

Synchronisation

Einführung

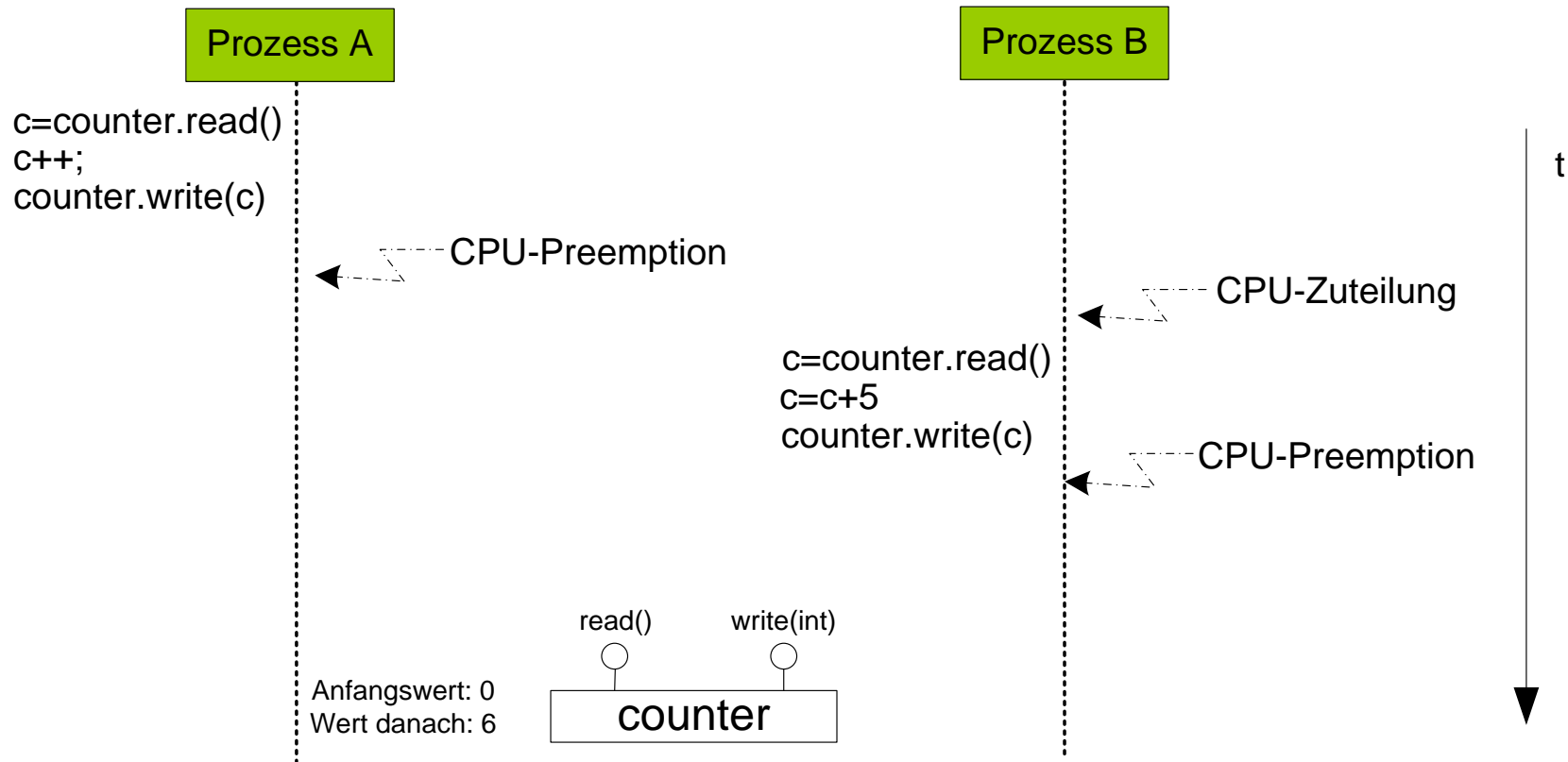
- Nebenläufigkeit
 - Parallele oder quasi-parallele Ausführung von Befehlsfolgen in Prozessen und Threads
 - Verdrängung jederzeit durch das Betriebssystem möglich ohne Einfluss des Anwendungsprogrammierers
- Atomare Aktion
 - Codebereiche, die in einem Stück, also atomar, ausgeführt werden müssen, um Inkonsistenzen zu vermeiden
 - Aber: Eine Unterbrechung durch Verdrängung ist jederzeit möglich

Fallbeispiel 2 (1)

- Ein gemeinsam genutzter Zähler (Counter) wird von zwei Prozessen verändert
- Auch hier kann es zu Inkonsistenzen kommen, die als Lost-Update bezeichnet werden

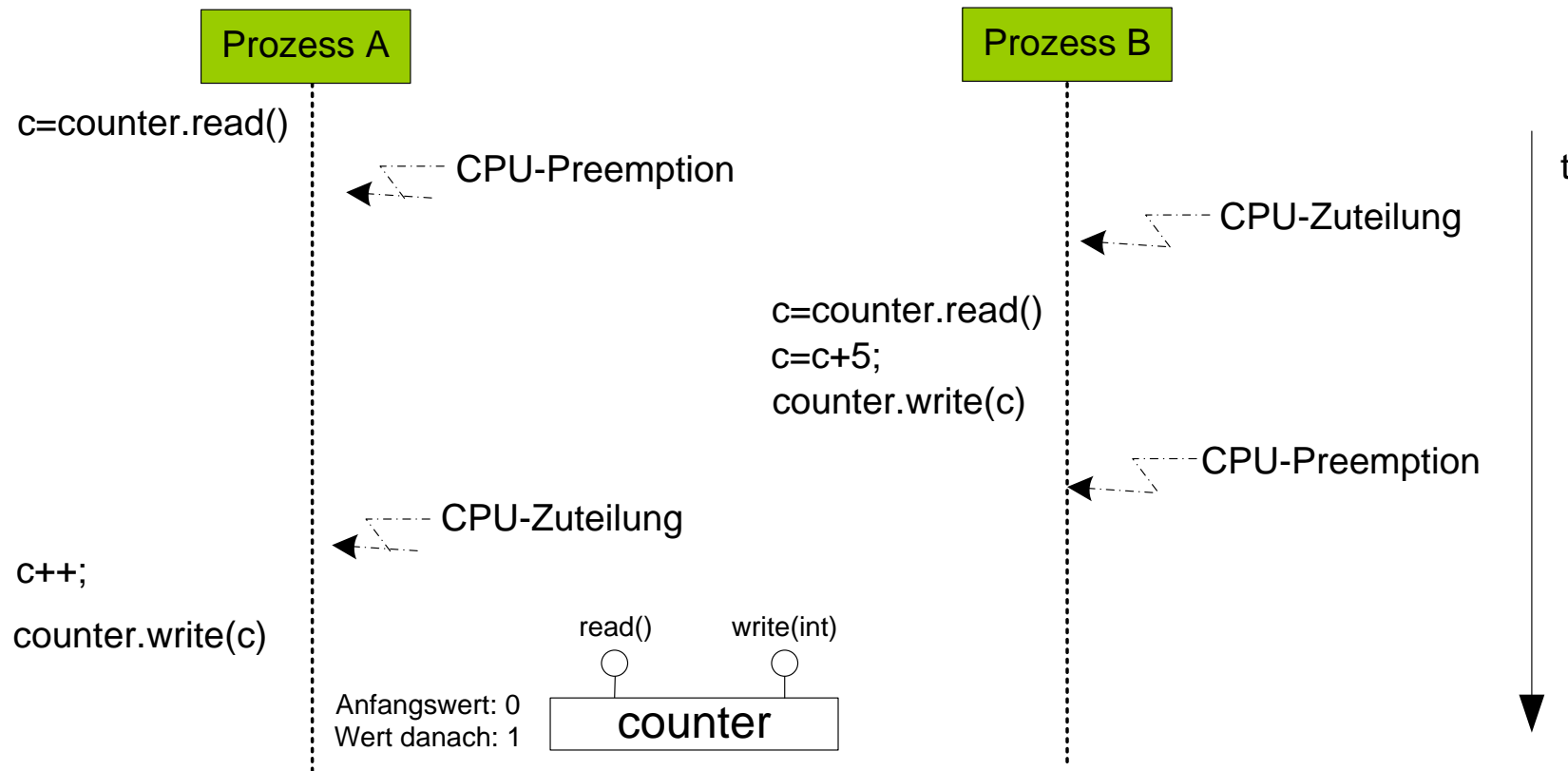
Fallbeispiel 2 (2)

- Counter steht anfangs auf 0
- **Unproblematischer** Ablauf



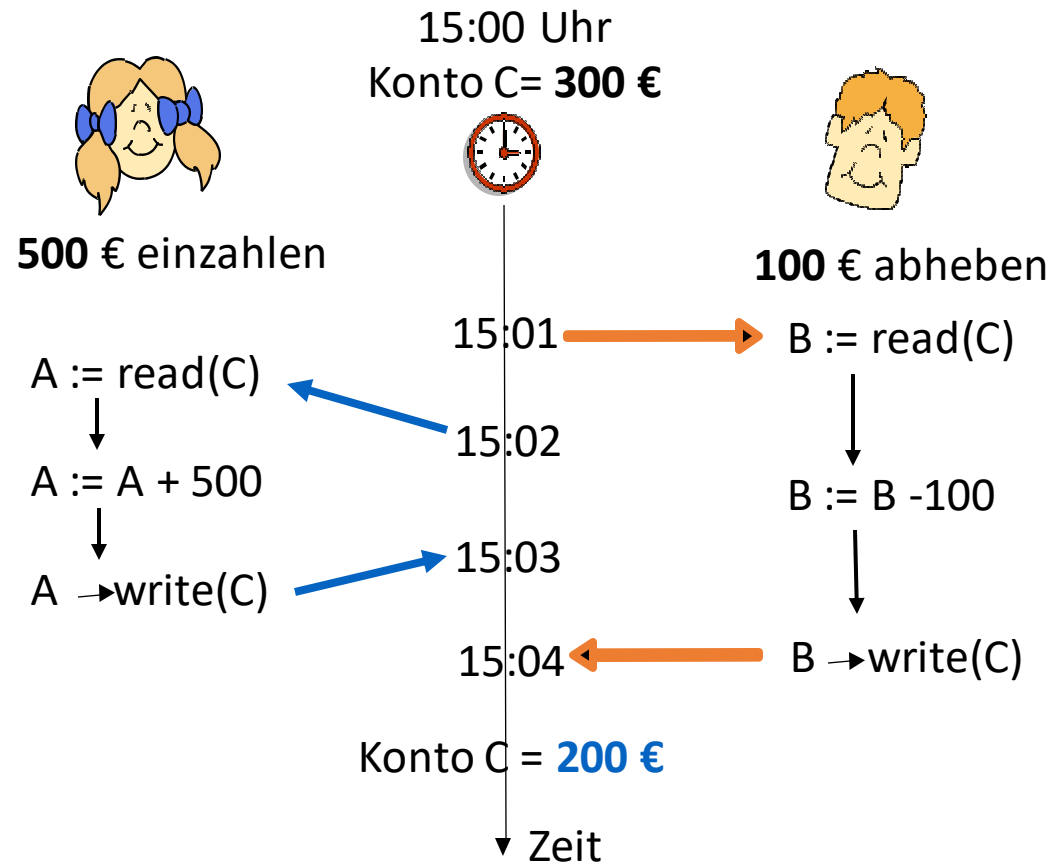
Fallbeispiel 2 (3)

- Fehlerfall: Was passiert bei diesem Ablauf, wenn counter zunächst auf 0 steht?



Fallbeispiel 3 - Kontoführung

- Was passiert wenn zwei Personen gleichzeitig Transaktionen an einem Bankkonto durchführen?
- Paralleler Zugriff auf globale Variable C
- Parallele Prozesse von vielen Benutzern arbeiten gleichzeitig auf den Daten
- Falls **gleichzeitig** zwei Benutzer auf das **gleiche Konto** einzahlen möchten, so kann es sein, dass der Kontostand hinterher nicht stimmt.
- Eine Fehlbuchung tritt nur auf, wenn ein Prozess den anderen Prozess überholt („**race condition**“)



Race Conditions

- Die gezeigten Situationen bezeichnet man auch als Race Conditions
- Problem:
 - mehrere Prozesse greifen unkontrolliert auf gemeinsam genutzte Ressourcen zu
 - nicht vorhersehbar, wann der Scheduler einen laufenden Prozess unterbricht und einem anderen die CPU zuteilt
 - kann in kritischen Situationen erfolgen, ohne dass man es beeinflussen kann
- Sind zwei oder mehrere Prozesse oder Threads nutzen ein gemeinsames Betriebsmittel (Liste, Counter,...)
- Endergebnisse der Bearbeitung sind von der zeitlichen Reihenfolge abhängig
- **Ansatz:** Kontrolle der Ausführungsreihenfolge

Kritische Abschnitte und gegenseitiger Ausschluss

- Auf gemeinsam von mehreren Prozessen oder Threads bearbeitete Daten darf nicht beliebig zugegriffen werden
 - Prozesse bzw. Threads müssen sich zur Bearbeitung gemeinsamer (shared) Ressourcen miteinander **koordinieren**
 - **Synchronisation** erforderlich

Kritische Abschnitte und gegenseitiger Ausschluss

- Man benötigt ein Konzept, das es ermöglicht, gewisse Arbeiten **logisch nicht unterbrechbar** zu machen
 - Die Codeabschnitte, die nicht unterbrochen werden dürfen, werden auch als **kritische Abschnitte** (critical sections) bezeichnet
 - In einem kritischen Abschnitt darf sich immer nur ein Prozess zu einer Zeit befinden
 - Das Betreten und Verlassen eines kritischen Abschnitts muss unter den Beteiligten abgestimmt (synchronisiert) werden
- Ziel: **Gegenseitigen Ausschluss** (mutual exclusion) garantieren

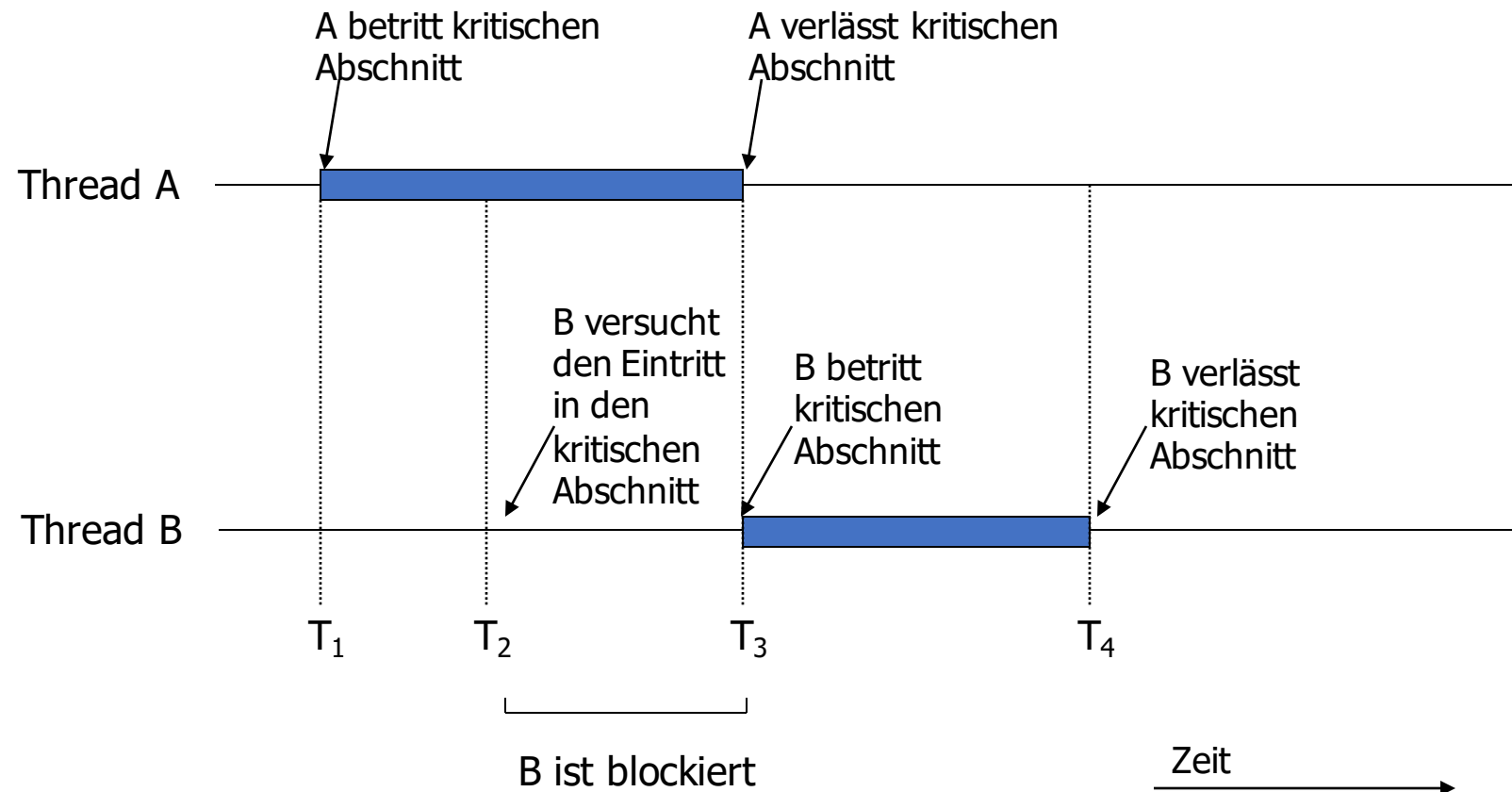
Anforderungen an kritische Abschnitte

Kriterien von Dijkstra:

Eine gute Lösung erfüllt diese vier Bedingungen

1. **Keine zwei Prozesse/Threads dürfen gleichzeitig** in einem kritischen Abschnitt sein (mutual exclusion)
2. **Keine Annahmen über die Abarbeitungsgeschwindigkeit** und die Anzahl der Prozesse/Threads bzw. der verfügbaren Prozessoren – kritischer Abschnitt muss unabhängig davon geschützt werden
3. Kein Prozess/Thread außerhalb eines kritischen Abschnitts darf einen anderen Prozess/Thread **blockieren**
4. Jeder Prozess/Thread, der am Eingang eines kritischen Abschnitts wartet, muss ihn irgendwann betreten dürfen → **kein ewiges Warten** (fairness condition)

Kritische Abschnitte und gegenseitiger Ausschluss



Eigenschaften nebenläufiger Programme

- Blockieren
 - Ein Prozess 1 belegt ein Betriebsmittel, ein zweiter Prozess 2 benötigt dasselbe Betriebsmittel ebenfalls und wird daher blockiert, bis es der Prozess 1 freigegeben hat.
- Verhungern
 - Ein Prozess erhält trotz Rechenbereitschaft keine CPU-Zeit zugeteilt
- Verklemmen
 - Zwei oder mehr Prozesse halten jeder für sich ein oder mehrere Betriebsmittel belegt und versuchen ein weiteres zu belegen
 - Ein anderer Prozess belegt dieses
 - Zyklus von Abhängigkeiten
 - Kein Prozess gibt seine Betriebsmittel frei, und alle Prozesse warten daher ewig

Eigenschaften nebenläufiger Programme

- Sicherheit
 - nie zwei Prozesse in einem kritischen Abschnitt befinden
 - keine Verklemmungen
- Lebendigkeit
 - ein Prozess tritt irgendwann evtl. nach Wartezeit in einen kritischen Abschnitt ein
 - Irgendwann treten alle gewünschten Zustände ein

Wie kann man nun verhindern, dass mehrere Prozesse einen kritischen Abschnitt betreten?

- **Sperre** über eine **Sperrvariable**
- Reine **Softwarelösungen** für **Sperren**
 - **Softwarelösungen** zum **Setzen von Sperren** für **kritische Abschnitte** z.B.: Dekker oder Peterson
- **Interrupt Sperre**
- **Hardwareunterstützung** durch **spezielle Maschinenbefehle**


Keine Lösung – Sperrvariable

- die Prozesse haben eine gemeinsame variable die 0 oder 1 sein kann
 - 0 – kritischer Abschnitt ist blockiert
 - 1 – kritischer Abschnitt kann betreten werden

Probleme:

- Wenn P1 genau die CPU genau hier verliert, **kann P2 auch in die critical Section – Verletzt Regel 1**
- Busy waiting
- Auch der nochmalige check **verschleiert** nur das Problem

```
bool v; /*Gem. Sperrvariable*/
while(true) {
    if(v==false) {
        v=true;
        ...
        /* kritischer Abschnitt*/
        ...
        v=false;
    }
}
```



Sperren: Implementierungsvarianten (1)

- Eine einfache Lösung zur Realisierung von kritischen Abschnitten ist **busy waiting** (aktives Warten)
 - Ein Prozess/Thread testet eine sog. Synchronisationsvariable
 - Test solange, bis Variable einen Wert hat, der den Zutritt erlaubt → Man braucht einen speziellen Befehl dazu
- Dieses **Polling** wird auch als **Spinlock** bezeichnet
 - Spinlocks in Betriebssystemen sind oft anzutreffen
- Manchmal ist es besser, einen Prozess/Thread „schlafen“ zu legen und erst wieder zu wecken, wenn er in den kritischen Bereich darf
 - Das ist aber oft nicht so leistungsfähig wie Spinlocks

Sperren: Implementierungsvarianten (2)

- **Vorteile** von Spinlocks:

- Weniger Kontextwechsel notwendig
- Gut, wenn Sperrzeit üblicherweise sehr kurz ist, kürzer als Kontextwechsel

- **Nachteile** von Spinlocks:

- Warten benötigt CPU-Zeit
- Blockierungen bei Singleprozessoren nicht auszuschließen, Beispiel:
 - Thread mit niedriger Priorität hält den Lock und wird suspendiert
 - Höher priorisierter Thread möchte den Lock und bleibt in der Warteschleife (bekommt immer vor dem anderen Thread die CPU)
 - Niedrig priorisierter Thread bekommt die CPU nicht mehr (wird aber bei guten Schedulingen vermieden)

Sperren, Implementierungsvarianten (3)

- Hardware-Unterstützung zur Synchronisation:
 - Alle **Interrupts ausschalten**; Geht nur bei Monoprozessoren. Warum?
→ Ist aber meist eine schlechte Lösung!

...

```
Interrupts sperren  
  (Maskieren, z.B. Windows IRQL hochsetzen)
```

```
/* Kritischer Abschnitt beginnt */
```

```
...
```

```
/* Kritischer Abschnitt endet */
```

```
Interrupts freigeben  
  (Demaskieren)
```

Sperren, Implementierungsvarianten (3)

Interrupts sperren

- Prozess sperrt vor Eintritt in die critical Section alle Interrupts
- Prozess gibt nach critical section Interrupts wieder frei
- Scheduler kann keinen anderen Prozess wählen (timer interrupt)

Probleme:

- Betrifft in Mehrkern Systemen nur eine CPU
- Braucht Spezialprivilegien
- Kein verdrängendes (preemptive) Multitasking mehr: Wenn ein Prozess die CPU nicht mehr freigibt ist das ganze System blockiert

Hinweis: wird im Kernel für sehr kurze kritische Sections benutzen.

Sperren, Implementierungsvarianten (4)

- Hardware-Unterstützung zur Synchronisation erforderlich:
 - **Atomare Instruktionsfolge** über nicht unterbrechbare Maschinenbefehle in einem einzigen **nicht** unterbrechbaren Speicherzyklus → wichtig bei Multiprozessoren!
 - Praktische Beispiele hierfür:
 - **Test-and-Set-Lock** (TSL = test and set lock) bzw. **TAL**
 - Lesen und Ersetzen einer Speicherzelle in einem Speicherzyklus
 - **Compare-and-Swap**
 - Vergleich und Austausch zweier Variablenwerte in einem Speicherzyklus
 - **Fetch-and-Add**
 - Lesen und Inkrementieren einer Speicherzelle in einem Speicherzyklus
 - **Exchange-Befehl CMPXCHG dest, src (im Intel-Befehlssatz)**
 - Inhalte von src und dest werden ausgetauscht (Register und Speicherbereiche als Quelle und Ziel möglich)

Semaphore

- Dijkstra (1965) führte das Konzept der Semaphore zur Lösung des Mutual-Exclusion-Problems ein
- Zwei elementare Operationen
 - **P()**
 - Aufruf bei Eintritt in den kritischen Abschnitt, Operation auf Semaphor
 - Aufrufender Prozess wird in den Wartezustand versetzt, wenn sich ein anderer Prozess im kritischen Abschnitt befindet → Warteschlange
 - **V()**
 - Aufruf bei Verlassen des kritischen Abschnitts
 - Evtl. wird einer der wartenden Prozesse aktiviert und darf den kritischen Abschnitt betreten

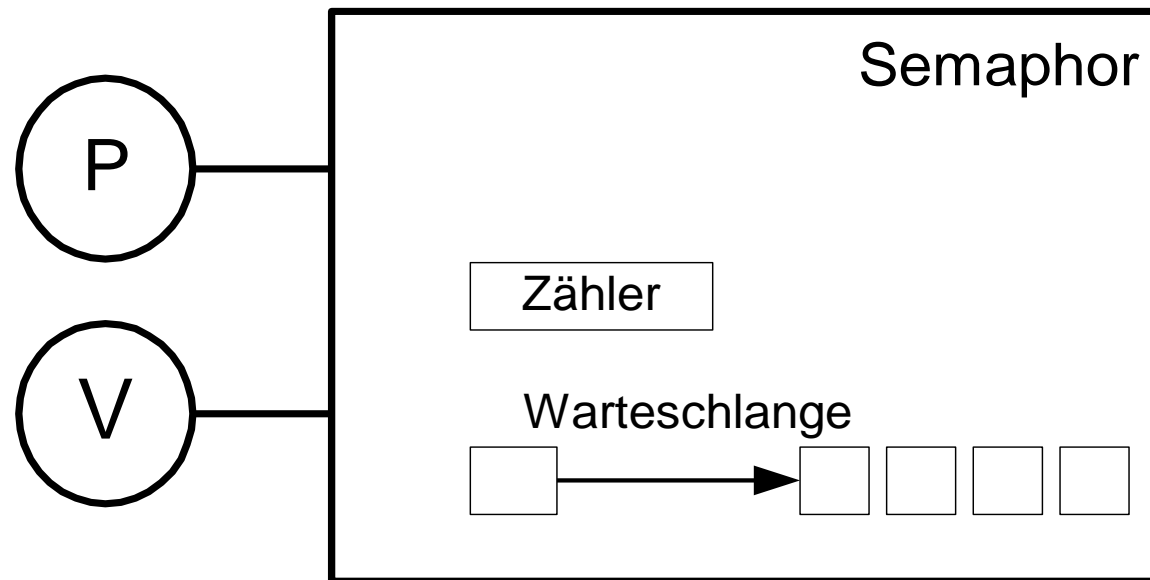


Semaphore

- Man braucht:
 - Semaphor-Zähler
 - Warteschlange

Niederländisch:

- P kommt evtl. von probeeren = versuchen oder passeeren = passieren
- V kommt evtl. von verhogen = erhöhen oder vrijgeven = freisetzen



P: P-Operation auch Down-Operation genannt

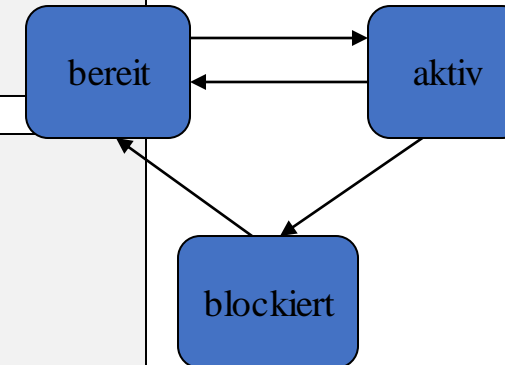
V: V-Operation, auch Up-Operation genannt

Semaphore, Algorithmus

- s ist der Semaphor-Zähler, Init: $s \geq 1$

```
01: void P() {  
02:     if (s >= 1) {  
03:         s = s - 1; // Der die P-Operation ausführende Prozess  
                     // setzt seinen Ablauf fort  
04:     } else {  
05:         // Der die P-Operation ausführende Prozess wird zunächst  
         // gestoppt, in den Wartezustand versetzt und in einer  
         // dem Semaphor S zugeordneten Warteliste eingetragen  
06:     }  
07: }
```

```
01: void V() {  
02:     s = s + 1;  
03:     if (Warteliste ist nicht leer) {  
04:         // Aus der Warteliste wird ein Prozess ausgewählt  
         // und aufgeweckt  
05:     }  
06: }  
07: // Der die V-Operation ausführende Prozess macht weiter
```

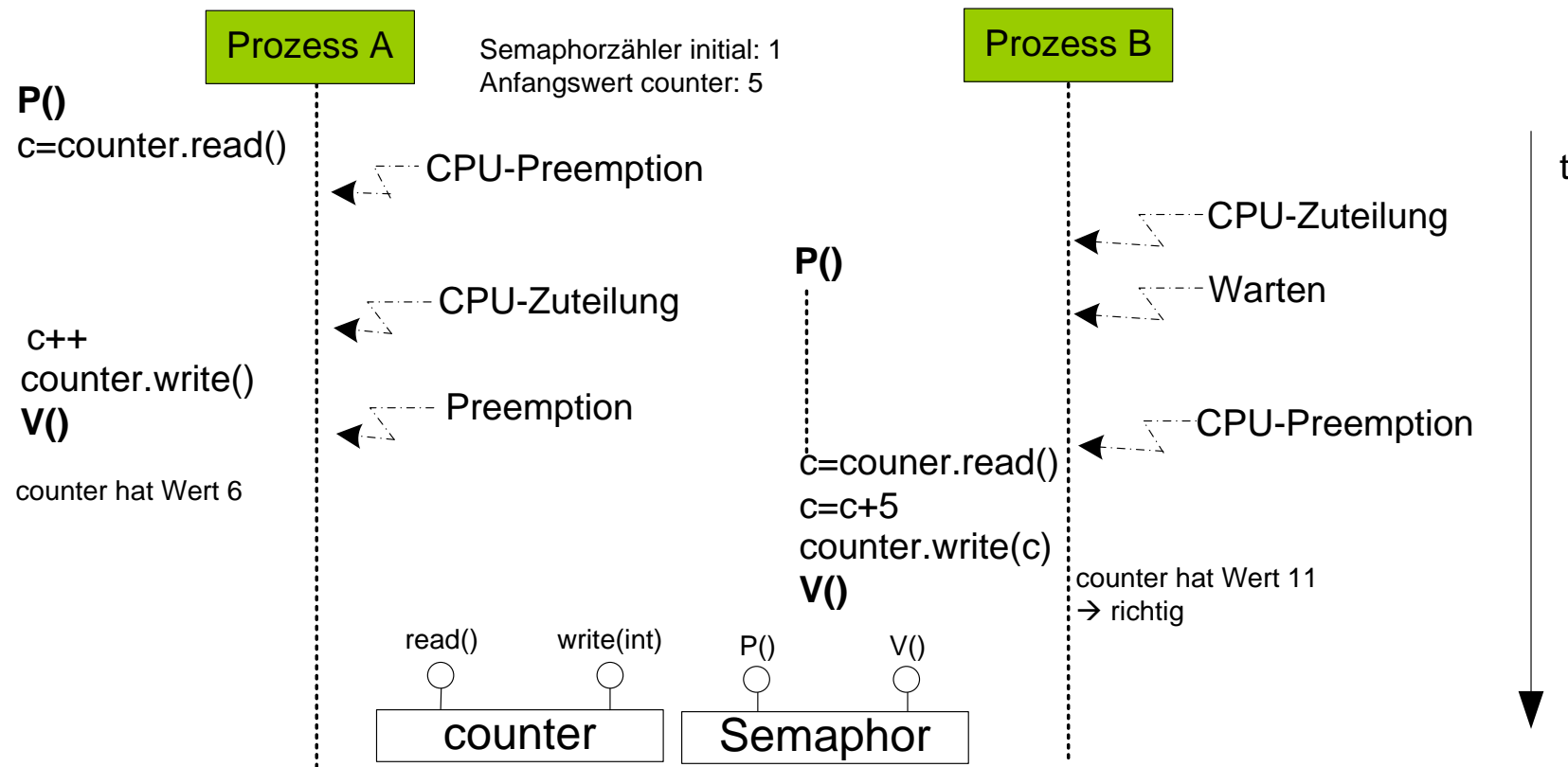


Semaphore, Beispielnutzung

```
...  
01: P();           // kritischer Abschnitt besetzt  
02: c=counter.read(); // Im kritischen Abschnitt  
03: c++;             // Im kritischen Abschnitt  
04: counter.write(c);  
05: V();           // Verlassen des kritischen  
                        // Abschnitts, Aufwecken eine  
...                  // wartenden Prozesses
```

- P() und V() sind selbst wieder ununterbrechbar, also **atomare Aktionen**
- Atomare Aktionen werden **ganz** oder **gar nicht** ausgeführt

Semaphore: Vermeidung des Lost-Update-Problems



Semaphore, einfache Form: Mutex

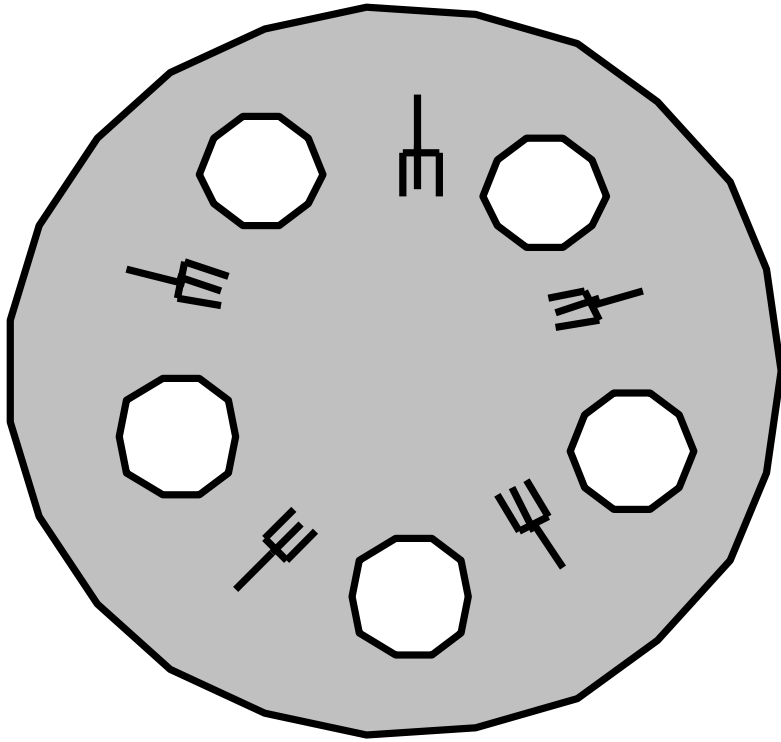
- Wenn man auf den Zähler im Semaphor verzichten kann, kann eine einfachere Form angewendet werden
 - Diese wird als **Mutex** bezeichnet
- Ein Mutex ist leicht und effizient zu implementieren
- Ein Mutex ist eine Variable, die nur zwei Zustände haben kann:
 - locked und unlocked
- Man braucht also nur 1 Bit zur Implementierung
- Zwei Operationen:
 - mutex_lock
 - mutex_unlock

Philosophenproblem

- Dijkstra und Hoare (1965), Dining Philosophers Problem:
 - Fünf Philosophen sitzen um einen Tisch herum
 - Jeder hat einen Teller mit Spaghetti vor sich
 - Zwischen den Tellern liegt je eine Gabel (5 Gabeln)
 - Zum Essen braucht ein Philosoph 2 Gabeln
 - Ein Philosoph isst und denkt abwechselnd
 - Wenn er hungrig wird, versucht er in beliebiger Reihenfolge die beiden Gabeln links und rechts von seinem Teller zu nehmen
 - Hat er sie bekommen, isst er und legt sie dann wieder auf ihren Platz zurück

Prozessverwaltung: Philosophenproblem

Lösungsalgorithmus:



```
...
01: static int n = 5;

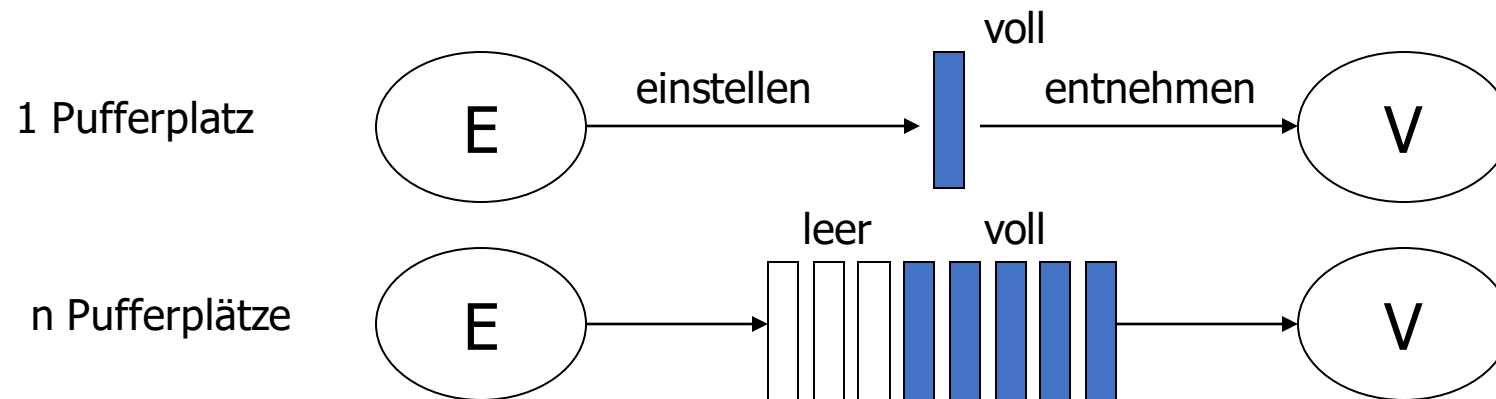
02: void philosopher(int i) {
03:   while (true) {
04:     think();
05:     take_fork(i);
06:     take_fork((i+1) % n);
07:     eat();
08:     put_fork(i);
09:     put_fork((i+1) % n);
10:   }
11: }
```

Warum funktioniert der angegebene Algorithmus nicht?

Finden Sie eine Lösung, bei der zwei Philosophen gleichzeitig essen können und keiner verhungern muss!

Erzeuger-Verbraucher-Problem

- Ein oder mehrere Erzeugerprozesse (producer) produzieren
- Ein oder mehrere Verbraucherprozesse (consumer) konsumieren
- Endlich große Pufferbereiche zwischen den Prozessen
 - Erzeuger füllt auf
 - Verbraucher nimmt heraus
- **Flusskontrolle** erforderlich
 - Erzeuger legt sich schlafen, wenn Puffer voll ist und wird vom Verbraucher aufgeweckt, wenn wieder Platz ist
 - Verbraucher legt sich schlafen, wenn Puffer leer ist und wird vom Erzeuger wieder aufgeweckt, wenn wieder ein Objekt im Puffer ist



Erzeuger-Verbraucher-Problem

- Lösung mit zwei Semaphoren und einem Mutex:
 - **mutex** für den gegenseitigen Ausschluss beim Pufferzugriff
 - **frei** und **belegt** zur Synchronisation

Erzeuger

```
01: while (true) {  
02:   produce(item);  
03:   P(frei);  
04:   P(mutex);  
05:   putInBuffer(item);  
06:   V(mutex);  
07:   V(belegt);  
08: }
```

Verbraucher

```
01: while (true) {  
02:   P(belegt);  
03:   P(mutex);  
04:   getFromBuffer(item);  
05:   V(mutex);  
06:   V(frei);  
07:   consume(item);  
08: }
```

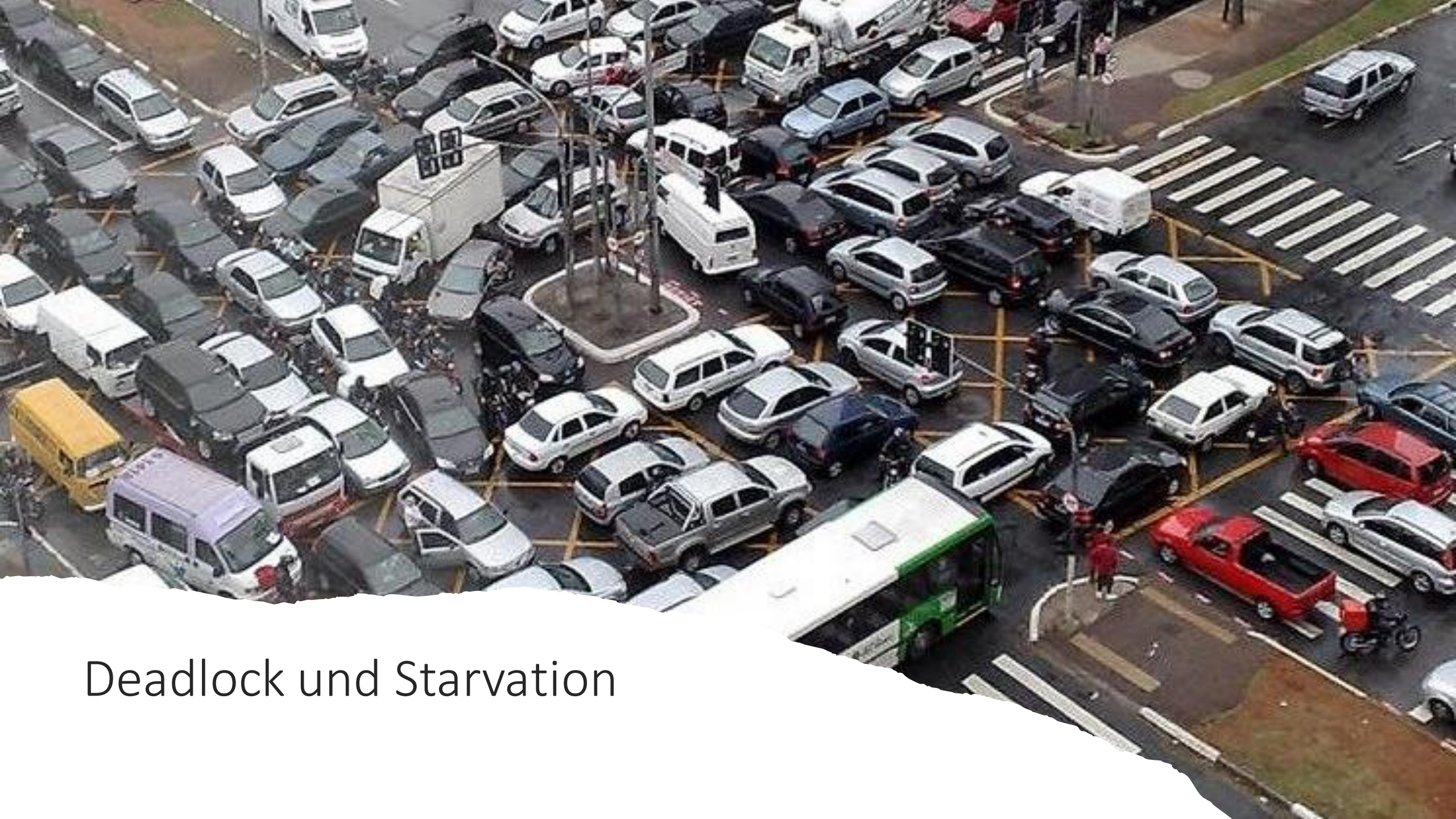
Initialisierung:

belegt = 0; // Verbraucher muss erst mal warten, zählt die belegten Puffer

frei = N (Puffergröße); // Am Anfang ist Puffer leer, zählt die leeren Puffer

mutex = 1; // Mutual Exclusion Zugang nur für Pufferbearbeitung

→ am Anfang darf nur Erzeuger etwas tun



Deadlock und Starvation

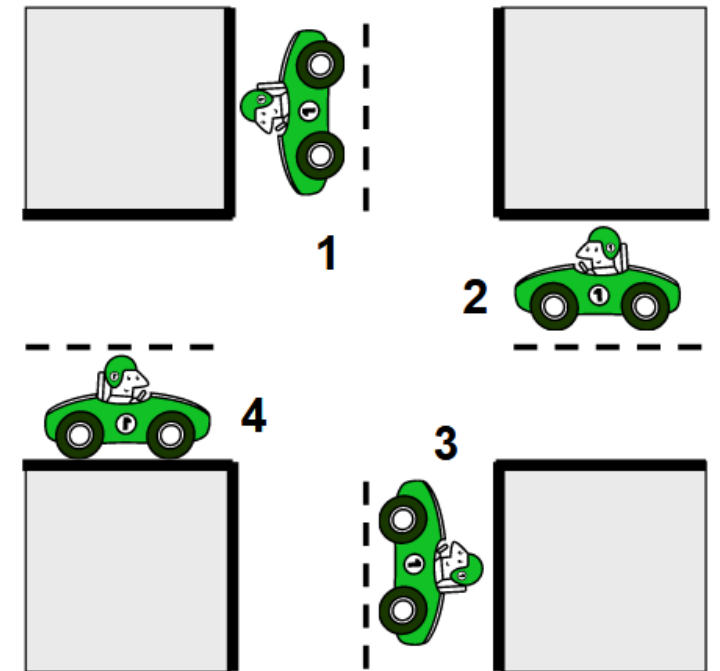
Deadlock und Starvation

Beim „Wettbewerb“ der Prozesse um Betriebsmittel kann es zum Deadlock kommen.

Definition: „Eine Menge von Prozessen befindet sich in einem Deadlock wenn jeder Prozess aus der Menge auf eine Ereignis wartet, dass nur eine anderer Prozess aus der Menge auslösen kann.“

Bsp:

- P1 fordert Drucker an und bekommt exklusiven Zugriff
- P2 fordert Bandstation und bekommt exklusiven Zugriff
- P1 fordert Bandstation an ohne Drucker freizugeben
- P2 fordert Drucker an ohne Bandstation freizugeben



Deadlock – 4 Bedingungen

3 Voraussetzungen des **Systems** + **Bestimmte Ereignisfolge**:

1. **Mutual Exclusion (Exklusive Nutzung)**

exklusiver Zugriff auf Ressourcen – BM ist frei oder belegt

2. **Hold and Wait (Wartebedingung)**

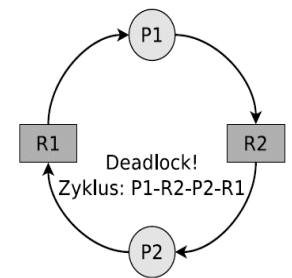
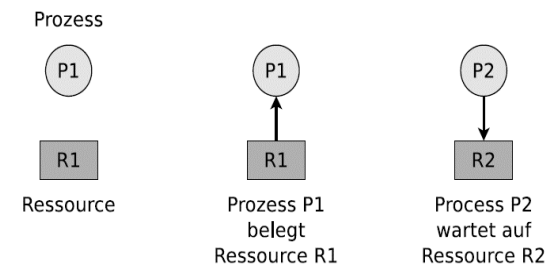
Prozess kann Ressourcen halten, während er auf andere Ressourcen wartet

3. **No Preemption (Nichtentziehbarkeit)**

zugewiesene Ressourcen werden den Prozessen nicht weggenommen

4. **Circular Wait (Geschlossene Kette)**

Geschlossene Kette von Prozessen, von denen jeder Prozess mindestens eine Ressource, die von einem anderen Prozess benötigt wird, hält



Deadlock – Beispiel

```
/* Prozess 1 */  
...  
Fordere Ressource 1 an  
...  
Fordere Ressource 2 an  
...  
Benutze beide Ressourcen  
...  
Gib beide Ressourcen frei
```

```
/* Prozess 2 */  
...  
Fordere Ressource 2 an  
...  
Fordere Ressource 1 an  
...  
Benutze beide Ressourcen  
...  
Gib beide Ressourcen frei
```

Annahme: exklusiver Zugriff auf Ressourcen!

Wenn Prozess 1 seine Arbeit rechtzeitig vor Anforderung von Ressource 2 von Prozess 2 beendet, **tritt kein Deadlock** auf

Achtung: Das Betriebssystem kann zu jedem Zeitpunkt jeden beliebigen nicht blockierten Prozess ausführen. Streng sequentielle Ausführung ist nicht unbedingt optimal.

Deadlock – Möglicher Deadlock

/* Prozess 1 */

...

t_1 Fordere Ressource 1 an

...

t_4 Fordere Ressource 2 an

...

Benutze beide Ressourcen

...

Gib beide Ressourcen frei

/* Prozess 2 */

...

t_2 Fordere Ressource 2 an

...

t_3 Fordere Ressource 1 an

...

Benutze beide Ressourcen

...

Gib beide Ressourcen frei

Annahme: exklusiver Zugriff auf Ressourcen!

t_1, \dots, t_4 sind die Zeitpunkte wobei für alle t_n $n:N$ gilt t_n passiert strikt vor t_{n+1}

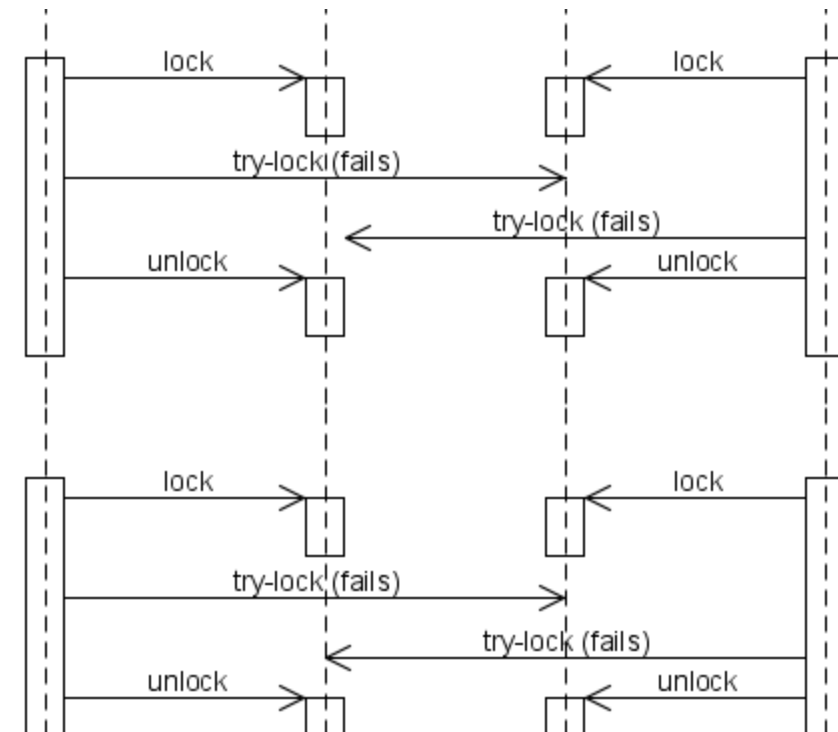
Livelock

Sind die Prozesse höflich, geben Sie die erste Sperre wieder auf, wenn sie die zweite Sperre nicht bekommen können.

- Soll helfen Deadlocks zu vermeiden
- Machen es beide Prozesse befinden sie sich in einem Livelock – sie blockieren nicht aber können nicht sinnvoll weitermachen.

After the try-locks fail, both threads release their lock and no work is done

The same locking pattern is repeated



Livelock: Wie zwei Personen die am Gang aneinander vorbeigehen wollen und immer in dieselbe Richtung ausweichen

Deadlock: sie stehen sich gegenüber und keiner weicht aus.

Deadlock – Strategien zur Behandlung

4 Strategien

1. Problem ignorieren
2. Erkennen und beheben
3. Dynamische Verhinderung durch vorsichtige Ressourcenzuteilung
4. Vermeidung von Deadlocks – eine der 4 Bedingungen muss unerfüllbar werden

Strategien zur Behandlung - Ignorieren

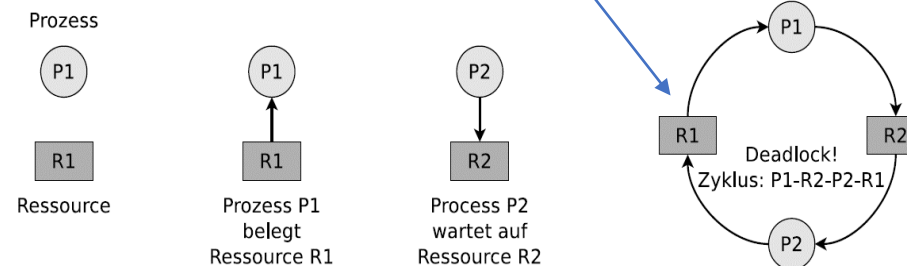
Vogel Strauß Taktik – Ignorieren des Problems

- Frage: wie oft tritt ein Deadlock auf?.
- Bei z.B. einer Verklemmung pro Monat kann das Problem durchaus ignoriert werden!
- Interaktive Benutzer verlieren Geduld Abbruch/Reboot
- Bei Batch-Systemen wird der Deadlock bei der täglichen oder wöchentlichen Systemwartung entdeckt.

Strategien zur Behandlung - Erkennen

Entdecken und Beheben:

- BS hält Anforderungen und Freigaben von BM fest
- Darstellung als Graph
- Bei jeder Anforderung/Freigabe wird der Graph auf Zyklen untersucht



Alternativ und billig, aber gefährlich:

- Zyklisch Prüfen ob ein Prozess lange (einige Std) blockiert ist und entfernen
- Kann zu inkonsistenten Daten führen

Strategien zur Behandlung - Verhinderung

Banker Algorithmus:

Wie eine Bank niemals so viel Geld bereithält, dass alle Kreditrahmen der Kunden voll ausgeschöpft werden können, kann auch das BS die Ressourcen so verwalten, **dass immer mindestens ein Prozess voll befriedigt werden kann**. Dazu muss bei jedem Prozess ein Maximalwert für die anzufordernden BM existieren.

Strategien zur Behandlung - Verhinderung

Beispiel: Der Rechner hat 10 Magnetbandstationen

| Zustand 1 | | | Zustand 2 | | | Zustand 3 | | |
|-----------|------|------|-----------|------|------|-----------|------|------|
| Prozess | bel. | Max. | Prozess | bel. | Max. | Prozess | bel. | Max. |
| A | 0 | 6 | A | 1 | 6 | A | 1 | 6 |
| B | 0 | 5 | B | 1 | 5 | B | 2 | 5 |
| C | 0 | 4 | C | 2 | 4 | C | 2 | 4 |
| D | 0 | 7 | D | 4 | 7 | D | 4 | 7 |
| Gesamt: | 0 | | | 8 | | | 9 | |
| sicher | | | sicher | | | unsicher | | |
| 10 frei | | | 2 frei | | | 1 frei | | |

Ein Zustand ist dann sicher, wenn das BS **mindestens bei einem Prozess seine Maximalforderung erfüllen kann** (die anderen müssen u. U. warten).

- Zustand 1 ist sicher, da jeder Prozess befriedigt werden kann.
- Zustand 2 ist sicher, da Prozess C befriedigt werden kann.

Strategien zur Behandlung - Vermeidung

Vermeidung durch unterlaufen der Bedingung Mutual Exclusion

- Beispiel Drucker: Es wird ein Prozess eingeführt, der als einziger Prozess den Drucker exklusiv besitzt.
- Durchbrechen der Wartebedingung, indem ein Prozess die vorher angeforderten BM freigibt. Nur wenn die Anforderung erfolgreich war, erhält er sie zurück.

Vermeidung durch unterlaufen der Bedingung Hold & Wait

- Durch geeignete BM-Zuweisung kann eine Verklemmung vermieden werden, wenn einige Informationen im Voraus verfügbar sind.
- Prozesse müssen im Vorhinein wissen welche BM sie benötigen (möglich bei Stapelverarbeitung – am Anfang des Jobs – Belastung für Programmierer)

Strategien zur Behandlung - Vermeidung

Vermeidung durch unterlaufen der Bedingung Ununterbrechbarkeit

- Gewaltsam Ressourcen entziehen (z.B. Drucker – Prozess druckt gerade)
- Schwierig bis unmöglich – es können nicht alle Ressourcen virtualisiert werden.

Vermeidung durch unterlaufen der zyklischen Wartebedingung

- Durchnummerieren der Ressourcen – jeder Prozess muss sie in dieser Reihenfolge anfordern
- Könnte passieren, dass keine Ordnung gefunden wird - zu viele Ressourcen um sinnvoll zu sein.