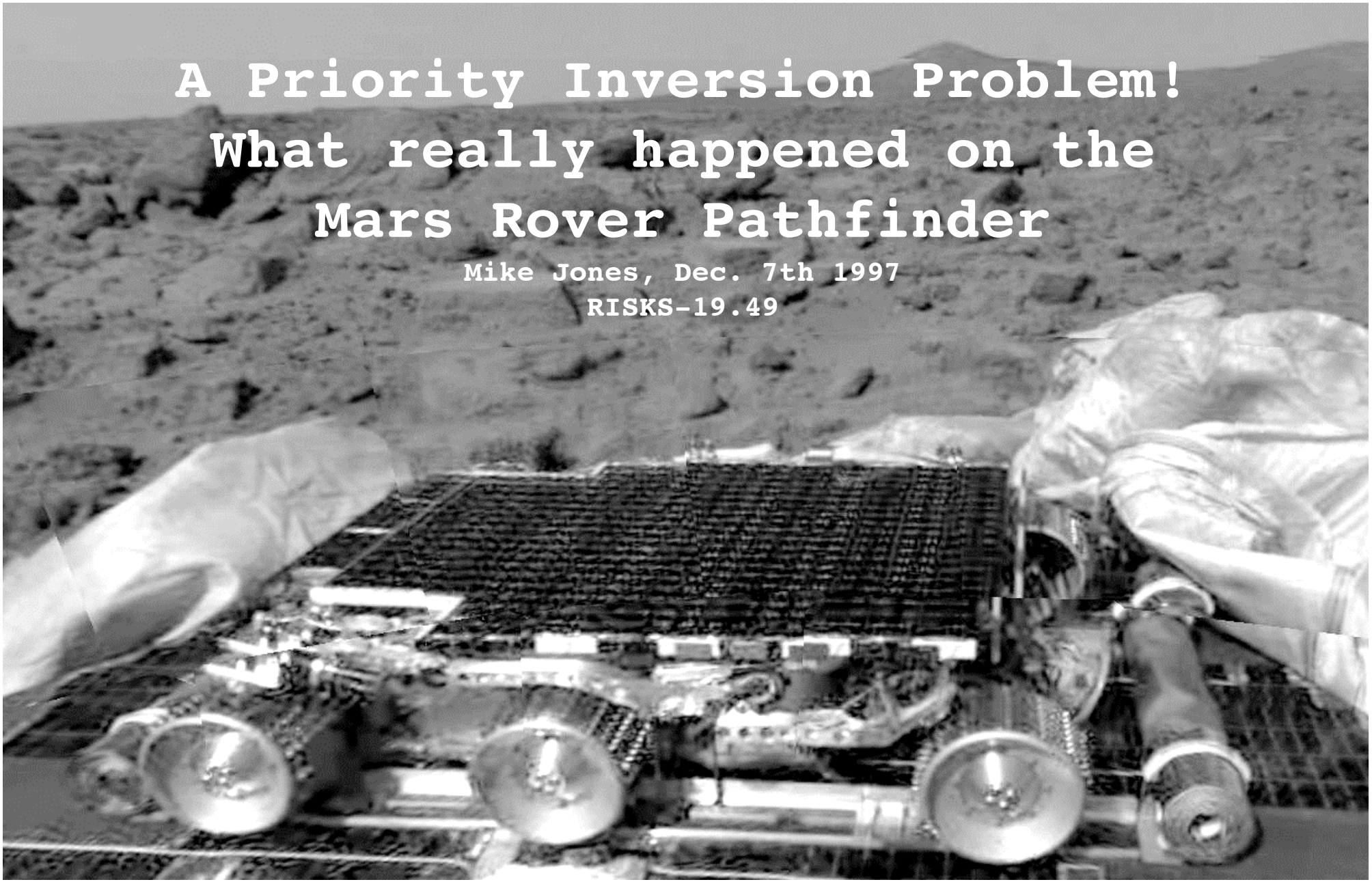
Planen und Synchronisation

1. Das Problem der Prioritätsumkehrung
   (Priority Inversion Problem)

2. Das Prioritätsvererbungsprotokoll
   (Priority Inheritance Protocol)

3. Das Prioritätshöchstgrenzenprotokoll
   (Priority Ceiling Protocol)

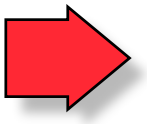


A Priority Inversion Problem! What really happened on the Mars Rover Pathfinder

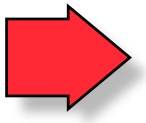
Mike Jones, Dec. 7th 1997

RISKS-19.49

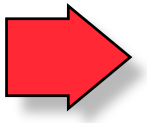
Grundproblem:
Abhängigkeit von Tasks, die auf eine gemeinsame Ressource zugreifen.



Da die Tasks nebenläufig ausgeführt werden, ergibt sich die Notwendigkeit der Synchronisation, um Inkonsistenzen zu vermeiden.

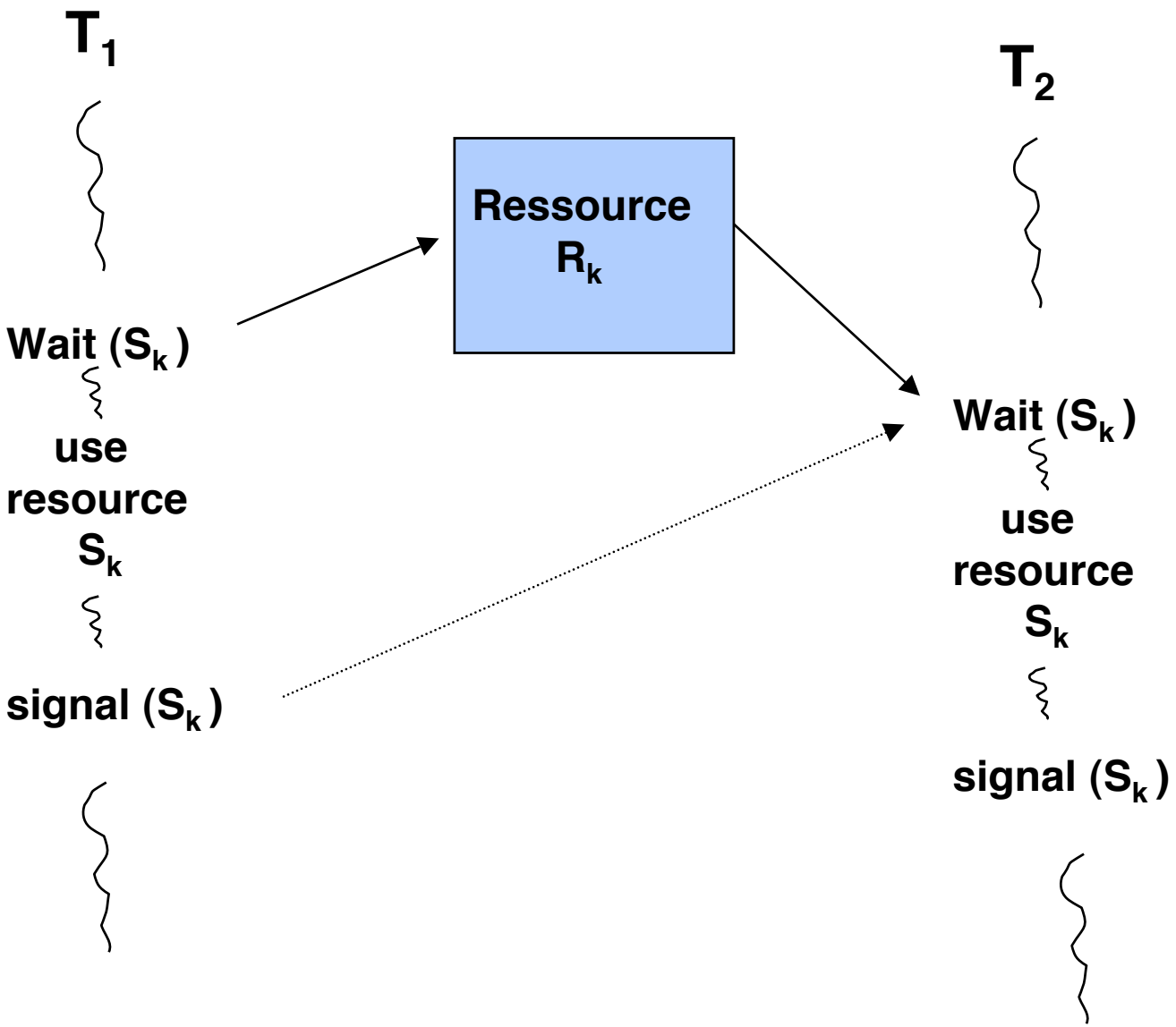


Synchronisation führt zur Verzögerung von Tasks, wenn die entsprechende Ressource nicht frei ist.

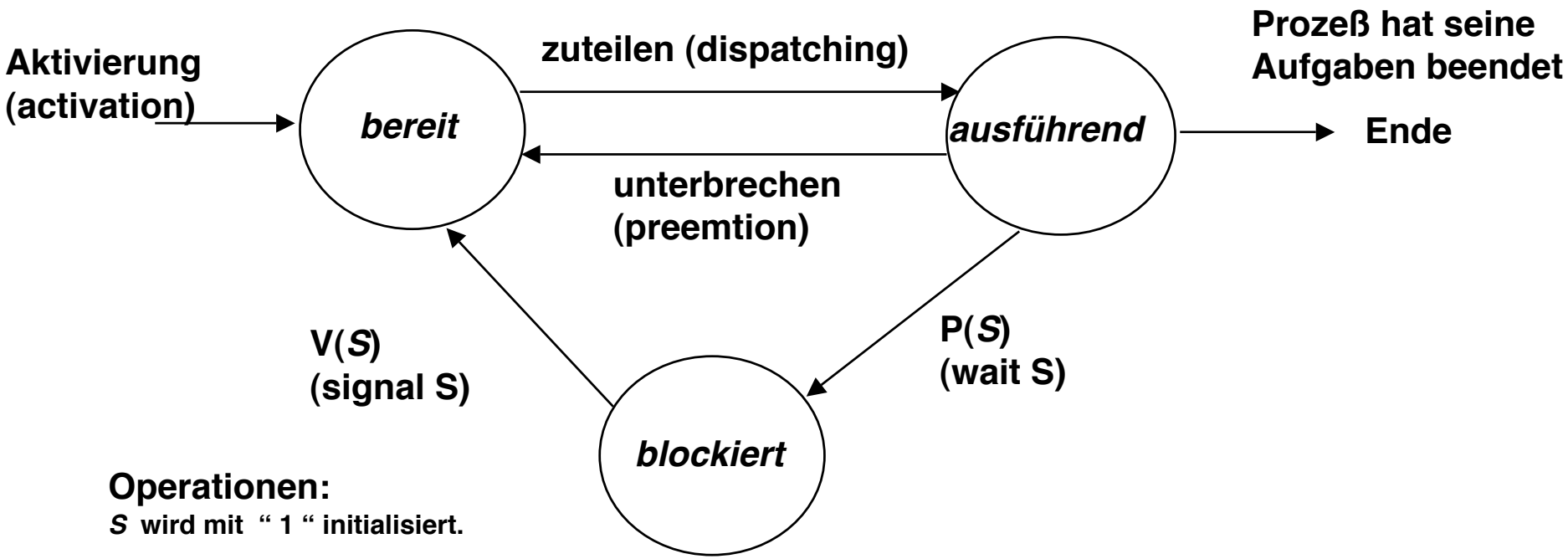


Al Mok hat gezeigt: Das Problem zu entscheiden, ob es einen brauchbaren Plan für periodische Tasks gibt, die Semaphore zum wechselseitigen Ausschluß benutzen ist NP-hart.

Synchronisation an einer gemeinsam genutzten Ressource



Semaphore



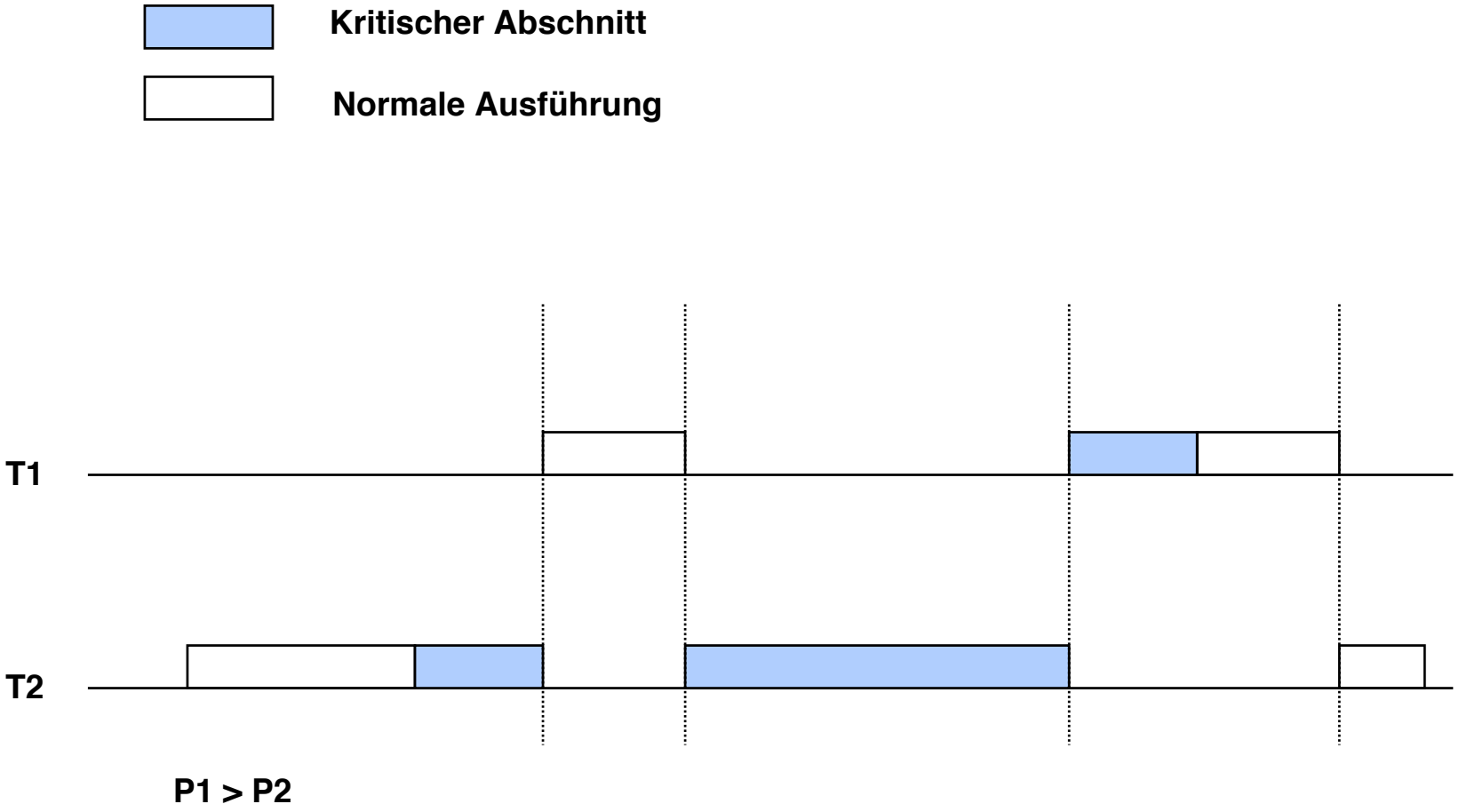
Operationen:
S **wird mit " 1 " initialisiert.**

wait(S)

```
P(S) {
     S = S - 1;
     if (S < 0)  blockiere (S);  /* Task wird in den Zustand blockiert überführt und in die Warteschlange eingereiht
     }
```

signal(S)

```
V(S) {
     S = S + 1;
     if (S ≤ 0) wecke (S);      /* Task wird in den Zustand bereit überführt und in die Ready Queue eingereiht
     }
```



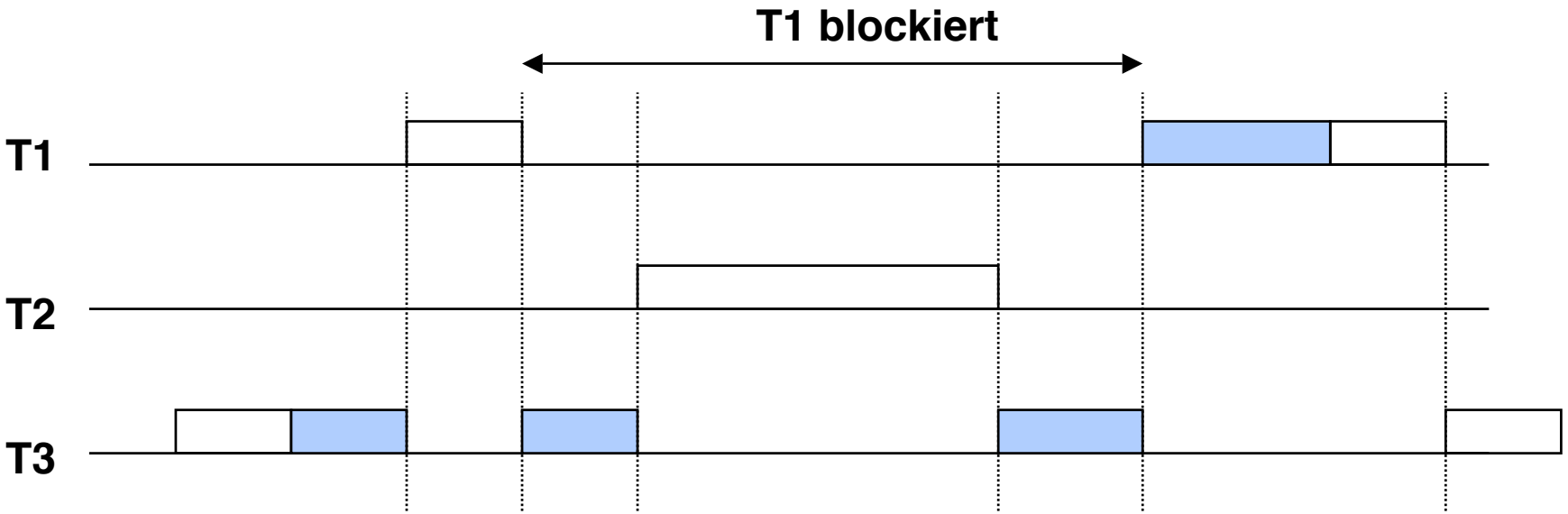
Einfache Form des Problems: T1 wird von T2 nur so lange blockiert, wie T2 braucht, um seinen kritischen Abschnitt zu bearbeiten. Danach wird *S* freigegeben.

Das Problem: Prioritätsumkehrung (Priority Inversion)

Annahme: Prioritätsbasiertes, unterbrechbares Scheduling.
Standard für die meisten RT-Kerne, ADA

Problem: nicht vorhersagbare Blockierung einer Task mit hoher Priorität durch Tasks mit niedrigerer Priorität.

Prioritätsumkehrung !



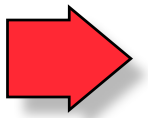
Applied Murphy:

T1 muß für eine unbestimmt lange Zeit warten, da Tasks i, die Prioritäten $p(T_4) < p(T_i) < p(T_1)$ haben, T_4 immer wieder unterbrechen oder nicht mehr zur Ausführung kommen lassen. T_1 wird also von niedriger priorisierten Prozessen für unbestimmte Zeit blockiert.

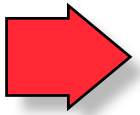
1. Lösungsmöglichkeit:

Wenn sich ein Prozeß in einem kritischen Abschnitt befindet, darf er nicht unterbrochen werden. Der *Kernelized Monitor* (Al Mok) basiert auf dieser (einfachen aber diskussionswürdigen) Lösung.

Problem: Alle anderen Prozesse höherer Priorität müssen warten, auch wenn sie von dem Prozeß in der kritischen Region völlig unabhängig sind, d.h. keine gemeinsamen Ressourcen anfordern werden.

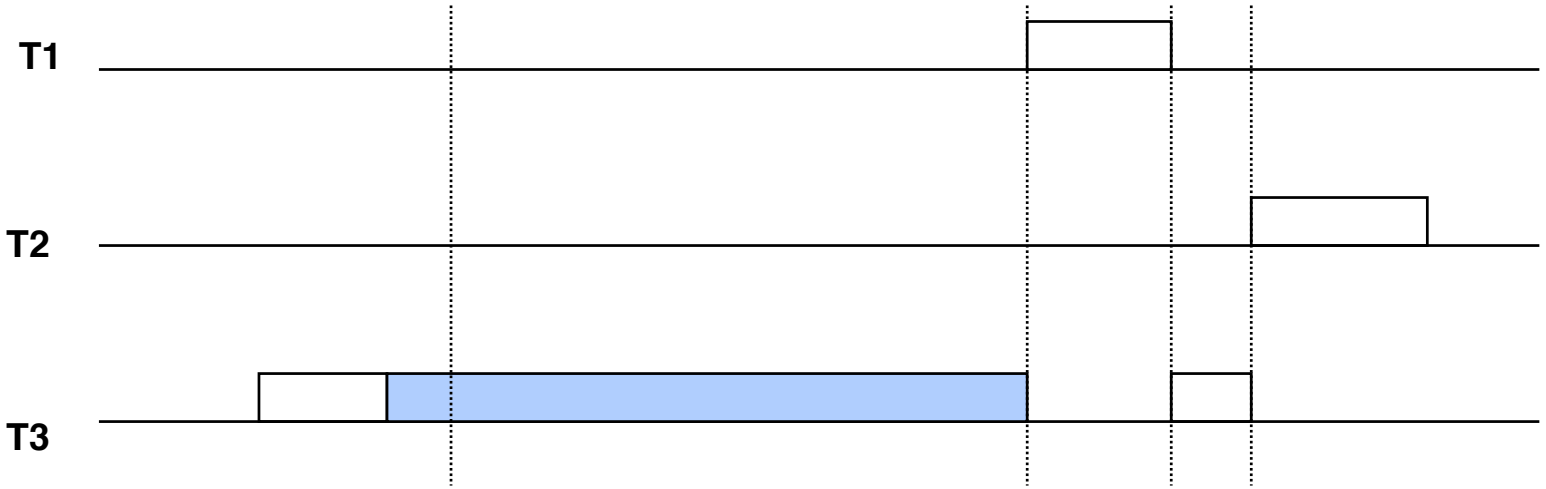


Kritische Abschnitte müssen kurz sein.



In jedem Fall ist dies eine gefährliche Situation in Echtzeitsystemen. <u>Für abhängige Prozesse ist die Situation nicht zu ändern !</u> (Möglicherweise schlechtes Design !) Aber es wäre völlig unbefriedigend, wenn in einer Alarmsituation ein kritischer Prozeß auf einen Prozeß warten muß, der z.B. mit niedrigster Priorität ein Bildschirm-Refresh durchführt und mit anderen Prozessen niedrigster Priorität gemeinsame Ressourcen benutzt.

Beispiel: Nicht unterbrechbare kritische Abschnitte



T1 wird von T3 blockiert obwohl keine Ressourcenkonflikte bestehen.

⇨ **indirektes Blockieren**

Unterscheidung zwischen:

Direktem Blockieren: **d.h. eine höher priorisierte Task T_i wird blockiert, weil sie mit einer niedriger priorisierten Task T_j um eine Ressource konkurriert. Im Beispiel wird T1 direkt von T4 blockiert**

Direktes Blockieren ist der Preis, der für die Konsistenz gemeinsam benutzter Daten bezahlt werden muß !

Indirektem Blockieren: **d.h. eine höher priorisierte Task T_i wird blockiert, weil eine niedriger priorisierte Task T_j gerade ihren kritischen Abschnitt ausführt.**
Task T_i hat keinen Ressourcekonflikt mit der niedriger priorisierten Task T_j, d.h. T_i und T_j sind unabhängig.

Indirektes Blockieren ist ein Artefakt, der von einer unzureichenden Problemlösung herrührt !

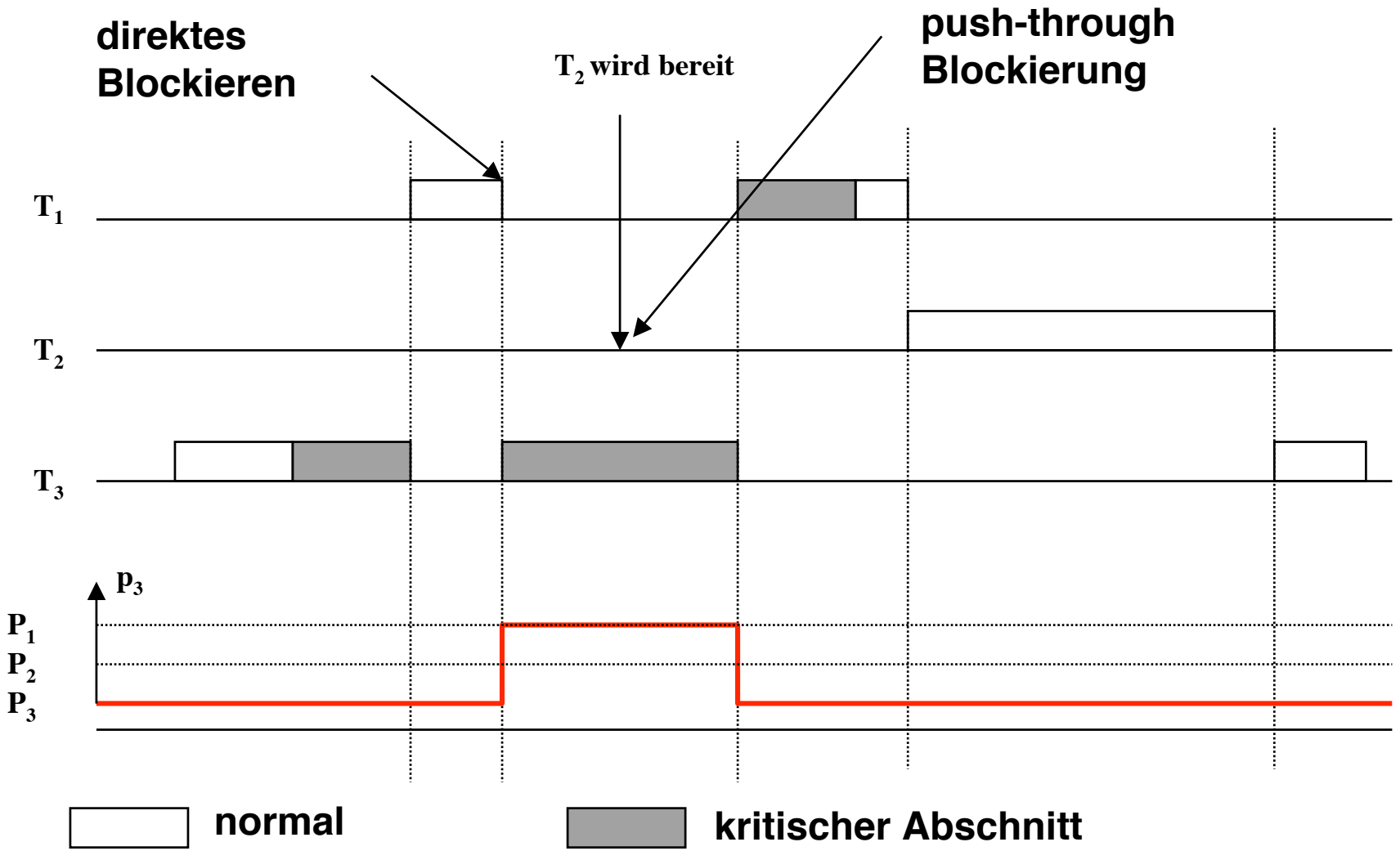
Das "Priority Inheritance Protocol "
(Prioritätsvererbung)

(L. Sha, R. Rajikumar, j. Lehoczky, CMU)

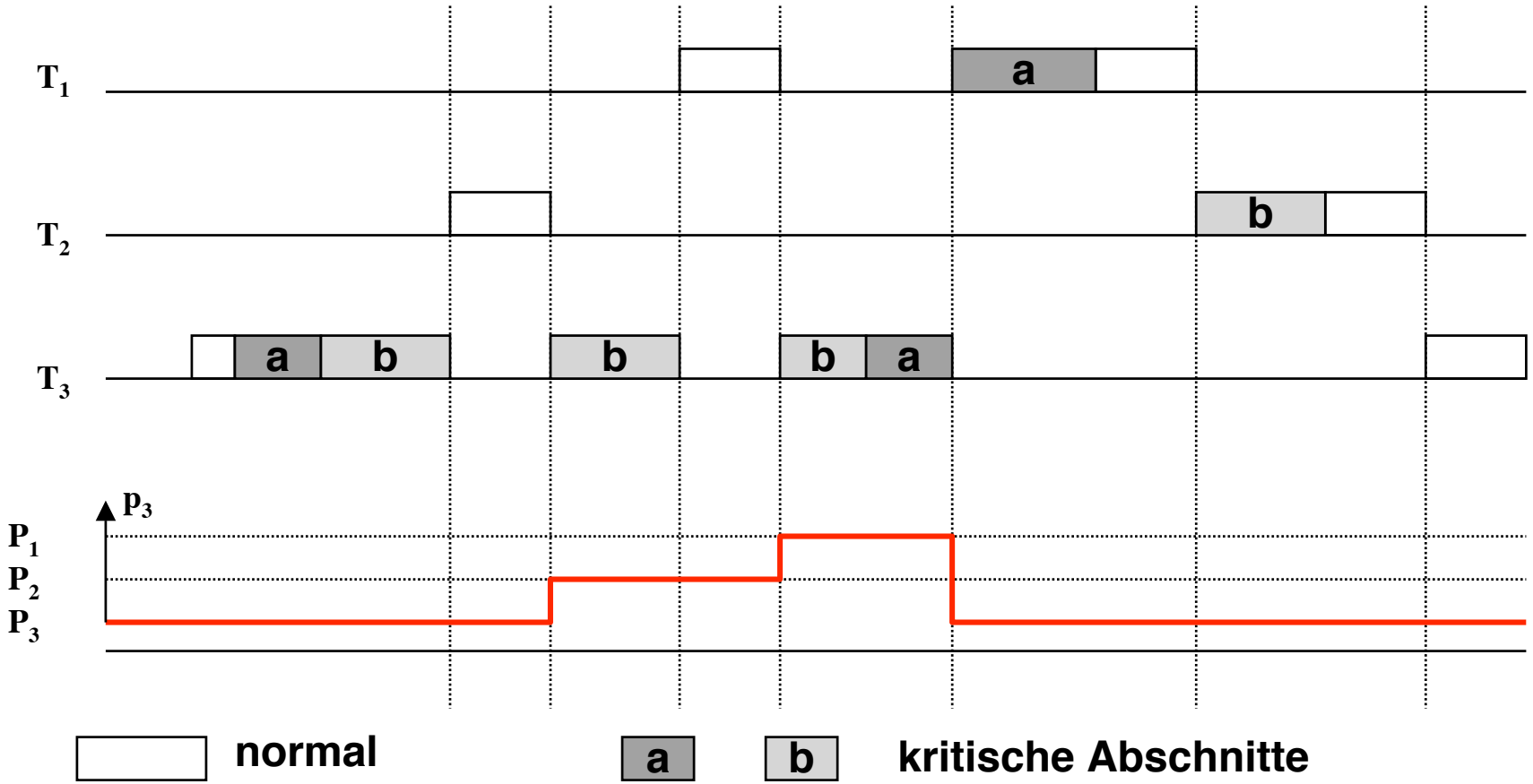
Ziele:

1. **Vermeidung von indirektem Blockieren**
2. **Obere Schranken für die Verzögerung von Prozessen**

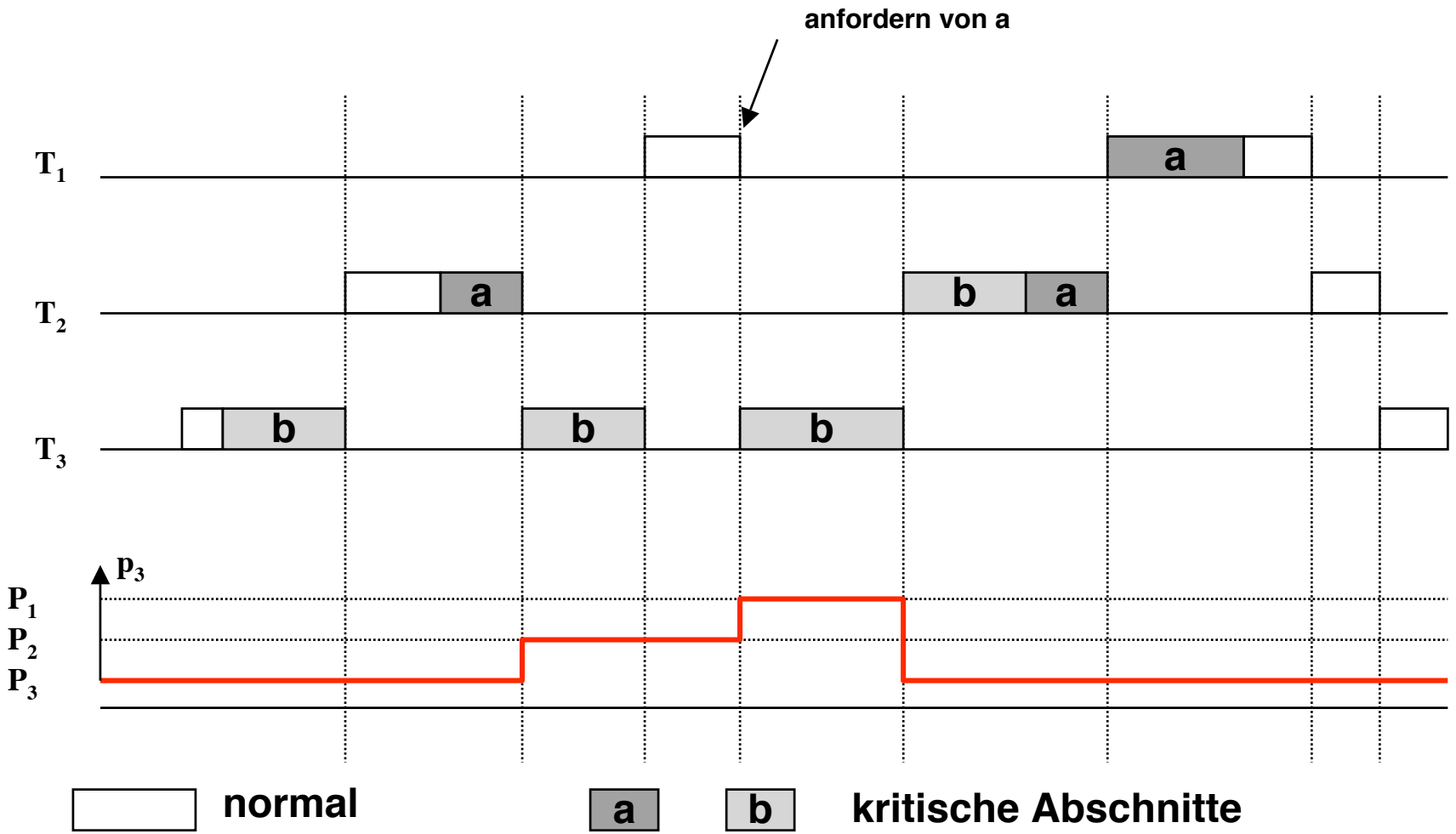
Beispiel zur Prioritätsvererbung



Beispiel zur Prioritätsvererbung mit geschachtelten kritischen Abschnitten



Beispiel zur transitiven Prioritätsvererbung



Satz (Sha, Rajkumar, Lehoczky):

Eine Task kann unter Verwendung des Prioritätsvererbungsprotokolls für höchstens min(n,m) kritische Abschnitte blockiert werden, wobei n die Anzahl der Tasks mit niedrigerer Priorität und m die Anzahl der unterschiedlichen Semaphore darstellt.

Planbarkeitsanalyse

1. Berechnung der kleinsten oberen Schranke für die Taskmenge

2. Berechnung der Blockierungszeiten

Zur Erinnerung: Resultat für RM:

Im Fall, dass keine Blockierungen auftreten, kann eine Taskmenge unter RM geplant werden, wenn gilt:

$$\sum_{i=1,..,n} \Delta e_j / \Delta p_j \leq n\,(2^{1/n}-1)$$

Wenn eine Task mit der Möglichkeit von Blockierungen durch Tasks mit niedrigerer Priorität erfolgreich eingeplant werden sollen, müssen folgende Faktoren berücksichtigt werden:

1. **Die Dauer der Unterbrechungen, die von Tasks höherer Priorität hervorgerufen werden:**
   $\sum_{k=1,..,i-1} \Delta e_k / \Delta p_k$

2. **Die Dauer der Ausführung von T_i selbst:**
   $(\Delta e_i / \Delta p_i)$

3. **Die Effekte der Blockierung durch Tasks mit niedrigerer Priorität:**
   $B_i / \Delta p_i$

Eine Menge periodischer Tasks, die das Prioritätsvererbungsprotokoll nutzen, kann nach RM geplant werden, wenn gilt:

$$\forall i,\ 1 \le i \le n, \qquad \sum_{k=1,..,i-1} \Delta e_k / \Delta p_k + \Delta e_i / \Delta p_i + B_i / \Delta p_i \le i\,(2^{1/i} - 1)$$

$$= \sum_{k=1,..,i} \Delta e_k / \Delta p_k + B_i / \Delta p_i \le i\,(2^{1/i} - 1)$$

Beispiel:

	T1	T2	T3
Δe_i	1	1	2
Δp_i	2	4	8
B_i	1	1	0

kann nicht durch niedriger priorisierte Task blockiert werden.

Perioden sind harmonisch, d.h. eine Auslastung von U=1 kann durch RM erreicht werden.

Folgende Ungleichungen müssen geprüft werden:

$$\Delta e_1/\Delta p_1 + B_1/\Delta p_1 = 1/2 + 1/2 \leq 1$$
$$\Delta e_1/\Delta p_1 + \Delta e_2/\Delta p_2 + B_2/\Delta p_2 = 1/2 + 1/4 + 1/4 \leq 1$$
$$\Delta e_1/\Delta p_1 + \Delta e_2/\Delta p_2 + \Delta e_3/\Delta p_3 = 1/2 + 1/4 + 2/8 \leq 1$$

Da alle Ungleichungen gelten, schließen wir, dass die Taskmenge planbar ist. Falls die k-te Ungleichung nicht gelten sollte, wissen wir, dass die Task T_k ihre Deadline verpassen würde. In diesem Fall müssten wir die Implementierung von T_k ändern, so dass ein möglicher Plan gefunden werden kann.

Berechnung der Blockierungszeiten

Notation:

σ_i: **die Menge der Semaphore, die von T_i benötigt werden**

$\beta_{i,j}$: **Menge aller kritischen Abschnitte, mit denen eine Task T_j mit niedrigerer Priorität als T_i, Task T_i blockieren kann (# der gemeinsamen S)**

$\gamma_{i,k}$: **Menge aller kritischen Abschnitte, die von Semaphor S_k geschützt werden, die T_i blockieren können. (# der Tasks mit gem. S)**

$Z_{i,k}$: **der längste kritische Abschnitt einer Task T_i, der von Semaphor S_k geschützt wird.**

$\Delta Z_{i,k}$: **Dauer von $Z_{i,k}$**

Annahme:

- **Alle kritischen Abschnitte $\Delta Z_{i,k}$ sind bekannt, bzw. können durch Analyse des Programmcodes ermittelt werden.**

Folgende Schritte sind notwendig, um B_i für T_i zu berechnen:

1. **Für alle Tasks T_j , deren Priorität* $P_j > P_i$ wird die Menge $\beta_{i,j}$ aller kritischen Abschnitte bestimmt, die T_i blockieren kann.** **Bestimmung der gemeins. Semaphore**

2. **Für alle Semaphore S_k wird die Menge $\gamma_{i,k}$ aller kritischen Abschnitte, die durch S_k geschützt werden, bestimmt, die T_i blockieren können.** **Bestimmung der Tasks, mit denen man gem. Semaphore nutzt.**

3. **Summieren der Dauer $\Delta Z_{i,k}$ der längsten kritischen Abschnitte in jeder Menge $\beta_{i,j}$ für jede Task T_i mit $P_j > P_i$. Sei B_i^l die Summe.** **(über die Menge der Semaphore)**

4. **Summieren der Dauer $\Delta Z_{i,k}$ der längsten kritischen Abschnitte in jeder Menge $\gamma_{i,k}$ jedes Semaphors S_k . Sei B_i^s die Summe.** **(über die Menge der Tasks)**

5. **Berechne B_i als Minimum zwischen B_i^l und B_i^s .**

*** Der höhere numerische Wert bezeichnet eine niedrigere Priorität!**

Beispiel:

	T_1	T_2	T_3	T_4
$S_1(P_1)$	$\Delta Z_{1,1}$	$\Delta Z_{2,1}$	$\Delta Z_{3,1}$	$\Delta Z_{4,1}$
$S_2(P_1)$	$\Delta Z_{1,2}$	$\Delta Z_{2,2}$	$\Delta Z_{3,2}$	$\Delta Z_{4,2}$
$S_3(P_2)$	$\Delta Z_{1,3}$	$\Delta Z_{2,3}$	$\Delta Z_{3,3}$	$\Delta Z_{4,3}$

	T_1	T_2	T_3	T_4
$S_1(P_1)$	1	-	8	6
$S_2(P_1)$	2	9	7	5
$S_3(P_2)$	-	3	-	4

$T_1 : \beta_{1,2}, \beta_{1,3}, \beta_{1,4}$

$\beta_{1,2} = \{\Delta Z_{2,2}\} = \{9\}$

$\beta_{1,3} = \{\Delta Z_{3,1}, \Delta Z_{3,2}\} = \{8,7\}$

$\beta_{1,4} = \{\Delta Z_{4,1}, \Delta Z_{4,2}\} = \{6,5\}$

$T_2 : \beta_{2,3}, \beta_{2,4}$

$\beta_{2,3} = \{\Delta Z_{3,2}\} = \{7\}$

$\beta_{2,4} = \{\Delta Z_{4,2}, \Delta Z_{4,3}\} = \{5,4\}$

$T_3 : \beta_{3,4},$

$\beta_{3,4} = \{\Delta Z_{4,1}, \Delta Z_{4,2}\} = \{6,5\}$

$S_1 | T_1 : \gamma_{1,1} = \{\Delta Z_{3,1}, \Delta Z_{4,1}\} = \{8, 6\}$

$S_1 | T_2 : \gamma_{2,1} = \{\varnothing\}$

$S_1 | T_3 : \gamma_{3,1} = \{\Delta Z_{4,1}\} = \{6\}$

$S_2 | T_1 : \gamma_{1,2} = \{\Delta Z_{2,2}, \Delta Z_{3,2}, \Delta Z_{4,2}\} = \{9, 7,5\}$

$S_2 | T_2 : \gamma_{2,2} = \{\Delta Z_{3,2}, \Delta Z_{4,2}\} = \{7,5\}$

$S_2 | T_3 : \gamma_{1,2} = \{\Delta Z_{4,2}\} = \{5\}$

$S_3 | T_1 : \gamma_{1,3} = \{\varnothing\}$

$S_3 | T_2 : \gamma_{2,3} = \{\Delta Z_{4,3}\} = \{4\}$

$S_3 | T_3 : \gamma_{2,2} = \{\varnothing\}$

B_1^l = 9 + 8 + 6 = 23

B_1^s = 8 + 9 = 17

B_1= 17

B_2^l = 7 + 5 = 12

B_2^s = 7 + 4 = 11

B_2= 11

B_3^l = 6 = 6

B_3^s = 6+5 = 11

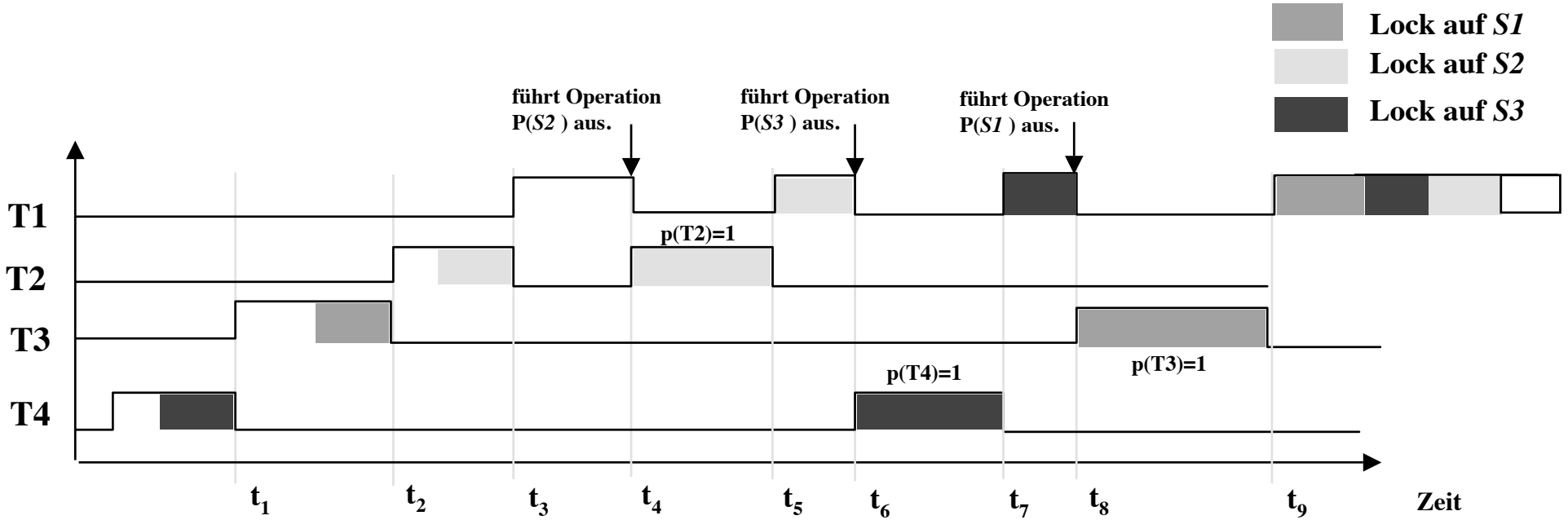
B_3= 6

B_4 = 0

B_4= 0

Das Priority Inheritance Protocol - Immer noch nicht zufriedenstellend !

Problem 1 -Blockierungsketten

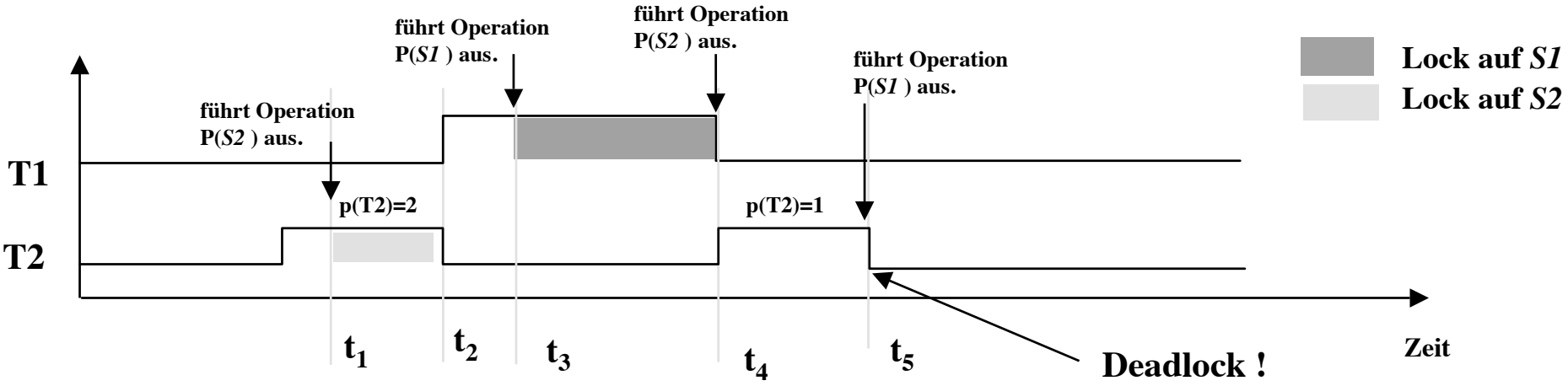


Die Task T1 wird nacheinander durch die Tasks T4, T3 und T2 blockiert.
Man spricht von einer Blockierungskette (Chain of Blocking). Die Vorhersagbarkeit der Blockierungsdauer ist nicht gewährleistet.

Die Worst-Case-Blockierungsdauer ist zwar beschränkt.
Es kann aber nicht zur Laufzeit bestimmt werden, wie viele Glieder die Blockierungskette tatsächlich hat.

Das Priority Inheritance Protocol - Immer noch nicht zufriedenstellend !

Problem 2 - Deadlock Situationen:



t_1 : T2 führt die Operation P(*S2*) aus, erhält den Lock und tritt in den kritischen Abschnitt ein.
t_2 : T1 unterbricht T2, da T1 eine höhere Priorität hat.
t_3 : T1 führt die Operation P(*S1*) aus, erhält den Lock und tritt in den kritischen Abschnitt ein.
t_4 : T1 führt die Operation P(*S2*) aus und blockiert, da T2 den Lock auf *S2* hält.
T2 erhält dadurch die Priorität von T1 und fährt in der Ausführung seines kritischen Abschnitts fort.
Dabei führt er die Operation P(*S1*) aus und blockiert seinerseits, da T1 den Lock auf *S1* hält.

Das Priority Inheritance Protocol kann diese Deadlock-Situation nicht auflösen !
Es verfehlt daher das Ziel der Vorhersagbarkeit der maximalen Verzögerung unter der Annahme gemeinsam genutzter Ressourcen.

Das Priority Ceiling Protocol (Prioritätshöchstgrenzen)

(L. Sha, R. Rajikumar, j. Lehoczky, CMU)

Ziele:

- **Die Ziele des Priority Inheritance Protocols**
- **Vermeidung von Deadlocks**
- **Vermeidung von Blockierungsketten**

Def.:

Die Prioritätshöchstgrenze (Priority Ceiling) eines Semaphors S_k wird definiert als die höchste Priorität, unter der ein kritischer Abschnitt, geschützt durch S_k, je ausgeführt wird.

$$C(S_k) = \max (P_i : S_k \in \sigma_i)$$

σ_i: Menge der Semaphore, die T_i benötigt

Grundidee:

Eine Task kann ein Semaphor nur sperren kann, wenn ihre Priorität höher ist, als die Prioritätshöchstgrenze aller Semaphore, die bereits gesperrt sind.

Einer Task wird daher nicht erlaubt einen kritischen Abschnitt zu betreten, wenn es bereits gesperrte Semaphore gibt, an denen sie blockieren könnte. D.h. eine Task , die einmal ihren ersten kritischen Abschnitt betritt wird bis zum Abschluss nicht von Tasks mit niedrigerer Priorität blockiert.

Grundidee (cont.):

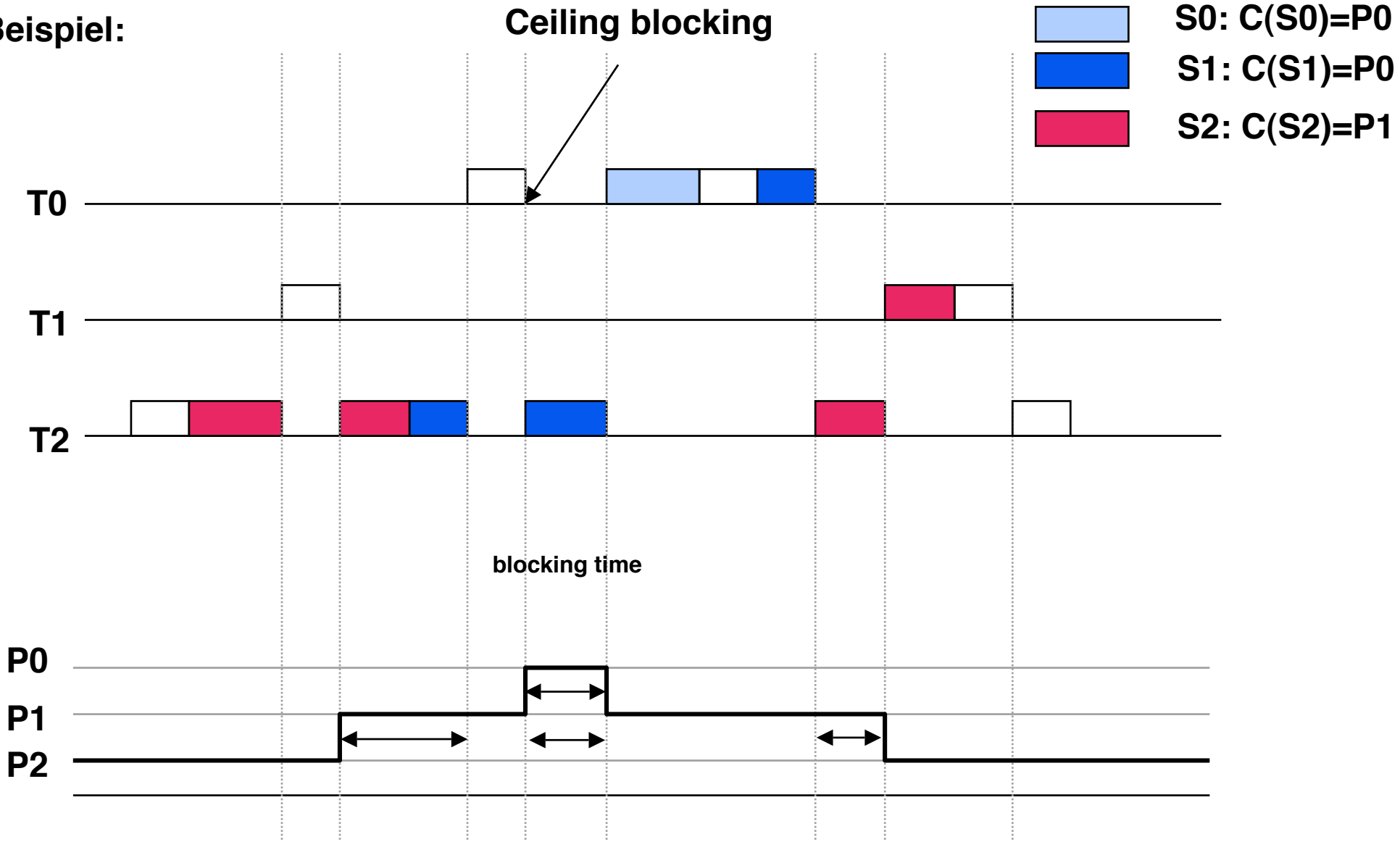
- **Falls noch kein Semaphor gesperrt ist, kann jede Task ein Semaphor sperren und ihren kritischen Abschnitt betreten.**

- **Falls irgendeine Task ein Semaphor gesperrt hat, wird die Regel angewendet, dass die Task die ein Semaphor sperren will, dies nur kann, wenn ihre Priorität höher ist, als die PHG aller bereits gesperrten Semaphore.**

  Dies bedeutet, dass die Task kein Semaphor aus der Menge der bereits gesperrten Semaphore benutzt, denn sonst läge ihre Priorität ja nicht über der Höchstgrenze !!

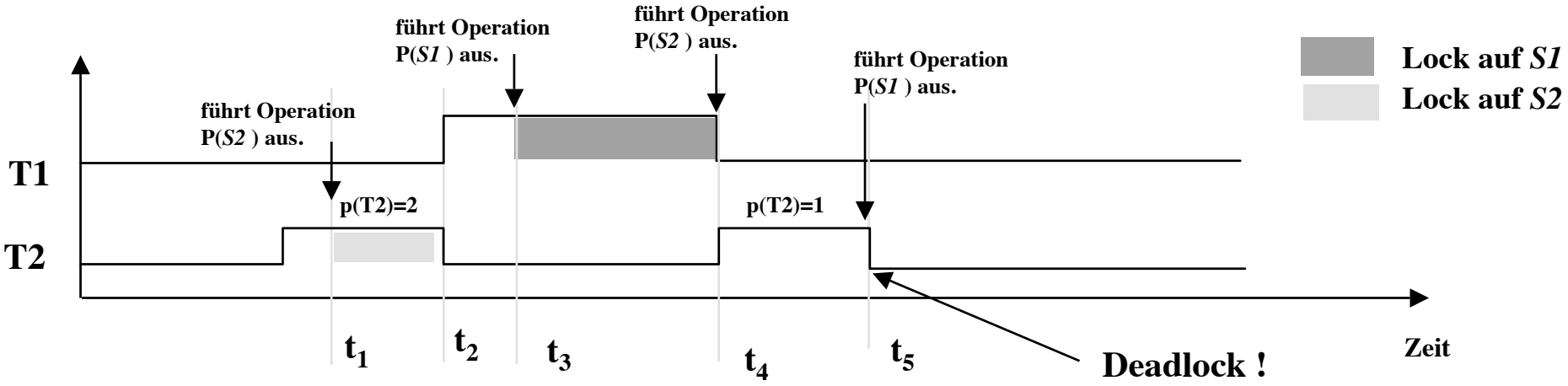
  D.h. PHP verhindert präventiv, dass eine Situation für Blockierungsketten oder Deadlocks entstehen kann.

Beispiel:

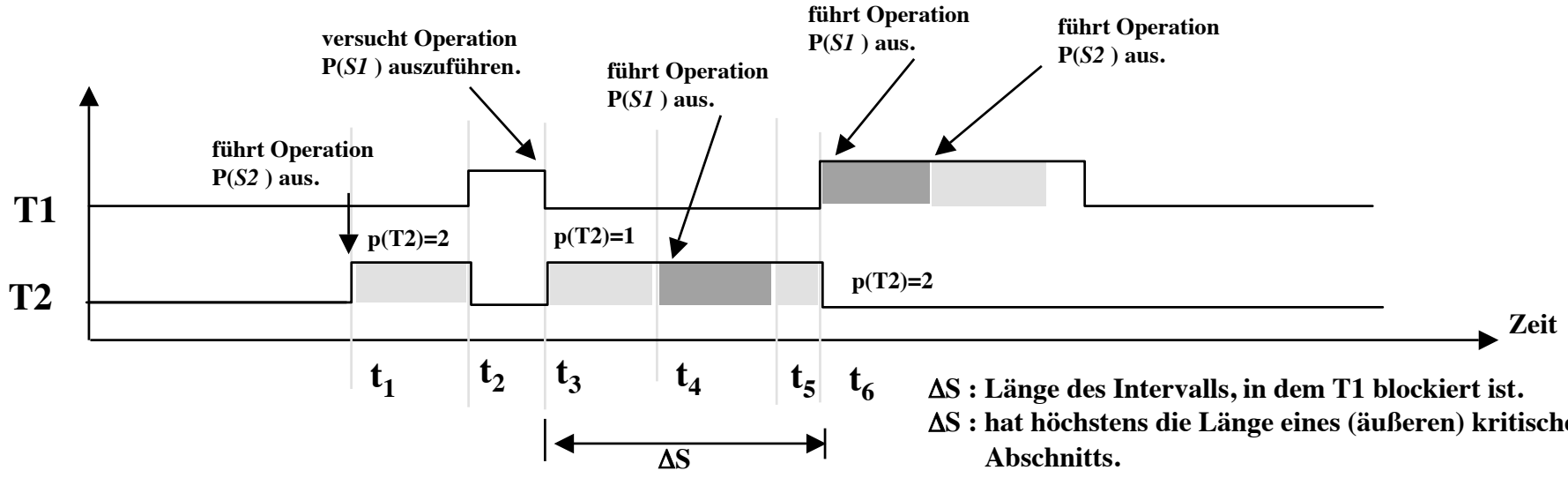

Ceiling blocking
S0: C(S0)=P0
S1: C(S1)=P0
S2: C(S2)=P1
T0
T1
T2
blocking time
P0
P1
P2

Problem 1 - Deadlock Situationen:

Priority Inheritance Protocol:

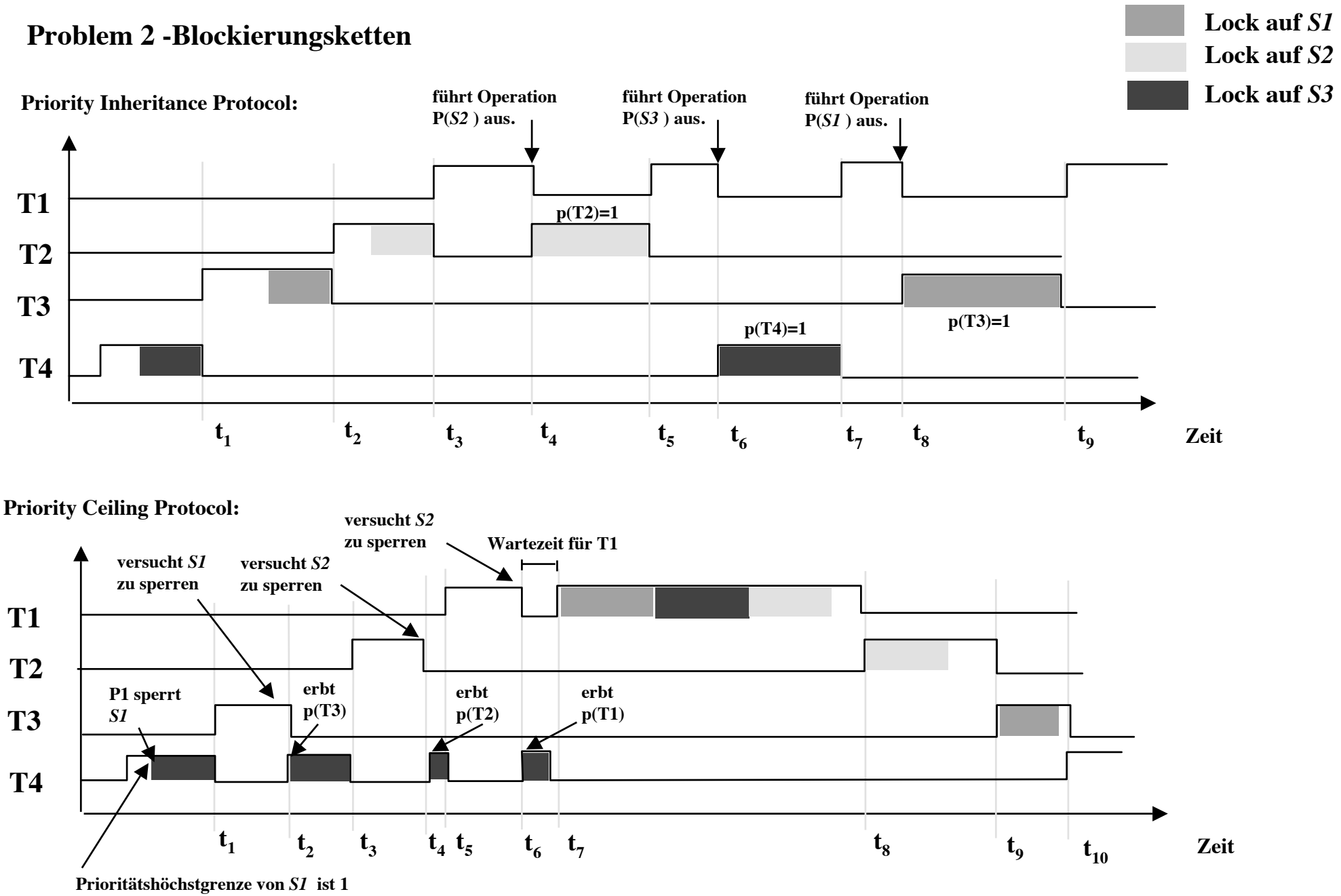


Priority Ceiling Protocol:



ΔS : Länge des Intervalls, in dem T1 blockiert ist.
ΔS : hat höchstens die Länge eines (äußeren) kritischen Abschnitts.

Problem 2 -Blockierungsketten



Eigenschaften des Priority Ceiling Protocol

- **Das Priority Ceiling Protocol vermeidet Deadlocks**

  Folge: Programmierer können beliebige Folgen von geschachtelten Zugriffen auf Semaphore definieren. Solange ein Prozeß nicht mit sich selbst einen Deadlock bildet, ist das System Deadlock-frei.

- **Unter den Annahmen des Priority Ceiling Protocol kann eine Task von einer anderen Task mit niedrigerer Priorität höchstens für die Dauer *eines* kritischen Abschnitts blockiert werden.**

Kritik: **Das Priority Ceiling Protocol nimmt Worst Case Bedingungen bezüglich der Prioritäten an. D. h. ein Prozeß mit niedriger Priorität kann einen Prozeß höherer Priorität für die Dauer eines kritischen Abschnitts blockieren, obwohl das von der Anwendung her nicht notwendig wäre.**

Planbarkeitsanalyse

Lemma:
Unter Anwendung des PCP kann ein kritischer Abschnitt $Z_{j,k}$ (gehört zu Task T_j und wird durch Semaphore S_k geschützt) eine andere Task T_i nur dann blockieren, wenn gilt:

$$P_j > P_i \text{ und } C(S_k) \leq P_i$$

Daraus folgt für die Blockierungsdauer B_i für Task T_i :

$$B_j = \max_{j,k} \{ \Delta D_{j,k} \mid P_j > P_i ,\ C(S_k) \leq P_i \}$$

$\Delta D_{j,k}$ bezeichnet die Dauer des längsten kritischen Abschnitts aller Tasks (hier: T_j) die eine geringere Priorität als Task T_i haben und von Semaphor S_k geschützt werden.

Beispiel:

$C(S_k)$

	T_1	T_2	T_3	T_4
$S_1 (P_1)$	1	-	8	6
$S_2 (P_1)$	2	9	7	5
$S_3 (P_2)$	-	3	-	4

B_1= max (8, 6, 9, 7, 5) = 9
B_2= max (8, 6, 7, 5, 4) = 8
B_3= max (6, 5, 4) = 6
B_4= 0

T_1 kann von T_2, T_3 , T_4 an S_1 und S_2 blockiert werden

T_2 kann von T_3 , T_4 an S_2 und S_3 und indirekt auch an S_1 blockiert werden

T_3 kann von T_4 an S_1 und S_2 und indirekt auch an S_3 blockiert werden