EMES: Eigenschaften mobiler und eingebetteter Systeme

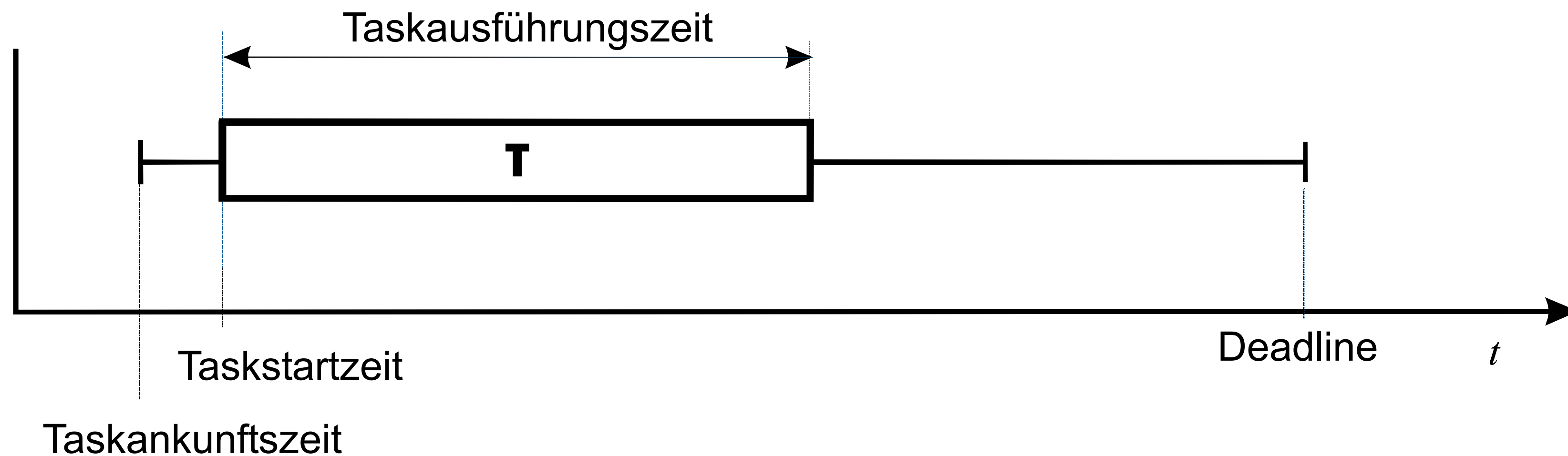
# Weitere Probleme des RT-Schedulings

Dr. Felix Salfner, Dr. Siegmar Sommer

Wintersemester 2010/2011

# Wiederholung: Parameter einer Task

- Taskankunft $r_i$
- Taskstartzeit
- Ausführungszeit $e_i$
- Deadline $D_i$
- Periode $P_i$
- Maß für "Wichtigkeit" (Priorität)

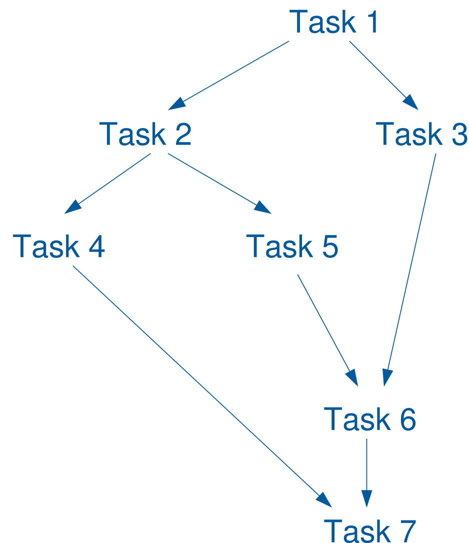


# Bisher nicht angesprochene Probleme

- 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:



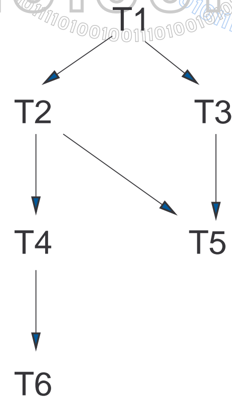
# Algorithmus – Voraussetzungen

- 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: $D_1 \leq D_2 \leq \ldots \leq D_n$

# Algorithmus – Umsetzung

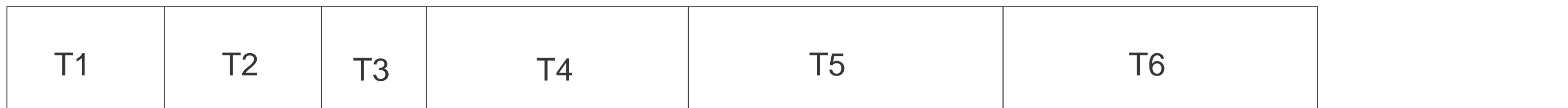
1. Für alle noch nicht geplanten Tasks:
   - $A$ sei die Menge aller noch nicht geplanten Tasks mit folgender Eigenschaft
     - Nachfolger sind schon geplant, oder
     - hat keine Nachfolger
   - Schedule Task $T_k$ im Intervall $[D_k - e_k, D_k]$ ('so spät wie möglich') mit $k = \max\{m|m \in A\}$
2. Bewege alle Tasks so weit wie möglich nach "vorne" unter Bewahrung der bis hier festgelegten Ordnung

# Algorithmus – Beispiel




Schritt 2:

Schritt 3:



Zeit

# Ressourcen

- Tasks können Ressourcen mit exklusivem Zugriff benutzen (beispielsweise Ausgabegeräte)

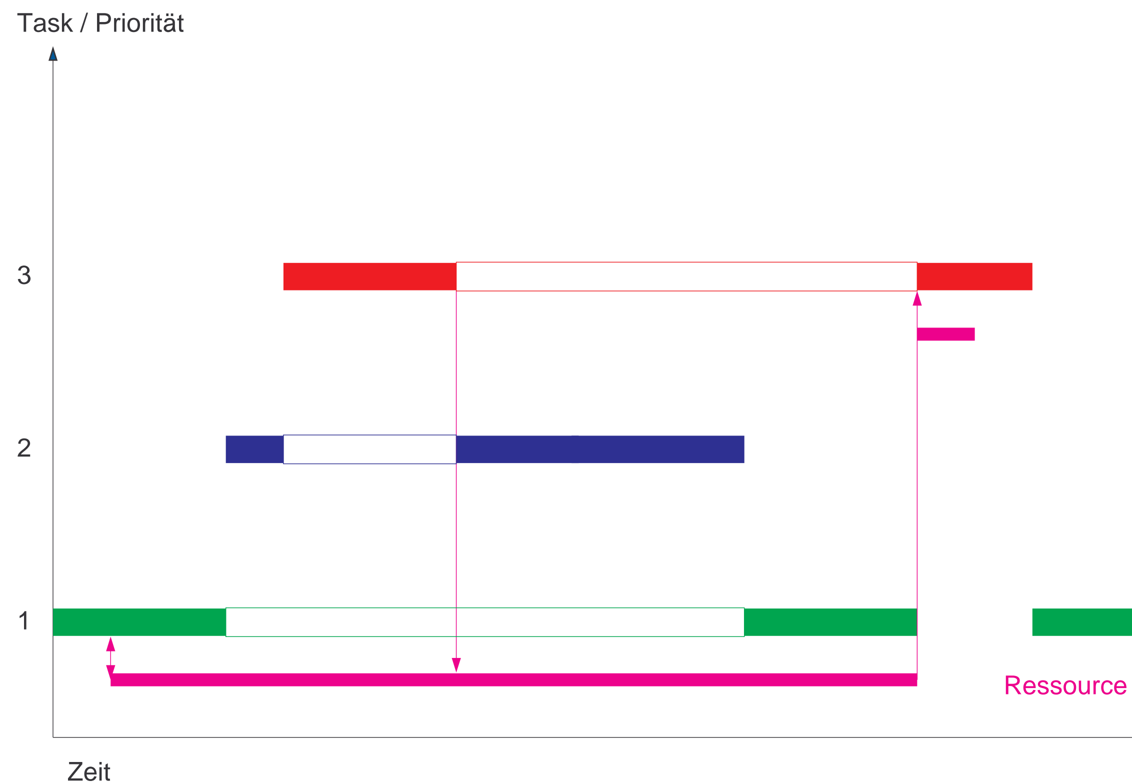
- Zugriffssteuerung per Mutual Exclusion wie üblich, falls nötig

- Für nicht-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 hochpriore Task wartet, und blockiert diese damit



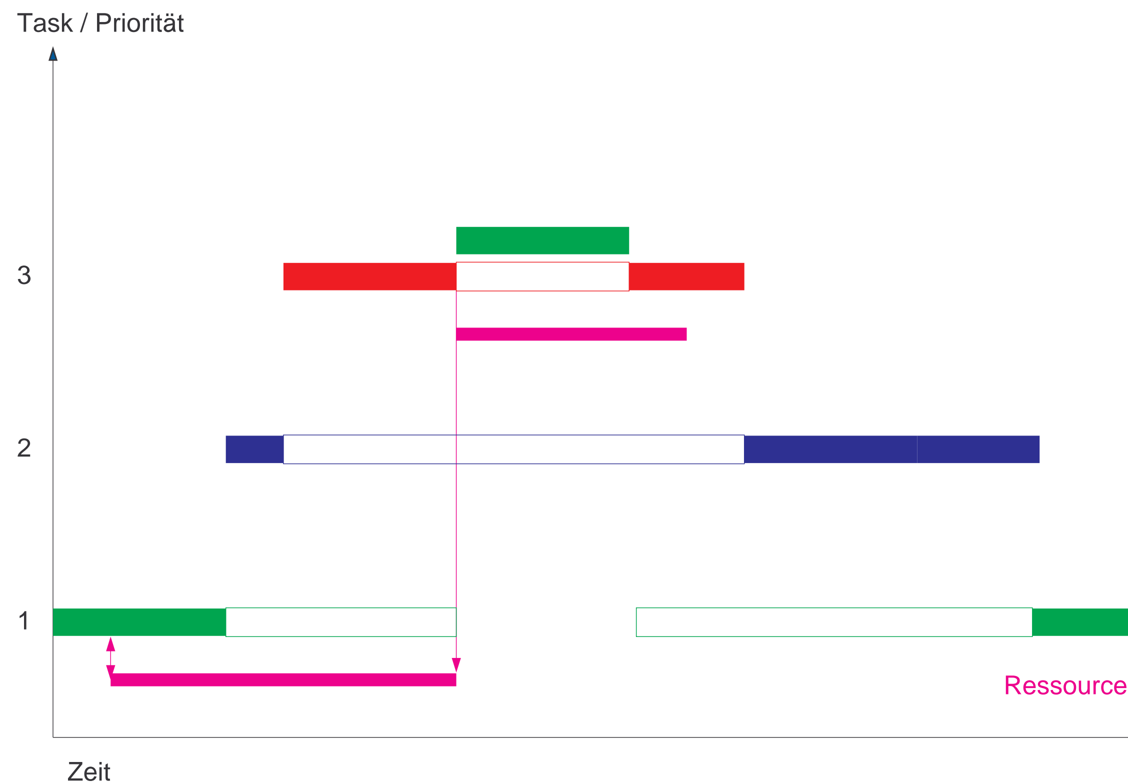
# Priority Inversion: Mars Pathfinder

- What really happened on Mars ? - Glenn E. Reeves
- Pathfinder identifizierte verpasste harte Deadline $\rightarrow$ Reset der gesamten Hard- und Software
- Grund: Priority inversion für eine Semaphore innerhalb von select()
- Vorher nicht entdeckt, da Tasks mittlerer Priorität ungewollt viele Daten verarbeiten mussten
- Von Bodenstation durch Simulation der Aktivitäten ermittelt
- Lösung: Aktivierung von 'priority inheritance' für die Semaphore durch Konfiguration (vxWorks)

# Priority Inheritance

Idee:

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

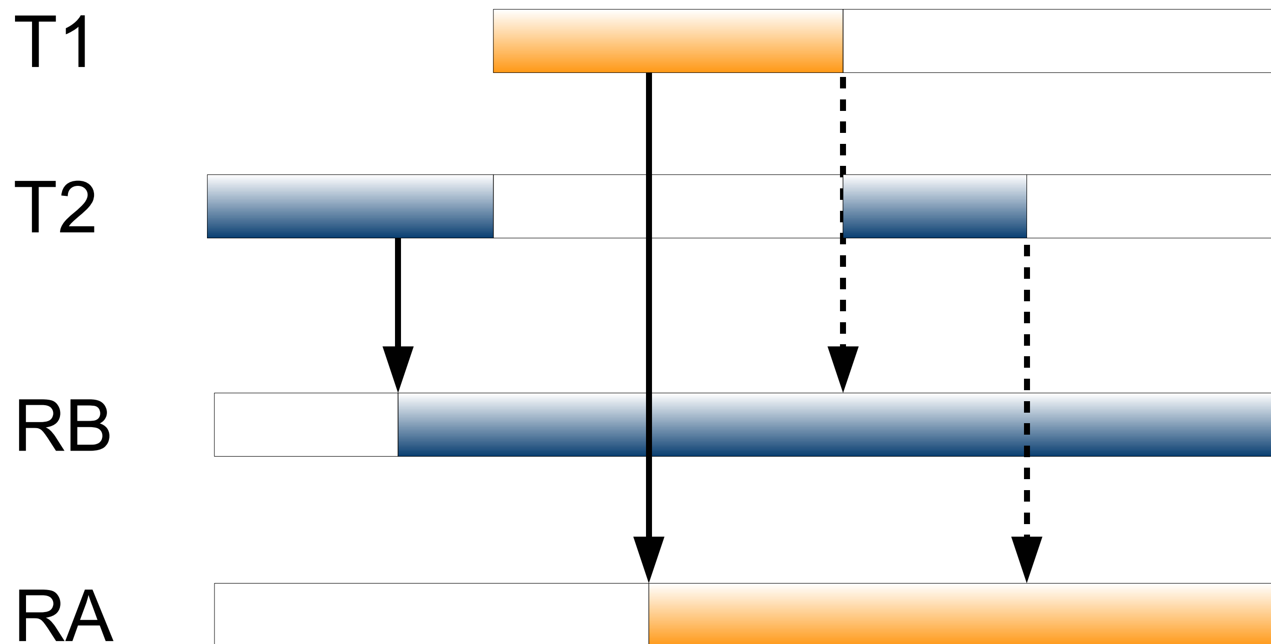


# Probleme mit Priority Inheritance

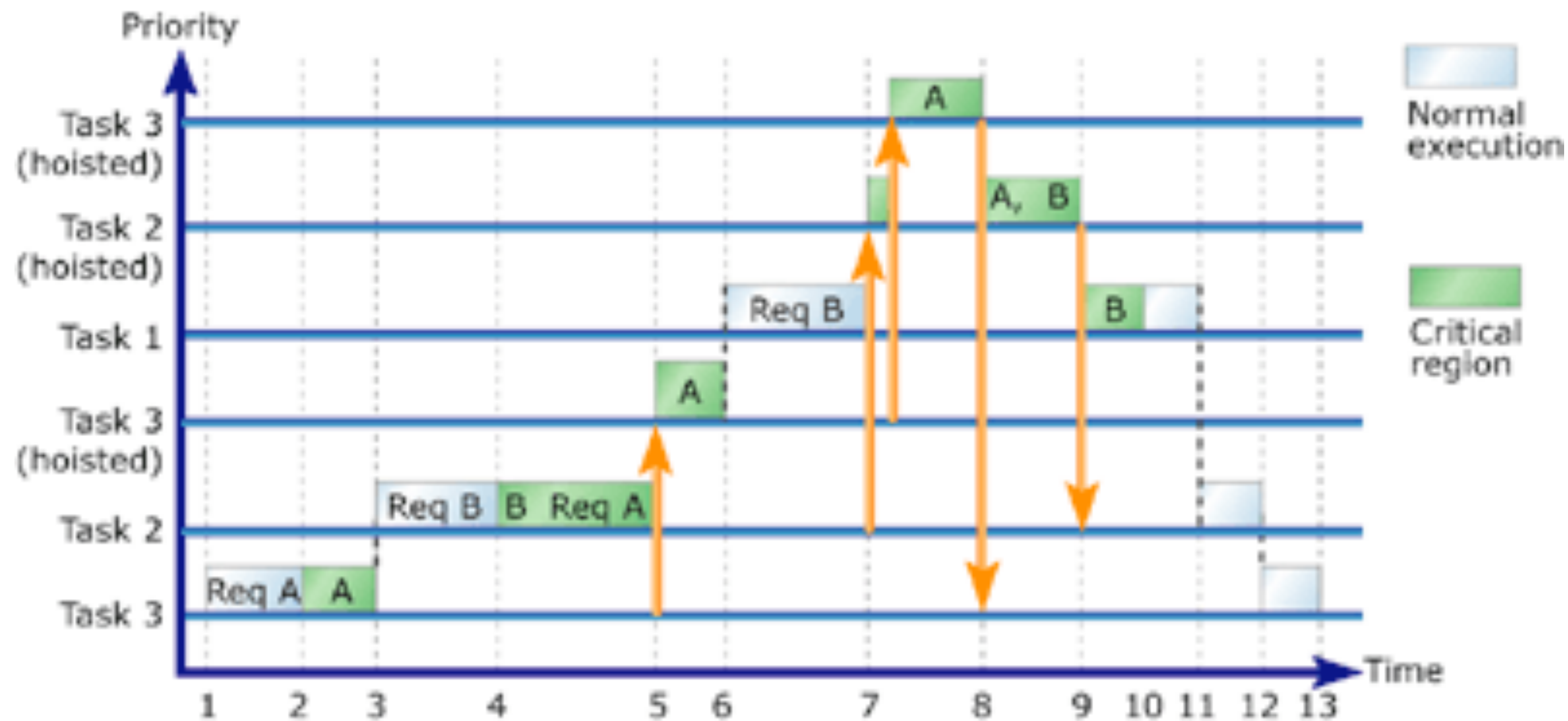
- Ketten von Blockaden können auftreten
- Deadlocks können auftreten
- Hochpriore Prozesse können inakzeptabel lange blockiert werden
- Obere Grenze für die Anzahl der Blockaden kann berechnet werden, liefert einen sehr pessimistischen schlechtesten Fall

Lösung: Priority Ceiling — die Ressourcen erhalten ebenfalls Prioritäten

# Ein Beispiel für ein Deadlock



# Komplexität von Priority Inheritance




(C) Kyle Renwick and Bill Renwick

# Priority Ceiling

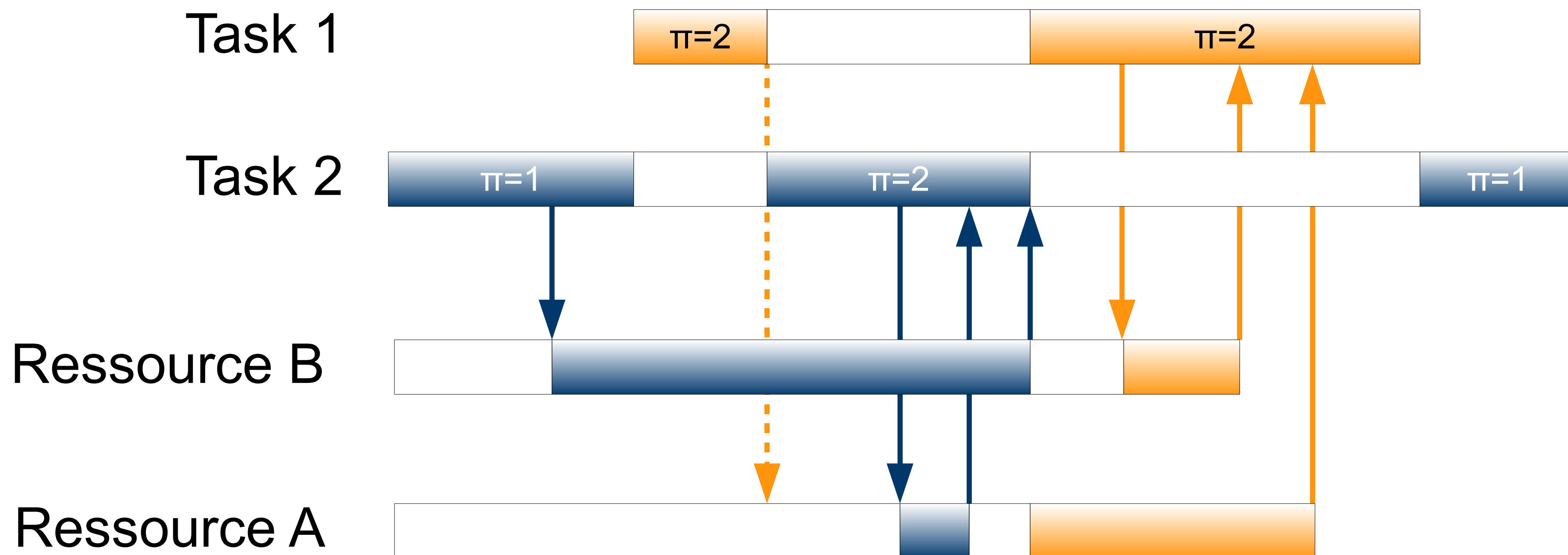
- Voraussetzung: Alle Ressourcenanforderungen vorher bekannt
- Jeder Prozeß hat eine statische Priorität $p_i$ (aus dem Schedulingverfahren) und dynamische Priorität $\pi_i$
- Prioritätsschranke einer Ressource (*ceiling value*) $C_k$: Maximum der Prioritäten der Prozesse, die die Ressource benutzen werden
- Aktuelle Prioritätsschranke des Systems $CS(t)$: Maximum der Prioritätsschranken aller Ressourcen, die zur Zeit in Benutzung sind

## Priority Ceiling

- Scheduling-Regel: Zur Taskankunft ist $p_i = \pi_i$, Scheduling entsprechend $\pi_i$
- Zuweisungsregel: Task $T_i$ möchte auf Ressource $R_j$ zugreifen
  - $T_i$ blockiert, wenn $R_j$ von einer anderen Task benutzt wird
  - Wenn $R_j$ frei ist und $\pi_i > CS(t)$ erhält $T_i$ den Zugriff, sonst
  - wenn $T_i$ die Ressource, welche den aktuellen $CS(t)$ begründet, schon besitzt erhält sie die Ressource, sonst
  - blockiert $T_i$
- Vererbungsregel: Wenn $T_i$ von $T_j$ blockiert wird, dann wird $T_j$ auf $\pi_i$ angehoben, bis es alle Ressourcen mit $C_k \geq \pi_i$ abgibt

# Priority Ceiling Beispiel

- Tasks: Priorität $p_1 = 2$, $p_2 = 1$
- Beide nutzen Ressourcen A und B $\Rightarrow C_A = 2$, $C_B = 2$



# IRIS-Tasks I

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 Steuerqualiät, höherer Komfort für den Nutzer, ...)

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

# IRIS-Tasks II

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

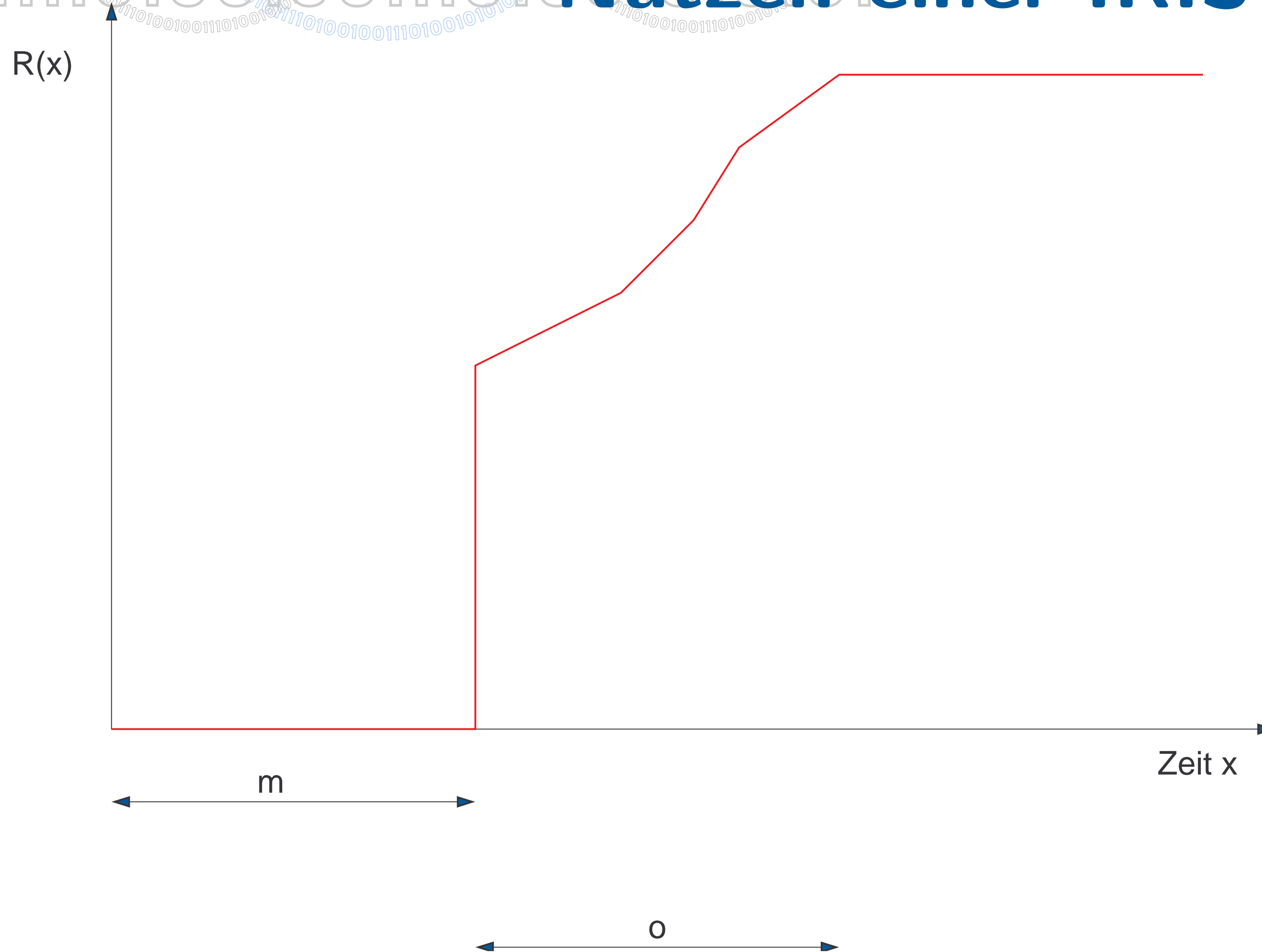
# Nutzen einer IRIS-Task I

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

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

# Scheduling von IRIS-Tasks II

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 "schedulen"
  - Minimieren der Idle-Zeiten durch Verschieben der Tasks und Ausnutzen der freien Zeiten für die Ausführung der optionalen Teile der Tasks

# 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

# Dhall's Effekt

- Sudarshan K. Dhall and C. L. Liu, On a Real-Time Scheduling Problem, Operations Research, Vol. 26, No. 1, Scheduling (Jan. - Feb., 1978), pp. 127-140

- Unterbrechendes Scheduling von periodischen Echtzeit-Task in einem Multiprozessorsystem

- Ziel: Einhalten aller Deadlines mit einer minimalen Anzahl von Prozessoren

# Dhall's Effekt

- Scheduling von $m+1$ periodischen Tasks auf m-Prozessorsystem mit RMS
- Eine Task $T_1$ mit Ausführungszeit $e_1 = 1$ und Periode $P_1 = 1 + x$
- $m$ Tasks mit Ausführungszeit $e_i = 2x$ und Periode $P_i = 1$
  - $T_1$ kann nicht seine Deadline einhalten
  - Für $m \to \infty$ und $x \to 0$ ergibt sich $U \to 0$ und immernoch eine verpasste Deadline.
  - Zuweisung der höchsten Priorität für $T_1$ erlaubt Scheduling
- Ergebnis: Uniprozessor-Algorithmen sind nicht direkt anwendbar

# Multiprozessor-Scheduling

- Zwei unabhängige Ansätze
- Partitionierendes Scheduling: Tasks statisch gruppieren und pro Prozessor schedulen $\rightarrow$ nur noch Partitionierungsproblem
  - Partitionierung orientiert sich am (Uniprozessor-)Schedulingansatz
  - Rucksackproblem, für optimale (RMS-basierte) Lösung mit Behältergrösse $ln2 < g < 1$
  - Utilization-Schranke von 50% theoretisch nachgewiesen
- Globales Scheduling: Globale statische Priorität, Unterbrechung auf allen Prozessoren möglich
  - Neuere Publikationen erhöhen die Utilization-Schranke durch verbesserte Verfahren

# RM-US[US-LIMIT]

- Schwere Tasks mit $U_i > US - LIMIT \rightarrow$ höchste Priorität
- Leichte Tasks mit $U_i < US - LIMIT \rightarrow$ Priorität entsprechend RMS
- Optimales $US - LIMIT = 0.37482$ [Lundberg02]
- Beispiel
  - $T_1$: $e_1 = 1, P_1 = 4 \rightarrow$ RMS Priorität
  - $T_2$: $e_2 = 2, P_2 = 5 \rightarrow$ hohe Priorität
  - $T_3$: $e_3 = 2, P_3 = 7 \rightarrow$ RMS Priorität
  - $T_4$: $e_4 = 4, P_4 = 9 \rightarrow$ hohe Priorität