MAS: Betriebssysteme

Koordination und Synchronisation:
Kritische Abschnitte, Sperren, Semaphore und Mutexe

T. Pospíšek

Gesamtüberblick

1. Einführung in Computersysteme
2. Entwicklung von Betriebssystemen
3. Architekturansätze
4. Interruptverarbeitung in Betriebssystemen
5. Prozesse und Threads
6. CPU-Scheduling
7. **Synchronisation und Kommunikation**
8. Speicherverwaltung
9. Geräte- und Dateiverwaltung
10. Betriebssystemvirtualisierung

Zielsetzung

- Der Studierende soll die Probleme der Parallelverarbeitung verstehen und einschätzen können
- Der Studierende soll Konzepte zur Vermeidung von Race Conditions sowohl auf Betriebssystemebene als auch auf Anwendungsebene erläutern können
- Der Studierende soll verstehen, wie Sperren, Semaphore und Mutexe funktionieren und wie man sie implementiert

Überblick

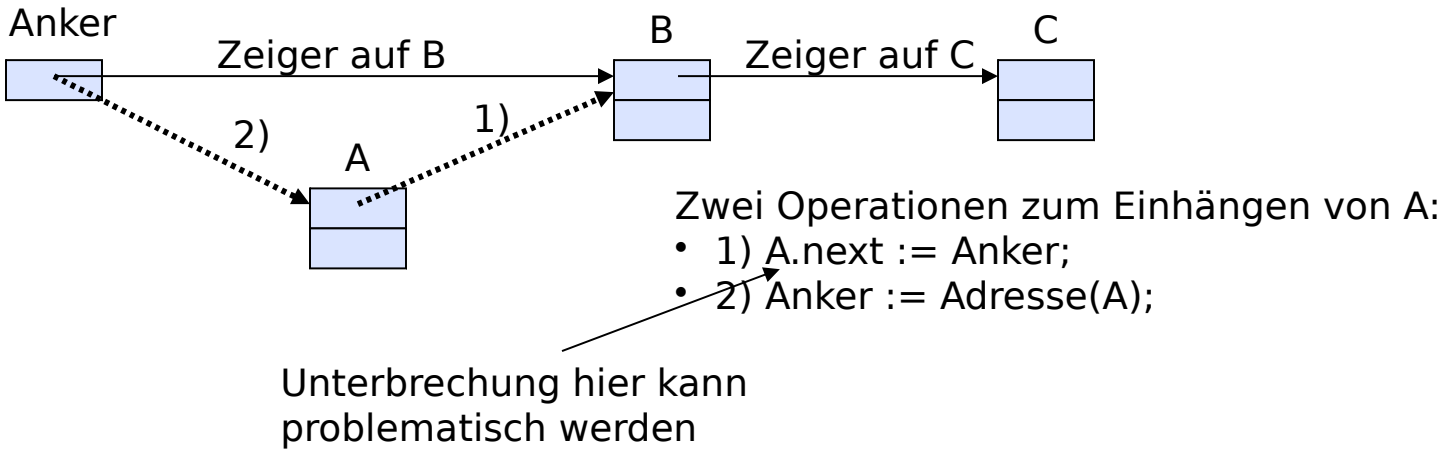
1. **Einführung**
2. Kritische Abschnitte und gegenseitiger Ausschluss
3. Sperren, Semaphore und Mutex
4. Diverse Synchronisationsprobleme

Einführung

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

Konflikte, Fallbeispiel 1 (1)

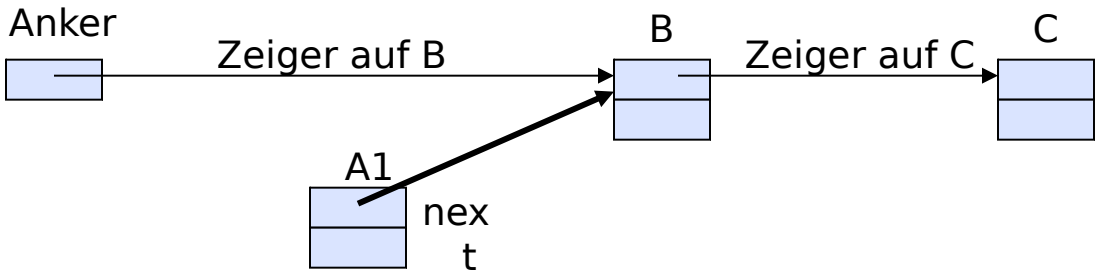
- Beispiel zur Verdeutlichung des Problems:
  - Mehrere Prozesse bearbeiten eine gemeinsame Liste von Objekten (z.B. die Prozesslisten der Run-Queue)
  - Ein Prozess hängt ein neues Objekt vorne in die Liste ein
    - Prozesse können zu beliebigen Zeiten unterbrochen werden
    - Versucht ein zweiter Prozess auch, ein Objekt vorne anzuhängen, gibt es möglicherweise Probleme



Konflikte, Fallbeispiel 1 (2)

- Prozess 1 führt 1. Anweisung aus
  - A1.next := Anker
  - jetzt wird die CPU entzogen

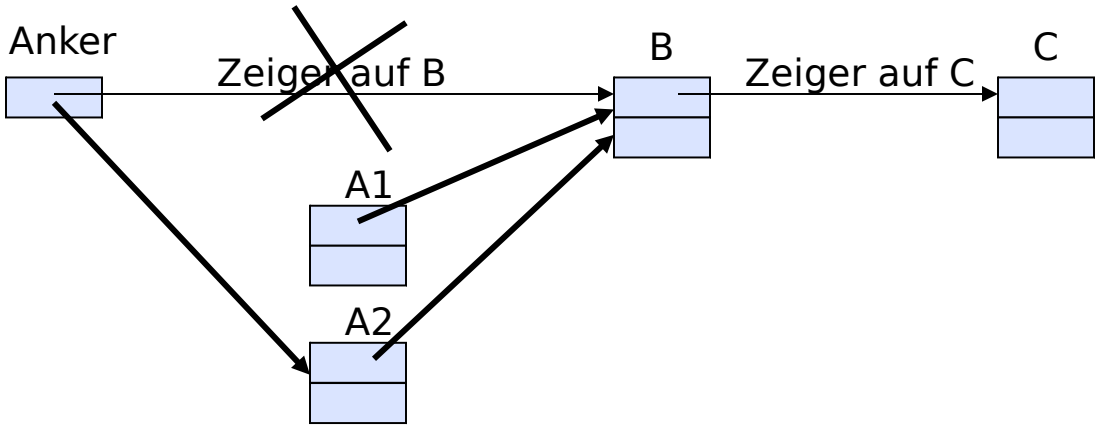
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- 2) Anker := Adresse(A);



Konflikte, Fallbeispiel 1 (3)

- Prozess 2 führt 1. und 2. Anweisung aus
  - A2.next := Anker
  - Anker := Adresse (A2)
  - Jetzt wird die CPU entzogen

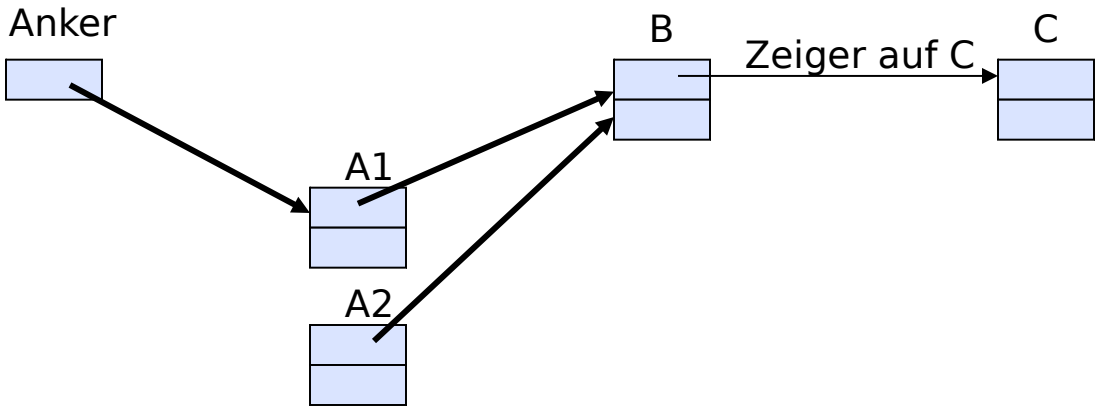
Zwei Operationen zum Einhängen:
- **1) A.next := Anker;**
- **2) Anker := Adresse(A);**



Konflikte, Fallbeispiel 1 (4)

- Prozess 1 führt die 2. Anweisung aus
  - Anker := Adresse (A1)
  - Jetzt wird die CPU entzogen

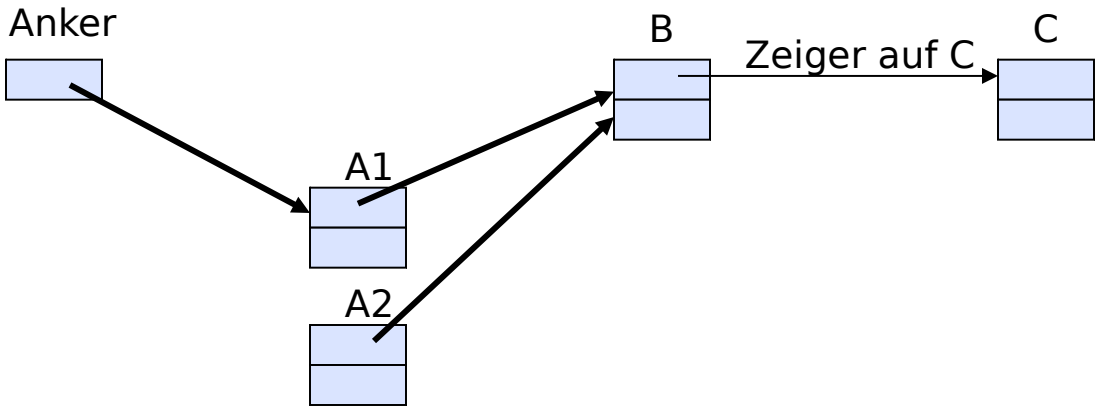
Zwei Operationen zum Einhängen:
- 1) A.next := Anker;
- 2) Anker := Adresse(A);



Konflikte, Fallbeispiel 1 (5)

- Prozess 1 führt die 2. Anweisung aus
  - Anker := Adresse (A1)
  - A2 wird zur Leiche

Zwei Operationen zum Einhängen:
- 1) A.next := Anker;
- **2) Anker := Adresse(A);**

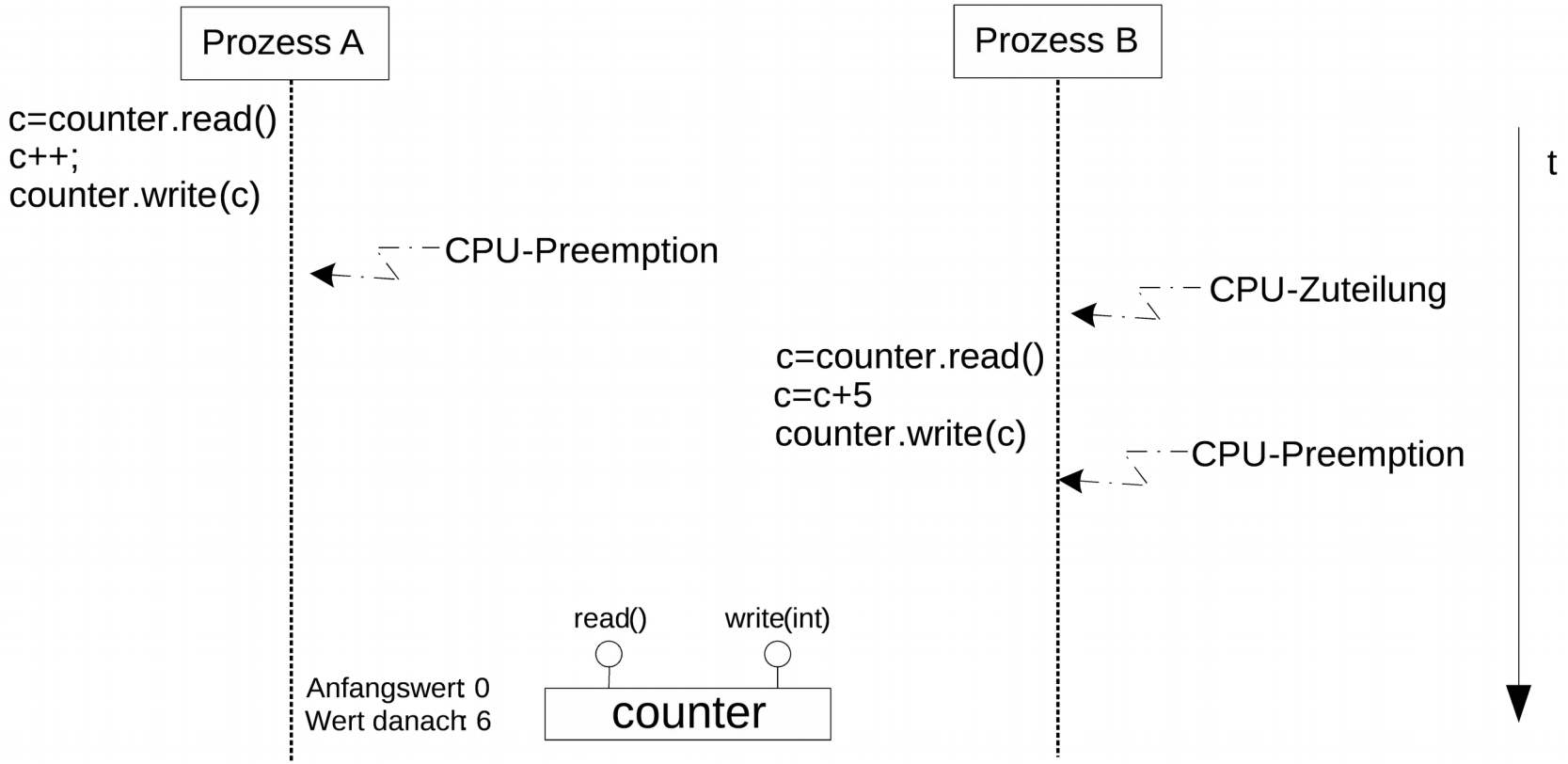


Konflikte, Fallbeispiel 2 (1)

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

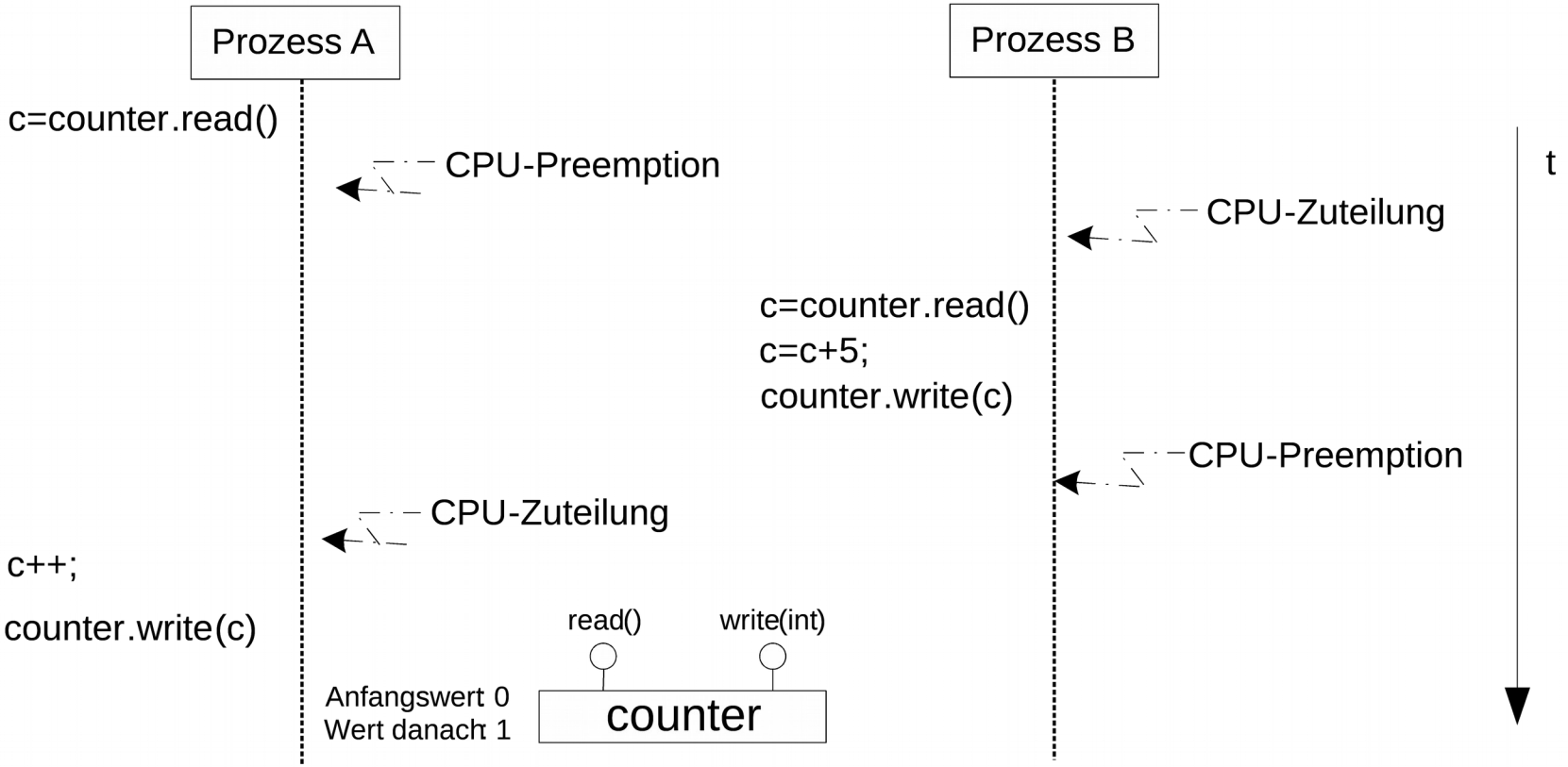
Konflikte, Fallbeispiel 2 (2)

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



Konflikte, Fallbeispiel 2 (3)

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



Race Conditions

- Diese Situationen bezeichnet man auch als Race Conditions
  - Zwei oder mehrere Prozesse oder Threads nutzen ein gemeinsames Betriebsmittel (Liste, Counter,...)
  - Endergebnisse der Bearbeitung sind von der zeitlichen Reihenfolge abhängig

Überblick

1. Einführung
2. **Kritische Abschnitte und gegenseitiger Ausschluss**
3. Sperren, Semaphore und Mutex
4. Diverse Synchronisationsprobleme

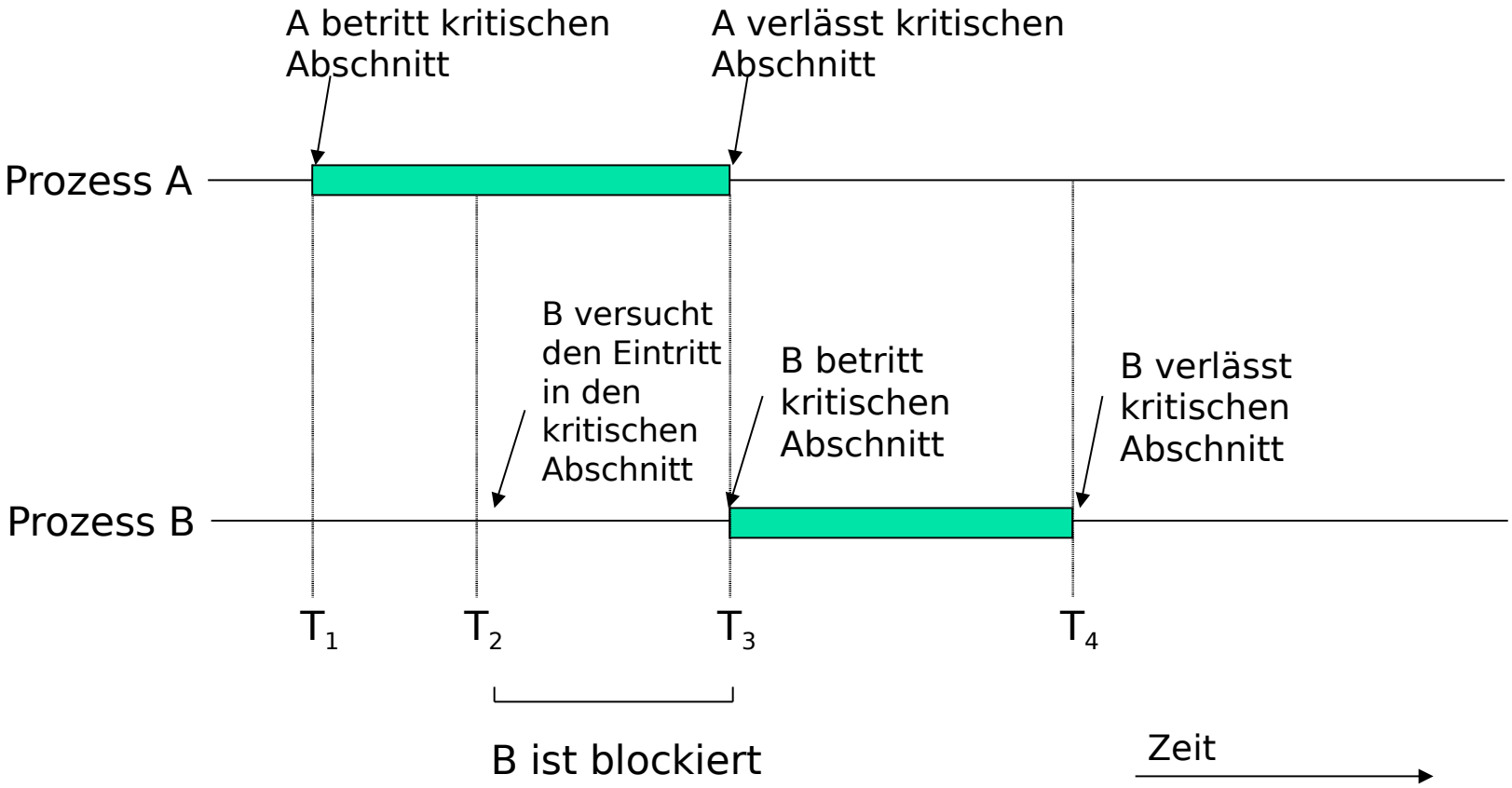
Kritische Abschnitte und gegenseitiger Ausschluss

- 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 abgestimmt (synchronisiert) werden
- Ziel: **Gegenseitigen Ausschluss** (mutual exclusion) garantieren

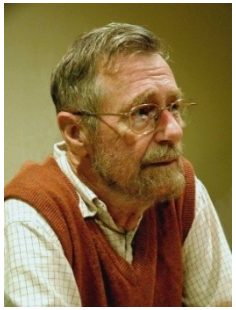
Kritische Abschnitte und gegenseitiger Ausschluss



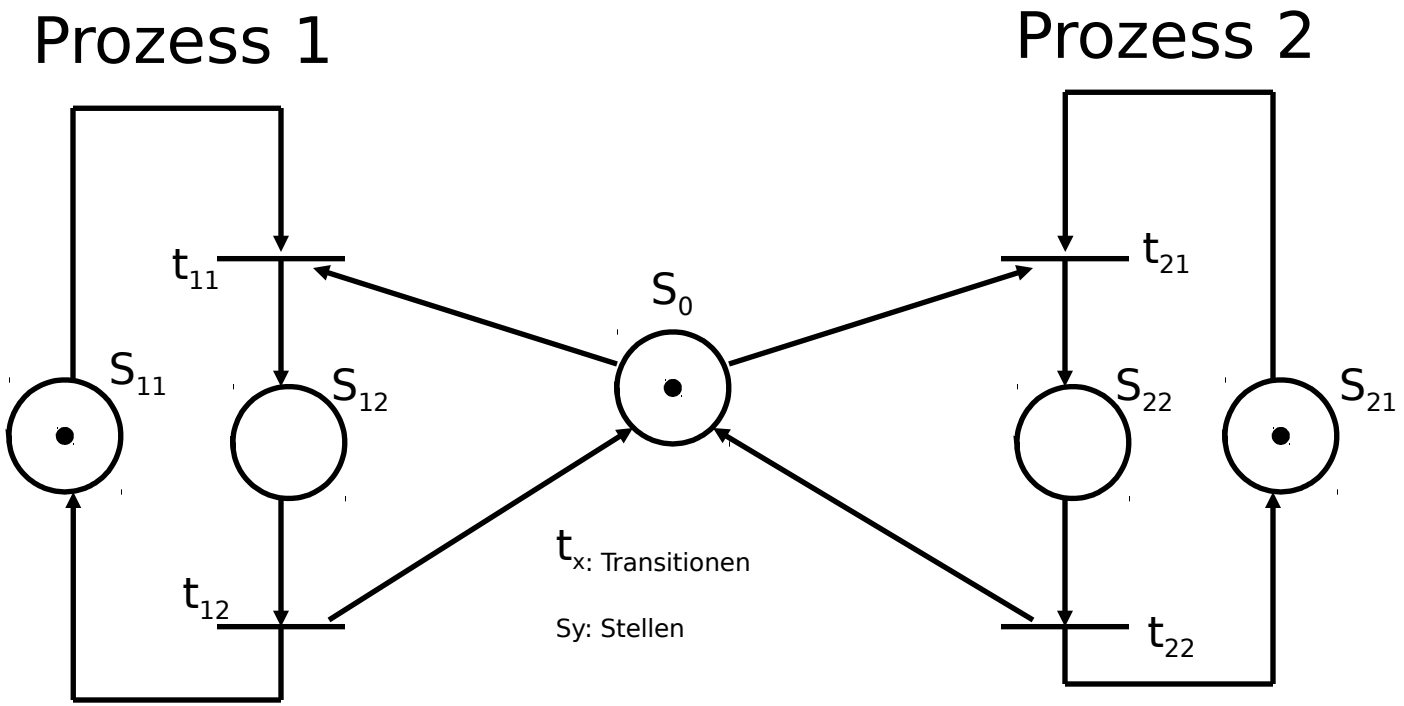
Nach Tanenbaum (2009)

Anforderungen an kritische Abschnitte

- Kriterien von Dijkstra (1965):
  - **Keine zwei Prozesse dürfen gleichzeitig** in einem kritischen Abschnitt sein (mutual exclusion)
  - **Keine Annahmen über die Abarbeitungsgeschwindigkeit** und die Anzahl der Prozesse bzw. Prozessoren
  - Kein Prozess außerhalb eines kritischen Abschnitts darf einen anderen Prozess **blockieren**
  - Jeder Prozess, der am Eingang eines kritischen Abschnitts wartet, muss ihn irgendwann betreten dürfen → **kein ewiges Warten** (fairness condition)

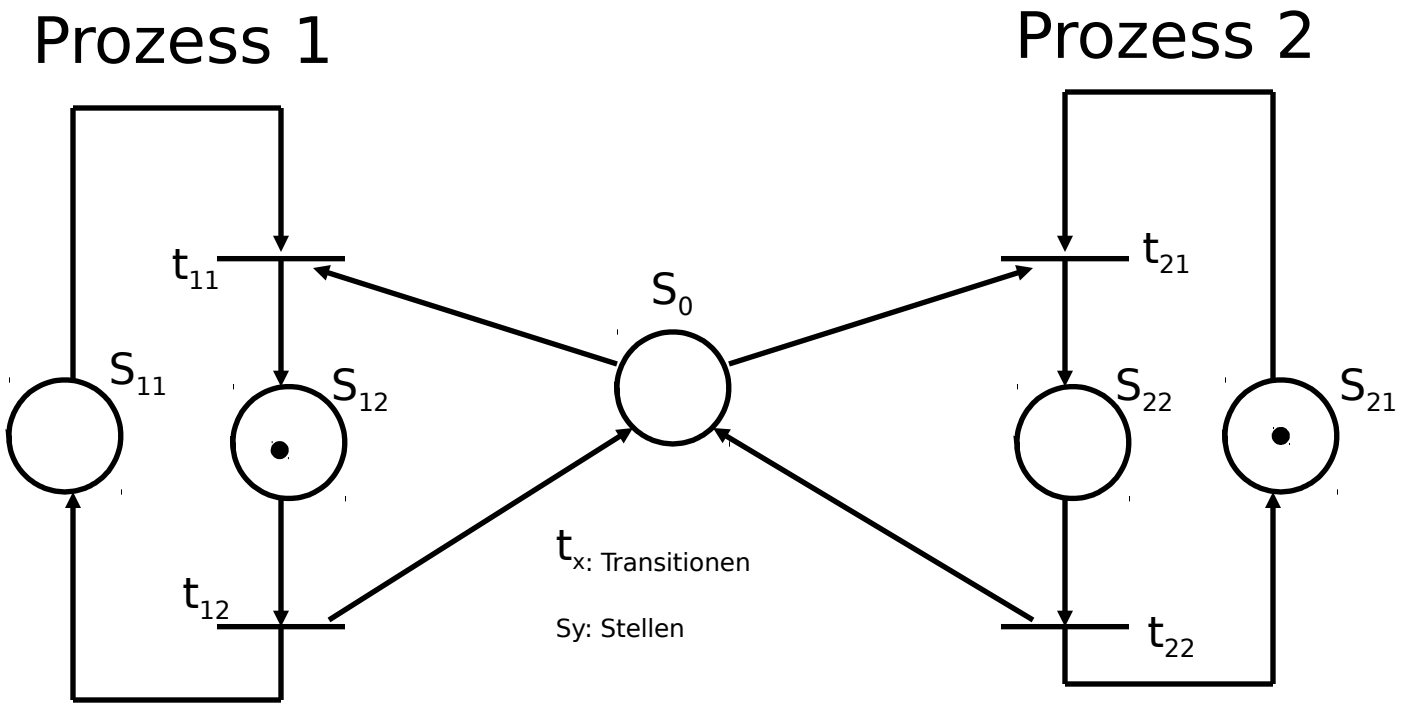


Kritischer Abschnitt als Petrinetz (1)



- S_{12} und S_{22} stellen die kritischen Abschnitte zweier Prozesse dar, S_{11} und S_{21} sind unkritisch
- Über die Stelle S_0 wird der gegenseitige Ausschluss realisiert (S_0 enthält maximal eine Marke)

Kritischer Abschnitt als Petrinetz (2)



- t_{11} schaltet hier und Prozess 1 ist im kritischen Abschnitt

Überblick

1. Einführung
2. Kritische Abschnitte und gegenseitiger Ausschluss
3. **Sperren, Semaphore und Mutex**
4. Diverse Synchronisationsprobleme

Sperren: Implementierungsvarianten (1)

- Eine einfache Lösung zur Realisierung von kritischen Abschnitten ist **busy waiting** (aktives Warten)
  - Ein Prozess testet eine sog. Synchronisationsvariable
  - Test solange, bis Variable einen Wert hat, der den Zutritt erlaubt → Man braucht einen speziellen Befehl dazu, oder via Token Passing
- Dieses **Polling** ist oft unwirtschaftlich, da es eine Verschwendung der CPU-Zeit bedeutet
  - Aber: **Spinlocks** in Betriebssystemen sind oft anzutreffen
    - Bei sehr kurzen Wartezeiten bei Multiprozessoren sehr gut!
    - Warum nicht bei Singleprozessoren?
- Manchmal ist es besser, einen Prozess „schlafen“ zu legen und erst wieder zu wecken, wenn er in den kritischen Bereich darf

Sperren, Implementierungsvarianten (2)

- 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)

- Hardware-Unterstützung zur Synchronisation:
  - **Atomare Instruktionsfolge** über nicht unterbrechbare Maschinenbefehle in einem einzigen ununterbrechbaren 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
    - **Swap**
      - Austausch zweier Variablenwerte in einem Speicherzyklus
    - **Fetch And Add**
      - Lesen und Inkrementieren einer Speicherzelle in einem Speicherzyklus
    - **Exchange-Befehl XCHG dest, src (im Intel-Befehlssatz)**
      - Inhalte von src und dest werden ausgetauscht (Register und Speicherbereiche als Quelle und Ziel möglich)

Sperren, Beispiel: TSL-Befehl

- Einfache Lock-Implementierung mit TSL-Befehl (auch: TAS-Befehl)
- LOCK ist eine Speichervariable, die vom TSL-Befehl in einem Speicherzyklus gesetzt und ausgelesen wird

```
...
MyLock_lock:
    TSL R1, LOCK        // Lies LOCK in R1 ein und
                        // setze Wert von LOCK auf 1
    CMP R1, #0          // Vergleiche Registerinhalt mit 0
                        // Wenn Vergleich zutrifft, dann ist Lock gesetzt
    JNE MyLock_lock     // Ansonsten erneut versuchen
    RET                 // Kritischer Abschnitt kann betreten werden

MyLock_unlock:
    MOVE LOCK, #0       // LOCK auf 0 setzen (freigeben)
    RET                 // Kritischer Abschnitt kann von anderem Prozess
                        // betreten werden
```

Nach Tanenbaum (2009)

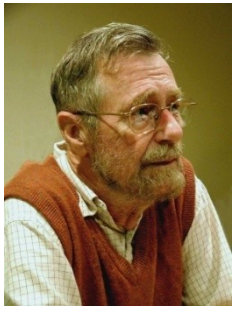
Sperren, Beispiel: Lock über XCHG-Befehl

- Pseudocode mit Intel-80386-Maschinenbefehlen

```
void akquireLock (var boolean lock)
{
    CODE {SYSTEM.i386}
        MOV EBX, lock[EBP]      ; EBX := ADR(lock)
        MOV AL, 1               ; AL := 1
    test:
        XCHG [EBX], AL          ; Setze und lese Lockvariable atomar
        CMP AL, 0               ; war lock frei?
        JE exit                 ; ja
        NOP                     ; nein, erneut versuchen
        JMP test
    exit:
}

void releaseLock (var boolean lock)
{
    lock := FALSE;
}
```

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



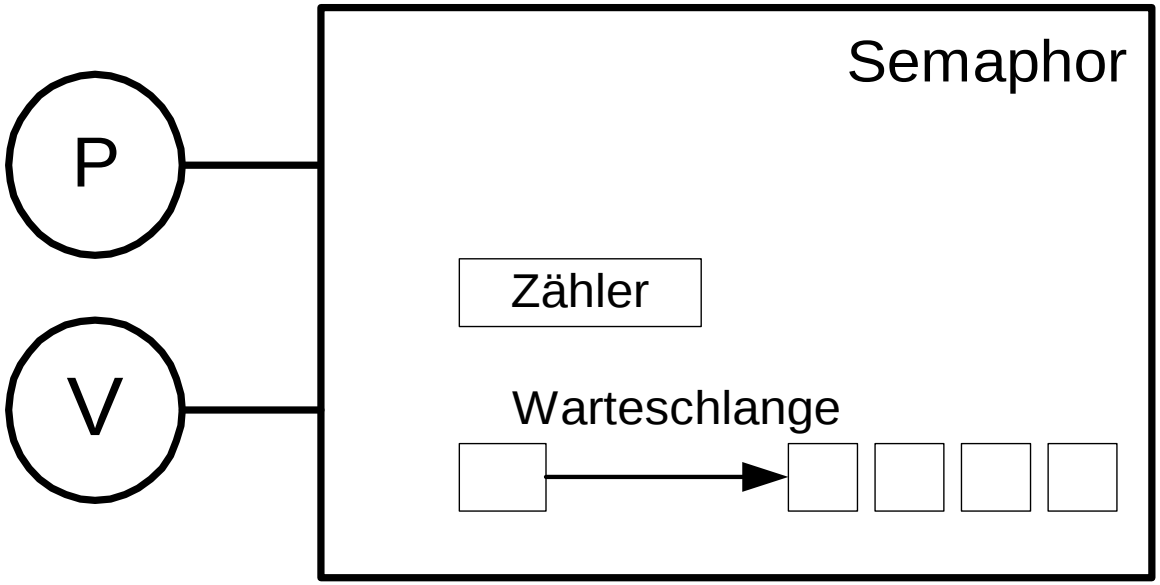
Edsger Wybe Dijkstra
(11.5.1930 – 06.08.2002)
Niederländischer
Computer-Wissenschaftler

Eisenbahnsignale
Quelle: wikipedia.de

Semaphore

- 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, Beispielnutzung

```
...
P();                    // kritischer Abschnitt besetzt?

c=counter.read();       // kritischer Abschnitt
c++;
counter.write(c);

V();                    // Verlassen des kritischen Abschnitts,
                        // Aufwecken eines wartenden Prozesses
```

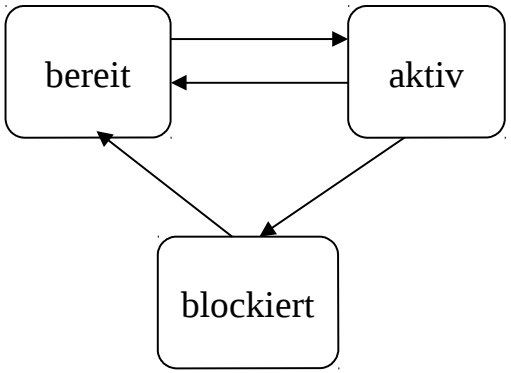
- P() und V() sind selbst wieder ununterbrechbar, also **atomare Aktionen**
- Atomare Aktionen werden **ganz** oder **gar nicht** ausgeführt

Semaphore, Algorithmus

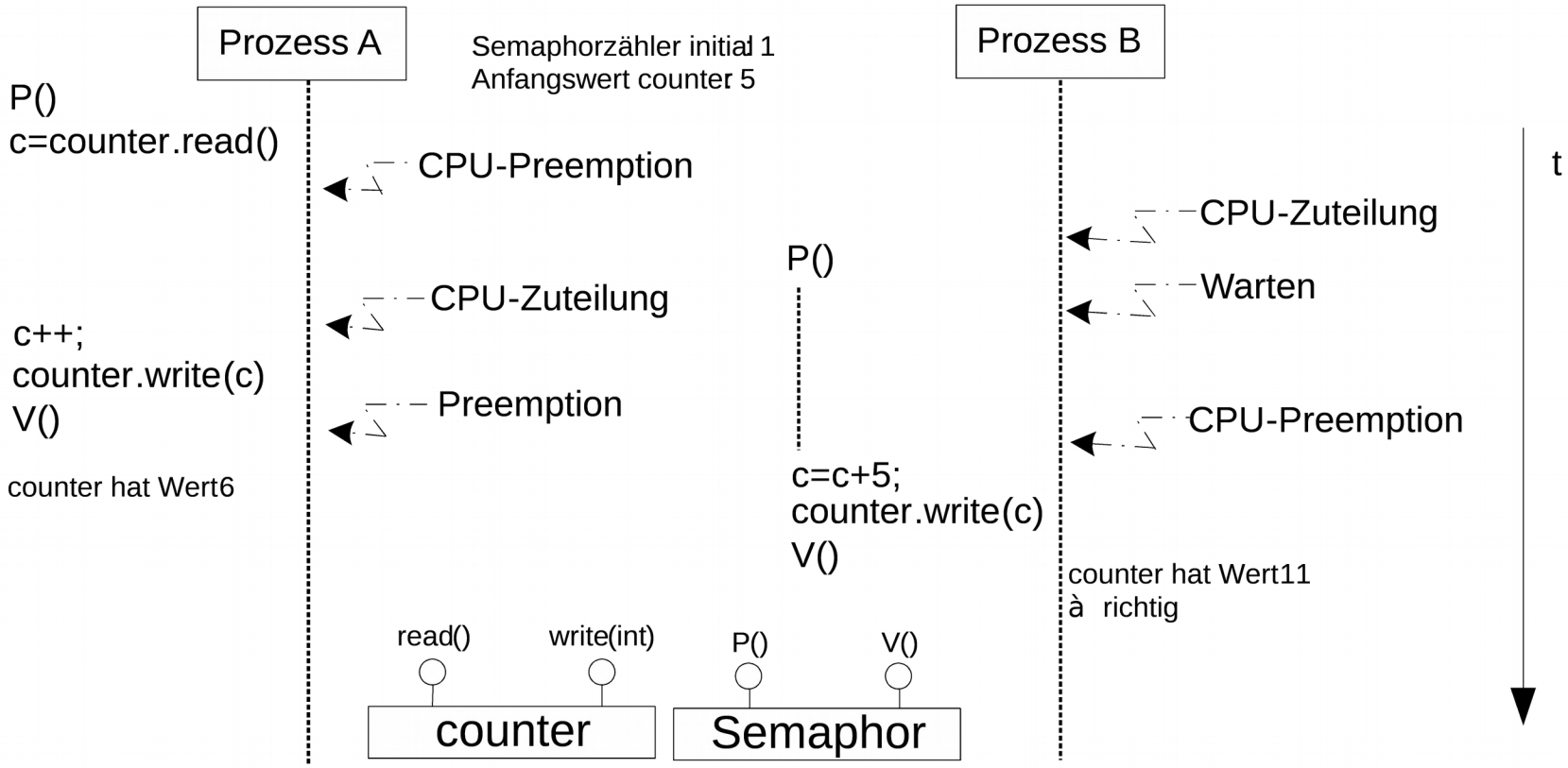
- s ist der Semaphor-Zähler, Init: s >= 1

```
void P() {
    if (s >= 1) {
        s = s – 1;      // der die P-Operation ausführende Prozess setzt
                        // seinen Ablauf fort
    } else {
        // der die P-Operation ausführende Prozess wird in seinem Ablauf
        // zunächst gestoppt, in den Wartezustand versetzt und in einer dem
        // Semaphor S zugeordneten Warteliste eingetragen
    }
}
```

```
void V() {
    s = s + 1;
    if (Warteliste ist nicht leer) {
        // aus der Warteliste wird ein Prozess ausgewählt
        // und aufgeweckt
    }
}
// der die V-Operation ausführende Prozess setzt seinen Ablauf fort
```



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

Überblick

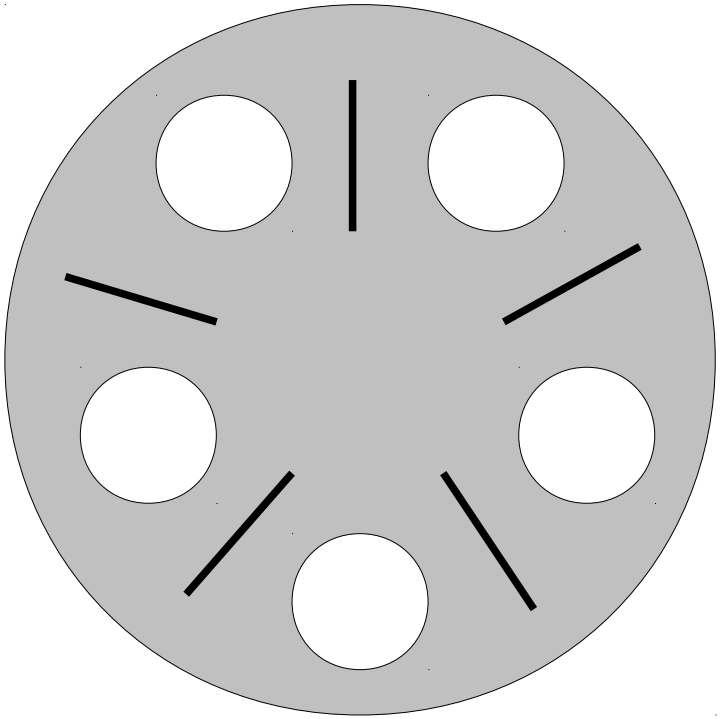
1. Einführung
2. Kritische Abschnitte und gegenseitiger Ausschluss
3. Semaphore und Mutex
4. **Diverse Synchronisationsprobleme**

Philosophenproblem

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

Sir Charles Antony Richard Hoare (11.01.1934)
Britischer Computerwissenschaftler

Prozessverwaltung: Philosophenproblem



Lösungsalgorithmus:

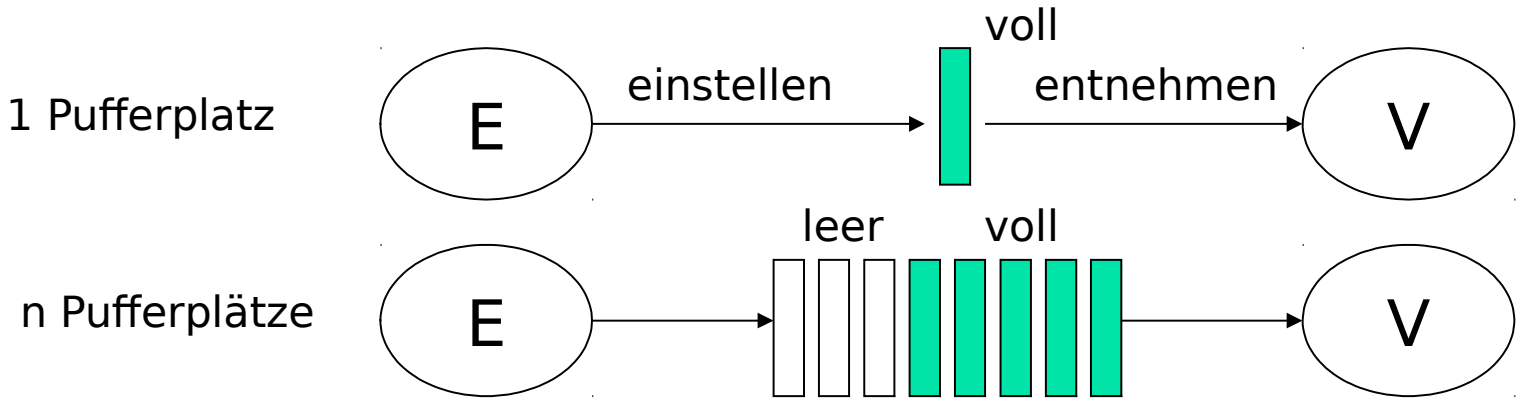
```
...
static int n = 5;

void philosopher(int i) {
    while (true) {
        think();
        take_stick(i);
        take_stick((i+1) % n);
        eat();
        put_stick(i);
        put_stick((i+1) % n);
    }
}
```

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 was drinnen ist



Erzeuger-Verbraucher-Problem

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

Erzeuger

```
while (true) {
    produce(item);
    P(frei);
    P(mutex);
    putInBuffer(item);
    V(mutex);
    V(belegt);
}
```

Verbraucher

```
while (true) {
    P(belegt);
    P(mutex);
    getFromBuffer(item);
    V(mutex);
    V(frei);
    consume(item);
}
```

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 was tun

Nach Tanenbaum (2009)

Überblick und Zusammenfassung

- ✓ Einführung
- ✓ Kritische Abschnitte und gegenseitiger Ausschluss
- ✓ Sperren, Semaphore und Mutex
- ✓ Diverse Synchronisationsprobleme