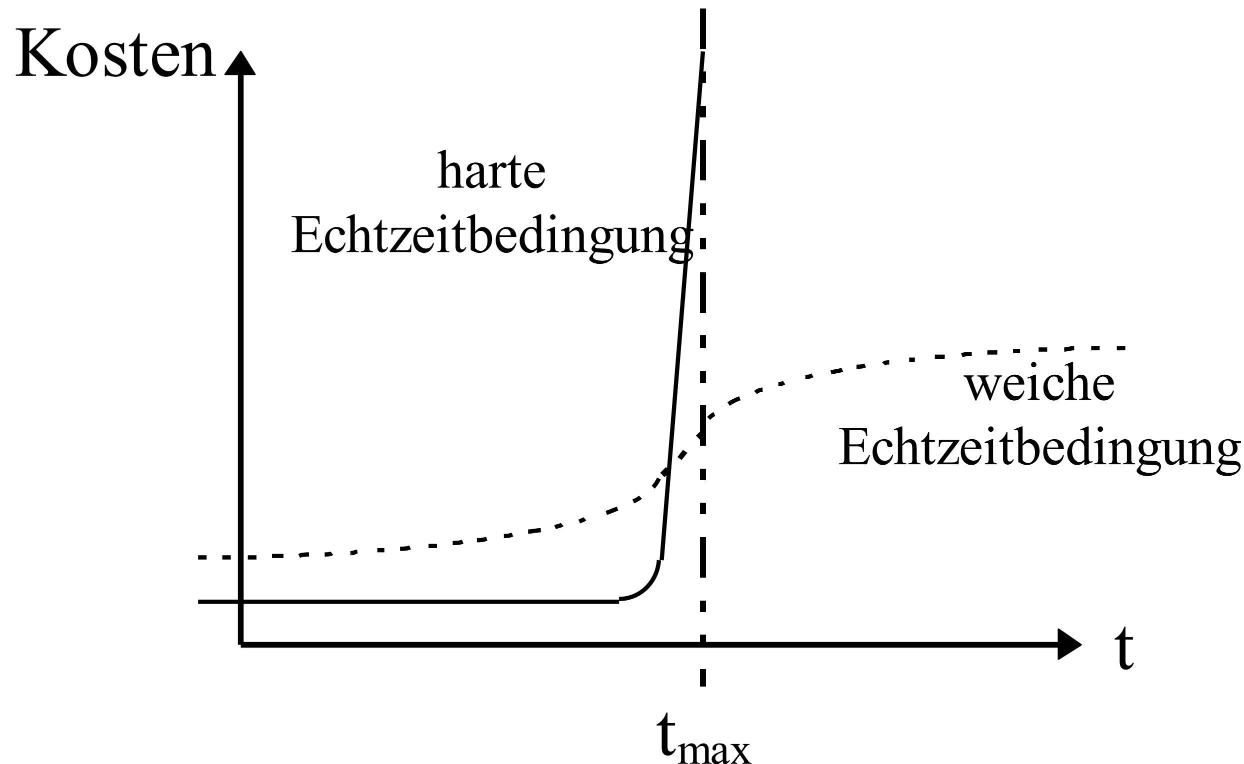
E. Grundlagen Echtzeitsysteme

E 1 Allgemeines

- **Forderung Echtzeitverhalten (Realzeitbetrieb): Der Rechner (PZR) muss mit den Vorgängen des technischen Prozesses, die er erfassen und steuern soll, trägheitslos Schritt halten können.(DIN 44300)**



- ► bei PZR u.U. um Antwortzeiten im Millisekunden- oder gar Mikrosekundenbereich:
    - *Beispiel: Bei einer numerisch gesteuerten Werkzeugmaschine durch ein PZR muss das Zeitintervall, welches für die Aufgaben*
        - *Messen und Umwandeln in Digitalsignale,*
        - *Erfassen,*

- *Verarbeiten,*
- *Ausgeben und umwandeln in Prozesssignale und*
- *Steuern*

benötigt wird, kleiner sein als bestimmte Regelstrecken-Zeitkonstanten, sonst ist eine online optimale rückgeführte Steuerung nicht möglich.

- **► Problem der zeitlichen Synchronisation der Rechenprozesse mit technischen Prozess:**
  - **Erfassung der unterschiedlichen Meßgrößen und/oder Ausgabe der Stellgrößen (Prozesssignalen) kann periodisch oder zeitlich variabel mit min/max Zeitangaben zu jeweils unterschiedlichen Zeiten nötig sein.**
    - **Berechnung der Stellgrößen hängt von der Verfügbarkeit der Messgrößen ab**
    - **unerwünschte/sicherheitsrelevante Prozesszustände müssen vorrangig und unmittelbar bearbeitet werden.**
- **► Problem bei einem PZR: die Zuteilung der exklusiv einmal verfügbaren Rechenzeit an die einzelnen Rechenprozesse, so dass Realzeitbetrieb gewährleistet**
- **Forderung Determiniertheit: Ein Echtzeitsystem heißt determiniert, wenn sich für jeden möglichen Zustand und für jede Menge an Eingangsinformationen eine eindeutige Menge von Ausgabeinformationen und ein eindeutiger nächster Zustand bestimmen läßt.**

E 2 Prozesssignalanbindung und PZR-interne Verarbeitung

- **Um schritthaltend den technischen Prozess bedienen zu können, müssen die Prozesssignale rechtzeitig verarbeitet werden. Dazu müssen :**
  - **zum einen die Prozesssignale ´in/an´ die PZR-CPU gelangen und von den Tasks entsprechend wahrgenommen werden können und**
  - **zum anderen entsprechend der Echtzeitbedingung die Verarbeitung der Signale so erfolgen, dass die entsprechenden Tasks die Arbeit rechtzeitig erledigen.(→Taskscheduling)**

► Aber: Wie und wann kommen die Prozesssignale zur Task und welche Task bekommt wann die CPU ?

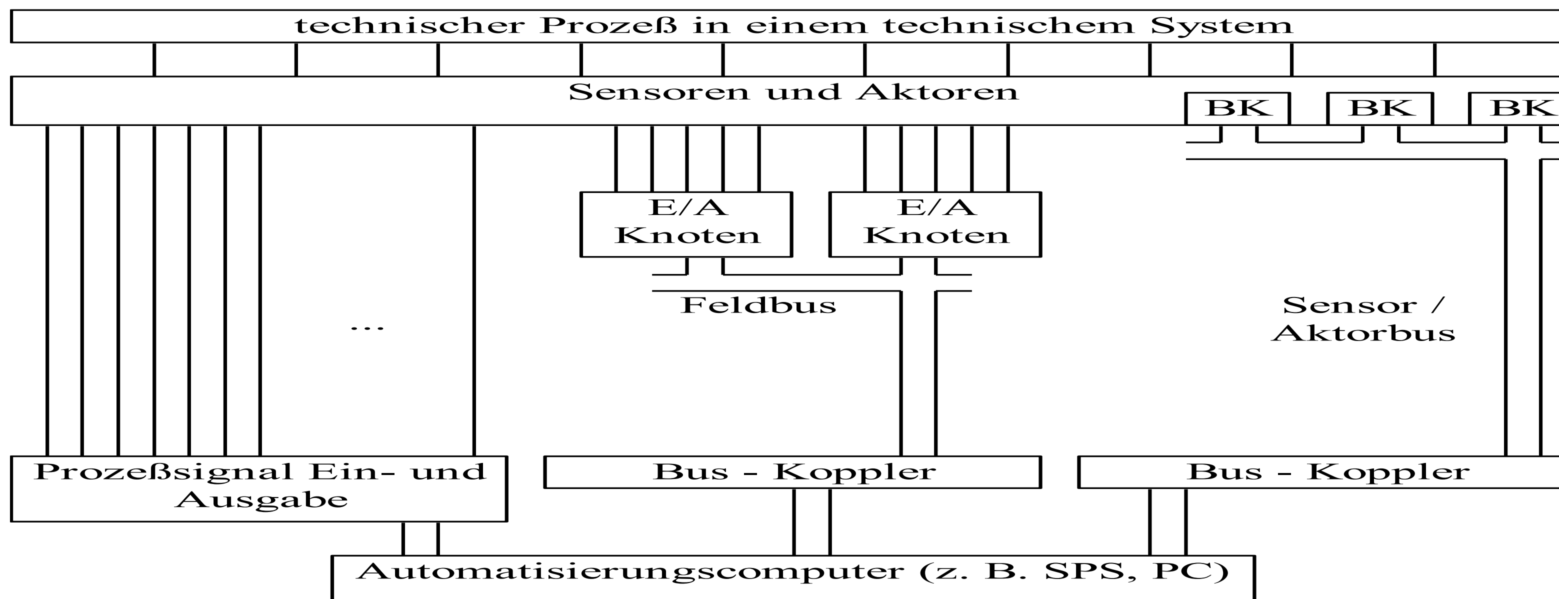


Bild aus /Sommer/

- **Prozesssignale aus technischem Prozess werden von Sensoren erfasst und liegen i.d.R. in digitaler Form in der Rechnerperipherie vor.**
  ►Das Ankommen eines neuen Prozesssignals(-zustands) wird i.d.R. in der Rechnerperipherie gespeichert
  ►'getriggertes Filpflop in einem Peripherie-Register'

- **Eine Möglichkeit den Wechsel des Prozesssignalzustands am Sensor zu bemerken ist das Polling der Rechnerperipherie**

- **Eine andere Möglichkeit ist, dass beim Auftreten eines Prozesssignalwechsels ein Hardwareinterrupt der CPU ausgelöst wird,** der i.d.R. zu einer Unterbrechung der laufenden CPU-Aktivität führt

  ► Einsprung **Interruptserviceroutine** (=kann auch die betreffende Task sein) ►Benachrichtigung der betreffenden Task
  ►Vorteil: Nur wenn ein Signalwechsel auftritt, erfolgt eine Bearbeitung und somit Belastung des Systems.

- **Prozesssignalpriorisierung (Ereignispriorisierung)** wenn mehrere Prozesssignale gleichzeitig auftreten:

  - **nur Prozesssignale (=externe Ereignisse) können zeitgleich auftreten** und werden i.d.R. durch CPU-Hardwaregegebenheiten (Interruptlevels) priorisiert.

  - ► I.d.R. gibt es am PZR mehrere voneinander unabhängige Interrupteingänge für externe Ereignisse die unterschiedliche Prioritäten untereinander haben.

(►N.B.: eine Möglichkeit der Priorisierung der Bearbeitung der Prozesssignale ohne Realzeitbetriebssystem (RBS) : Interruptpriorität=Taskpriorität und Task = Interruptserviceroutine).

- **Prozesssignale sind Hardwaresignale , die letztendlich von einer Task verarbeitet werden aber :**

  ► Tasks die sich gegenseitig zeitlich steuern wollen, benutzen programmgesteuert zusätzlich sog. Softwaresignale (Softwareinterrupts), logische Zusammenfassung: Prozesssignale und Softwaresignale ► Ereignisse

- **Ereignissquellen allgemein:**
  - **Extern:** Die Prozessperipherie angestoßen durch die Prozesssignale erzeugt hardwaremäßig Interrupts und gelangen über die CPU und Interruptserviceroutine an die zuständige Task. Alle Arten von interruptgebenden Timer (auch innerhalb des PZR) gehören auch dazu.
  - **Intern**: Tasks können interne Software-Ereignisse (Events) auslösen auf die Tasks warten können. Interne Ereignisse werden durch die Tasks erzeugt. => gleichzeitig kann immer nur eine Task laufen => interne Ereignisse können somit (bei SISD-Rechnern) nicht zeitgleich auftreten.
- **Ereignisempfänger ist immer eine Task**
  -

► in **Echtzeitbetriebssystemen (Realzeitbetriebssystem = RBS)** wird oft der **Interrupt in der Interruptserviceroutine** (in einem Devicetreiber) entgegengenommen und dort in der Interruptserviceroutine programmgesteuert in ein **Softwareereignis** (Software-Event) ‚umgewandelt' und dem RBS signalisiert. Das RBS verwaltet i.d.R. alle möglichen Softwareereignisse und benachrichtigt daraufhin die Task, die auf das Auftreten dieses Ereignisses gewartet hatte.

- **Zusammenfassung:** Die **Aktivierung eines Rechenprozesses (Task)** geschieht nun **durch** die **Interruptsteuerung (HW)/Ereignissteuerung (events , signals) (SW) anderer Prozesse** des PZR.

E 3 Tasks (Rechenprozesse)

- **Liste Taskeigenschaften und –attribute**
  - **Task hat eineindeutige Identifikation im System**
  - **Ist eine Sequenz von Anweisungen, besitzt privaten Adressraum, benötigt Betriebsmittel**
  - **Bedient rechtzeitig ein oder mehrere Prozesssignale (Rechtzeitigkeitsbedingung einhaltend)**
  - **Bearbeitung beliebig oft unterbrechbar und fortsetzbar oder teilweise ununterbrechbar ODER gänzlich ununterbrechbar**
  - **min./max. Verarbeitungszeit eines Aktivierungszyklusses**
  - **Minimale und maximale Taskreaktionszeit** ***response time***
  - **Prozesszeit (Wiederkehr desselben Prozesssignals)** ***arrival time***
  - **statische und/oder dynamische Wichtigkeit (Priorität) im Verhältnis zu den anderen beteiligten Tasks**
  - **Implementierungs-(Hardware, Software) und RBS-abhängig:**
    - **Größe (im Arbeitsspeicher),**
    - **Lade- (in den Arbeitsspeicher) , Start- (Anlegen der Taskverwaltungsdaten) und**
    - **Aktivierungszeiten (Kontextswitch)**
  - **Besitzt Taskzustand aus : bereit rechenwillig ; aktiv rechnend ;wartend auf**

- **Es existieren normalerweise mehrere Tasks gleichzeitig ► nebenläufige Tasks**

- **Anzahl der Tasks ist wesentlich größer als die Anzahl der zur Abarbeitung zur Verfügung stehenden Prozessoren ► zeitliche (Zeit eines Prozessors wird ‚portioniert') oder/und räumliche (Prozesse verteilt auf verschiedene Prozessoren) Teilung der verfügbaren Rechenzeit erforderlich (scheduling)**

- **Jede Task hat in dieser Betrachtung scheinbar einen Prozessor für sich alleine, Teilung ‚unsichtbar'**

- **Taskzustände:**

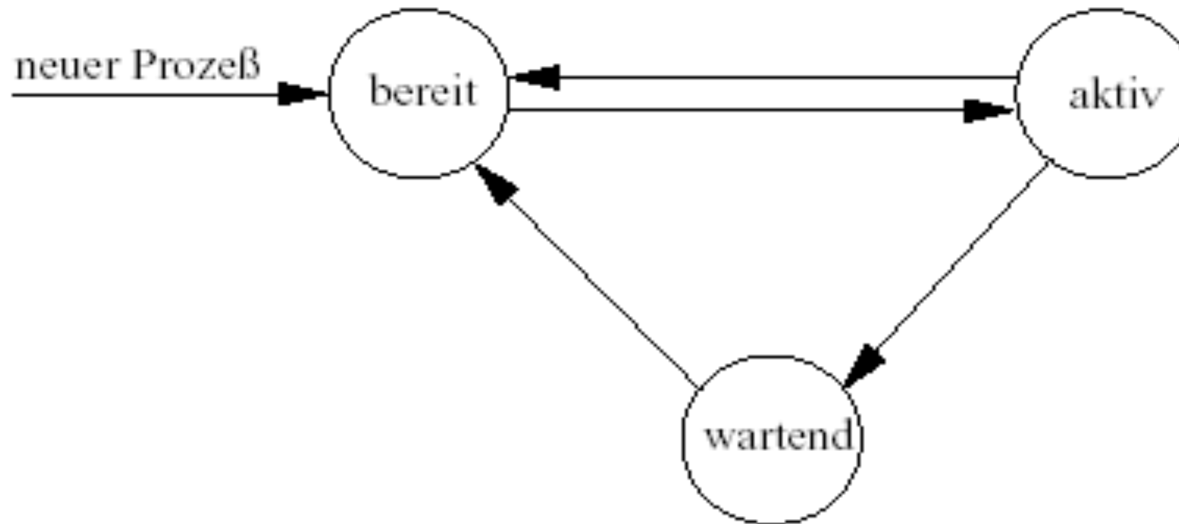


Bild /Robert Baumgart 2002/
aktiv **: Task hat Prozessor und rechnet ;** bereit: **Task will rechnen ist aber nicht ‚dran';** wartend**: Task wartet auf ein Ereignis und konkurriert** nicht!! **um die Prozessorzuteilung (egal welche Wichtigkeit sie hat!!); (**ruhend**: Menge möglicher zu startender Tasks im System)**

E 4 Strategien zur Zuteilung von Rechenzeit in Echtzeitsystemen (Taskscheduling)

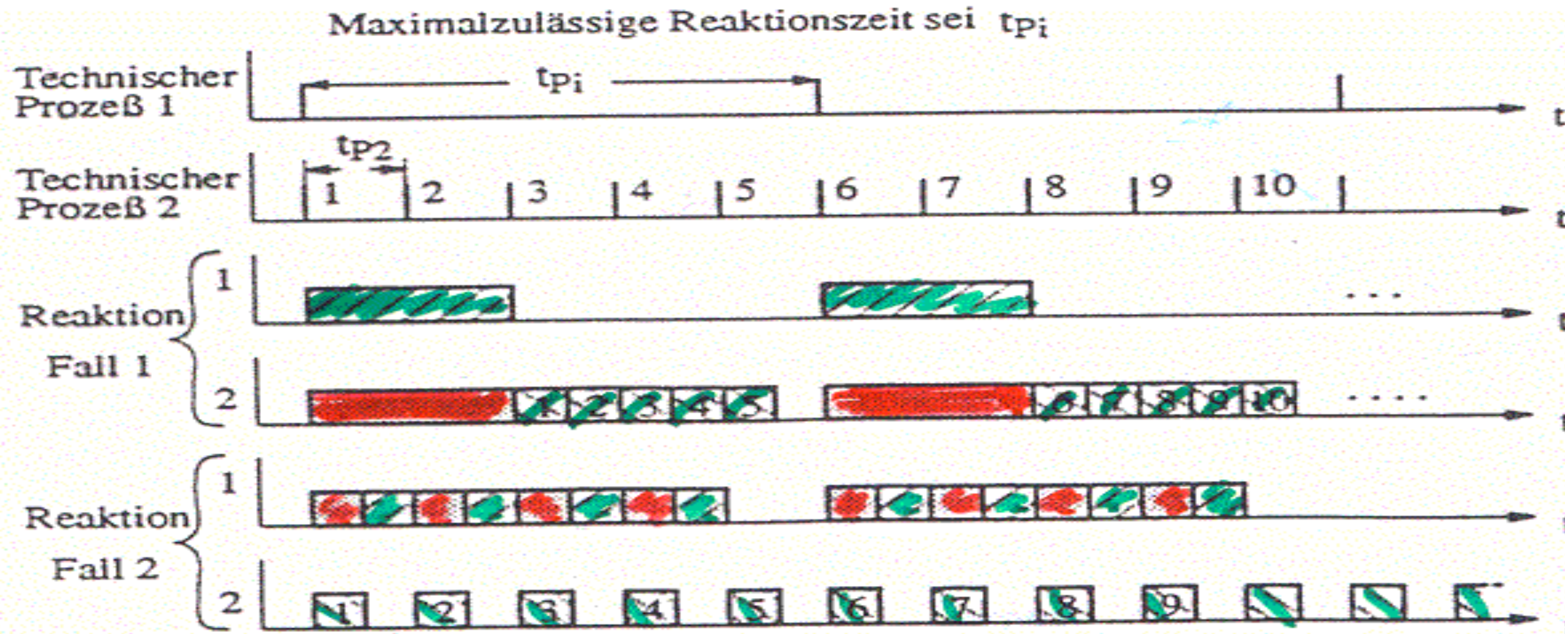
E 4.1 Ausgangssituation

► Forderung: PZR ist nach außen eine Blackbox die ‚rechtzeitig' alle Prozesssignale entgegennimmt und auch dem technischen Prozess zur Verfügung stellt

► Die Prozesssteuerungssoftware im PZR kann bestehen aus:

1. **Anwendungsteil ohne Betriebssystem, mit/ohne Interruptbearbeitung (‚kleine' embedded Systeme)**
2. **Anwendungsteil laufend unter Kontrolle eines (Echtzeit-)Betriebssystems (RBS) mit/ohne Interruptbearbeitung**

- **Beispiel : zwei Prozesssignale, die periodisch auftreten und eine unterschiedlich lange Bearbeitungszeit pro Auftreten benötigen**



- **Abarbeitung durch ein monolithisches Programm ist problematisch zu entwerfen weil:**
  - **ohne Interrupts müssen die Bearbeitungsalgorithmen ineinander verschränkt programmiert werden :-((► Fehleranfällig, schlecht wartbar, schlecht!**
  - **mit Interrupts ist die jeweilige Bearbeitung in die Interruptserviceroutinen verlagerbar; vorausgesetzt die CPU unterstützt Interruptprioritäten und sie sind richtig (►siehe Beispiel) verteilt**
    - **➔ Verfahren versagt wenn CPU nur ein Interrupteingang besitzt , die Interruptprioritäten hardwaremäßig falsch verteilt sind, wenn es mehr Prozesssignale als Interruptleitungen gibt**

- **Wunsch und Lösung:**

  - **Verteilung der Bearbeitung der einzelnen Prozessereignisse auf verschiedene in sich geschlossene Programmteile**

    ► **Sinnvoll nicht ein Programm** für alles zu konzipieren, sondern beispielsweise für jede zeitlich abgegrenzte für sich definierte Teilaufgabe: **Erfassung, Berechnung und Ausgabe einen eigenen Rechenprozess**(Task) zu entwickeln, die dann miteinender Daten austauschen.

    ► **Rechenprozesse (Tasks) = eigenständige ‚Hauptprogramme', aus deren Sicht sie jeweils alleine auf der CPU ausgeführt werden**

  - **Zuteilung der Rechenzeit (CPU) an die Tasks so, dass Realzeitbedingung zur Steuerung des technischen Prozesses eingehalten wird**

    ► **Einsatz Echtzeitbetriebssystem(-kern) mit Zuteilung der Rechenzeit zu Tasks (Taskscheduling)**

- **Aufbau und Funktionsweise von Echtzeitbetriebssystemen (RBS) ►später**

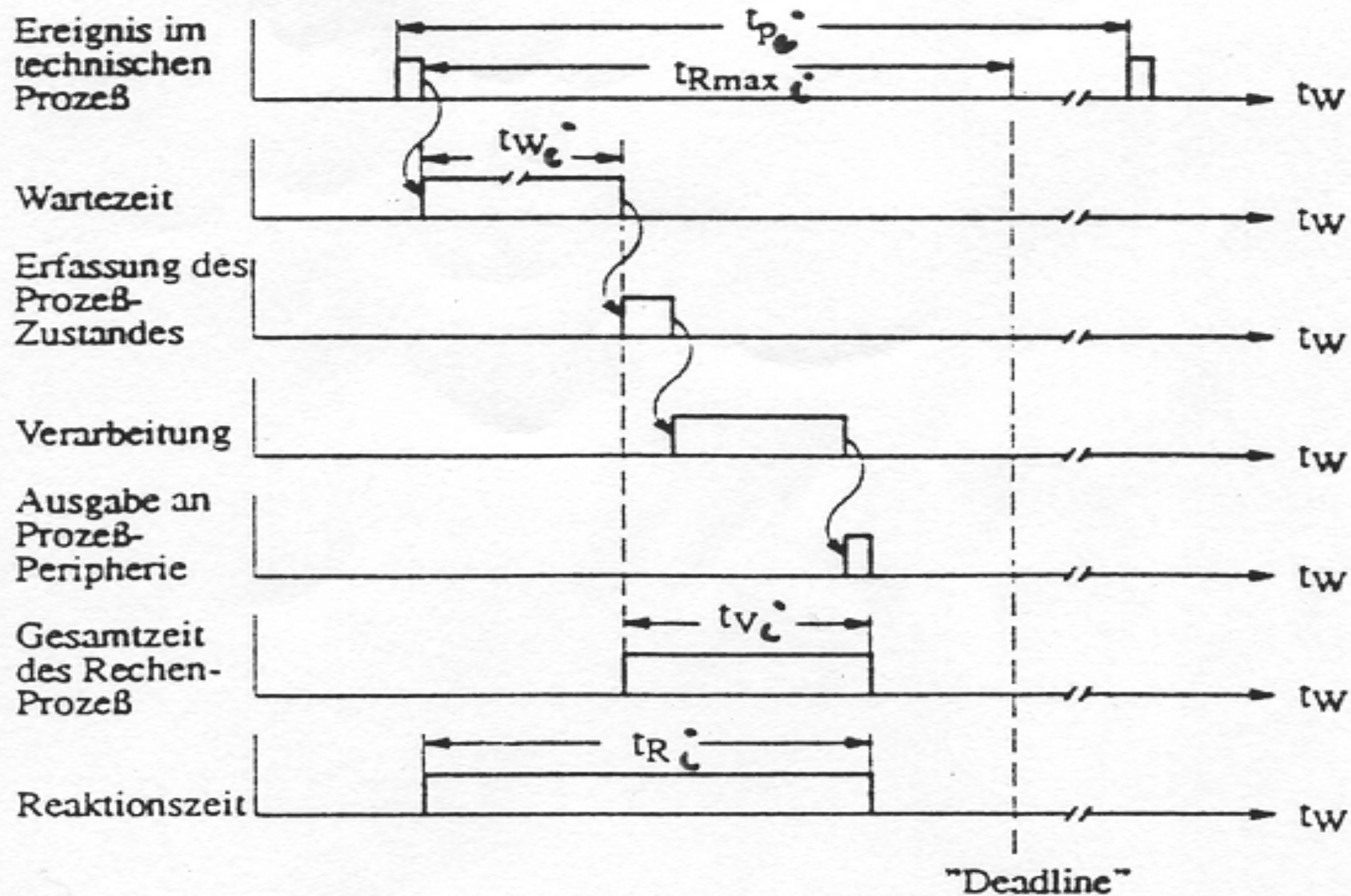
- **Welche prinzipiellen Strategien gibt es:**

  - **Rechenzeit zuzuteilen (Taskscheduling) und**

  - **die Echtzeitfähigkeit einer Lösung zu planen und zu überprüfen ?**

E 4.2 Überprüfung des Echtzeitverhaltens

Definition der zur Überprüfung der Echtzeit notwendigen Zeitgrößen (Bilder entnommen aus **/ Färber/):**



- **t_{pi} = minimale Prozesszeit des Prozesssignals i** ***periodic-arrival-time***

- **t_{wi} = Wartezeit der Task i (Inaktiv, z.B. bedingt durch Unterbrechnung)**
  -interference time

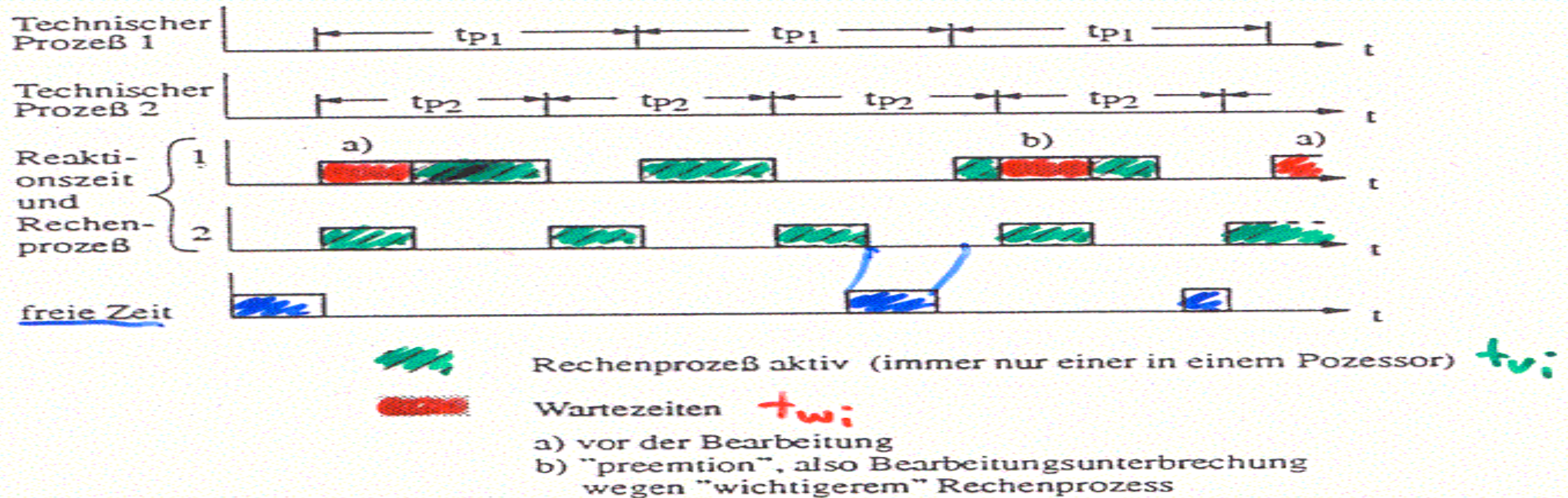
- **t_{vi} = Netto-maximale-Verarbeitungszeit der Task i -*WCET (Worst Case Execution Time, BCET Best Case Execution Time)***

- **$t_{Ri} = t_{vi} + t_{wi}$ ► erreichte Reaktionszeit -*response time*: tatsächliche Zeit von Beginn der Anforderung bis inklusive des Endes der Bearbeitung des Prozesssignals i .**
-

- **t_{Rimax} = vorgegebene maximal erlaubte Reaktionszeit -*absolute deadline t_{dmax}* - zur Bearbeitung des Prozesssignals i, ►** t_{Rimax} ist oftmals gleich t_{pi}

- **t_{Rimin}** = **vorgegebene minimal erlaubte Reaktionszeit –*minimum deadline t_{dimin}*- zur Bearbeitung des Prozesssignals i,** ► t_{Rimin} ist oftmals gleich 0

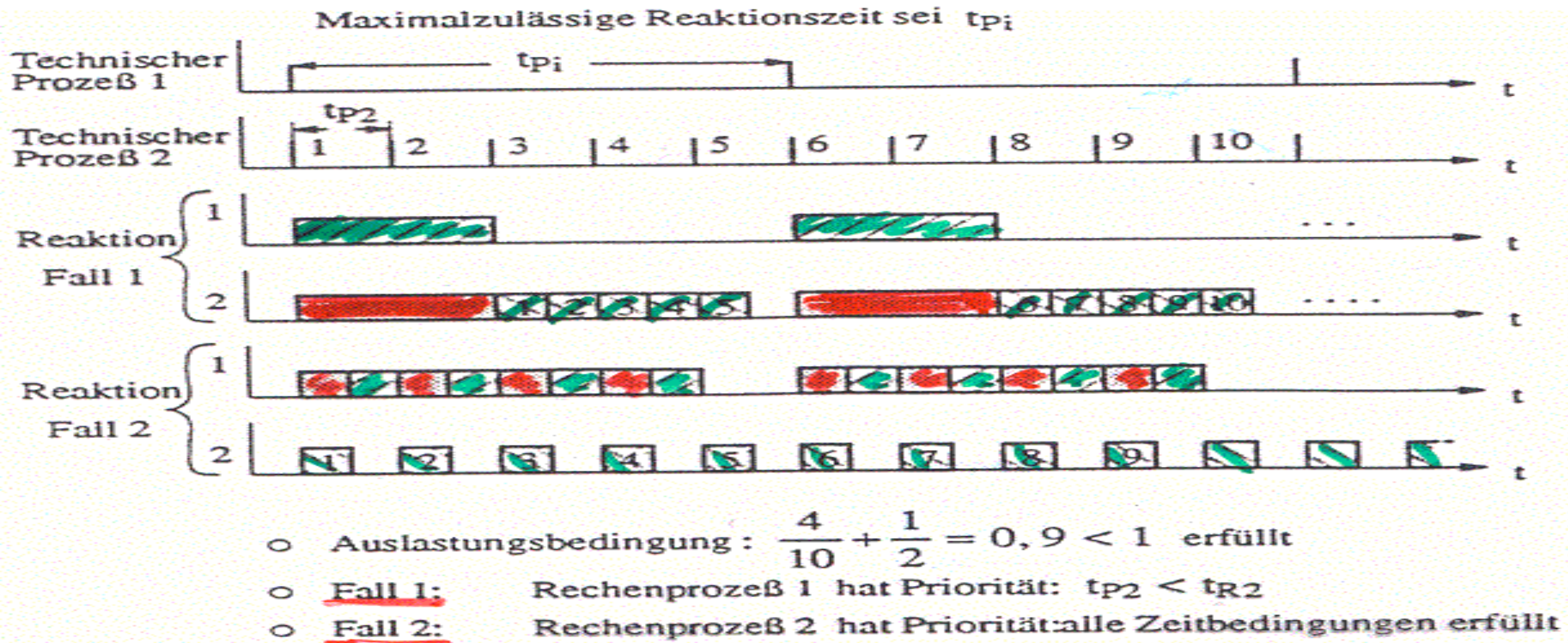
Die Echtzeitanforderung ist dann erfüllt wenn folgende Bedingungen gleichzeitig erfüllt sind:



- **Gleichzeitigkeit (utilization): Die Auslastung A der Rechnerkapazitäten muss unter 100% sein:**

A = $\sum_{alle\ i} \frac{tvi \max}{tpi \min} < 1$ **Die Bedingung ist notwendig aber nicht hinreichend!!.**

Im Bild: Auslastung A = 3/7 + 2/5 = 0,83 < 1 !!



Rechtzeitigkeit: Für jedes einzelne auftretende Prozessignal i muss gelten:

$$t_{Rimin} < t_{Ri} < t_{Rimax}$$

- ►**Im worst-case Fall muss also jede Task i mit der Bearbeitung sowohl vor vorgegebenem t_{Rimax} fertig sein als auch nach vorgegebenem t_{Rimin} fertig sein**
- ►**Diese Bedingung kann mittels graphischem Verfahrens im A-t-Diagramm oder mittels des algebraischen Verfahrens überprüft werden.**

Was tun wenn trotzdem die Echtzeitbedingungen nicht eingehalten werden können?

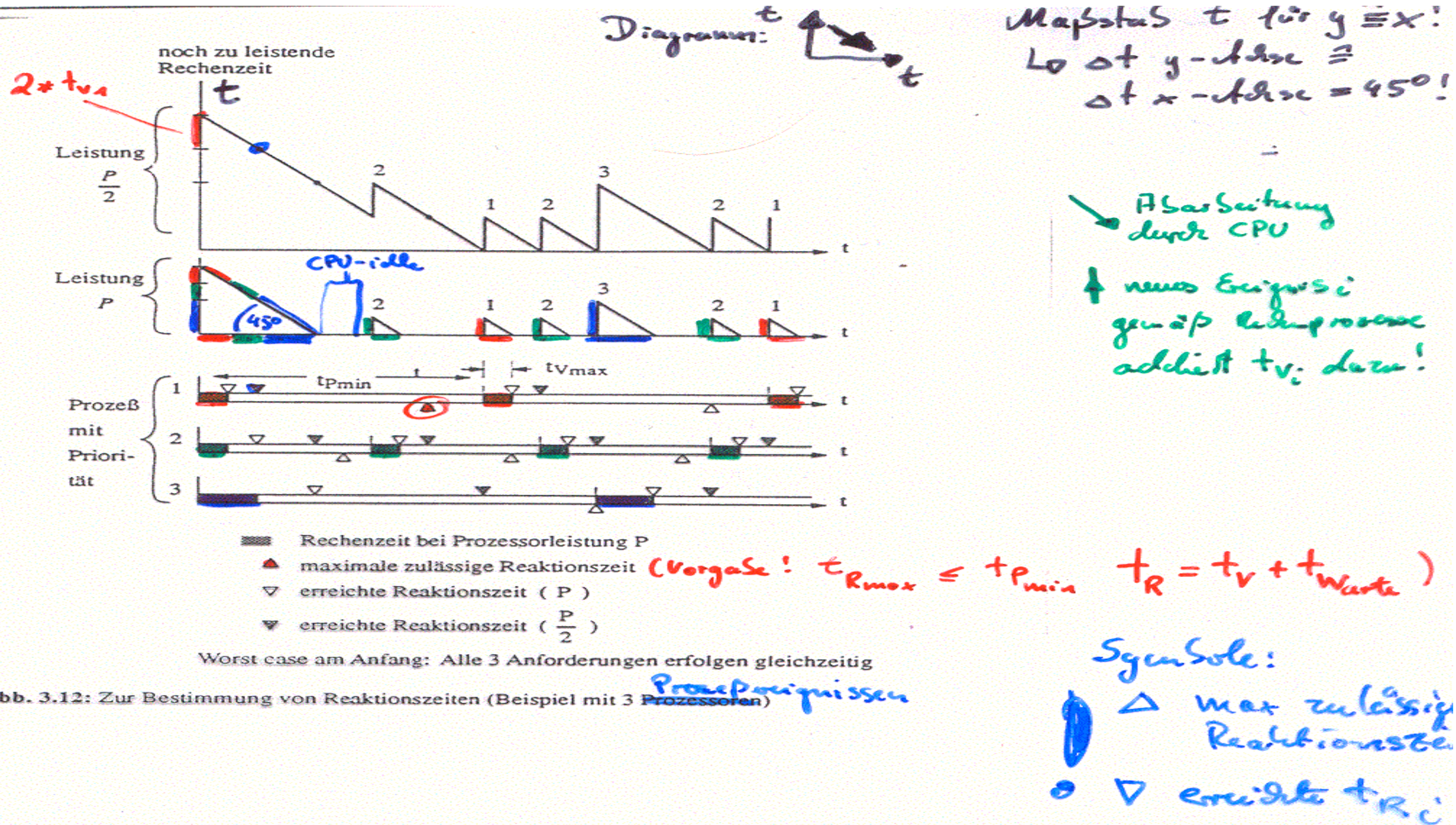


Abb. 3.12: Zur Bestimmung von Reaktionszeiten (Beispiel mit 3 Prozessoren)

Bild entnommen aus **/ Färber/**

► **Bild: CPU Leistung P/2 sei zu langsam ► Verdoppelung der Leistung auf P**

Graphisches Verfahren zur Überprüfung der Gleich- und Rechtzeitigkeit nach /Färber/ im Falle des Priortätsscheduling

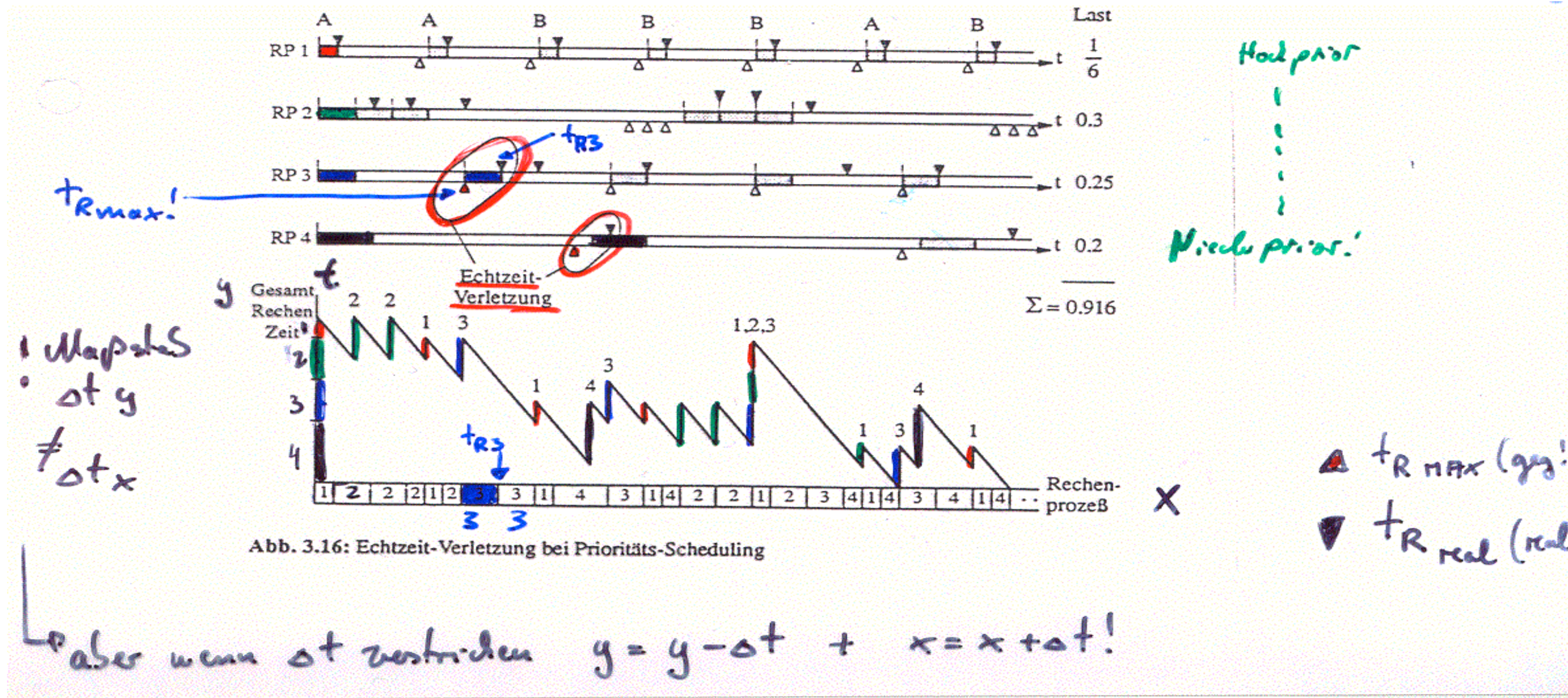


Abb. 3.16: Echtzeit-Verletzung bei Prioritäts-Scheduling

E 4.3 Rechenzeit-Zuteilungsstrategien (Taskscheduling)

- **Zuteilung der CPU an Tasks zur Bearbeitung der Prozessereignisse erfolgt im RBS durch sog. Scheduling-Algorithmen. ► zunächst seien alle Tasks beliebig unterbrechbar und fortsetzbar angenommen!**

Grundprinzip zyklische CPU-Zuteilung an die Tasks:

- Jede einzelne **Task erhält periodisch so oft Rechenzeit**, **dass er im Sinne des Realzeitbetriebs schritthaltend mit dem technischen Prozess seine individuelle zeitlich vorgegebene Aufgabe erledigen kann**

  ►sog. ‚round-robin'-Verfahren entweder mit gleichgroßen (häufig!) oder unterschiedlich großen Zeitscheiben (time-slices) je Task. ► jeder Prozess erhält somit periodisch eine individuelle oder einheitlich große Zeitscheibe der Rechenzeit und kann nicht gebrauchte Rechenzeit „zurückgeben" → ggf. Verkürzung eines ´round-robin´ Zyklusses

  ► **Dazu ist mindestens ein interruptgebender programmierbarer Timer im PZR nötig**, der spätestens periodisch am Ende der Zeitscheibe die laufende Task unterbricht, und damit in der Interruptserviceroutine (im RBS) dafür sorgt, dass die nächste Task dran kommt.

- Die **Erkennung** des Auftretens eines Prozesssignals durch die dazugehörige Task kann dabei durch **Pollen** der Rechnerperipherie, an der das Prozesssignal angeschlossen ist, erfolgen.

- → Wahl der Größe der Zeitscheibe? Verluste durch Kontextswitch beachten!
- Maximale Taskreaktionszeit T_{RMAXi} der Task i? Geg: Timeslice T, N-Tasks, Verarbeitungszeiten T_{Vi} , Aufgegebene Rechenzeit von Task i wird nicht zurückgegeben, Kontextswitchzeit vernachlässigbar.

$$T_{RMAXi} = T * (N-1) * \lceil T_{Vi}/T \rceil + T_{Vi}$$

► Besonderes Problem bei diesem Vorgehen: die **rechtzeitige** Erfassung von Alarmen oder nicht periodischen Ereignissen ► hier müssten die relevanten Eingänge durch einen RechenProzess so oft zeitlich abgetastet werden, dass noch Realzeitbetrieb gewährleistet ist

Grundprinzip ereignisgesteuerte priorisierte Zuteilung der CPU an die Tasks:

- 1. Das asynchrone Auftreten eines Prozesssignals (Ereignis) löst einen Interrupt und damit die Unterbrechung der gerade laufenden Task aus.
- 2. In der Interruptserviceroutine (im RBS) werden dann die Vorbereitungen für den ggf. stattzufindenden Taskwechsel getroffen. → **Das externe Ereignis**, der Interrupt, **wird in ein internes Ereignis, z.B. ein Event oder Signal, transformiert** und **„weckt“ die darauf wartende Task auf → Wechsel von wartend nach bereit**
- Da es i.d.R. unterschiedlich wichtige Prozessereignisse gibt und es weiterhin sichergestellt werden muss, dass die Realzeitbedingung für jede einzelne Task für sich betrachtet erfüllt sein muss, **benötigt man die Möglichkeit der Priorisierung der Ereignisbearbeitung**.
- **→Jede zu einem Prozesssignal zugeordnete Task hat hier deshalb als Eigenschaft eine "Wichtigkeit" = Priorität**

- Die in der Interruptserviceroutine „geweckte“ Task wird unmittelbar nach Ende der Interruptserviceroutine (Scheduler) nur dann drankommen wenn ihre Priorität größer ist als die der unterbrochenen Task.
- Vorteil dieses Verfahrens: Belastung des Systems nur bei Auftreten von Prozesssignalen

- Taskwechsel behandelt der Scheduler

- **Aktivierung des Schedulers (Wann?)**
    - In folgenden Situationen überprüft der Scheduler i.d.R, ob ein anderer Prozess die CPU erhalten sollte und damit gegebenenfalls ein Kontextwechsel veranlasst wird**:**
        - **Ende einer Systemfunktion (Übergang Kernel/User Mode):**
        - **Die RBS-Systemfunktion hat den aktiven Prozess blockiert (z.B. Warten auf Ende von I/O)**
        - **In der RBS-Systemfunktion sind die Scheduling-Prioritäten geändert worden.**
        - **Der aktive Rechenprozess terminiert.**
        - **Interrupts**
        - **Timer-Interrupt: der aktive Prozess hat sein Zeitscheibe verbraucht (Round Robin).**
        - **I/O signalisiert das Ende einer Wartebedingung (höher priorer Prozess wird „Bereit“).**
- **Mögliche weitere Scheduler-Bewertungskritierien, die zu einer**
    - Gerechtigkeit → Jeder Prozess soll einen fairen Anteil der CPU-Zeit erhalten.
    - Effizienz → Die CPU soll möglichst gut ausgelastet werden.
    - Durchlaufzeit → Ein Prozess soll so schnell wir möglich abgeschlossen sein.
    - Durchsatz → Es sollen so viele Jobs wie möglich pro Zeiteinheit ausgeführt werden.
    - Antwortzeit → Die Reaktion auf Ereignisse soll möglichst schnell erfolgen**.**

Ratenmonotones Prioritätsschedulingverfahren (RMS) (most popular!)

Vereinbarungen:

- **Task sind mit unterschiedlichen zur Laufzeit unveränderlichen statischen Prioritätswerten (gleiche Prioritäten verboten) ausgestattet.**
- **Task sind Prozesssignalen zugeordnet.**
- **Auslastung A < 100%**
- **Jede Task ist jederzeit unterbrech- und fortsetzbar (Preemption)**
- **Taskverarbeitungszeiten sind ‚konstant' oder bestenfalls weniger lang**
- **Die Taskprioritäten werden so verteilt, dass**
  - **die höchste Priorität an die Task geht, die das Prozesssignal mit der höchsten Wiederholrate behandelt**
  - **die zweithöchste Priorität an die Task geht mit der zweithöchsten Wiederholrate des Prozesssignals**
  - **usw.**
- **Ablaufwillige Tasks mit höherer Priorität unterbrechen sofort Tasks mit niederer Priorität.**

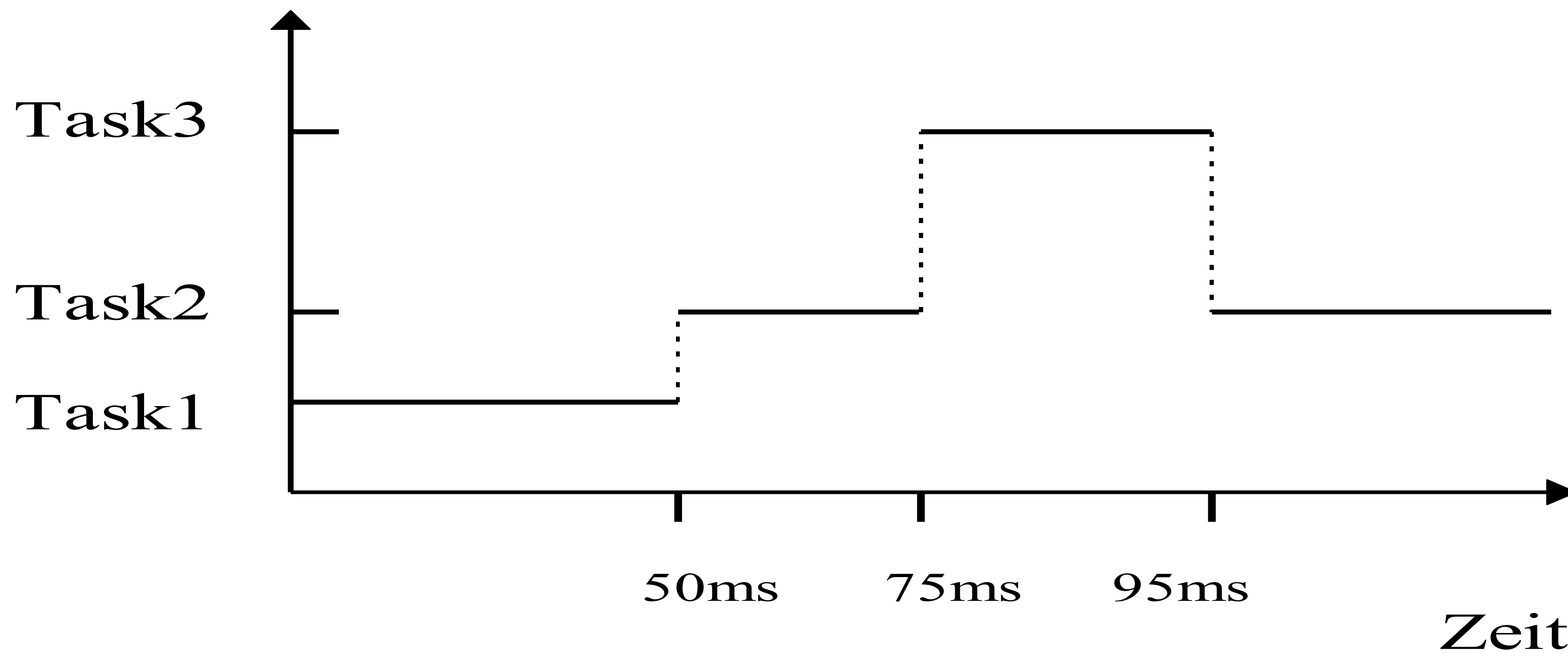
Algorithmus:

- **Das Auftreten eines externen oder internen Ereignisses löst u.A. die Ausführung des Prioritätsscheduler im RBS aus, der dann aus der Liste der laufwilligen Tasks diejenige mit der höchsten Priorität aktiviert.**

Offene Frage:

- **Wenn jede Task eine von allen anderen Tasks unterschiedliche Priorität hat, ist immer die Task am Zug, die die höchste Priorität hat. Diese Tasks nennen wir ab jetzt Echtzeittasks.**
- **Wie behandelt man Tasks die von der Aufgabe her keine Echtzeitanforderungen -Nicht-Echtzeittasks- stellen somit „untereinander gleichberechtigt" sind und sich die von den Echtzeittasks übriggebliebene Rechenzeit „gerecht" teilen sollen? RMS verlangt unterschiedliche Prioritäten!**

Ablaufdarstellung von Rechenprozessen (A-t Diagramm)



Beispiel: Überprüfung der Echtzeitfähigkeit:

Durch einen Prozessrechner mit Realzeitbetriebssystem und Prioritätsscheduling (3 Taskprioritätsebenen verfügbar, Ebene 3 höchste Proritát) werden drei Rechenprozesse (P1,P2 und P3) bearbeitet, die zusammen einen technischen Prozess steuern und überwachen.
Jeder RechenProzess Pi ist verantwortlich für die jeweilige schritthaltende Bearbeitung eines Prozessereignisses, das im Zeitabstand T_{pimin} (worst-case!!) periodisches auftritt, und der RechenProzess benötigt hierfür eine (geschätzte) Verarbeitungszeit T_{vimax} (worst-case!!).

RechenProzess P_i	Prozesszeit T_{pimin}	Max. Reaktionszeit erlaubt T_{Rmaxi}	Verarbeitungszeit T_{vimax}
P1	150 ms	150 ms	30 ms
P2	100 ms	100 ms	10 ms
P3	200 ms	200 ms	100 ms

- *Bestimmung der Auslastung (Gleichzeitigkeit): A = 30/150 + 10 /100 + 100/200 = 0.80 <<100%*
  und $U_{RMS} = 3^* (2^{1/3}-1) = 0.78$ => also 78% < A = 80% < 100% ok aber keine Aussage über Rechtzeitigkeit!.

- *Wie müssen die Prioritäten verteilt werden, damit jede Task schritthaltend verarbeitet werden kann?*
  Die Kontextswitchzeit von 10 usec ist zu vernachlässigen.
  - *Jede Task P_i muss sicher beendet sein, bevor sein zugehöriges Prozessereignis T_{pimin} wieder eintrifft.*

  *=> **Überprüfung durch Check im Diagramm Prozessorbelegung ob immer erfüllt ist, dass ein Ereignis bereits abgearbeitet wenn das gleiche Ereignis wieder auftritt!!***
  - *Jede Task P_i darf ansonsten jederzeit beliebig unterbrochen werden.*
  - ***worst-case** Ausgangsbedingung zum Startzeitpunkt der Diagramme ist, dass alle drei Prozessereignisse gleichzeitig zur Bearbeitung eintreffen.*

Algebraisches Verfahren zur Überprüfung des Echtzeitverhaltens im Falle RMS:

- **Wenn A <= U_{RMS}= n* ($2^{1/n}$ –1) * 100% : n=Anzahl der Tasks, dann kann mittels RMS eine Prioritätsverteilung gefunden werden, die Echtzeit garantiert**
- **Wenn U_{RMS} < A < 100% dann muss wie folgt gesondert überprüft werden:**
- **Worst-case Annahme, alle Tasks stehen jetzt t=0 gleichzeitig zur Bearbeitung an!**
- **Die erreichte Reaktionszeit für Task i sei $T_{Ri} = T_{vi} + T_{wi}$, wobei T_{vi} vorgegeben und T_{wi} wird von allen höherprioren Tasks j verursacht!**
- **Die Wiederkehr (Periodendauer) des zu Task i dazugehörigen Prozesssignals sei T_{Pi},**
- **Annahme: Task j sei höherprior als Task i und unterbricht diese**
- **Während T_{Ri} wird Task i von Task j genau $\lceil T_{Ri} / T_{Pj} \rceil$(nächste ganze Zahl, kein Bruch!) mal für die Dauer von T_{Vj} unterbrochen, also: $T_{Wi} = \lceil T_{Ri}/T_{Pj} \rceil * T_{Vj}$**
- **Gibt es mehr als eine Task j, die höherprior sind (Taskmenge HPRIO), so summieren sich die Unterbrechungszeiten verursacht durch jede Task j ($\lceil .. \rceil$sei eine Funktion die, die auf die nächste größere ganze Zahl aufrundet):**
-

$$T_{Wi} = \sum_{\forall j \in HPRIO} \left(\lceil T_{Ri} / T_{Pj} \rceil * T_{Vj} \right)$$

- **Die Taskreaktionszeit T_{Ri} der Task i ergibt sich dann zu:**

$$T_{Ri} = T_{Vi} + \sum_{\forall j \in HPRIO} \left(\lceil T_{Ri} / T_{Pj} \rceil * T_{Vj} \right)$$

- **Lösung der Gleichung für Task i durch Rekursion für n=0,... n aus den natürlichen Zahlen,mit dem Startwert $T^0_{Ri}=T_{Vi}$:**

$$T^{n+1}_{Ri} = T_{Vi} + \sum_{\forall j \in HPRIO} \left(\lceil T^{n}_{Ri} / T_{Pj} \rceil * T_{Vj} \right)$$

- **Wenn die Sequenz T^0_{Ri} , T^1_{Ri}, T^2_{Ri},...,T^n_{Ri} konvergiert und eine Lösung $T^{n+1}_{Ri} = T^n_{Ri}$ ergibt, dann ist ein Schedule für Task i erzielbar, ansonsten nicht**
- **Wenn für alle Tasks i je eine Lösung ihrer Gleichung T^{n+1}_{Ri} existiert, und T_{RMAXi} (erlaubt) > T^{n+1}_{Ri} dann ist auch bei der gegebenen Prioritätsverteilung die Echtzeitbedingung eingehalten**

Algebraische Lösung des Beispiels:

$$T_{Ri}^{n+1} = T_{Vi} + \sum_{\forall j \in HPRIO} \left(\left\lceil T_{Ri}^{n} / T_{Pj} \right\rceil * T_{Vj} \right)$$

- **Task Prioritäten: P2 hoch, P1 mittel und P3 niedrig.**
-

- **Task P3 mit $T^0_{R3}=T_{V3}$= 100 (alles in ms):**

T^{n+1}_{R3}	$T^n_{R3}/T_{P1}*T_{V1}$ +	$T^n_{R3}/T_{P2} * T_{V2}$ +	T_{V3}
140	$\lceil 100/150 \rceil$* 30 +	$\lceil 100/100 \rceil$* 10+	100
150	$\lceil 140/150 \rceil$* 30+	$\lceil 140/100 \rceil$* 10+	100
150	$\lceil 150/150 \rceil$* 30+	$\lceil 150/100 \rceil$* 10+	100
► Lösung existiert			

- **Task P1 mit $T^0_{R1}=T_{V1}$= 30 (alles in ms):**

T^{n+1}_{R1}	$T^n_{R1}/T_{P2} * T_{V2}$ +	T_{V1}
40	$\lceil 30/100 \rceil$* 10+	30
40	$\lceil 40/100 \rceil$* 10+	30
► Lösung existiert		

- **Task P2 mit $T^{n+1}_{R2}= T^n_{R2}= T^0_{R2}=T_{V2}$= 10,0 ms , Lösung existiert**

Damit ist mit der vorgeschlagenen Prioritätenverteilung im RMS Verfahren die Echtzeit gewährleistet

- **Alternative: Graphisches Verfahren zur Überprüfung des Echtzeitverhaltens im Falle RMS (Extra Umdruck)**

OS-9 Timesharing Algorithmus

- Es gibt auch **Mischformen** in denen sowohl **zyklische als auch ereignisgesteuerte prioritätsgeteuerte Scheduling**-Mechanismen implementiert sind → **Echtzeit/Nicht-Echtzeit-Tasks in einem System**
- OS9: **Echtzeittasksprioritäten** > **Maxage, Maxage < Timesharingtasks < Minage**
- OS9: **Laborsysteme: Maxage 255, Minage 0, Defaultprio = 128**

- Beispiel OS9 Scheduling, gegeben:

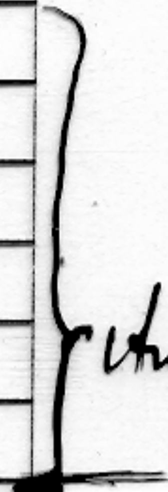
  Timeslice soll zwei Tics a 1ms entsprechen

  ► **pro Timeslice** ´= 2ms **altert´ (aging) ein bereiter rechenwilliger Prozess um 1**
  ► Maximal-Aging-Priorität = 255 (Maxage),
  ► Erhält ein bereiter Prozess endlich die CPU wird er auf seine Startpriorität zurückgestuft UND
  ► alle anderen bereiten Prozesse erhalten **gleichzeitig** einen Wartebonus von 1 Prioritätspunkt

Task	Startpriorität
Task A	Startpriorität 130
Task B	Startpriorität 128
Task C	Startpriorität 128

- Zu Beginn hat kein Prozess die CPU. Alle drei Tasks werden gleichzeitig gestartet.

Process Scheduling: A Simple Case



Iteration	CPU	Active Queue		
pre	<none>	A_{130}	B_{128}	C_{128}
1	A^{130}	A_{130}	B_{129}	C_{129}
2	A^{130}	B_{130}	C_{130}	A_{130}
3	B^{128}	C_{131}	A_{131}	B_{128}
4	C^{128}	A_{132}	B_{129}	C_{128}
5	A^{130}	B_{130}	A_{130}	C_{129}
6	B^{128}	A_{131}	C_{130}	B_{128}
7	A^{130}	C_{131}	A_{130}	B_{129}
8	C^{128}	A_{131}	B_{130}	C_{128}
9	A^{130}	B_{131}	A_{130}	C_{129}
10	B^{128}	A_{131}	C_{130}	B_{128}
11	A^{130}	C_{131}	A_{130}	B_{129}
12	C^{128}	A_{131}	B_{130}	C_{128}

Auswahl anderer bekannter scheduling-Verfahren:

- **statisches Scheduling vor Laufzeit des Systems**
  ►Die Prioritäten werden in der Planung fest vergeben und außerdem wird
  ► unabhängig zu den Prozesssignalen die Ablaufreihenfolge fest vorgegeben
  ► **Nicht gebrauchte Rechenzeit einer Task wird nicht von anderen verwendet**!
  ► damit streng wiederkehrende zyklische Ablaufreihenfolge der Tasks
  ► Einsatz in sicherheitsrelevanten Systemem, da das Funktionieren formal! nachweisbar ist

- **dynamisches Scheduling zur Laufzeit des Systems bei**
  ►Zuteilung des Prozessors an Rechenprozesse durch den im Betriebssystem enthaltenen Scheduler aufgrund der jeweils aktuellen Bedarfssituation. Rechenprozessemüssen damit *preemptiv* (unterbrechbar) sein. ► **Taskzustandsübergängen, bei externen Ereignissen, nach Zeitvorgaben, nach RBS-Aufruf**

  - **FCFS (First Come First Serve)** (keine Zeitscheibe)

  ► Die bereiten Prozesse sind in **einer Warteschlange nach ihrem Erzeugungszeitpunkt** geordnet.
  ► Jeder Prozess darf bis zu seinem Ende laufen, außer er geht in den Zustand „Blockiert“ über.
  ► Geht ein Prozess vom Zustand „Blockiert“ in den Zustand „Bereit“ über, wird er entsprechend seinem Erzeugungszeitpunkt wieder in die Warteschlange eingereiht, unterbricht aber den laufenden Prozess nicht.
  ► Anwendungen → Batch-Systeme um gleiche mittlere Wartezeiten für alle Prozesse zu erreichen.

- **round-robin (Zeitscheibenverfahren)**

► Alle Prozesse werden in eine Warteschlange eingereiht.
► Jedem Prozess wird eine Zeitscheibe (time slice, auch unterschiedlicher Größe)zugeteilt.
► Zeitgeber löst nach Ablauf der Zeitscheibe einen Interrupt aus.
► Ist ein Prozess 1. nach Ablauf seiner Zeitscheibes noch im Zustand „Aktiv“ oder 2. gibt er freiwillig den Rest seiner Zeitscheibe her:
 ► wird der Prozess verdrängt (preempted), d.h. in den Zustand „lauffähig“ versetzt und!
 ► wird der Prozess am Ende der Warteschlange eingereiht und!
 ► wird dem ersten Prozess in der Warteschlange die CPU zugeteilt.
► Geht ein Prozess vom Zustand „schlafend“ in den Zustand „lauffähig“ über, so wird er am Ende der Warteschlange eingereiht.
► Das Verhältnis zwischen Zeitscheibe und Kontextwechselzeit muss vernünftig sein.
► Große Zeitscheibe: effizient, aber lange Verzögerungszeiten und Wartezeiten möglich.
► Kleine Zeitscheibe: kurze Antwortzeiten, aber großer Overhead durch häufige Prozessumschaltung.

- **Time-Division-Multiplexing**

►Zyklisch erhält jede Task immer für die Dauer einer Zeitscheibe die CPU.
►Der Zeit-Abstand zweier aufeinanderfolgenden Aktivierungen ist dabei konstant. =>Damit kann bei n-Tasks jede Task fest mit 1/n der CPU-Leistung rechnen.

- **Deadline-Scheduling**

►Wirkung sehr ähnlich dem Prioritätsscheduling. ►Nahe Deadline verdrängt zeitlich weiter wegliegende.
►Die Wichtigkeit einer jeden Task ist hier nicht statisch, sondern ermittelt sich aus dem Zeitabstand zu dem Zeitpunkt an dem die betreffende Task fertig sein muss (Deadline), um nicht die Echtzeitbedingung zu verletzen.
►Je näher zur Deadline desto höher die Wichtigkeit.
→EDF: Planen nach Fristen (Earliest Deadline First): Der Prozess dessen Frist als nächstes endet erhält den Prozessor.

→ LST: Planen nach Spielraum (Least Slack Time): Der Prozess mit dem kleinsten Spielraum (Deadline-aktuelle Zeit -remaining time) erhält den Prozessor. Der Spielraum eines Prozesses der gegenwärtig ausgeführt wird ist konstant. Die Spielräume aller anderen Prozesse nehmen ab.

►Vorteil von LST: LST erkennt Fristverletzungen früher als EDF.

►Scheduling mit Bewertung der Deadlinezeiten erfolgt beim Eintreffen eines Ereignisses und damit verbundenen Aktivierung einer dann rechenwilligen Task.

► Deadline zuverlässig im vorhinein zu bestimmen ist nicht immer möglich

- **Scheduling nach POSIX 1003.1b**

► Prioritätengesteuertes Scheduling
►Auf jeder Prioritätsebene können sich mehrere Prozesse befinden.
► Innerhalb einer Prioritätsebene werden Prozesse gescheduled nach:
► First In First Out (First Come First Serve)
► Round Robin
► Prioritätsebene 0 besitzt die niedrigste Priorität.

E 4.4 Prioritätsinversion

► Das Problem der Prioritätsinversion bezeichnet eine Situation in der ein Prozess mit niedriger Priorität einen Prozess mit hoher Priorität blockiert. Wo kann das passieren?

► Im „kritischen Abschnitt“ beim Nutzen gemeinsamer exklusiv nutzbarer Betriebsmittel:

→Tasks kommunizieren i.d.R. untereinander und greifen z.B. auf gemeinsamen Speicher (shared memory) lesend wie schreibend zu

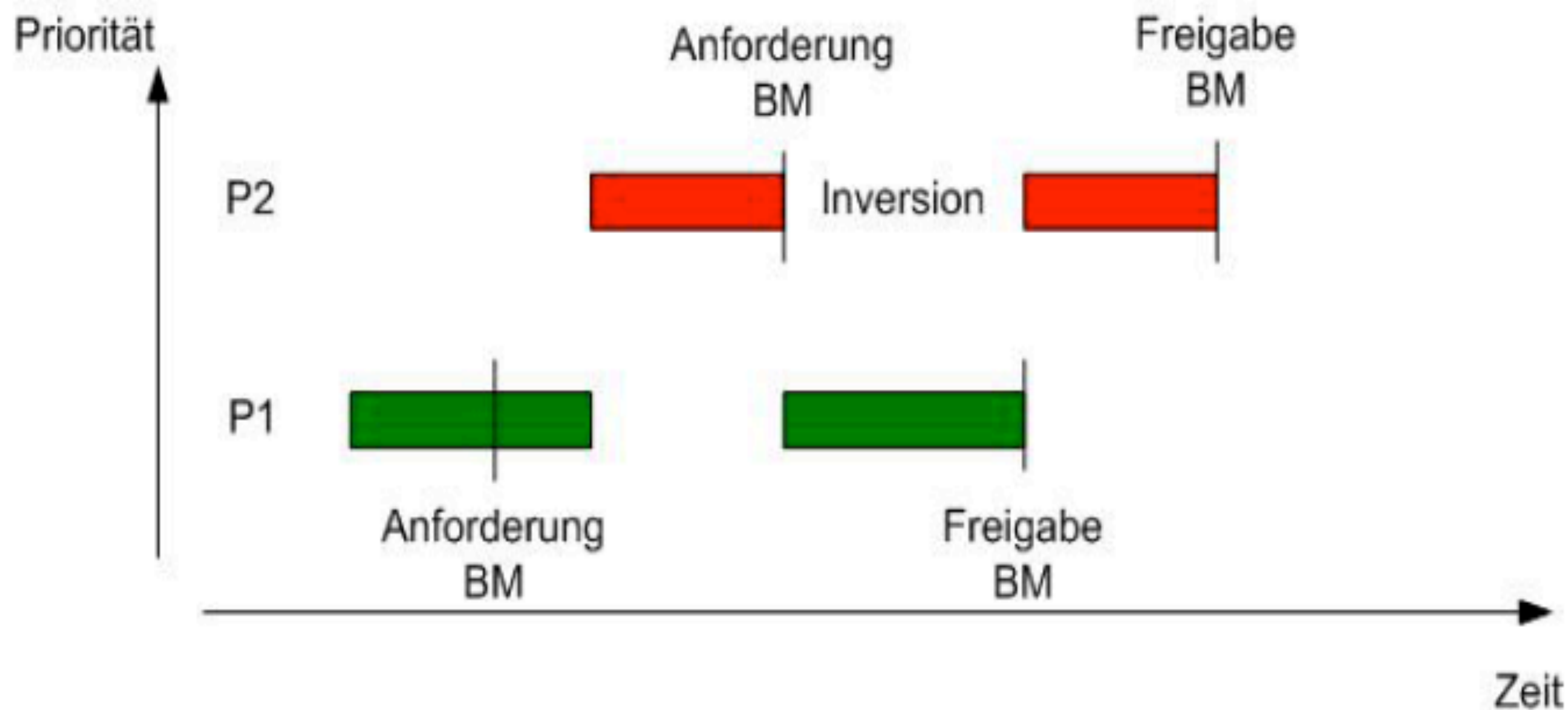
→ schreibt aktuell eine Task, so muss sichergestellt werden, dass während dessen keine! andere Task im gemeinsamen Speicher(-bereich) weder liest noch schreibt, ansonsten könnten Dateninkonsistenzen auftreten

→ Gelöst wird dies durch die Bereitstellung von Mechanismen, w.z.B. die Implementation von Semaphoren, die es gestattet sicherzustellen, dass nur eine Task gleichzeitig im „kritischen (Programm-)abschnitt“ ist, in dem gelesen oder geschrieben wird.

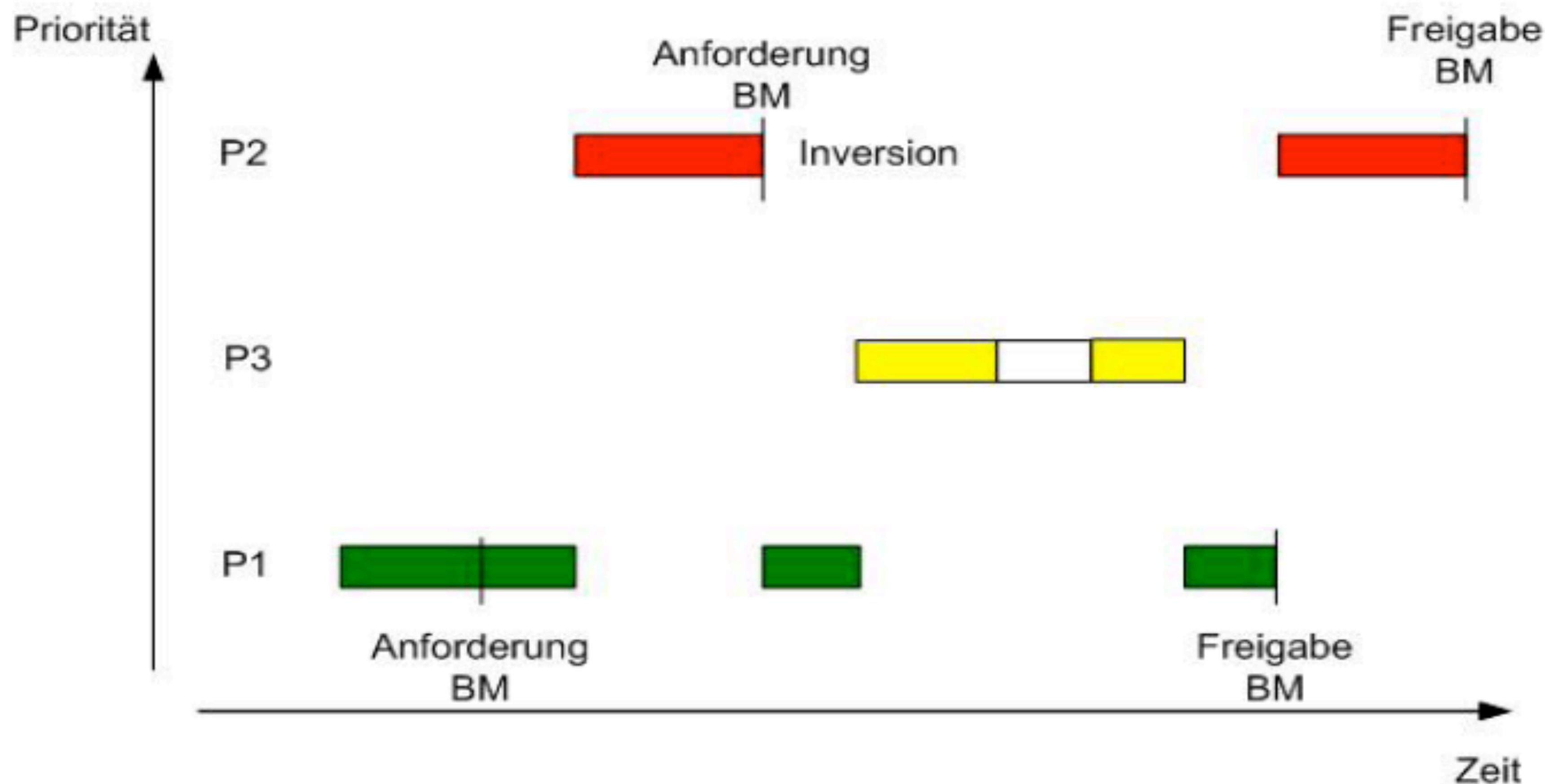
→ Wird nun einer Task niederer Priorität gestattet in den kritischen Abschnitt einzutreten und kurz danach eine höher priore Task aktiviert, die ihrerseits in denselben kritischen Abschnitt will, so muss diese trotz ihrer höheren Priorität warten bis die niederpriore Task den kritischen Abschnitt wieder verlässt. ► somit liegt eine Prioritätsinversion vor : niedere Priorität hat „scheinbar“ Vorrang.

► anstelle eines gemeinsamen Speicherbereichs kann man verallgemeinert von einem Betriebsmittel **(BM)** sprechen, das gleichzeitig immer nur von einer Task (Thread) genutzt werden darf

- ► **Es lassen sich zwei Arten von Prioritätsinversionen unterscheiden:**
  - ► **begrenzte (bounded) Prioritätsinversion, die nicht!! vermieden werden kann**: (Bild Prof. Dr. Knoll/TUM)



► **unbegrenzte (unbounded) Prioritätsinversion:** (Bild Prof. Dr. Knoll/TUM).



Die Task P3 mit der Priorität zwischen P2 und P1 blockiert P1 „beliebig“ lange. Damit gibt P1 „beliebig“ lange das BM nicht her und blockiert somit P2 weiterhin. Das Blockieren von P1 durch P3 gilt es zu verhindern!

Prioritätsvererbung (priority inheritance)

**Der niederpriore Prozess, der das entsprechende Betriebsmittel (BM) besitzt,
erbt die höhere Priorität des höherprioren Prozesses, der das Betriebsmittel anfordert,
solange der niederpriore Prozess das gemeinsame Betriebsmittel blockiert.**

→ andere rechenwillige Tasks mit dazwischenliegenden Prioritäten können somit die ehemals niederpriore Task nicht mehr unterbrechen.

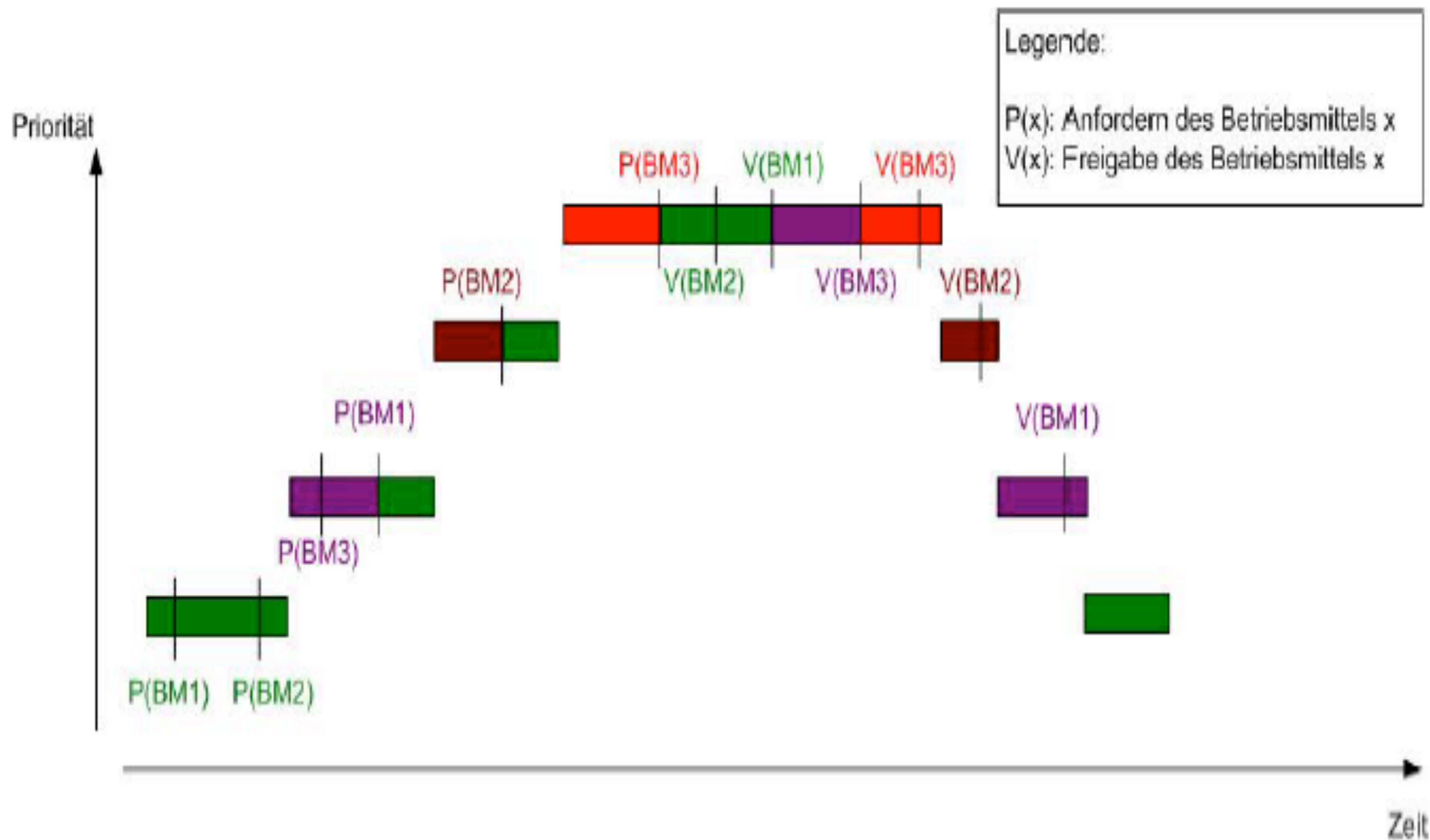
► dadurch begrenzte Prioritätsinversion
► Die Dauer der Blockierung und auch der Prioritätsanhebung wird auf die Dauer des kritischen Abschnitts begrenzt.

► Die Blockierungen werden „geschachtelt" hintereinander gereiht (Blockierungsketten).

(►Es verhindert keine Deadlocks.)

► VxWorks besitzt diesen Mechanismus → Marsmission

(Bild Prof. Dr. Knoll/TUM).
Anforderung: P(BMx) → Belegen des BMx, Aufruf blockiert wenn BMx nicht frei ; V(BMx) gibt BMx frei
→ **Task 1 belegt BM1,BM2 → Task 2 belegt BM3, BM1 → Task 3 belegt BM2 → Task 4 belegt BM3**



Prioritätsgrenzen (priority ceiling) → Spezialfall: Immediate priority ceiling

Vorab: Jedes Betriebsmittel s (BMs) erhält eine Prioritätsgrenze *ceil(s)*, dieses entspricht dem Maximum der Prioritäten aller Prozesse j, die je auf s Zugreifen könnten.

Start: alle BMs sind nicht belegt → aktceil=0

Zuteilungsregel Anforderumg BMi durch Prozess p:

→ BMi bereits durch anderen Prozess j belegt → **Prozess p blockiert sofort (1)**
→ BMi frei → Prüfung bestimme **aktceil=Maximum aller ceil(BMx) im Augenblick zugeteilter BMs (2)**
- **→ aktprio(p) > actceil,, BMs wird zugeteilt(2.1)**
- **→ aktprio(p)<=actceil, BMs wird nur zugeteilt wenn**
  - **→ Prozess p bereits die eine BMx mit dem gerade aktuellem actceil besitzt(2.2)**
  - **→ sonst blockiert Prozess p an freiem! BMi (2.3)**

Prioritätsvererbungsregel (3):
Blockiert ein Prozess p *an einer Ressource BMi*, so erbt der Prozess j, der diese Ressource BMi momentan besitzt, die (höhere) Priorität aktprio(p) des anfragenden Prozesses. Der Prozess j, der die Ressource BMi schon besitzt, wird nun unter dieser Priorität aktprio(p) weiterverarbeitet, bis er **alle Ressourcen BMx inkl BMi** freigegeben hat, deren Prioritätsschranke größer oder gleich der geerbten Priorität aktprio(p) ist.
Resultat:

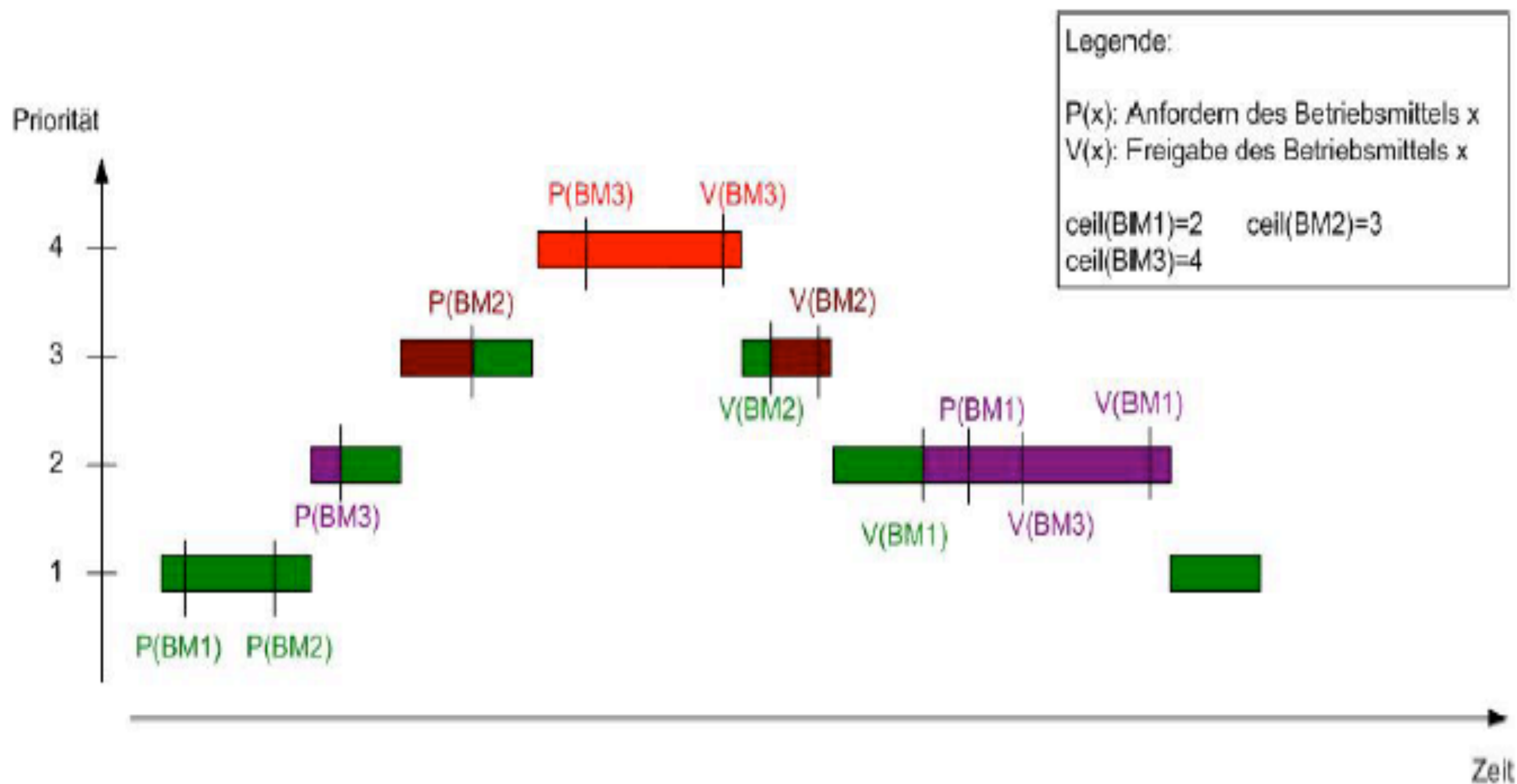
- ►Blockierung nur für die Dauer eines kritischen Abschnitts, verhindert Verklemmungen
- ►schwieriger zu realisieren, zusätzlicher Prozesszustand

(Vereinfachtes Protokoll: Immediate priority ceiling: Prozesse, die ein Betriebsmittel s belegen, bekommen sofort die Priorität ceil(s) zugewiesen)

(Bild Prof. Dr. Knoll/TUM).

Planungprio: ceil(BM1)=2 , ceil(BM2)=3, ceil(BM3)=4, actceil=0

Anforderung: → **Task 1 belegt BM1 (Regel 2.1) actceil=2; → Task 1 belegt BM2 (Regel 2.2) actceil=3; → Task 2 blockiert an BM3, weil aktprio=2 <= aktceil (Regel 2,3), Task 1 Prio ändert sich nicht!→ Task 3 wird geblockt BM2 (Regel 1), Task 1 Prio wird auf 3 gesetzt (Regel 3)→ Task 4 belegt BM3, weil aktprio=4 > actceil=3, actceil=4 → Task 4 gibt BM3 frei, actceil=3 → Task 3 erhält nach Freigabe durch Task 1 BM2 → actceil=2 sofort, da nun aktprio=3 > actceil=2 ist…**



E. Grundlagen Echtzeitsysteme 1
E 1 Allgemeines 1
E 2 Prozesssignalanbindung und PZR-interne Verarbeitung 3
E 3 Tasks (Rechenprozesse) 6
E 4 Strategien zur Zuteilung von Rechenzeit in Echtzeitsystemen (Taskscheduling) 8
E 4.1 Ausgangssituation 8
E 4.2 Überprüfung des Echtzeitverhaltens 11
E 4.3 Rechenzeit-Zuteilungsstrategien (Taskscheduling) 17
Grundprinzip zyklische CPU-Zuteilung an die Tasks: 17
Grundprinzip ereignisgesteuerte priorisierte Zuteilung der CPU an die Tasks: 18
Ratenmonotones Prioritätsschedulingverfahren (RMS) (most popular!) 20
Algebraisches Verfahren zur Überprüfung des Echtzeitverhaltens im Falle RMS: 23
Ablaufdarstellung von Rechenprozessen (A-t Diagramm) 20
Beispiel: Überprüfung der Echtzeitfähigkeit: 22
OS-9 Timesharing Algorithmus 26
Auswahl anderer bekannter scheduling-Verfahren: 28
E 4.4 Prioritätsinversion 31
Prioritätsvererbung (priority inheritance) 34
Prioritätsgrenzen (priority ceiling) → Spezialfall: Immediate priority ceiling 36