# Echtzeitsysteme

## Zugriffskontrolle

Lehrstuhl Informatik 4

10. Januar 2013

# Gliederung

# Fragestellungen

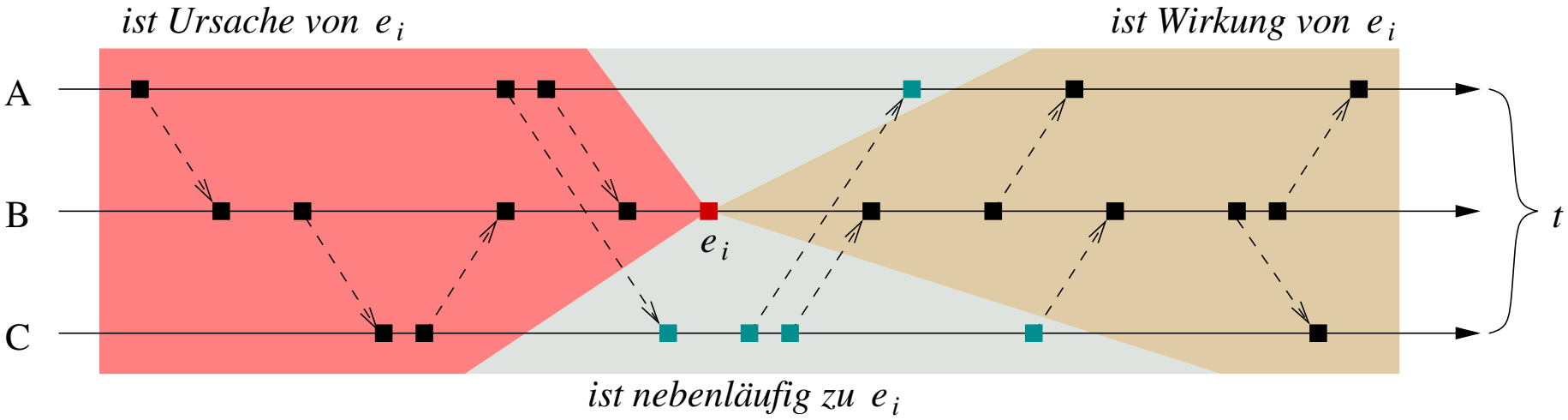
- Wie reguliert man den Eintritt in kritische Abschnitte?
  - Welchen Einfluss haben sie auf das Laufzeitverhalten?
  - Welche Probleme sind mit herkömmlichen Methoden verbunden, den gegenseitigen Ausschluss in kritischen Abschnitten herzustellen?
    - (unkontrollierte) Prioritätsumkehr, Verklemmungen, ...
- Wie löst man diese Probleme?
  - spezielle Synchronisationsprotokolle für Echtzeitsysteme
    - NPCS, Priority Inheritance, Priority Ceiling
  - begrenzter Einfluss kritischer Abschnitte auf den zeitlichen Ablauf
- Fokus: vorranggesteuerte Systeme mit statischen Prioritäten
  - taktgesteuerte Systeme erledigen die Zugriffskontrolle implizit
    - ein geeigneter Ablaufplan stellt einen geordneten Ablauf sicher
  - Vorrangsteuerrung erfordert explizite, konstruktive Maßnahmen
  - Übertragung der Konzepte auf dynamische Prioritäten ist möglich

# Gliederung

# Wiederholung: Kausalordnung

## Nebenläufigkeit als relativistischer Begriff von Gleichzeitigkeit

Relationen „ist Ursache von“, „ist Wirkung von“, „ist nebenläufig zu“:




- ein Ereignis ist nebenläufig zu einem anderen, wenn es im Anderswo des anderen Ereignisses liegt
  - d.h., weder in der Zukunft noch in der Vergangenheit des anderen

Rangfolge $\mapsto$ zeitlich geordnete Ereignisse

Zugriffskontrolle $\mapsto$ Ereignisse im Anderswo

# Koordinierung

## Reihenschaltung nebenläufiger Aktivitäten

Zugriffskontrolle koordiniert gleichzeitige Zugriffe auf gemeinsame aber unteilbare Betriebsmittel $\leadsto$ Synchronisation

- blockierend, wenn das Betriebsmittel nicht die CPU ist
- möglicherweise nicht-blockierend, sonst...

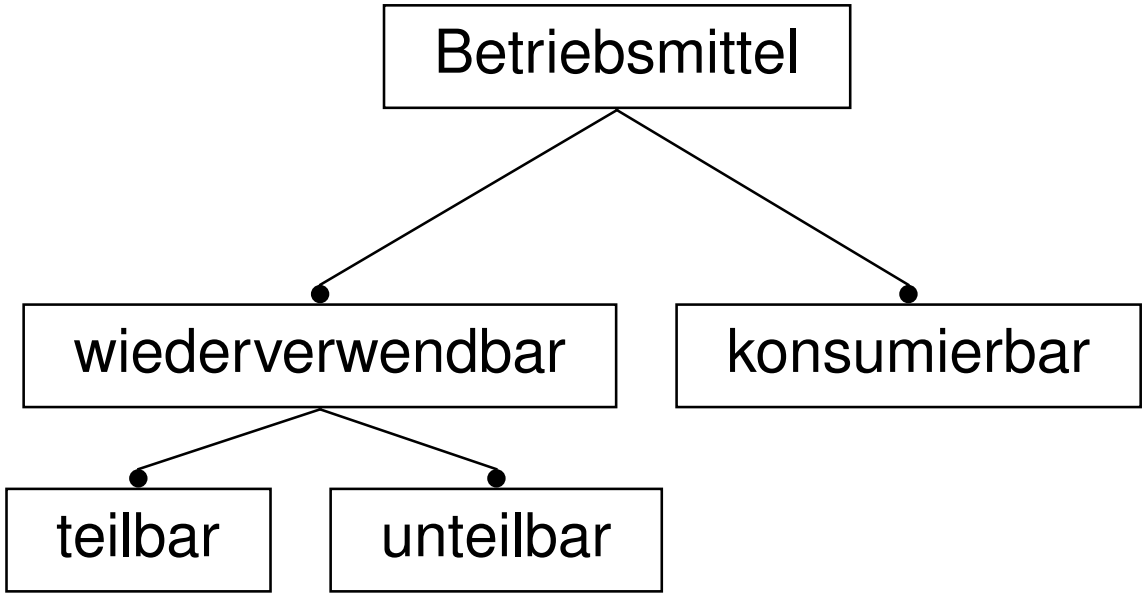
Synchronisation (gr. *sýn*: zusammen, *chrónos*: Zeit) bezeichnet das „Herstellen von Gleichzeitigkeit“

- Koordination der Kooperation und Konkurrenz zwischen Prozessen
- Abgleich von Echtzeituhren (oder Daten) in verteilten Systemen
- Sequentialisierung von Ereignissen entlang einer Kausalordnung
  - z.B. logische Uhren

☞ analytisch/konstruktiv – Einplanung/Implementierung (s. Folie VI/12)

# Konkurrenz

## Betriebsmittel und Betriebsmittelarten




Hardware

CPU, Speicher, Geräte, Signale

Software

Dateien, Prozesse, Seiten, Puffer, Signale, Nachrichten

Wettbewerb um Betriebsmittel (engl. *resource contention*) bezieht sich auf Anzahl und Art eines Betriebsmittels

einseitige Synchronisation $\mapsto$ konsumierbare Betriebsmittel

mehrseitige Synchronisation $\mapsto$ wiederverwendbare Betriebsmittel

- Begrenzung, gegenseitiger Ausschluss

# Konkurrenz (Forts.)

## Serialisierung von Arbeitsabläufen mit begrenzten/unteilbaren Betriebsmitteln

Betriebsmittel, die unteilbar sind, können von gleichzeitigen Prozessen bzw. Arbeitsaufträgen nur nacheinander belegt werden

Vergabe $\mapsto$ das Betriebsmittel sperren und dem Job zuteilen . . . . . P

- beim Versuch, ein gesperrtes Betriebsmittel erneut zu belegen, wird der anfordernde Job blockiert
- der blockierende Job erwartet das Ereignis/Signal zur Freigabe des gesperrten Betriebsmittels, ihm wird die CPU entzogen

Freigabe $\mapsto$ das Betriebsmittel dem Job wieder entziehen . . . . . V

- sollten Jobs die Freigabe dieses Betriebsmittels erwarten, wird es sofort der Wiedervergabe zugeführt; das bedeutet:
  (a) das Betriebsmittel entsperren und alle Jobs deblockieren, die sich dann wiederholt um die Vergabe zu bemühen haben *oder*
  (b) einen Job auswählen und ihm das Betriebsmittel zuteilen
- nur der das Betriebsmittel „besitzende“ Job kann es freigeben

# Konfliktsituation

## Blockierung von Arbeitsaufträgen

Arbeitsaufträge befinden sich untereinander im Konflikt, wenn. . .

- nur eine begrenzte Anzahl gemeinsamer Betriebsmitteln vorrätig ist
- sie unteilbare Betriebsmittel desselben Typs gemeinsam verwenden

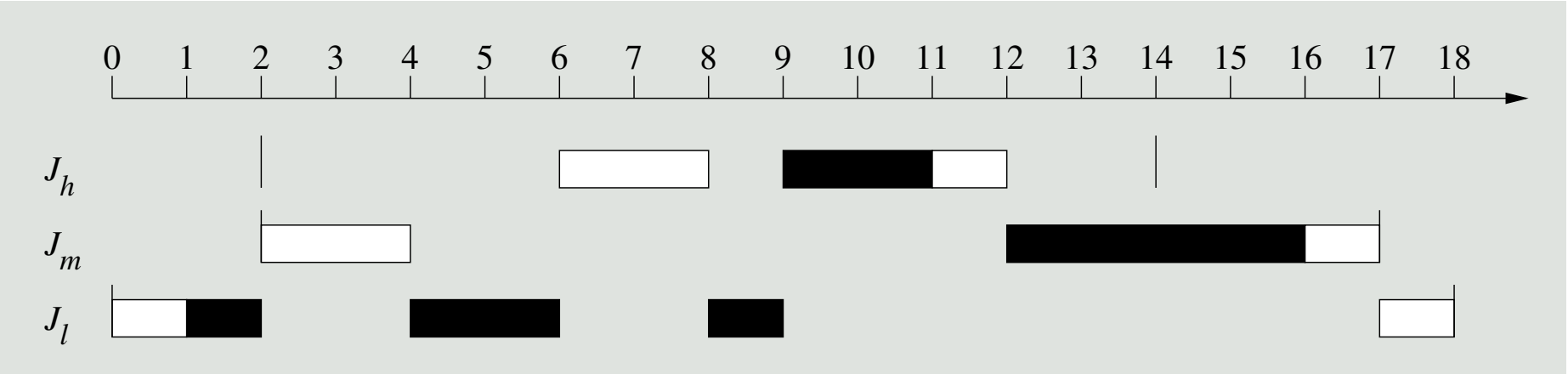
Arbeitsaufträge sind im Streit (engl. *contention*) um ein Betriebsmittel, wenn einer das Betriebsmittel anfordert, das ein anderer bereits besitzt

- der anfordernde Job blockiert und wartet auf die Freigabe des Betriebsmittels durch den Job, der das Betriebsmittel belegt
- der das Betriebsmittel belegende Job löst den auf die Freigabe des Betriebsmittels wartenden Job aus, d.h., deblockiert in wieder

☞ Nutzung begrenzter/unteilbarer Betriebsmittel impliziert Kooperation

# Wettstreit um Betriebsmittel

Beispiel: $J_l \mapsto 6]0, 18]$, $J_m \mapsto 7]2, 17]$, $J_h \mapsto 5]2, 14]$




$J_l$ (niedrige Priorität)

- $t_0$ startet
- $t_1$ belegt $R_i$
- $t_4$ setzt Ausführung fort
- $t_8$ setzt Ausführung fort
- $t_9$ gibt $R_i$ frei $\mapsto J_h$
- $t_{17}$ setzt Ausführung fort
- $t_{18}$ beendet die Ausführung

$J_m$ (mittlere Priorität)

- $t_2$ startet, verdrängt $J_l$
- $t_4$ fordert $R_i$ an, blockiert
- $t_{12}$ belegt $R_i$
- $t_{16}$ gibt $R_i$ frei
- $t_{17}$ beendet die Ausführung

$J_h$ (hohe Priorität)

- $t_6$ startet, verdrängt $J_l$
- $t_8$ fordert $R_i$ an, blockiert
- $t_9$ deblockiert, belegt $R_i$
- $t_{11}$ gibt $R_i$ frei $\rightsquigarrow J_m$
- $t_{12}$ beendet die Ausführung

# Gliederung

# Intervalle von Unverdrängbarkeit

Blockierung, Hemmung (engl. *blocking*)

Beispiel: ein kritischer Abschnitt (engl. *critical section*)

- eine Folge von Anweisungen, deren Ausführung einen gegenseitigen Ausschluss erfordern $\rightsquigarrow$ mehrseitige Synchronisation
  - (a) sich vor Überlappung schützen $\mapsto$ binärer Semaphor
  - (b) sich vor Verdrängung schützen $\mapsto$ Einlastung abschalten
- manche Betriebssysteme unterbinden während der Ausführung von Systemaufrufen die Verdrängung des laufenden Prozesses
  - klassisches UNIX-Modell (`runrun`-Flag) und das von UNIX *look-alikes*
- Job $J_l$ läuft auf solch einer Plattform und tätigt einen Systemaufruf
  - $J_l$ hat eine niedrige Priorität, durchläuft unverdrängbar den Kern
- während des Systemaufrufs, wird Job $J_h$ ereignisbedingt ausgelöst
  - $J_h$ hat eine hohe Priorität, wird eingeplant aber nicht eingelastet
- $J_l$ blockiert bzw. hemmt $J_h$, die Priorität von $J_h$ wird verletzt

☞ Synchronisation ist nicht-funktionale Eigenschaft eines Systemaufrufs

# Nebenläufige Zugriffe auf gemeinsame Betriebsmittel

## Prioritätsumkehr (engl. *priority inversion*)

Prioritätsumkehr [6] ist Folge der Blockierung eines höher priorisierten Jobs durch einen niedriger priorisierten Job

1. der niedrig priorisierte Job durchläuft einen kritischen Abschnitt und wird vom höher priorisierten Job verdrängt
2. der höher priorisierte Job möchte denselben kritischen Abschnitt betreten, wird vom niedrig priorisierten Job jedoch daran gehindert
3. der niedrig priorisierte Job kann weiter ausgeführt werden, obwohl ein höher priorisierter Job wartet

. . . die das Problem verschärfende Variante bringt weitere Jobs ins Spiel:

4. mittel priorisierte Jobs verdrängen den niedrig priorisierten Job und blockieren indirekt den höher priorisierten Job noch länger

☞ kritischer Abschnitt oder Betriebsmittel: Unteilbarkeit ist das Problem

# Anomalie im Laufzeitverhalten

Beispiel: $J_l \mapsto 7(0, 18]$, $J_m \mapsto 5(2, 17]$, $J_h \mapsto 5(2, 14]$; $J_m$ fordert $R_i$ nicht an


$J_l$ (niedrige Priorität)

- $t_0$ startet
- $t_1$ belegt $R_i$
- $t_4$ setzt Ausführung fort
- $t_{11}$ setzt Ausführung fort
- $t_{13}$ gibt $R_i$ frei $\mapsto J_h$
- $t_{16}$ setzt Ausführung fort
- $t_{17}$ beendet die Ausführung

$J_m$ (mittlere Priorität)

- $t_6$ startet, verdrängt $J_l$
- $t_{11}$ beendet die Ausführung

$J_h$ (hohe Priorität)

- $t_2$ startet, verdrängt $J_l$
- $t_4$ fordert $R_i$ an, blockiert
- $t_{13}$ belegt $R_i$
- $t_{14}$ verletzt seinen Termin
- $t_{15}$ gibt $R_i$ frei
- $t_{16}$ beendet die Ausführung

# *What really happened on Mars?*

Prioritätsumkehr beim *Mars Pathfinder* [11, 5]

*bc_sched* $\mapsto$ Task mit höchster Priorität (mit Ausnahme der VxWorks Task „tExec“)

- kontrollierte den Aufbau der Transaktionen über den „1553“-Bus
- dieser Bus koppelte Fahr- und Landeeinheit der Raumsonde

*bc_dist* $\mapsto$ Task mit dritthöchster Priorität

- steuerte die Einsammlung der Transaktionsergebnisse
- Dateneingabe über doppelt gepufferten gemeinsamen Speicher

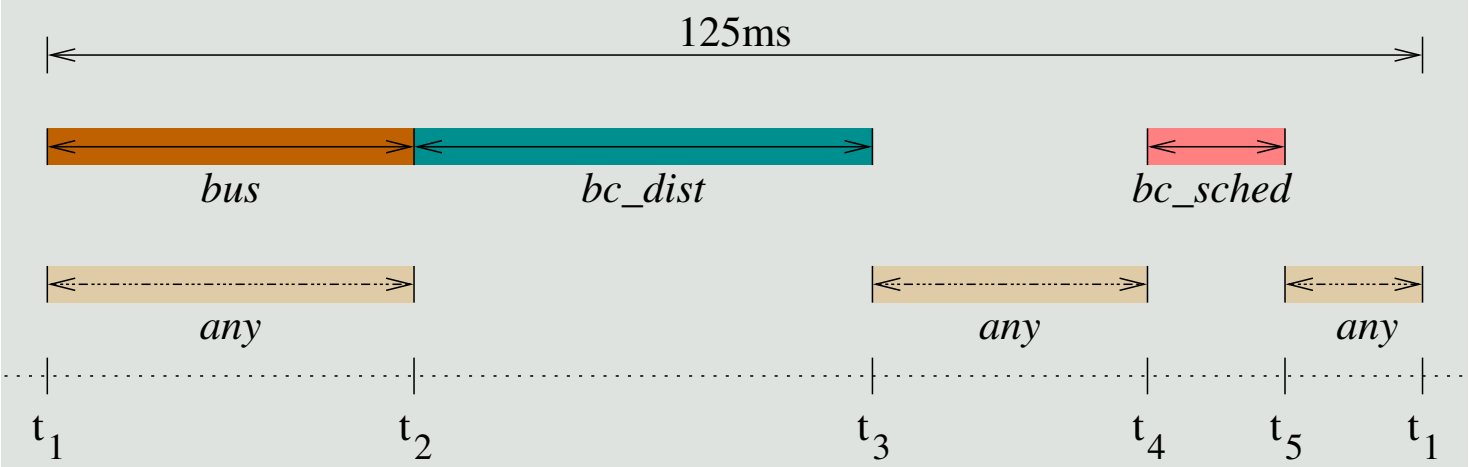
*ASI/MET* $\mapsto$ Task mit sehr niedriger Priorität

- sammelte in seltenen Durchläufen meteorologische Daten ein
- interoperierte mit *bc_dist* (blockierend) auf IPC-Basis

☞ die Hardware gab eine Periodenlänge von 8 Hz (d.h., 125 ms) vor

# *What really happened on Mars?* (Forts.)

## Aufbau eines Buszyklus




- $t_1$ Transaktion startet hardware-kontrolliert an einer 8 Hz Grenze
- $t_2$ Busverkehr ist zur Ruhe gekommen, *bc_dist* wird ausgelöst
- $t_3$ *bc_dist* hat die Datenverteilung abgeschlossen
- $t_4$ *bc_sched* wird ausgelöst, setzt Transaktion für nächsten Buszyklus auf
- $t_5$ *bc_sched* hat seine Aufgabe für diesen Zyklus beendet

☞ Intervalle $[t_1, t_2[, [t_3, t_4[, [t_5, t_1[$ standen u.a. *ASI/MET* zur Verfügung

# *What really happened on Mars?* (Forts.)

## Feste Randbedingung

*bc_dist* muss die Datenverteilung abgeschlossen haben, wenn *bc_sched* ausgelöst wird, um die Transaktion des nächsten Zyklus aufzusetzen:

- stellt *bc_sched* fest, dass *bc_dist* noch nicht abgeschlossen ist, wird ein Total-*reset* durchgeführt
- der *reset* hat die Initialisierung der gesamten Hard- und Software zur Folge, insbesondere den Abbruch aller bodengesteuerten Aktivitäten
  - bereits aufgezeichnete wiss. Daten sind dann zwar gesichert, aber die noch anstehende Tagesarbeit kann nicht mehr vollbracht werden

Kategorie „feste Echtzeit“ (engl. *firm real-time*); zur Erinnerung:

fest (engl. *firm*) auch „stark“

- das Ergebnis einer zu einem vorgegebenen Termin nicht geleisteten Arbeit ist wertlos und wird verworfen
- Terminverletzung ist tolerierbar, führt zum Arbeitsabbruch

# *What really happened on Mars?* (Forts.)

## Fehlersituation

*ASI/MET* (niedrige Priorität) hat *bc_dist* (hohe Priorität) blockiert:

- *ASI/MET* belegte wiederverwendbares, unteilbares Betriebsmittel
  - das von *bc_dist* angefordert wurde, bevor *ASI/MET* es wieder frei gab
- im weiteren Verlauf verdrängten Tasks mittlerer Priorität *ASI/MET*
  - dadurch verlängerte sich die Blockierungszeit für *bc_dist*
  - als Folge war *bc_dist* noch nicht abgeschlossen als *bc_sched* startete
- *bc_sched* stellte die Zeitverletzung fest und löste einen *reset* aus

Fehlererkennung und -beseitigung:

- die Semaphorinitialisierung war in VxWorks falsch eingestellt
- sie wurde bodengesteuert (durch ein Skriptprogramm) korrigiert
  - der Semaphor wurde auf Prioritätsvererbung umgestellt

# Synchronisation *Considered Harmful*

## Prioritätsorientierte Einplanung

Prioritätsumkehr (s. Folie VII/13), mögliches Phänomen abhängiger Tasks, vorausgesetzt zwei Bedingungen sind erfüllt:

1. eine Task hoher Priorität wartet auf eine Task niedriger Priorität $\mapsto$ gegenseitiger Ausschluss
   - das ist die (normale) Prioritätsumkehr
2. die Task niedriger Priorität wird von einer oder mehrerer Tasks mittlerer Priorität verdrängt
   - auch als unkontrollierte Prioritätsumkehr bezeichnet

Lösungsansätze sind, sofern blockierende Synchronisation in der jeweils gegebenen Situation nicht vermieden werden kann:

- Verdrängungssteuerung ☞ S. VII/21
- Prioritätsvererbung ☞ S. VII/26
- Prioritätsobergrenzen ☞ S. VII/30

# Gliederung

# Verdrängung zeitweise unterbinden

Verdrängungsfreie kritische Abschnitte (engl. *non-preemptive critical sections*, NPCS)

Arbeitsaufträge werden für die Gesamtzeit der Belegung von (unteilbaren) Betriebsmitteln nicht von anderen Arbeitsaufträgen verdrängt

- die Benutzung der Betriebsmittel kontrolliert ein Monitor [3, 4]
  - *kernelized monitor* [8]

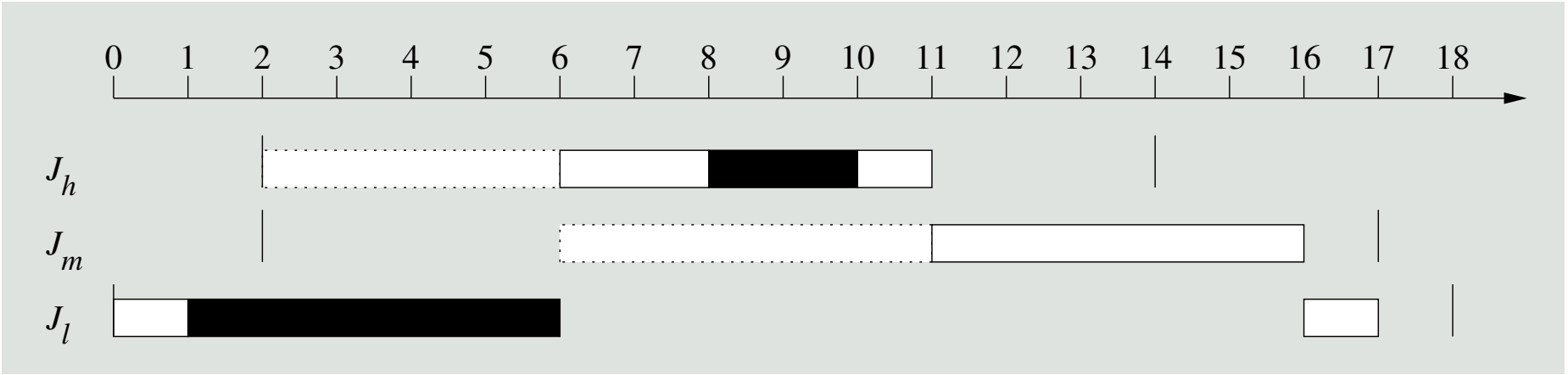
  Eintrittsprotokoll $\mapsto$ Verdrängung abwehren
  - ausgelöste Jobs einplanen, aber nicht einlasten

  Austrittsprotokoll $\mapsto$ Verdrängung wieder zulassen
  - höher priorisierte Jobs (nachträglich) einlasten

- belegt ein Job ein Betriebsmittel, läuft er unverdrängbar weiter
  - verklemmungsfreies Verfahren durch Verklemmungsvorbeugung
    - engl. *deadlock prevention*

# *Kernelized Monitor*

Beispiel (S. VII/14): $J_l \mapsto 7(0, 18]$, $J_m \mapsto 5(2, 17]$, $J_h \mapsto 5(2, 14]$




$J_l$ (niedrige Priorität)

- $t_0$ startet
- $t_1$ belegt $R_i$ unverdrängbar
- $t_6$ gibt $R_i$ frei $\mapsto J_h$
- $t_{16}$ setzt Ausführung fort
- $t_{17}$ beendet die Ausführung

$J_m$ (mittlere Priorität)

- $t_6$ wird ausgelöst
- $t_{11}$ startet
- $t_{16}$ beendet die Ausführung

$J_h$ (hohe Priorität)

- $t_2$ wird ausgelöst
- $t_6$ startet
- $t_8$ belegt $R_i$ unverdrängbar
- $t_{10}$ gibt $R_i$ frei
- $t_{11}$ beendet die Ausführung

# Blockierungszeit

## Feste obere Schranke

Verzögerungen nebenläufiger Arbeitsaufträge durch die Zugriffskontrolle sind nach oben begrenzt

- die feste obere Schranke $b^{np}$ bestimmt sich aus der größten WCET aller kritischen Abschnitte aller niedriger priorisierten Jobs: $\max(cs)$
- höher priorisierte Jobs werden schlimmstenfalls einmal durch einen niedriger priorisierten Job blockiert

NPCS verzögert eine periodische Task $T_i$ von $n$ periodischen Tasks im taktweisen Betrieb um $b_i^{np} = \max_k(cs_k)$, für $i + 1 \leq k \leq n$:

*fixed-priority* bei Abarbeitung nach absteigender Priorität

*dynamic-priority* z.B. EDF: Jobs in $T_i$ mit relativem Termin $D_i$ können nur durch Jobs mit längeren relativen Terminen als $D_i$ blockiert werden ☞ $i < j$ wenn $D_i < D_j$

# Pragmatischer Ansatz

Effektiv, bei vergleichsweise geringem Aufwand

Vorteil: erfordert kein *à priori* Wissen über Betriebsmittelanforderungen

- beugt unkontrollierter Prioritätsumkehr vor
  - $J_h$ blockiert nur wenn bei Auslösung $J_l$ bereits das Betriebsmittel hält
  - beendet $J_l$ seinen kritischen Abschnitt, sind alle Betriebsmittel frei
  - Jobs geringerer Priorität als $J_h$ können ihm diese nicht streitig machen
- beugt Verklemmung (engl. *deadlock*) vor, da Nachforderungen von Betriebsmitteln implizit unteilbar geschehen werden
  - eine notwendige Verklemmungsbedingung [9, VIII-60] wird entkräftet
  - genauer: der „*hold and wait*“ Fall kann nicht eintreten
- einfach zu implementieren; ein gutes Verfahren, wenn. . .
  - alle Belegungszeiten aller Betriebsmittel kurz sind
  - die meisten Jobs im Konflikt zueineinander stehen

☞ eignet sich für Systeme mit fester und dynamischer Priorität

# Pragmatischer Ansatz mit Schönheitsfehlern

Alternativen, sofern bestimmte Voraussetzungen gegeben sind

Nachteil: höher priorisierte Jobs können durch niedriger priorisierte Jobs blockiert werden, obwohl zwischen ihnen kein Konflikt besteht

- im Beispiel (S. VII/22) wird $J_m$ durch $J_l$ blockiert, obwohl beide Jobs nicht im gegenseitigen Ausschluss zueinanden stehen

Verbesserungsmöglichkeiten...

- so Verklemmungen nicht auftreten können oder durch eine andere Technik vorgebeugt oder vermieden werden:
  - den ein Betriebsmittel haltenden Job für die restliche Belegungszeit ggf. auf die Priorität des jeweils anfordernden Jobs hochsetzen
  - er wird durch Prioritätsvererbung (S. VII/26) „beschleunigt“
- so Betriebsmittelanforderungen *à priori* bekannt sind:
  - der ein Betriebsmittel haltende Job läuft mit der höchsten Priorität aller Jobs, die das Betriebsmittel beanspruchen
  - das Betriebsmittel besitzt eine Prioritätsobergrenze (S. VII/30)

# Priorität zeitweise erhöhen

Wechsel zwischen zugewiesene und aktuelle (geerbte) Priorität

Arbeitsaufträge werden für die Restzeit der Belegung von (unteilbaren) Betriebsmitteln durch andere Arbeitsaufträge höher priorisiert

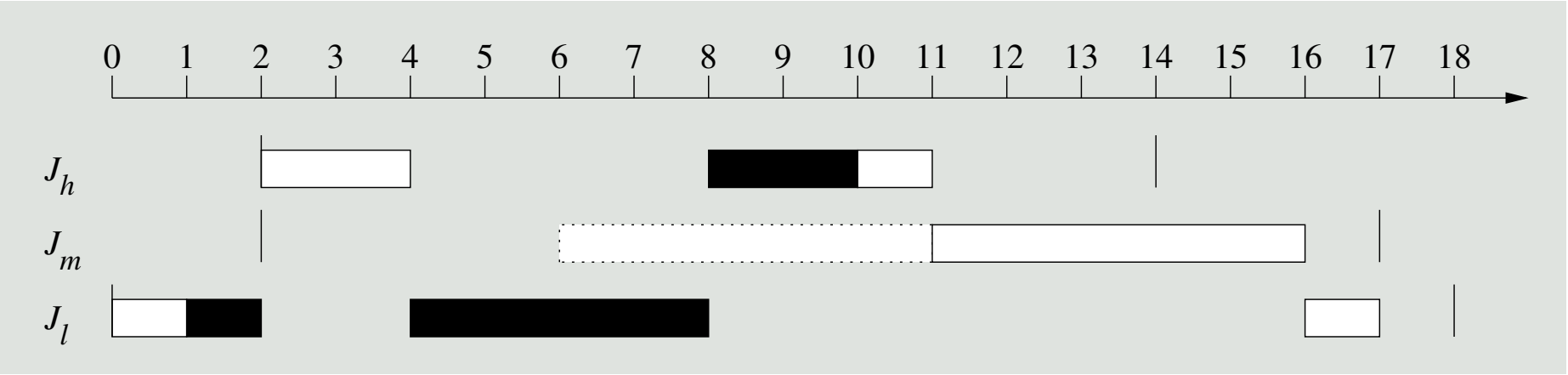
- fordert ein Job ein gesperrtes Betriebsmittel an, vererbt er seine Priorität an den das Betriebsmittel haltenden Job
  - der anfordernde Job hat zu dem Zeitpunkt die höchste Priorität
    - er hat den das Betriebsmittel haltenden Job (indirekt) verdrängt
  - die Priorität des das Betriebsmittel haltenden Jobs wird erhöht
- gibt der Job das durch ihn gesperrte Betriebsmittel frei, nimmt er die ihm ursprünglich zugewiesene Priorität wieder an
  - der das Betriebsmittel freigebende Job wird ggf. sofort verdrängt
  - der auf die Freigabe wartende Job wird ggf. sofort eingelastet

Prioritätsumkehr wird nicht wirklich vermieden, jedoch entschärft:

- wie bei NPCS behalten niedriger priorisierte Jobs die CPU zugeteilt, obwohl höher priorisierte Jobs auf Zuteilung der CPU warten

# *Priority Inheritance*

Beispiel (S. VII/14): $J_l \mapsto 7(0, 18]$, $J_m \mapsto 5(2, 17]$, $J_h \mapsto 5(2, 14]$




$J_l$ (niedrige Priorität)

- $t_0$ startet
- $t_1$ belegt $R_i$
- $t_4$ läuft mit Priorität $J_h$
- $t_8$ gibt $R_i$ frei $\mapsto J_h$
- $t_{16}$ läuft mit alter Priorität
- $t_{17}$ beendet die Ausführung

$J_m$ (mittlere Priorität)

- $t_6$ wird ausgelöst
- $t_{11}$ startet
- $t_{16}$ beendet die Ausführung

$J_h$ (hohe Priorität)

- $t_2$ startet
- $t_4$ fordert $R_i$ an $\mapsto J_l$
- $t_8$ belegt $R_i$
- $t_{10}$ gibt $R_i$ frei
- $t_{11}$ beendet die Ausführung

# Transitive Blockierung

## Nachforderung unteilbarer Betriebsmittel

Zugriffskontrolle durch Prioritätsvererbung bedeutet zunächst zweierlei:

direkte Blockierung eines höher priorisierten Jobs ($J_h$), der ein gesperrtes Betriebsmittel anfordert, das ein niedriger priorisierter Job ($J_l$) belegt

Blockierung durch Vererbung (engl. *inheritance blocking*) eines nicht im gegenseitigen Ausschluss befindlichen höher priorisierten Jobs ($J_m$)

- der einen Job ($J_l$) gemäß dessen „Altpriorität“ verdrängen würde
- dies jedoch wegen dessen aktuellen (geerbten) Priorität nicht kann

Blockierungen wirken ggf. transitiv: geschachtelte kritische Abschnitte

1. $J_l$ startet zuerst, belegt $R_1$ und wird von $J_m$ verdrängt
2. $J_m$ belegt $R_2$ und wird von $J_h$ verdrängt
3. $J_h$ fordert $R_2$ an und vererbt seine Priorität an $J_m$
4. $J_m$ läuft weiter, fordert $R_1$ an und vererbt „seine“ Priorität an $J_l$

# Blockierungszeit

Feste obere Schranken, die kaskadenartig zur Wirkung kommen können

Situation des schlimmsten Falls: höher priorisierter Job $J_i$ benötigt $n > 1$ Betriebsmittel, steht mit $k > 1$ niedriger priorisierten Jobs im Konflikt

- der höher priorisierte Job kann $\min(n, k)$-mal blockiert werden
- jeweils für die Dauer der WCET des äußersten kritischen Abschnitts
- die Blockierungszeit ist also maximal $b_i^{np} = \min(n, k) \max_l(cs_l)$
  - $cs_l$ sind kritische Abschnitte von Jobs niedrigerer Priorität
  - das ist pessimistisch: unterschiedliche Jobs haben schließlich auch unterschiedlich lange kritische Abschnitte

GAU: $n > 1$ Betriebsmittel; $k > 1$ Jobs; $J_i$ Vorrang vor $J_j$, wenn $i < j$

1. $J_k$ startet zuerst, belegt $R_n$; $J_{k-1}$ verdrängt $J_k$, belegt $R_{n-1}$; ...
2. $J_1$ verdrängt $J_2$, belegt $R_1$
3. $J_0$ verdrängt $J_1$, fordert $R_i$ an in der Reihenfolge $i = 1, 2, 3, \ldots, n$

# Priorität zeitweise deckeln

## Vorabwissen über Arbeitsaufträge und Betriebsmittel

Prioritätsobergrenze (engl. *priority ceiling*) eines Betriebsmittels $R_i$ ist die höchste Priorität aller Arbeitsaufträge, die $R_i$ benötigen

- die aktuelle Prioritätsobergrenze des Systems gleicht der höchsten Prioritätsobergrenze der zur Zeit belegten Betriebsmittel
  - $\hat{\Pi}(t)$, in Abhängigkeit vom betrachteten Zeitpunkt $t$
- ist kein unteilbares Betriebsmittel belegt, existiert die aktuelle Prioritätsobergrenze (theoretisch) nicht
  - sie ist dann niedriger als die niedrigste Priorität aller Jobs
- die jeweiligen Werte sind für alle Betriebsmittel im Voraus bekannt

Arbeitsaufträge können während ihrer Ausführung (d.h., bei Anforderung eines gesperrten Betriebsmittels) eine durch $x$ begrenzte Priorität erben

- wenn sie ein Betriebsmittel mit Prioritätsobergrenze $x$ benötigen

☞ Prioritititätsobergrenzen sind eine Variante von Prioritätsvererbung

# Betriebsmittelvergabe und Prioritätsvererbung

## Vorabwissen über Arbeitsaufträge und Betriebsmittel

Vergabe von Betriebsmittel $R$ zum Zeitpunkt $t$ an Arbeitsauftrag $J$ hängt ab vom Zustand von $R$ und von der aktuellen Priorität $\pi(t)$ von $J$:

belegt $\mapsto$ $R$ ist gesperrt, $J$ blockiert

frei $\mapsto$ $R$ wird $J$ zugeteilt und gesperrt, falls...

1. $\pi(t) > \hat{\Pi}(t)$: $J$'s Priorität ist größer als die Obergrenze
2. $\pi(t) \leq \hat{\Pi}(t)$: $J$ ist ein Job, der zum Zeitpunkt $t$ mindestens ein Betriebsmittel mit Prioritätsobergrenze $\hat{\Pi}(t)$ hält

anderenfalls bleibt $R$ frei und $J$ blockiert (S. VII/32)

Prioritätsvererbung findet (auch hier) nur statt, wenn $J$ suspendiert wird:

- $J_l$, der $J$ blockiert, erbt die aktuelle Priorität $\pi(t)$ von $J$
  - $J_l$ lässt sich auch bestimmen, falls $R$ frei bleibt und $\pi(t) \leq \hat{\Pi}(t)$

  $\rightsquigarrow$ $J_l$ ist genau der Job, der eine Ressource $R$ mit Obergrenze $\hat{\Pi}(t)$ hält
- $J_l$ behält diese Priorität, bis er alle Betriebsmittel freigibt, deren Prioritätsobergrenze größer oder gleich $\pi(t)$ ist
  - er nimmt dann wieder die Priorität bei der Betriebsmittelzuteilung an

# Verklemmungsvorbeugung

Entkräftung der hinreichenden Bedingung [9, VIII-60]: zirkulares Warten

Betriebsmittelvergabe kontrolliert durch Prioritätsobergrenzen ist weniger „gefräßig“ (engl. *greedy*), als bloße Prioritätsvererbung

- die Anforderung von $J$ kann zurückgewiesen werden, obwohl das angeforderte Betriebsmittel $R$ frei ist
- dies ist der Fall, wenn die durch die Menge von Prioritätsobergrenzen definierte (ansteigende) lineare Ordnung verletzt werden sollte
  - $\pi(t) \leq \hat{\Pi}(t)$ trifft zu und $J$ hält kein Betriebsmittel mit $\hat{\Pi}(t)$
  - d.h., die direkte/indirekte Priorität von $J$ durchbricht die Ordnung
- alle Betriebsmittel des Systems sind linear geordnet aufgestellt

Blockierung durch Prioritätsobergrenzen wird auch als Aufhebungssperre (engl. *avoidance blocking*) bezeichnet

- da implizit Kosten anfallen, um auch Verklemmungen vorzubeugen

# Blockierungszeit

Zugriffskontrolle durch Prioritätsobergrenzen impliziert drei Arten der Blockierung nebenläufiger Arbeitsaufträge:

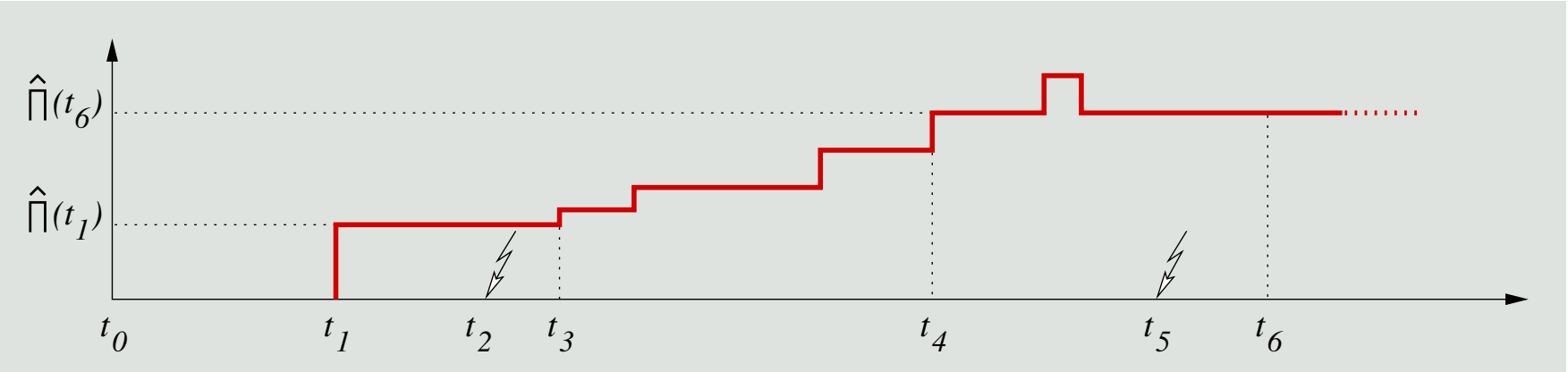
1. direkte Blockierung,
2. Blockierung durch Vererbung,
3. Blockierung durch Aufhebungssperre

} Prioritätsvererbung

Effekt von 3. ist, dass jeder Arbeitsauftrag höchstens einmal blockiert und dass eine Blockierung nicht transitiv ist [10]

- die Blockierungszeit ist begrenzt durch die größte WCET aller kritischen Abschnitte aller niedriger priorisierten Jobs
- unabhängig von der Anzahl der im Konflikt stehenden Jobs
  - (a) wenn ein Job blockiert, dann nur durch höchstens einen Job
  - (b) ein Job, der einen anderen blockiert, wird selbst nicht blockiert
- Blockierungszeit $b_i^{np} = \max_k(cs_k)$ analog zu NPCS (s. Folie VII/23)
  - nur nieder-priore Jobs $J_{i+1}, \ldots, J_n$ blockieren $J_i$: $i + 1 \leq k \leq n$
  - der Job $J_i$ blockiert aber weniger unbeteiligte Jobs höherer Priorität

# Blockierungszeit (Forts.)




$J_l$ (niedrige Priorität)

$t_0$ startet

$t_1$ belegt $R_x \rightsquigarrow \hat{\Pi}(t_1)$

$J_m$ (mittlere Priorität)

$t_2$ verdrängt $J_l$

$t_3$ belegt $R_y$, $\pi_m > \hat{\Pi}(t_1)$

$J_h$ (hohe Priorität)

$t_5$ verdrängt $J_m$

$t_6$ blockiert

- direkte Blockierung von $J_h$ durch $J_l$ ist nicht möglich: (a) sonst wäre $\hat{\Pi}(t_3)$ wenigstens $\pi_h$ und (b) $J_m$ könnte $R_y$ überhaupt nicht belegen
- werden alle ab $t_2$ angeforderten Betriebsmittel zum Zeitpunkt $t_6$ nur von $J_m$ belegt, kann $J_h$ nur durch $J_m$ blockiert werden
- würde ein Job $J_k$ bei $t_4$ Betriebsmittel $R_z$ belegen, wäre $J_h$ aus demselben Grund nicht mehr durch $J_m$ blockierbar, wie $J_h$ nicht durch $J_l$ blockierbar ist

# Vereinfachung durch Stapelorientierung

Stapelbezogene Einplanung (engl. *stack-based scheduling*)

Ausgangspunkt ist die stapelorientierte Einplanung von Prozessen, so dass $N$ Prozesse einen Stapel gemeinsam nutzen können [1, 2]

- nicht immer ist es möglich, jeden Job durch einen eigenen Faden zu repräsentieren bzw. mit einem eigenen Stapel zu versehen
  - wenn die Jobanzahl zu hoch und/oder der Speicherplatz zu gering ist
- gemeinsame Nutzung desselben Stapels setzt voraus, dass kein Job bei Anforderung eines gemeinsamen Betriebsmittels blockiert
  - sonst droht die Überschreibung der Stapelbereiche anderer Jobs
- die Jobs dürfen ihre Ausführung niemals selbst aussetzen, sie dürfen jedoch von höher priorisierten Jobs verdrängt werden
  - oben auf dem Stapel läuft immer der Job mit der höchsten Priorität
  - die logische Konsequenz, wenn Selbstaussetzung ausgeschlossen ist

☞ stapelbezogene Prioritätsobergrenzen (*stack-based priority ceiling*)

# Stapelbezogene Grenzprioritäten

## Regeln

1. Aktualisierung der Grenzpriorität $\hat{\Pi}(t)$ (s. Folie VII/30)
   - erfolgt mit jeder Vergabe/Freigabe von Betriebsmitteln
   - sind alle Betriebsmittel frei, gibt es keine Grenzpriorität
2. Einplanung und Einlastung von Jobs
   - nach erfolgter Auslösung muss ein Job ggf. solange warten, bis die ihm zugewiesene Priorität die Grenzpriorität übersteigt
   - Jobs werden jeweils entsprechend ihrer zugewiesenen Priorität und verdrängend ausgeführt
3. Zuteilung eines Betriebsmittels
   - erfolgt sofort, wann immer ein Job das Betriebsmittel anfordert

Jobs blockieren nicht, indem ihnen die Betriebsmittelzuteilung nach Ausführungsbeginn verweigert werden würde

- im Gegensatz zu „normalen“ Prioritätsobergrenzen (s. Folie VII/31)

# Stapelbezogene Grenzprioritäten (Forts.)

## Implikationen

Verklemmungen nebenläufig ausgeführter Arbeitsaufträge sind durch eine indirekte Methode zur Verklemmungsvorbeugung[1] ausgeschlossen

(a) beginnt ein Job seine Ausführung, sind alle von ihm im weiteren Verlauf benötigten Betriebsmittel frei
- sonst wäre die Grenzpriorität größer oder gleich seiner Priorität
- in dem Fall wäre jedoch die Einlastung des Jobs verzögert worden

(b) bei Verdrängung eines Jobs sind alle von ihm benötigten Betriebsmittel (noch oder bereits wieder) frei
- sonst hätte die Grenzpriorität eine Verdrängung unterbunden
- der verdrängende Job wird also immer komplett durchlaufen können

(c) auf ein benötigtes Betriebsmittel kann direkt zugegriffen werden

[1] zirkulares Warten wird vorgebeugt (siehe auch S. VII/32)

# Grundmodell und Stapelorientierung

## Feste vs. dynamische Priorität (vergleiche S. IV-1/14)

feste Priorität $\mapsto$ einfach, wegen *à priori* Wissen

dynamische Priorität $\mapsto$ Prioritäten periodischer Aufgaben ändern sich, während die von ihnen beanspruchten Betriebsmittel konstant bleiben:

- Grenzprioritäten der Betriebsmittel ändern sich mit der Zeit
- Aktualisierung der Grenzprioritäten bei jeder Auslösung eines Jobs
  1. dem ausgelösten Job zum Ereigniszeitpunkt eine Priorität zuweisen
     - relativ zu den anderen bereits eingeplanten/laufbereiten Jobs
     - (dynamische) prioritätsorientierte Einplanung je nach Verfahren
  2. Grenzprioritäten aller Betriebsmittel aktualisieren
     - auf Basis der neuen Taskprioritäten
  3. Grenzpriorität des Systems aktualisieren
     - auf Basis der neuen Grenzprioritäten der Betriebsmittel
- für, auf Jobebene, statische oder dynamische Prioritäten

# Gliederung

# Weitere Lockerung der Restriktionen

Weitere Lockerung von A5 und A7 zugunsten mehrseitiger Synchronisation

Mathematische Ansätze zur Analyse periodischer Echtzeitsysteme schränken solche Systeme häufig stark ein:

A1 ~~Alle Aufgaben sind periodisch.~~

A2 ~~Alle Arbeitsaufträge können an ihren Auslösezeitpunkten eingeplant und ausgeführt werden.~~

A3 Termine und Perioden sind identisch.

A4 Kein Arbeitsauftrag gibt die Kontrolle über den Prozessor ab.

A5 ~~Alle Aufgaben sind unabhängig voneinander, d.h. die einzige gemeinsame Ressource ist die CPU und es existieren keine Einschränkungen hinsichtlich der Auslösezeiten der Arbeitsaufträge.~~

A6 Der Overhead durch Unterbrechungen, Ablaufplanung oder Verdrängung ist vernachlässigbar.

A7 ~~Alle Aufgaben verhalten sich voll-präemptiv.~~

# Fadensynchronisation $\rightsquigarrow$ Blockierungszeit

Die Blockierungszeit verzögert die Fertigstellung von Arbeitsaufträgen

- die Blockierungszeit direkt hängt vom Synchronisationsprotokoll ab

NPCS (s. Folie VII/23) $b_i^{np} = \max_{i+1 \leq k \leq n}(cs_k)$

Priority Inheritance (s. Folie VII/29) $b_i^{np} = \min(n, k) \max_{i+1 \leq l \leq n}(cs_l)$

Priority Ceiling (s. Folie VII/33) $b_i^{np} = \max_{i+1 \leq k \leq n}(cs_k)$

- dies heißt bei einer Aufgabe $T_i$ mit der Blockierungszeit $b_i^{np}$
  - für die Bestimmung der Antwortzeit:

$$\omega_i(t) = e_i + b_i^{np} + \sum_{k=1}^{i-1} \left\lceil \frac{t}{p_k} \right\rceil e_k; 0 < t \leq p_i$$

  - ...oder auch für die Betrachtung der CPU-Auslastung:

$$\sum_{k=1}^{n} \frac{e_k}{\min(D_k, p_k)} + \frac{b_i^{np}}{\min(D_i, p_i)} \leq 1 \quad ; i = 1, 2, \ldots, n$$

# Weitere Lockerung der Restriktionen

## Aufhebung von A4 für blockierende Synchronisation

Mathematische Ansätze zur Analyse periodischer Echtzeitsysteme schränken solche Systeme häufig stark ein:

A1 ~~Alle Aufgaben sind periodisch.~~

A2 ~~Alle Arbeitsaufträge können an ihren Auslösezeitpunkten eingeplant und ausgeführt werden.~~

A3 Termine und Perioden sind identisch.

A4 ~~Kein Arbeitsauftrag gibt die Kontrolle über den Prozessor ab.~~

A5 ~~Alle Aufgaben sind unabhängig voneinander, d.h. die einzige gemeinsame Ressource ist die CPU und es existieren keine Einschränkungen hinsichtlich der Auslösezeiten der Arbeitsaufträge.~~

A6 Der Overhead durch Unterbrechungen, Ablaufplanung oder Verdrängung ist vernachlässigbar.

A7 ~~Alle Aufgaben verhalten sich voll-präemptiv.~~

# Selbstsuspendierung

Selbstsuspendierung ermöglicht die erneute Blockierung eines Jobs

- Aufgaben, die sich für eine bestimmte Zeit selbst suspendieren verhalten sich nicht mehr wie periodische Aufgaben [7, S. 164]
  - sie beanspruchen in bestimmten Zeitintervallen mehr Rechenzeit
  - $\rightsquigarrow$ hierdurch erfahren andere Arbeitsaufträge zusätzliche Verzögerung
  - diese Blockierungszeit $b_i^{ss}$ von $J_i$ lässt sich nach oben abschätzen:

$$b_i^{ss} = s_i + \sum_{k=1}^{i-1} \min(e_k, s_i)$$

    - hierbei ist $s_i$ die längste Selbstsuspendierung von $T_i$
    - diese Zeit wird von $T_i$ ggf. später beansprucht
- insgesamt ergibt sich für die Blockierungszeit

$$b_i = b_i^{ss} + (K_i + 1) b_i^{np}$$

  - die Aufgabe $T_i$ suspendiert sich hier bis zu $K_i$-mal
  - jedes mal kann sie erneut für $b_i^{np}$ Zeiteinheiten blockiert werden

# Gliederung

# Resümee

Konkurrenz und Koordination nebenläufiger Aktivitäten

- Nebenläufigkeit, Kausalität, Kausalordnung
- Konfliktsituationen $\rightsquigarrow$ synchronisieren ohne Prioritätsumkehr

Verdrängungssteuerung $\mapsto$ verdrängungsfreie kritische Abschnitte

- benötigt kein *à priori* Wissen; Verklemmungsvorbeugung
- pragmatisch/effektiv, beeinträchtigt unabhängige Jobs

Prioritätsvererbung $\mapsto$ Priorität zeitweise erhöhen

- benötigt kein *à priori* Wissen
- direkte Blockierung, Blockierung durch Vererbung; transitiv

Prioritätsobergrenzen $\mapsto$ Priorität zeitweise deckeln

- benötigt *à priori* Wissen; Verklemmungsvorbeugung
- Grundmodell vs. (einfachere) stapelorientierte Variante

Ablaufplanung $\mapsto$ berücksichtigt Blockierungszeit

- Verzicht auf den Prozessor ermöglicht eine mehrfache Blockierung

# Literaturverzeichnis

[1] Baker, T. P.:
A Stack-Based Resource Allocation Policy for Real-Time Processes.
In: *Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS '90)*.
Lake Buena Vista, FL, USA : IEEE, Dez. 5–7, 1990, S. 191–200

[2] Baker, T. P.:
Stack-Based Scheduling of Realtime Processes.
In: *Real-Time Systems* 3 (1991), Nr. 1, S. 67–99

[3] Hansen, P. B.:
*Operating System Principles*.
Prentice Hall International, 1973

[4] Hoare, C. A. R.:
Monitors: An Operating System Structuring Concept.
In: *Communications of the ACM* 17 (1974), Okt., Nr. 10, S. 549–557

[5] Jones, M. B.:
*What really happened on Mars?*
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/, 1997

# Literaturverzeichnis (Forts.)

[6] Lampson, B. W. ; Redell, D. D.:
Experiences with Processes and Monitors in Mesa.
In: *Communications of the ACM* 23 (1980), Nr. 2, S. 105–117

[7] Liu, J. W. S.:
*Real-Time Systems*.
Prentice-Hall, Inc., 2000. –
ISBN 0–13–099651–3

[8] Mok, A. K.-L. :
*Fundamental Design Problems of Distributed Systems for Hard Real-Time Environments*.
Cambridge, MA, USA, Massachusetts Institute of Technology, MIT, Diss., Mai 1983. –
Technical Report MIT/LCS/TR-297

[9] Schröder-Preikschat, W. :
*Softwaresysteme 1*.
www4.informatik.uni-erlangen.de/Lehre/SS07/V_SOS1, 2007. –
Lecture Notes

[10] Sha, L. ; Rajkumar, R. ; Lehoczky, J. P.:
Priority Inheritance Protocols: An Approach to Real-Time Synchronization.
In: *IEEE Transactions on Computers* 39 (1990), Sept., Nr. 9, S. 1175–1185

# Literaturverzeichnis (Forts.)

[11] Wilner, D. :
*Vx-Files: What really happened on Mars?*
San Francisco, CA : Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97), Dez. 1997