Planen und Synchronisation

1. Das Problem der Prioritätsumkehrung
   (Priority Inversion Problem)

2. Das Prioritätsvererbungsprotokoll
   (Priority Inheritance Protocol)

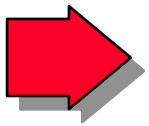
3. Das Prioritätshöchstgrenzenprotokoll
   (Priority Ceiling Protocol)

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

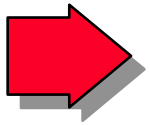
Mike Jones, Dec. 7th 1997

RISKS-19.49

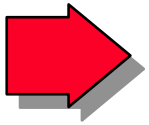
Grundproblem:
Abhängigkeit von Tasks, die auf eine gemeinsame Ressource zugreifen.



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

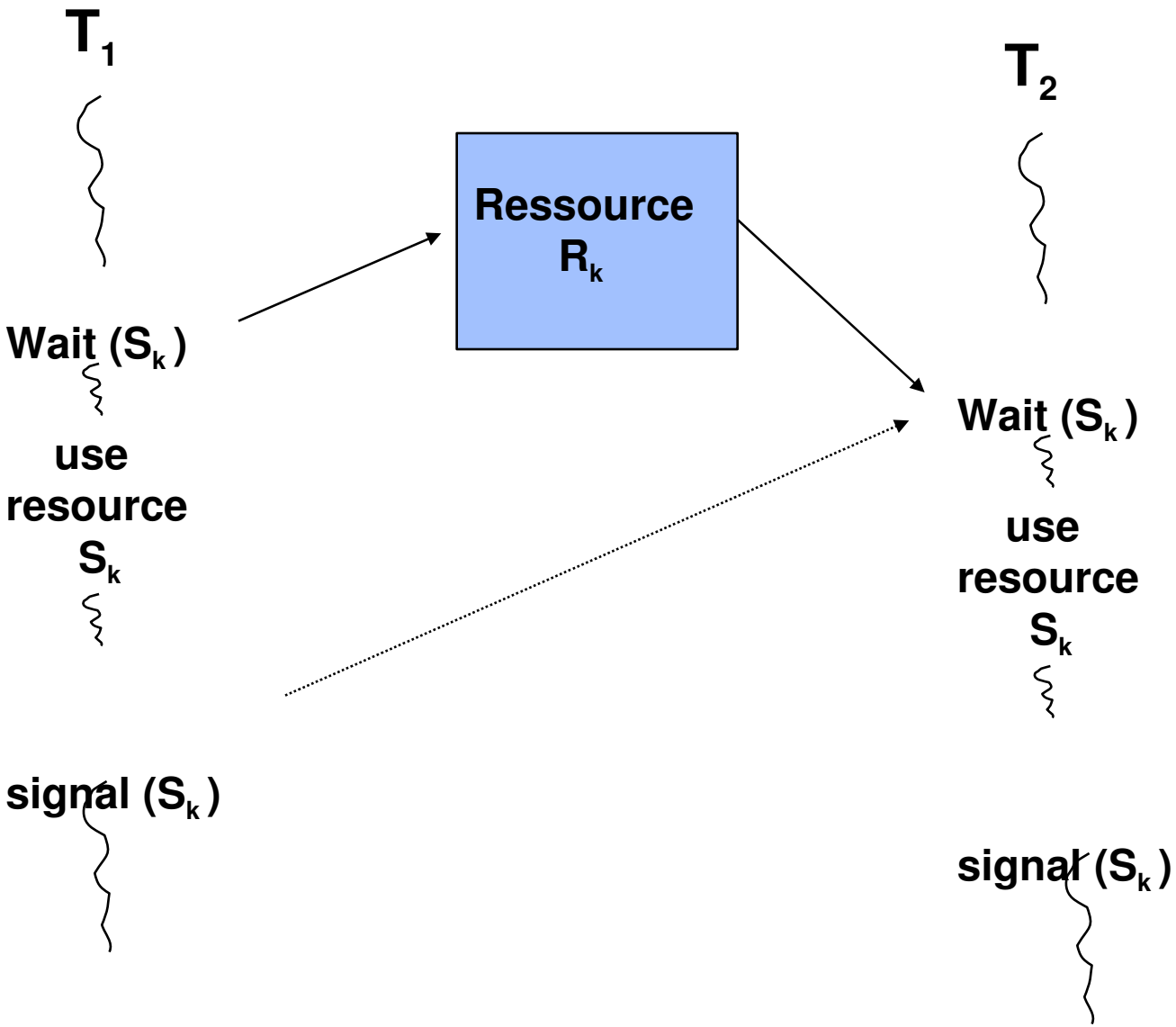


Synchronisation führt zur Verzögerung von Tasks, wenn die entsprechende Ressource nicht frei ist.

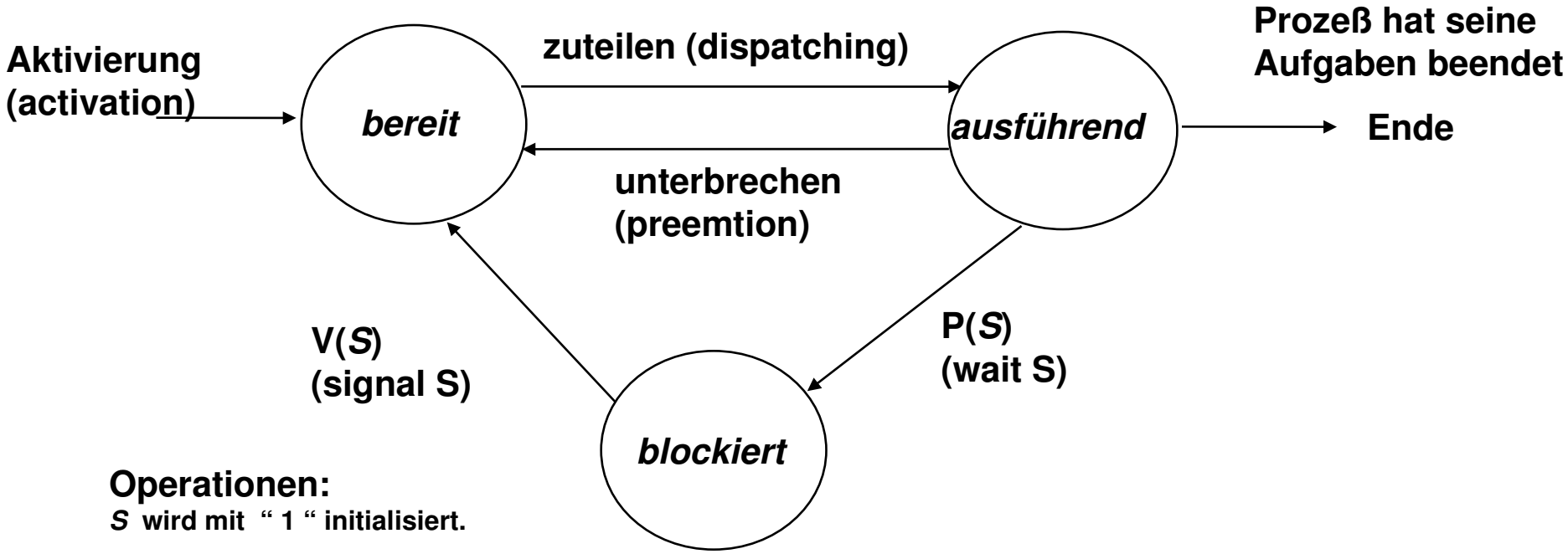


Al Mok hat gezeigt: Das Problem zu entscheiden, ob es einen brauchbaren Plan für periodische Tasks gibt, die Semaphore zum wechselseitigen Ausschluß benutzen ist NP-hart.

Synchronisation an einer gemeinsam genutzten Ressource



Semaphore



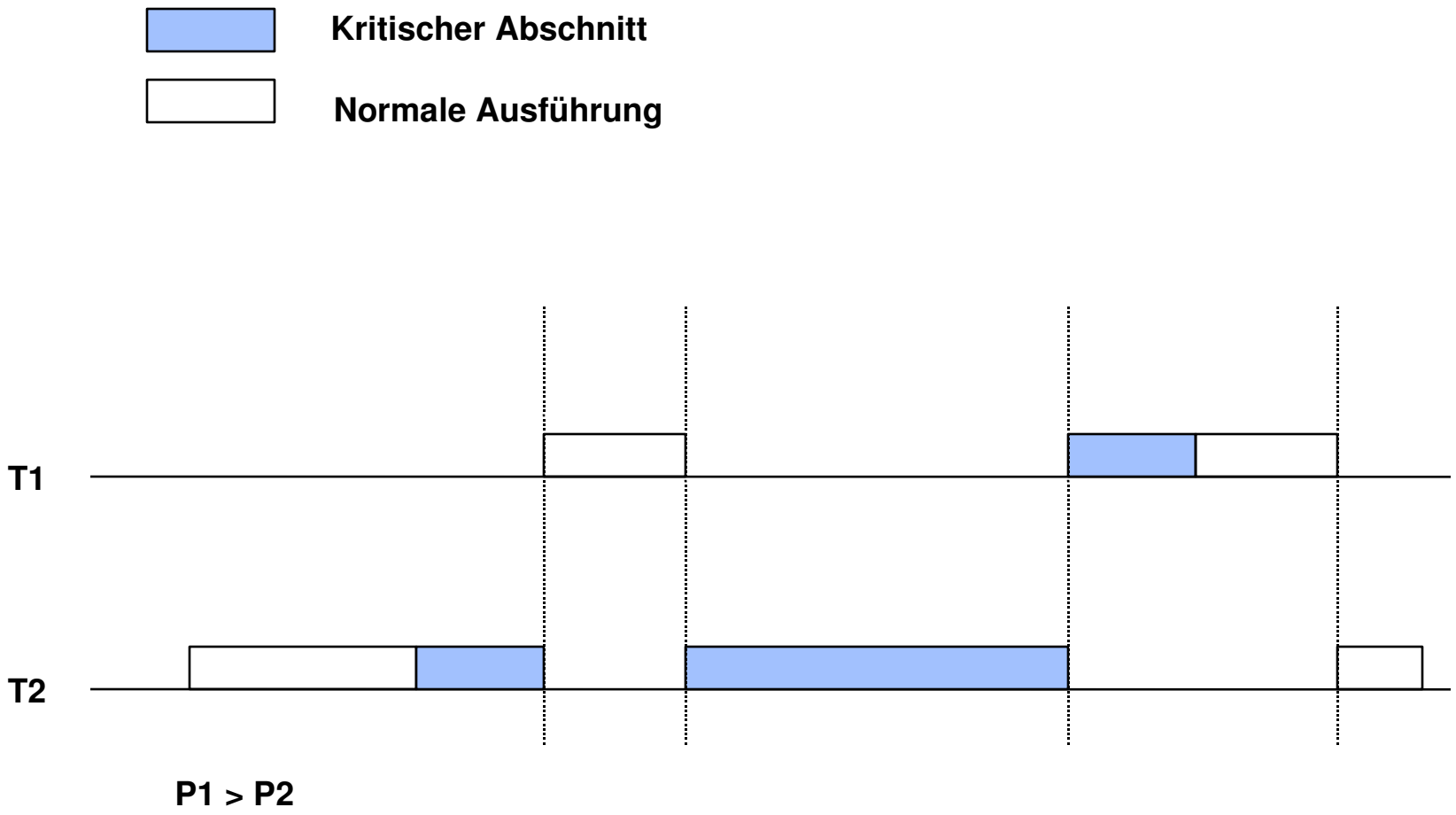
Operationen:
S **wird mit " 1 " initialisiert.**

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
     }
```



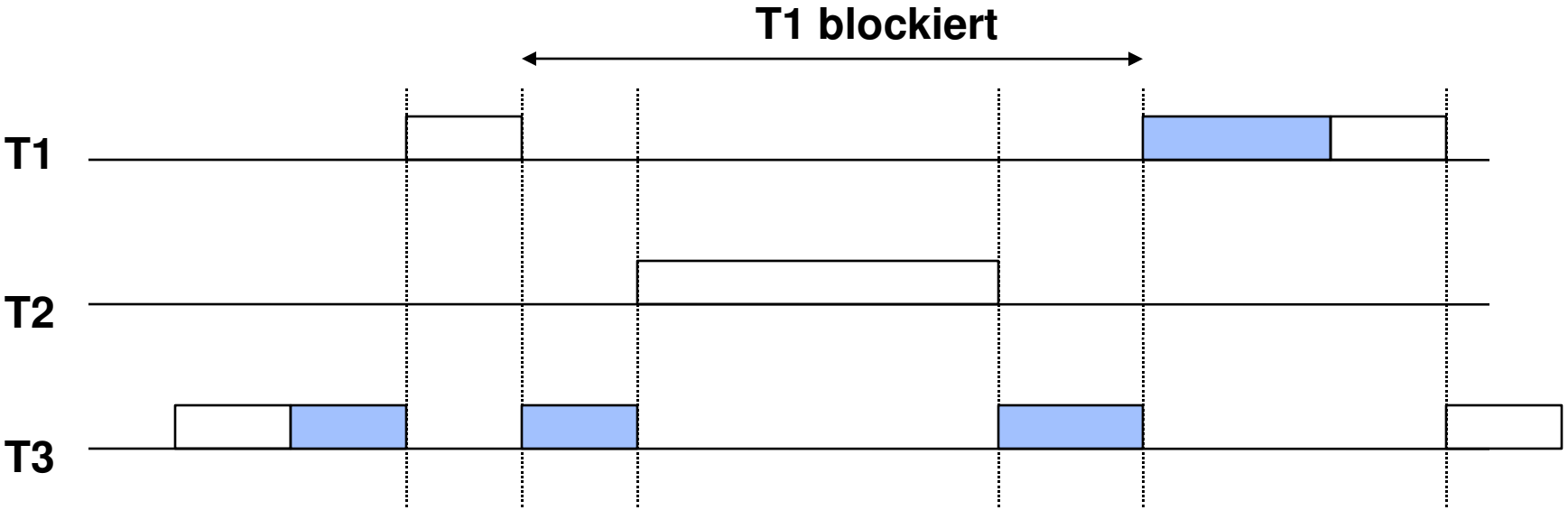
Einfache Form des Problems: T1 wird von T2 nur so lange blockiert, wie T2 braucht, um seinen kritischen Abschnitt zu bearbeiten. Danach wird *S* freigegeben.

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 !



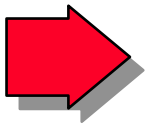
Applied Murphy:

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.

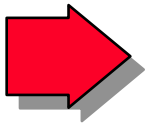
1. Lösungsmöglichkeit:

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.

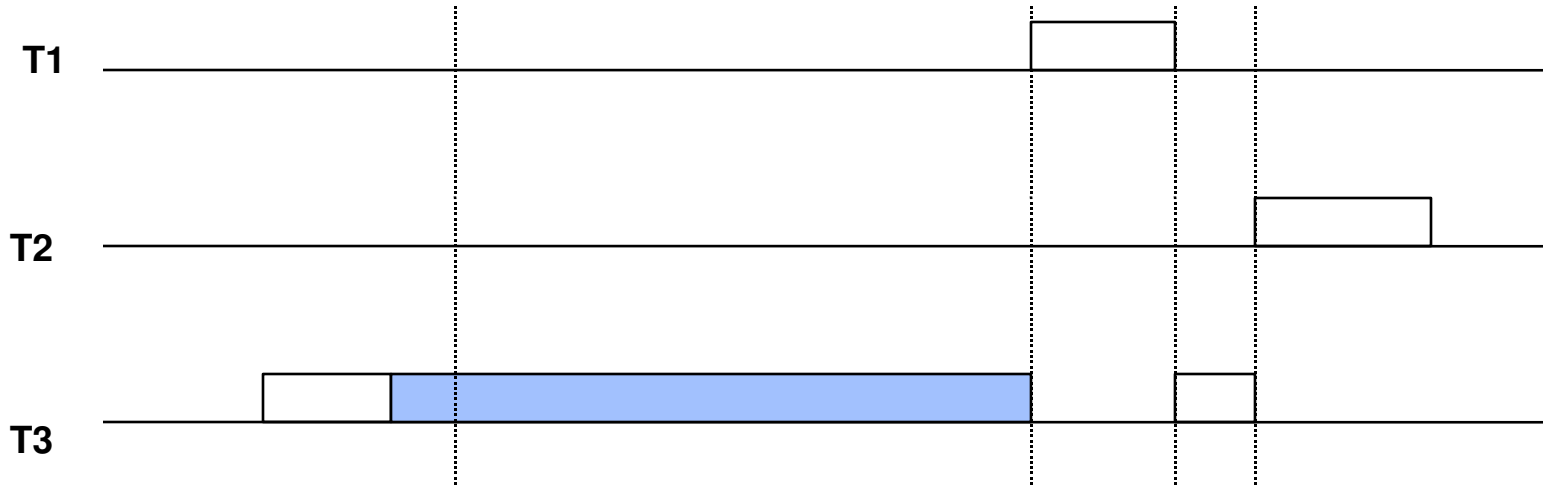


Kritische Abschnitte müssen kurz sein.



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 !

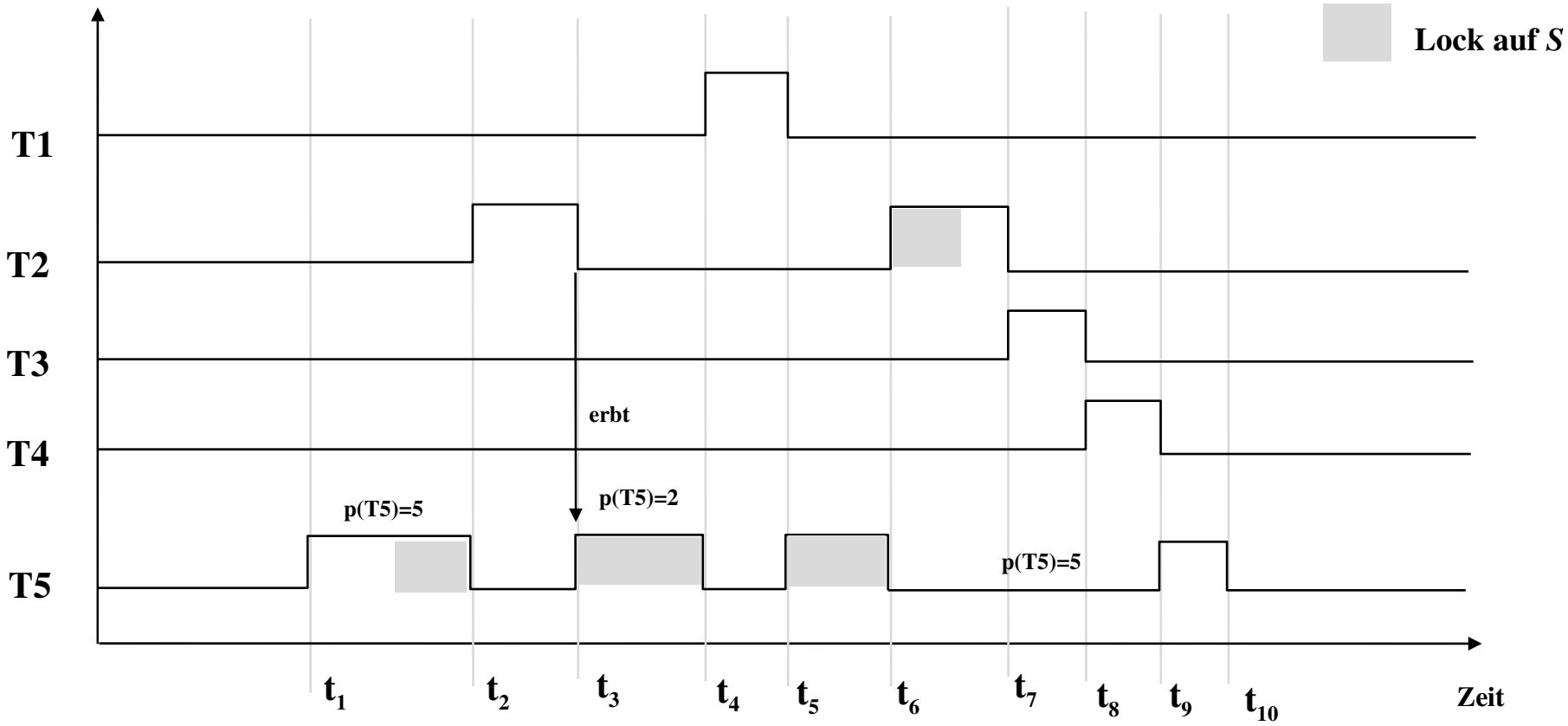
Das "Priority Inheritance Protocol "
(Prioritätsvererbung)

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

Ziele:

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

Beispiel 1 zum Priority Inheritance Protocol: **T2 und T5 nutzen gemeinsam eine durch S geschützte Ressource**



t_1 : T5 führt P(*S*) aus und erhält den Lock auf der gemeinsam benutzten Ressource.

t_2 : T2 unterbricht T5.

t_3 : T2 führt P(*S*) aus und wird blockiert. Dadurch erbt T5 die Priorität von T2 ,wird zugeteilt und kann weiter in seinen kritischen Abschnitt ausführen.

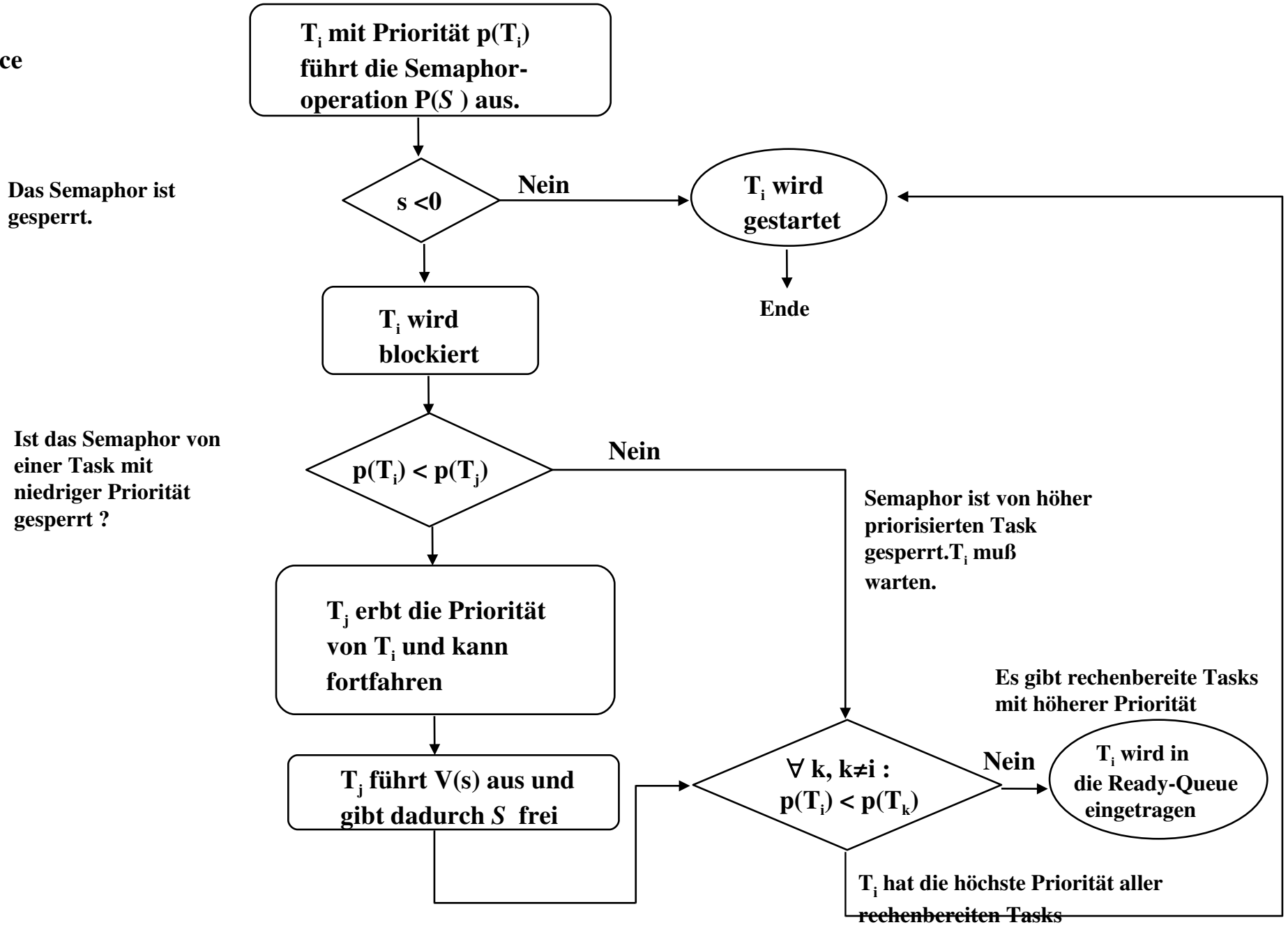
t_4 : T1 unterbricht T5. T1 ist unabhängig von T5. Da T1 eine höhere Priorität hat als die aktive Priorität von T5, erhält T1 die Prozessorzuteilung.

t_5 : T1 wird beendet und T5 wird zugeteilt.

t_6 : T5 beendet seinen kritischen Abschnitt und führt V(*S*) aus. Daraufhin wird T2 bereit, unterbricht T5, führt P(*S*) aus und kann seine Aufgabe, nachdem er V(*S*) ausgeführt hat, beenden.

t_{10} : NachdemT3 und T4 zum Zuge gekommen und beendet sind, kann T5 schließlich seine Aufgabe beenden.

Priority Inheritance Protocol



Priority Inheritance Protocol

1.) Regeln, die den Erwerb eines Locks beteffen:

- **T_i hat die höchste aktive Priorität aller rechenbereiter Tasks. Der Prozessor wird T_i zugeteilt.**
- **Bevor T_i ihren kritischen Abschnitt ausführen kann, muß sie eine Sperre (Lock) auf Semaphor *S* erhalten, durch das der kritische Abschnitt geschützt wird.**
- **T_i wird blockiert, wenn *S* bereits gesperrt ist. Man sagt: T_i wird durch die Task blockiert, die *S* gesperrt hat.**
- **Falls keine Sperre existiert, sperrt T_i Semaphor *S* und tritt in den kritischen Abschnitt ein.**
- **Wenn T_i den kritischen Abschnitt verlässt, wird die Sperre auf *S* aufgehoben und alle möglicherweise durch T_i blockierten Tasks werden rechenbereit.**

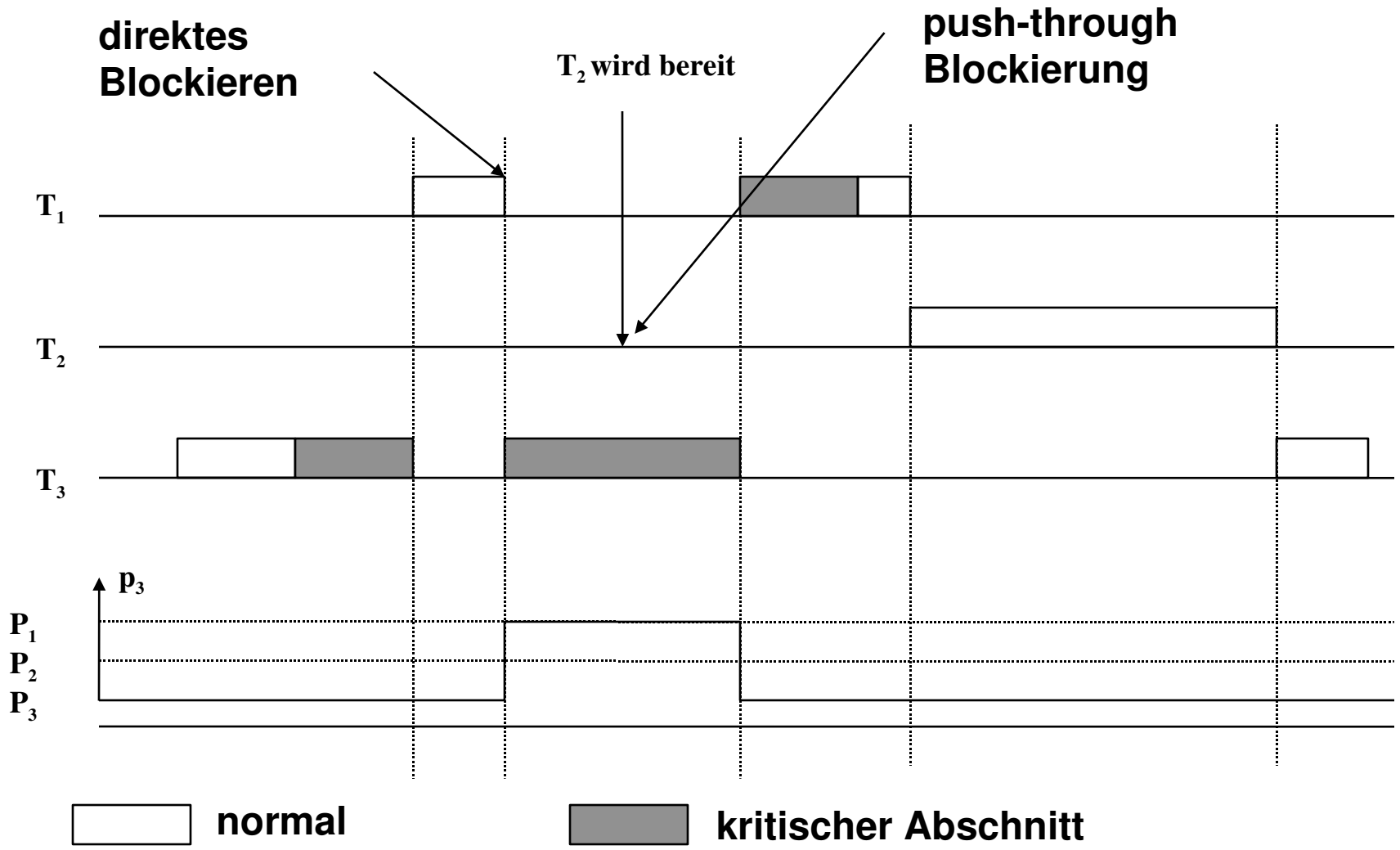
2.) Regeln zur Prioritätsvererbung:

- **T_i läuft unter der nominalen Priorität P_i, es sei denn, T_i befindet sich in einem kritischen Abschnitt und blockiert einen Prozeß mit höherer Priorität.**
- **Wenn T_i Tasks mit höherer Priorität blockiert, erbt T_i die höchste Priorität aller Tasks die durch T_i blockiert werden.**
- **Prioritätsvererbung ist transistiv. Seien T_1 , T_2 ,T_3 Tasks mit $p_1 < p_2 < p_3$. Wenn T_3 die Task T_2 blockiert und T_2 die Task T_1 , dann erbt T_3 die Priorität von T_1 über T_2.**
- **Wenn T_i den äußersten seiner möglicherweise geschachtelten kritischen Abschnitte verlässt, erhält T_i ihre nominale Priorität zurück.**
- **Die Operationen der Prioritäts-Vererbung und Rückführung müssen atomar (unteilbar) sein.**

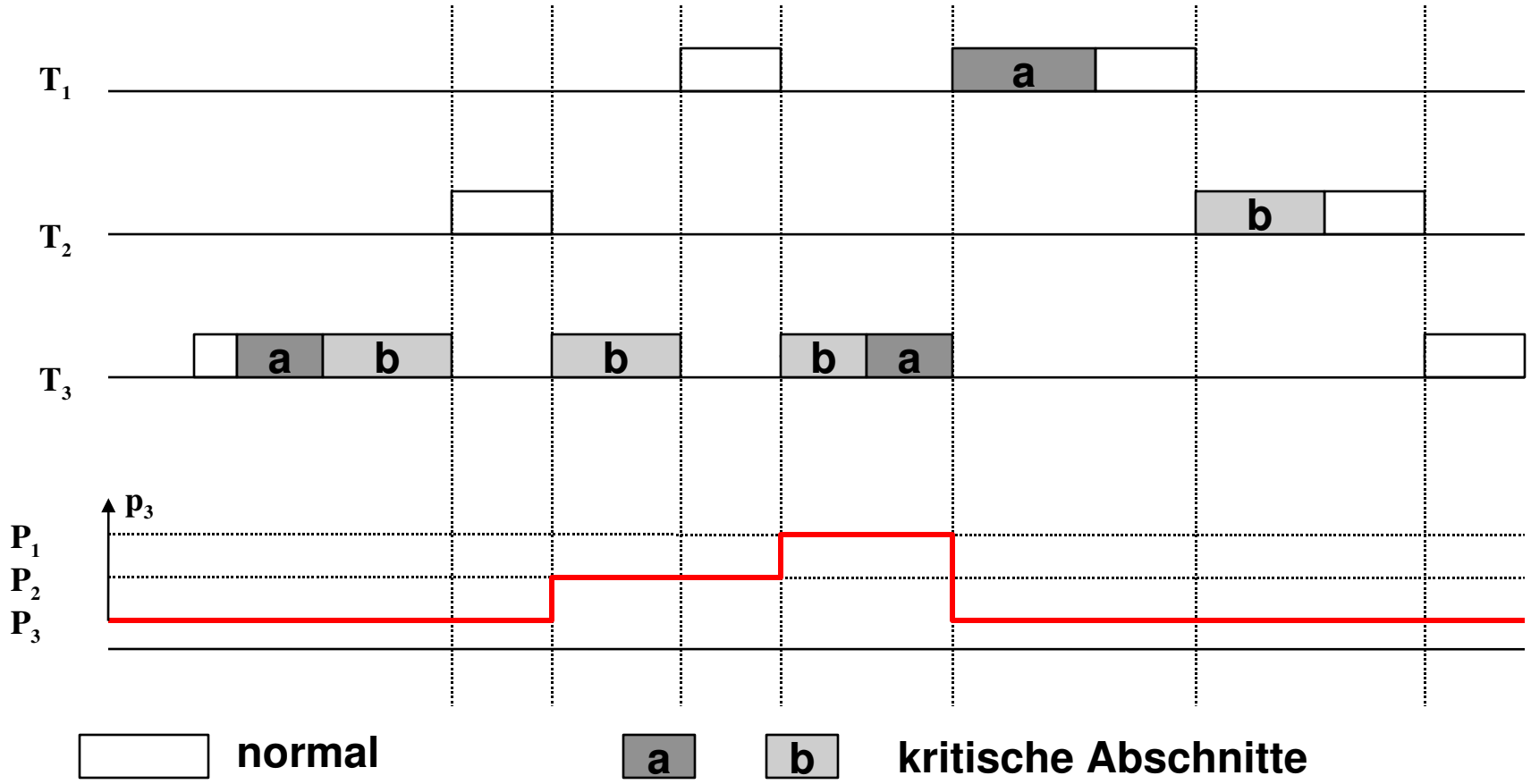
3.) Regeln für die Unterbrechung von Prozessen:

- **Eine Task T_i kann T_j jederzeit unterbrechen, wenn T_i nicht blockiert ist und gilt: $p_i < p_j$, wobei p_i eine nominale oder eine aktive Priorität sein kann.**

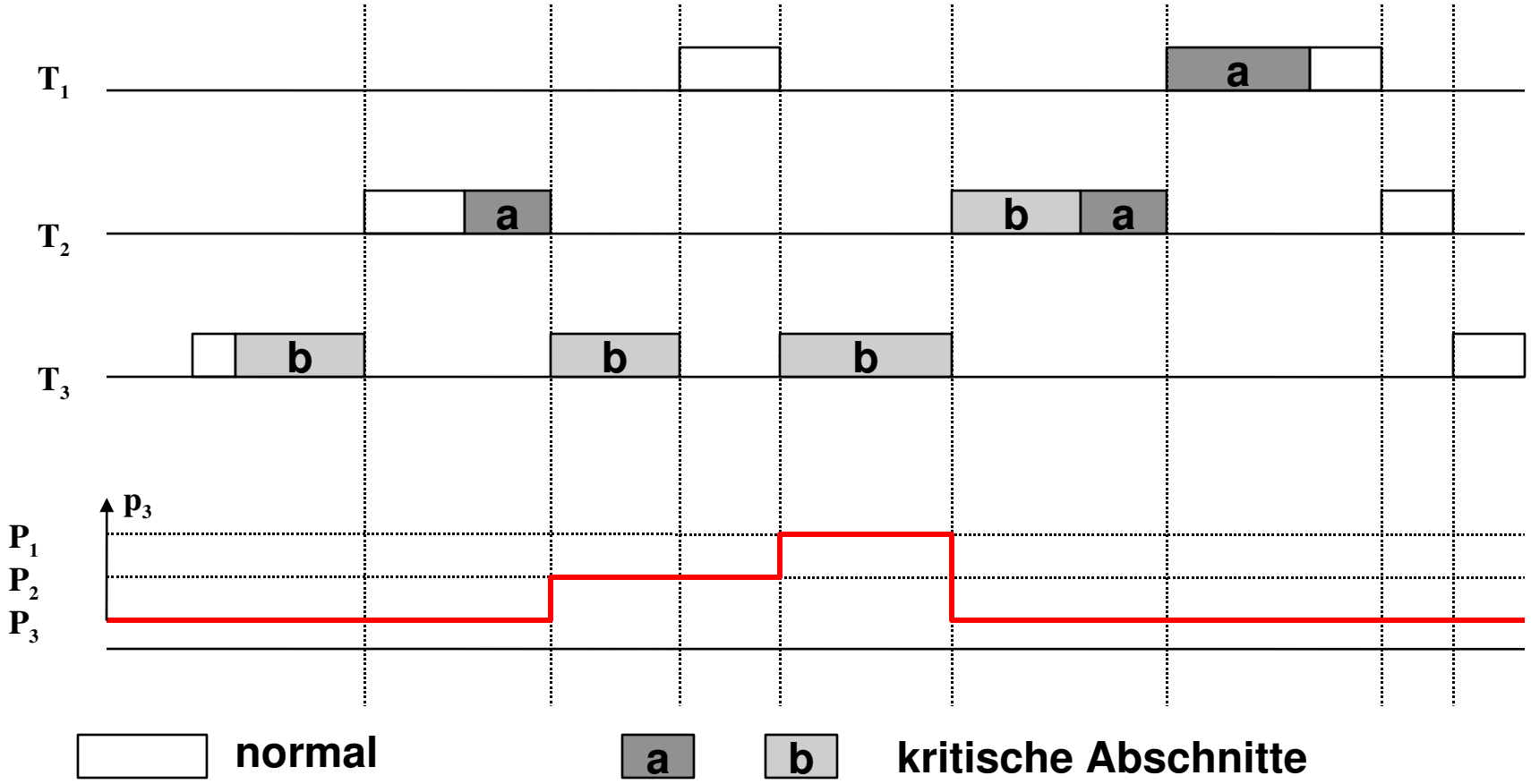
Beispiel zur Prioritätsvererbung



Beispiel zur Prioritätsvererbung mit geschachtelten kritischen Abschnitten



Beispiel zur transitiven Prioritätsvererbung



Eigenschaften des Priority Inheritance Protocol (PIP)

1.) Obere Schranken für die Dauer der Blockierung, ausgedrückt durch die maximale Anzahl der Prozesse mit niedrigerer Prioritätsstufe.

Satz: Ein Prozeß P kann unter der Annahme, daß n Prozesse mit niedrigerer Prioritätsstufe exisitieren, für höchstens die Dauer von n kritischen Abschnitten* blockiert werden, wenn das Priority Inheritance Protocol angewendet wird. Dabei ist n unabhängig von der Anzahl der benutzten Semaphore.

2.) Obere Schranken für die Dauer der Blockierung ausgedrückt durch die maximale Anzahl der Semaphore, die gemeinsam benutzt werden.

Satz: Ein Prozeß P kann unter der Annahme, daß m Semaphore existieren, durch die er blockiert werden kann, für höchstens die Dauer von m kritischen Abschnitten* blockiert werden, wenn das Priority Inheritance Protocol angewendet wird.

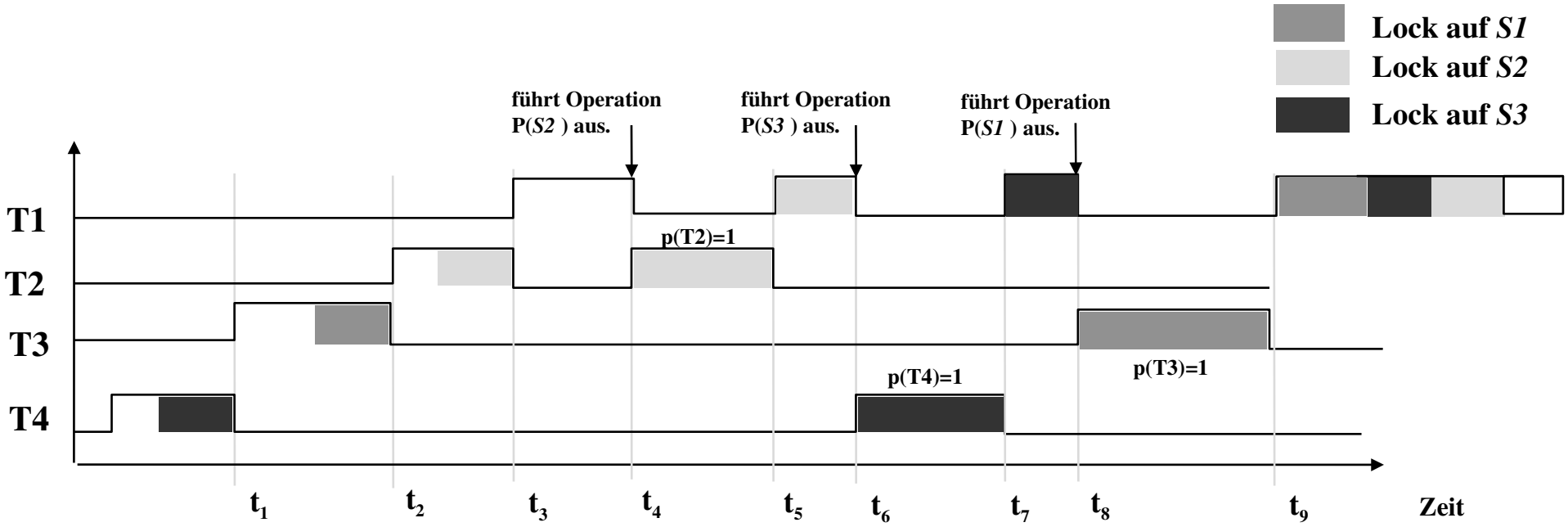
*** die Abschnitte können geschachtelt sein !**

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.

Das Priority Inheritance Protocol - Immer noch nicht zufriedenstellend !

Problem 1 -Blockierungsketten

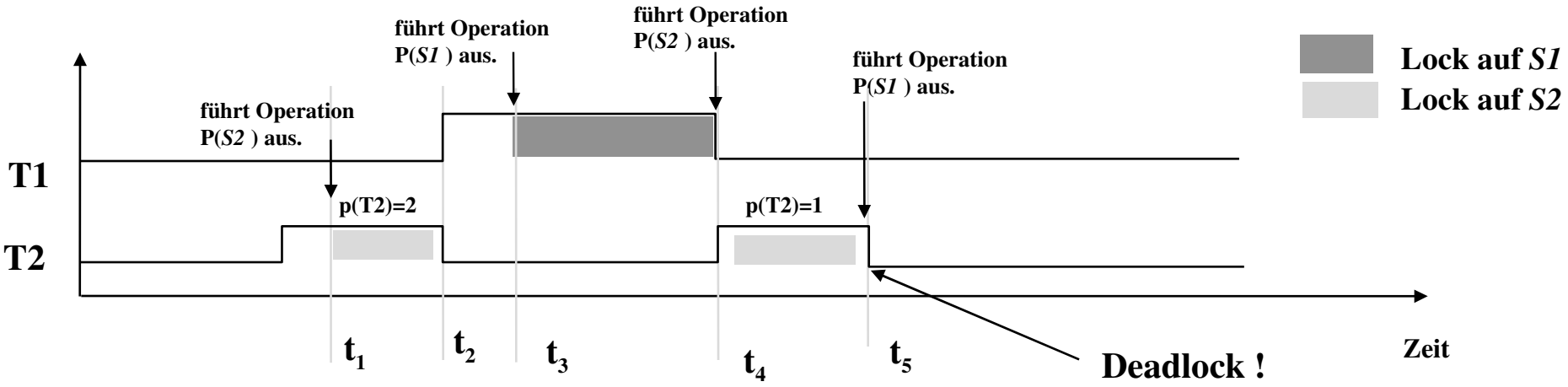


Die Task T1 wird nacheinander durch die Tasks T4, T3 und T2 blockiert.
Man spricht von einer Blockierungskette (Chain of Blocking). Die Vorhersagbarkeit der Blockierungsdauer ist nicht gewährleistet.

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.):

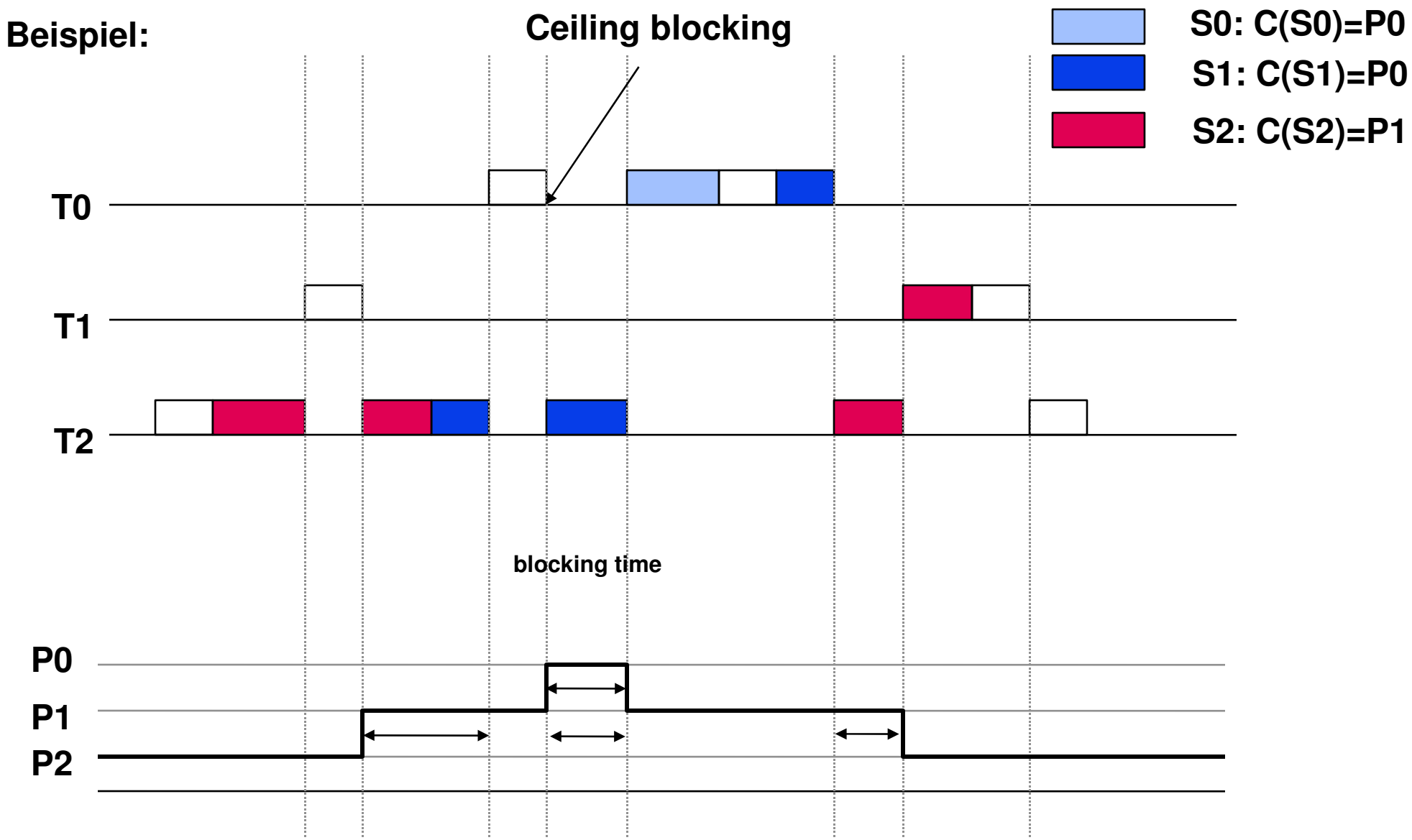
- **Falls noch kein Semaphor gesperrt ist, kann jede Task ein Semaphor sperren und ihren kritischen Abschnitt betreten.**

- **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

Beispiel zum Priority Ceiling Protocol

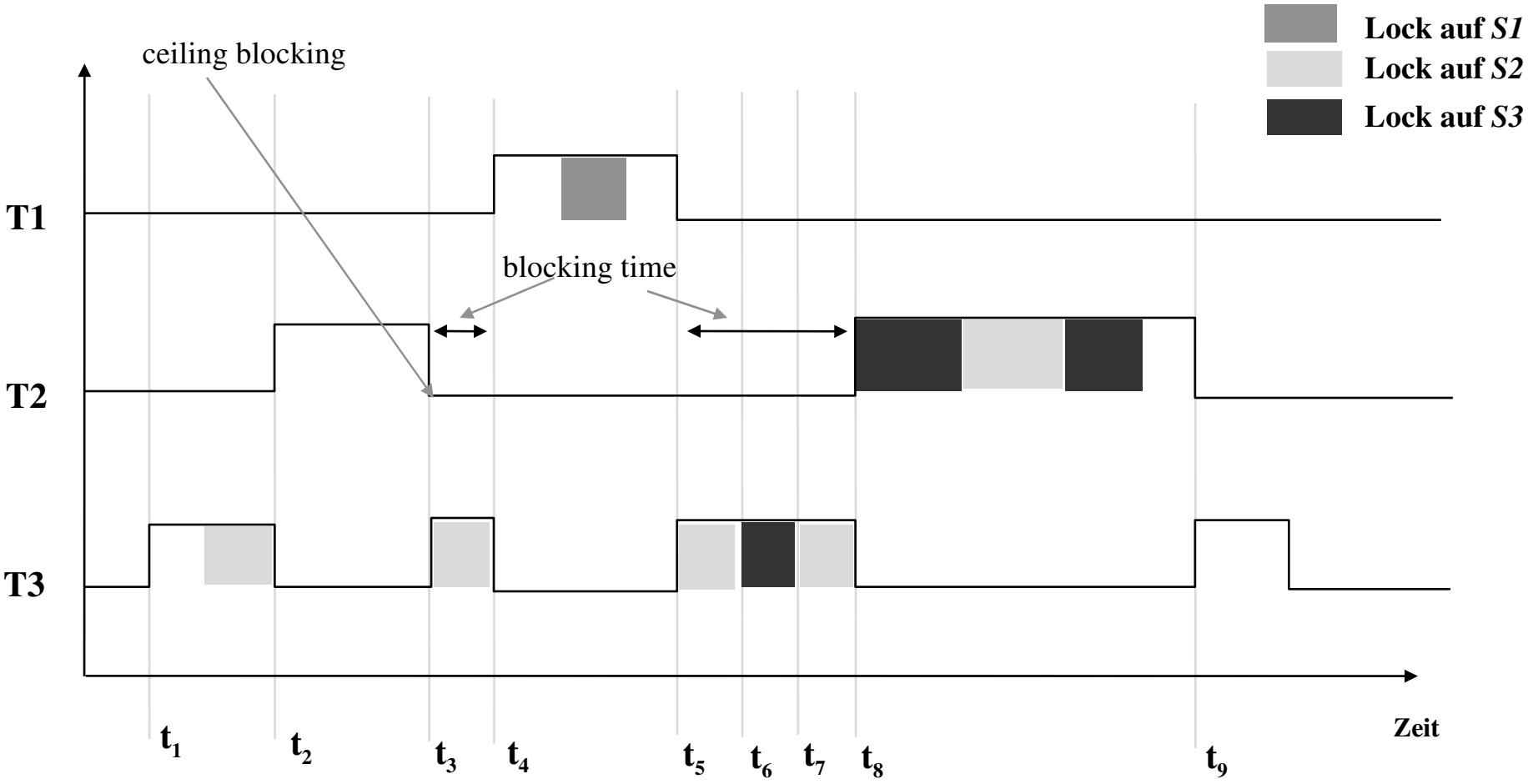
Gegeben sind drei Tasks T1, T2, T3 und zwei gemeinsam benutzte Datenstrukturen, die durch die binären Semaphore *S2* ind *S3* geschützt sind.Die Folge der Schritte bei der Ausführung der Prozesse sei:

T1 = { . . . . , P(*S1*), ... , V(*S1*), . . . , P(*S3*), ... , V(*S3*), }
T2 = { . . . . , P(*S3*), ... , P(*S2*), . . . , V(*S2*), ... , V(*S3*), ...}
T3 = { . . . . , P(*S2*), ... , P(*S3*), . . . , V(*S3*), ... , V(*S2*), ...}

Die Prioritäten der Tasks seien:
p(T1) = 1, p(T2) = 2, p(T3) = 3

Die Prioritätshöchstgrenzen für die Semaphore ergeben sich dann wie folgt:
***S1* hat die Prioritätshöchstgrenze 1,**
***S2* hat die Prioritätshöchstgrenze 2,**
***S3* hat die Prioritätshöchstgrenze 1.**

Beispiel zum Priority Ceiling Protocol (2):



Beispiel zum Priority Ceiling Protocol (3)

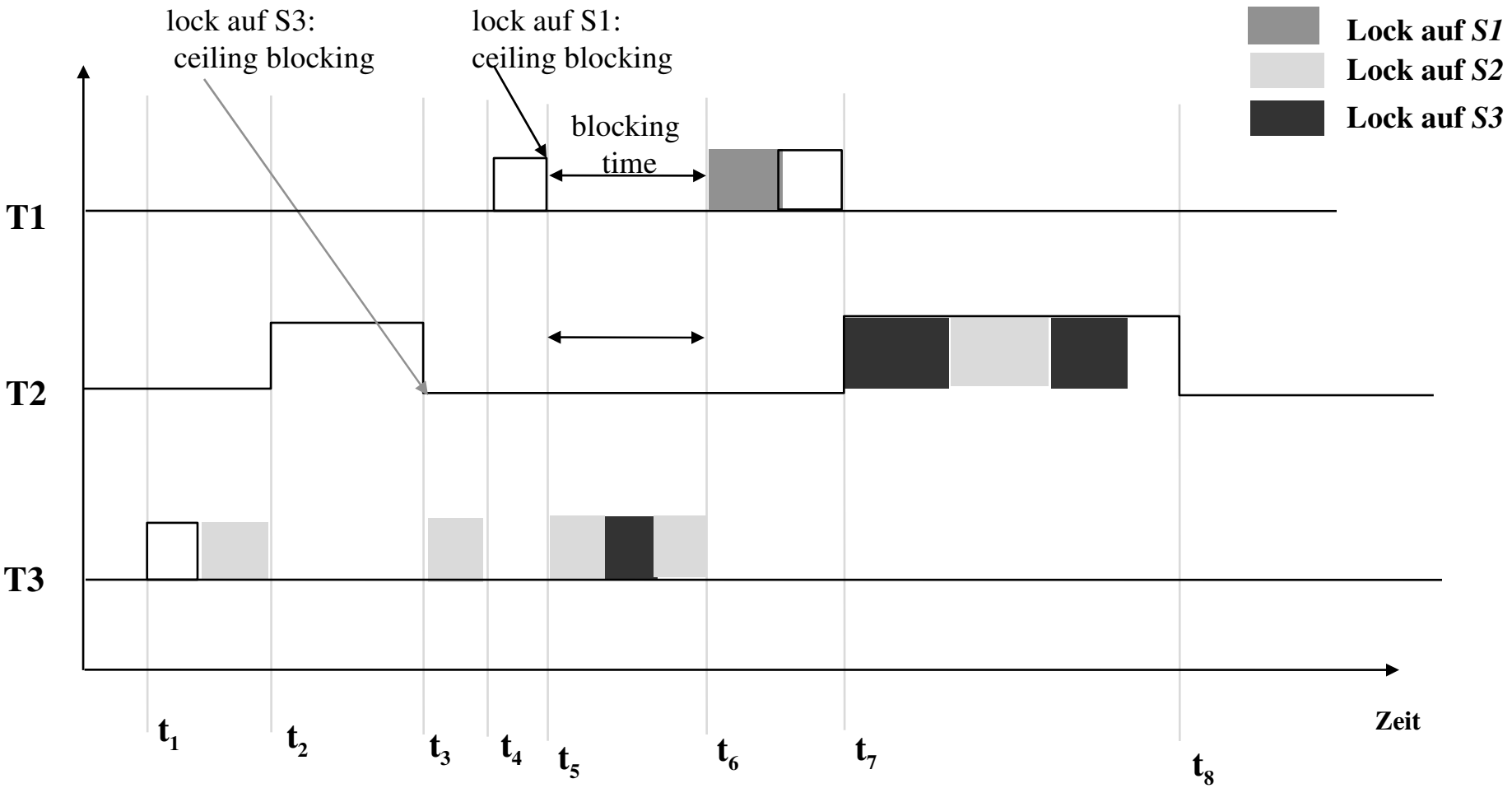
Gegeben sind drei Tasks T1, T2, T3 und zwei gemeinsam benutzte Datenstrukturen, die durch die binären Semaphore *S2* ind *S3* geschützt sind.Die Folge der Schritte bei der Ausführung der Prozesse sei:

T1 = { . . . . , P(*S1*), ... , V(*S1*), . . . , P(*S2*), ... , V(*S2*), }
T2 = { . . . . , P(*S3*), ... , P(*S2*), . . . , V(*S2*), ... , V(*S3*), ...}
T3 = { . . . . , P(*S2*), ... , P(*S3*), . . . , V(*S3*), ... , V(*S2*), ...}

Die Prioritäten der Tasks seien:
p(T1) = 1, p(T2) = 2, p(T3) = 3

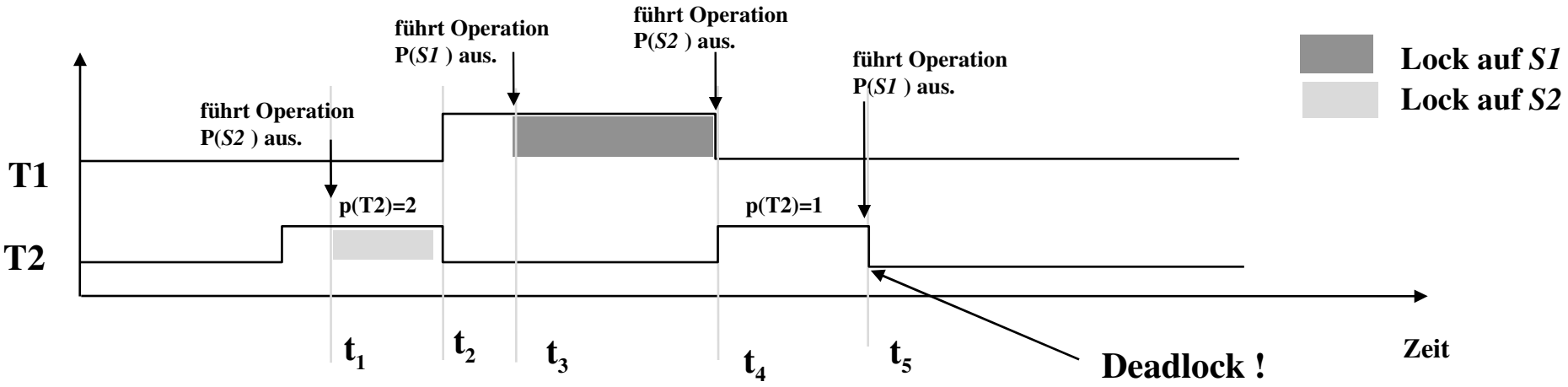
Die Prioritätshöchstgrenzen für die Semaphore ergeben sich dann wie folgt:
***S1* hat die Prioritätshöchstgrenze 1,**
***S2* hat die Prioritätshöchstgrenze 1,**
***S3* hat die Prioritätshöchstgrenze 2.**

Beispiel 1 zum Priority Ceiling Protocol (3):

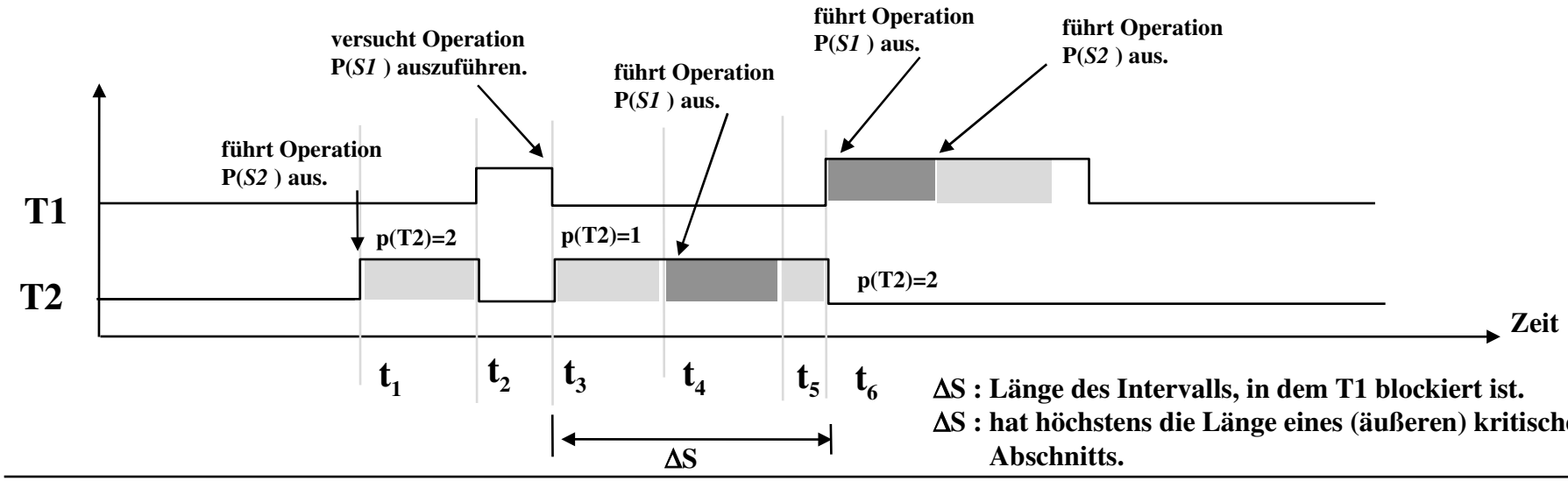


Problem 1 - Deadlock Situationen:

Priority Inheritance Protocol:

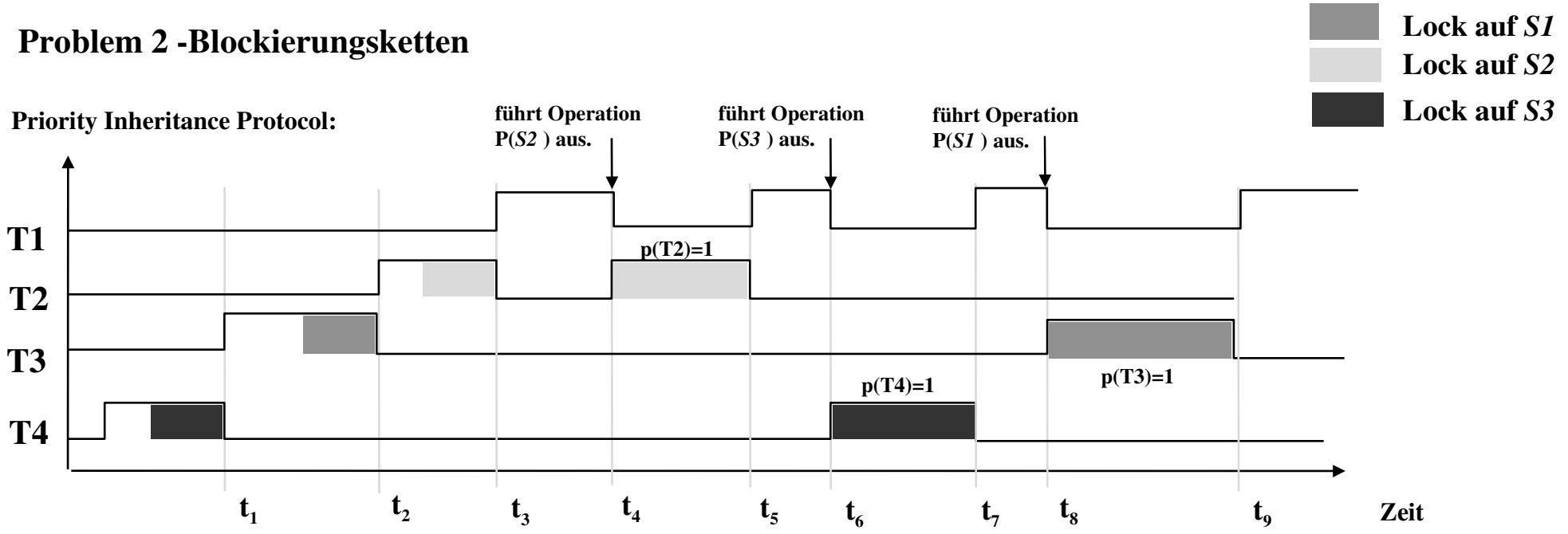


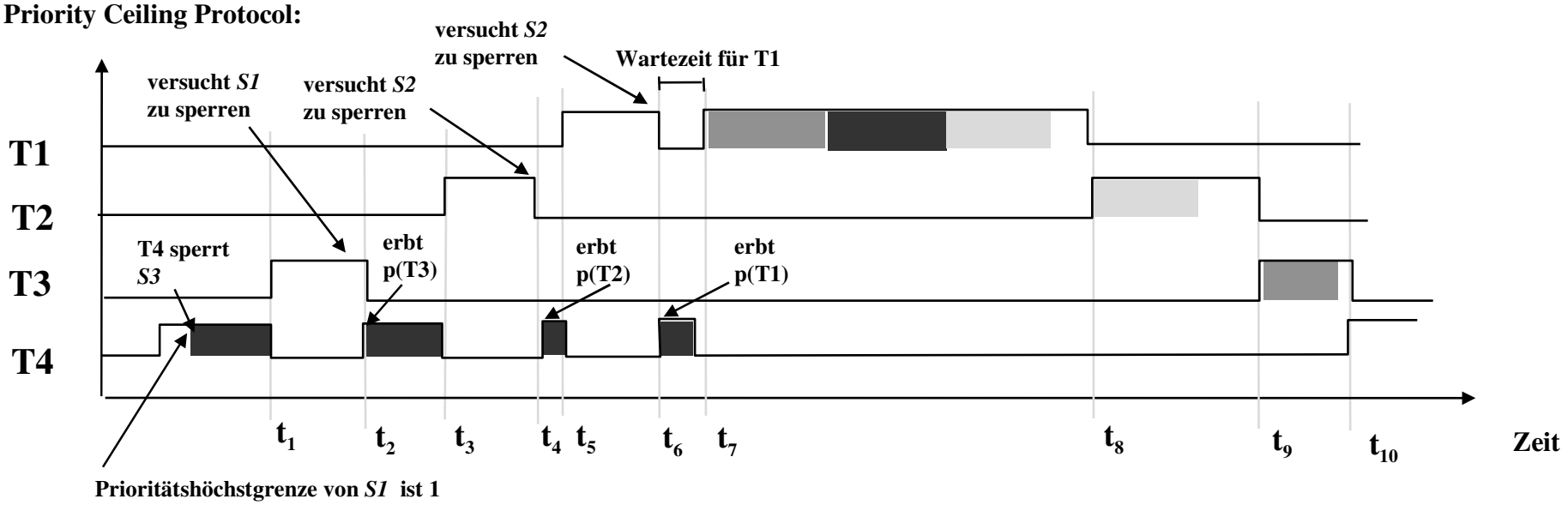
Priority Ceiling Protocol:



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.**