Kapitel 2

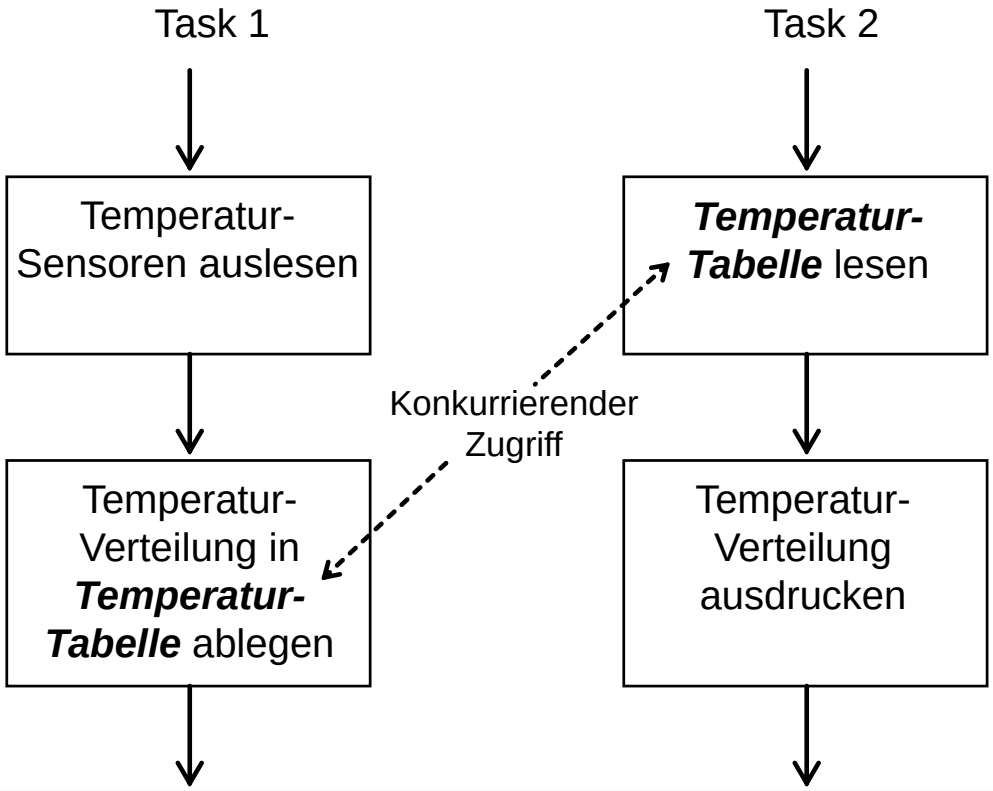
Echtzeitbetriebssysteme

Synchronisation und Verklemmungen von gemeinsam genutzten Betriebsmitteln

Gemeinsame Betriebsmittel

- Daten
- Geräte
- Programme

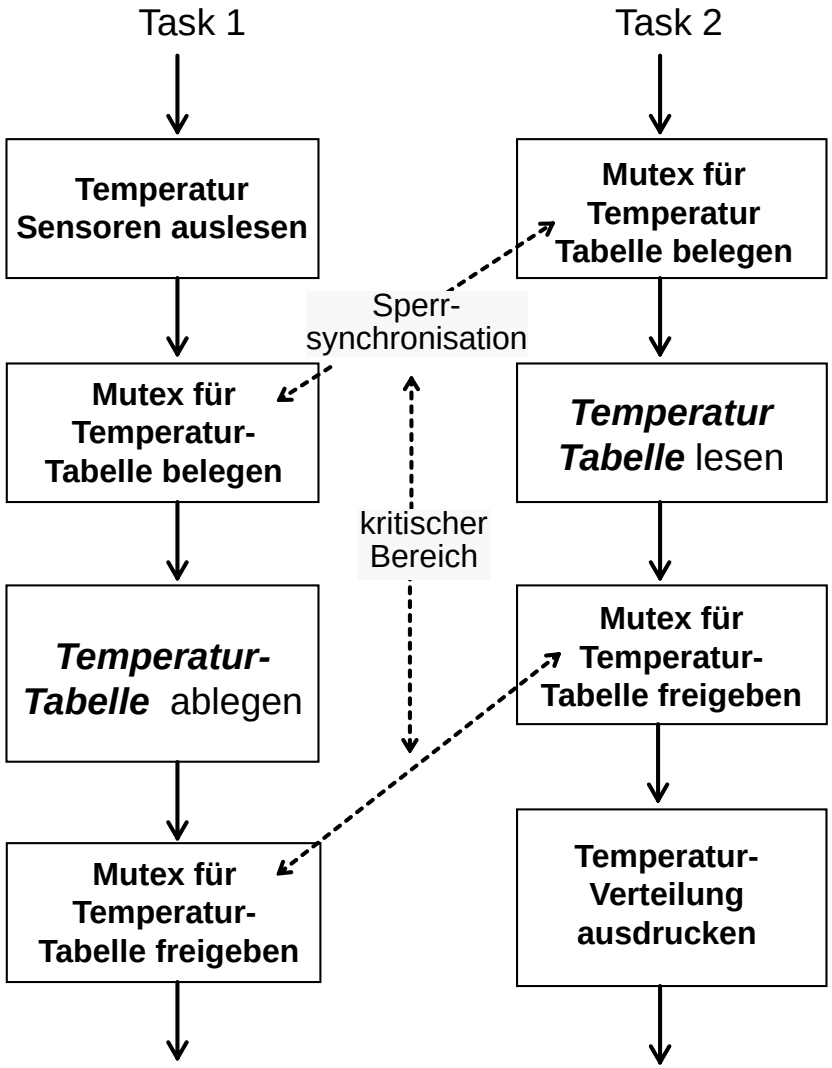
***Beispiel*: gemeinsam genutzte „Temperatur-Tabelle“**



Sperrsynchronisation und Reihenfolgensynchronisation

Synchronisation – Variante 1

- Die Sperrsynchronisation verhindert gleichzeitige Zugriff
- Mutex = wechselseitiger Ausschluss
- Task belegt Mutex und gibt ihn wieder frei
- Bei belegtem Mutex wird die zugreifende Task verzögert (blockiert)
- Diejenige Task, welche Sperre setzt, gibt sie wieder frei

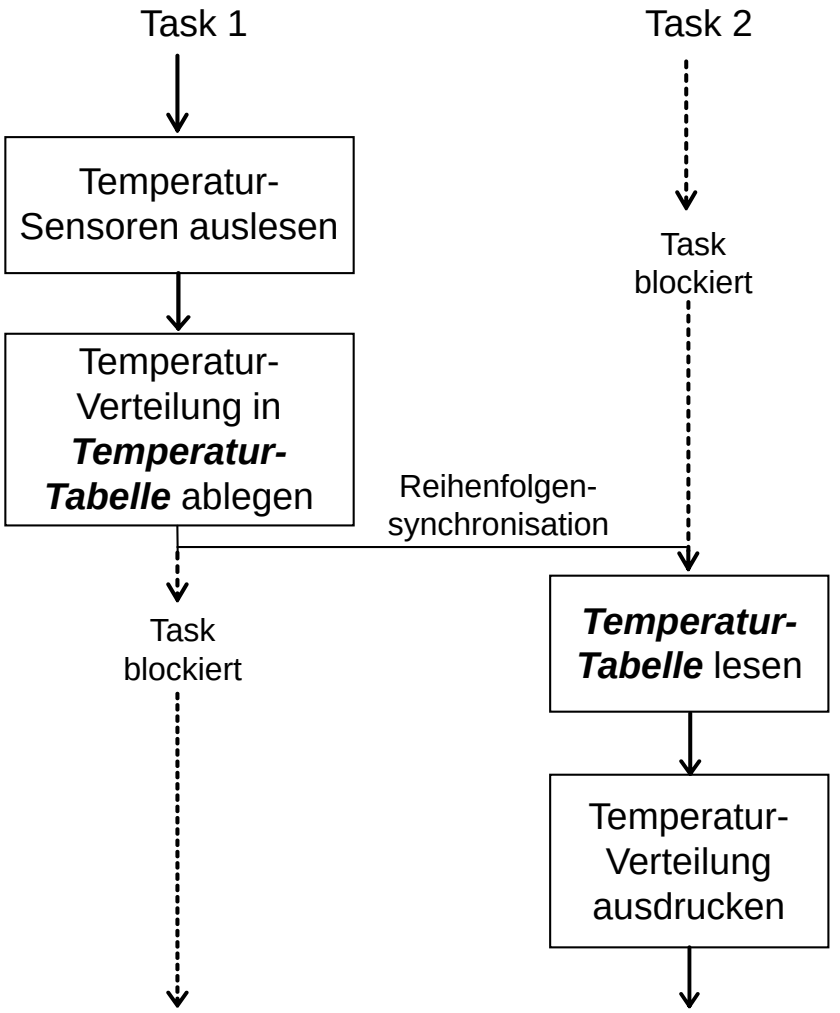


Intelligente Prozess-automation und Robotik

Sperrsynchronisation und Reihenfolgensynchronisation

Synchronisation – Variante 2

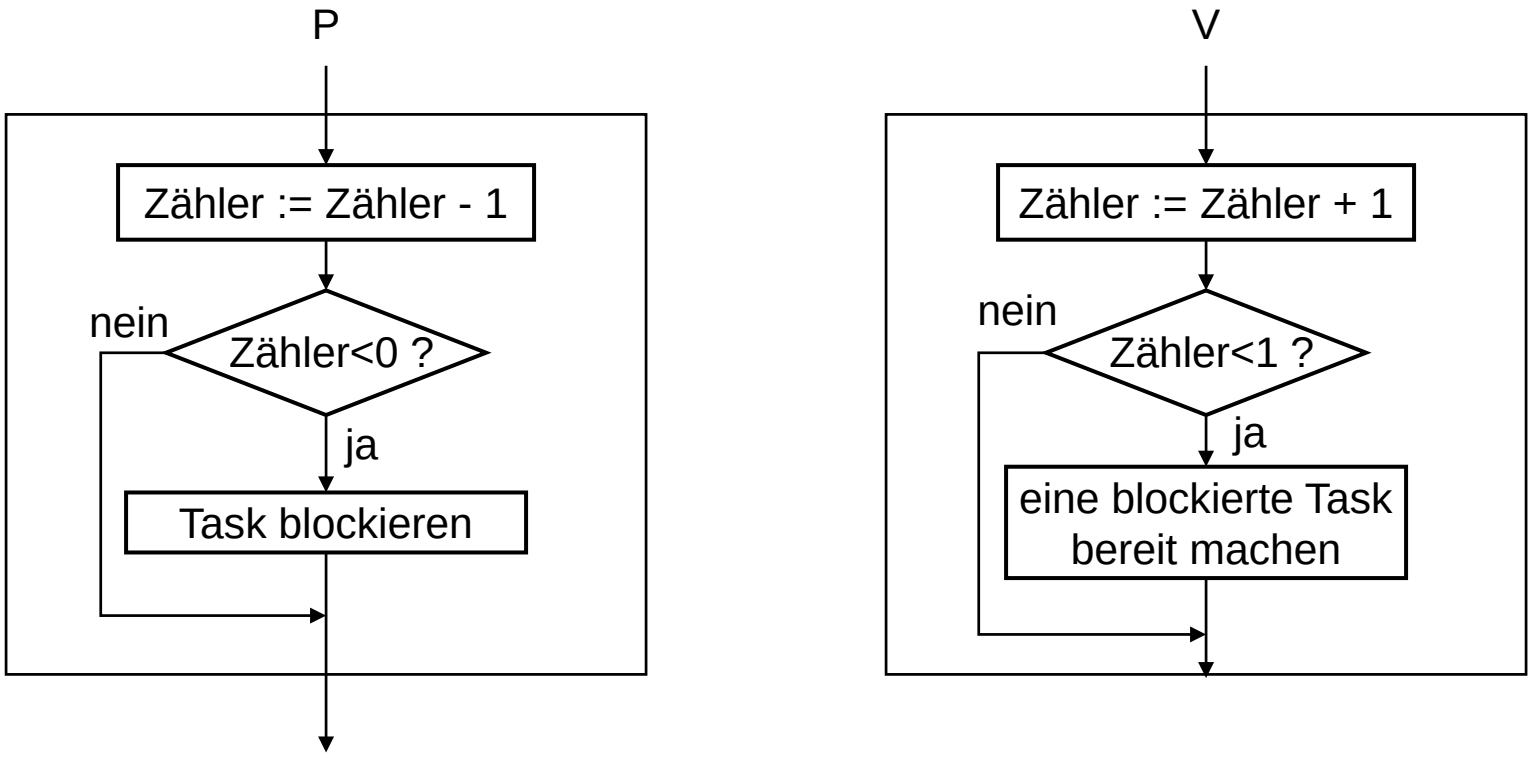
- Die Reihenfolgensynchronisation regelt die Reihenfolge der Taskzugriffe auf gemeinsame Freischaltung durch andere Task



Semaphore mit den Operationen „Passieren“ und „Verlassen“ für die Synchronisation

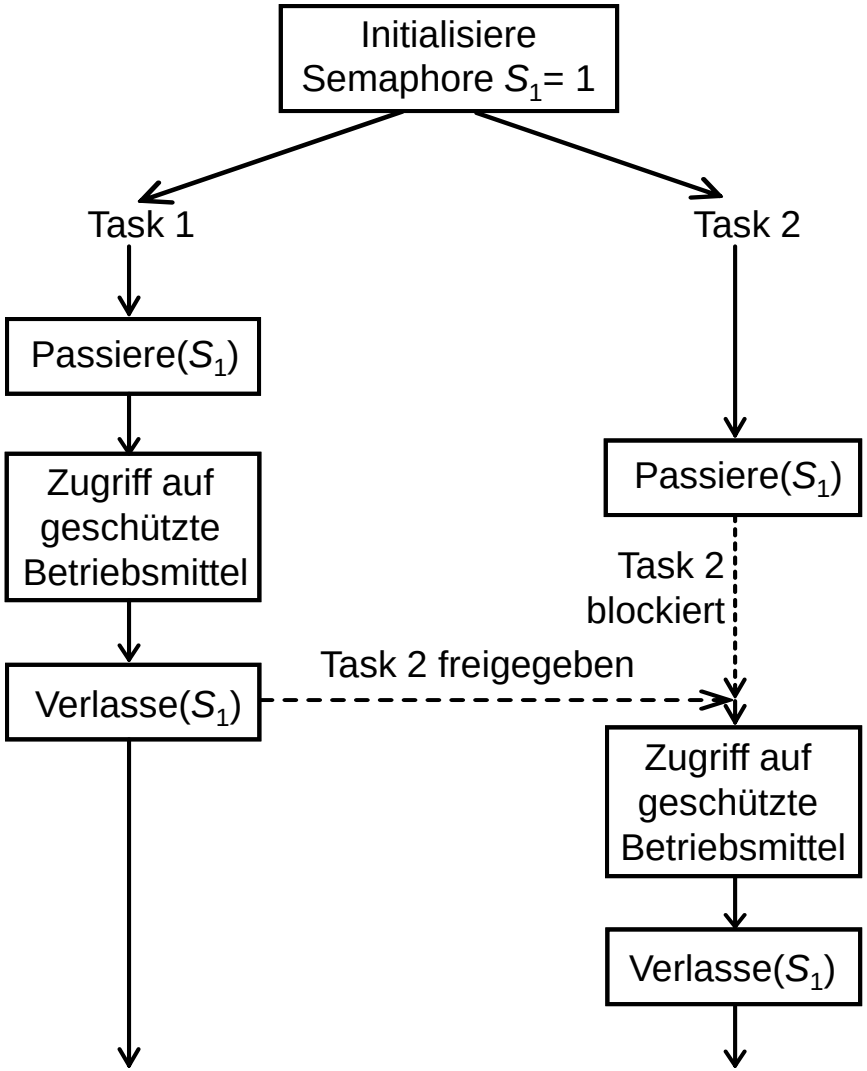
Ein ***Semaphore*** besteht aus:

- einer Zählvariablen
- zwei nicht unterbrechbaren Operationen P (Passieren) und V (Verlassen)

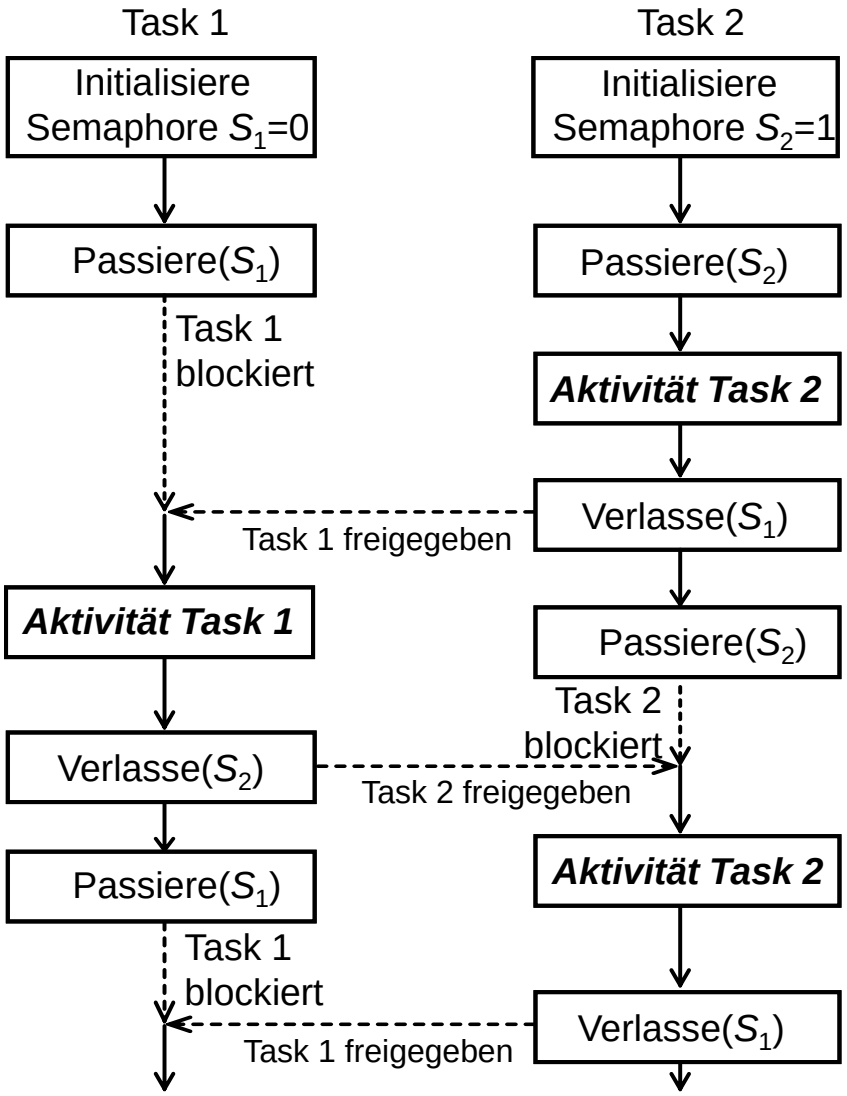


Bei Standardbetriebssystemen: Bereitmachen einer beliebigen wartenden Task bei V
Bei Echtzeitbetriebssystemen: Bereitmachen der höchstprioren wartenden Task bei V

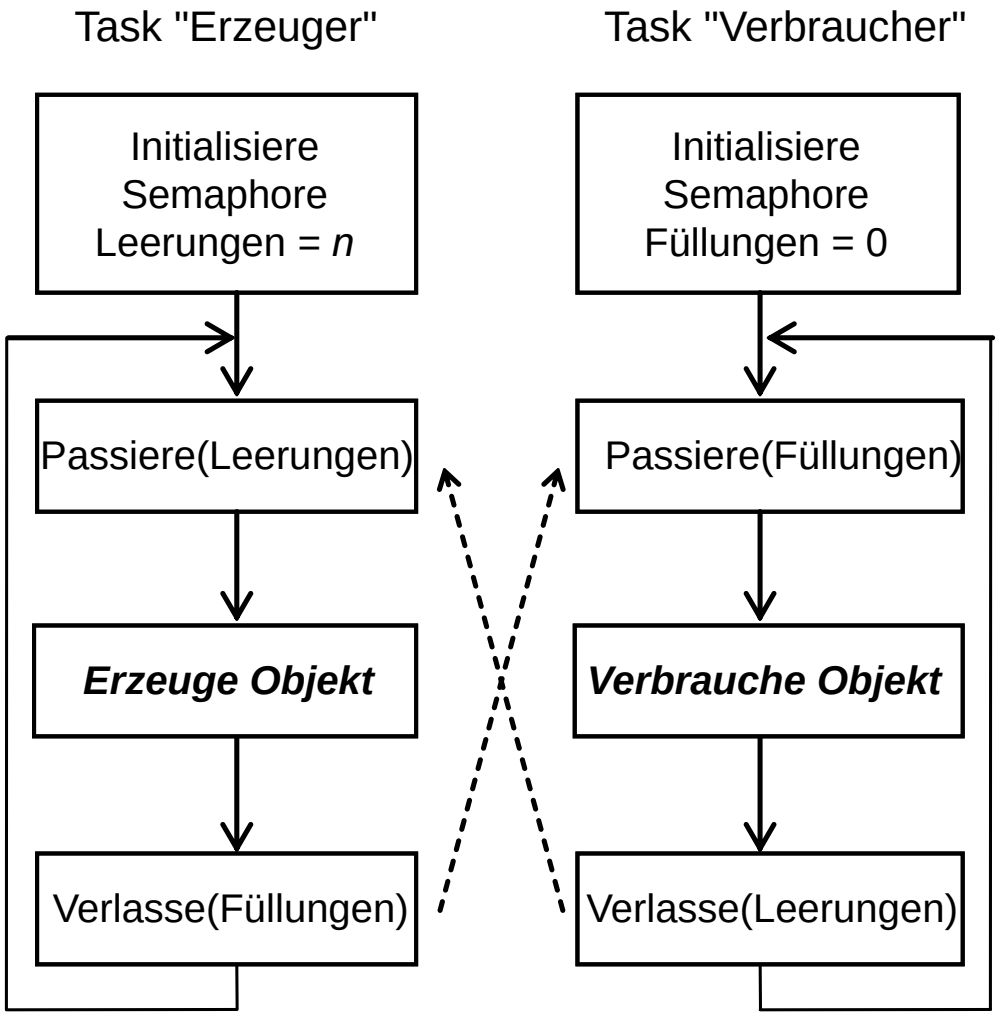
Sperrsynchronisation mit einem Semaphore



Reihenfolgensynchronisation mit zwei Semaphoren



Erzeuger-/Verbraucher-Synchronisation mit zwei Semaphoren

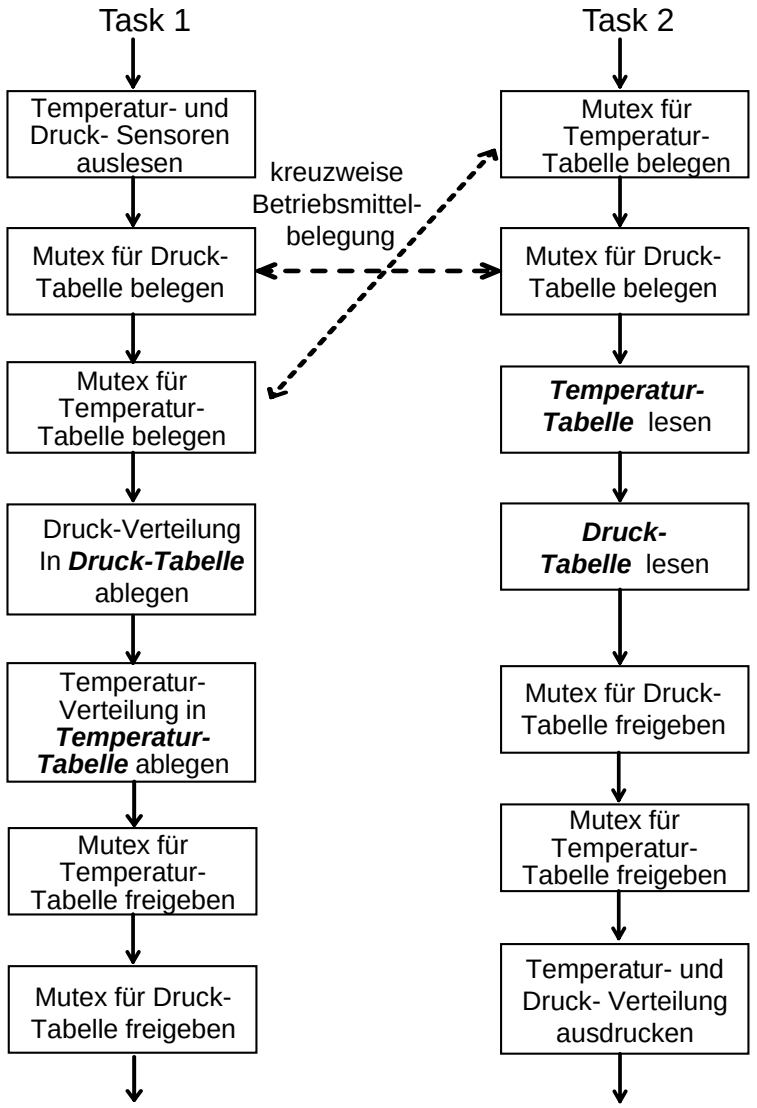


Leerungen: Freie Plätze für Objekte des Erzeugers, Füllungen: erzeugte Objekte für Verbr.

Verklemmungen: Deadlocks

Deadlock (Blockierung, *Deadly Embrace*): mehrere Tasks warten auf die Freigabe von Betriebsmitteln, die sich gegenseitig blockieren.

Beispiel: kreuzweise Betriebsmittelbelegung

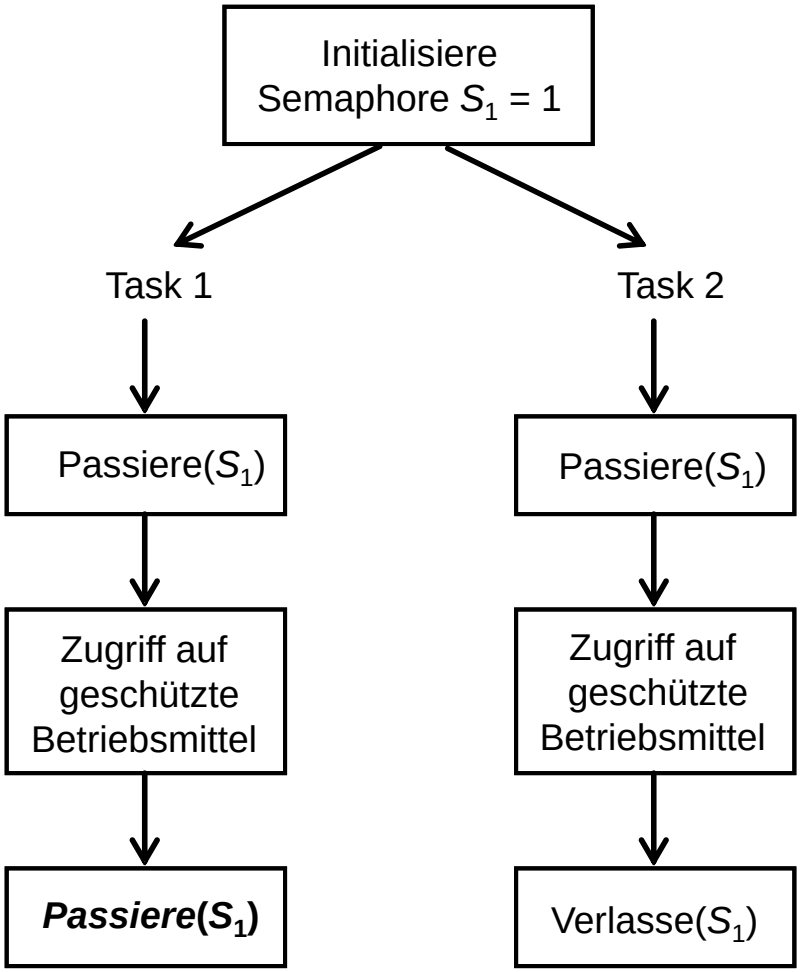


Gegenmaßnahmen:

- Abhängigkeitsanalyse
- Vergabe der Betriebsmittel in gleicher Reihenfolge
- Timeout

Verklemmungen: Deadlocks

Ein Deadlock durch Programmierfehler



Gegenmaßnahme:

- höhere Synchronisationskonstrukte (z.B. Monitore)

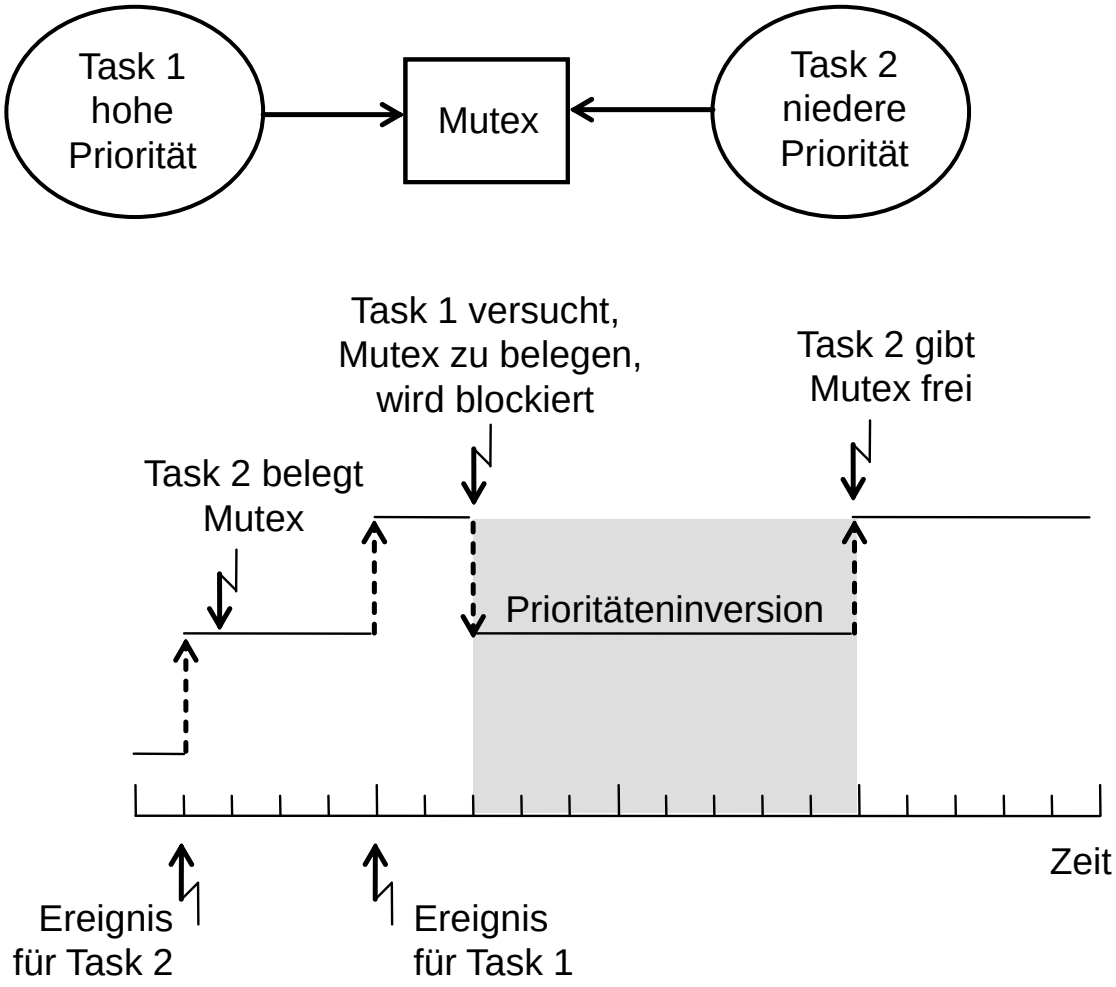
Verklemmungen: Livelocks

Livelocks (*Starvation,* Aushungern): eine Task wird durch andere Tasks ständig an der Ausführung gehindert.

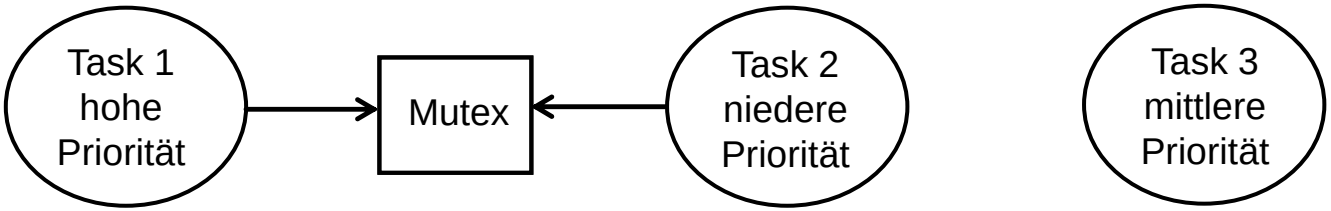
Beispiel: eine hochpriore Task verhindert ständig die Ausführung einer niederprioren Task

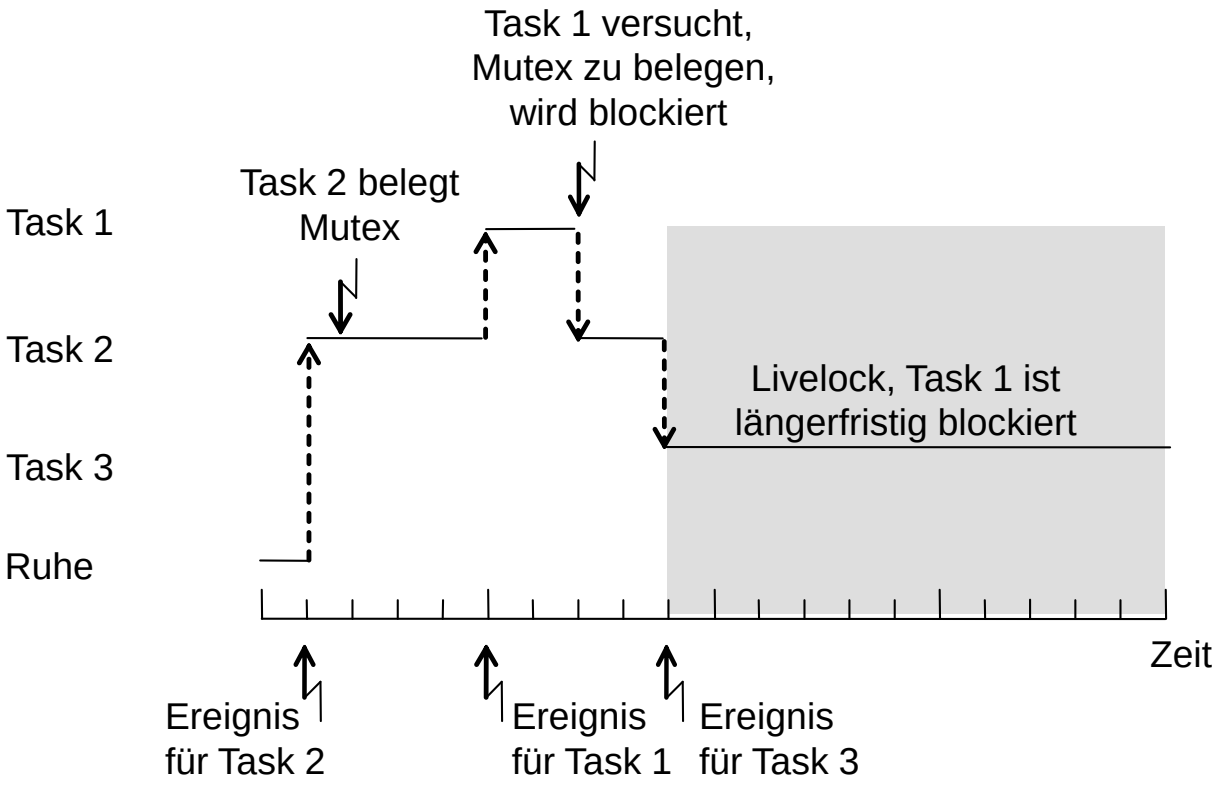
Auch hochpriore Tasks können „Opfer“ von Livelocks werden,
Problem Prioritäteninversion

Prioritäteninversion

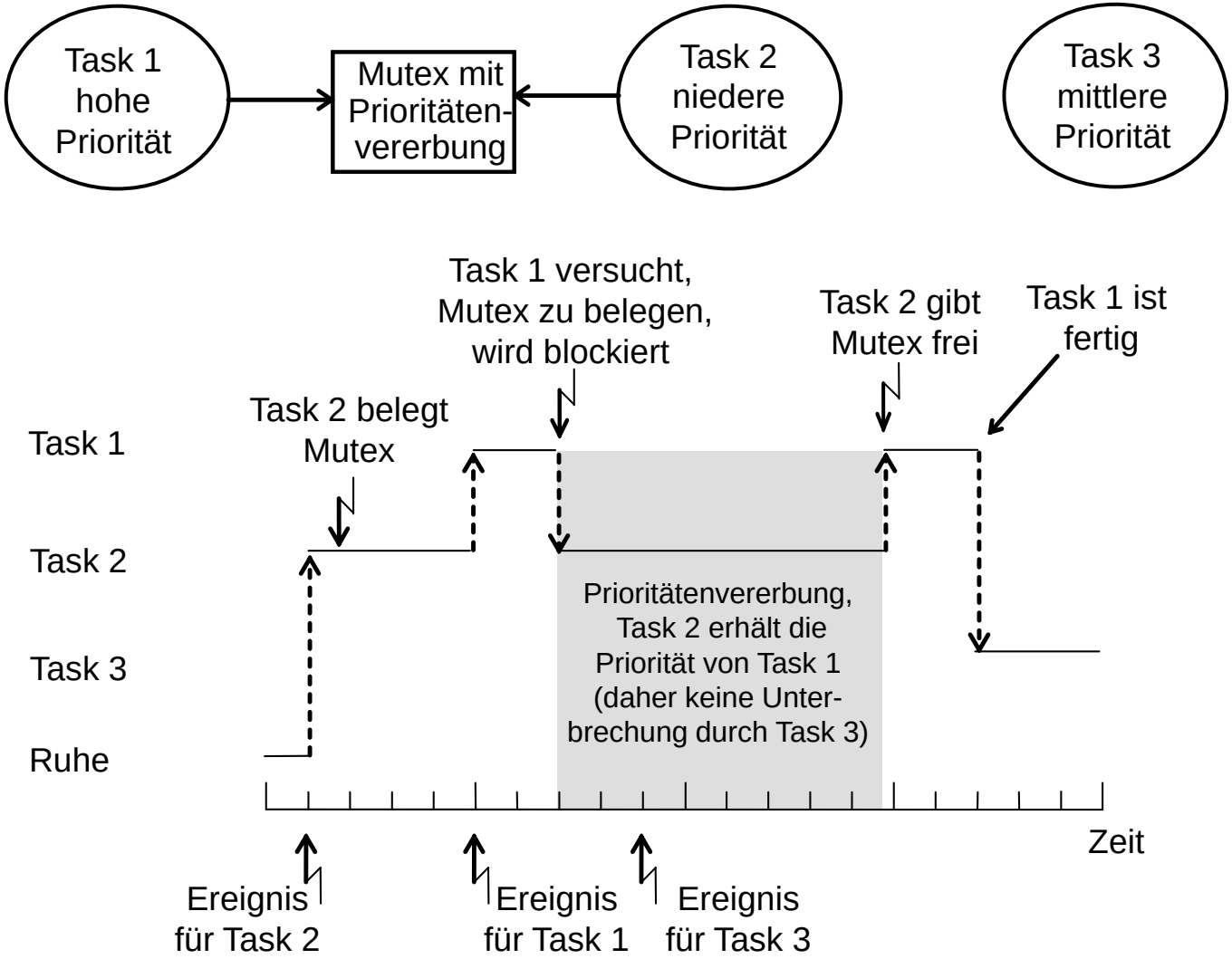


Ein Livelock bei Prioritäteninversion





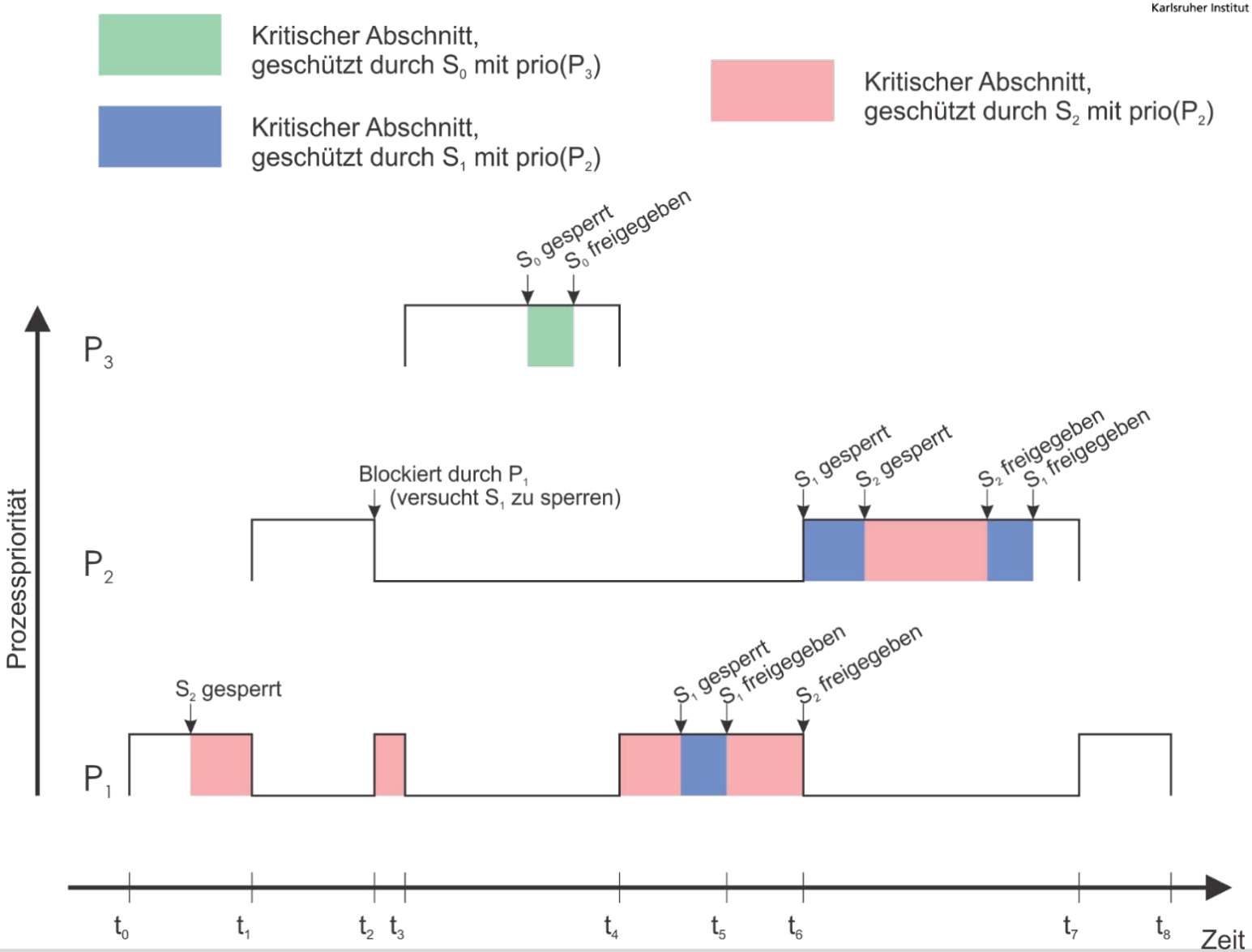
Vermeidung des Livelocks durch einen Mutex mit Prioritätenvererbung



Priority Ceiling Protocol

- The ceiling of a semaphore, **ceil(s)**, is the priority of the highest priority task that uses the semaphore
- Notation: **pri(i)** is the priority of task **i**
- At run-time:
  - If a task **i** wants to lock a semaphore **s**, it can only do so if **pri(i)** is strictly higher than the ceilings of all semaphores currently locked by other tasks
  - If not, task **i** will be blocked (task **i** is said to be blocked on the semaphore, **S∗**, with the highest priority ceiling of all semaphores currently locked by other jobs and task **i** is said to be blocked by the task that holds **S∗**)
  - When task **i** is blocked on **S∗**, the task currently holding **S∗** inherits the priority of task **I**
- Properties:
  - Deadlock free
  - A given task **i** is delayed at most once by a lower priority task
  - The delay is a function of the time taken to execute the critical section

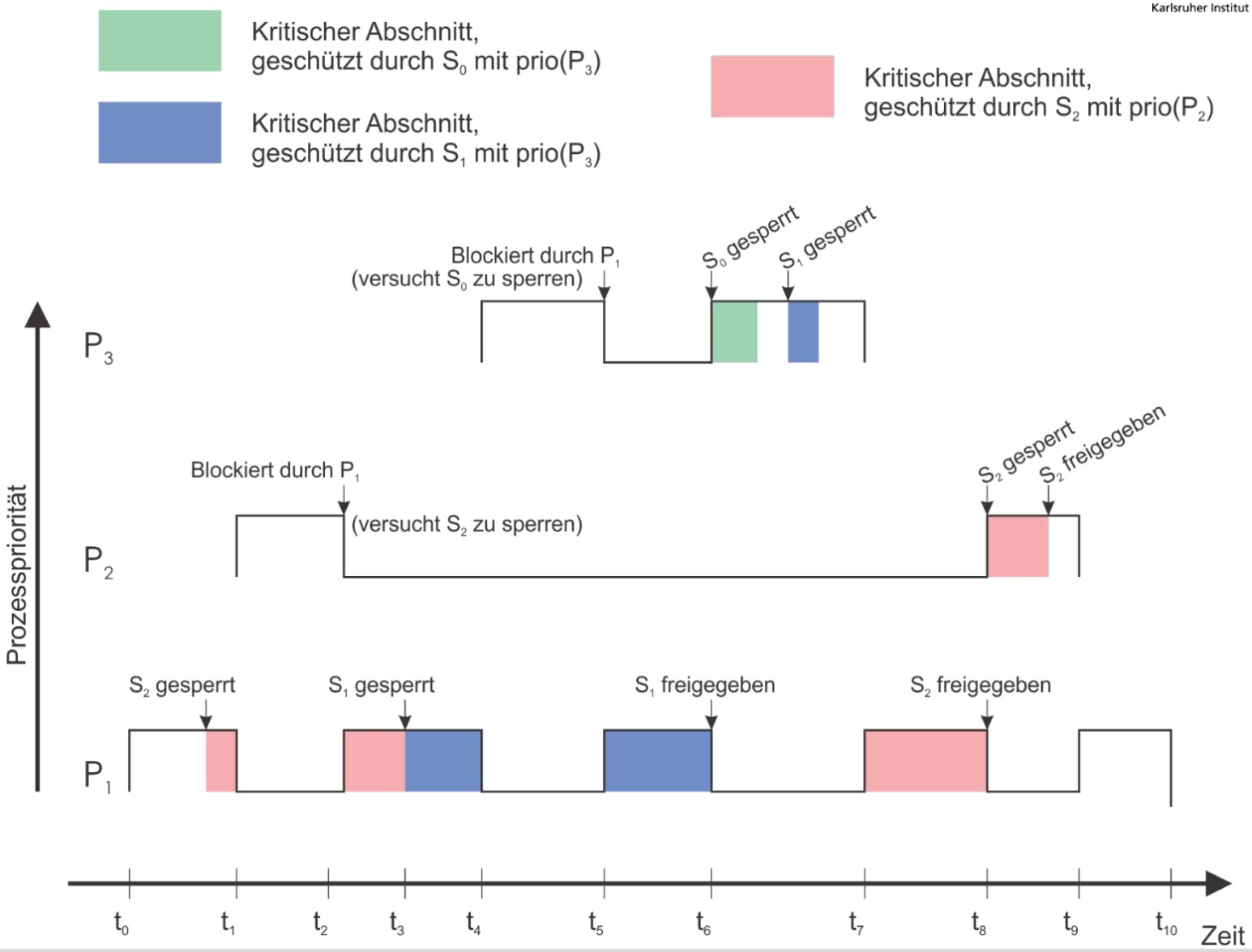
Priority Ceiling Protocol (1)



Priority Ceiling Protocol (1)

t_0	Die Ausführung von Prozess P_1 beginnt, der dann S_2 sperrt.
t_1	P_2 startet und unterbricht P_1 wegen höherer Priorität.
t_2	P_2 versucht den durch S_1 geschützten kritischen Abschnitt durch den nichtunterbrechbaren Aufruf $P(S_1)$ zu verwenden. Jedoch ist die Priorität von P_2 nicht höher als die Priority Ceiling des Semaphors S_2. Das System unterbricht P_2 ohne S_1 zu sperren. P_1 erbt die Priorität von P_2 und wird fortgesetzt. Anmerkung: P_2 wird außerhalb seiner kritischen Abschnitte blockiert. Da P_2 S_1 nicht sperrt, sondern selber blockiert, wird einer potenziellen Verklemmung (Deadlock) zwischen P_2 und P_1 vorgebeugt.
t_3	P_1 befindet sich immer noch in einem kritischen Abschnitt während ein Prozess P_3 mit höherer Priorität initiiert wird. P_1 wird blockiert. Später wird S_0 durch P_3 gesperrt. Da die Priorität von P_3 höher ist als die Priority Ceiling des gesperrten Semaphors S_2, darf P_3 S_0 sperren. P_3 betritt den durch S_0 geschützten Bereich während P_1 weiterhin blockiert bleibt.
t_4	P_3 hat den durch S_0 geschützten Bereich mit dem Aufruf $V(S_0)$ verlassen und beendet sich. Die Ausführung von P_1 wird fortgesetzt, weil P_2 immer noch durch P_1 blockiert wird und nicht ausgeführt werden kann. P_1 sperrt S_1.
t_5	P_1 gibt S_1 frei.
t_6	P_1 gibt S_2 frei und erhält seine ursprüngliche Priorität zurück. Nun hat P_2 die höchste Priorität und wird fortgesetzt, wodurch P_1 wieder unterbrochen wird. P_2 sperrt S_1, führt den verschachtelten kritischen Abschnitt aus und gibt S_1 wieder frei. Danach wird auch S_2 wieder freigegeben und der Code aus dem nichtkritischen Abschnitt wird ausgeführt.
t_7	P_2 beendet seine Ausführung und P_1 wird fortgesetzt.
t_8	P_1 beendet sich.

Priority Ceiling Protocol (2)



Priority Ceiling Protocol (2)

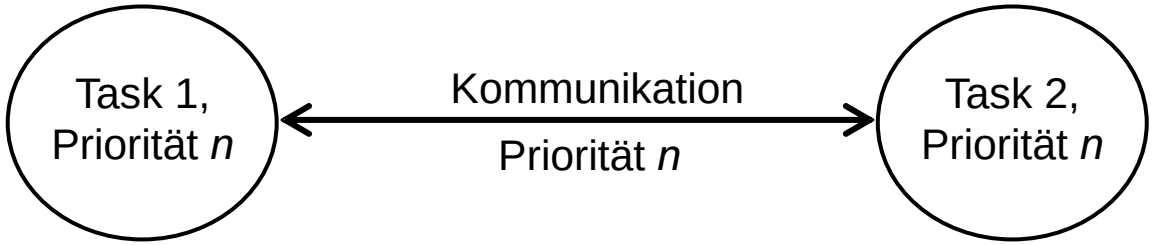
t_0	Die Ausführung von Prozess P_1 beginnt, der dann S_2 sperrt.
t_1	P_2 startet und unterbricht P_1.
t_2	P_2 versucht auf den durch S_2 geschützten Abschnitt zuzugreifen, der jedoch noch durch P_1 gesperrt ist. P_1 wird mit der Priorität von P_2 fortgeführt.
t_3	P_1 sperrt S_1. P_1 darf S_1 sperren, da es kein Semaphor S^* gibt, das von anderen Prozessen P^* blockiert wird.
t_4	P_1 befindet sich immer noch im durch S_1 geschützten Abschnitt, während P_3, ein Prozess mit höherer Priorität, gestartet wird und damit P_1 unterbricht. Dies ist möglich, da die Priorität von P_3 höher ist als die von P_1, der zur Zeit die von P_2 geerbte Priorität aufweist.
t_5	P_3 versucht auf den durch S_0 geschützten Abschnitt zuzugreifen. S_0 ist zur Zeit von keinem anderen Prozess in Verwendung, trotzdem wird P_3 von P_1 blockiert, da die Priorität von P_3 nicht höher ist als die Priority Ceiling des Semaphors S_1. P_1 bekommt die Priorität von P_3 vererbt und wird im durch S_1 geschützten Abschnitt fortgesetzt.
t_6	P_1 gibt S_1 wieder frei und bekommt wieder die vorher geerbte Priorität von P_2 zugewiesen. P_3 unterbricht P_1 aufgrund höherer Priorität. Jetzt kann P_3 S_0 in Anspruch nehmen und fortgeführt werden. Später wird S_0 wieder freigegeben, S_1 wird gesperrt und später auch wieder freigegeben.
t_7	P_3 beendet sich, und P_1 wird mit der Priorität von P_2 fortgeführt.
t_8	P_1 verlässt den durch S_2 geschützten Bereich, gibt ihn wieder frei und erhält seine ursprüngliche Priorität wieder. Nun kann P_2 auf den durch S_2 geschützten Abschnitt zugreifen. Aufgrund der höheren Priorität wird also P_1 von P_2 blockiert. Später wird S_2 von P_2 wieder freigegeben und unkritischer Code ausgeführt.
t_9	P_2 beendet sich, wodurch die Ausführung von P_1 wieder aufgenommen werden kann.
t_{10}	P_1 beendet sich.

Kommunikation zwischen Tasks

Zwei grundlegende Varianten der ***Task-Kommunikation***:

- Gemeinsamer Speicher (schneller)
- Nachrichten

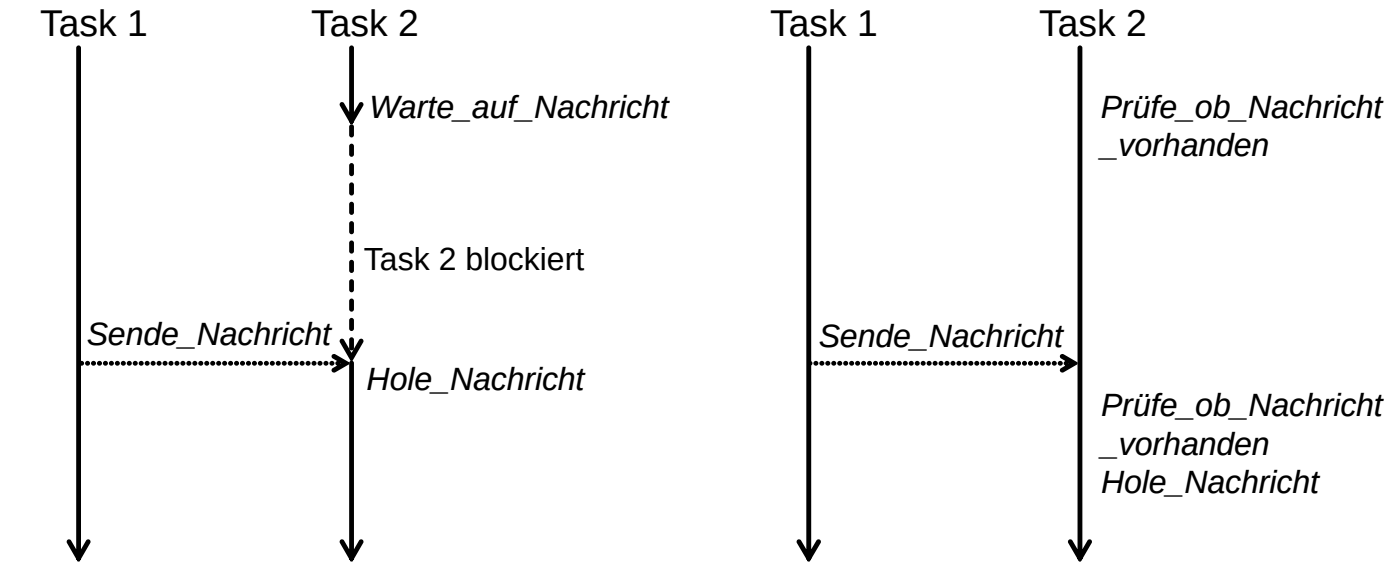
Bei Echtzeitbetriebssystemen: Wahrung der End-zu-End-Prioritäten durch prioritäts-basierte Kommunikation (hochpriore Nachrichten überholen niederpriore, keine Blockierung)



Synchrone und asynchrone Kommunikation

Weiteres Unterscheidungsmerkmal

- zeitliche Koordination
  - Synchrone Kommunikation
  - Asynchrone Kommunikation



a) synchrone Kommunikation

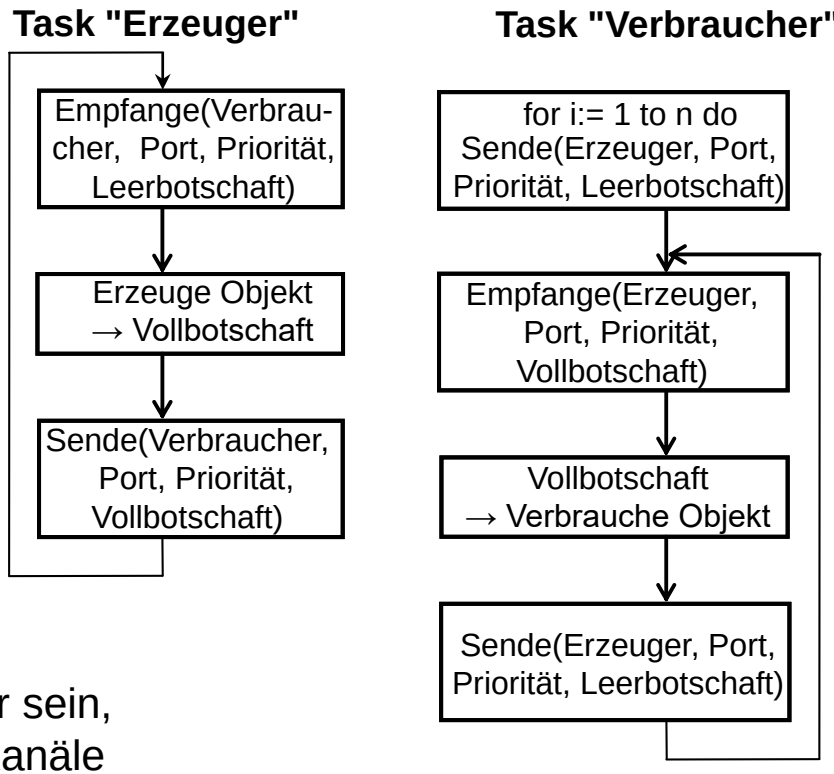
b) asynchrone Kommunikation

- in der Regel wird eine Task blockiert
- die Nachricht wird gepuffert, kein Warten der Tasks
- für ES sehr gut

Beispiel nachrichtenbasierter Kommunikation: Botschaftenaustausch

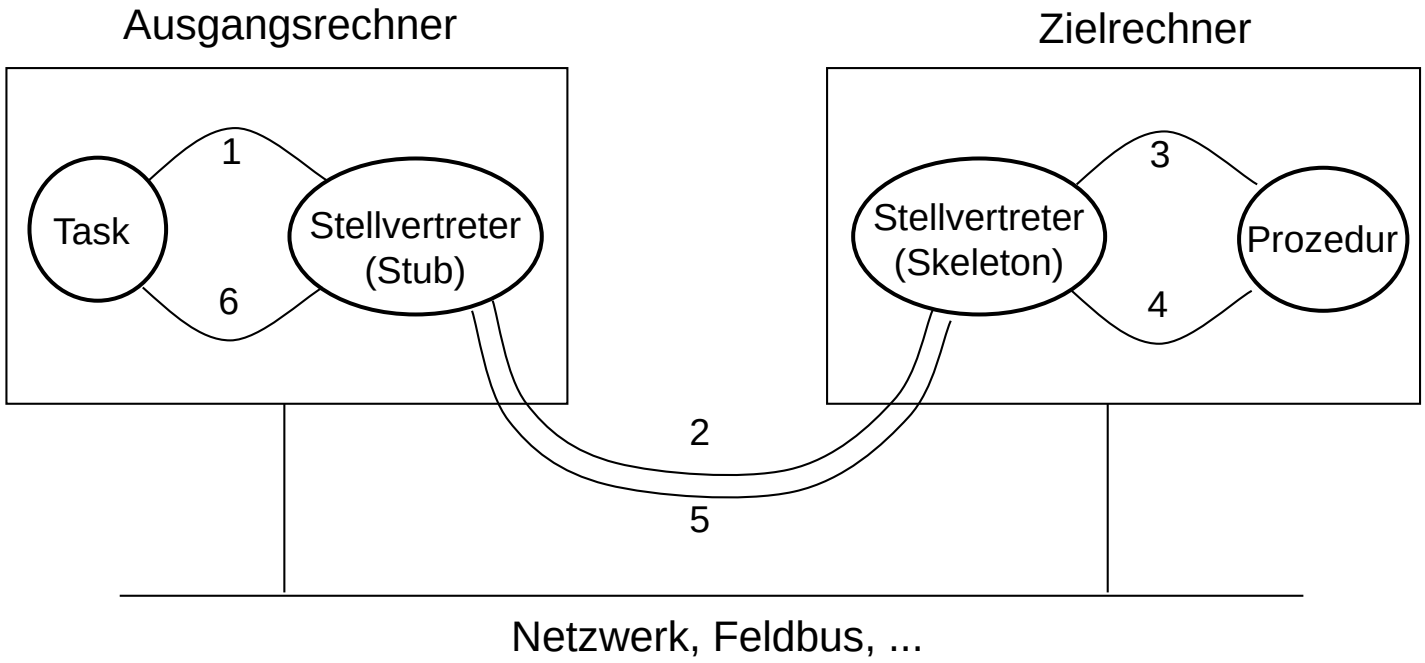
Für asynchrone Kommunikation: der Kommunikationskanal muss Botschaften puffern
Konzepte:

- ***Briefkasten***: individuelle Botschaftenwarteschlange zwischen Sender und Empfänger
- ***Port***: spezielle Form des Briefkastens, bei der mehrere Sender Daten mit einem Empfänger austauschen können



Kanal kann stationär oder temporär sein,
bei ES hat man häufig stationäre Kanäle

Entfernter Prozeduraufruf mittels Botschaftenaustausch



(1), (3), (4), (6) - Prozeduraufrufe
(2), (5), - Botschaftentransfer

Implementierungsaspekte der Taskverwaltung mit Taskliste mit Taskkontrollblock und Taskkontext

Taskkontext muss bei Unterbrechen/Blockieren gerettet werden.

Taskkontext für schwergewichtige Tasks:		
Taskverwaltung	Speicherverwaltung	Ein-/Ausgabeverwaltung
Programmzähler Statusregister Steuerregister Stapelzeiger Registerblock Taskidentifikation Taskpriorität Taskparameter Elterntask Kindertasks	Startadresse, Größe und Zustand von: Taskcode Taskdaten Taskstapel dynamisch belegtem Taskspeicher (Taskhalde)	Erteilte Ein-/Ausgabeaufträge Zugriffsrechte auf Geräte Angeforderte Geräte Belegte Geräte Zeiger auf Gerätepuffer Zustand belegter Geräte Zeiger auf offene Dateien Zugriffsrechte auf Dateien Zeiger auf Dateipuffer Zustand offener Dateien

Taskkontext für leichtgewichtige Threads:

- **Programmzähler**
- **Statusregister**
- **Steuerregister**
- **Stapelzeiger**
- **Registerblock**