

# Systeme I: Betriebssysteme

## **Kapitel 5** **Nebenläufigkeit und** **wechselseitiger Ausschluss**

Maren Bennewitz

---



# Inhalt Vorlesung

- Aufbau einfacher Rechner
- Überblick: Aufgabe, Historische Entwicklung, unterschiedliche Arten von Betriebssystemen
- Verschiedene Komponenten / Konzepte von Betriebssystemen
  - Dateisysteme
  - Prozesse
  - Nebenläufigkeit und wechselseitiger Ausschluss
  - Deadlocks
  - Scheduling
  - Speicherverwaltung

# Einführung

- Größere Softwaresysteme sind häufig realisiert als eine **Menge von nebenläufigen Prozessen**
- Nebenläufigkeit = „potentieller Parallelismus“
- Nebenläufige Prozesse können **parallel** auf mehreren Prozessorkernen ausgeführt werden
- Sie können aber auch „**pseudo-parallel**“ auf einem Prozessor mit nur einem Kern ausgeführt werden

# Nebenläufigkeit

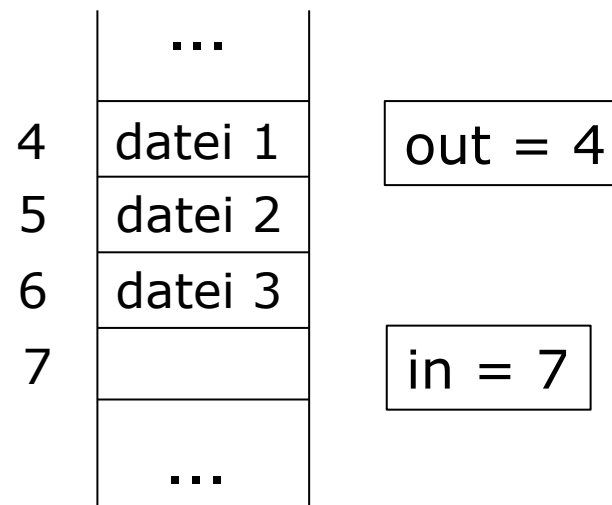
- Das Betriebssystem muss die Ressourcen der aktiven Prozesse verwalten
- Bei Zugriff auf gemeinsame Ressourcen muss **wechselseitiger Ausschluss** garantiert werden
- Die Korrektheit des Ergebnisses muss **unabhängig von der Ausführungsgeschwindigkeit** sein

# Beispiel: Druckerpuffer

- Prozess möchte Datei drucken: Trägt Dateinamen in Druckwarteschlange ein
- Drucker-Daemon überprüft zyklisch, ob es Dateien zu drucken gibt
- Druckerpuffer: Nummerierte Liste von Dateinamen
- Zwei Variablen zugänglich für alle Prozesse:
  - *out*: Nummer der nächsten zu druckenden Datei
  - *in*: Nummer des nächsten freien Eintrags

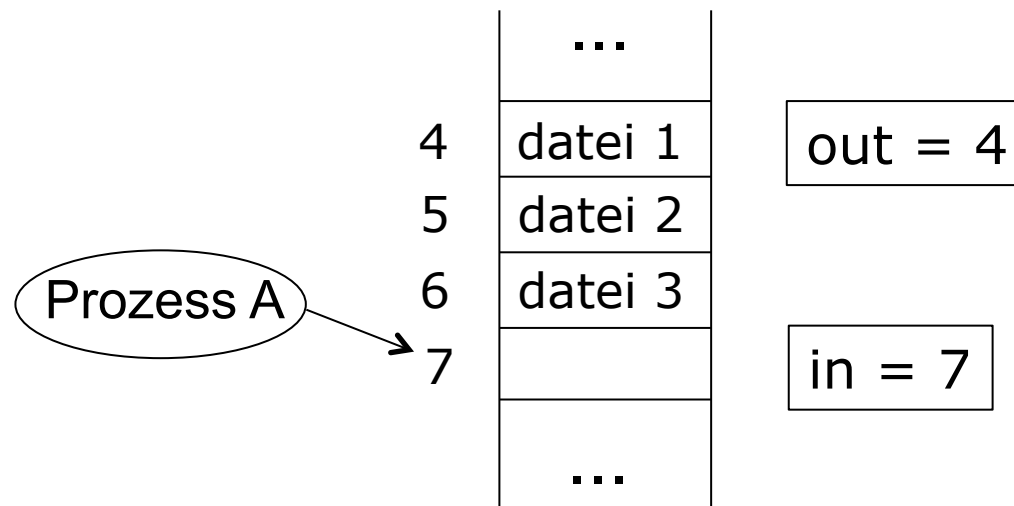
# Beispiel: Druckerpuffer

- Einträge 0 bis 3 leer (bereits gedruckt)
- Einträge 4-6 belegt (noch zu drucken)
- Prozesse A und B entscheiden „gleichzeitig“ eine Datei zu drucken



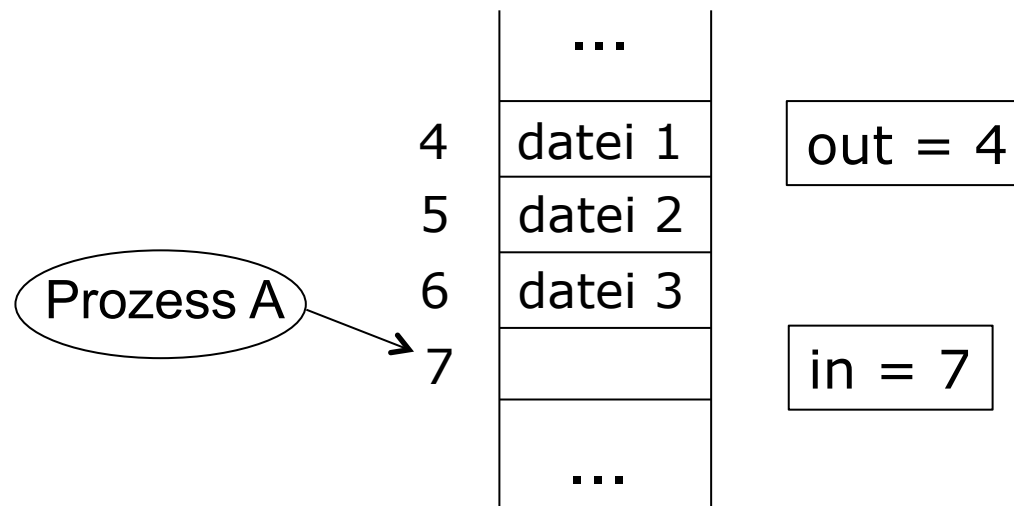
# Beispiel: Druckerpuffer

- Prozess A liest *in* aus und speichert 7



# Beispiel: Druckerpuffer

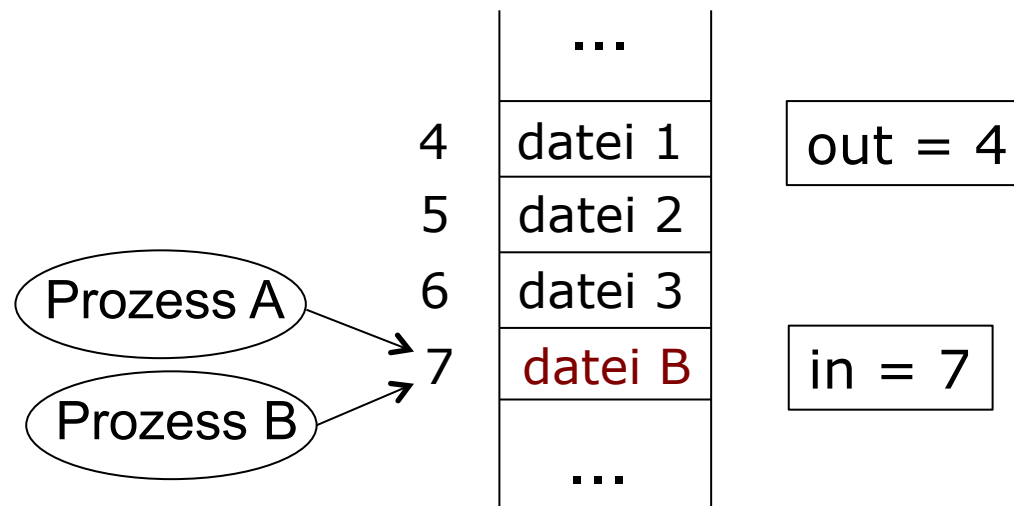
- Prozess A liest *in* aus und speichert 7
- Dann Prozesswechsel zu Prozess B





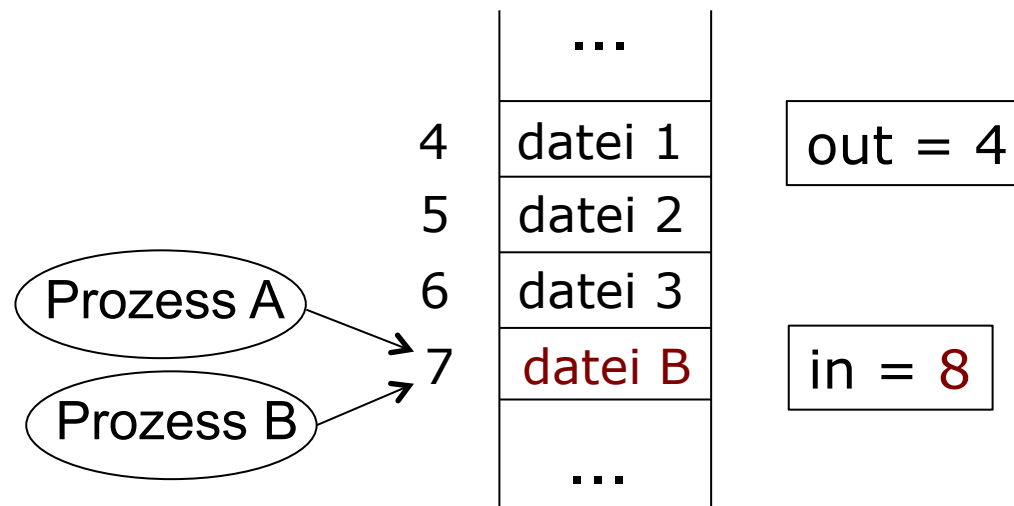
# Beispiel: Druckerpuffer

- Prozess A liest *in* aus und speichert 7
- Dann Prozesswechsel zu Prozess B
- Prozess B liest ebenfalls *in* aus mit Wert 7 und schreibt den Namen seiner Datei



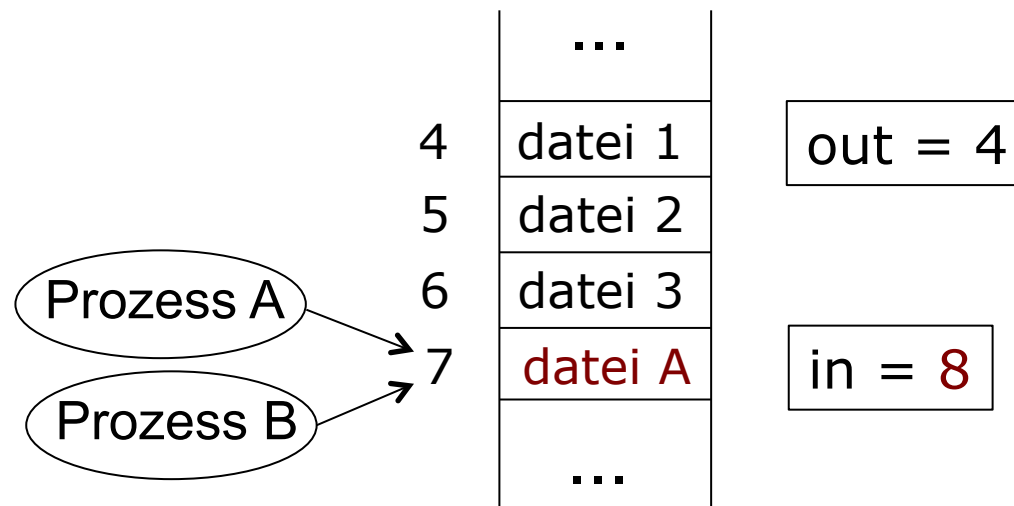
# Beispiel: Druckerpuffer

- Prozess A liest *in* aus und speichert 7
- Dann Prozesswechsel zu Prozess B
- Prozess B liest ebenfalls *in* aus mit Wert 7 und schreibt den Namen seiner Datei
- Prozess B aktualisiert *in* zu 8



# Beispiel: Druckerpuffer

- Schließlich: Prozess A läuft weiter und schreibt seinen Dateinamen in 7, weil er sich die Position als frei gemerkt hatte
- Dateiname von Prozess B wird überschrieben!



# Wettstreit um Ressourcen

- Wettstreit zwischen Prozessen („Race Condition“):
  - Zwei oder mehr Prozesse lesen oder beschreiben gemeinsamen Speicher
  - Endergebnis hängt davon ab, wer wann läuft
- Benötigt: Wechselseitiger Ausschluss von Prozessen („Mutual Exclusion“)
- Verbiete Prozessen „gleichzeitig“ mit einem anderen Prozess die **gemeinsam genutzten Daten zu lesen oder zu beschreiben**

# Kritische Regionen

- Teile des Programms, in denen auf gemeinsam genutzten Speicher zugegriffen wird
- Um Race Conditions zu vermeiden: Stelle sicher, dass **niemals zwei Prozesse gleichzeitig in ihren kritischen Regionen** sind
- Dies reicht jedoch nicht aus, um bei gemeinsam genutzten Daten einen **effizienten** Ablauf zu gewährleisten

# Anforderungen an Lösungen für das Problem der krit. Region

- Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein (wechselseitiger Ausschluss)
- Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der Rechenkerne gemacht werden
- Kein Prozess, der außerhalb seiner kritischen Regionen läuft, darf andere Prozesse blockieren
- Kein Prozess sollte ewig darauf warten müssen, in seine kritische Region einzutreten

# Wechselseitiger Ausschluss durch Interrupts

- Sorge dafür, dass ein Prozess in seiner kritischen Region nicht unterbrochen wird
- Schalte alle Interrupts nach Eintritt in die kritische Region aus und danach wieder an
- CPU wechselt nicht zu anderem Prozess
- Sperrung von Interrupts führt bei Mehrkernsystemen nicht zum Erfolg
- Mehrere Prozesse werden gleichzeitig ausgeführt, wechselseitiger Ausschluss kann nicht garantiert werden

# Lösungen für wechselseitigen Ausschluss

- **Software-Lösungen:** Verantwortlichkeit liegt bei Prozessen, Anwendungsprogramme sind gezwungen sich zu koordinieren
- **Hardware-Unterstützung:** Spezielle Maschinenbefehle, reduzieren Verwaltungsaufwand
- **Ins Betriebssystem integrierte Lösungen**



# Versuch 1a: Sperren von Variablen

- Situation: Prozesse konkurrieren um eine Ressource
- Prozesse können auf eine gemeinsame (Sperr-) Variable *turn* zugreifen
- *turn* ist mit 0 initialisiert
- Wenn ein Prozess in seine kritische Region eintreten möchte, fragt er die Sperre ab

# Versuch 1a: Sperren von Variablen

- Wenn Wert von *turn* 0 ist, setzt der Prozess sie auf 1 und betritt seine kritische Region
- Falls der Wert von *turn* 1 ist, wartet der Prozess, bis der Wert 0 wird
- **Problem:**
  - Prozess liest die Sperre aus und sieht Wert 0
  - Prozess wird unterbrochen, bevor er die Sperre auf 1 setzen kann
  - Zweiter Prozess startet, liest Wert 0, setzt Sperre auf 1, betritt kritische Region
  - Erster Prozess wieder aktiv, setzt Sperre auf 1
  - Beide Prozesse in ihren kritischen Regionen!

# Versuch 1b: Strikter Wechsel

- Ganzzahlige Variable *turn*, gibt an, **wer an der Reihe ist**, die kritische Region zu betreten
- Anfangs: Prozess 0 stellt fest, dass der Wert von *turn* 0 ist und betritt seine kritische Region
- Prozess 1 sitzt fest bis der Wert 1 wird

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

# Versuch 1b: Strikter Wechsel

- Möglicher Verlauf: Prozess 0 verlässt kritische Region, setzt *turn* auf 1, macht mit nichtkritischer Region weiter

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

# Versuch 1b: Strikter Wechsel

- Prozess 1 ist es nun erlaubt, die kritische Region zu betreten, führt diese aus

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

# Versuch 1b: Strikter Wechsel

- Prozess 1 setzt nach Beendigung der kritischen Region *turn* auf 0
- Beide Prozesse befinden sich nun in ihren nichtkritischen Regionen

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

# Versuch 1b: Strikter Wechsel

- *turn* ist 0, Prozess 0 fragt Wert ab und führt seine kritische Region aus

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

# Versuch 1b: Strikter Wechsel

- Prozess 0 verlässt seine kritische Region (schnell) und setzt *turn* auf 1

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```



# Versuch 1b: Strikter Wechsel

- *turn* ist 1 und beide Prozesse sind in ihren nichtkritischen Regionen

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

# Versuch 1b: Strikter Wechsel

- Prozess 0 will in seine kritische Region wieder eintreten, aber
- *turn* ist 1 und Prozess 1 ist noch mit seiner nichtkritischen Region beschäftigt

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

# Versuch 1b: Strikter Wechsel

- Prozess 0 hängt (ohne Grund) in der solange-Schleife, bis Prozess 1 den Wert von *turn* auf 0 setzt! Strenges Abwechseln keine gute Idee!
- Prozess 0 wird von einem Prozess blockiert, ohne dass dieser in seiner kritischen Region ist (verletzt 3. Anforderung, Folie 14)

```
/* Prozess 0 */  
wiederhole  
{  
  solange (turn ≠ 0)  
    tue nichts;  
  /* kritische Region */  
  turn := 1;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nichtkritische Region */  
}
```

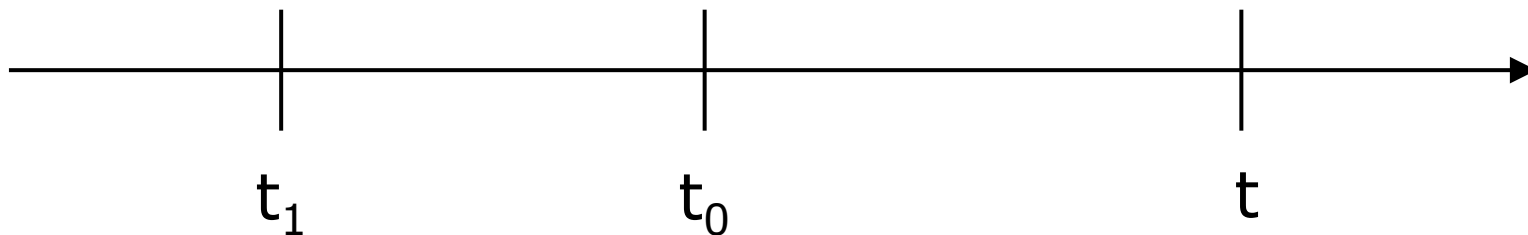
# Wechselseitiger Ausschluss

## Satz:

Dieses Vorgehen **garantiert wechselseitigen Ausschluss**, falls in den kritischen und nichtkritischen Regionen **keine zusätzlichen Zuweisungen an *turn*** erfolgen

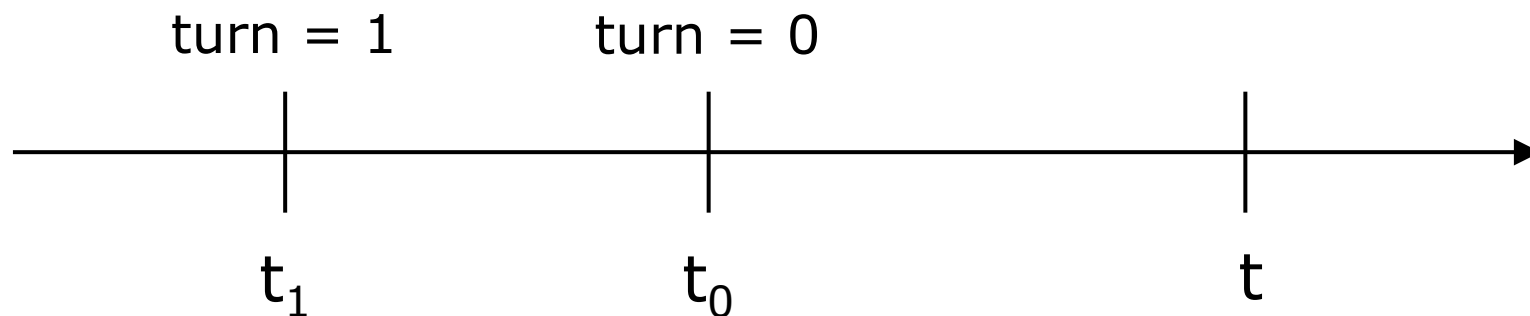
# Beweis durch Widerspruch

- Annahme: Es gibt einen Zeitpunkt  $t$ , zu dem beide Prozesse in den kritischen Regionen sind
- $t_1$ : Der letzte Zeitpunkt  $< t$ , zu dem **Prozess 1** die solange-Schleife verlassen hat und in seine **kritische Region** gegangen ist
- $t_0$ : Der letzte Zeitpunkt  $< t$ , zu dem **Prozess 0** die solange-Schleife verlassen hat und in seine **kritische Region** gegangen ist, o.B.d.A.  $t_1 < t_0$



# Beweis durch Widerspruch

- Zwischen  $t_1$  und  $t_0$  muss  $turn = 0$  ausgeführt worden sein
- $turn = 0$  kommt nur bei der Initialisierung vor und beim Setzen in Prozess 1 **nach** krit. Region
- Prozess 1 ist aber zwischen  $t_1$  und  $t$  permanent in seiner kritische Region
- $turn = 0$  kann nicht zwischen  $t_1$  und  $t$  ausgeführt worden sein!



# Beweis durch Widerspruch

- Zwischen  $t_1$  und  $t_0$  muss  $turn = 0$  ausgeführt worden sein
- $turn = 0$  kommt nur bei der Initialisierung vor und beim Setzen in Prozess 1 **nach** krit. Region
- Prozess 1 ist aber zwischen  $t_1$  und  $t$  permanent in seiner kritische Region
- $turn = 0$  kann nicht zwischen  $t_1$  und  $t$  ausgeführt worden sein!
- Prozess 0 kann nicht in kritischer Region sein
- Widerspruch zur Annahme!
- Es gibt also keinen Zeitpunkt  $t$ , zu dem beide Prozesse in ihren kritischen Regionen sind

# Versuch 1b: Analyse

- **Vorteil:** Wechselseitiger Ausschluss ist garantiert
- **Nachteile:**
  - Nur abwechselnder Zugriff auf kritische Region kann zu starker Verzögerung führen
  - Beispiel: Prozess 0 ist schnell, Prozess 1 hat einen sehr langen nichtkritischen Abschnitt
  - Aktives Warten („Busy Waiting“) führt zu Verschwendung von Rechenzeit



# Nächster Versuch

- Vorher wurde der Name des Prozesses der an der Reihe ist, gespeichert
- Besser: Speichere Zustandsinformation der Prozesse (in kritischer Region: ja/nein)
- Dadurch: Wenn ein Prozess langsamer ist, kann der andere immer noch in seine kritische Region eintreten (sofern der andere nicht in seiner ist)

## Versuch 2

- Benutze **zwei** gemeinsame Variablen zur Kommunikation: zwei boolesche Werte
- Prozess 0 schreibt in `flag[0]`, liest beide
- Prozess 1 schreibt in `flag[1]`, liest beide
- Bedeutung von `flag[i] = true`:  
Prozess  $i$  möchte in seine kritische Region eintreten / ist schon drin
- Beide Variablen sind anfangs *false*

# Versuch 2

```
/* Prozess 0 */  
wiederhole  
{  
  solange (flag[1] = true)  
    tue nichts;  
  flag[0] := true;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (flag[0] = true)  
    tue nichts;  
  flag[1] := true;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 2

- Garantiert das wechselseitigen Ausschluss?

```
/* Prozess 0 */  
wiederhole  
{  
  solange (flag[1] = true)  
    tue nichts;  
  flag[0] := true;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (flag[0] = true)  
    tue nichts;  
  flag[1] := true;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 2

- Beide Variablen sind anfangs *false*
- Prozess 0 schließt Schleife ab, muss dann die CPU abgeben

```
/* Prozess 0 */  
wiederhole  
{  
  solange (flag[1] = true)  
    tue nichts;  
→ flag[0] := true;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (flag[0] = true)  
    tue nichts;  
  flag[1] := true;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 2

- Prozess 1 schließt die Schleife ab

```
/* Prozess 0 */  
wiederhole  
{  
  solange (flag[1] = true)  
    tue nichts;  
→ flag[0] := true;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (flag[0] = true)  
    tue nichts;  
  flag[1] := true;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 2

- Prozess 1 setzt flag[1], betritt seine kritische Region, muss dann die CPU abgeben

```
/* Prozess 0 */  
wiederhole  
{  
  solange (flag[1] = true)  
    tue nichts;  
→ flag[0] := true;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (flag[0] = true)  
    tue nichts;  
  flag[1] := true;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 2

- Prozess 0 setzt flag[0] betritt seine kritische Region
- Beide Prozesse können ungehindert „gleichzeitig“ ihre kritischen Regionen betreten

```
/* Prozess 0 */  
wiederhole  
{  
  solange (flag[1] = true)  
    tue nichts;  
  flag[0] := true;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (flag[0] = true)  
    tue nichts;  
  flag[1] := true;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```



# Versuch 2: Analyse

- **Vorteil:** Auch nicht-alternierender Zugriff auf kritische Region ist erlaubt
- **Nachteile:**
  - Wieder aktives Warten
  - Wechselseitiger Ausschluss ist nicht garantiert!

# Versuch 3

- Bei Versuch 2 kam der Wunsch, die kritische Region zu betreten zu spät
- Ziehe also den Wunsch, die kritischen Region zu betreten, vor

```
/* Prozess 0 */  
wiederhole  
{  
  flag[0] := true;  
  solange (flag[1] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  solange (flag[0] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 3

- Führt das zu Erfolg?

```
/* Prozess 0 */
wiederhole
{
  flag[0] := true;
  solange (flag[1] = true)
    tue nichts;
  /* kritische Region */
  flag[0] := false;
  /* nichtkritische Region */
}
```

```
/* Prozess 1 */
wiederhole
{
  flag[1] := true;
  solange (flag[0] = true)
    tue nichts;
  /* kritische Region */
  flag[1] := false;
  /* nichtkritische Region */
}
```

# Versuch 3

- Führt das zu Erfolg?
- Ja, wechselseitiger Ausschluss ist garantiert (s. Übung)
- Aber...

```
/* Prozess 0 */  
wiederhole  
{  
  flag[0] := true;  
  solange (flag[1] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  solange (flag[0] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 3

- Beide Variablen sind anfangs *false*
- Prozess 0 setzt `flag[0] := true` und muss die CPU abgeben

```
/* Prozess 0 */  
wiederhole  
{  
→ flag[0] := true;  
  solange (flag[1] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  solange (flag[0] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 3

- Prozess 1 setzt `flag[1] := true`
- Prozess 1 hängt in Schleife fest, muss CPU abgeben

```
/* Prozess 0 */  
wiederhole  
{  
→ flag[0] := true;  
  solange (flag[1] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  solange (flag[0] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 3

- Auch Prozess 0 hängt in Schleife fest
- Jetzt werden beide Prozesse ihre Schleife nie verlassen!
- Verklemmung („Deadlock“)

```
/* Prozess 0 */  
wiederhole  
{  
  flag[0] := true;  
  solange (flag[1] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  solange (flag[0] = true)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 3: Analyse

- **Vorteile:**
  - Auch nicht-alternierender Zugriff auf kritische Region ist erlaubt
  - Wechselseitiger Ausschluss ist garantiert
- **Nachteile:**
  - Wieder aktives Warten
  - Deadlock kann auftreten!



# Ergebnis

- Anforderung zu früh: Deadlock (Versuch 3)
- Anforderung zu spät: Kein wechselseitiger Ausschluss (Versuch 2)
- Füge hinzu: Möglichkeit, dem anderem Prozess Vorrang zu lassen

# Richtige Lösung: Petersons Algorithmus (1981)

- Statusbeobachtung reicht nicht aus
- Variable *turn* bestimmt, welcher Prozess auf seiner Anforderung bestehen darf
- Gemeinsame Variablen:  
turn, flag[0], flag[1]
- Initialisierung:  
flag[0] := *false*; flag[1] := *false*; *turn*: beliebig

# Peterson-Algorithmus

- *turn* löst Gleichzeitigkeitskonflikte
- Dem anderen Prozess wird die Möglichkeit gegeben, in die kritische Region einzutreten

```
/* Prozess 0 */  
wiederhole  
{  
  flag[0] := true;  
  turn := 1;  
  solange (flag[1] = true  
           und turn = 1)  
    tue nichts;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  turn := 0;  
  solange (flag[0] = true  
           und turn = 0)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Versuch 5: Peterson-Algorithmus

- Behauptung: Wechselseitiger Ausschluss ist garantiert
- Beweis durch Widerspruch
- Annahme: Es gibt einen Zeitpunkt  $t$ , zu dem beide Prozesse in ihren kritischen Regionen sind

# Beweis durch Widerspruch

- $t_1$ : Der letzte Zeitpunkt  $< t$ , zu dem Prozess 1 die solange-Schleife verlässt
- $t_1'$ : Der letzte Zeitpunkt  $< t$ , zu dem Prozess 0 die solange-Schleife verlässt
- $t_2$ : Der letzte Zeitpunkt  $< t$ , zu dem Prozess 1  $turn := 0$  ausführt
- $t_2'$ : Der letzte Zeitpunkt  $< t$ , zu dem Prozess 0  $turn := 1$  ausführt
- $t_3$ : Der letzte Zeitpunkt  $< t$ , zu dem Prozess 1  $flag[1] := true$  ausführt
- $t_3'$ : Der letzte Zeitpunkt  $< t$ , zu dem Prozess 0  $flag[0] := true$  ausführt

# Beweis durch Widerspruch

**/\* Prozess 0 \*/**

**wiederhole**

**{**

**flag[0] := true;** ←  $t_3'$

**turn := 1;** ←  $t_2'$

**solange (flag[1] = true  
und turn = 1)**

**tue nichts;**

**/\* kritische Region \*/** ←  $t_1'$

**flag[0] := false;**

**/\* nichtkritische Region \*/**

**}**

**/\* Prozess 1 \*/**

**wiederhole**

**{**

**flag[1] := true;** ←  $t_3$

**turn := 0;** ←  $t_2$

**solange (flag[0] = true  
und turn = 0)**

**tue nichts;**

**/\* kritische Region \*/** ←  $t_1$

**flag[1] := false;**

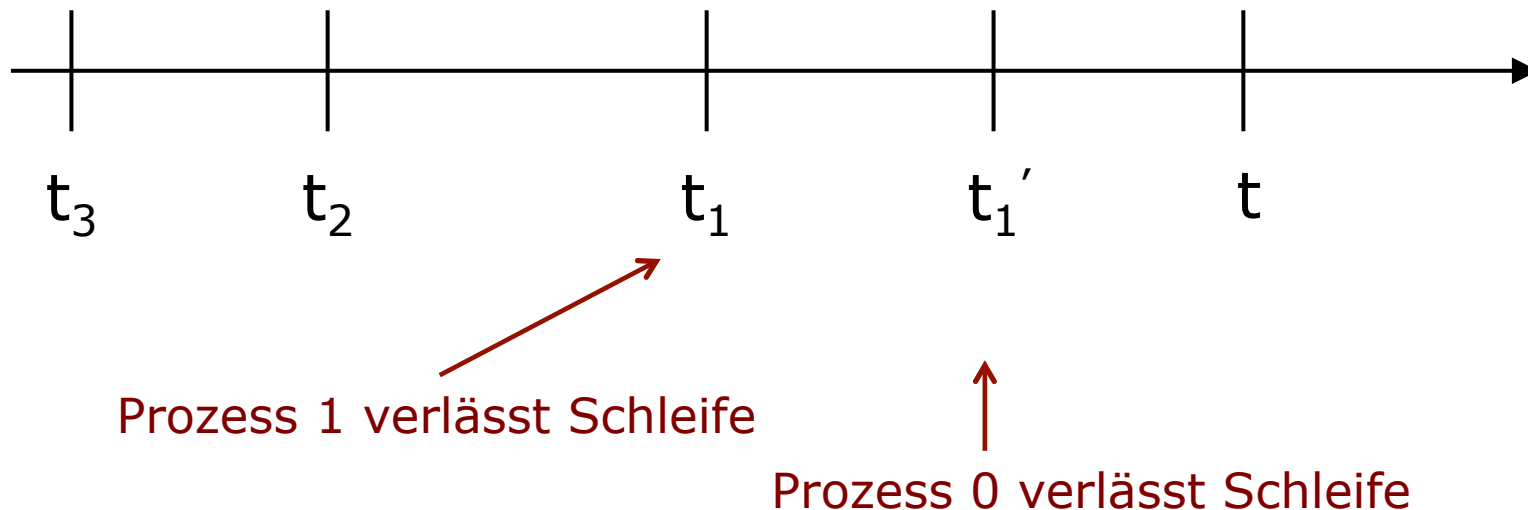
**/\* nichtkritische Region \*/**

**}**

$t$  →

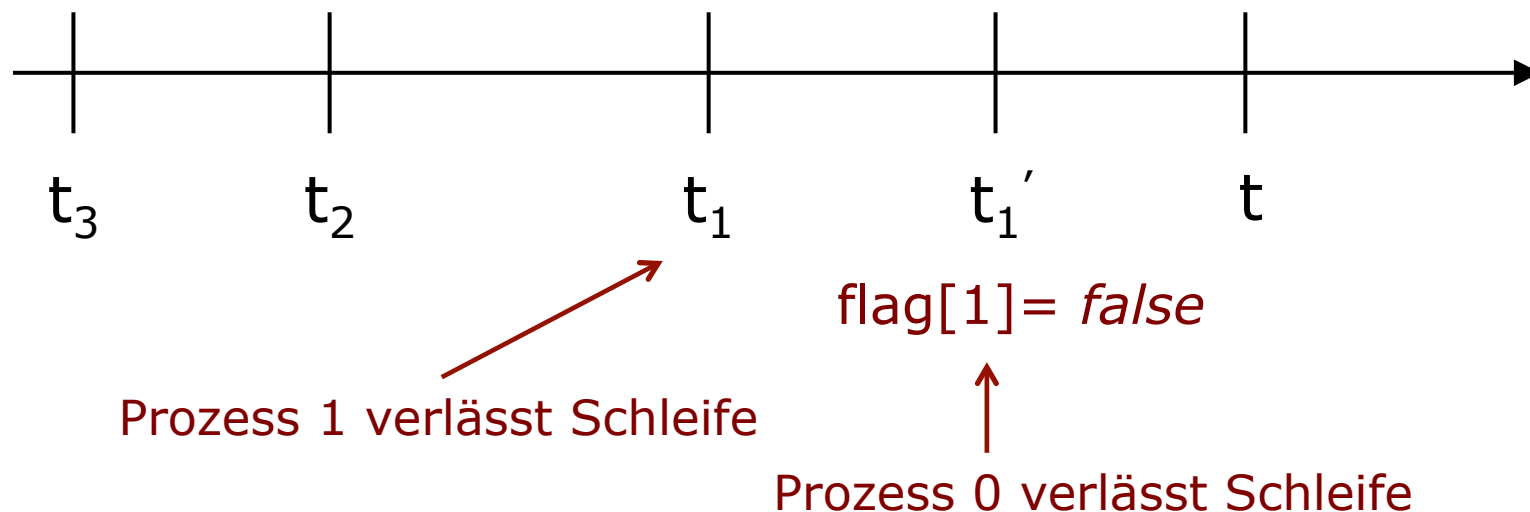
# Beweis durch Widerspruch

- O.B.d.A. geht Prozess 1 vor Prozess 0 in seinen kritischen Abschnitt, d.h.  $t_1 < t_1'$



# Beweis durch Widerspruch

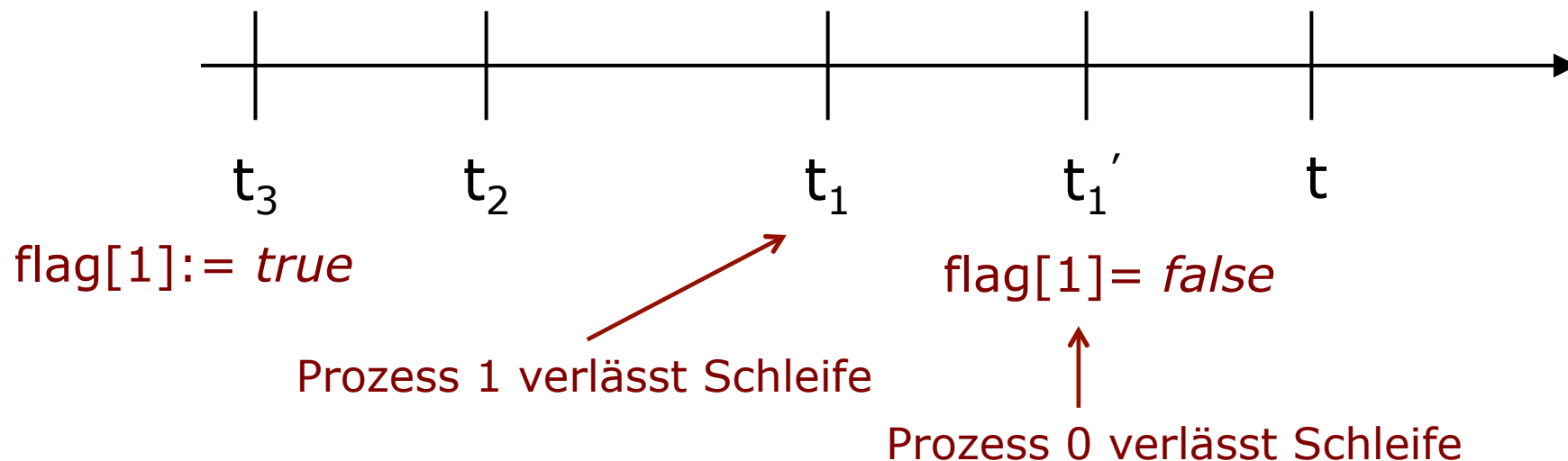
- O.B.d.A. geht Prozess 1 vor Prozess 0 in seinen kritischen Abschnitt, d.h.  $t_1 < t_1'$
- Fallunterscheidung nach dem Grund des Verlassens der solange-Schleife in Prozess 0 zur Zeit  $t_1'$
- **Fall 1:**  $\text{flag}[1] = \text{false}$  zum Zeitpunkt  $t_1'$





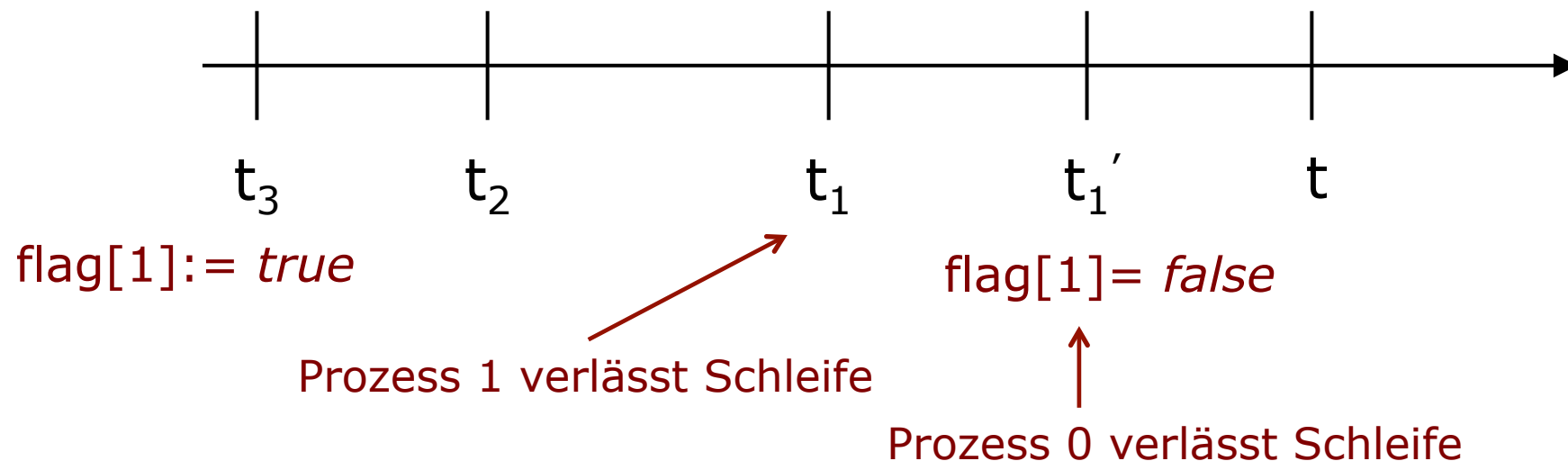
# Beweis durch Widerspruch

- O.B.d.A. geht Prozess 1 vor Prozess 0 in seinen kritischen Abschnitt, d.h.  $t_1 < t_1'$
- Fallunterscheidung nach dem Grund des Verlassens der solange-Schleife in Prozess 0 zur Zeit  $t_1'$
- **Fall 1:**  $\text{flag}[1] = \text{false}$  zum Zeitpunkt  $t_1'$
- Zum Zeitpunkt  $t_3$  wurde  $\text{flag}[1] := \text{true}$  gesetzt



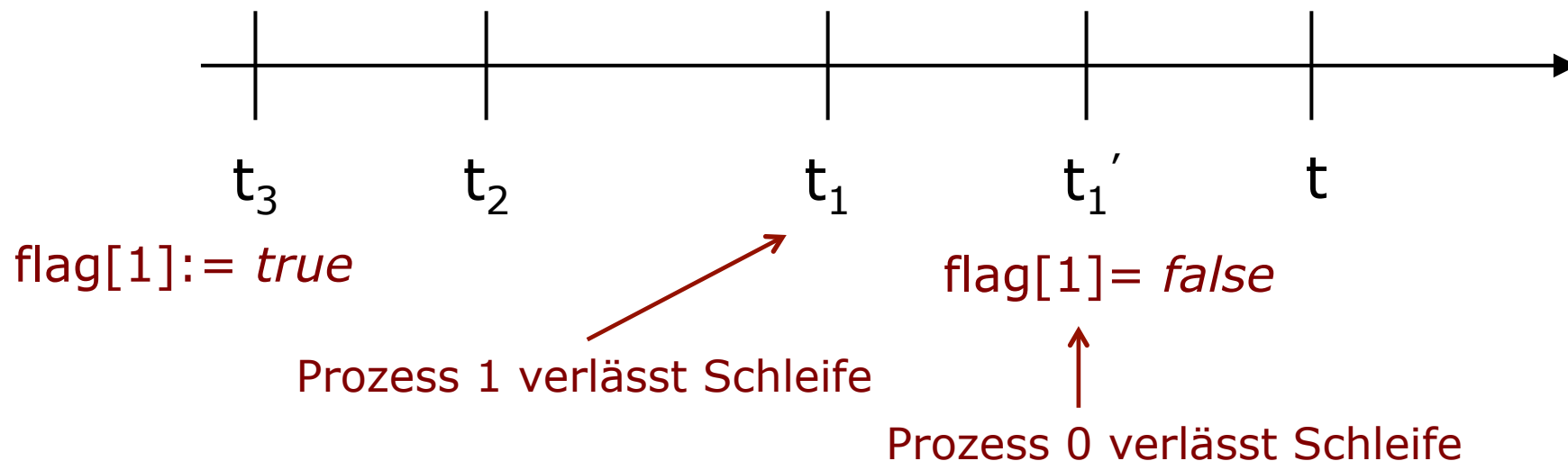
# Beweis durch Widerspruch

- Es müsste zwischen  $t_3$  und  $t_1'$   $\text{flag}[1] := \text{false}$  gesetzt worden sein



# Beweis durch Widerspruch

- Es müsste zwischen  $t_3$  und  $t_1'$   $\text{flag}[1] := \text{false}$  gesetzt worden sein
- $\text{flag}[1] := \text{false}$  passiert aber nur bei der Initialisierung und **nach** der kritischen Region von Prozess 1
- Zwischen  $t_3$  und  $t_1'$  erreicht Prozess 1 aber nicht das Ende der kritischen Region! (entspr. Annahme)



# Beweis durch Widerspruch

- Es müsste zwischen  $t_3$  und  $t_1'$  `flag[1] := false` gesetzt worden sein
- `flag[1] := false` passiert aber nur bei der Initialisierung und **nach** der kritischen Region von Prozess 1
- Zwischen  $t_3$  und  $t_1'$  erreicht Prozess 1 aber nicht das Ende der kritischen Region! (entspr. Annahme)
- Prozess 0 kann nicht die Schleife verlassen haben und in seiner kritischen Region sein
- Widerspruch zur Annahme!
- Es gibt also keinen Zeitpunkt  $t$ , zu dem beide Prozesse in ihren kritischen Regionen sind

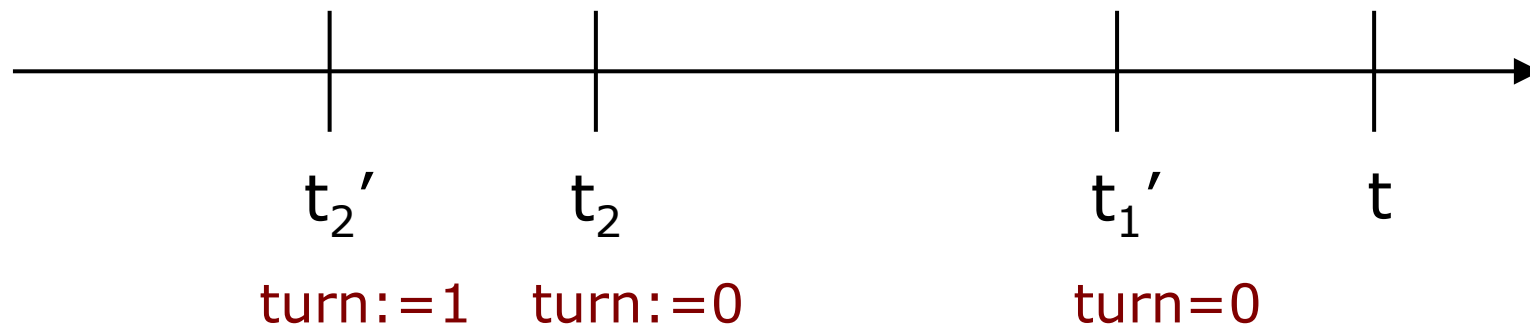
# Beweis durch Widerspruch

- **Fall 2:** Verlassen der Schleife von Prozess 0 durch  $turn=0$  zum Zeitpunkt  $t_1'$



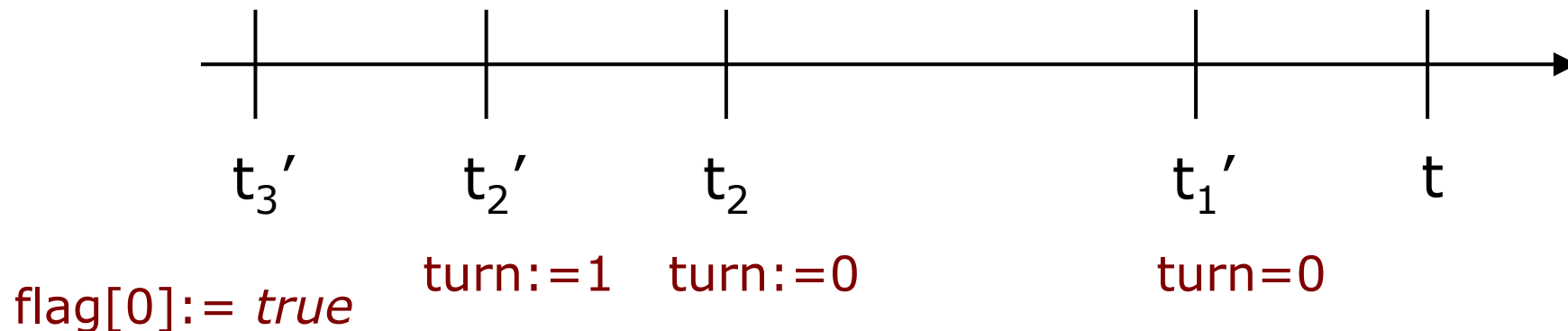
# Beweis durch Widerspruch

- **Fall 2:** Verlassen der Schleife von Prozess 0 durch  $turn=0$  zum Zeitpunkt  $t_1'$
- Zur Zeit  $t_2'$  wurde  $turn:=1$  gesetzt in Prozess 0
- Zur Zeit  $t_2$  wurde  $turn:=0$  gesetzt in Prozess 1
- Also muss gelten:  $t_2' < t_2 < t_1'$



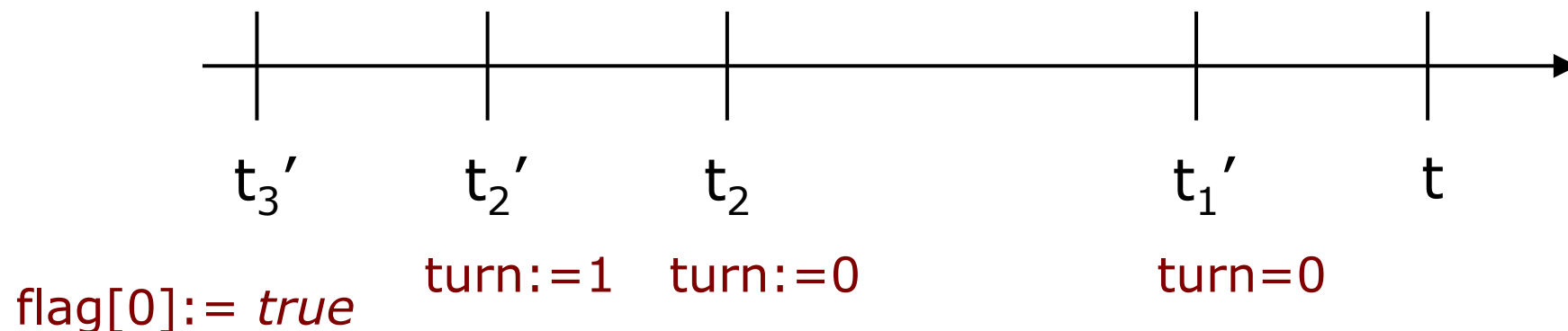
# Beweis durch Widerspruch

- Zum Zeitpunkt  $t_3'$  wurde  $\text{flag}[0] := \text{true}$  gesetzt



# Beweis durch Widerspruch

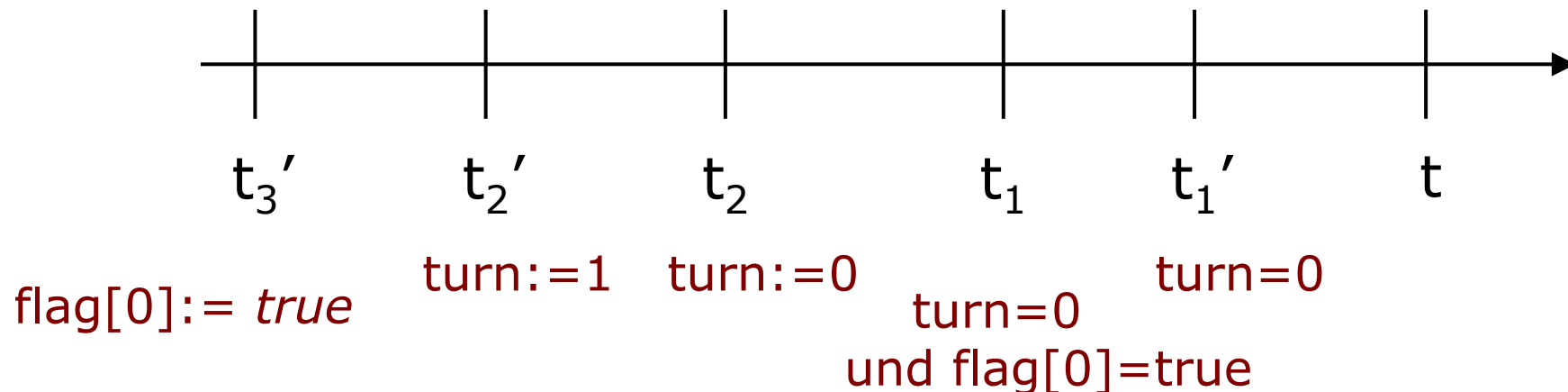
- Zum Zeitpunkt  $t_3'$  wurde  $\text{flag}[0] := \text{true}$  gesetzt
- Zwischen  $t_3'$  und  $t$  wird  $\text{flag}[0] := \text{false}$  nicht ausgeführt (entspr. Annahme), weil diese Anweisung nur bei der Initialisierung und **nach** der kritischen Region von Prozess 0 erfolgt





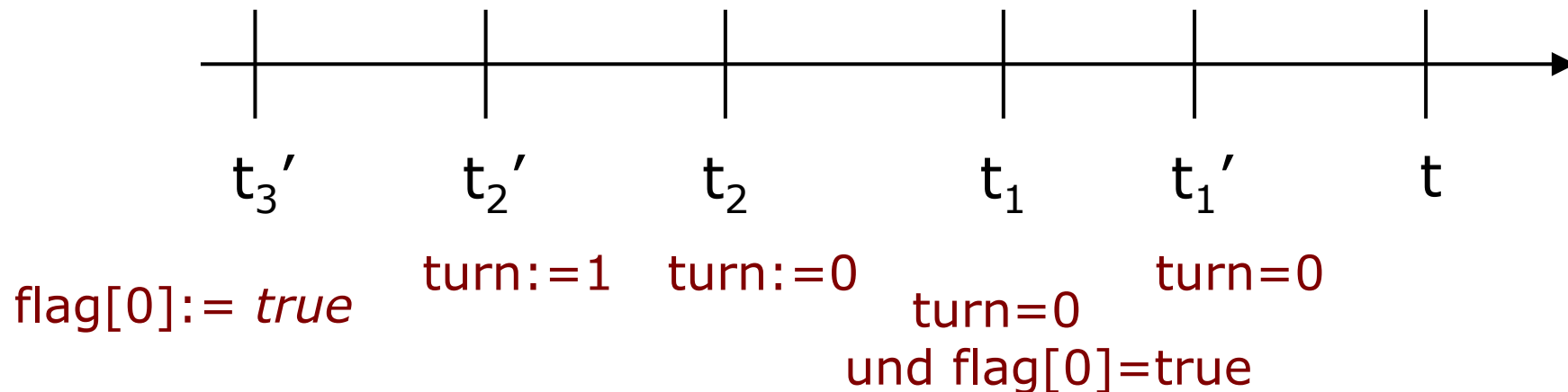
# Beweis durch Widerspruch

- Zum Zeitpunkt  $t_3'$  wurde  $\text{flag}[0] := \text{true}$  gesetzt
- Zwischen  $t_3'$  und  $t$  wird  $\text{flag}[0] := \text{false}$  nicht ausgeführt (entspr. Annahme), weil diese Anweisung nur bei der Initialisierung und **nach** der kritischen Region von Prozess 0 erfolgt
- Also muss zum Zeitpunkt  $t_1$  gelten:



# Beweis durch Widerspruch

- Zum Zeitpunkt  $t_1$  ist  $\text{flag}[0]=\text{true}$  und  $\text{turn}=0$ !
- Also: Prozess 1 kann zum Zeitpunkt  $t_1$  die Schleife nicht verlassen haben und in seiner kritischer Region sein
- Widerspruch zur Annahme!
- Es gibt also keinen Zeitpunkt  $t$ , zu dem beide Prozesse in ihren kritischen Regionen sind



# Peterson-Algorithmus: Keine gegenseitige Blockierung

- Annahme: Prozess 0 ist in Schleife blockiert
- Also:  $\text{flag}[1]=\text{true}$  und  $\text{turn}=1$

```
/* Prozess 0 */  
wiederhole  
{  
  flag[0] := true;  
  turn := 1;  
  solange (flag[1] = true  
           und turn = 1)  
    tue nichts;  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  turn := 0;  
  solange (flag[0] = true  
           und turn = 0)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Peterson-Algorithmus: Keine gegenseitige Blockierung

Betrachte drei Fälle

- Prozess 1 hat kein Interesse an kritischem Abschnitt: Nicht möglich, weil  $\text{flag}[1]=\text{true}$
- Prozess 1 wartet darauf, die kritische Region betreten zu können:  
Er wird nicht daran gehindert,  $\text{turn}$  ist 1
- Prozess 1 nutzt die kritische Region mehrfach ohne Rücksicht auf Prozess 0:  
Kann nicht passieren, da Prozess 1  $\text{turn}$  auf 0 setzt, bevor er die Region betritt und somit Prozess 0 die Möglichkeit des Zugriffs gibt

# Peterson-Algorithmus: Analyse

- **Vorteile:**
  - Auch nicht-alternierender Zugriff auf kritische Region ist erlaubt
  - Wechselseitiger Ausschluss ist garantiert
  - Deadlocks können nicht auftreten (bei Nichtberücksichtigung von Prioritäten und Ausfall von Prozessen)
- **Nachteil:** Wieder aktives Warten
- Es gibt eine Verallgemeinerung auf  $n$  Prozesse (wesentlich komplizierter!)

# Wechselseitiger Ausschluss in Software: Zusammenfassung

- Wechselseitiger Ausschluss ist in Software schwer zu realisieren
- Fehler durch kritische Wettläufe, subtile Fehler
- Software-Lösungen für wechselseitigen Ausschluss benötigen aktives Warten

# Zur Erinnerung:

## 2. Versuch in Software

- Warum scheiterte dieser Versuch?
- Weil Testen und Setzen von Flags nicht in einem einzigen Schritt durchführbar
- Prozesswechsel zwischen Testen und Setzen ist möglich: Wechselseitiger Ausschluss nicht garantiert

```
/* Prozess 0 */  
wiederhole  
{  
  solange (flag[1] = true)  
    tue nichts;  
→ flag[0] := true  
  /* kritische Region */  
  flag[0] := false;  
  /* nichtkritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  solange (flag[0] = true)  
    tue nichts;  
  flag[1] := true;  
  /* kritische Region */  
  flag[1] := false;  
  /* nichtkritische Region */  
}
```

# Wechselseitiger Ausschluss in Hardware (1)

- Neues Konzept:
  - **Atomare** Operationen
  - **Hardware** garantiert atomare Ausführung
- Testen und Setzen zusammen bilden eine atomare Operation:
  - Befehl **TSL** (**T**est and **S**et **L**ock)
  - Prozesswechsel kann nicht zwischen Testen und Setzen erfolgen



# Wechselseitiger Ausschluss in Hardware (2)

- Gemeinsame Sperrvariable LOCK für eine kritische Region
- Befehl in Maschinensprache: **TSL RX, LOCK** („Test and Set Lock“)
- Inhalt von LOCK wird in Register RX eingelesen und
- **Gleichzeitig** wird 1 an der Speicheradresse von LOCK abgelegt

# Wechselseitiger Ausschluss in Hardware (3)

Prozesse, die Zugriff auf die kritische Region erhalten wollen, führen folgende Befehle aus:

enter\_region:

```
TSL RX, LOCK      // Kopiere Lock-Inhalt und setze Lock
CMP RX, #0        // Hatte die Sperrvariable den Wert 0?
JNE enter_region  // Wenn nein, schon gesperrt, Schleife
...              // Fahre fort, betritt krit. Region
```

# Wechselseitiger Ausschluss in Hardware (4)

Alternative: Befehl XCHG (benutzen alle Prozessoren der Intel x86-Serie)

```
enter_region:
```

```
    MOVE RX, #1    // Speichere 1 im Register
```

```
    XCHG RX, LOCK // Tausche Inhalte von Register+Sperre
```

```
    CMP RX, #0     // Hatte die Sperrvariable den Wert 0?
```

```
    JNE enter_region // Wenn nein, schon gesperrt, Schleife
```

```
    ...           // Fahre fort, betritt krit. Region
```

# Wechselseitiger Ausschluss in Hardware (5)

Wenn ein Prozess seine kritische Region verlässt, setzt er die Sperrvariable wieder zurück auf 0:

```
MOVE LOCK, #0 // Speichere 0 in Sperrvariable
```

# Wechselseitiger Ausschluss in Hardware: Analyse

- **Vorteile:**
  - Bei beliebiger Anzahl von Prozessen und sowohl Ein- wie auch Mehrkernsystemen anwendbar
  - Kann für die Überprüfung mehrere kritischer Regionen eingesetzt werden: Jeweils eigene Sperrvariable
  - Wechselseitiger Ausschluss garantiert
  - Kein Deadlock (bei Nichtberücksichtigung von Prioritäten und Ausfall von Prozessen)
- **Nachteil:** Aktives Warten wie vorher

# Prioritäten

- Bei **Prioritäten** von Prozessen, können diese Lösungen trotzdem zu **Verklemmungen** führen (bei Hardwarelösung und Peterson-Algorithmus):
  - Prozess 0 betritt kritischen Abschnitt.
  - Prozess 0 wird unterbrochen, Prozess 1 hat höhere Priorität
  - Prozess 1 wird nun der Zugriff verweigert, aktive Warteschleife
  - Prozess 0 wird aber niemals zugeteilt, weil niedrigere Priorität
- Aktives Warten ist nicht nur ein Effizienzproblem!

# Ausfall von Prozessen in kritischer Region

- Es kann zu einer Verklemmung kommen, wenn Prozess nach Setzen des Flags ausfällt
- Andere Prozesse warten ewig

# Zwischenstand

- Sowohl Software- wie auch Hardwarelösungen weisen Nachteil des aktiven Wartens auf
- Besser: Integriere wechselseitigen Ausschluss ins Betriebssystem
- Prozesse **blockieren statt zu warten**



# Wechselseitiger Ausschluss, ins Betriebssystem integriert

- Systemaufruf **sleep**(id) zum Blockieren von Prozessen (id: eindeutige Bezeichnung für kritische Region)
- Pro kritischer Region gibt es eine **Warteschlange** von Prozessen, die auf Zutritt warten
- Systemaufruf **wakeup**(id) nach Verlassen des kritischen Abschnitts:  
Prozess weckt einen anderen Prozess auf, der auf die Erlaubnis wartet, die kritische Region mit Namen id zu betreten

# Mutex-Verfahren

- Zu einem Mutex  $m$  gehören:
  - binäre Lock-Variable  $\text{lock}_m$
  - Warteschlange  $\text{queue}_m$
  - eindeutige  $\text{id}_m$

# Mutex-Verfahren

- Vor Eintritt in die kritische Region: Aufruf von `mutex_lock(m)`
- Darin: Überprüfung, ob die kritische Region schon belegt ist
- Falls ja: Prozess blockiert (sleep) und wird in Warteschlange eingefügt
- Falls nein: Lock-Variable wird gesetzt und Prozess darf in kritische Region eintreten

# mutex\_lock

- Implementierung in Pseudocode
- `testset(lockm)` führt TSL-Befehl aus und liefert *false* genau dann, wenn die Lock-Variable `lockm` vorher 1 war

```
function testset(int wert)
{
    // ...
    else {
        wert:=1;
        return true;
    }
}
```

```
function mutex_lock(mutex m)
{
    solange (testset(lockm) = false)
        sleep(idm);
    return;
}
```

Nur zur Veranschaulichung;  
in Wirklichkeit TSL-Befehl

# Mutex: Warteschlange

- $\text{sleep}(id_m)$ : Prozess wird vom Betriebssystem in die Warteschlange des Mutex mit  $id_m$  eingefügt
- $queue_m$ : Warteschlange für alle Prozesse, die auf den Eintritt in kritische Region mit der Nummer  $id_m$  warten

# mutex\_unlock

- Nach Verlassen der kritischen Region wird `mutex_unlock(m)` aufgerufen
- Nach `wakeup(idm)` wird der erste Prozess in der Warteschlange bereit (aber nicht notwendigerweise aktiv)
- Implementierung in Pseudocode

```
function mutex_unlock(mutex m)
{
  lockm = 0;
  wakeup(idm);
  return;
}
```

# Initialisierung & Benutzung

- Zu einer kritischen Region vergibt der Programmierer einen Mutex mit einem Namen
- Zur Initialisierung des Mutex gibt es einen Betriebssystemaufruf, welcher die zugehörige lock-Variable reserviert
- Alle Aufrufe von `mutex_lock` und `mutex_unlock` für die entsprechende kritische Region benutzen intern die gleiche `id` als Parameter für `sleep` und `wakeup`
- Nächste Woche Übungsaufgabe

# Mutexe: Zusammenfassung

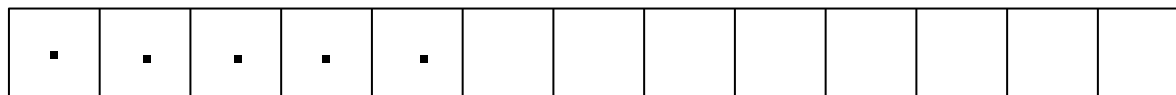
- Einfachste Möglichkeit um wechselseitigen Ausschluss mit Systemfunktionen zu garantieren
- Zwei Zustände: Mutex ist gesperrt / nicht gesperrt
- Kein aktives Warten von Prozessen, Prozesse sind blockiert
- Der Prozess, der den Mutex gesperrt hat, gibt ihn auch wieder frei



# Das Produzenten-Konsumenten-Problem (1)

Ein typisches Problem bei nebenläufiger Datenverarbeitung (z.B. Druckerpuffer)

- Gemeinsamer Puffer
- Erzeuger (Produzenten) schreiben in den Puffer: `insert_item(item)`
- Verbraucher (Konsumenten) lesen aus dem Puffer: `remove_item()`

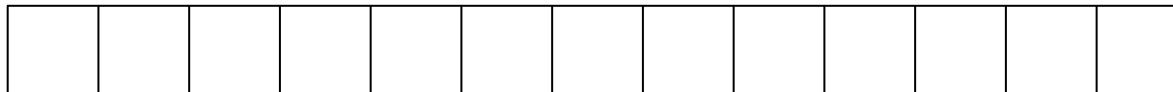


# Das Produzenten-Konsumenten-Problem (2)

- Die Puffergröße ist beschränkt und der Puffer kann leer sein
- Wenn der Puffer voll ist, dann sollen Erzeuger nichts einfügen
- Wenn der Puffer leer ist, sollen Verbraucher nichts entfernen
- Aus Effizienzgründen: Blockieren der Erzeuger/Verbraucher statt aktivem Warten

# Das Produzenten-Konsumenten-Problem – Eine Idee

- Gemeinsame Variable count für die Anzahl der Elemente im Puffer (initialisiert mit 0)
- Benutze sleep und wakeup, wenn die Grenzen 0 bzw. MAX\_BUFFER erreicht sind
- Anfangs schläft Verbraucher



# Das Produzenten-Konsumenten-Problem – Eine Idee

## Prozedur **producer**

```
{
  ...
  wiederhole
  {
    item = produce_item();           // produziere nächstes
    wenn (count = MAX_BUFFER)       // schlafe, wenn Puffer voll
      sleep(producer_id)
    insert_item(item);              // füge Objekt in Puffer ein
    count = count + 1;
    wenn (count = 1)                // wenn Puffer vorher leer
      wakeup(consumer_id);         // wecke Konsumenten
  }
}
```

# Das Produzenten-Konsumenten-Problem – Eine Idee

## Prozedur **consumer**

```
{
  ...
  wiederhole
  {
    wenn (count = 0) // schlafe, wenn Puffer leer
      sleep(consumer_id);
    item = remove_item(); // entferne Objekt aus Puffer
    count = count - 1;
    wenn (count = MAX_BUFFER - 1) // wenn Puffer vorher voll
      wakeup(producer_id); // wecke Erzeuger
    consume_item(item); // verbrauche Objekt
  }
}
```

# Das Produzenten-Konsumenten-Problem – Eine Idee

Prozedur **consumer**

```
{  
  ...  
  wiederhole  
  {  
    wenn (count = 0) // schlafe, wenn Puffer leer  
      sleep(consumer_id);  
    item = remove_item(); // entferne Objekt aus Puffer  
    count = count - 1;  
    wenn (count = MAX_BUFFER - 1) // wenn Puffer vorher voll  
      wakeup(producer_id); // wecke Erzeuger  
    consume_item(item); // verbrauche Objekt  
  }  
}
```

Ist diese Lösung korrekt?

# Das Produzenten-Konsumenten-Problem – Fehlersituation (1)

1. Fehlersituation mit zwei Verbrauchern:

- Es gibt 1 Objekt im Puffer
- 1. Verbraucher entnimmt Objekt, wird unterbrochen, **bevor** count reduziert ist
- 2. Verbraucher will Objekt aus Puffer entnehmen, es gilt noch count=1
- Schutz gegen Entnahme aus dem Puffer funktioniert nicht!

# Das Produzenten-Konsumenten-Problem – Eine Idee

Prozedur **consumer**

```
{
  ...
  wiederhole
  {
    wenn (count = 0) // schlafe, wenn Puffer leer
      sleep(consumer_id);
    item = remove_item(); // entferne Objekt aus Puffer
    count = count - 1;
    → wenn (count = MAX_BUFFER - 1) // wenn Puffer vorher voll
      wakeup(producer_id); // wecke Erzeuger
    consume_item(item); // verbrauche Objekt
  }
}
```

Ist diese Lösung korrekt?



# Das Produzenten-Konsumenten-Problem – Fehlersituation (2)

2. Fehlersituation mit zwei Verbrauchern und einem schlafenden Erzeuger

- Puffer ist voll ( $\text{count} = \text{MAX\_BUFFER}$ )
- 1. Verbraucher entnimmt Objekt, zählt count runter:  $\text{count} = \text{MAX\_BUFFER} - 1$ , wird dann unterbrochen
- 2. Verbraucher entnimmt ebenfalls Objekt, zählt count runter:  $\text{count} = \text{MAX\_BUFFER} - 2$
- Aufwecken des Erzeugers geht verloren!

# Das Produzenten-Konsumenten-Problem – Eine Idee

Prozedur **consumer**

```
{
  ...
  wiederhole
  {
    wenn (count = 0) // schlafe, wenn Puffer leer
    → sleep(consumer_id);
    item = remove_item(); // entferne Objekt aus Puffer
    count = count - 1;
    wenn (count = MAX_BUFFER - 1) // wenn Puffer vorher voll
      wakeup(producer_id); // wecke Erzeuger
    consume_item(item); // verbrauche Objekt
  }
}
```

Ist diese Lösung korrekt?

# Das Produzenten-Konsumenten-Problem – Fehlersituation (3)

3. Fehlersituation mit je einem Verbraucher und Erzeuger

- Puffer ist leer
- „wenn (count=0)“ wird ausgeführt, dann Unterbrechung **vor** sleep(consumer\_id)
- Es wird komplett Erzeuger ausgeführt und Aufruf „wakeup(consumer\_id)“ hat keinen Effekt
- Aufwecken geht verloren!

# Das Produzenten-Konsumenten-Problem – Fehlersituation (3)

- Dann wird Verbraucher weiter ausgeführt, „sleep(consumer\_id)“
- Situation: Verbraucher schläft, obwohl Puffer nicht leer!
- Evtl. noch weitere Erzeuger-Aufrufe, aber es gilt nie mehr `count=1`, daher gibt es nie mehr den Aufruf „wakeup(consumer\_id)“
- Analog bei Unterbrechung nach „wenn (`count = MAX_BUFFER`)“ in Erzeuger

# Das Produzenten-Konsumenten-Problem – Zusammenfassung

- Diese Idee ist keine Lösung!
- Problem beim letzten Beispiel:
  - „wenn (count=0)  
sleep(consumer\_id)“  
ist **keine atomare** Operation
- Entscheidende Befehle dürfen nicht unterbrochen werden

# Bemerkungen

- Warum Abfrage auf „=“ und nicht auf „<=“ bzw. „>=“ vor wakeup?
  - Würde fast immer zum Überprüfen der Warteschlange führen (auch wenn keine Prozesse darin sind)
  - Meistens überflüssig, zu viele Systemaufrufe
- Warum nicht um alles eine kritische Region?
  - Blockieren möglich
  - Falls z.B. sleep ausgeführt wird bei leerem Buffer, kann Produzent nicht dafür sorgen, dass ein Aufwecken stattfindet

# Das Produzenten-Konsumenten-Problem – Elegante Lösung

- **Semaphor**: Datenstruktur zur Verwaltung beschränkter Ressourcen (Dijkstra, 1965)
- Wert des Semaphors repräsentiert die Anzahl der Weckrufe, die ausstehen

# Wert eines Semaphors

Drei mögliche Situationen für den Wert eines Semaphors:

- Wert  $< 0$ : weitere Weckrufe stehen schon aus, nächster Prozess legt sich auch schlafen
- Wert  $> 0$ : frei, nächster Prozess darf fortfahren
- Wert  $0$ : keine Weckrufe sind bisher gespeichert, nächster Prozess legt sich schlafen



# Semaphor: down/up Operationen

- Initialisiere Zähler des Semaphors:  $\text{count}_s = 1$
- **down-Operation:**
  - Verringere den Wert von  $\text{count}_s$  um 1
  - Wenn  $\text{count}_s < 0$ , blockiere den aufrufenden Prozess, sonst fahre fort
- **up-Operation:**
  - Erhöhe den Wert von  $\text{count}_s$  um 1
  - Wenn  $\text{count}_s \leq 0$ , wecke einen der blockierten Prozesse auf

# Semaphore

Ein Semaphor  $s$  hat drei Komponenten:

- Integer-Variable  $\text{count}_s$
- Warteschlange  $\text{queue}_s$
- Lock-Variable  $\text{lock}_s$

# Wechselseitiger Ausschluss mit Semaphoren (1)

- Annahme: n Prozesse sind gestartet, konkurrieren um kritischen Abschnitt
- $\text{count}_s$  ist initialisiert mit 1

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (2)

- 1. Prozess will in kritische Region

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (2)

- 1. Prozess will in kritische Region
- `down(s)` wird ausgeführt:  $\text{count}_s = 0$

```
/* Prozess i */  
wiederhole  
{  
→ down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (2)

- 1. Prozess will in kritische Region
- `down(s)` wird ausgeführt:  $\text{count}_s=0$
- 1. Prozess muss nicht blockieren, betritt kritische Region

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  → /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (3)

- Nun: 2. Prozess will in kritische Region

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (3)

- Nun: 2. Prozess will in kritische Region
- `down(s)` wird ausgeführt:  $\text{count}_s = -1$

```
/* Prozess i */  
wiederhole  
{  
→ down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```



# Wechselseitiger Ausschluss mit Semaphoren (3)

- Nun: 2. Prozess will in kritische Region
- `down(s)` wird ausgeführt:  $\text{count}_s = -1$
- 2. Prozess wird schlafengelegt + eingefügt in  $\text{queue}_s$

```
/* Prozess i */  
wiederhole  
{  
→ down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (3)

- Nun: 2. Prozess will in kritische Region
- `down(s)` wird ausgeführt:  $\text{count}_s = -1$
- 2. Prozess wird schlafengelegt + eingefügt in  $\text{queue}_s$
- Analog für 3. Prozess: `down(s)` führt zu  $\text{count}_s = -2$ , Prozess wird schlafengelegt + eingefügt in  $\text{queue}_s$

```
/* Prozess i */  
wiederhole  
{  
→ down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (4)

- 1. Prozess verlässt irgendwann kritische Region

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  → up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (4)

- 1. Prozess verlässt irgendwann kritische Region
- $up(s)$  wird ausgeführt:  $count_s = -1$

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  → up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (4)

- 1. Prozess verlässt irgendwann kritische Region
- $up(s)$  wird ausgeführt:  $count_s = -1$
- $count_s \leq 0$ : Einer der wartenden Prozesse wird aufgeweckt und kann kritische Region betreten

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  → up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (4)

- 1. Prozess verlässt irgendwann kritische Region
- $up(s)$  wird ausgeführt:  $count_s = -1$
- $count_s \leq 0$ : Einer der wartenden Prozesse wird aufgeweckt und kann kritische Region betreten
- Annahme: 2. Prozess wird gewählt, betritt krit. Region

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (5)

- 2. Prozess verlässt irgendwann kritische Region
- $up(s)$  wird ausgeführt:  $count_s = 0$
- $count_s \leq 0$ : Einer der wartenden Prozesse wird aufgeweckt und kann kritische Region betreten

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  → up(s);  
  /* nichtkritische Region */;  
}
```

# Wechselseitiger Ausschluss mit Semaphoren (5)

- 2. Prozess verlässt irgendwann kritische Region
- $up(s)$  wird ausgeführt:  $count_s=0$
- $count_s \leq 0$ : Einer der wartenden Prozesse wird aufgeweckt und kann kritische Region betreten
- Prozess 3 betritt kritische Region usw.

```
/* Prozess i */  
wiederhole  
{  
  down(s);  
  /* kritische Region */;  
  up(s);  
  /* nichtkritische Region */;  
}
```



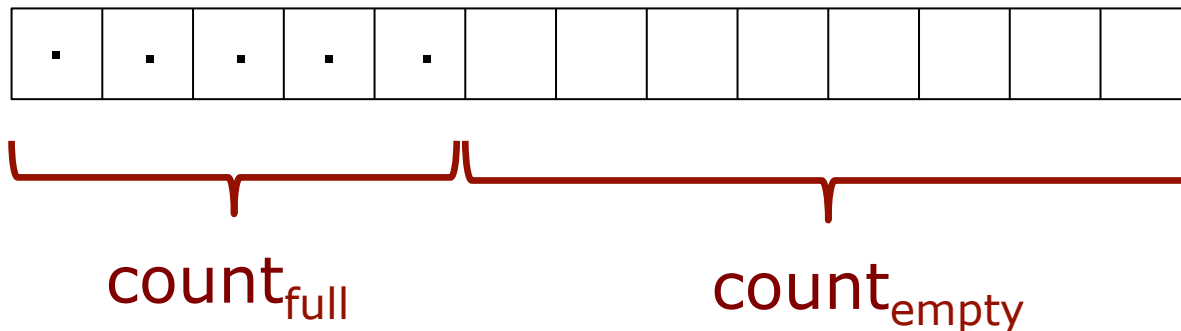
# Semaphoren für Produzenten-Konsumenten-Problem

- Auf 1 initialisierte Semaphoren heißen **binäre Semaphoren**
- Behandlung mehrfach nutzbarer Ressourcen möglich durch Initialisierung:  $\text{count}_s = m, m > 1$
- Wenn  $\text{count}_s \geq 0$ , dann Wert von  $\text{count}_s$ :  
Anzahl der Prozesse, die  $\text{down}(s)$  ausführen können ohne zu blockieren (ohne dass zwischenzeitlich  $\text{up}(s)$  ausgeführt wird)
- $\text{count}_s < 0$ , dann gilt:  
 $|\text{count}_s|$  ist die Anzahl der wartenden Prozesse in  $\text{queue}_s$

# Produzenten-Konsumenten-Problem mit Semaphoren (1)

Drei verschiedene Semaphore werden benötigt:

- mutex: für wechselseitigen Ausschluss
- empty: zählt freie Plätze
- full: zählt belegte Plätze



# Produzenten-Konsumenten-Problem mit Semaphoren (2)

- Puffer ist anfangs leer:
  - $\text{count}_{\text{empty}} = \text{MAX\_BUFFER}$
  - $\text{count}_{\text{full}} = 0$

# Produzenten-Konsumenten-Problem mit Semaphoren (2)

- Puffer ist anfangs leer:
  - $\text{count}_{\text{empty}} = \text{MAX\_BUFFER}$
  - $\text{count}_{\text{full}} = 0$
- Idee:
  - Immer wenn etwas entfernt werden soll, führe  $\text{down}(\text{full})$  aus; wenn  $\text{count}_{\text{full}} < 0$ : Blockiere
  - Immer wenn etwas hinzugefügt werden soll: führe  $\text{down}(\text{empty})$  aus; wenn  $\text{count}_{\text{empty}} < 0$ : Blockiere
  - In up werden evtl. schlafende Prozesse geweckt

# Produzenten-Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; countmutex = 1;           // mutex für kritische Regionen
semaphore empty; countempty = MAX_BUFFER;    // zählt freie Plätze
semaphore full; countfull = 0;              // zählt belegte Plätze
```

## Prozedur **producer**


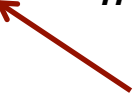
```
{
  ...
  wiederhole
  {
    item = produce_item(); // produziere nächstes Objekt
    → down(empty);        zähle runter; überprüfe, ob noch 1 Platz frei; ggf. sleep
    down(mutex);
    insert_item(item);    // füge Objekt in Puffer ein
    up(mutex);
    up(full);
  }
}
```

# Produzenten-Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; countmutex = 1;           // mutex für kritische Regionen
semaphore empty; countempty = MAX_BUFFER; // zählt freie Plätze
semaphore full; countfull = 0;           // zählt belegte Plätze
```

## Prozedur **producer**

```
{
  ...
  wiederhole
  {
    item = produce_item(); // produziere nächstes Objekt
    down(empty);
    down(mutex);
    insert_item(item); // füge Objekt in Puffer ein
    up(mutex);
    up(full);
  }
}
```

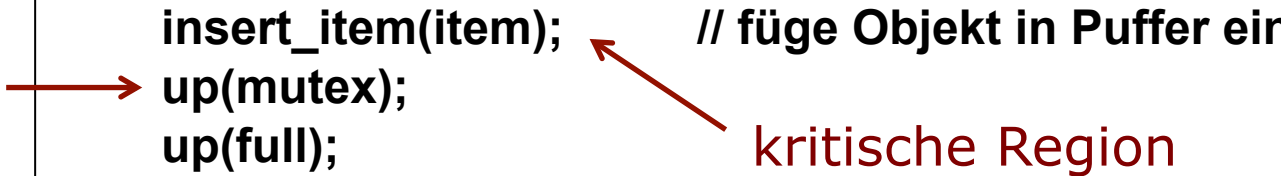
  kritische Region

# Produzenten-Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; countmutex = 1;           // mutex für kritische Regionen  
semaphore empty; countempty = MAX_BUFFER; // zählt freie Plätze  
semaphore full; countfull = 0;           // zählt belegte Plätze
```

## Prozedur **producer**

```
{  
  ...  
  wiederhole  
  {  
    item = produce_item(); // produziere nächstes Objekt  
    down(empty);  
    down(mutex);  
    insert_item(item); // füge Objekt in Puffer ein  
    up(mutex);  
    up(full);  
  }  
}
```

kritische Region

# Produzenten-Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; countmutex = 1;           // mutex für kritische Regionen
semaphore empty; countempty = MAX_BUFFER; // zählt freie Plätze
semaphore full; countfull = 0;           // zählt belegte Plätze
```

Prozedur **producer**

```
{
  ...
  wiederhole
  {
    item = produce_item(); // produziere nächstes Objekt
    down(empty);
    down(mutex);
    insert_item(item);     // füge Objekt in Puffer ein
    up(mutex);
    up(full);
  }
}
```

zähle Anzahl voller Plätze hoch; wecke ggf. einen Konsumenten auf, der vorher blockiert hatte



# Produzenten-Konsumenten-Problem mit Semaphoren (4)

```
semaphore mutex; countmutex = 1;           // mutex für kritische Regionen
semaphore empty; countempty = MAX_BUFFER;    // zählt freie Plätze
semaphore full; countfull = 0;             // zählt belegte Plätze
```

## Prozedur **consumer**

```
{
  ...
  wiederhole
  {
    → down(full); zähle runter; überprüfe, ob mind. 1 Element; ggf. sleep
    down(mutex);
    item = remove_item(); // entferne Objekt aus Puffer
    up(mutex);
    up(empty);
    consume_item(item); // konsumiere Objekt
  }
}
```

# Produzenten-Konsumenten-Problem mit Semaphoren (4)

```
semaphore mutex; countmutex = 1;           // mutex für kritische Regionen
semaphore empty; countempty = MAX_BUFFER;    // zählt freie Plätze
semaphore full; countfull = 0;              // zählt belegte Plätze
```

## Prozedur **consumer**

```
{
  ...
  wiederhole
  {
    down(full);
    [ down(mutex);
      item = remove_item(); // entferne Objekt aus Puffer
      up(mutex);
      up(empty);
      consume_item(item); // konsumiere Objekt
    ]
  }
}
```

# Produzenten-Konsumenten-Problem mit Semaphoren (4)

```
semaphore mutex; countmutex = 1;           // mutex für kritische Regionen
semaphore empty; countempty = MAX_BUFFER;   // zählt freie Plätze
semaphore full; countfull = 0;             // zählt belegte Plätze
```

## Prozedur **consumer**

```
{
  ...
  wiederhole
  {
    down(full);
    down(mutex);
    item = remove_item(); // entferne Objekt aus Puffer
    up(mutex);
    → up(empty); zähle Anzahl freier Plätze hoch; wecke ggf. Produzenten
    consume_item(item); // konsumiere Objekt
  }
}
```

# Produzenten-Konsumenten-Problem mit Semaphoren (5)

- Funktioniert für eine Anzahl  $m > 1$  von Prozessen, wenn die Operationen zusammenhängend ausgeführt werden
- Das Betriebssystem garantiert die atomare Ausführung


# Produzenten-Konsumenten-Problem mit Semaphoren (5)

Frage: Funktioniert das immer noch, wenn in Prozedur consumer

- `up(mutex)` und `up(empty)` vertauscht werden?

```
Prozedur consumer
{
  ...
  wiederhole
  {
    down(full);
    down(mutex);
    item = remove_item(); // entferne Objekt aus Puffer
    up(mutex);
    up(empty);
    consume_item(item); // konsumiere Objekt
  }
}
```

ja!



# Produzenten-Konsumenten-Problem mit Semaphoren (6)

Führt im Wesentlichen zu Effizienzproblem:

- Durch `up(empty)` kann ein Produzent aufgeweckt werden und in seine kritische Region wollen
- Dieser ist durch Mutex geschützt, ist noch nicht frei und der Produzent blockiert deswegen noch einmal
- Konsument ruft `up(mutex)` erst später auf, erst dann wird Produzent wieder aufgeweckt

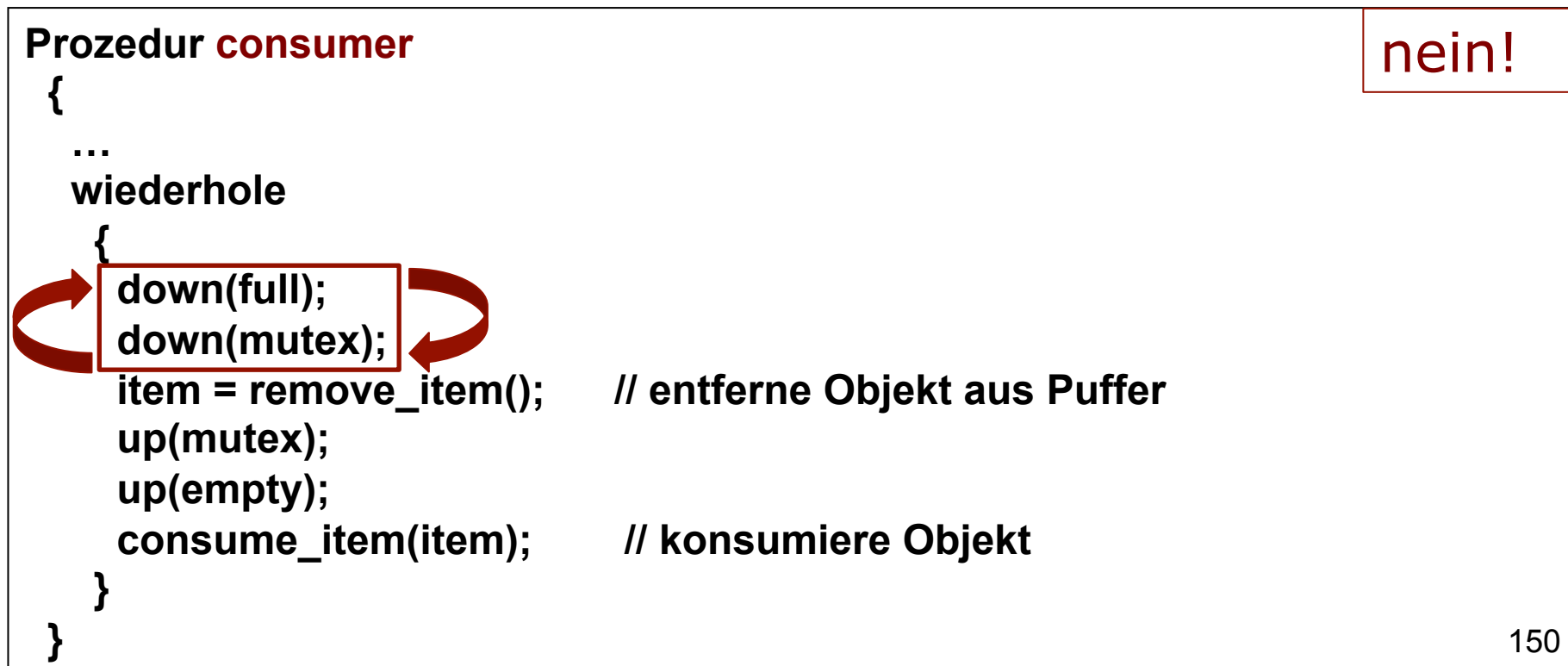
# Produzenten-Konsumenten-Problem mit Semaphoren (7)

Frage: Funktioniert das immer noch, wenn in Prozedur consumer

- `down(full)` und `down(mutex)` vertauscht werden?

```
Prozedur consumer
{
  ...
  wiederhole
  {
    down(full);
    down(mutex);
    item = remove_item(); // entferne Objekt aus Puffer
    up(mutex);
    up(empty);
    consume_item(item); // konsumiere Objekt
  }
}
```

**nein!**



# Produzenten-Konsumenten-Problem mit Semaphoren (6)

Kann zu Deadlock führen:

- Konsument führt `down(mutex)` aus
- Annahme: Der Buffer leer ist, dann blockiert Konsument beim Ausführen von `down(full)`
- Produzent braucht aber Zugriff auf die kritische Region, um danach `up(full)` auszuführen und so den Konsumenten wieder aufzuwecken
- Wenn Konsument aber nie geweckt wird, kann er auch nie `up(mutex)` ausführen
- Deadlock!



# Implementierung von Semaphoren: Versuch 1

- Implementierung der Systemaufrufe down und up
- $\text{mutex}_s$  innerhalb des Semaphors, Pseudocode:

```
down(semaphore s)
{
    mutex_lock(mutex_s);
    count_s = count_s - 1;
    wenn (count_s < 0)
    {
        füge diesen Prozess in
                               queue_s ein;
        blockiere den Prozess und
        führe unmittelbar vor
        Abgabe des Prozessors noch
        mutex_unlock(mutex_s) aus
    }
    sonst
        mutex_unlock(mutex_s);
}
```

```
up(semaphore s)
{
    mutex_lock(mutex_s);
    count_s = count_s + 1;
    if (count_s <= 0)
    {
        entferne einen Prozess P aus
                               queue_s;
        füge Prozess P in Liste
        der bereiten Prozesse ein
    }
    mutex_unlock(mutex_s);
}
```

# Implementierung von Semaphoren: Versuch 1 Analyse

- down und up sind nicht wirklich atomar, aber trotzdem stören sich verschiedene Aufrufe von down und up nicht
- Zwei Queues müssen verwaltet werden:
  - Liste von Prozessen, die auf Freigabe des Mutex warten
  - Liste von Prozessen, die auf Erhöhung der Semaphor-Variable warten

# Implementierung von Semaphoren: Versuch 2

- Implementierung der Systemaufrufe down und up
- Benutze TSL, Lock-Variable  $lock_s$ , Pseudocode:

```
down(semaphore s)
{
  { solange (testset(locks) = false)
    tue nichts;
    counts = counts - 1;
    wenn (counts < 0)
      {
        füge diesen Prozess in
                               queues ein;
        blockiere den Prozess und
        führe unmittelbar vor
        Abgabe des Prozessors noch
        locks = 0 aus
      }
  sonst
    locks = 0
}
```

```
up(semaphore s)
{
  { solange (testset(locks) = false)
    tue nichts;
    counts = counts + 1;
    if (counts <= 0)
      {
        entferne einen Prozess P aus
                               queues;
        füge Prozess P in Liste
        der bereiten Prozesse ein
      }
    locks = 0;
}
```

Freigabe der kritischen Region  
des Semaphors

# Implementierung von Semaphoren: Versuch 2 Analyse

- Aktives Warten, aber nicht so gravierend:  
Beschränkt auf up und down
- down und up sind nicht wirklich atomar,  
aber trotzdem stören sich verschiedene  
Aufrufe nicht
- Nur eine Warteschlange

# Zusammenfassung

- CPU (ein oder mehrere Kerne) wird von mehreren Prozessen geteilt
- Verwaltung gemeinsamer Ressourcen bei nebenläufigen Prozessen ist notwendig und nicht trivial
- Kritische Wettläufe möglich, formale Beweise nötig
- Verschiedene Konzepte für wechselseitigen Ausschluss
- Bekanntes Produzenten-Konsumenten-Problem

# Wichtige Begriffe (1)

- **Atomare Operation**: Sequenz, die nicht unterbrochen werden kann
- **Kritische Region**: Stück Code, der Zugriff auf gemeinsame Ressource fordert
- **Wechselseitiger Ausschluss**: Wenn ein Prozess in kritischer Region ist, darf kein anderer in eine kritische Region, die Zugang zum selben Speicherbereich fordert
- **Deadlock**: Von zwei Prozessen kann keiner fortfahren, weil jeder darauf warten muss, dass der andere etwas tut

## Wichtige Begriffe (2)

- **Semaphor**: Für mehrfach nutzbare Ressourcen; zählt Anzahl der Weckrufe, die ausstehen
- **Binäres Semaphor**: Mit 1 initialisiert
- **Mutex**: Ähnlich wie binäres Semaphor
- Der Prozess, der Mutex sperrt, muss ihn auch wieder freigeben