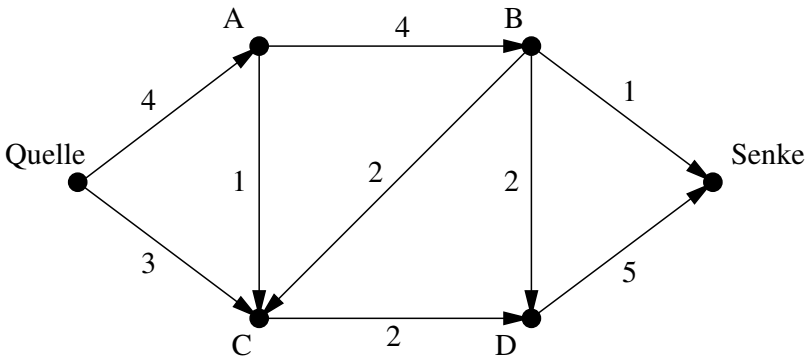


Abbildung 21: Beispiel für Netzwerk (noch ohne Fluß)

- für alle Knoten außer Q und S muß die Summe der Zuflüsse gleich der Summe der Abflüsse sein („Kirchhoff-Regel“)
- Gesucht ist der *maximale Durchfluß* zwischen Quelle und Senke unter Beachtung der Kantenkapazität
- es existieren schnelle Lösungsalgorithmen mit polynomialem Aufwand
- (u.a.) für viele Scheduling-Probleme nutzbar

3.4.7 Algorithmus von Ford und Fulkerson (1956)

- benötigt zu einem Netzwerk $G = (V, E)$ und einem Fluß darin einen „Restgraphen“ $RG = (V, E_\varphi)$ mit der gleichen Knotenmenge und den Kanten:
  - $\forall e \in E$ mit $\varphi(e) < c(e)$ gibt es eine Kante e in E_φ mit $c_\varphi(e) = c(e) - \varphi(e)$
  - $\forall e \in E$ mit $\varphi(e) > 0$ gibt es eine umgekehrte Kante e' mit $\varphi(e') = \varphi(e)$

  Beispiel:

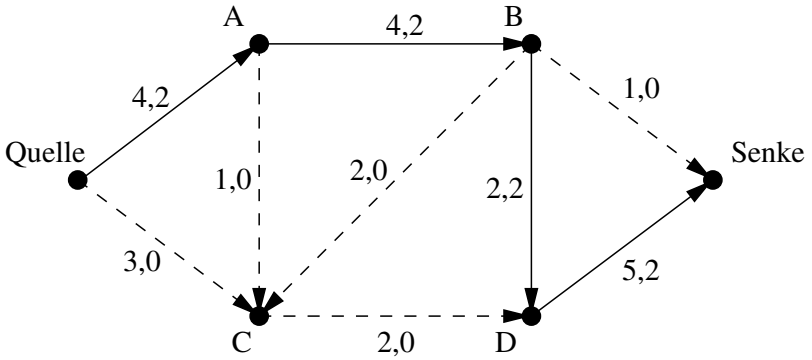


Abbildung 22: Fluß in Netzwerk, Kantenbewertung: $(c(e), \varphi(e))$

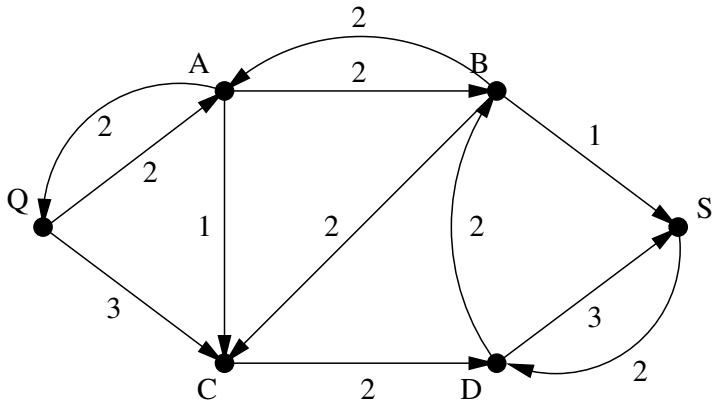


Abbildung 23: zugehöriger Restgraph

- Algorithmus:
  1. START, initialer Fluß $\varphi = 0$.
  2. Markiere Q.
  3. Markiere weitere Knoten nach folgender Regel: „Wenn x markiert ist, und es gibt im RG eine Kante $e = (x, y)$, dann markiere y, wenn dieser noch unmarkiert ist.“
  4. S markiert?
     nein $\rightarrow$ Maximaler Fluß erreicht; END.
  5. Erhöhung des aktuellen Flusses φ durch Adaption des RG: Kanten, die auf dem Weg von Q nach S liegen, seien e_i. Ermittle $\delta = \min(c_\varphi(e_i))$. Adaption des Flusses entlang des

neuen Weges:
- wenn $e_i \in E$, dann $\varphi(e_i) := \varphi(e_i) + \delta$
- wenn $\text{rev}(e_i) \in E$, dann $\varphi(e_i) := \varphi(e_i) - \delta$

6. Aufhebung aller Markierungen, GOTO 2.

- einfachste Lösung des Problems
- viele weitere Algorithmen existieren, die geringere Zeitkomplexität aufweisen, aber schwieriger zu verstehen sind, z. B. :

1969	Edmonds/Karp	$O(n^5)$
1970	Dinic	$O(n^4)$
1988	Goldberg/Tarjan	$O(n^3)$

Beispiel:

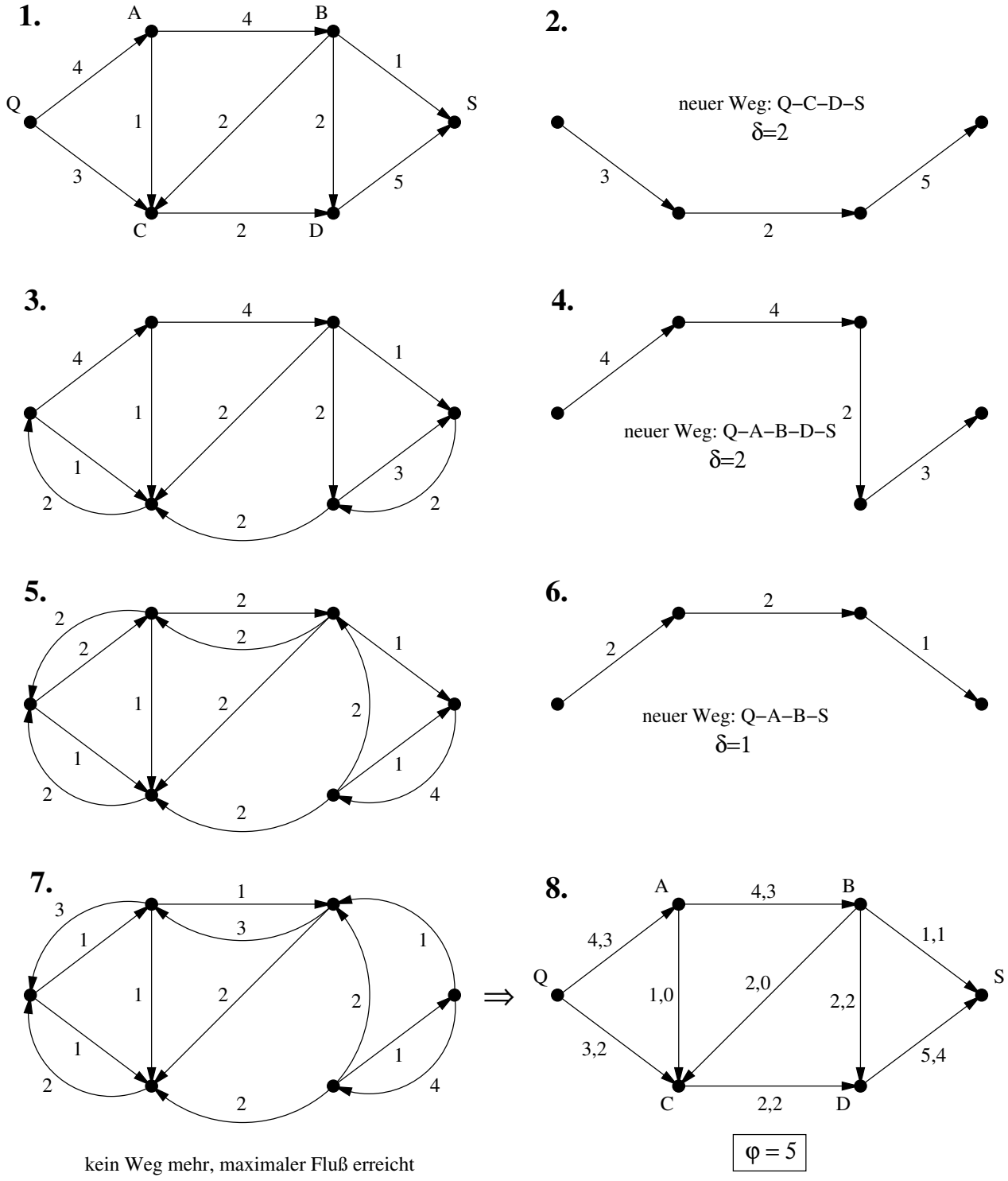


Abbildung 24: Beispiel zum Ford-Fulkerson-Algorithmus

3.4.8 Formulierung des Scheduling-Problems als Fluß im Netzwerk

Annahmen: Preemption immer möglich, Tasks unabhängig, alle möglichen Framegrößen f ermittelt

Idee: beginnend bei Maximum alle f testen, ob Schedule möglich (je kleiner f, desto ungünstiger)

Konstruktion des Netzwerks

1. für jeden Job und jedes Frame einen Knoten, zusätzlich Quelle Q und Senke S
2. von Q zu jedem Job J_i eine Kante mit Kapazität $t_{e,i}$
3. von jedem Frame zu S eine Kante mit Kapazität f (Framegröße)
4. von Job zu Frame eine Kante mit Kapazität f gdw. Job in Frame abgearbeitet werden kann (Timingconstraints!),
   $\rightarrow$ Ein Job J_i kann nur in den Frames geplant werden, deren Startzeit größer oder gleich $t_{\phi,i}$ ist und die nicht später enden als $t_{d,i}$!

Ermittlung des maximalen Flusses von Jobs zu Frames mittels geeignetem Verfahren.

Ist der maximale Fluß durch den Graphen φ größer als $\sum_i t_{e,i}$, dann repräsentiert der Graph einen brauchbaren Schedule.

Anordnung der Jobs in den Frames ist bei Einhaltung von Bedingung 4 beliebig.

Beispiel: 2 periodische Tasks: $T_1(4, 3)$ und $T_2(6, 1.5)$

- $H = 12, f = 4 \rightsquigarrow 3$ Frames in Hyperperiode:
  $f_1[0, 4),\ f_2[4, 8),\ f_3[8, 12)$
- mehrere Aktivierungen der Tasks in H, d.h. individuelle Jobs $J_x(t_\phi, t_d, t_e)$ mit verschiedenen Bereitzeiten und Deadlines

- $T_1 \rightarrow J_{1,1}(0, 4, 3), J_{1,2}(4, 8, 3), J_{1,3}(8, 12, 3)$
- $T_2 \rightarrow J_{2,1}(0, 6, 1.5), J_{2,2}(6, 12, 1.5)$

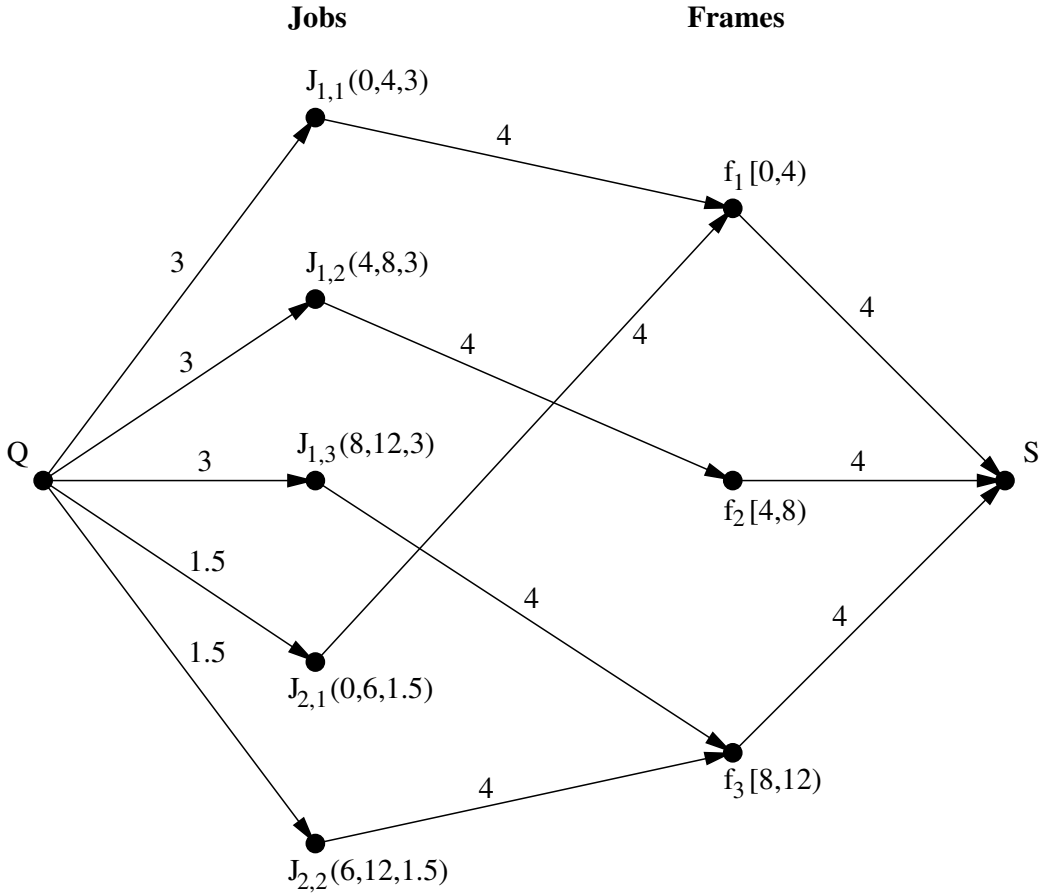


Abbildung 25: zugehöriges Netzwerk

Wie leicht ermittelbar, ist $\varphi = 11 < \sum_i t_{e,i} = 12$.

Ein Schedule ist damit für $f = 4$ nicht möglich.

Verbesserung: Jobslicing, Verkleinerung des Frames auf $f = 2$; damit erhält man

- 6 Frames: $f_1\,[0, 2),\ f_2\,[2, 4),\ f_3\,[4, 6),\ f_4\,[6, 8),\ f_5\,[8, 10),\ f_6\,[10, 12)$
- Dekomposition der Jobs von T_1:

alt	$J_{1,1}(0,4,3)$	$J_{1,2}(4,8,3)$	$J_{1,3}(8,12,3)$
neu	$J_{111}(0,4,2)$	$J_{121}(4,8,2)$	$J_{131}(8,12,2)$
	$J_{112}(2,4,1)$	$J_{122}(6,8,1)$	$J_{132}(10,12,1)$

- $J_{2,1}$ und $J_{2,2}$ bleiben unverändert

Das zugehörige Netzwerk mit einem gültigen Fluß zeigt Abbildung 26. Daraus läßt sich der Schedule in Abbildung 27 ableiten.

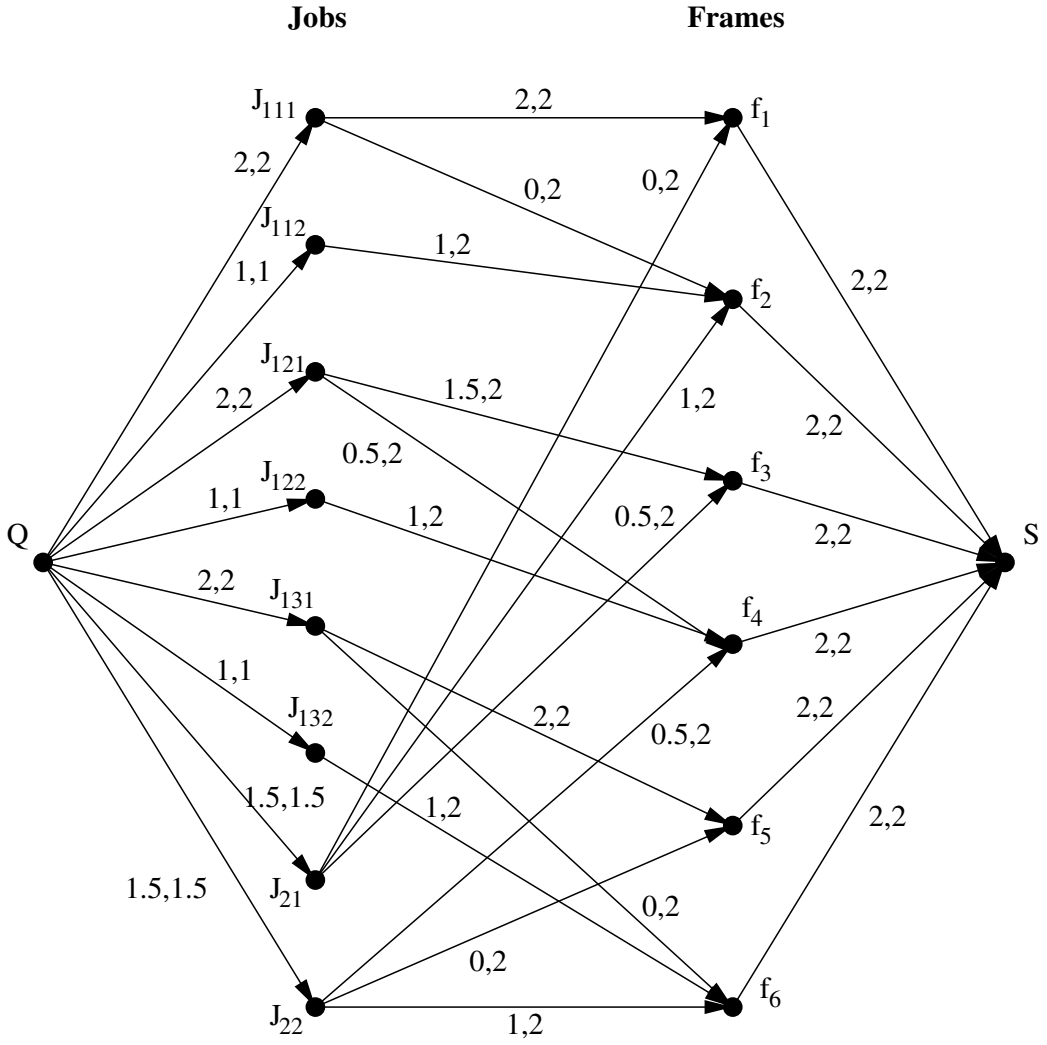


Abbildung 26: Netzwerk mit maximalem Fluß ($\varphi = 12$) für Framegröße $f = 2$

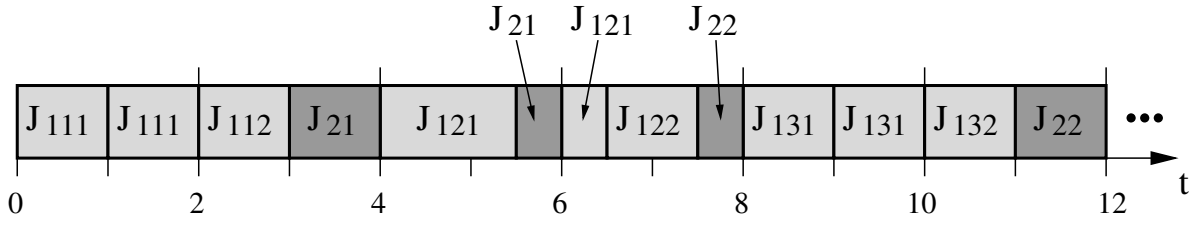


Abbildung 27: Optimaler Schedule für Framegröße $f = 2$

Weitere Verfahren

Xu und Parnas [16] beschreiben ein universelles Verfahren, welches auch Prozeßpräzedenzen einbezieht und auf der Branch-and-Bound-Technik beruht.

Die Frage, ob On-Line- oder Off-Line-Scheduling geeigneter für Echtzeitsysteme ist, wird in der Fachwelt kontrovers diskutiert ([2], [17]), und ist nicht einfach zu beantworten.

Für On-Line-Scheduling sprechen einerseits:

- Off-Line verfahren sind inhärent inflexibel (komplette Neuberechnung nötig bei Modifikationen an der Taskmenge).
- Es existieren on-line-Verfahren mit sehr geringem Schedulingoverhead (RMS, DMS).

Andererseits sind On-Line-Verfahren wie RMS und DMS nicht generell optimal, wie im folgenden Abschnitt gezeigt wird. Das bedeutet, es gibt Taskmengen, die per Off-Line-Verfahren planbar sind, per On-Line-Verfahren jedoch nicht geplant werden können.

3.5 Prioritätsbasiertes Scheduling periodischer Tasks

3.5.1 Einführung

In diesem Abschnitt gehen wir auf on-line-Verfahren ein, die sich Prioritäten bedienen.

Modellannahmen:

- ein Prozessor
- unabhängige Tasks
- keine aperiodischen und sporadischen Tasks

Unterscheidung:

1. feste (statische) Prioritäten: jeder Job einer Task hat die gleiche Priorität
2. variable (dynamische) Prioritäten: Jeder Job erhält eine individuelle Priorität
   (a) feste Job-Prioritäten ("task-level-dynamic, job-level-fixed"): eine einem Job zugeordnete Priorität bleibt konstant
   (b) variable Job-Prioritäten ("job-level-dynamic"): die einem Job zugeordnete Priorität kann sich ändern

Dynamische Verfahren sind flexibler und lasten den Prozessor möglicherweise besser aus, aber das korrekte Systemverhalten statischer Verfahren ist weitaus leichter zu garantieren!

3.5.2 Einplanbarkeitstests

On-Line-Schedulingverfahren treffen zur Laufzeit Entscheidungen der Art

„Gegeben ist die Taskmenge $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ der aktuell abgearbeiteten Tasks. Ist es möglich, eine neue Task T_{n+1} in die Abarbeitung aufzunehmen, so daß alle Tasks ihre Deadlines einhalten?“

„Ja“ $\rightarrow \mathcal{T} := \mathcal{T} \cup T_{n+1}$

„Nein“ $\rightarrow T_{n+1}$ wird abgewiesen

Diese Entscheidung, der *Einplanbarkeitstest* (schedulability test), wird zur Laufzeit gefällt $\Rightarrow$ einfaches Verfahren wichtig.

3.5.3 Ratenmonotones Scheduling (RMS) [8]

Algorithmus: Die Priorität einer Task ergibt sich aus ihrer Periode. Je höher die Rate einer Task (d.h., je kürzer ihre Periode) desto höher ihre Priorität.

Beispiel: 3 (periodische) Tasks $T_i(t_{p,i}, t_{e,i})$
$T_1(4,1) \quad T_2(5,2) \quad T_3(20,5) \quad \Rightarrow \quad \text{Prio}(T_1) > \text{Prio}(T_2) > \text{Prio}(T_3)$

Bereitzeiten:
$t_{r,1} = \{0, 4, 8, 12, 16, 20, \ldots\}$
$t_{r,2} = \{0, 5, 10, 15, 20, \ldots\}$
$t_{r,3} = \{0, 20, 40 \ldots\}$

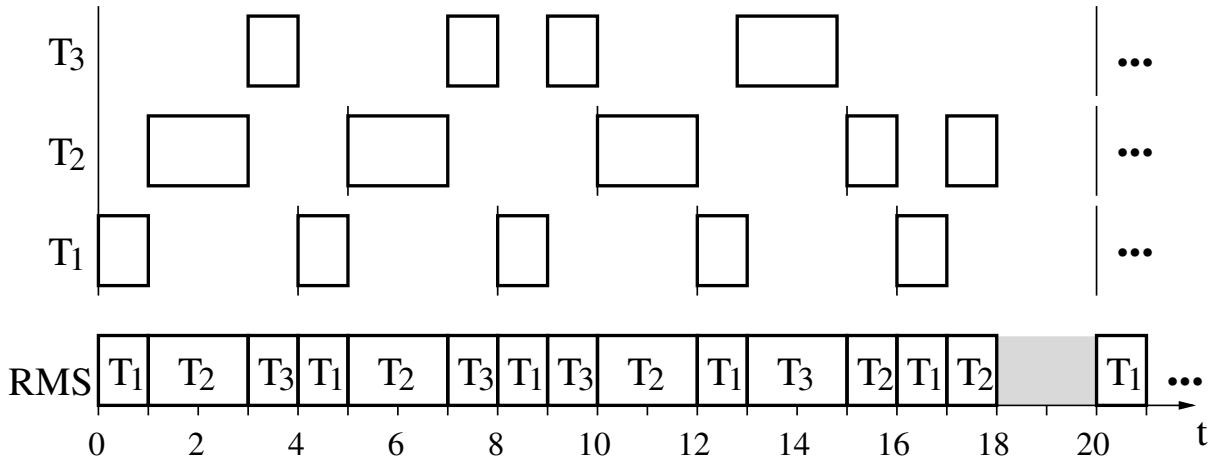


Abbildung 28: Beispiel eines Schedule nach RMS

Vorzeitiger Prozessorentzug zu t = 4, 8, 10 (T_3) und t = 16 (T_2).

Beispiel 2: 2 (periodische) Tasks $T_i(t_{p,i}, t_{e,i})$
$T_1(2, 0.9) \quad T_2(5, 2.3) \quad \Rightarrow \quad \text{Prio}(T_1) > \text{Prio}(T_2)$

Bereitzeiten:
$t_{r,1} = \{0, 2, 4, 6, 8, 10, \ldots\} \qquad t_{r,2} = \{0, 5, 10, 15, \ldots\}$

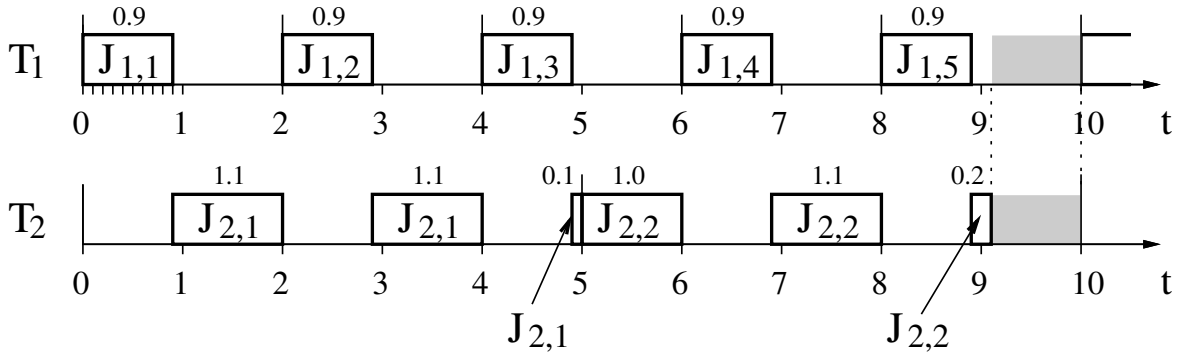


Abbildung 29: RMS-Schedule nach Beispiel 2

T_2 wird „im Hintergrund“ von T_1 ausgeführt.

Auslastung u:

$$u = \sum_{i=1}^{n} \frac{t_{e,i}}{t_{p,i}} = 0.91$$

Weitere Eigenschaften von RMS:

- höchstpriorisierte Task wird nie „gestört“ (unterbrochen, verzögert)
- je größer Periode, desto mehr Unterbrechungen und Verzögerungen (abhängig von Last)
- leicht zu implementieren
- Aufwand zur Laufzeit gering (viel geringer als EDF), da statische Prioritäten
- im Überlastfall gute Vorhersagbarkeit: von n Tasks werden garantiert nur k ($k < n$) beeinflußt, die k niedrigstpriorisierten (k ist bestimmbar) $\rightsquigarrow$ Vorteil gegenüber EDF! (vgl. Abschnitt 3.5.7)
- Einplanbarkeitstest nicht so simpel wie bei EDF (Sonderfall: einfach periodische Taskmengen)

3.5.4 Deadlinemonotones Scheduling (DMS) [3]

Algorithmus: Je näher die (relative) Deadline, desto größer die Priorität.

Beispiel (DMS): 3 (periodische) Tasks $T_i(t_{\phi,i}, t_{p,i}, t_{e,i}, t_{d,i})$

$$\begin{aligned} T_1(50, 50, 25, 100) \qquad & u_1 = 0.5 \\ T_2(0, 62.5, 10, 20) \qquad & u_2 = 0.16 \\ T_3(0, 125, 25, 50) \qquad & u_3 = 0.2 \end{aligned}$$

$\text{Prio}_{\text{DMS}}(T_2) > \text{Prio}_{\text{DMS}}(T_3) > \text{Prio}_{\text{DMS}}(T_1)$

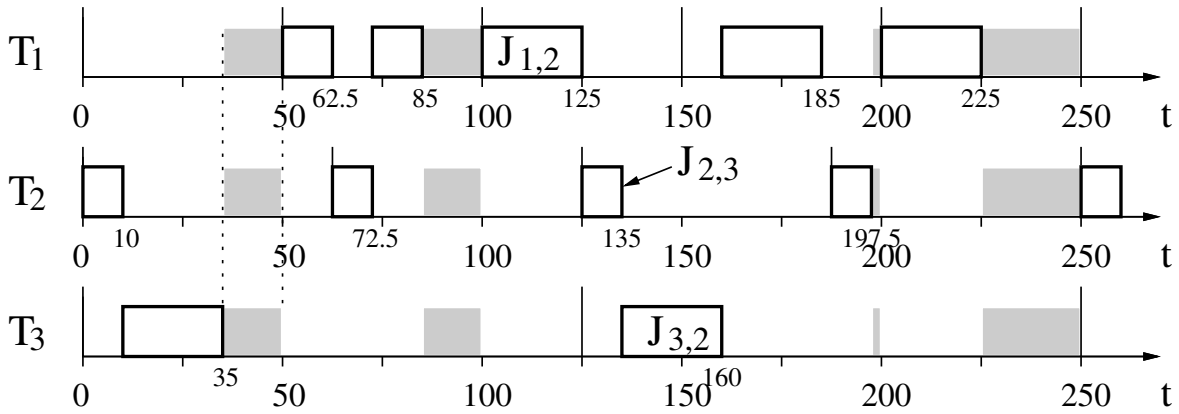


Abbildung 30: Beispiel eines Schedule nach DMS

- $t_d = t_p \rightsquigarrow$ DMS wird zu RMS
- $t_d \neq t_p \rightsquigarrow$ DMS besser als RMS (d.h. generiert u. U. noch einen brauchbaren Schedule, wenn RMS bereits nicht mehr in der Lage dazu ist)

Beispiel: der per RMS generierte Schedule der Taskmenge aus dem vorigen Beispiel:
$T_1(50, 50, 25, 100) \quad T_2(0, 62.5, 10, 20) \quad T_3(0, 125, 25, 50)$
$\text{Prio}_{\text{RMS}}(T_1) > \text{Prio}_{\text{RMS}}(T_2) > \text{Prio}_{\text{RMS}}(T_3)$

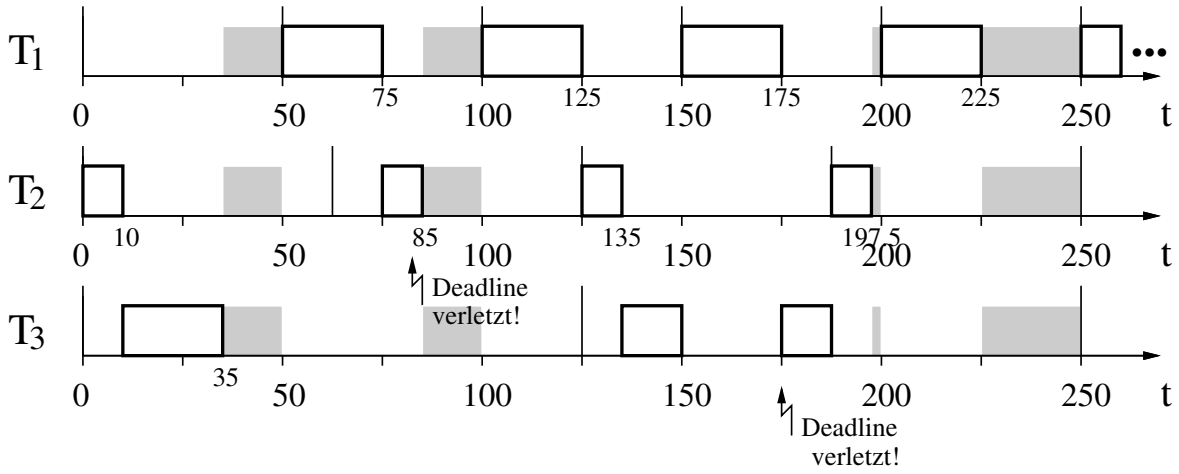


Abbildung 31: ungültiger Schedule nach RMS

T_2 verletzt seine (absolute) Deadline $t_d = 82.5$ (erst zu t=85 beendet) und T_3 verletzt seine (absolute) Deadline $t_d = 175$ (erst zu t = 185 beendet).

Tasks mit kurzer relativer Deadline (hier T_2 mit $t_p = 62.5, t_d = 20$) werden durch RMS benachteiligt.

3.5.5 EDF mit periodischen Tasks

Beispiel: 2 (periodische) Tasks $T_i(t_{p,i}, t_{e,i})$
$T_1(2, 0.9)$ $T_2(5, 2.3)$ (Taskpaar aus Beispiel 2 für RMS)

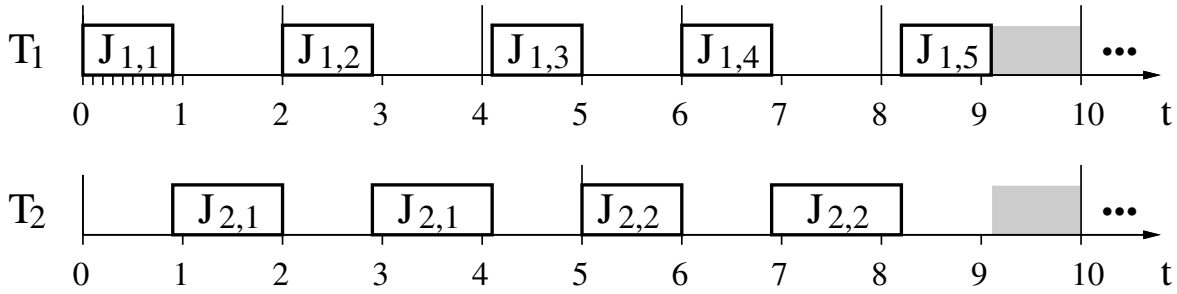


Abbildung 32: EDF-Schedule zweier periodischer Tasks

- t=0: $t_{d,1} = 2,\ t_{d,2} = 5 \Rightarrow J_{1,1}$ abgearbeitet
- t=2: $t_{d,1} = 4,\ t_{d,2} = 5 \Rightarrow J_{1,2}$ abgearbeitet
- t=4: $t_{d,1} = 6,\ t_{d,2} = 5 \Rightarrow J_{2,1}$ abgearbeitet
- d.h., im Intervall [0,4) hat T_1 die höhere Priorität und im Intervall [4,5) hat T_2 die höhere Priorität
- $\Rightarrow$ dynamische Priorität auf Task-Level (Priorität individueller Jobs ist fest)
- $\rightarrow$ höherer Schedulingaufwand als R/DMS, da Neuberechnung der Prioritäten nötig

Einplanbarkeitstest für EDF:

Gegeben: n unabhängige, periodische Tasks, 1 Prozessor

Das System ist nach EDF schedulbar, wenn gilt:

$$\sum_{i=1}^{n} \frac{t_{e,i}}{\min(t_{d,i}, t_{p,i})} \leq 1 \quad . \tag{6}$$

3.5.6 LST mit periodischen Tasks

Beispiel: 3 (periodische) Tasks $T_i(t_{p,i}, t_{e,i})$
$T_1(2, 0.8) \quad T_2(5, 1.5) \quad T_3(5.1, 1.5)$

zugehörige Slack-Zeiten:

t	0	0.8	2	2.8	4	4.3	4.6	5.4
$t_{s,1}$	1.2	–	1.2	–	1.2	0.9	0.6	–
$t_{s,2}$	3.5	2.7	2.7	1.9	0.7	–	–	3.1
$t_{s,3}$	3.6	2.8	1.6	0.8	0.8	0.5	–	3.3

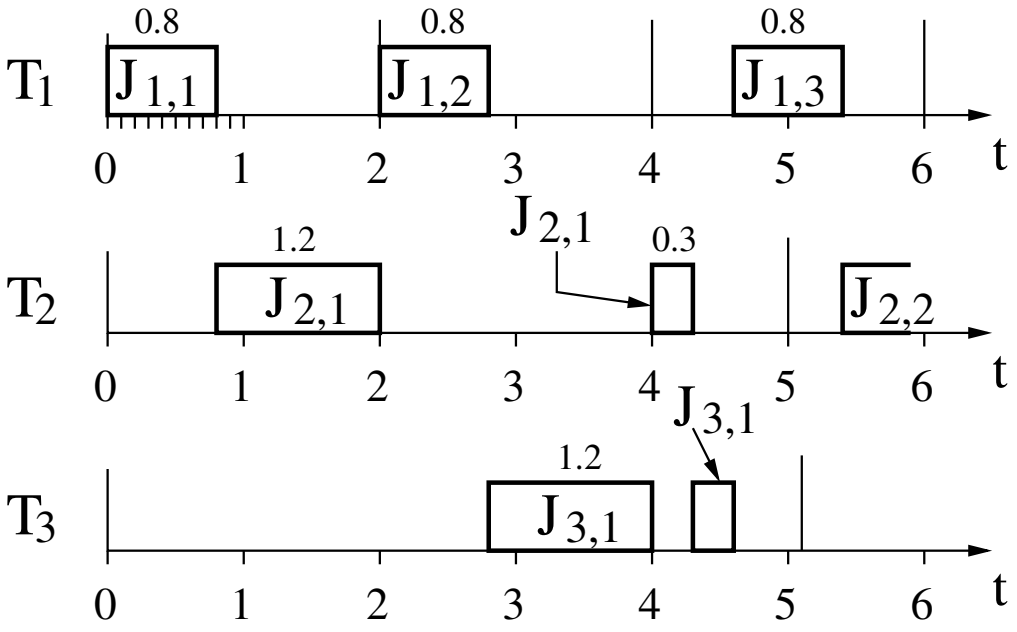


Abbildung 33: LST-Schedule dreier periodischer Tasks

⇒ dynamische Priorität auf Task-Level und auf Job-Level! (z. B. $J_{2,1}$ hat zu $t = 0$ mittlere, zu $t = 2$ niedrigste und zu $t = 4$ höchste Priorität).

- „nonstrict“ LST, da Berechnung der Slackzeiten nur, wenn Zustandswechsel eines Jobs (komplettiert oder bereit)
- eigentliches („strict“) LST müßte jedesmal, wenn eine Änderung einer Slackzeit auftritt, die Prioritäten neu verteilen (Aufwand!)

Situation in letztem Beispiel zu $t = 3.9$:

$t_{s,2} = t_{d,2} - t_{e,2} - t = 5 - 0.3 - 3.9 \quad = 0.8$

$t_{s,3} = t_{d,3} - t_{e,3} - t = 5.1 - 0.4 - 3.9 = 0.8$

$J_{2,1}$ „überholt“ $J_{3,1}$ in der Priorität. Beide müßten von nun nach *round-robin* den Prozessor erhalten (Länge der Zeitscheibe?), da sie sich von nun an ständig in der höchsten Priorität abwechseln.

Aufwandsproblem zur Laufzeit (“runtime overhead”): ständiges Überwachen aller *slack times* nötig. Daher *strict* LST nicht praktikabel.

3.5.7 Periodisches EDF und Überlast

Frage: Wie verhält sich EDF, wenn $u > 1$?

1. Beispiel: 2 (periodische) Tasks $T_i(t_{p,i}, t_{e,i})$

$T_1(2,1) \quad T_2(5,3) \qquad u = \frac{1}{2} + \frac{3}{5} = \frac{11}{10} = 1.1$

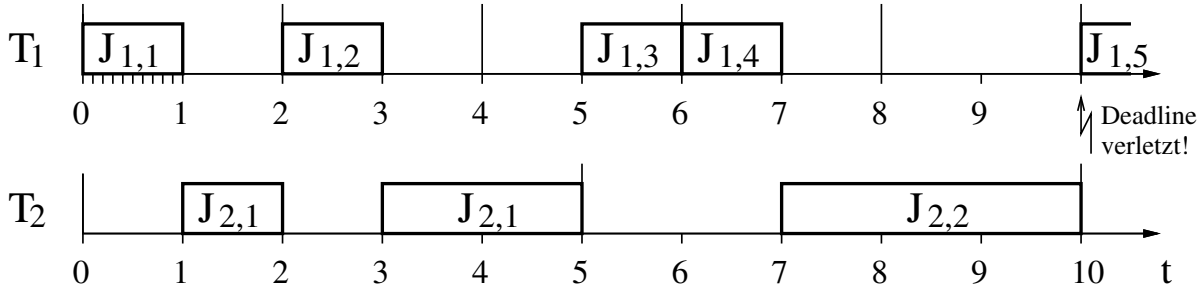


Abbildung 34: EDF-Schedule unter Überlast

Eine Task (T_1) verpaßt ihre Deadline zu $t = 10$.

2. Beispiel: 2 (periodische) Tasks $T_i(t_{p,i}, t_{e,i})$

$T_1(2, 0.8) \quad T_2(5, 3.5) \qquad u = \frac{0.8}{2} + \frac{3.5}{5} = \frac{11}{10} = 1.1$

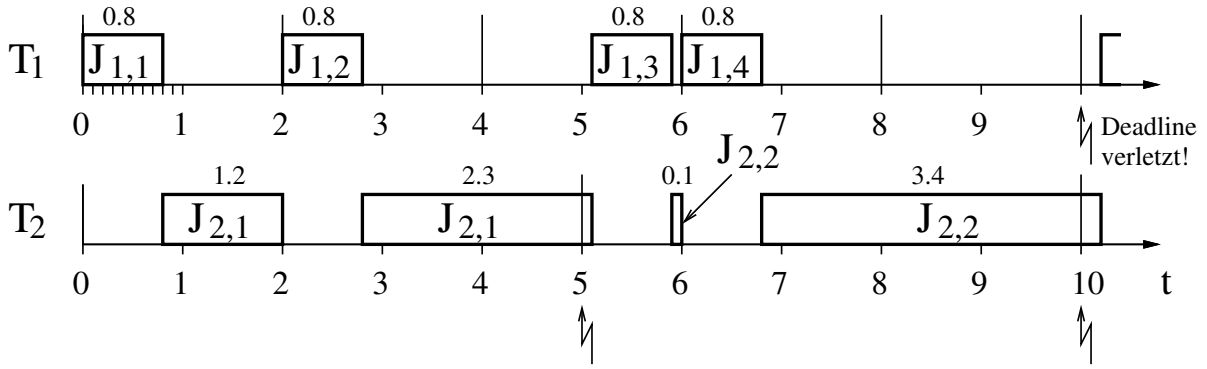


Abbildung 35: EDF-Schedule unter Überlast

Alle Jobs von T_2 sowie $J_{1,5}$ von T_1 verpassen ihre Deadline.

Problem: Trotz gleicher Last unterschiedliches Verhalten. Es gibt keinen (einfachen) Test für EDF, *welche* Jobs bei Überlast ihre Deadline verpassen.

3. Beispiel: 2 (periodische) Tasks $T_i(t_{p,i}, t_{e,i})$

$T_1(2, 0.8) \quad T_2(5, 4) \qquad u = \frac{0.8}{2} + \frac{4}{5} = \frac{12}{10} = 1.2$

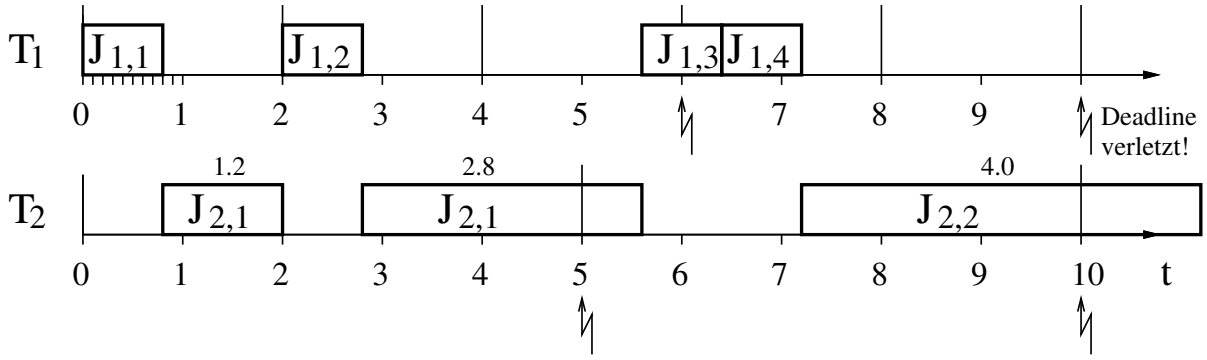


Abbildung 36: EDF-Schedule unter Überlast

Problem: Ein Job, der seine Deadline verpaßt, wird weiter ausgeführt
$\rightsquigarrow$ verursacht u.U. das Überschreiten weiterer Deadlines.

Im Beispiel 3 schafft es nach der ersten Deadline-Verletzung zu $t = 5$ nur noch $J_{1,4}$, pünktlich zu enden. Alle (!) anderen Jobs verletzen ihre Deadline.

Schlußfolgerungen:

- Bei EDF kann bei Überlast keine Aussage getroffen werden, *welcher* Job (bzw. welche Task) seine Deadline noch erreicht.
- Die Verletzung einer Deadline bei EDF kann die Verletzung weiterer Deadlines auch anderer Tasks bewirken (keine "Erholung" von einer temporären Überlastung möglich)
- $\Rightarrow$ EDF nur zum Scheduling von periodischen Tasks geeignet, wenn $u \leq 1$ *garantiert*.

3.5.8 Einplanbarkeitstests für RMS

Definition: Wenn innerhalb einer Taskmenge $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ für jedes Taskpaar $T_i, T_j \in \mathcal{T}$ gilt:

$$t_{p,i} < t_{p,j} \;\Rightarrow t_{p,j} = m \cdot t_{p,i} \qquad m \in \mathbb{N}$$

so nennt man $\mathcal{T}$ *einfach periodisch*.

A) Einplanbarkeitstest für RMS mit einfach periodischen Taskmengen

Gegeben: n unabhängige, periodische Tasks, 1 Prozessor, Taskmenge einfach periodisch

Das System ist nach RMS schedulbar, wenn gilt:

$$u = \sum_{i=1}^{n} \frac{t_{e,i}}{\min(t_{d,i}, t_{p,i})} \leq 1 \quad . \tag{7}$$

B) Einplanbarkeitstest für allgemeines RMS

Voraussetzung: n periodische, unabhängige Tasks mit beliebigen Perioden (*keine* einfach periodische Taskmenge)

Frage: Wann ist die Antwortzeit $t_{resp,i} = t_{c,i} - t_{r,i}$ eines Jobs J_i maximal, wenn nach RMS geschedult wird?

Vermutung: Wenn zu seiner Bereitzeit $t_{r,i}$ alle höherpriorisierten Jobs (d.h. Jobs mit kürzerer Periode) ebenfalls bereit werden. Der Job wird dann maximal verzögert. Dieser spezielle Zeitpunkt wird *kritischer Zeitpunkt* $t_{crit,i}$ des Jobs J_i genannt.

Veranschaulichung:

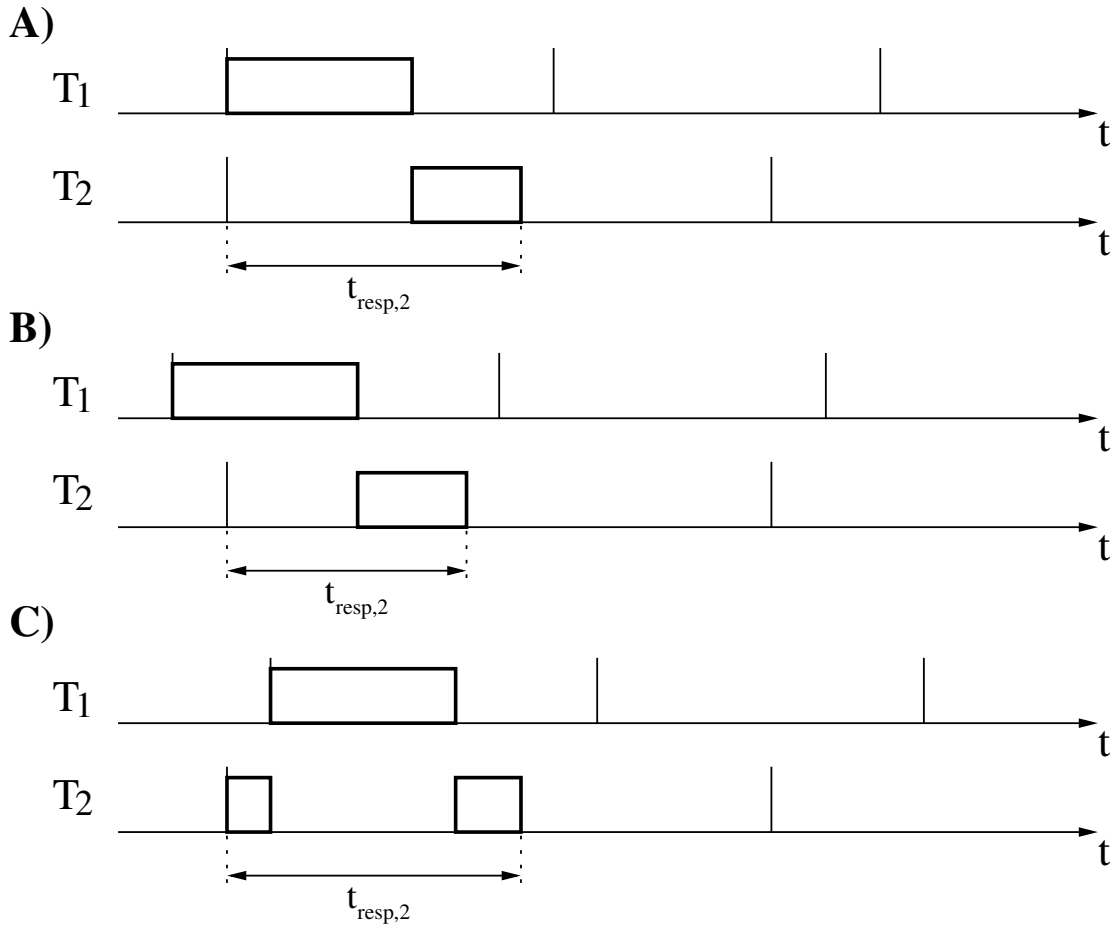


Abbildung 37: Veranschaulichung zum kritischen Zeitpunkt

Jeweils 2 Tasks mit Prio(T_1) > Prio(T_2), 3 mögliche Fälle

A) $t_{r,1} = t_{r,2}$

B) $t_{r,1} < t_{r,2} \rightsquigarrow$ Verzögerung von J_2 geringer

C) $t_{r,1} > t_{r,2} \rightsquigarrow$ Verzögerung von J_2 nicht größer als im Fall A)

$\Rightarrow$ Wird ein Job zu seinem kritischer Zeitpunkt bereit, so wird seine Antwortzeit maximal.

Idee eines Einplanbarkeitstests für RMS:

„Wenn die maximale Antwortzeit einer Task kleiner ist als ihre Periode (= Deadline), dann kann die Task gefahrlos nach RMS eingeplant werden."

$\rightsquigarrow$ Eine Menge an Tasks kann folglich nach RMS geschedult werden, wenn diese Bedingung für alle Tasks aus der Menge gilt.

Iteratives Verfahren zur Bestimmung der maximalen Antwortzeit $t_{maxresp,i}$ für eine Task T_i:

Voraussetzungen:

- Tasks $T_1, \ldots, T_{i-1}$ haben eine höhere Priorität und sind nach dieser absteigend geordnet
- Startwert der Iteration: $t^{(0)} = t_{e,i}$
- iteriert, bis eine der Abbruchbedingungen erfüllt
- Iteration ist monoton wachsend, d.h., $t^{(l+1)} \geq t^{(l)} \quad \forall\, l \in \mathbb{N}$

Iterationsvorschrift:

$$t^{(l+1)} = t_{e,i} + \sum_{k=1}^{i-1} \left\lceil \frac{t^{(l)}}{t_{p,k}} \right\rceil \cdot t_{e,k} \tag{8}$$

Abbruchbedingungen:

1. $t^{(l)} > t_{p,i} \rightsquigarrow$ Task T_i kann nach RMS nicht geschedult werden, da im *worst case* Verzögerung über Deadline hinaus,
2. $t^{(l+1)} = t^{(l)} \rightsquigarrow$ Konvergenz; $t_{maxresp,i} = t^{(l)} \rightsquigarrow$ Task T_i kann nach RMS geschedult werden.

Beispiel: 3 Tasks $T_x = (t_p, t_e)$
$T_1 = (3, 1) \quad T_2 = (5, 1.5) \quad T_3 = (7, 1.25)$

Frage: „Ist T_3 nach RMS einplanbar?“

Iterationsgleichung:

$$t^{(l+1)} = 1.25 + \left\lceil \frac{t^{(l)}}{3} \right\rceil \cdot\ 1 + \left\lceil \frac{t^{(l)}}{5} \right\rceil \cdot\ 1.5$$

Iteration:

$$\begin{aligned} t^{(0)} &= t_{e,3} = 1.25 \\ t^{(1)} &= 1.25 + \left\lceil \frac{1.25}{3} \right\rceil + \left\lceil \frac{1.25}{5} \right\rceil \cdot\ 1.5 = 3.75 \\ t^{(2)} &= 1.25 + \left\lceil \frac{3.75}{3} \right\rceil + \left\lceil \frac{3.75}{5} \right\rceil \cdot\ 1.5 = 4.75 \\ t^{(3)} &= 1.25 + \left\lceil \frac{4.75}{3} \right\rceil + \left\lceil \frac{4.75}{5} \right\rceil \cdot\ 1.5 = 4.75 \end{aligned}$$

$t^{(3)} = t^{(2)} = 4.75 < t_{p,3} = 7 \rightsquigarrow$ T_3 nach RMS schedulbar

Ein weiterer Einplanungstest für RMS

Satz: ([8]) Für eine Menge von n unabhängigen periodischen Tasks kann nach RMS ein brauchbarer Schedule erzeugt werden, wenn die Auslastung u die folgende (hinreichende, nicht notwendige) Bedingung erfüllt:

$$u \leq u_{RM}(n) = n\left(2^{\frac{1}{n}} - 1\right) = n\left(\sqrt[n]{2} - 1\right) \quad . \tag{9}$$

Bemerkungen:

- Funktion fällt monoton mit steigendem n (vgl. Abbildung 38)
- $\lim\limits_{n\to\infty} u_{RM}(n) = \ln(2) \approx 0.693$
- pessimistisches Verfahren: falls Bedingung nicht erfüllt, ist trotzdem noch gültiger Schedule möglich (Voraussetzung: $u < 1$)
- einfacher als vorangegangener Einplanungstest

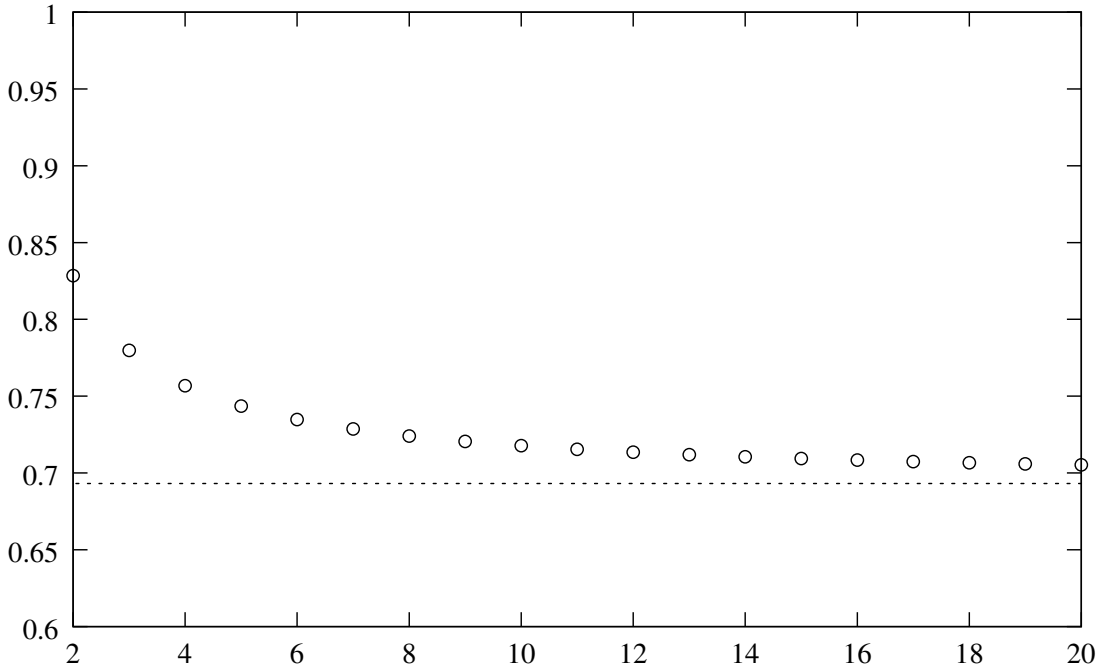


Abbildung 38: Verlauf der Funktion $u_{RM}(n)$

Beispiel: 3 Tasks $T_x = (t_p, t_e)$
$T_1 = (3, 1)$ $T_2 = (5, 1.5)$ $T_3 = (7, 1.25)$

$$\begin{aligned} u &= \frac{1}{3} + \frac{1.5}{5} + \frac{1.25}{7} \approx 0.812 \\ u_{RM}(3) &= 3\left(\sqrt[3]{2} - 1\right) \approx 0.800 \\ u &> u_{RM}(3) \end{aligned}$$

d.h., Einplanungstest 2 weist die Taskmenge als nicht planbar zurück, es existiert jedoch ein Plan (vgl. vorangegangenes Beispiel).

3.5.9 Optimalität von RMS

Satz: Unter den Verfahren mit fester Priorität ist RMS optimal.

Beweis: Bereits bewiesen: Antwortzeit einer Task maximal zu ihrem kritischen Zeitpunkt.

Idee: Wir betrachten zunächst 2 Tasks. Wir zeigen, daß *wenn* es für ein Taskpaar T_1, T_2 einen nicht-RMS-basierten gültigen Schedule gibt, *dann* gibt es für dieses Taskpaar auch einen gültigen Schedule nach RMS.

Es sei $T = \{T_1, T_2\}$ mit $t_{p,1} < t_{p,2}$

Wenn $\text{Prio}(T_1) < \text{Prio}(T_2)$, und es gibt einen gültigen (non-RMS-) Schedule dann gilt offenbar (Abb. 39):

$$\begin{aligned} t_{e,1} + t_{e,2} &\leq t_{p,1} \qquad \text{also} \\ t_{e,2} &\leq t_{p,1} - t_{e,1} \end{aligned} \tag{10}$$

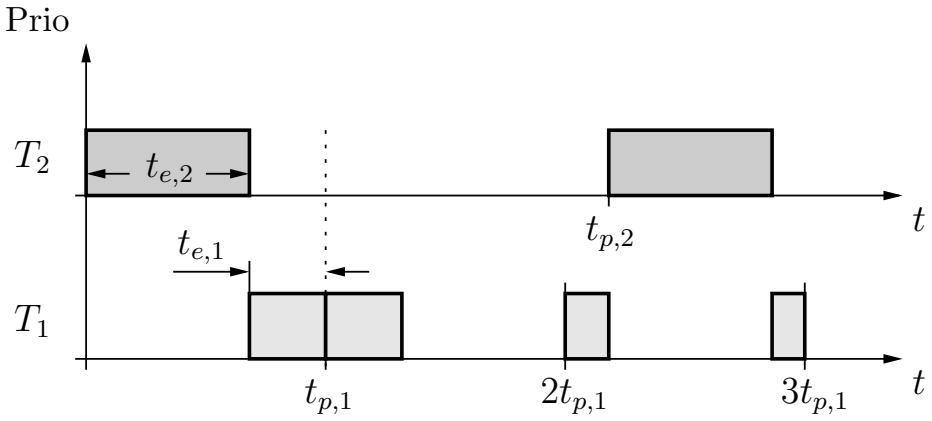


Abbildung 39: Veranschaulichung von Ungleichung 10

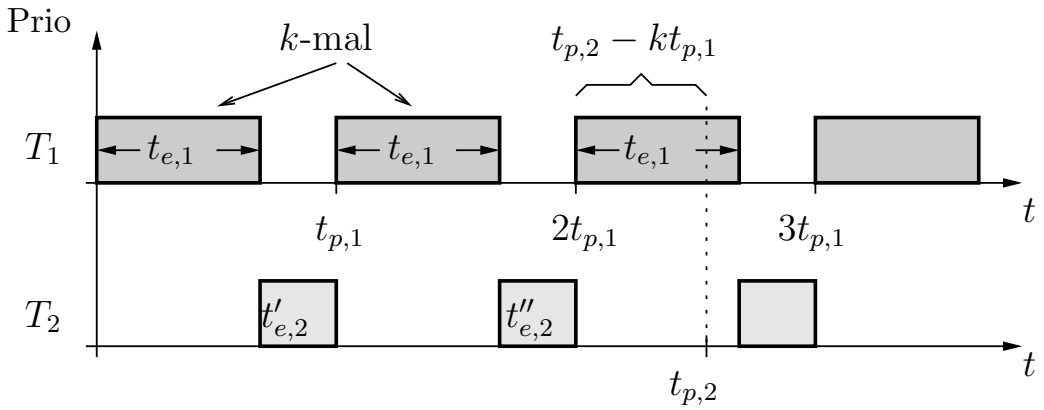
(Im Worst Case muß in der Periode von T_1 noch Platz sein für einen ganzen Job von T_2, da letzterer nach Vereinbarung die höhere Priorität hat.)

Nun gelte $\text{Prio}(T_1) > \text{Prio}(T_2)$.

Dann ist die folgende Bedingung für die Ausführbarkeit des Schedules

hinreichend (vgl. Abbildung 40):

$$kt_{e,1} + (t_{p,2} - kt_{p,1}) + t_{e,2} \leq t_{p,2} \quad \text{mit} \quad k = \left\lfloor \frac{t_{p,2}}{t_{p,1}} \right\rfloor \tag{11}$$



$$t_{e,2} = t'_{e,2} + t''_{e,2}$$

Abbildung 40: Veranschaulichung von Ungleichung 11

Umgestellt nach $t_{e,2}$ ergibt sich:

$$t_{e,2} \leq k\,(t_{p,1} - t_{e,1})$$

Diese Bedingung folgt überraschenderweise unmittelbar aus Gleichung 10:

$$t_{e,2} \leq t_{p,1} - t_{e,1} \leq k\,(t_{p,1} - t_{e,1}) \tag{12}$$

, da $k \geq 1$ und $t_{p,1} \geq t_{e,1}$.

Damit ist gezeigt, daß, wenn zwei Tasks T_1, T_2 nach einem non-RMS-Verfahren planbar sind, diese auch stets nach RMS planbar sind, also ihre Priorität ausgetauscht werden kann.

Bei einer Menge von mehr als 2 Tasks muß diese Vertauschung entsprechend permutiert werden. RMS ist damit unter den Verfahren mit fester Priorität optimal. □

Satz: Ratenmonotones Scheduling ist nicht generell optimal.

Beweis:

Angabe einer Taskmenge, für die mittels RMS kein brauchbarer Schedule erzeugt werden kann und Konstruktion eines brauchbaren Schedules.

2 Tasks $T_x = (t_p, t_e)$ $T_1 = (8, 4)$ $T_2 = (12, 5)$

Beide Einplanungstests schlagen fehl.

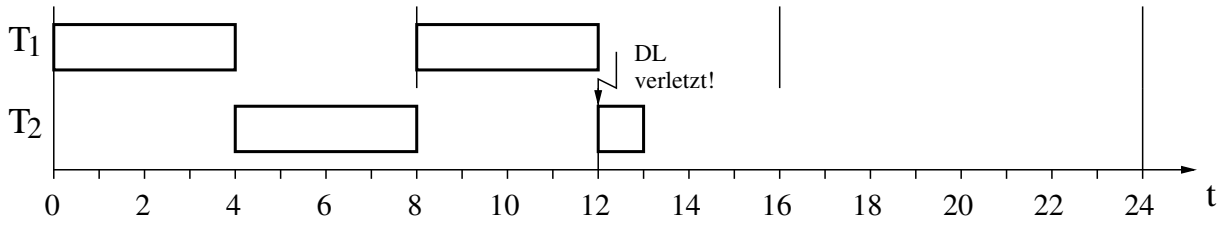


Abbildung 41: (ungültiger) Schedule nach RMS

Durch Verzögerung von T_1 kann jedoch einen brauchbarer Schedule konstruiert werden.

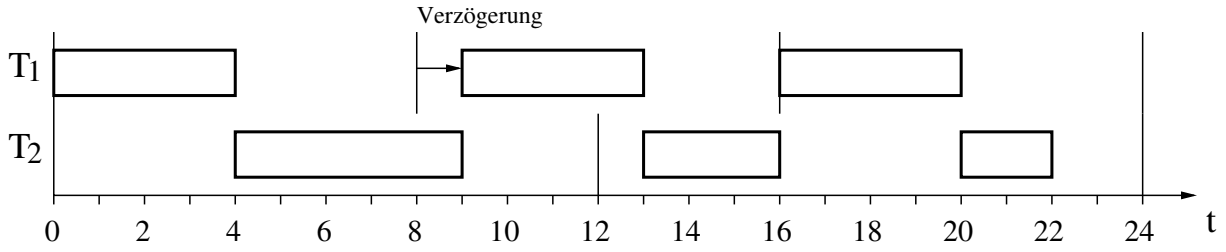


Abbildung 42: (gültiger) Schedule

$\Rightarrow$ RMS ist nicht optimal □

Der konstruierte Schedule kann durch kein prioritätsbasiertes Verfahren mit statischen Prioritäten erzeugt werden!

3.5.10 RMS in Multiprozessorsystemen

Versuch der Verallgemeinerung von RMS auf mehr als einen Prozessor; Voraussetzungen:

- Jede Task kann auf jedem Prozessor abgearbeitet werden (SMP).
- sonst wie 1-Prozessor-RMS (Tasks unabhängig!)

Variante A: Globale Warteschlange:

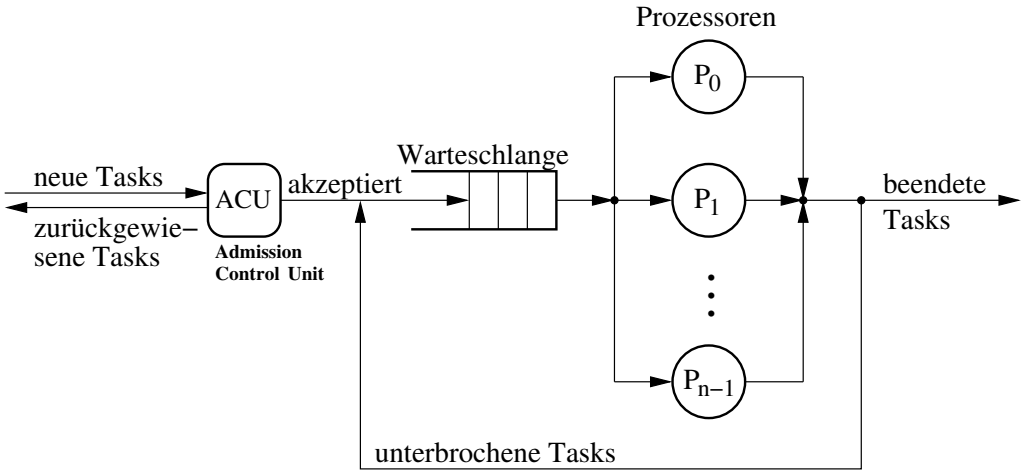


Abbildung 43: MP-RMS, Single Queue

- RMS nicht ohne weiteres verwendbar, da Verteilung auf mehrere Prozessoren nötig
- Komplexität der ACU, Einplanungstest?
- Zugriff auf Warteschlange erfordert Synchronisation (problematisch bei vielen Prozessoren)
- nur mit gemeinsamem Speicher möglich

Variante B: Lokale Warteschlangen:

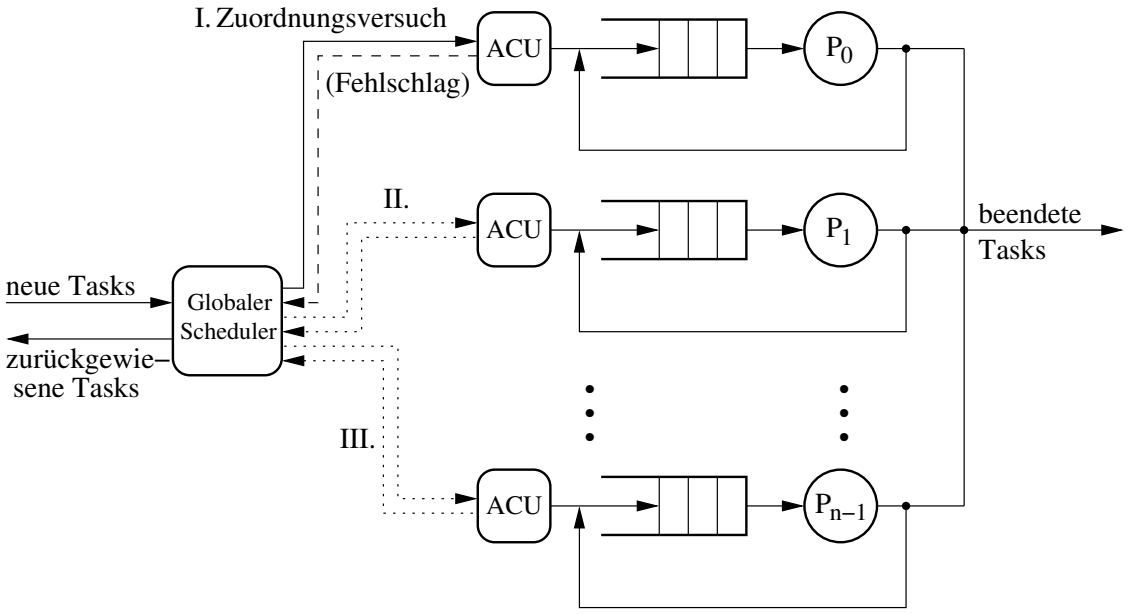


Abbildung 44: MP-RMS, Multiple Queues

Eigenschaften:

- Globaler Scheduler verteilt Tasks
- ACUs führen konventionellen Einplanungstest für RMS aus
- jeder Prozessor: präemptives Scheduling nach RMS
- keine gemeinsam zwischen Prozessoren genutzten Datenstrukturen
- sowohl verteilt als auch mit gemeinsamem Speicher möglich

Verteilungsstrategie des Globalen Schedulers:

- nach der Auslastung: *Highest* oder *Lowest Utilization First*
- nach der Warteschlangenlänge: *Longest* oder *Shortest Queue First*
- "*Adaptive Grouping*": Gruppen von Prozessoren bearbeiten nur Anforderungen mit bestimmten harmonischen Perioden
- Ziel: Maximierung der Auslastung aller Prozessoren

Praktikabilität:

- kein Verfahren bekannt, die Korrektheit für bestimmte Taskmengen zu beweisen $\leadsto$ unbenutzbar für harte Echtzeitsysteme
- bisher nur Simulation mit synthetischen Taskmengen
- nicht betrachtet:
  - sporadische und aperiodische Tasks
  - Overhead für Task-Verteilung

4 Verwaltung von Ressourcen

4.1 Protokolle zur Zugriffssteuerung

Bisherige Voraussetzung: Tasks wechselwirken nicht miteinander. $\rightsquigarrow$ meist unrealistisch!

Beispiele für Wechselwirkungen:

- Kooperation: mehrere Tasks arbeiten gemeinsam an einer Aufgabe, dazu notwendig:
  - Synchronisation: zeitliche Abstimmung bestimmter Teilaktionen
  - Kommunikation: Synchronisation und Datenübertragung
- Konkurrenz um *exklusiv nutzbare* Betriebsmittel, d.h., zu einem beliebigen Zeitpunkt darf maximal 1 Task auf diese Betriebsmittel zugreifen; Zugriffsoperation bildet einen sogenannten *kritischen Abschnitt* (critical section)

Beispiel für exklusiv nutzbare Betriebsmittel: alle HW-Ressourcen, Datenstrukturen des Betriebssystems

Definition. Ein *kritischer Abschnitt* ist eine Folge von Anweisungen, während deren Ausführung die Werte eines exklusiv nutzbaren Betriebsmittels durch andere Tasks potentiell geändert werden können.

Beim Zugriff auf exklusiv nutzbare Betriebsmittel muß ein *wechselseitiger Ausschluß* der Tasks garantiert werden, der Zugriffskonflikte steuert. Eine Realisierungsform ist:

1. Anforderung der Ressource (Operation blockiert, falls Ressource belegt)
2. Nutzung der Ressource

3. Freigabe der Ressource

Schritte 1. und 3. sind Dienste des Betriebssystems oder einer Bibliothek.

4.1.1 Prioritätsumkehrung

Konkurrenz um exklusive Ressourcen kann das Systemverhalten erheblich beeinflussen.

Beispiel:

- 3 Jobs $J_i(t_{r,i}, t_{e,i}, t_{d,i})$ mit
  $J_1 = (6, 5, 14) \quad J_2 = (2, 7, 17) \quad J_3 = (0, 6, 18)$,
- Planung nach EDF,
- Jeder der Jobs benötigt Zugriff auf ein und dieselbe exklusiv nutzbare Ressource R: J_1 zu t=8 für 2 Zeiteinheiten, J_2 zu t=4 für 4 Zeiteinheiten und J_3 zu t=1 für 4 Zeiteinheiten.

Ablauf:

t=0	Nur J_3 ist bereit, wird abgearbeitet
t=1	Zugriff auf $R \rightsquigarrow$ erlaubt, J_3 arbeitet weiter (mit dieser Ressource)
t=2	J_2 wird bereit, unterbricht J_3, da höhere Priorität
t=4	J_2 versucht auf R zuzugreifen; dieses ist jedoch im Besitz von J_3 $\Rightarrow$ J_2 wird blockiert, J_3 weiter abgearbeitet
t=6	J_1 wird bereit, verdrängt J_3
t=8	J_1 versucht, auf R zuzugreifen, welche immer noch durch J_3 gehalten wird, $\rightsquigarrow$ J_1 blockiert, J_3 abgearbeitet
t=9	J_3 gibt R frei, R wird dem höchstpriorisierten anfordernden Job J_1 zugeteilt, dieser wird abgearbeitet
t=11	J_1 gibt R frei, arbeitet weiter, da höchste Priorität
t=12	J_1 beendet $\rightsquigarrow$ J_2 erhält R und arbeitet weiter
t=16	J_2 gibt R frei, arbeitet weiter, da höchste Priorität

t=17 J_2 beendet, J_3 arbeitet weiter
t=18 J_3 beendet.

resultierender Schedule:

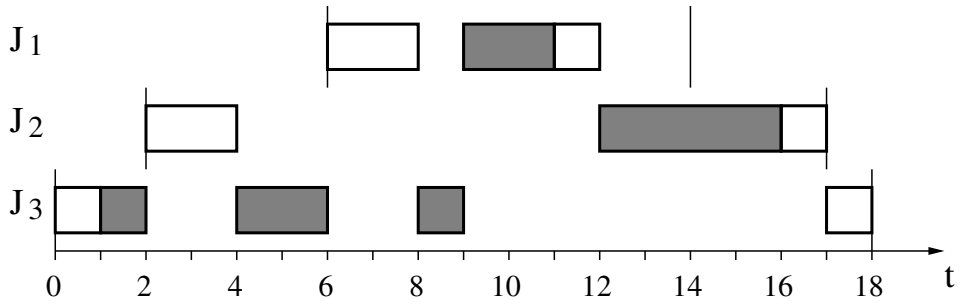


Abbildung 45: Beeinflussung von Jobs durch Ressourcenkonflikte

Eine niedrigpriorisierte Task blockiert eine höherpriorisierte Task. Diese Phänomen wird *Prioritätsumkehrung* (priority inversion) genannt. Im Beispiel: Intervalle [4,6] und [8,9].

Konkurrenz um exklusive Ressourcen führt wiederum zu Anomalien im Zeitverhalten.

Beispiel: Jobmenge aus dem vorangegangenen Beispiel. Modifikation: J_3 benötigt die Ressource R anstatt für 4 nur noch für 2.6 Zeiteinheiten (zu t=1).

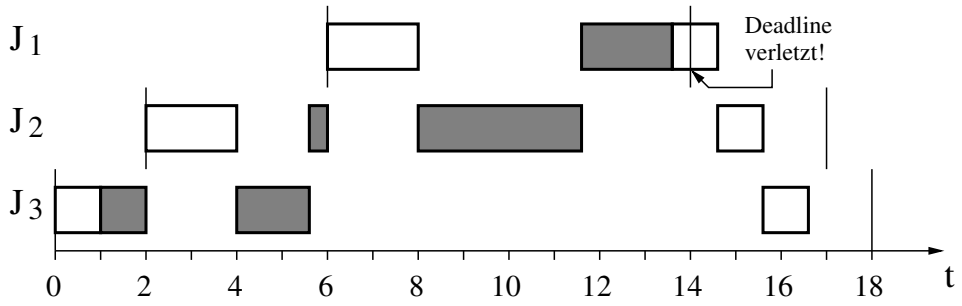


Abbildung 46: Anomalie im Zeitverhalten bei Prioritätsumkehrung

Eine *Verkürzung* des kritischen Abschnitts von J_3 führt zur *Verlängerung* der Komplettierungszeit (und zur Verletzung der

Deadline) von J_1!

Ohne ein vernünftiges Protokoll zur Steuerung kritischer Abschnitte ist die Dauer der Prioritätsumkehrung unbegrenzt:

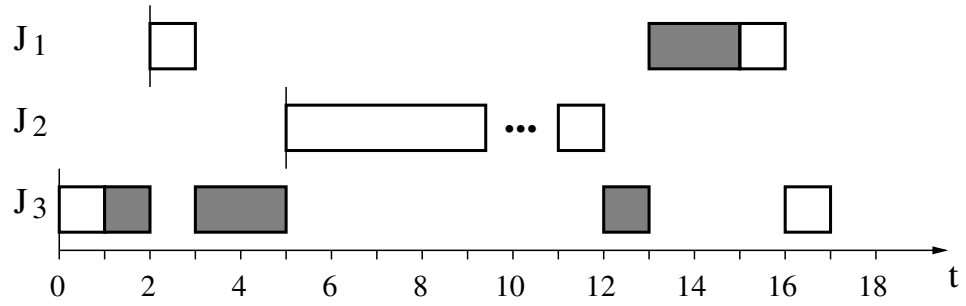


Abbildung 47: Beispiel für ungesteuerte Prioritätsumkehrung

J_2 kann J_3 (und damit J_1) verzögern. Genauer gesagt sind *alle* Jobs, deren Priorität größer ist als die Priorität von J_3 in der Lage, den höchstpriorisierten Job J_1 zu blockieren.

⇒ Strategien zur Zuteilung exklusiv nutzbarer Ressourcen notwendig.

Zielstellung:

- Verhinderung von Prioritätsumkehrung
- Minimierung der Blockierungszeiten
- Verhinderung von Deadlocks

Definition: Ein Job J_i wird *direkt blockiert* durch einen niedriger priorisierten Job J_k, wenn J_i eine exklusive Ressource anfordert (und nicht erhält), die J_k besitzt.

Anmerkungen:

- Verdrängung (preemption) infolge zu niedriger Priorität ist keine Blockierung (→ niedrigstpriorisierter Job kann nicht blockieren)
- bei komplexeren Verfahren weitere Varianten der Blockierung möglich (z. B. indirekte)

- Zeitspanne, die ein Job J_i an einer Ressource R blockiert, ist *Blockierungszeit* $t_{b,i}(R)$

Die Dauer des Zugriffs eines Jobs J_i auf eine Ressource R notieren wir mit $t_{a,i}(R)$.

4.1.2 Nichtunterbrechbare kritische Abschnitte

(non-preemptive critical section – NPCS)

Algorithmus: Ein Job, der auf eine exklusive Ressource zugreift, wird nicht unterbrochen (z.B., indem er temporär die maximale Priorität erhält).

Beispiel: 3 Jobs $J_i(t_{r,i}, t_{e,i}, t_{d,i})$ $\pi_1 > \pi_2 > \pi_3$

$J_1 = (6, 5, 14)$ $J_2 = (2, 7, 17)$ $J_3 = (0, 6, 18)$,

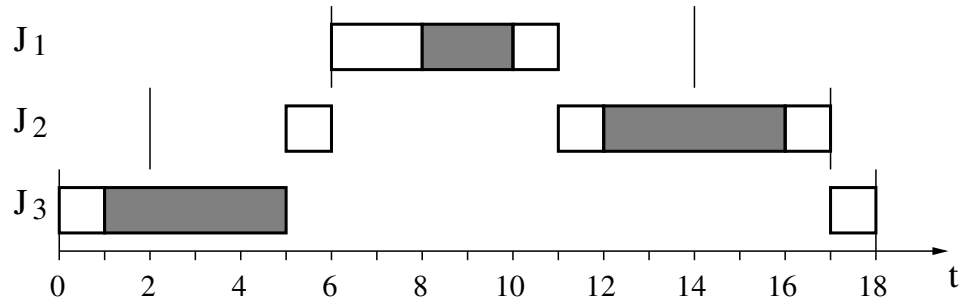


Abbildung 48: Beispiel für NPCS

- keine ungesteuerte Prioritätsumkehrung mehr möglich; maximale Blockierungszeit durch Warten auf Ressource ist *beschränkt*:

$$t_{b,i}(R) \leq \max_{i+1 \leq k \leq n} t_{a,k}(R),$$

  wenn Jobs nach absteigender Priorität geordnet sind.

- Das heißt, die maximal mögliche Blockierungszeit ist die Dauer des größten kritischen Abschnittes, der von Prozessen geringerer Priorität durchlaufen wird.

- für Beispiel: $t_{b,3} = 0$, $t_{b,2} \leq 4$, $t_{b,1} \leq 4$
- leicht zu implementieren
- Nachteil: Jeder Job kann potentiell durch einen niedriger priorisierten Job blockiert werden, selbst wenn gar kein Ressourcenkonflikt zwischen beiden vorliegt!
- sehr grober Eingriff in Priorisierungsschema, hochpriorisierte Jobs benachteiligt

4.1.3 Prioritätsvererbung (Priority Inheritance)

Idee: Wenn ein Job einen höherpriorisierten Job durch den Zugriff auf eine exklusive Ressource blockiert, so wird die Priorität des blockierenden Jobs erhöht (damit dieser nicht durch „mittel“priorisierte Jobs behindert wird).

Algorithmus:

1. Zu $t = t_{r,i}$ erhält ein Job J_i die aktuelle Priorität $\pi_i(t)$, die seiner zugeordneten Priorität entspricht. Diese bleibt konstant, außer wenn er eine höhere Priorität erbt (vgl. 4.)
2. Jeder Job J_i wird präemptiv entsprechend seiner aktuellen Priorität $\pi_i(t)$ geschedult.
3. Wenn ein Job J_i eine exklusive Ressource R anfordert, so ist diese entweder
   - belegt durch einen anderen Job J_k. J_i wird blockiert, J_k erbt die Priorität von J_i, wie unter 4. beschrieben.
   - oder frei. J_i erhält R zugeteilt.
4. Wenn J_i blockiert wird, erbt der blockierende Job J_k die Priorität $\pi_i(t)$. Er verliert diese Priorität wieder (sie sinkt auf das Niveau des Zeitpunktes, zu dem J_k die Ressource R zugeteilt erhielt), wenn er R wieder freigibt.

Beispiel 1: 3 Jobs $J_i(t_{r,i}, t_{e,i}, t_{d,i})$

$J_1 = (6, 5, 14) \quad J_2 = (2, 7, 17) \quad J_3 = (0, 6, 18)$

$\pi_1 > \pi_2 > \pi_3$

resultierender Schedule (aktuelle Prioritäten π_i sind in den Fortschrittsbalken vermerkt):

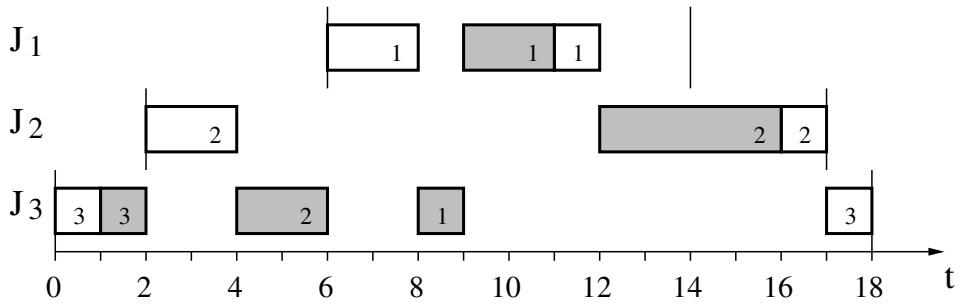


Abbildung 49: Beispiel 1 für Prioritätsvererbung

Beispiel 2: $\pi_1 > \pi_2 > \pi_3 > \pi_4 > \pi_5$
2 exklusive Ressourcen: *schwarz*, *grau*; Notation des k.A.: [Ressource, Zugriffsdauer, Zugriffszeitpunkt innerhalb des Jobs]

J_i	$t_{r,i}$	$t_{e,i}$	krit. Abschnitte
J_1	7	3	[grau, 1, t=1]
J_2	5	3	[schwarz, 1, t=1]
J_3	4	2	–
J_4	2	6	[grau, 4, t=1], [schwarz, 1.4, t=3]
J_5	0	6	[schwarz, 4, t=1]

Tabelle 1: Jobparameter für Beispiel 2

resultierender Schedule :

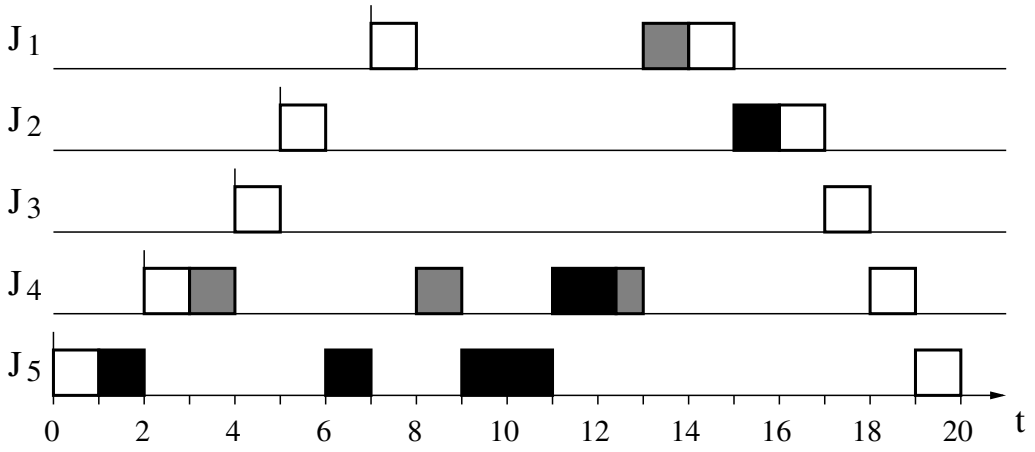


Abbildung 50: Beispiel 2 für Prioritätsvererbung

Eigenschaften der Prioritätsvererbung

- Mechanismus ist transitiv. Beispiel Intervall [9,11] J_5 erbt die Priorität von J_4, welches seine aktuelle Priorität (zu t=8) von J_1 geerbt hat.
- Kein Ausschluß von Deadlocks. (z.B., wenn J_5 zu t=6 die Ressource *grau* fordern würde)
- 2 Arten Blockierung unterscheidbar:
  a) direkte Blockierung zwischen um Ressourcen konkurrierenden Jobs (z.B. J_4 blockiert J_1 zu t=8)
  b) indirekte Blockierung ohne Ressourcenkonflikt als Folge der Prioritätsanhebung (z.B. J_3 wird durch J_5 zu t=6 blockiert)
- Blockierungszeit wird nicht minimiert (ohne Beweis).

Verbesserung: kürzere Blockierungszeiten, Ausschluß von Deadlocks
$\rightsquigarrow$ Verfahren mit Prioritätsgrenze!

4.1.4 Prioritätsgrenze (Basic Priority Ceiling)

Annahmen:

1. Zugeordnete Prioritäten der Jobs sind konstant.
2. Vor dem Systemstart ist bekannt, welche (exklusiven) Ressourcen jeder Job fordert (Zeitpunkte unbekannt).

Parameter:

- $\pi_i(t)$ bezeichnet die aktuelle Priorität des Jobs J_i zum Zeitpunkt t.
- *Prioritätsgrenze* $\Pi(R_i)$ einer Ressource R_i ist Maximum der Prioritäten aller Jobs, die diese Ressource (irgendwann) anfordern
- die aktuelle Prioritätsgrenze $\hat{\Pi}(t)$ (kurz: die *Grenze*) des Systems zu einem Zeitpunkt t ist das Maximum der Prioritätsgrenzen aller Ressourcen, die momentan zugeteilt sind. Keine Ressourcen belegt $\rightsquigarrow \hat{\Pi}(t) = \Omega$, ein nichtexistierender, niedrigster Level.

Algorithmus:

1. Zu $t = t_{r,i}$ erhält ein Job J_i die aktuelle Priorität $\pi_i(t)$, die seiner zugeordneten Priorität entspricht. Diese bleibt konstant, außer wenn er eine höhere Priorität erbt (vgl. 4.)
2. Jeder Job J_i wird präemptiv entsprechend seiner aktuellen Priorität $\pi_i(t)$ geschedult.
3. Wenn ein Job J_i eine exklusive Ressource R anfordert, so ist diese entweder
   - belegt durch einen anderen Job J_k. J_i wird blockiert, J_k erbt die Priorität von J_i, wie unter 4. beschrieben.
   - oder frei. Fallunterscheidung:
     $\pi_i(t) > \hat{\Pi}(t)$, d.h., J_i' aktuelle Priorität $\pi_i(t)$ ist größer als die Grenze des Systems $\hat{\Pi}(t)$, so erhält J_i die Ressource zugewiesen.
     $\pi_i(t) \leq \hat{\Pi}(t)$, dann erhält J_i nur dann die Ressource R zugeteilt, wenn J_i selbst die Ressource besitzt, deren

Prioritätsgrenze gleich $\hat{\Pi}(t)$ ist. Anderenfalls wird J_i blockiert.

4. Wenn J_i blockiert wird, erbt der blockierende Job J_k die Priorität $\pi_i(t)$. Er verliert diese Priorität wieder (sie sinkt auf das Niveau des Zeitpunktes, zu dem J_k die Ressource zugeteilt erhielt), wenn er die Ressource wieder freigibt.

Beispiel (Jobmenge aus Tabelle 1)

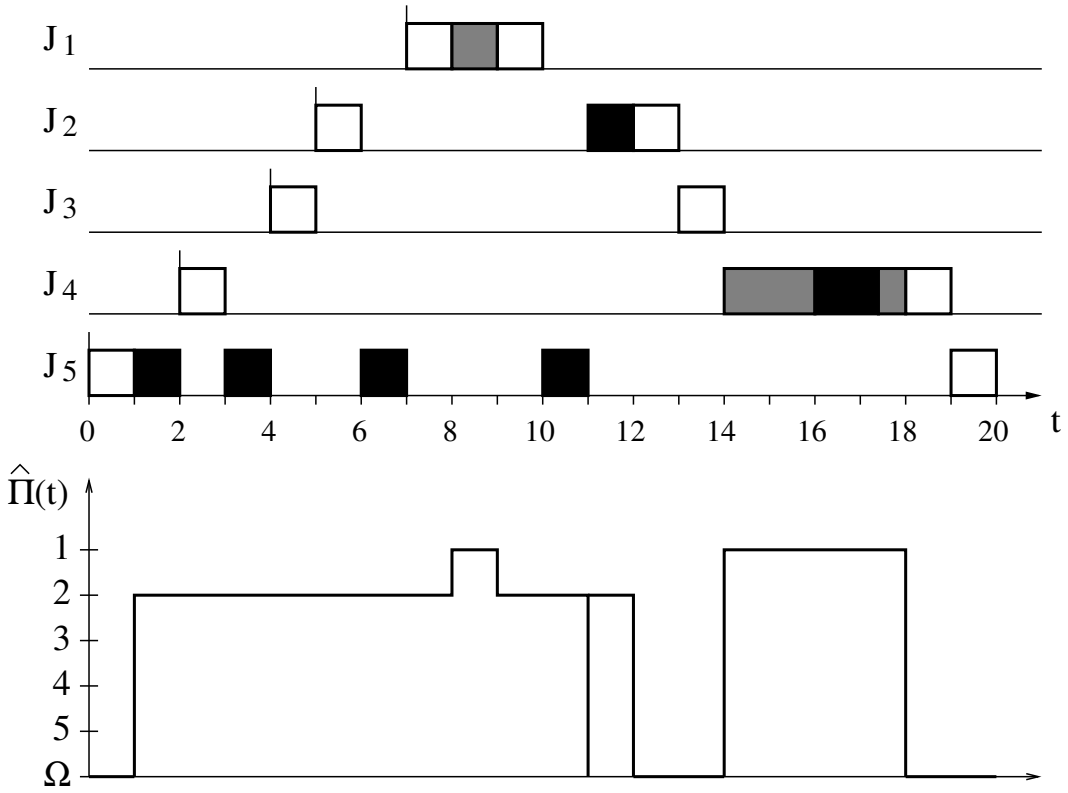


Abbildung 51: Beispiel zum Verfahren Priority Ceiling

Vergleich der Komplettierungszeiten:

	Prioritätsvererbung	Prioritätsgrenze
J_i	$t_{c,i}$	$t_{c,i}$
J_1	15	10
J_2	17	13
J_3	18	14
J_4	19	19
J_5	20	20

Das Verfahren der Prioritätsgrenze reduziert die Komplettierungszeiten hochpriorisierter Jobs auf Kosten niedrigpriorisierter Jobs.

Prioritätsvererbung teilt *immer* Ressource zu, wenn diese frei ist. Prioritätsgrenze nicht! ⇒ Prioritätsvererbung ist verschwenderisch (*greedy*).

3 verschiedene Formen der Blockierung beim Prioritätsgrenzenverfahren:

a) direkte Blockierung: ein Job, der eine Ressource beansprucht, wird durch einen niederprioren Job, der diese Ressource besitzt, blockiert.

b) Blockierung durch Prioritätsvererbung: Ein Job wird blockiert, weil ein (eigentlich) niederpriorer Job momentan eine höhere Priorität „ererbt“ hat.

c) Blockierung durch Priority Ceiling (*avoidance blocking*): Ein Job J_i wird blockiert, da er auf eine freie Ressource R zugreifen möchte. Ein anderer Job besitzt eine (andere) Ressource S, deren Prioritätsgrenze $\Pi(S)$ höher ist als die aktuelle Priorität $\pi_i(t)$ von J_i.

Form c) der Blockierung ist der Preis, der für die Vermeidung von Deadlocks zu entrichten ist!

Satz: Wenn die Ressourcen eines Systems nach dem Verfahren der Prioritätsgrenze zugeteilt werden, so sind Deadlocks ausgeschlossen (ohne Beweis).

Stack-Based Priority Ceiling Protocol

Algorithmus:

1. Die Grenze $\hat{\Pi}(t)$ des Systems wird jedesmal (nach der Regel im *Basic PCP*) aktualisiert, wenn eine Ressource zugeteilt oder freigegeben wird.
2. Nachdem ein Job J_i bereit wurde, wird er geblockt, bis seine zugeordnete Priorität π_i höher ist als $\hat{\Pi}(t)$.
3. Wenn ein Job eine Ressource anfordert, so erhält er diese.

Beispiel: (Jobmenge aus Tabelle 1)

resultierender Schedule: Abb. 52

Beobachtungen:

- einfachere Formulierung des Protokolls,
- Wenn ein Job ausgeführt wird, sind die von ihm benötigten Ressourcen frei und er wird nicht mehr blockiert (höchstens preemptet).
- Es entsteht ein anderer Schedule als bei *Basic PCP*.
- Die beteiligten Jobs könnten sich einen Stack teilen (Name des Verfahrens!).
- weniger Kontextwechsel als bei *Basic PCP*,
- hochpriorisierte Jobs komplettieren eher als bei *Basic PCP* (wenigstens gleichzeitig)

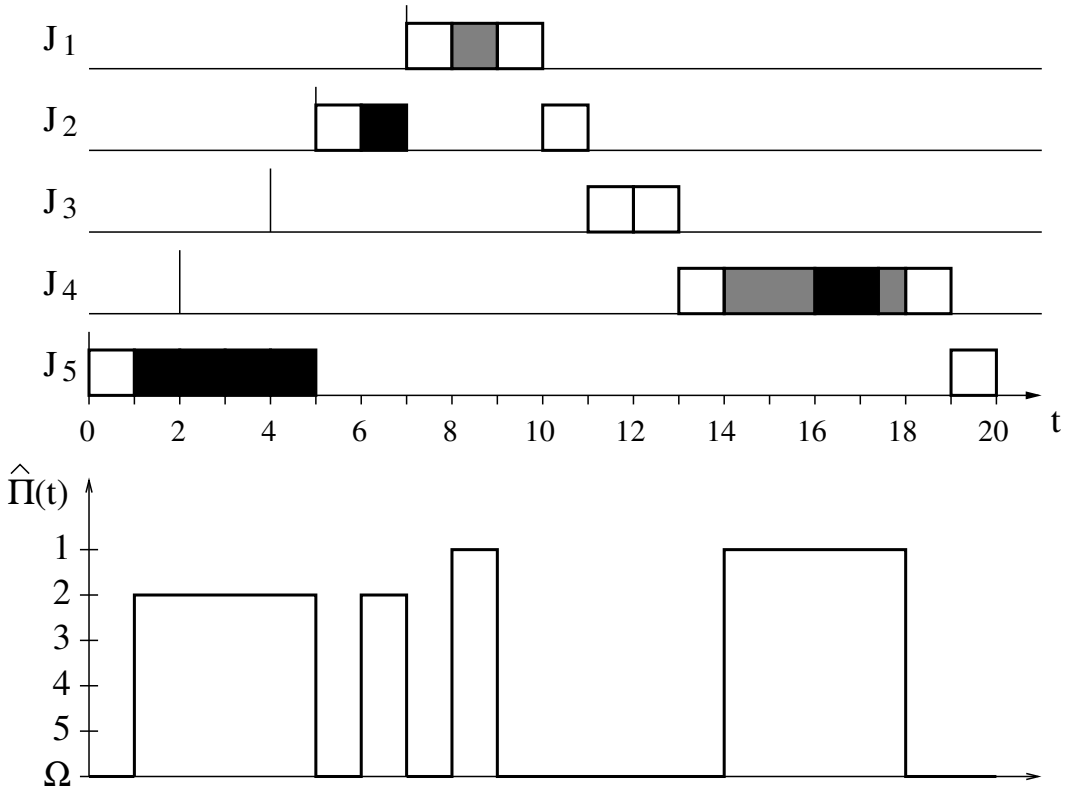


Abbildung 52: Beispiel zum Stack-Based Priority Ceiling

Satz: Im *Worst Case* sind die Blockierungszeiten bei Stack-Based Priority Ceiling identisch zu denen bei Basic Priority Ceiling.

Eine alternative Formulierung des Stack-based Priority-Ceiling-Protokolls ist das Protokoll *Ceiling-Priority*[a]:
Algorithmus Ceiling-Priority:

1. Jobs, die keine Ressource besitzen, werden nach ihrer zugeordneten Priorität geplant.
2. Die Priorität eines Jobs, der Ressource(n) besitzt, ergibt sich aus dem Maximum der Prioritätsgrenzen aller Ressourcen, die er besitzt.
3. Eine Ressourcenanforderung wird stets befriedigt

[a] Um die Verwirrung zu maximieren

Ermittlung der Blockierungszeiten

Definition: Die *Blockierungszeit* $t_{b,i}$ eines Jobs J_i ist die Zeit, die J_i durch Zugriff auf eine exklusive Ressource verzögert wird, wenn diese Ressource durch einen niedriger priorisierten Job gehalten wird.

Die Wartezeit, die infolge Verdrängung durch höher priorisierte Jobs entsteht, ist *keine* Blockierungszeit! (Sie wird auch anders ermittelt, vgl. die entsprechenden Einplanungstests).

Satz: Ein Job kann infolge eines Ressourcenkonfliktes durch einen niederpriorisierten Job maximal für die Dauer eines kritischen Abschnitts blockiert werden, wenn das Verfahren der Prioritätsgrenze zum Einsatz kommt.

Beispiel: Es gelte $\pi_1 > \pi_2 > \pi_3 > \pi_4$, sowie die in Abbildung 53 dargestellten kritischen Abschnitte.

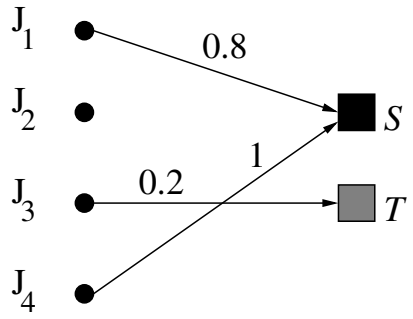


Abbildung 53: Beispiel zur Bestimmung der Blockierungszeiten

- J_1 kann von J_4 maximal für 1 Einheit blockiert werden $\Rightarrow t_{b,1} = 1$
- obwohl J_2 und J_3 die Ressource S nicht benötigen, können sie vererbungsblockiert (Fall b)) werden durch J_4, da J_4 die Priorität von J_1 in dem Fall geerbt hat. $\Rightarrow \quad t_{b,2} = t_{b,3} = 1$ (!)
- J_4 ist schon der niedrigstpriorisierte Job (kann nicht blockiert werden) $\Rightarrow t_{b,4} = 0$

Es gibt 3 verschiedene Blockierungszeiten, die in Betracht gezogen

werden müssen:

a) direkte Blockierung infolge der Konkurrenz um eine exklusive Ressource

b) indirekte Blockierung durch Prioritätsvererbung (ein eigentlich niedrigpriorisierter Job erbt eine hohe Priorität und verdrängt damit einen anderen Job, der eigentlich eine höhere Priorität besitzt)

c) Blockierung durch *priority ceiling*: Ein Job fordert eine freie Ressource an, die er aufgrund der aktuellen Prioritätsgrenze des Systems nicht erhält.

Die systematische Ermittlung der Blockierungszeiten bei Priority Ceiling wird im Rahmen der Hausaufgaben erläutert (Literatur: [9, S.297–298]).

Da jeder Job maximal für die Dauer *eines* kritischen Abschnittes blockiert werden kann, erhöht sich die maximale Antwortzeit jedes Jobs um die ermittelte Blockierungszeit.

Für eine höchstpriorisierte Task T_j ergibt sich für RMS:

$$t_{maxresp,j} = t_{e,j} + t_{b,j}$$

Sind alle Tasks nach absteigender Priorität geordnet (höchste Priorität bei T_1), so ergibt sich für eine beliebige Task T_i:

$$t_{maxresp,i} = t_{e,i} + t_{b,i} + \sum_{k=1}^{i-1} t_{maxresp,k}$$

Damit kann der Einplanungstest für RMS entsprechend erweitert werden:

$$t^{(l+1)} = t_{e,i} + t_{b,i} + \sum_{k=1}^{i-1} \left\lceil \frac{t^{(l)}}{t_{p,k}} \right\rceil \cdot t_{e,k} \qquad (13)$$

4.1.5 Nichtblockierender Zugriff

Idee: Operation `try(Ressource)` versucht, Zugriff auf Ressource zu erlangen. Kehrt immer zurück, erfolgreich oder erfolglos.

⇒ kein Deadlock möglich (aber Lifelock!)

Nachteile:

- nicht mehr zur Synchronisation nutzbar
- Alternative zum sofortigen Eintritt in kritischen Abschnitt notwendig
- komplexeres Programmverhalten

Bsp: Flag `IPC_NOWAIT` (System-V-Kommunikation), `sem_trywait()` (POSIX-Semaphore)

4.2 Verwaltung von Massenspeichern

4.2.1 Aufbau und Funktionsweise

Literatur: [5, 13]

Prinzipaufbau einer Festplatte:

- Stapel von rotierenden Magnetplatten, ca. 5400 – 10100 $U \cdot min^{-1}$
- konstante Rotationsgeschwindigkeit (CAV *Constant Angular Velocity*)
- konzentrische Spuren (*Tracks*)
- übereinanderliegende Spuren bilden einen sog. *Zylinder*
- 1 Schreib-Lesekopf pro Plattenoberfläche, lineare Bewegung
- kleinste ansprechbare Einheit: physischer Sektor (512 Byte)

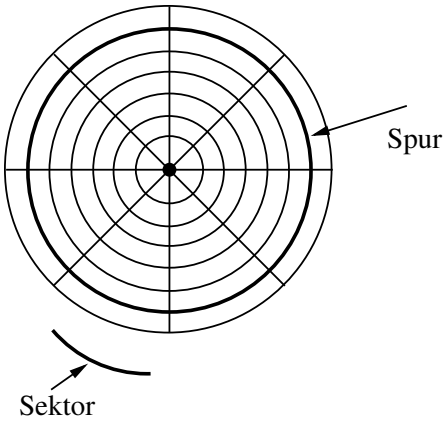


Abbildung 54: Festplatte in der Draufsicht (schematisch)

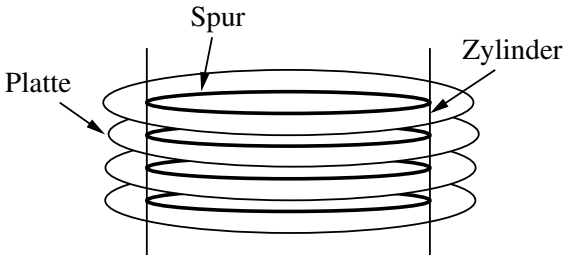


Abbildung 55: Seitenansicht einer Festplatte (schematisch)

Festplatten werden als eindimensionales Feld *logischer Sektoren* betrachtet. **Abbildung** physischer auf logische Sektoren:

- logischer Sektor 0 ist der erste physische Sektor des äußersten Zylinders auf der obersten Platte
- es folgen reihum die restlichen physischen Sektoren der Spur, danach die weiteren Spuren dieses Zylinders, danach alle weiteren Zylinder von außen nach innen
- d.h., der letzte logische Sektor liegt im innersten Zylinder auf der untersten Platte

Ablauf eines Zugriffs:

1. Konvertierung der logischen Adresse des Datums in

Sektornummer (Dateisystem und Festplattentreiber)

2. Positionierung des Schreib/Lesekopfes über richtiger Spur (Positionierzeit, *seek time* t_{seek})
3. Warten bis Anfang des gewünschten Sektors unter Kopf (Rotationsverzögerung, *rotational latency* t_{rot}), im Mittel $\frac{1}{2}$ Umdrehung
4. Einlesen des Sektors in den Pufferspeicher der Festplatte
5. Übertragung der Daten via Bussystem (SCSI, PCI) in den Hauptspeicher

Zugriffszeit durch mechanischen Teil dominiert.

Beispiel: Maxtor DiamondMax VL40, AMD Athlon 1 GHz:
Festplatte rotiert mit 5400 $U \cdot \text{min}^{-1} \Rightarrow \frac{1}{2}$ U in 5.5 ms

Prozessor: $\approx$ 5.5 Millionen Befehle (1 pro Taktzyklus)

Optimierung des Festplattenzugriffs wesentlich!

Einflußmöglichkeiten:

- Datenlayout entsprechend Zugriffsmuster (Minimierung der Rotationsverzögerung)
- Planung der Plattenaufträge (Minimierung der Positionierungszeit)

4.2.2 Scheduling von Zugriffen

Idee: Optimierung eines Zielparameters (z. B. Durchsatz) durch Umsortierung der auszuführenden Lese- bzw. Schreibvorgänge

Anmerkungen:

- nur sinnvoll, wenn stets genügend Aufträge existieren
  Voraussetzung: Lineare Zuordnung zwischen Adresse (logische Blocknummer) und tatsächlicher Position der Daten auf Platte

- Optimierung in 2 Stufen
  - Optimierung der Kopfbewegung (Minimierung t_{seek})
  - zusätzliche Einbeziehung der Rotationslatenz (Minimierung $t_{seek} + t_{rot}$)

a) Ziel klassisches Festplatten-Scheduling

- maximaler Durchsatz,
- minimale mittlere Antwortzeit,
- Fairneß.

b) Ziel Echtzeit-Festplatten-Scheduling

- Einhaltung der Deadlines aller Aufträge,
- Minimierung des Pufferspeichers,
- Maximierung der Anzahl gleichzeitiger Datenströme (CM - *continuous media*),
- Kompromiß zwischen Zeitgarantie und Effizienz ($\rightarrow$ Einsatz in *weichen* Echtzeitsystemen)

First Come First Serve (FCFS, FIFO)

Algorithmus: Die Aufträge werden in der Reihenfolge ihres Eintreffens ausgeführt.

- fair, Verhungern von Aufträgen unmöglich,
- keinerlei Optimierung $\rightarrow$ ineffizient.

Beispiel (40 Sektoren):

Reihenfolge des Eintreffens: 11, 1, 36, 16, 9, 12, 34 Reihenfolge der Bearbeitung: 11, 1, 36, 16, 9, 12, 34 mittlere Weglänge = $\frac{10+35+20+7+3+22}{6} = \frac{97}{6} = 16.2$

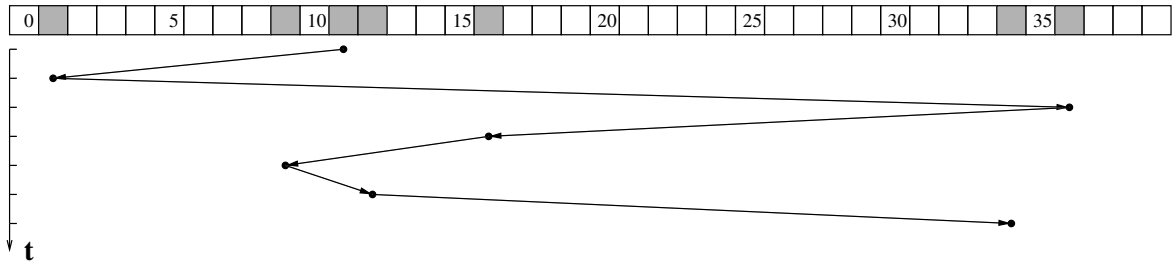


Abbildung 56: resultierende Kopfbewegung bei FCFS

Shortest Seek Time First (SSTF)

Algorithmus: Aus allen existierenden Aufträgen wird derjenige zur Bearbeitung ausgewählt, dessen Kopfdistanz zum letzten bearbeiteten Auftrag am kleinsten ist.

- kleine mittlere Antwortzeit
- hoher Durchsatz
- unfair, „außen“ liegende Aufträge stark benachteiligt
- Verhungern möglich: durch kontinuierliches Eintreffen neuer „naher“ Forderungen werden entfernte Forderungen blockiert
- Distanz muß für alle Aufträge bei jedem Auftrag errechnet werden (Aufwand!)

Beispiel:

Reihenfolge des Eintreffens: 11, 1, 36, 16, 9, 12, 34 Reihenfolge der Bearbeitung: 11, 12, 9, 16, 1, 34, 36 mittlere Weglänge = $\frac{1+3+7+15+33+2}{6} = \frac{61}{6} = 10.2$

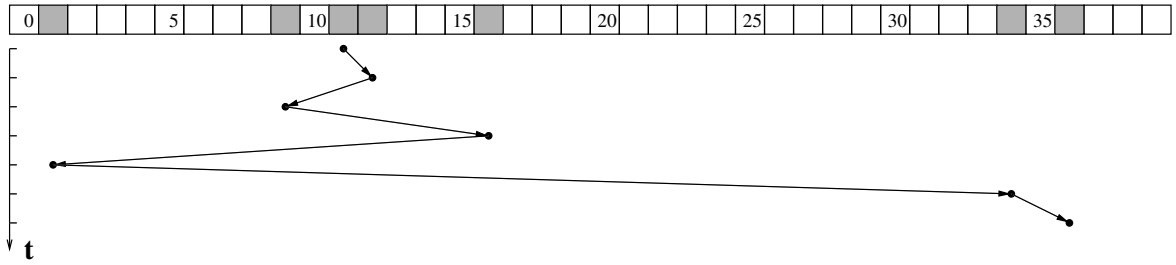


Abbildung 57: resultierende Kopfbewegung bei SSTF

Elevator-Algorithmus (SCAN)

Algorithmus: Der Kopf wird unidirektional von der äußersten zur innersten Spur bewegt. Alle Aufträge, denen er dabei „begegnet“, werden abgearbeitet. Wenn keine Aufträge in die Bewegungsrichtung mehr anstehen erfolgt Umkehrung der Richtung, und der Vorgang wiederholt sich mit den inzwischen eingetroffenen neuen Aufträgen.

- etwas schlechtere Leistung als SSTF, da schlechtere Ausnutzung der Lokalität
- Verhungern unmöglich
- maximale Wartezeit: 2 komplette Durchläufe (Scans)
- Dauer eines Scans abhängig von Anzahl Aufträge
- Benachteiligung der äußeren Sektoren

Beispiel (identische Voraussetzungen):

Reihenfolge des Eintreffens: 11, 1, 36, 16, 9, 12, 34 Reihenfolge der Bearbeitung: 11, 12, 16, 34, 36 (U), 9, 1
(initiale Richtung: aufwärts)
mittlere Weglänge $= \frac{1+4+18+2+27+8}{6} = \frac{60}{6} = 10.0$

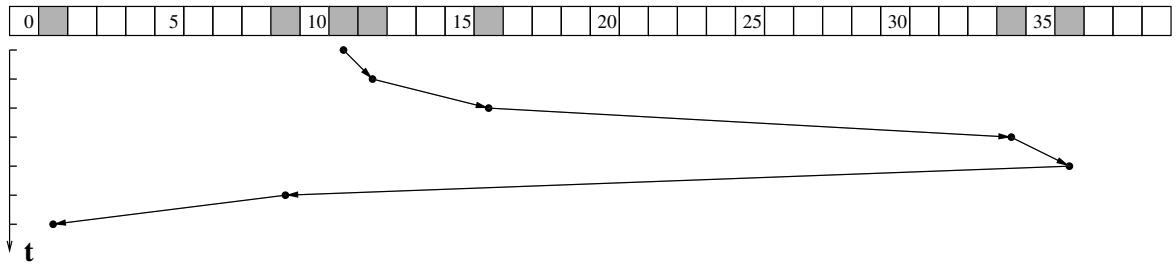


Abbildung 58: resultierende Kopfbewegung bei SCAN

Circular Scan (C-SCAN)

- unidirektionale Bewegung des Kopfes bis keine Aufträge in dieser Richtung mehr anstehen, dann Zurückfahren des Kopfes an Anfang und erneuter Durchlauf in gleicher Richtung
- (geringer) Zeitverlust durch Rücklauf des Kopfes
- Benachteiligung der äußeren Sektoren eliminiert
- geringere Varianz der Antwortzeit als bei SCAN

Beispiel (identische Voraussetzungen):

Reihenfolge des Eintreffens: 11, 1, 36, 16, 9, 12, 34 Reihenfolge der Bearbeitung: 11, 12, 16, 34, 36 (R), 1, 9
(Richtung: aufwärts)
mittlere Weglänge = $\frac{1+4+18+2+35+8}{6} = \frac{68}{6} = 11.3$

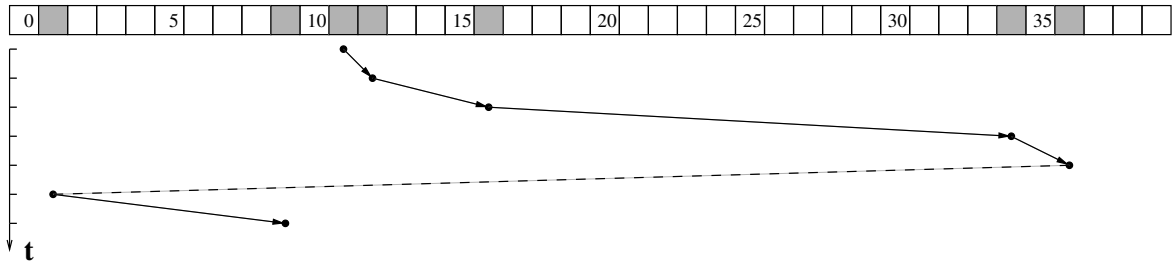


Abbildung 59: resultierende Kopfbewegung bei C-SCAN-Algorithmus

Eignung der Verfahren für Echtzeitsysteme

1. FIFO:

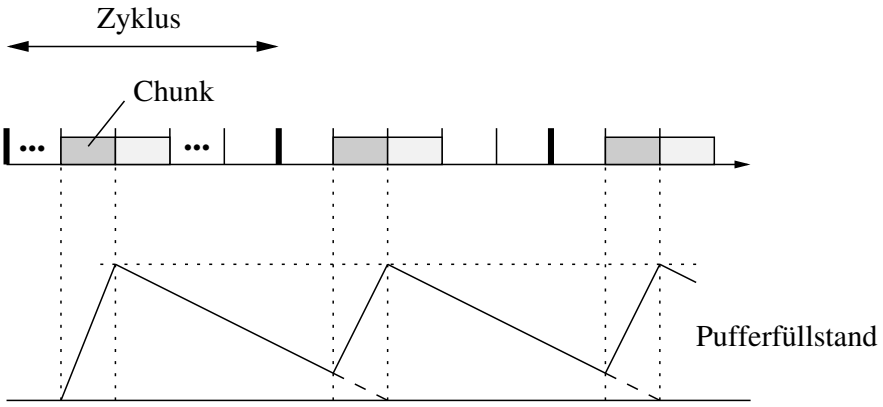


Abbildung 60: Timing und Verlauf des Pufferfüllstandes bei FIFO

- Playout kann nach Lesen des 1. Chunks starten
- maximaler Pufferfüllstand konstant
- da keine Optimierung: geringe Zahl gleichzeitig bedienbarer Ströme

2. SSTF:

- ungeeignet, da Verhungern möglich

3. SCAN:

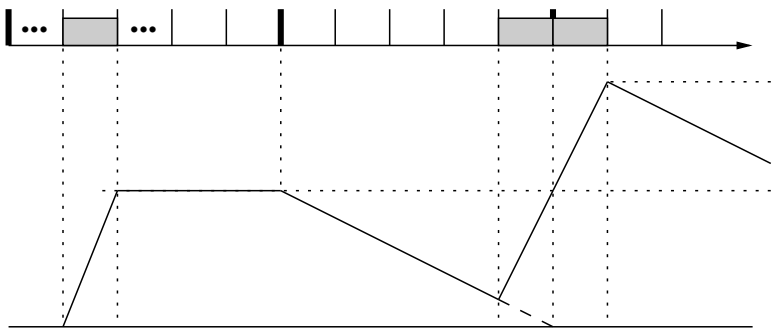


Abbildung 61: Timing und Verlauf des Pufferfüllstandes bei SCAN

- Playout kann erst nach Ende des 1. Zyklus starten

- maximaler Pufferfüllstand variabel → doppelt so große Puffer wie bei FIFO nötig
- hohe Anzahl gleichzeitig bedienbarer Ströme

Grouped Sweeping Scheduling (GSS) [18]

Motivation: Kompromiß aus benötigtem Pufferspeicher und Anzahl gleichzeitig bedienbarer Ströme

Prinzip:

- Aufteilung der Anzahl n der zu bedienenden Ströme in g Gruppen mit gleicher Anzahl Elemente (Chunks)
- → jede Gruppe maximal $\lceil n/g \rceil$ Einträge
- Abarbeitung Gruppe für Gruppe nach SCAN
- Parameter g bestimmt Systemverhalten:
  - ⋆ $g = 1$: SCAN
  - ⋆ $g = n$: FIFO

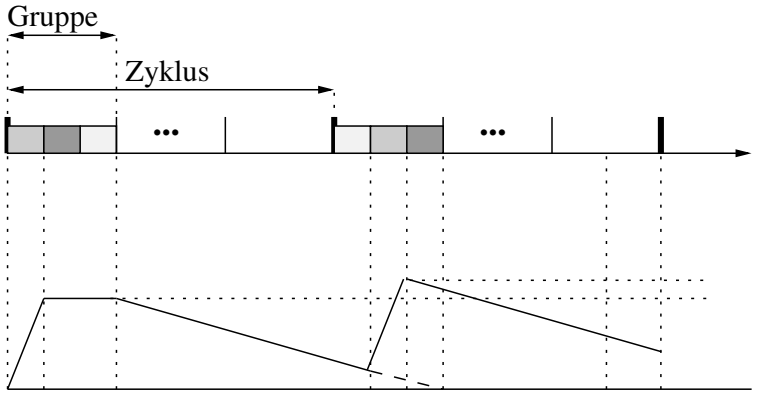


Abbildung 62: Timing und Verlauf des Pufferfüllstandes bei GSS

- Playout kann frühestens nach Ende der 1. Gruppe starten
- Speicherbedarf geringer als bei SCAN, höher als bei FIFO
- mittlere Effizienz

Benötigter Pufferspeicher B_b:
Es seien k Anzahl Datenblocks pro Chunk, B_m Größe eines Datenblocks in Bytes

- für jeden Strom muß genau 1 Chunk vorrätig gehalten werden (nkB_m)
- zusätzlich noch die Platz für eine zusätzlichen Gruppe, da die Reihenfolge innnerhalb einer Gruppe beliebig ist ($\lceil n/g \rceil kB_m$)

daher:

$$B_b = \left(n + \left\lceil \frac{n}{g} \right\rceil \right) kB_m \tag{14}$$

Aufgabe: Ermitteln Sie den benötigten Pufferspeicherplatz für SCAN und FIFO!

GSS ist auch mit Deadlines möglich:

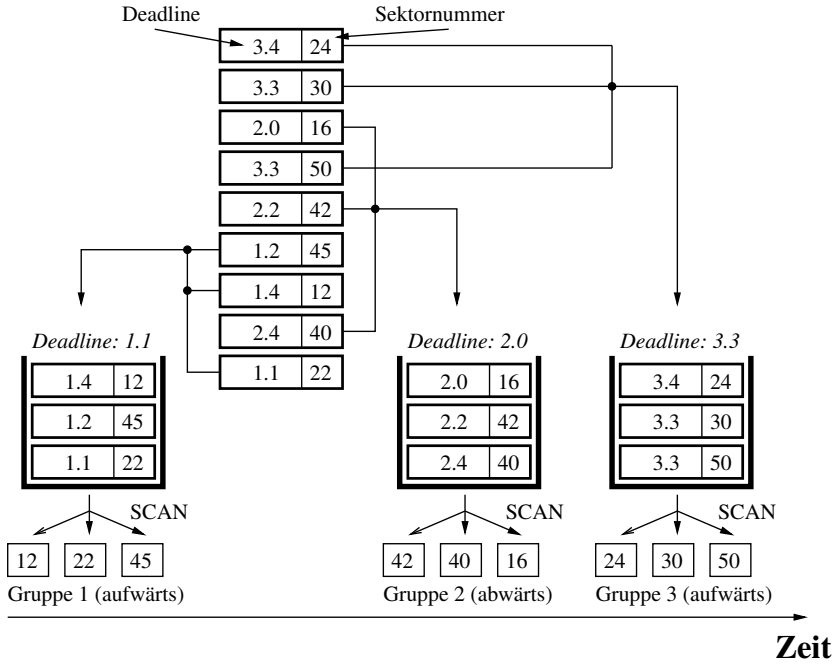


Abbildung 63: Grouped Sweeping Scheduling (GSS) mit Deadlines

Earliest Deadline First (EDF)

- Annahme: jedem Auftrag ist Deadline zugeordnet

- Es wird stets der Auftrag mit der kleinsten Deadline ausgewählt.
- gleiche Deadlines: zufällige Auswahl oder Reihenfolge des Eintreffens
- geringer Durchsatz,
- hohe Positionierzeiten

⇒ Kombination mit SCAN:

SCAN-EDF

- Es wird stets der Auftrag mit der kleinsten Deadline ausgewählt. Mehrere Aufträge mit gleicher Deadline → SCAN
- Leistung abhängig von Anzahl gleicher Deadlines (Versuch, durch Modifikation der Deadlines Durchsatz zu erhöhen)

Balancierte Puffer

- 1 Puffer pro Datenstrom (Klient)
- 2 Kriterien: Beschränkung des Pufferplatzes, hohe Effizienz des Transfers
- Je nach Situation Einsatz zweier Verfahren:
  - SSTF: wenn Puffer ausreichend und etwa gleich gefüllt
  - balanciertes Auffüllen: wenn Gefahr eines Unterlaufs in einem oder mehreren Puffern

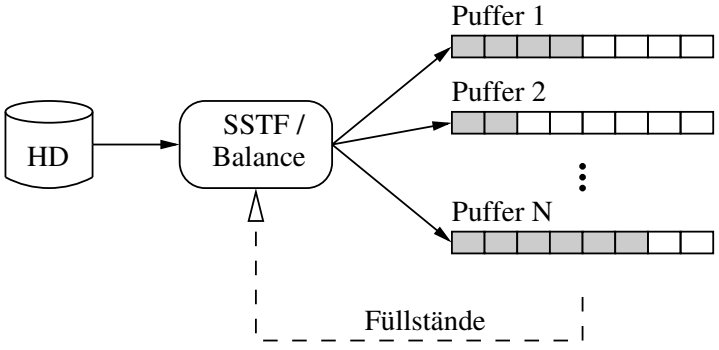


Abbildung 64: Prinzip der Balancierten Puffer

Weitere Aspekte

Festplattenverhalten schwer zu modellieren, weil:

- Caching: Ersetzungsstrategie und Zugriffsgeschwindigkeit undokumentiert
- thermische Einflüsse erfordern periodische Rekalibrierung ($\rightarrow$ AV-Platten verschieben Rekalibrierung bis Platte *idle*)
- Einfluß des Bussystems zu beachten
- Zoning: Anzahl Sektoren einer Spur variabel je nach Position auf Platte
- Read-Ahead: Festplatte liest einen oder mehrere Folgesektoren „vorsorglich“ in Cache

Leseanregungen

- guter Überblicksartikel [12]
- weitere Schedulingverfahren: [1],[7], [6]

4.3 Flash-Speicher

4.3.1 Grundlagen

- elektrisch löschbar, nichtflüchtig
- drei grundlegende Operationen: *read*, *write*, *erase*
- (noch) nicht ganz so schnell wie Festplatte
- mechanisch unempfindlich
- Massenspeichermedium, besonders für eingebettete Systeme
- Beschreiben nur in eine „Richtung“ möglich (z. B. nur 1$\rightarrow$0), danach muss gelöscht werden

- begrenzte Anzahl Löschzyklen – ca. 10.000-1.000.000 → Verteilung der Schreibzugriffe (*Wear Levelling*) notwendig
- zwei grundlegende Typen: NOR- und NAND-Flash

	NOR	NAND
Zugriff	wortweise	seitenweise
Löschoperation	wortweise	blockweise
	langsam	schnell
Speicherdichte	niedrig	hoch
XiP	möglich	unmöglich
Performance	langsam	schnell

Tabelle 2: Vergleich von NOR- und NAND-Flash

NOR-Flash wird meist zum Ablegen von Code (BIOS, Firmware, etc.] eingesetzt, NAND-Flash zur Speicherung von Daten.

4.3.2 Flash-Medien

- Ursprung: Consumer Electronics (Digitalkameras, ...)
- NAND-Flash
- wesentliche Parameter: Kapazität, (Zugriffsgeschwindigkeit)
- erreichte Kapazität (06/2008): 32 GB, schnell wachsend
- Vielfalt konkurrierender Standards und Substandards:
  - Smart Media (SM) – Toshiba
  - xD Picture Card – Olympus, Fujifilm
  - Memory Stick (MS) – Sony
  - Multimedia Card (MMC) – Ingentix, SanDisk
  - Secure Digital Memory Card (SD) – SanDisk

  - CompactFlash (CF) – SanDisk
  - USB-Sticks – M-Systems
  - Solid State Drives (SSD)

- Medium kapselt einen Controller (außer SM und xD) sowie Speicherbaustein(e)
- Controller realisiert das Kommunikationsprotokoll nach außen, *Wear Leveling* sowie die Ansteuerung des Speichers
- Firmware der Controller i. a. undokumentiert bzw. gut versteckt
- eingesetztes Dateisystem: typisch FAT32

Timingverhalten individueller Medientypen weicht stark voneinander ab: ([10]):

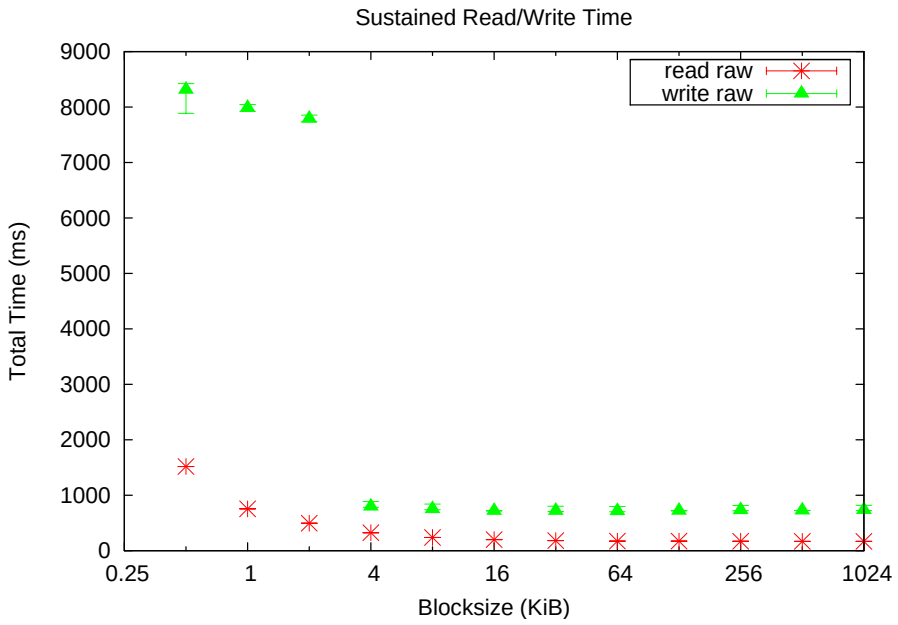


Abbildung 65: Zugriffszeiten für verschiedene Blockgrößen bei einem CF-Medium

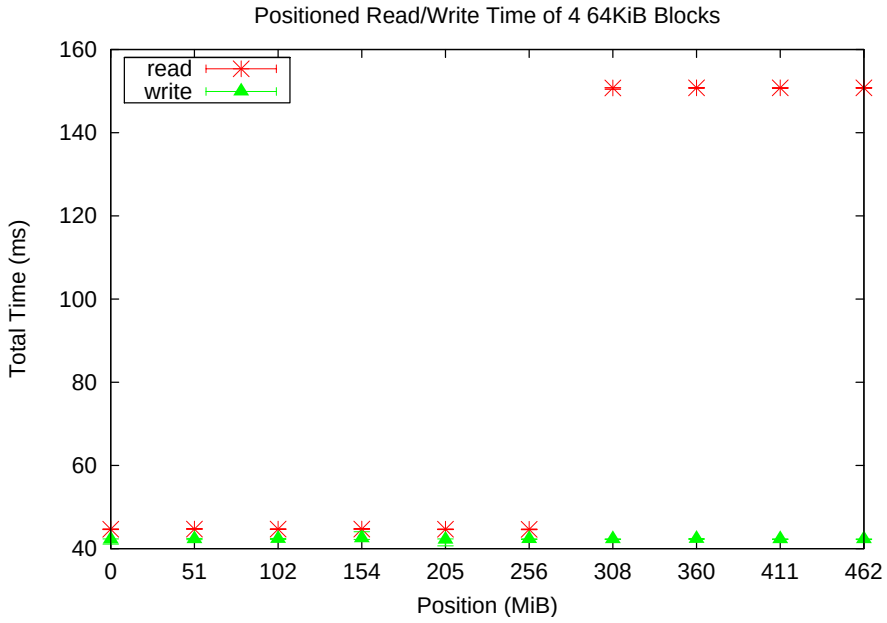


Abbildung 66: Zugriffszeiten an verschiedenen Positionen eines MMC-Mediums

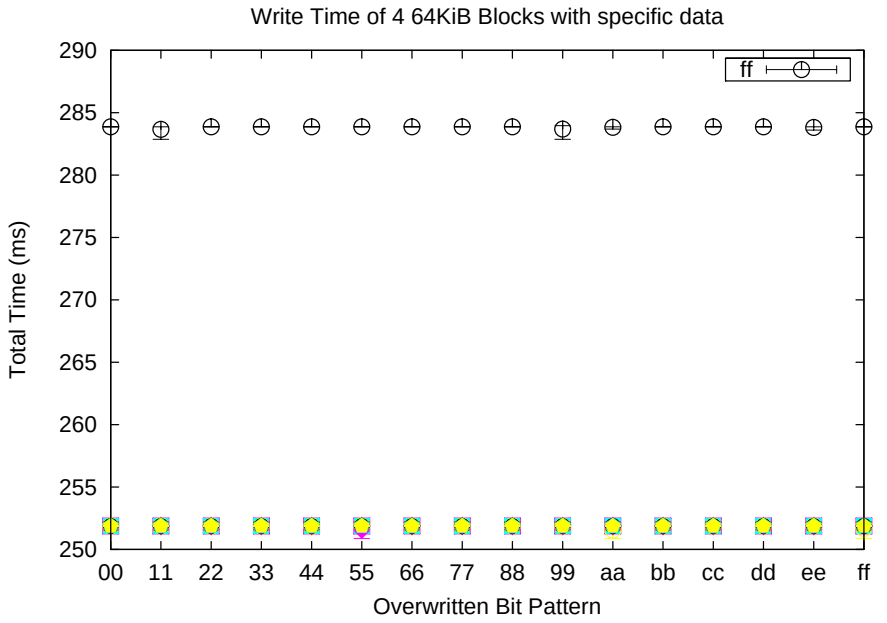


Abbildung 67: Zeiten für das Schreiben bestimmter Bitmuster bei einem SD-Medium

Fazit: obwohl das Timing der einzelnen Operationen simpel ist, bestehen bei Flash-Medien Anomalien hinsichtlich

- genutzter Blockgröße,
- zugegriffener Adresse,
- geschriebenem Bitmuster (!).

Das Timing sorgfältig selektierter Medien ist jedoch weitaus deterministischer als das entsprechender Festplatten.

4.3.3 Dateisysteme für Flashspeicher

Idee: Host übernimmt Management des Flashspeichers in Form eines spezialisierten Dateisystems.

Obwohl verschiedene Dateisysteme für „rohen“ Flashspeicher existieren, ist bislang keines explizit für Echtzeitverarbeitung entworfen worden.

- Journaling Flash File System: JFFS,
- Yet Another Flash File System: YAFFS,
- LogFS.

4.4 Verwaltung des Hauptspeichers

4.4.1 Prinzip des virtuellen Speichers

Grundideen:

- **Privatisierung**: jeder Prozeß besitzt eigenen (virtuellen) Adreßraum $\rightsquigarrow$ Schutz! (virtueller Prozessor)
- **Segmentierung**: virtueller und physischer Speicher wird in Segmente fester Größe eingeteilt (Kacheln, Frames, Pages, Seiten)

- Vergrößerung des nutzbaren Speichers durch Hinzunahme der Festplatte

Abbildung virtueller auf physische Adressen: Hardware (MMU) + Betriebssystem (HS-Verwaltung)

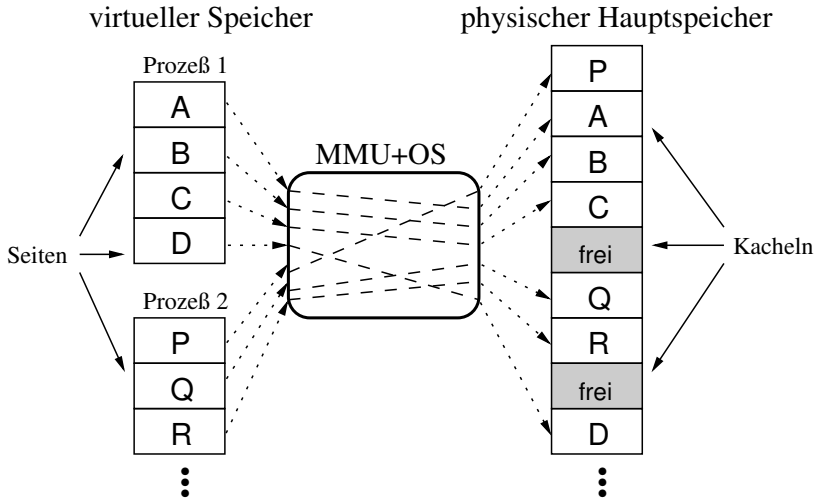


Abbildung 68: Abbildung logischer Seiten auf physische Kacheln

Mechanismus der Adreßübersetzung

- Zugriff auf physischen Hauptspeicher mittels *Seitentabellen* (page tables) – Datenstrukturen im Hauptspeicher
- 1–n Seitentabellen pro Prozeß
- Eintrag (*Page Table Entry – PTE*) in einer Seitentabelle vermerkt:
  - (physische) Anfangsadresse der Seite in Hauptspeicher oder auf Platte
  - Zugriffsrechte (Lesen, Schreiben, Ausführen)
  - Ort der Speicherung (Hauptspeicher oder Festplatte): *Valid Bit*
- Aufteilung der virtuellen Adresse: Index und Seitennummer

- Seitennummer selektiert Eintrag in Seitentabelle (physische Adresse der gewünschten Seite)
- Index selektiert individuelles Datum innerhalb der Seite
- physische Adresse des gesuchten Datums durch Addition von Index und Seitenadresse
- bei *jeder* Speicherreferenz nötig ⇒ Hardware!

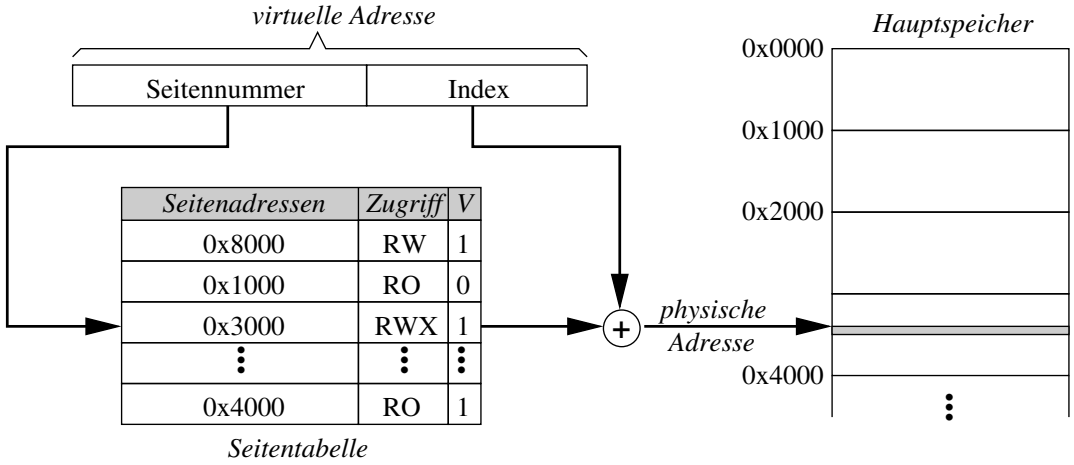


Abbildung 69: Prinzip der gestreuten Speicherung

Problem: Länge der Seitentabelle (muß im Ganzen im Speicher gehalten werden)

⇒ Hierarchive von Seitentabellen

Beispiel: zweistufige Seitentabellen (i386)

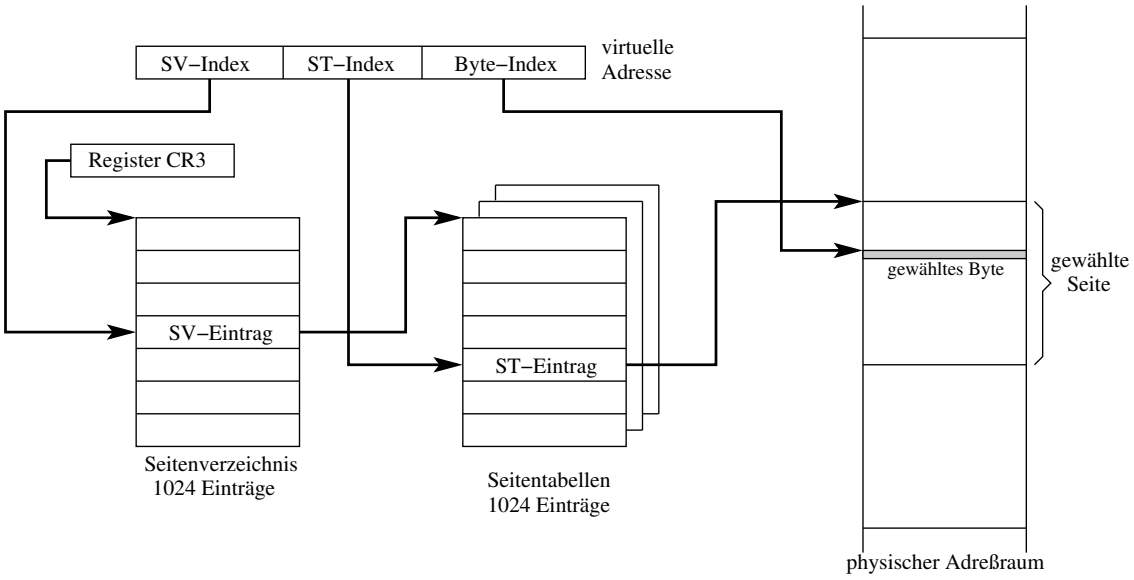


Abbildung 70: Virtuelles Speichermodell des i386

Verbesserung der Adreßumsetzung durch Translation Lookaside Buffer (TLB) = Cache für Umsetzungen logische → physische Adressen. z.B. Pentium III: 96 Einträge

Optimierungsprinzipien:

- "lazy evaluation": Operationen so lange wie möglich verzögern
- "copy-on-write": Kopien erst bei schreibendem Zugriff

Anwendungsbeispiele:

- verzögertes Kopieren beim `fork()`-Systemruf
- Verzögerung von Schreibvorgängen auf Dateien

Demand Paging

Idee: Benötigte Seiten werden erst bei Bedarf in den Hauptspeicher geladen. Ist der physische Speicher restlos ausgenutzt, so muß zuvor eine andere Seite ausgelagert werden.

prinzipieller Ablauf

1. Prozeß referenziert eine Adresse (neuer Befehl, Zugriff auf Datum)
2. MMU führt Adreßübersetzung aus
3. BS prüft:
   - referenzierte Seite im physischen Speicher → Ausführung der Instruktion, Weiterarbeit
   - referenzierte Seite momentan ausgelagert → Seitenfehler (*page fault*, eine Art Interrupt)

Seitenfehler:

1. Prozeß wird angehalten (blockiert)
2. BS sucht freie HS-Kachel
3. falls keine freie HS-Kachel verfügbar → Auswahl einer auszulagernden Seite, Auslagerung auf Festplatte
4. Einlesen der referenzierten Seite in (nun) freie HS-Kachel von Festplatte
5. Weiterarbeit des blockierten Prozesses

Pro Seitenfehler bis zu 2 Festplattenzugriffe notwendig!

Optimierungskriterium: Minimierung der Seitenfehlerrate

Virtueller Speicher – Vorteile:

- riesiger Adreßraum
- Schutz der Aktivitäten voreinander
- bessere Auslastung des Hauptspeichers, sehr flexibel
- einfache Programmierung

Probleme

- Anomalien: Erhöhung der Seitenanzahl eines Prozesses kann zu höherer Seitenfehlerrate führen (Belady's Anomalie)

- Seitenflattern (*thrashing*): Prozesse verdrängen gegenseitig ihre Seiten
- Unvorhersehbarkeit des zeitlichen Eintreffens von Seitenfehlern

4.4.2 Virtueller Speicher in Echtzeitsystemen

Beispiel: Task mit Zeitanforderung $(t_p, t_e) = (30 \text{ ms}, 6 \text{ ms})$. Seitenfehler und einfacher Plattenzugriff blockiert Task für 10 ms (Festplattenzugriff). Seitenfehler und doppelter Plattenzugriff blockieren für 20 ms. $\Rightarrow t_e = [6...26 \text{ ms}]$ (vgl. Hausaufgabe)

Inakzeptable Varianz von t_e!

Weiteres Problem: Zeitpunkte und Reihenfolge der Seitenfehler praktisch nicht vorhersehbar.

Lösung A: Pinning

Idee: Zeitkritische Seiten (z.B. von Echtzeittasks) werden vom Auslagern auf die Festplatte ausgenommen.

statisch: Seiten werden über die gesamte Laufzeit fest im Speicher gehalten.

dynamisch: Seiten werden für bestimmte Zeitspannen im Speicher festgehalten.

Voraussetzung: Applikationen erhalten (teilweise) Kontrolle über Speicherverwaltung $\rightsquigarrow$ User-level Paging

Probleme:

- Anzahl gepinnter Seiten: Je mehr Seiten gepinnt, desto größer die Seitenfehlerrate für verbleibende (auslagerbare) Seiten $\Rightarrow$ sinkende Systemleistung.
- Sicherheit: pinnenden Prozessen muß vertraut werden

Beispiel: UNIX-Systemrufe (POSIX Realtime Extensions, IEEE

1003.1b):

- `mlock()`: pinnt den als Argument übergebenen Speicherbereich
- `munlock()`: hebt die Sperrung wieder auf
- `mlockall()`: schaltet das Paging für den betreffenden Prozeß komplett ab (*alle* Seiten des Prozesses gepinnt)
- `munlockall()`: schaltet das Paging für den Prozeß wieder ein

(Nur Aktivitäten mit `root`-Rechten dürfen diese Systemrufe ausführen)

Lösung B: Verzicht auf Virtuellen Speicher

- viele Prozessoren besitzen keine MMU (z.B. viele DSPs, 8-Bit-Mikrocontroller wie der Motorola HC11, Power PC 403 ...)
- MMU kann u. U. auch abgeschaltet werden
- entsprechendes BS nötig: uClinux, eCos, ...

Vorteile:

- preiswertere Hardware
- zeitliche Determinierbarkeit aller Hauptspeicherzugriffe
- $\Rightarrow$ Prozessoren für Echtzeitsysteme besitzen keine MMU (z.B. DSPs)

Nachteile:

- Verlust des Schutzes zwischen Applikationen,
- geringere Flexibilität bzgl. Speicheranforderungen zur Laufzeit,
- Begrenzung Applikationsgröße auf Größe des physischen Adreßraums.

Literatur

[1] Eunjeong Park anmd Namyun Kim, Sangsoo Park, Jinhwan Kim, and Heonshik Shin. Dynamic Disk Scheduling for Multimedia Storage Servers. In *Proceedings of the IEEE Region 10 Conference*, pages 1483–1486, 1999.

[2] N. Audsley, K. Tindell, and A. Burns. The End of the Line for Static Cyclic Scheduling? In *Proceedings of the Fifth Euromicro Workshop on Real-time Systems*, pages 36–41, Oulu, Finland, June 1993. IEEE Computer Society Press,.

[3] Neil C. Audsley. Deadline Monotonic Scheduling. Technical Report YCS 146, Department of Computer Science, University of York, September 1990.

[4] Theodore P. Baker and Alan C. Shaw. The Cyclic Executive Model and Ada. *Real-Time Systems Journal*, 1(1), 1989.

[5] D. James Gemmell, Harrick M. Vin, Dilip D. Kandlur, P. Venkat Rangan, and Lawrence A. Rowe. Multimedia Storage Servers: A Tutorial. *IEEE Computer*, 28(5), 1995.

[6] Yin-Fu Huang and Jiing-Maw Huang. Disk Scheduling on Multimedia Storage Servers. *IEEE Transactions on Computers*, 53(1):77–82, January 2004.

[7] Ibrahim Kamel, T. Niranjan, and Shahram Ghandeharizedah. A Novel Deadline Driven Disk Scheduling Algorithm for Multi-Priority Multimedia Objects. In *Proceedings of the 16th International Conference on Data Engineering*, pages 349–361, 2000.

[8] C. L. Liu and James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[9] Jane W. S. Liu. *Real-Time Systems*. Prentice Hall, 1. edition, 2000.

[10] Daniel Parthey and Robert Baumgartl. Analyzing Access Timing of Removable Flash Media. In *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'07)*, Daegu, Korea, August 2007.

[11] Krithi Ramamritham and John A. Stankovic. Scheduling Algorithms and Operating Systems Support for Real-Time Systems. *Proceedings of IEEE*, 82(1):55–67, January 1994.

[12] A. L. Narashima Reddy and James C. Wylie. I/O Issues in a Multimedia System. *IEEE Computer*, 27(3):69–74, March 1994.

[13] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–29, 1994.

[14] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, September 1990.

[15] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio C. Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *IEEE Computer*, 29(6):16–25, June 1995.

[16] Jia Xu and David Lorge Parnas. Scheduling Processes with Release Times, Deadlines, Precedence and Exclusion Relations. *IEEE Transactions on Software Engineering*, 16(3), March 1990.

[17] Jia Xu and David Lorge Parnas. Priority Scheduling Versus Pre-Run-Time Scheduling. *International Journal of Time-Critical Computing Systems*, (18):7–23, 2000.

[18] Philip S. Yu, Mon-Song Chen, and Dilip D. Kandlur. Grouped sweeping scheduling for DASD-based multimedia storage management. *Multimedia Systems*, 1(3):99–109, November 1993.