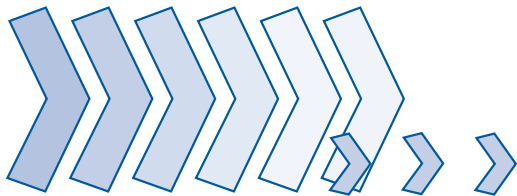
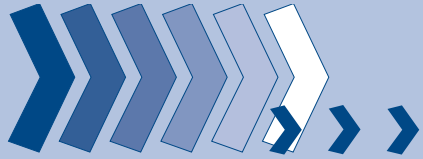


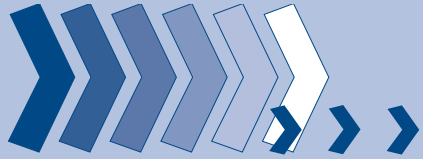
# Weitere Probleme des RT-Schedulings





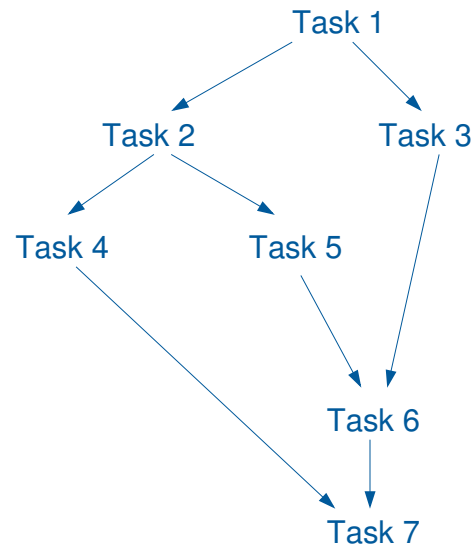
## *Bisher nicht angesprochene Probleme*

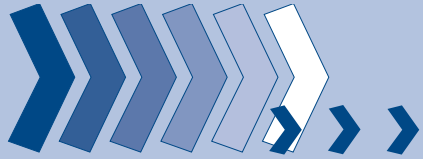
- Scheduling mit Abhängigkeiten (Taskgraphen, Ressourcenzugriff)
- IRIS-Tasks - je länger eine Task läuft, desto “besser” das Ergebnis
- Berechnung von Ausführungszeiten
- Taskzuweisung bei Multiprozessoren



## Scheduling mit Abhängigkeiten (Taskgraph)

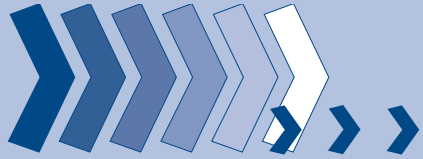
- Bisher:  
Tasks sind unabhängig, können also in beliebiger Reihenfolge ausgeführt werden
- Nun:  
Abhängigkeiten zwischen den Tasks beschränken die möglichen Schedules:



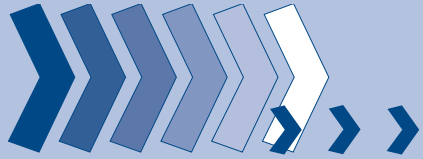


## Algorithmus – Voraussetzungen

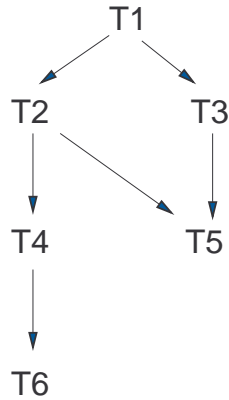
- Release-Time 0
- Deadline, Ausführungszeit für jede Task bekannt
- Deadlines und Ausführungszeiten sind so gewählt, daß gilt: Wenn jede Task spätestens zu ihrer Deadline beendet ist, dann ist genug Zeit für die Ausführung ihrer Nachfolger vorhanden ( $U \leq 1$ )
- Tasks sind so numeriert, daß gilt:  $D_1 \leq D_2 \leq \dots \leq D_n$



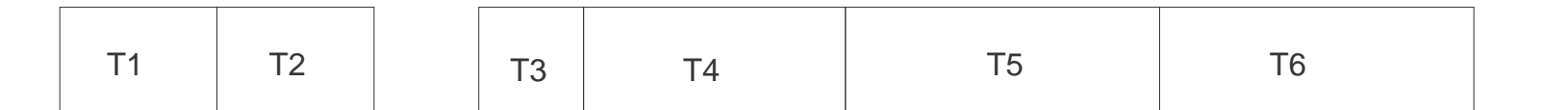
1. Schedule Task  $T_n$  im Intervall  $[D_n - e_n, D_n]$
2. Für alle noch nicht “geschedule’ten” Tasks:
  - $A$  sei die Menge aller noch nicht “geschedule’ten” Tasks, deren Nachfolger (so vorhanden) alle bereits “geschedule’t” sind
  - Schedule Task  $T_k$  so spät wie möglich mit  $k = \max\{m | m \in A\}$
3. Bewege alle Tasks so weit wie möglich nach “vorne” unter Bewahrung der im Schritt 2 festgelegten Ordnung



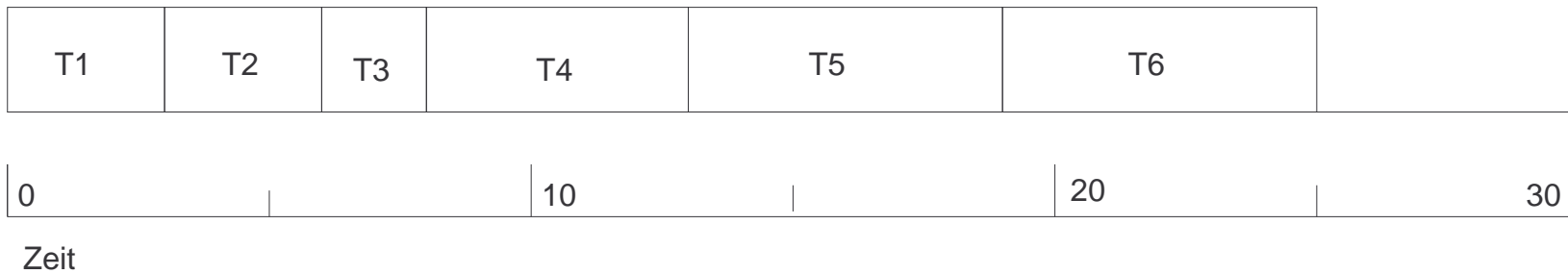
# Algorithmus – Beispiel

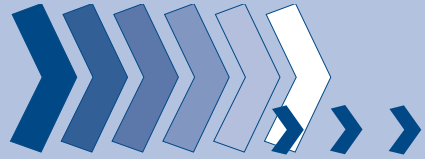


Schritt 2:

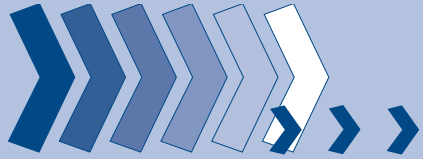


Schritt 3:





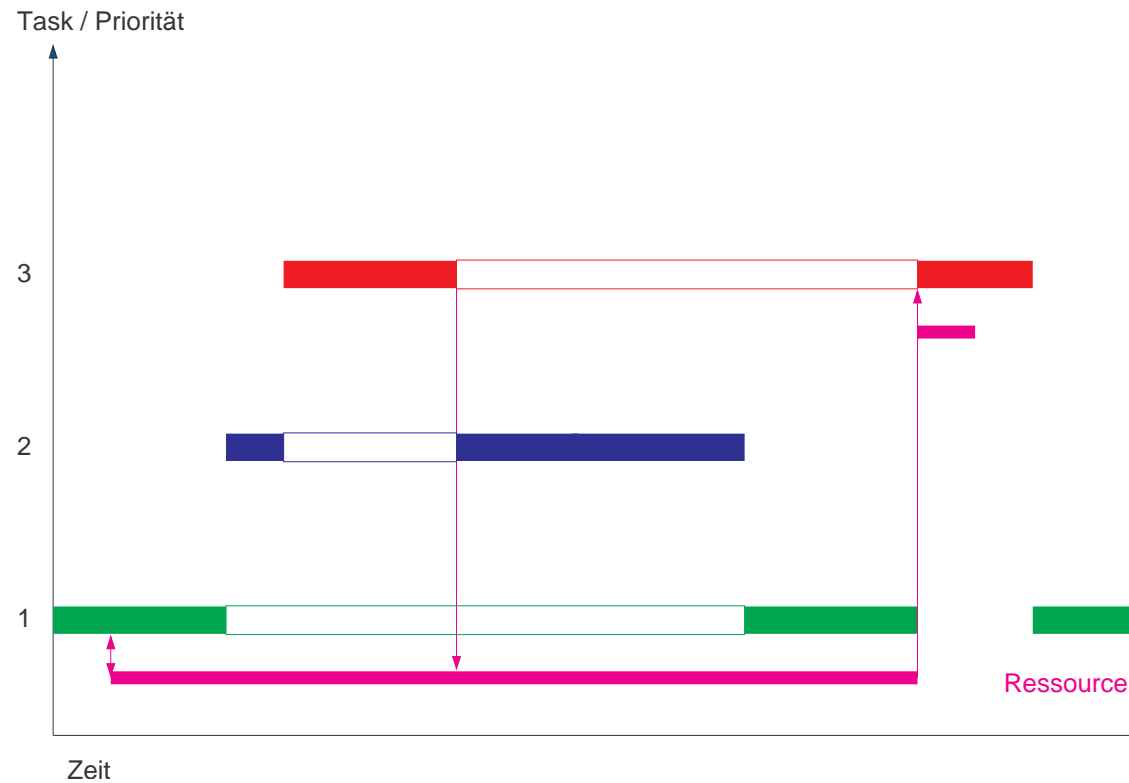
- Tasks können Ressourcen mit exklusivem Zugriff benutzen (beispielsweise Ausgabe-geräte)
- Zugriffssteuerung per Mutual Exclusion wie üblich, falls nötig
- Für non-preemptive Tasks kein Problem:  
Ressource wird allokiert, benutzt, und wieder freigegeben, ohne daß ein Konflikt auftreten kann
- Was passiert im preemptiven Fall?
  - Genügen Semaphore?
  - Können Probleme auftreten, die in Nicht-Echtzeitsystemen so nicht vorkommen?

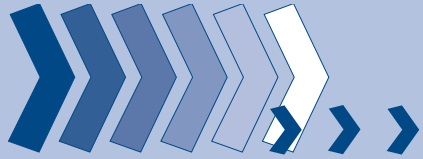


# Priority Inversion

Problem:

Niedrigpriorisierte Task hält eine Ressource, auf die eine hochpriorie Task wartet, und blockiert diese damit

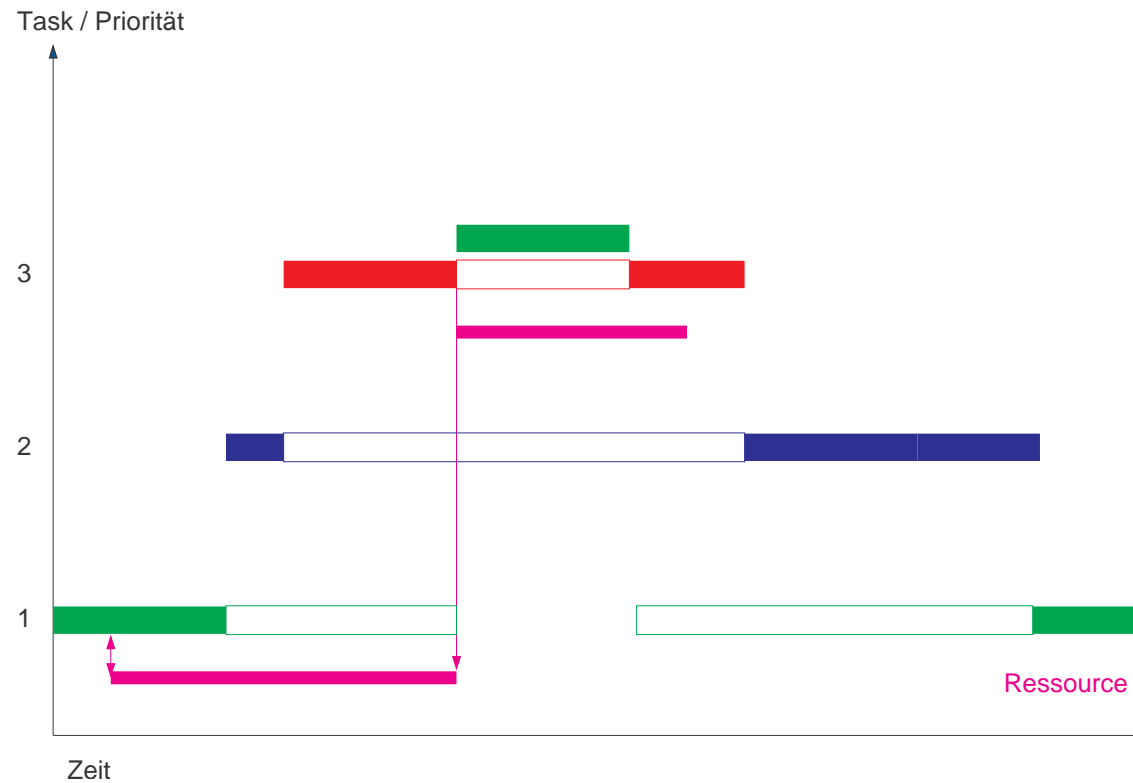


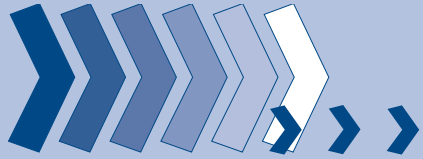


# Priority Inheritance

Idee:

Wenn Prozeß  $a$  durch Prozeß  $b$  blockiert wird, läuft  $b$  bis zur Freigabe der Ressource mit der Priorität von  $a$ .

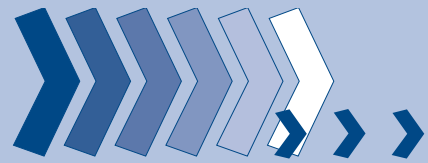




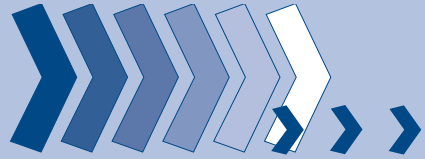
## *Probleme mit Priority Inheritance*

- Ketten von Blockaden können auftreten
- Deadlocks können auftreten
- Hochprioritäre Prozesse können inakzeptabel lange blockiert werden
- Obere Grenze für die Anzahl der Blockaden kann berechnet werden, liefert einen sehr pessimistischen schlechtesten Fall

Lösung: Priority Ceiling — die Ressourcen erhalten ebenfalls Prioritäten



1. Jeder Prozeß hat eine statische Priorität (beispielsweise als Ergebnis eines Schedulingverfahrens)
2. Jede Ressource hat einen statischen *ceiling value*, der als das Maximum der Prioritäten der Prozesse definiert ist, die die Ressource benutzen
3. Jeder Prozeß hat eine dynamische Priorität, die das Maximum der eigenen statischen Priorität und der *ceiling values* aller benutzen Ressourcen ist
4. Ein Prozeß kann nur dann eine Ressource anfordern, wenn seine dynamische Priorität höher ist als die *ceilings* aller gerade gelockten Ressourcen (mit Ausnahme derer, die er selbst benutzt)



## IRIS: Increased Reward with Increased Service

### Idee:

- Es gibt Probleme, bei denen das Ergebnis mit wachsender Laufzeit des Algorithmus immer “besser” wird
- Ein akzeptables Ergebnis liegt dabei nach einer bestimmten Zeit vor
- Laufzeit darüber hinaus führt zu besseren Ergebnissen (bessere Steuerqualität, höherer Komfort für den Nutzer, ...)

Beispiel: Berechnung von  $\pi$ , Steuerungsaufgaben, Optimierungsprobleme

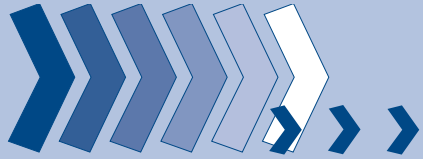


### Teile einer IRIS-Task

- *mandatory portion*: Dieser Teil der Task muß bis zur Deadline ausgeführt werden (obligatorischer Teil)
- *optional portion*: Dieser Teil der Task kann ausgeführt werden, wenn die Zeit es zuläßt (optionaler Teil)

### Unterschied zu weicher Echtzeit:

- Ein Ergebnis (mindestens obligatorischer Teil) muß bis zur Deadline vorliegen
- Ergebnisse nach der Deadline haben keinen Wert, Task beendet Ausführung spätestens mit der Deadline

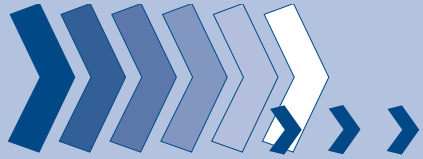


## Nutzen einer IRIS-Task I

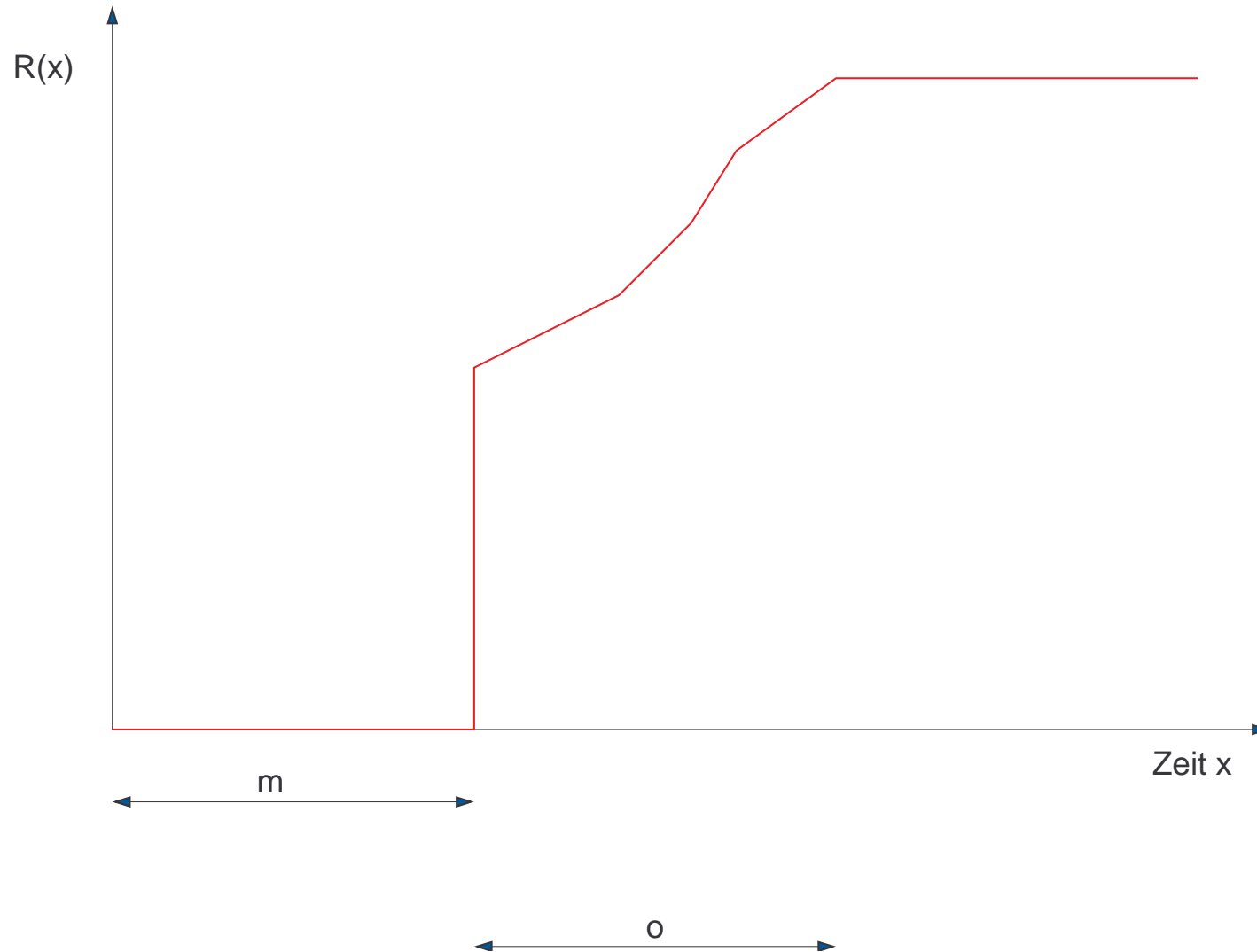
$$R(x) = \begin{cases} 0 & \Leftrightarrow x < m \\ r(x) & \Leftrightarrow m \leq x \leq o + m \\ r(o + m) & \Leftrightarrow o + m < x < D \\ \leq 0 & \Leftrightarrow x > D \end{cases}$$

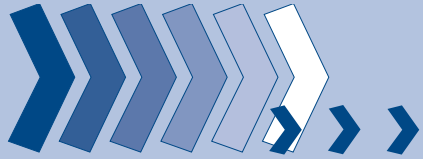
dabei:

- $r(x)$  monoton steigend (nicht notwendigerweise streng monoton steigend)
- $m$  Länge des obligatorischen Teils
- $o$  Maximal mögliche Länge des optionalen Teils, danach keine Verbesserung des Ergebnisses mehr möglich
- $D$  Deadline



## Nutzen einer IRIS-Task II





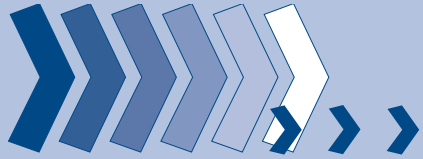
## *Scheduling von IRIS-Tasks I*

Ziel:

- Deadlines müssen eingehalten werden (d.h. alle obligatorischen Teile müssen ausgeführt werden)
- Maximierung des Nutzens

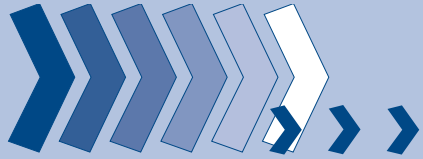
Gesamtnutzen:

- ist Funktion der Nutzenfunktionen der einzelnen Tasks (beispielsweise gleichwertige Aufsummierung)
- kann zielgerichtet durch verschiedene Bewertung einzelner Tasks parametrisiert werden



### Beispiel: Lineare Nutzenfunktionen und Gleichberechtigung

- Zerlegung der Task  $T_i$  in  $M_i$  (obligatorische Teile) und  $O_i$  (optionale Teile) als Teiltasks
- Wenn Taskset aus  $T_i$  nach EDF feasible: fertig (optimaler Schedule)
- Wenn Taskset aus  $M_i$  nach EDF nicht feasible: Es gibt keinen Schedule
- Sonst:
  - $M_i$  nach EDF “schedulingen”
  - Minimieren der Idle-Zeiten durch Verschieben der Tasks und Ausnutzen der freien Zeiten für die Ausführung der optionalen Teile der Tasks



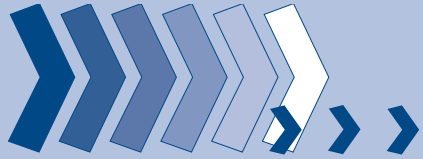
## *Berechnung von Ausführungszeiten*

- Schedulingverfahren basieren auf Kenntnis der Ausführungszeit einer Task
- Zeitliche Garantien können nur gegeben werden, wenn man Ausführungszeiten kennt

Problem: Wie mißt man eine Ausführungszeit?

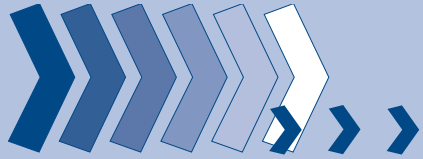
- Stoppuhr? Logic-Analyzer?

Genügt das?



## *Worst Case Execution Time I*

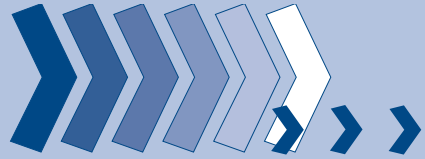
- Kenntnis **einer** Ausführungszeit genügt nicht, denn Laufzeit eines Programmes kann von den Eingaben abhängen
- Interessant ist die längstmögliche Ausführungszeit (WCET)
- WCET hängt ab von
  - Programmcode
  - Compiler, benutzte Bibliotheken
  - Pfade durch den Programmcode (abhängig von Eingaben)
  - benutzter CPU
  - Parametern der Umgebung der CPU (Speicher, Caches)
  - sowie Wechselwirkungen zwischen diesen Parametern



## *Worst Case Execution Time II*

Es gibt zwei generelle Probleme:

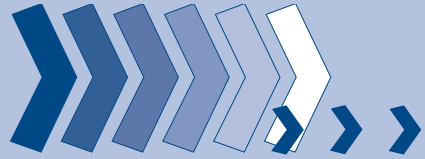
- Welches ist der längste Pfad durch ein Programm?  
Unterproblem: Wie bewertet man einen Pfad?
- Wie berechnet man zu einem gegebenen Pfad die Ausführungszeit?



Beispiel: Wie hoch ist die WCET dieses Programmstückes?

```
int i,k;
int j = readsensor();
while(i<j)
{
    k+=sensorfunction(i);
    i++;
}
```

Problem: Wie oft wird diese Schleife durchlaufen?

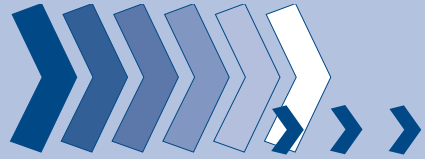


Beispiel: Wie hoch ist die WCET dieses Programmstückes?

```
int i,k;
int j = readsensor();
while(i<j)
{
    k+=sensorfunction(i);
    i++;
}
```

Problem: Wie oft wird diese Schleife durchlaufen?

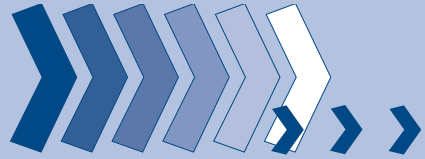
Lösung: Angabe von Obergrenzen



Beispiel: Wie hoch ist die WCET dieses Programmstückes?

```
int i,k;
int j = readsensor();
while(i<j)
{
    k+=sensorfunction(i);
    i=somefunction();
}
```

Problem: Terminiert diese Schleife überhaupt?

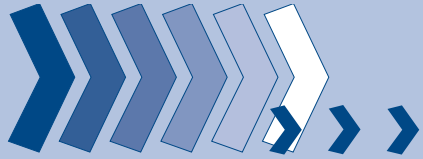


Beispiel: Wie hoch ist die WCET dieses Programmstückes?

```
int i,k;
int j = readsensor();
while(i<j)
{
    k+=sensorfunction(i);
    i=somefunction();
}
```

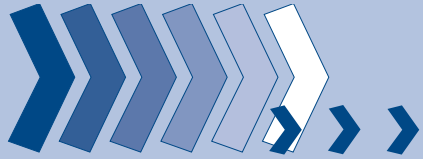
Problem: Terminiert diese Schleife überhaupt?

Generell: Halteproblem ist nicht entscheidbar!



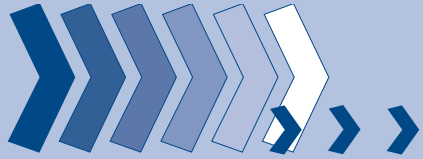
## *Längste Pfade III*

- Im allgemeinen ist die Suche eines längsten Pfades nicht entscheidbar
- Unter speziellen Annahmen sind Berechnungen dennoch möglich:
  - Kenntnis der maximal möglichen Anzahl von Durchläufen einer Schleife
  - Keine rekursiven Aufrufe
  - Keine dynamischen Datenstrukturen
- Folge: Strenge Einschränkung von Programmkonstrukten, die in Echtzeitprogrammen überhaupt “erlaubt” sind.



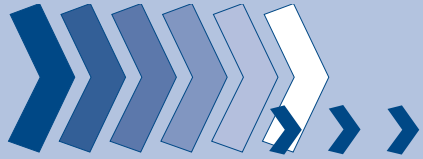
## *Berechnung einer Ausführungszeit*

- Gegeben: Programmcode und ein Ausführungspfad bzw. eine Folge von Maschinenbefehlen, sowie eine passende Hardware zur Ausführung
- Gesucht: WCET für dieses Programmstück
- Eigenschaften
  - Keine Verzweigungen
  - Genau bekannte Anzahl von Schleifendurchläufen
- Ansatz 1: Messen
- Ansatz 2: Berechnen auf Grundlage von Handbüchern zur CPU



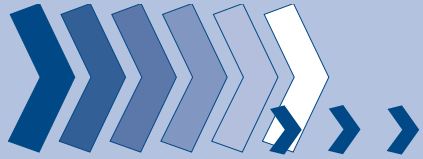
## *Ansatz 1: Messen einer WCET*

- Bei CPUs ohne Cache und Pipelines erfolgversprechende Methode
- Caches und/oder Pipelines:
  - Keine reproduzierbare Zeit in Abhängigkeit von Ausführungsumgebung (davor abgearbeitete Programme, Zustand des Speichers/Caches, Einfluß durch andere Programme (Preemption oder Interrupts))
  - Vollständiges Durchtesten aller möglichen Ausgangszustände ist durch hohe Komplexität unmöglich
- Methode kann benutzt werden, wenn ausreichende Annahmen über das System getroffen werden können (beispielsweise keine Preemption, stets gleicher Ausgangszustand, keine Interrupts)



## *Ansatz 2: Berechnen einer WCET*

- CPU-Hersteller liefern zu ihren Prozessoren (meist) Dokumentationen zum Zeitverhalten einzelner Instruktionen
- Diese Zeiten können auf geeignete Weise aufaddiert werden
- Pipeline-Effekte werden durch Überlappungszeiten repräsentiert
- Cache-Effekte müssen durch geeignete Cache-Vorhersage oder Worst-Case-Annahmen berücksichtigt werden



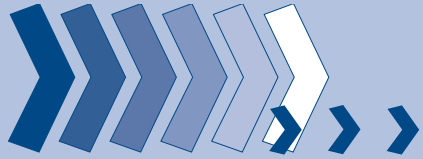
# Beispiel: Motorola 68000

Tabelle mit Ausführungszeiten

### Move Byte and Word Instruction Execution Times

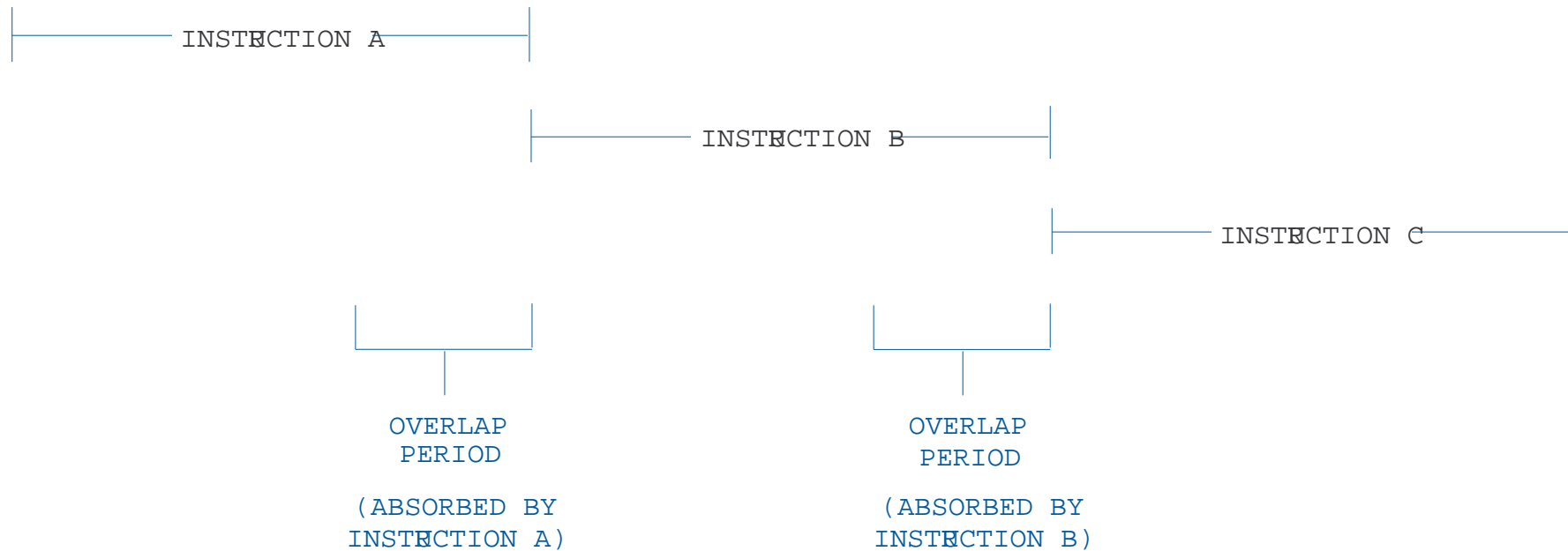
| Source                 | Destination |         |         |         |         |                        |               |         |         |
|------------------------|-------------|---------|---------|---------|---------|------------------------|---------------|---------|---------|
|                        | Dn          | An      | (An)    | (An)+   | -(An)   | (d <sub>16</sub> , An) | (dg, An, Xn)* | (xxx).W | (xxx).L |
| Dn                     | 4(1/0)      | 4(1/0)  | 8(1/1)  | 8(1/1)  | 8(1/1)  | 12(2/1)                | 14(2/1)       | 12(2/1) | 16(3/1) |
| An                     | 4(1/0)      | 4(1/0)  | 8(1/1)  | 8(1/1)  | 8(1/1)  | 12(2/1)                | 14(2/1)       | 12(2/1) | 16(3/1) |
| (An)                   | 8(2/0)      | 8(2/0)  | 12(2/1) | 12(2/1) | 12(2/1) | 16(3/1)                | 18(3/1)       | 16(3/1) | 20(4/1) |
| (An)+                  | 8(2/0)      | 8(2/0)  | 12(2/1) | 12(2/1) | 12(2/1) | 16(3/1)                | 18(3/1)       | 16(3/1) | 20(4/1) |
| -(An)                  | 10(2/0)     | 10(2/0) | 14(2/1) | 14(2/1) | 14(2/1) | 18(3/1)                | 20(3/1)       | 18(3/1) | 22(4/1) |
| (d <sub>16</sub> , An) | 12(3/0)     | 12(3/0) | 16(3/1) | 16(3/1) | 16(3/1) | 20(4/1)                | 22(4/1)       | 20(4/1) | 24(5/1) |
| (dg, An, Xn)*          | 14(3/0)     | 14(3/0) | 18(3/1) | 18(3/1) | 18(3/1) | 22(4/1)                | 24(4/1)       | 22(4/1) | 26(5/1) |
| (xxx).W                | 12(3/0)     | 12(3/0) | 16(3/1) | 16(3/1) | 16(3/1) | 20(4/1)                | 22(4/1)       | 20(4/1) | 24(5/1) |
| (xxx).L                | 16(4/0)     | 16(4/0) | 20(4/1) | 20(4/1) | 20(4/1) | 24(5/1)                | 26(5/1)       | 24(5/1) | 28(6/1) |
| (d <sub>16</sub> , PC) | 12(3/0)     | 12(3/0) | 16(3/1) | 16(3/1) | 16(3/1) | 20(4/1)                | 22(4/1)       | 20(4/1) | 24(5/1) |
| (dg, PC, Xn)*          | 14(3/0)     | 14(3/0) | 18(3/1) | 18(3/1) | 18(3/1) | 22(4/1)                | 24(4/1)       | 22(4/1) | 26(5/1) |
| #<data>                | 8(2/0)      | 8(2/0)  | 12(2/1) | 12(2/1) | 12(2/1) | 16(3/1)                | 18(3/1)       | 16(3/1) | 20(4/1) |

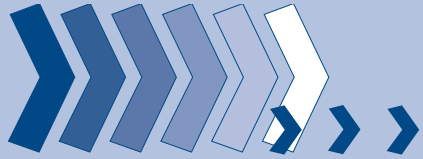
\*The size of the index register (Xn) does not affect execution time.



# Beispiel: Motorola CPU32 I

## Überlappende Ausführung von Instruktionen

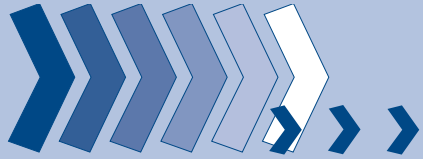




## Beispiel: Motorola CPU32 II

Tabelle mit Ausführungszeiten und Überlappungsparametern

| Instruction  | Head | Tail | Cycles    |
|--|------|------|-----------|
| MOVE Rn, Rn  | 0    | 0    | 2(0/1/0)  |
| MOVE <FEA>, Rn   | 0    | 0    | 2(0/1/0)  |
| MOVE Rn, (Am)  | 0    | 2    | 4(0/1/x)  |
| MOVE Rn, (Am)+   | 1    | 1    | 5(0/1/x)  |
| MOVE Rn, -(Am)   | 2    | 2    | 6(0/1/x)  |
| MOVE Rn, <CEA>   | 1    | 3    | 5(0/1/x)  |
| MOVE <FEA, (An)  | 2    | 2    | 6(0/1/x)  |
| MOVE <FEA>, (An)+  | 2    | 2    | 6(0/1/x)  |
| MOVE <FEA>, -(An)  | 2    | 2    | 6(0/1/x)  |
| MOVE #, <CEA>  | 2    | 2    | 6(0/1/x)* |
| MOVE <CEA>, <FEA>  | 2    | 2    | 6(0/1/x)  |
| X = There is one bus cycle for byte and word operands and two bus cycles for long operands. For long bus cycles, add two clocks to the tail and to the number of cycles. |      |      |           |
| *An # fetch effective address time must be added for this instruction:<br><FEA> + <CEA> + <OPER>   |      |      |           |

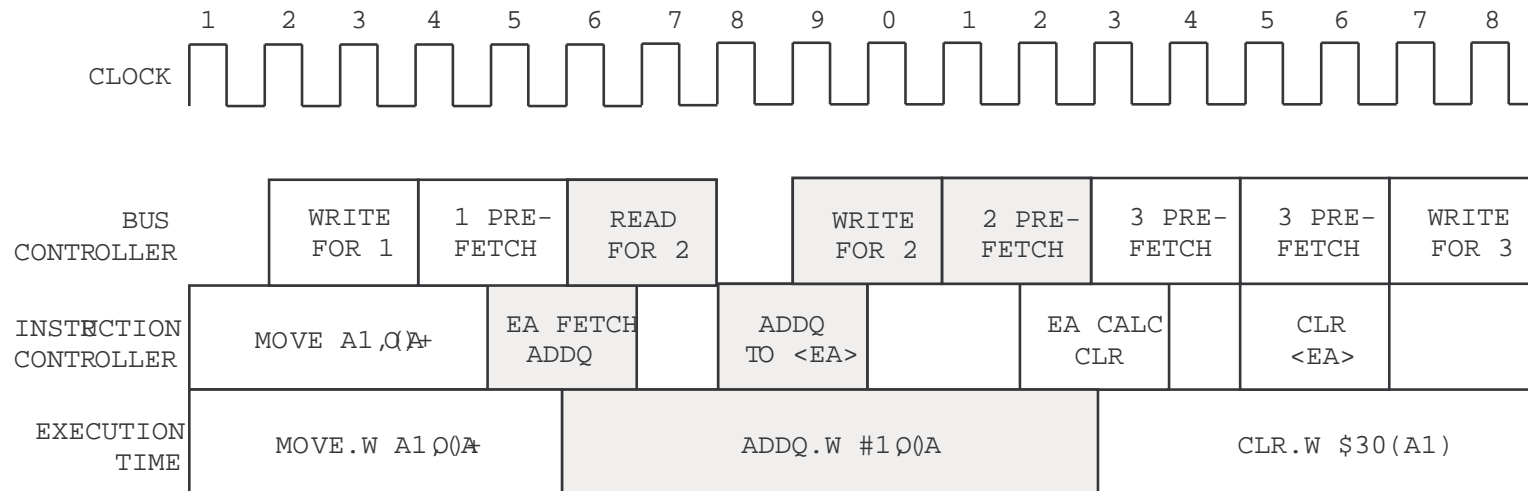


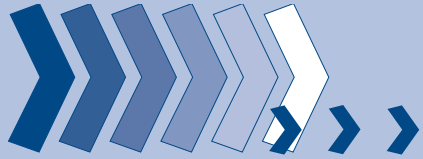
# Beispiel: Motorola CPU32 III

## Beispiel einer Ausführung

### Instructions

MOVE.W A1, (A0) +  
 ADDQ.W #1, (A0)  
 CLR.W \$30 (A1)





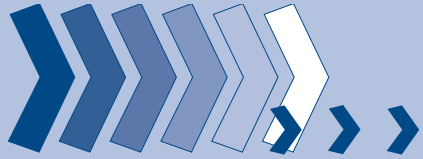
## *Taskzuweisung bei Multiprozessoren*

Bisher (Single-CPU):

- Ressourcenplanung in der Zeit
- Eindimensional im Raum (Tasks wurden alle auf dem gleichen Prozessor ausgeführt)

Multiprozessor:

- Ressourcenplanung in Zeit (wann wird ausgeführt)
- Ressourcenplanung im Raum (wo wird ausgeführt)
- Zusätzlich: Kommunikation benötigt Zeit
- Optimales Scheduling für Multiprozessoren ist (in fast allen praktisch relevanten Fällen) ein NP-vollständiges Problem
- Darum: Benutzung von Heuristiken



## *Heuristiken für Multiprozessor-Scheduling*

Beispiele:

- Last-Balancierung  
Task wird der am wenigsten ausgelasteten CPU zugewiesen
- Next-Fit für RM-Scheduling  
Task wird der CPU zugewiesen, der zum frühesten Zeitpunkt einen passenden Slot im RM-Schedule frei hat
- Bin-Packing-Algorithmen  
Benutzung bekannter Bin-Packing-Algorithmen mit dem Ziel der Optimierung der Anzahl der benötigten CPUs für einen gegebenen Taskset