

Systeme I: Betriebssysteme

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

Wolfram Burgard



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

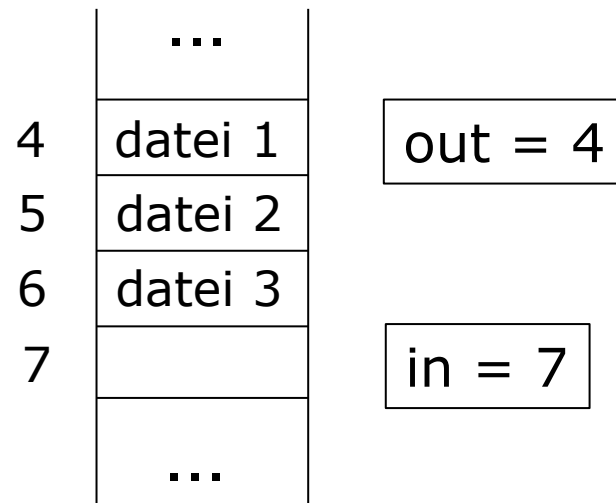
- Das Betriebssystem muss die Ressourcen der aktiven Prozesse verwalten
- Bei Zugriff auf gemeinsame Ressourcen muss **wechselseitiger Ausschluss** garantiert werden
- Die Korrektheit des Ergebnisses muss **unabhängig von der Ausführungsgeschwindigkeit** sein

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

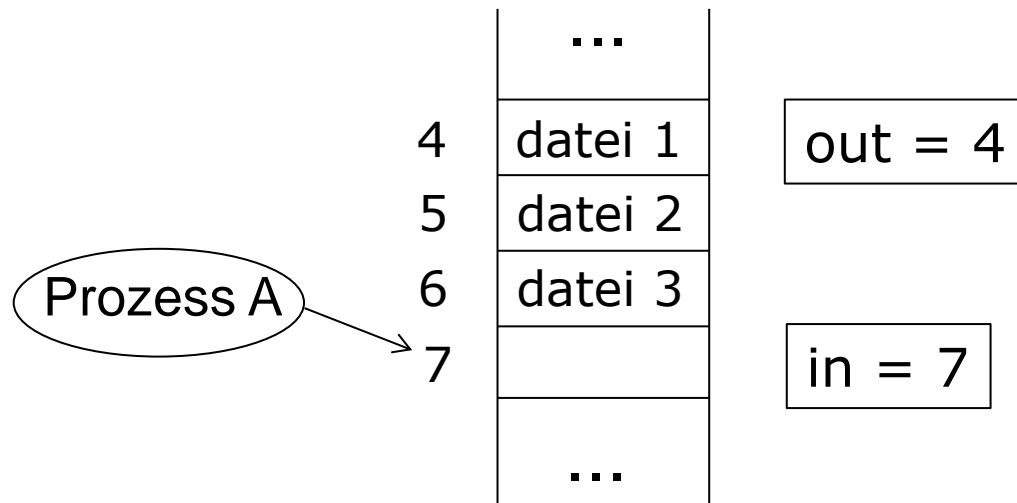
Beispiel: Druckerpuffer

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



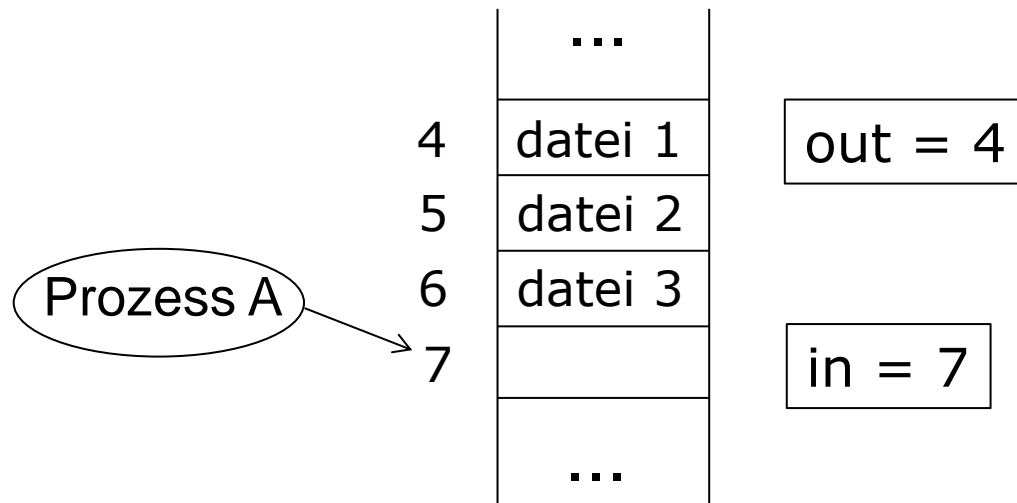
Beispiel: Druckerpuffer

- Prozess A liest *in* aus und speichert 7



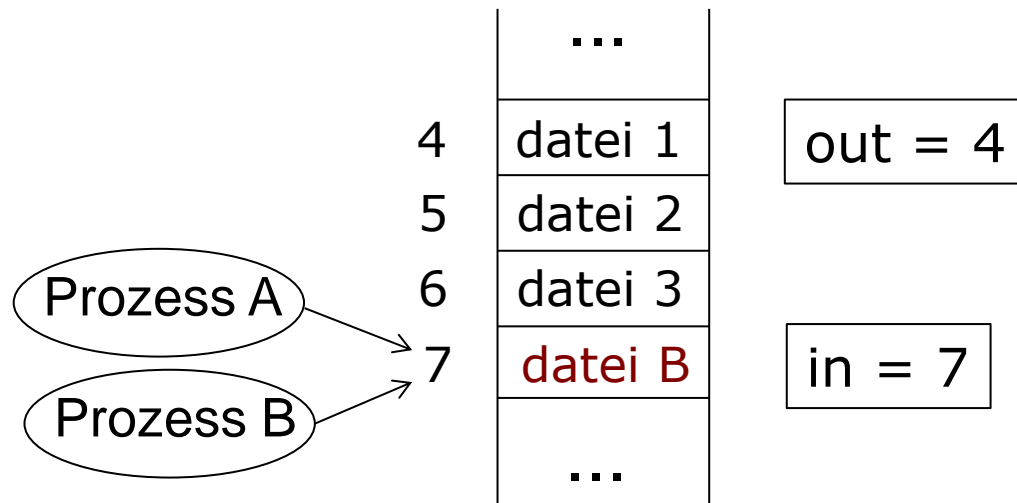
Beispiel: Druckerpuffer

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



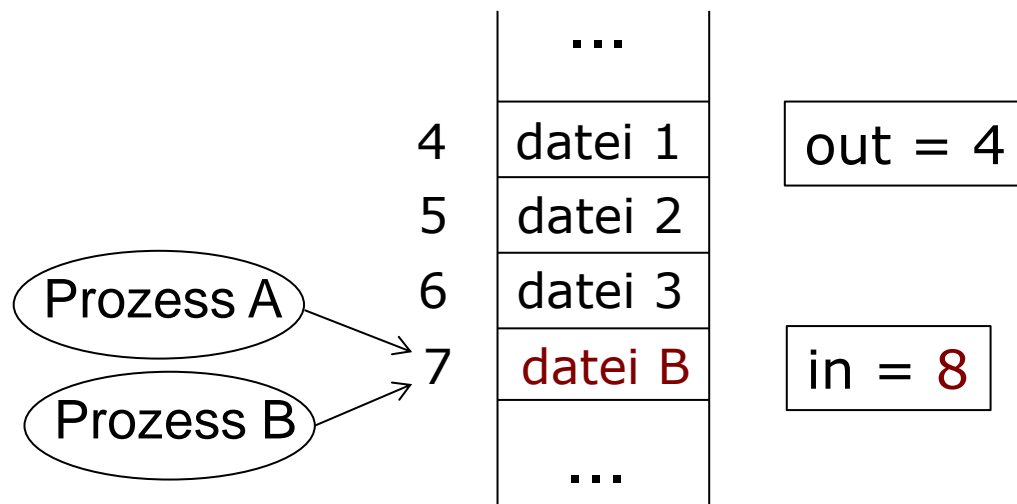
Beispiel: Druckerpuffer

- 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



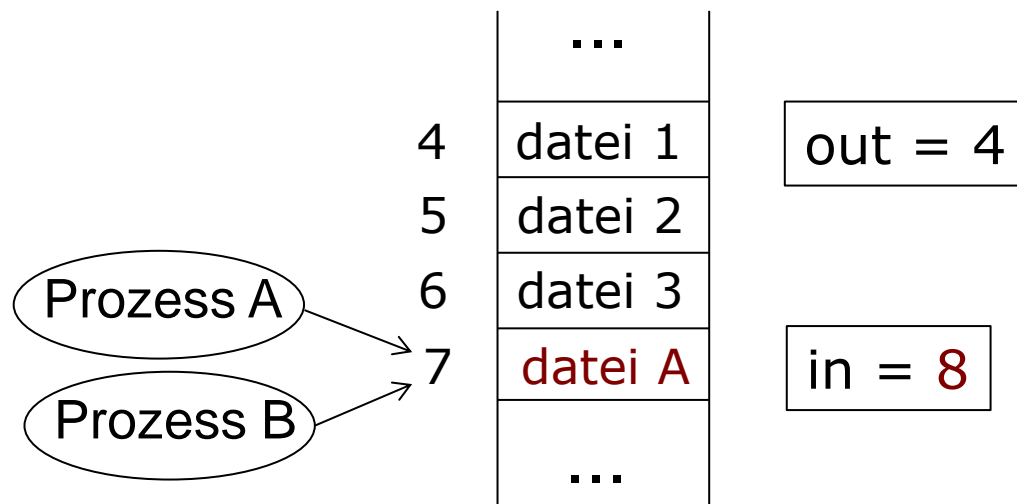
Beispiel: Druckerpuffer

- 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“)
- Verbiете 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

1. Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein (wechselseitiger Ausschluss)
2. Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der Rechenkerne gemacht werden
3. Kein Prozess, der außerhalb seiner kritischen Regionen läuft, darf andere Prozesse blockieren
4. 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
- **In das Betriebssystem integrierte Lösungen**

Versuch 1a: Sperren mit Variable

- 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 mit Variable

- Falls $turn = 0$, setzt der Prozess $turn$ auf 1 und betritt seine kritische Region
- Falls von $turn = 1$, wartet der Prozess, bis $turn = 0$
- **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 sind 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 $turn = 0$ und betritt seine kritische Region
- Prozess 1 sitzt fest bis $turn = 1$

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

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

Versuch 1b: Strikter Wechsel

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

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

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

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

Versuch 1b: Strikter Wechsel

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

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

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

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

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


Versuch 1b: Strikter Wechsel

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

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

```
/* Prozess 1 */  
wiederhole  
{  
  solange (turn ≠ 1)  
    tue nichts;  
  /* kritische Region */  
  turn := 0;  
  /* nicht-kritische 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 nicht-kritischen Region beschäftigt

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

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

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

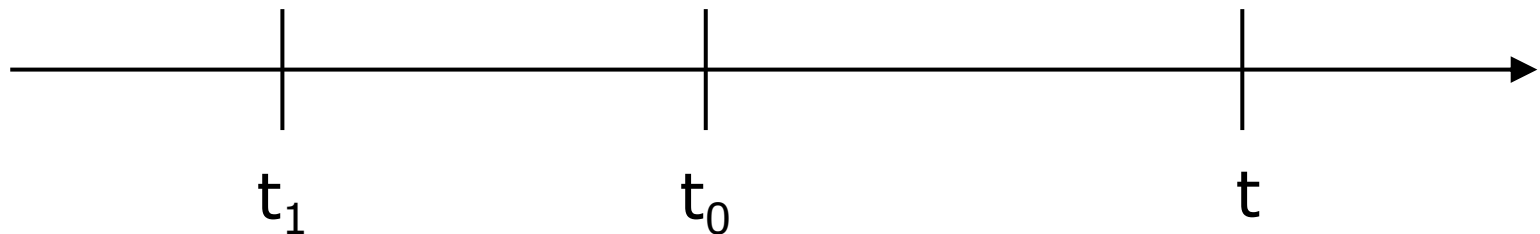
Wechselseitiger Ausschluss

Satz:

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

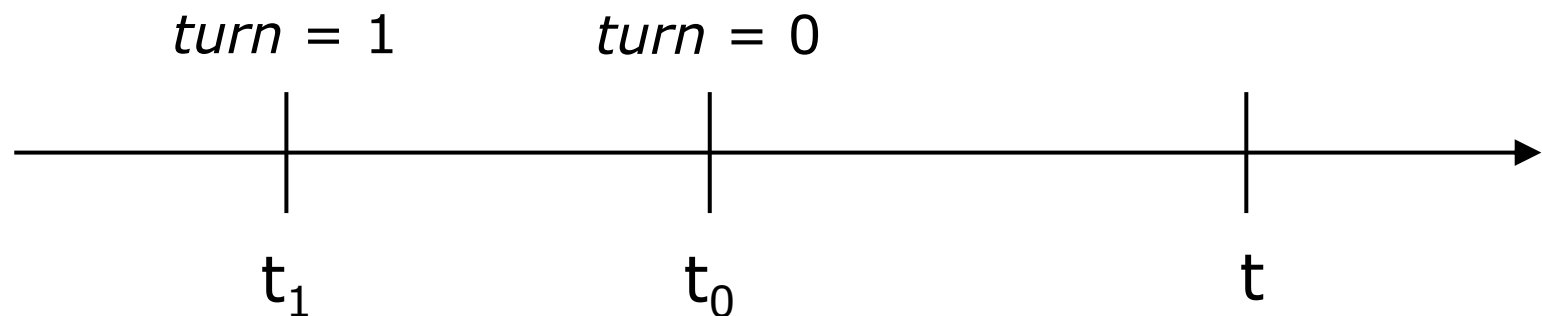
Beweis durch Widerspruch

- 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 nicht-kritischen Abschnitt
 - Aktives Warten („Busy Waiting“) führt zu Verschwendung von Rechenzeit

Nächster Versuch

- Im bisherigen Ansatz 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 als ein anderer, kann der andere immer noch in seine kritische Region eintreten (sofern der andere nicht in seiner eigenen kritischen Region 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;  
  /* nicht-kritische Region */  
}
```

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

Versuch 2

- Garantiert das wechselseitigen Ausschluss?

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

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

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

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

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

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

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

Versuch 3

- Führt das zu wechselseitigem Ausschluss?

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

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

Versuch 3

- Führt das zu wechselseitigem Ausschluss?
- 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;  
  /* nicht-kritische Region */  
}
```

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

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

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

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

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;  
  /* nicht-kritische Region */  
}
```

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

- $t_{1,0}$: Der letzte Zeitpunkt $< t$, zu dem **Prozess 0** die solange-Schleife verlässt
- $t_{1,1}$: Der letzte Zeitpunkt $< t$, zu dem **Prozess 1** die solange-Schleife verlässt
- $t_{2,0}$: Der letzte Zeitpunkt $< t$, zu dem **Prozess 0** $turn := 1$ ausführt
- $t_{2,1}$: Der letzte Zeitpunkt $< t$, zu dem **Prozess 1** $turn := 0$ ausführt
- $t_{3,0}$: Der letzte Zeitpunkt $< t$, zu dem **Prozess 0** $flag[0] := true$ ausführt
- $t_{3,1}$: Der letzte Zeitpunkt $< t$, zu dem **Prozess 1** $flag[1] := true$ ausführt

Beweis durch Widerspruch

/* Prozess 0 */

wiederhole

```
{  
  flag[0] := true;      ←  $t_{3,0}$   
  turn := 1;           ←  $t_{2,0}$   
  solange (flag[1] = true  
           und turn = 1)  
    tue nichts;  
  /* kritische Region */ ←  $t_{1,0}$   
  flag[0] := false;  
  /* nicht-kritische Region */  
}
```

/* Prozess 1 */

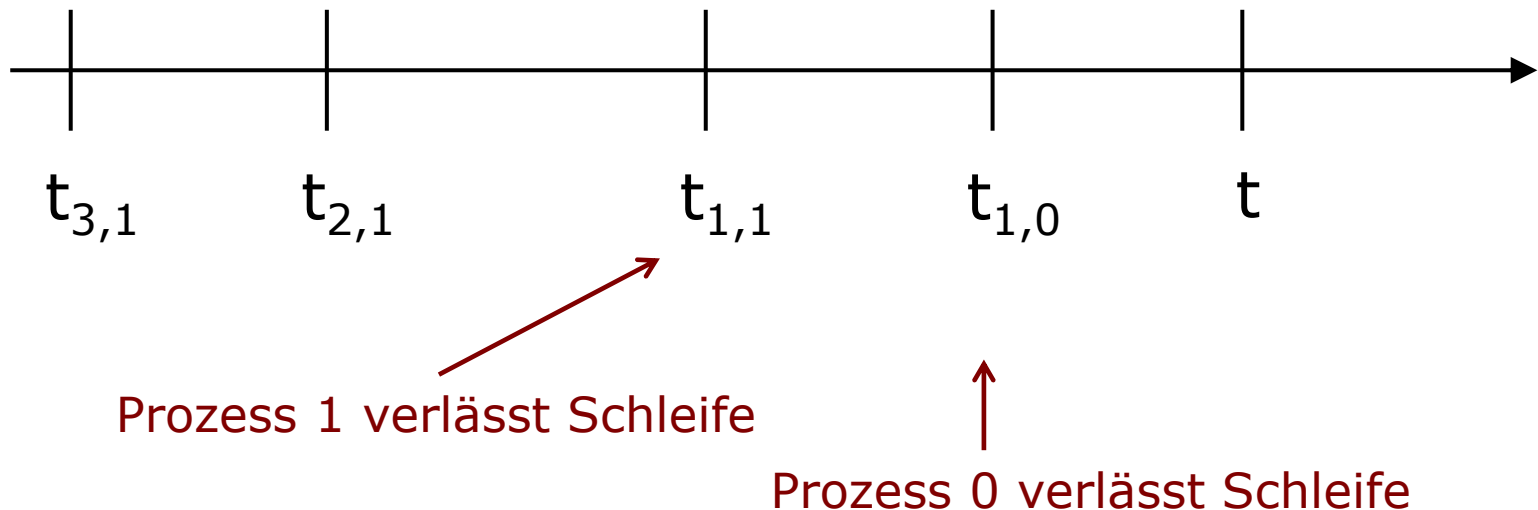
wiederhole

```
{  
  flag[1] := true;      ←  $t_{3,1}$   
  turn := 0;           ←  $t_{2,1}$   
  solange (flag[0] = true  
           und turn = 0)  
    tue nichts;  
  /* kritische Region */ ←  $t_{1,1}$   
  flag[1] := false;  
  /* nicht-kritische Region */  
}
```

t →

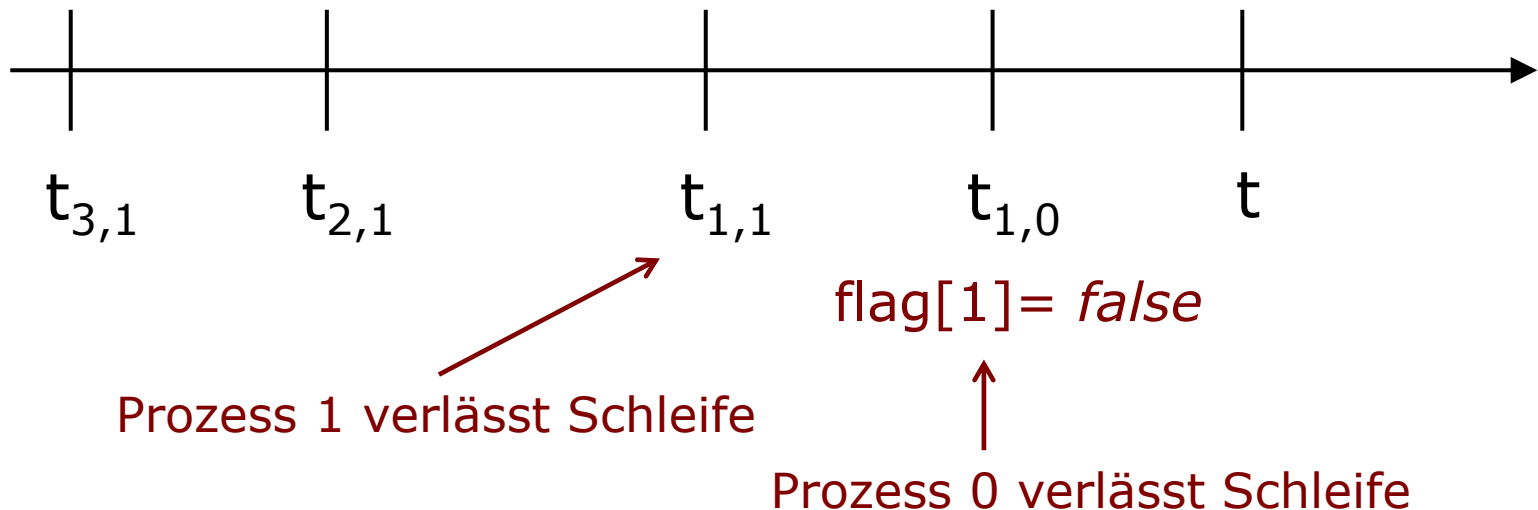
Beweis durch Widerspruch

- O.B.d.A. geht Prozess 1 vor Prozess 0 in seinen kritischen Abschnitt, d.h. $t_{1,1} < t_{1,0}$



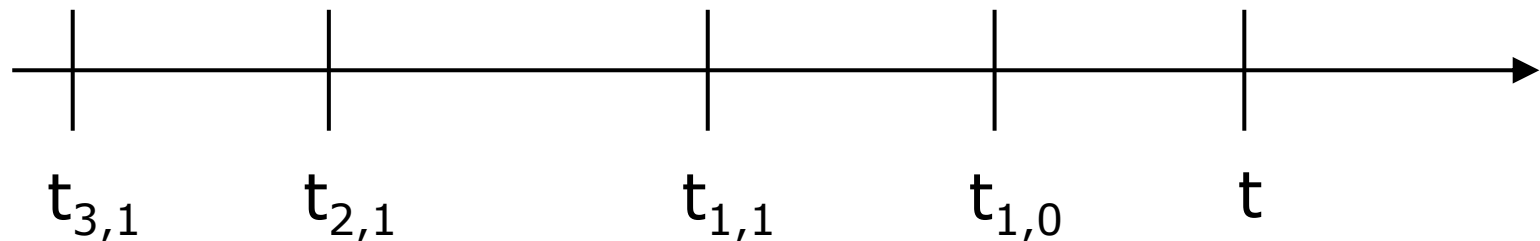
Beweis durch Widerspruch

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



Beweis durch Widerspruch

- O.B.d.A. geht Prozess 1 vor Prozess 0 in seinen kritischen Abschnitt, d.h. $t_{1,1} < t_{1,0}$
- Fallunterscheidung nach dem Grund des Verlassens der solange-Schleife in Prozess 0 zur Zeit $t_{1,0}$
- **Fall 1:** $\text{flag}[1] = \text{false}$ zum Zeitpunkt $t_{1,0}$
- Zum Zeitpunkt $t_{3,1}$ wurde $\text{flag}[1] := \text{true}$ gesetzt



$\text{flag}[1] := \text{true}$

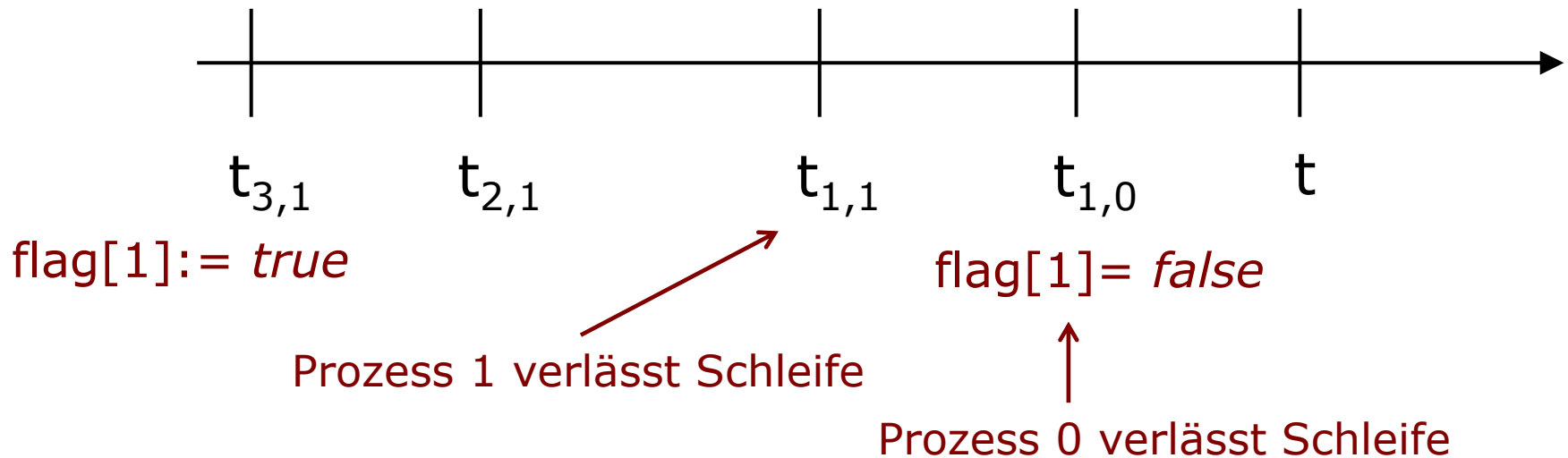
Prozess 1 verlässt Schleife

$\text{flag}[1] = \text{false}$

Prozess 0 verlässt Schleife

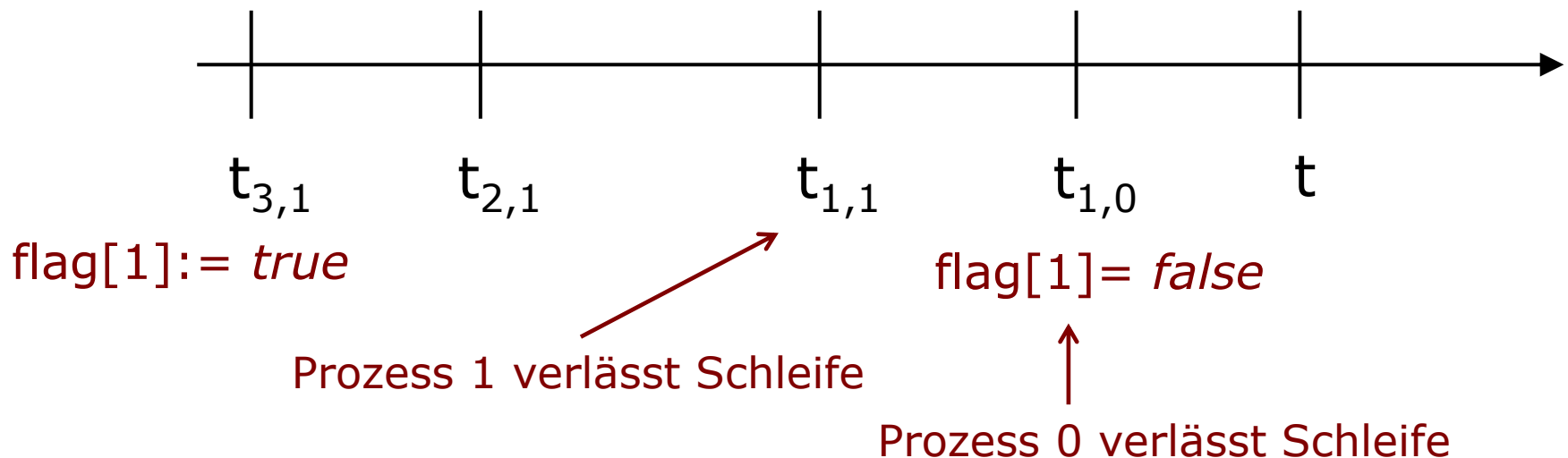
Beweis durch Widerspruch

- Es müsste zwischen $t_{3,1}$ und $t_{1,0}$ $\text{flag}[1] := \text{false}$ gesetzt worden sein



Beweis durch Widerspruch

- Es müsste zwischen $t_{3,1}$ und $t_{1,0}$ $\text{flag}[1] := \text{false}$ gesetzt worden sein
- $\text{flag}[1] := \text{false}$ passiert nur bei der Initialisierung und **nach** der kritischen Region von Prozess 1
- Zwischen $t_{3,1}$ und $t_{1,0}$ erreicht Prozess 1 aber nicht das Ende der kritischen Region! (entspr. Annahme)

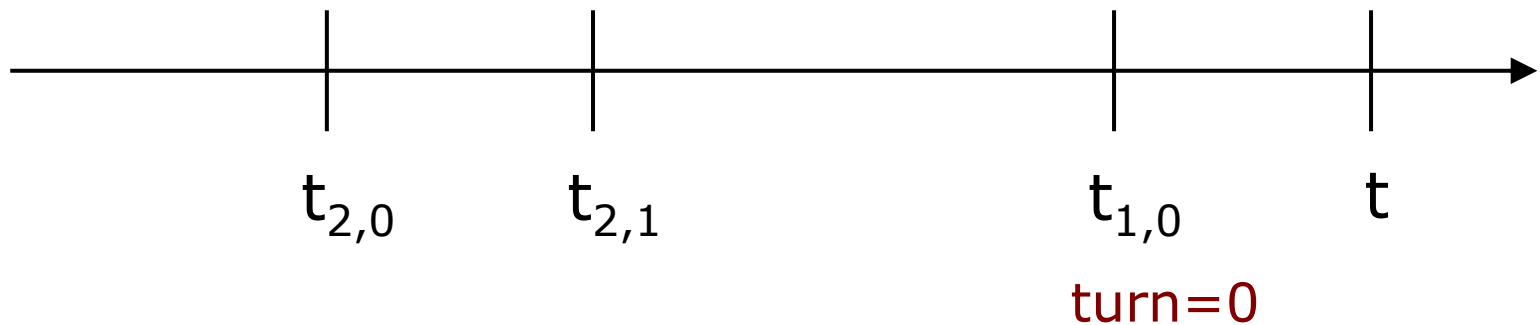


Beweis durch Widerspruch

- Es müsste zwischen $t_{3,1}$ und $t_{1,0}$ $\text{flag}[1] := \text{false}$ gesetzt worden sein
- $\text{flag}[1] := \text{false}$ passiert nur bei der Initialisierung und **nach** der kritischen Region von Prozess 1
- Zwischen $t_{3,1}$ und $t_{1,0}$ erreicht Prozess 1 aber nicht das Ende der kritischen Region! (entspr. Annahme)
- Prozess 0 kann wegen dieser Bedingung nicht die Schleife verlassen haben und in seiner kritischen Region sein
- Widerspruch zur Annahme!
- Dieser Fall kann also nicht dazu geführt haben, dass zum Zeitpunkt t beide Prozesse in ihren kritischen Regionen sind

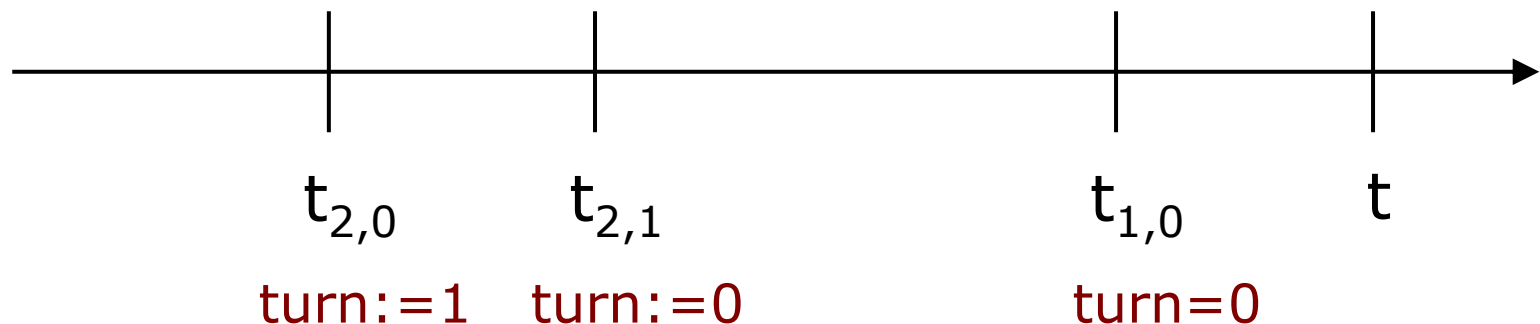
Beweis durch Widerspruch

- **Fall 2:** Verlassen der Schleife von Prozess 0 durch $turn=0$ zum Zeitpunkt $t_{1,0}$



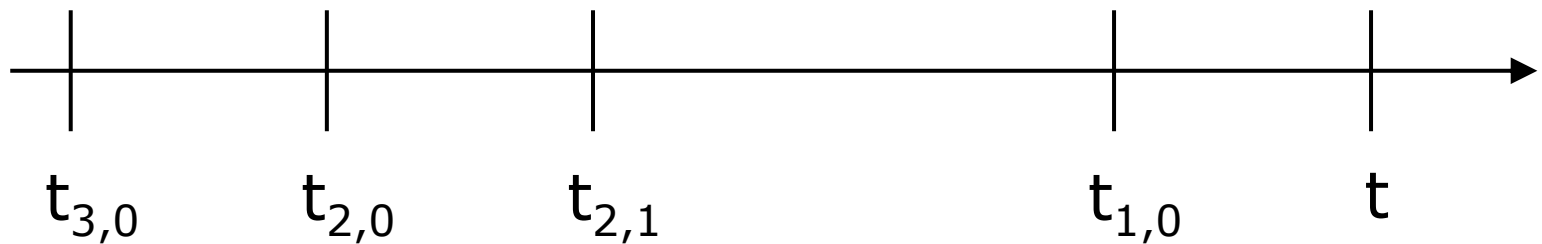
Beweis durch Widerspruch

- **Fall 2:** Verlassen der Schleife von Prozess 0 durch $turn=0$ zum Zeitpunkt $t_{1,0}$
- Zur Zeit $t_{2,0}$ wurde $turn:=1$ gesetzt in Prozess 0
- Zur Zeit $t_{2,1}$ wurde $turn:=0$ gesetzt in Prozess 1
- Also muss gelten: $t_{2,0} < t_{2,1} < t_{1,0}$



Beweis durch Widerspruch

- Zum Zeitpunkt $t_{3,0}$ wurde $\text{flag}[0] := \text{true}$ gesetzt



$\text{flag}[0] := \text{true}$

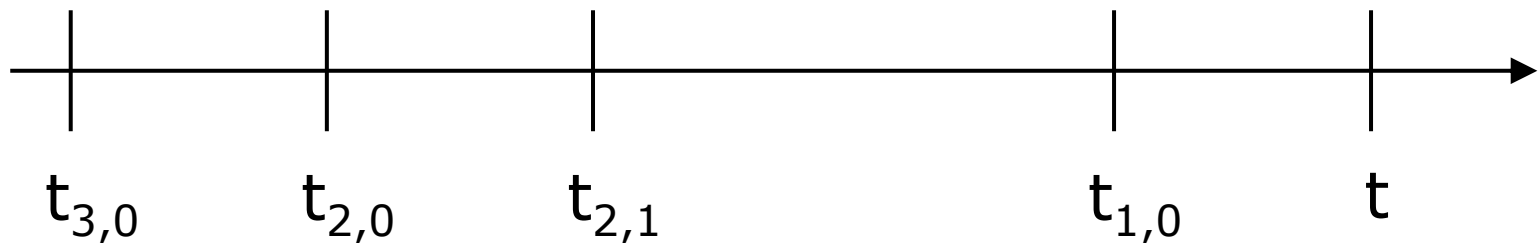
$\text{turn} := 1$

$\text{turn} := 0$

$\text{turn} = 0$

Beweis durch Widerspruch

- Zum Zeitpunkt $t_{3,0}$ wurde $\text{flag}[0] := \text{true}$ gesetzt
- Zwischen $t_{3,0}$ 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



$\text{flag}[0] := \text{true}$

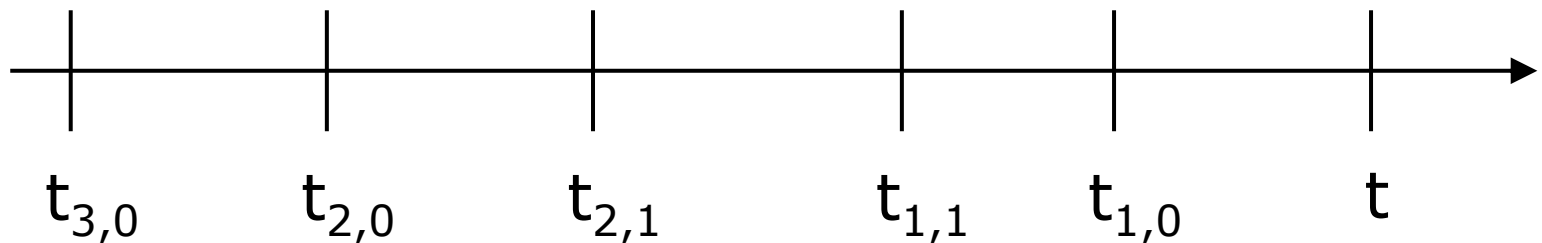
$\text{turn} := 1$

$\text{turn} := 0$

$\text{turn} = 0$

Beweis durch Widerspruch

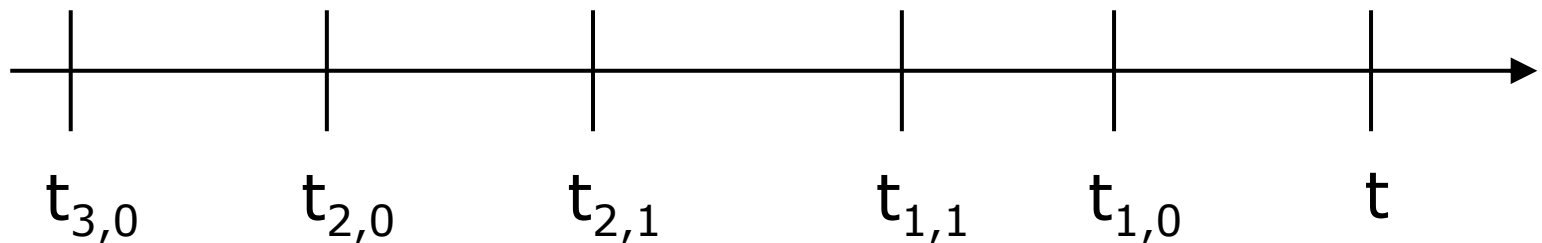
- Zum Zeitpunkt $t_{3,0}$ wurde $\text{flag}[0] := \text{true}$ gesetzt
- Zwischen $t_{3,0}$ und t wird $\text{flag}[0] := \text{false}$ nicht ausgeführt (entspr. Annahme), weil diese Anweisung nur bei der Initialisierung und **nach** der kritischen Region von Prozess 0 erfolgt
- Also muss zum Zeitpunkt $t_{1,1}$ gelten:



$\text{flag}[0] := \text{true}$ $\text{turn} := 1$ $\text{turn} := 0$ $\text{turn} = 0$ $\text{turn} = 0$
und $\text{flag}[0] = \text{true}$

Beweis durch Widerspruch

- Zum Zeitpunkt $t_{1,1}$ ist $\text{flag}[0]=\text{true}$ und $\text{turn}=0$!
- Also: Prozess 1 kann zum Zeitpunkt $t_{1,1}$ 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



$\text{flag}[0] := \text{true}$

$\text{turn} := 1$

$\text{turn} := 0$

$\text{turn} = 0$

$\text{turn} = 0$

und $\text{flag}[0] = \text{true}$

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;  
  /* nicht-kritische Region */  
}
```

```
/* Prozess 1 */  
wiederhole  
{  
  flag[1] := true;  
  turn := 0;  
  solange (flag[0] = true  
           und turn = 0)  
    tue nichts;  
  /* kritische Region */  
  flag[1] := false;  
  /* nicht-kritische 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, turn ist 1
- Prozess 1 nutzt die kritische Region mehrfach ohne Rücksicht auf Prozess 0:
Kann nicht passieren, da Prozess 1 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

Organisatorisches

- Raumänderung am **16.12.2015:**
Kinohörsaal (HS 00-006, Gebäude 082)
- Vorlesung vor Weihnachten (23.12.)
 - Option 1: Normale Vorlesung
 - Option 2: Fragestunde
 - Option 3: Vorlesung fällt aus

Zur Erinnerung: Krit. Regionen und Anforderungen an Lösungen

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

Zur Erinnerung: Algorithmus von Peterson

- Ein Flag pro Prozess, um Interesse zu bekunden
- Turn-Variable, um Vorrang festzulegen
- Korrekte Lösung (erfüllt die Anforderungen)
- aber aktives Warten

Heutige Vorlesung

- Hardware-Unterstützung für *atomare* Operationen
- Betriebssystem-Unterstützung für kritische Regionen
 - Mutex
 - Semaphoren
- Produzenten-/Konsumenten-Problem

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;  
    /* nicht-kritische Region */  
}
```

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

Wechselseitiger Ausschluss in Hardware (1)

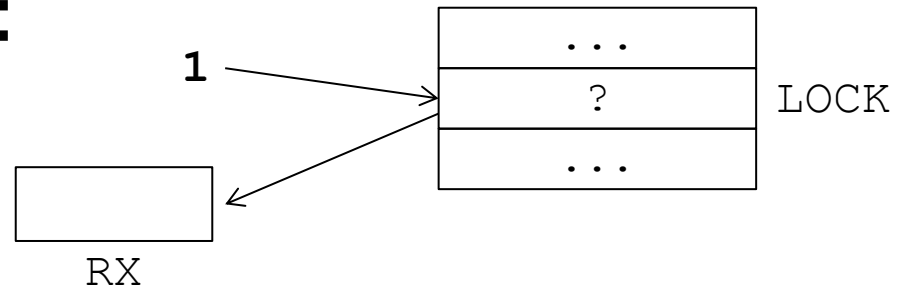
- Neues Konzept:
 - **Atomare** Operationen
 - **Hardware** garantiert atomare Ausführung
- Testen und Setzen zusammen bilden eine atomare Operation:
 - Befehl **TSL** (**T**est and **S**et **L**ock)
 - zwischen Testen und Setzen wird verhindert, d.
 - Prozesswechsel erfolgt
 - anderer Prozessor auf den Speicher zugreift

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)

Maschinenbefehl **TSL**:
führt atomar beide
Pfeile aus

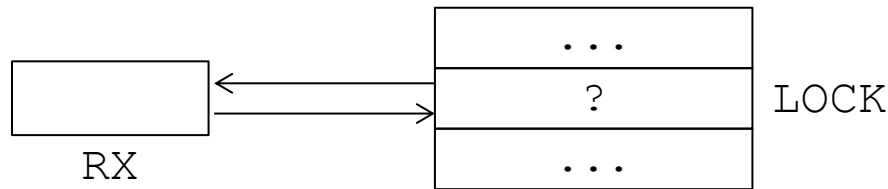


Prozesse, die Zugriff auf die kritische Region erhalten wollen, führen folgende Befehle aus:

```
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, betrete 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, betrete 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
```

- der nächste Prozess kann die kritische Region betreten
 - bei der nächsten **TSL/XCHG**-Operation wird eine 0 statt einer 1 in **RX** geschrieben

Wechselseitiger Ausschluss in Hardware: Analyse

- **Vorteile:**
 - Bei beliebiger Anzahl von Prozessen und sowohl Ein- wie auch Mehrkernsystemen anwendbar
 - Kann für mehrere kritische 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 aktivem Warten

- 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

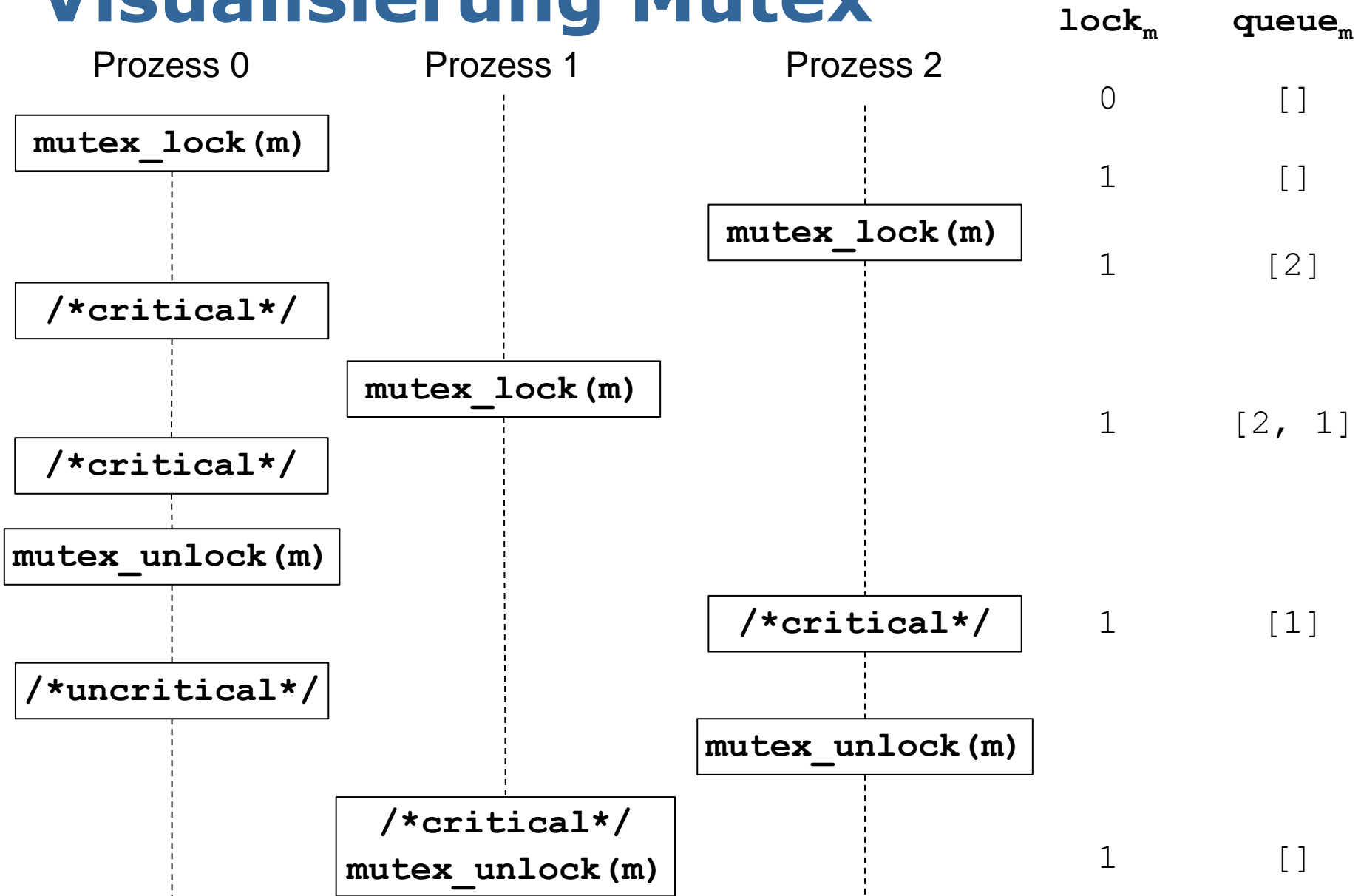
Mutex

- Synchronisations-Mechanismus, um wechselseitigen Ausschluss zu erzwingen
- Zu einem Mutex m gehören:
 - binäre Lock-Variable lock_m
 - Warteschlange queue_m
 - eindeutige id_m
- Ein Mutex besitzt zwei Operationen
 - `mutex_lock(m)`
 - `mutex_unlock(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

Visualisierung Mutex



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

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

Nur zur Veranschaulichung;
in Wirklichkeit TSL/XCHG-Befehl

Mutex: Warteschlange

- `sleep(idm)`: Prozess wird vom Betriebssystem in die Warteschlange des Mutex mit `idm` eingefügt
- `queuem`: Warteschlange für alle Prozesse, die auf den Eintritt in kritische Region mit der Nummer `idm` 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 m gibt es einen Betriebssystemaufruf, welcher die zugehörige Variable lock_m reserviert
- Alle Aufrufe von mutex_lock und mutex_unlock für die entsprechende kritische Region benutzen intern die gleiche id_m als Parameter für sleep und wakeup

Mutexe: Zusammenfassung

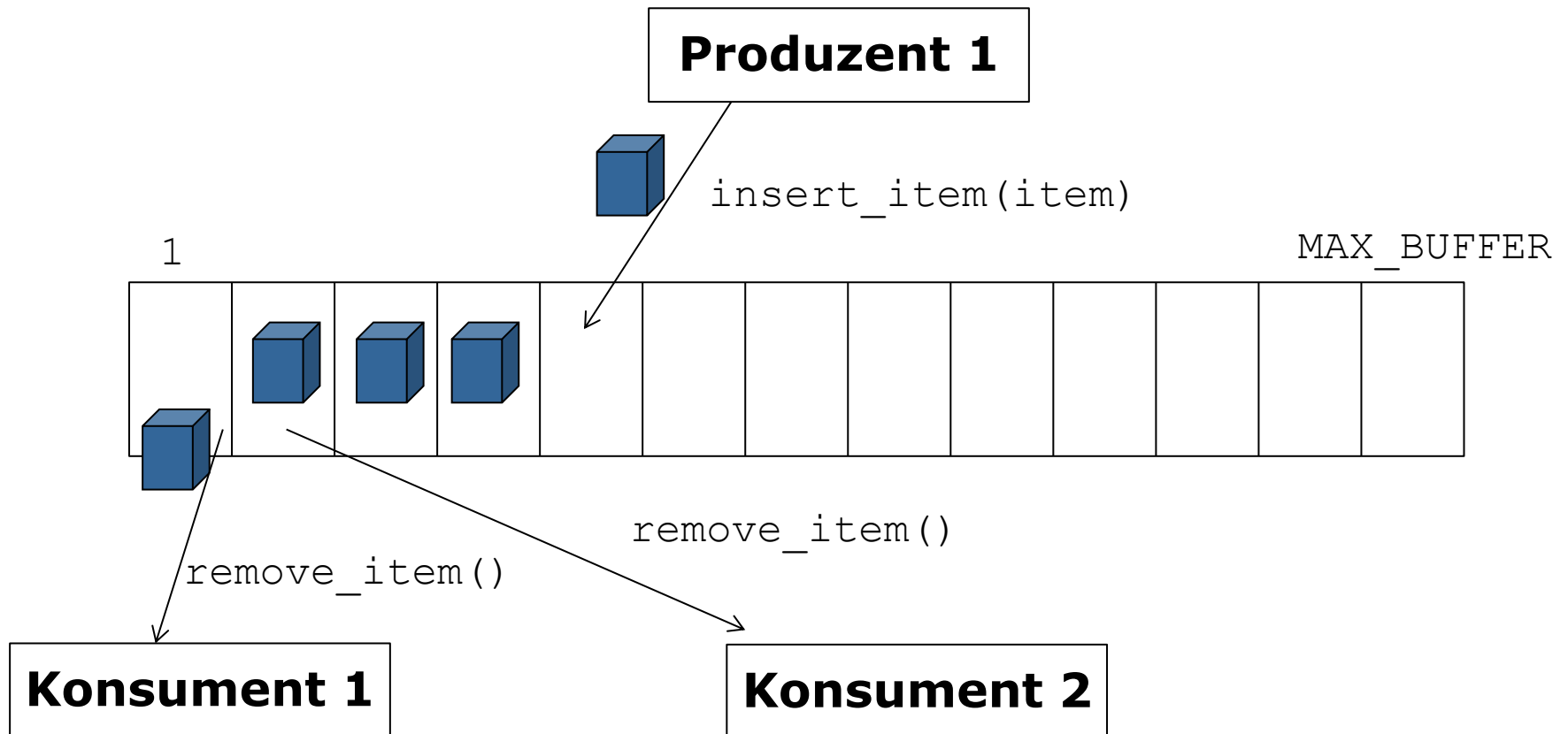
- Einfachste Möglichkeit um wechselseitigen Ausschluss mit Systemfunktionen zu garantieren
- Zwei Zustände: Mutex ist gesperrt / nicht gesperrt
- Kein aktives Warten von Prozessen, Prozesse sind blockiert
- Der Prozess, der den Mutex gesperrt hat, gibt ihn auch wieder frei

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()`

Illustration Produzenten-Konsumenten

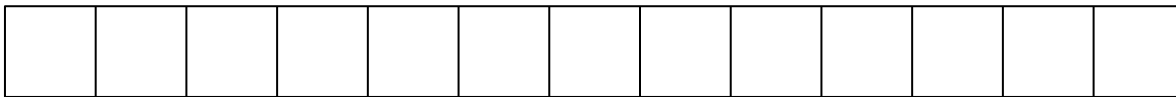


Das Produzenten-Konsumenten-Problem (2)

- 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**

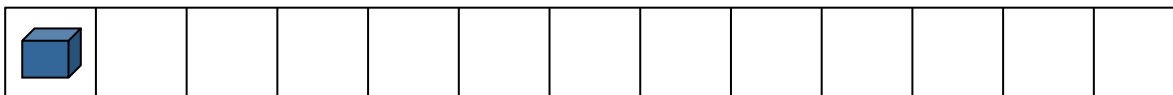
```
{
  ...
  wiederhole
  {
    wenn (count = 0)           // schlafe, wenn Puffer leer
      sleep(consumer_id);
    item = remove_item();     // entferne Objekt aus Puffer
    count = count - 1;
    wenn (count = MAX_BUFFER - 1) // Puffer vorher voll
      wakeup(producer_id);   // wecke Erzeuger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

Fehlersituation bei leerem Puffer


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) // Puffer vorher voll
      wakeup(producer_id);   // wecke Erzeuger
    consume_item(item);       // verbrauche Objekt
  }
}
```

Ist diese Lösung korrekt?



Fehlersituation bei leerem Puffer

Buffer 

count 1

```
wenn (count = 0)  
    sleep(consumer_id)
```

```
item = remove_item()
```

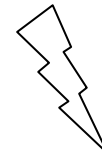
```
count = count - 1
```



```
wenn (count = 0)  
    sleep(consumer_id)
```

```
item = remove_item()
```

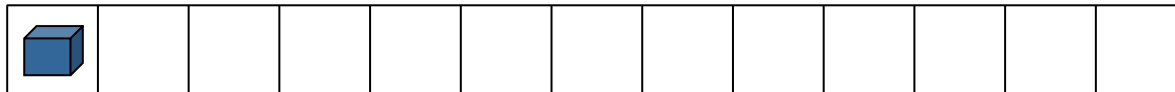
```
count = count - 1
```



Das Produzenten-Konsumenten-Problem – Fehlersituation (1)

1. Fehlersituation mit zwei Verbrauchern:

- 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!

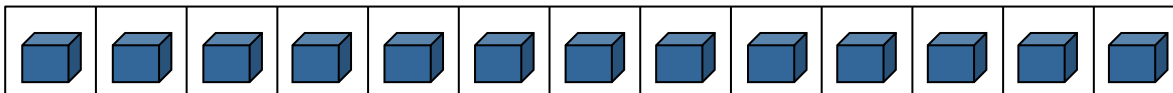


Fehlersituation bei vollem Puffer

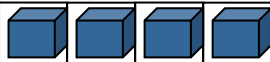
Prozedur **consumer**

```
{
  ...
  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) // Puffer vorher voll
      wakeup(producer_id);   // wecke Erzeuger
    consume_item(item);       // verbrauche Objekt
  }
}
```

Ist diese Lösung korrekt?



Fehlersituation bei vollem Puffer

Buffer 

count 4 3 2

```
wenn (count = 0)  
    sleep(consumer_id)
```

```
item = remove_item()
```

```
count = count - 1
```

```
wenn (count=MAX_BUFFER-1)  
    wakeup(producer_id);
```



```
wenn (count = 0)  
    sleep(consumer_id)
```

```
item = remove_item()
```

```
count = count - 1
```

```
wenn (count=MAX_BUFFER-1)  
    wakeup(producer_id);
```

Produzent wird nicht aufgeweckt

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) // 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

Elegante Lösung: Semaphor

- **Semaphor**: Datenstruktur zur Verwaltung beschränkter Ressourcen (Dijkstra, 1965)
- Ein Semaphor s hat drei Komponenten:
 - Integer-Variable count_s , repräsentiert Kapazität bzw. Zahl der ausstehenden Weckrufe
 - Warteschlange queue_s
 - Lock-Variable lock_s

Wert eines Semaphors

Drei mögliche Situationen für den Wert count_s eines Semaphors:

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

Semaphor: down/up Operationen

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

Wechselseitiger Ausschluss mit Semaphoren (1)

- Annahme: n Prozesse sind gestartet, konkurrieren um kritischen Abschnitt
- `counts` ist initialisiert mit 1

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

Wechselseitiger Ausschluss mit Semaphoren (2)

- 1. Prozess will in kritische Region

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

Wechselseitiger Ausschluss mit Semaphoren (2)

- 1. Prozess will in kritische Region
- `down(s)` wird ausgeführt: `counts = 0`

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

Wechselseitiger Ausschluss mit Semaphoren (2)

- 1. Prozess will in kritische Region
- `down(s)` wird ausgeführt: `counts=0`
- 1. Prozess muss nicht blockieren, betritt kritische Region

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

Wechselseitiger Ausschluss mit Semaphoren (3)

- Nun: 2. Prozess will in kritische Region

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

Wechselseitiger Ausschluss mit Semaphoren (3)

- Nun: 2. Prozess will in kritische Region
- `down(s)` wird ausgeführt: `counts = -1`

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


Wechselseitiger Ausschluss mit Semaphoren (3)

- Nun: 2. Prozess will in kritische Region
- `down(s)` wird ausgeführt: `counts = -1`
- 2. Prozess wird schlafengelegt + eingefügt in `queues`

```
/* Prozess i */  
wiederhole  
{  
→ down(s);  
  /* kritische Region */;  
  up(s);  
  /* nicht-kritische 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 queue_s
- Analog für 3. Prozess: `down(s)` führt zu $\text{count}_s = -2$, Prozess wird schlafengelegt + eingefügt in queue_s

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

Wechselseitiger Ausschluss mit Semaphoren (4)

- 1. Prozess verlässt irgendwann kritische Region


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

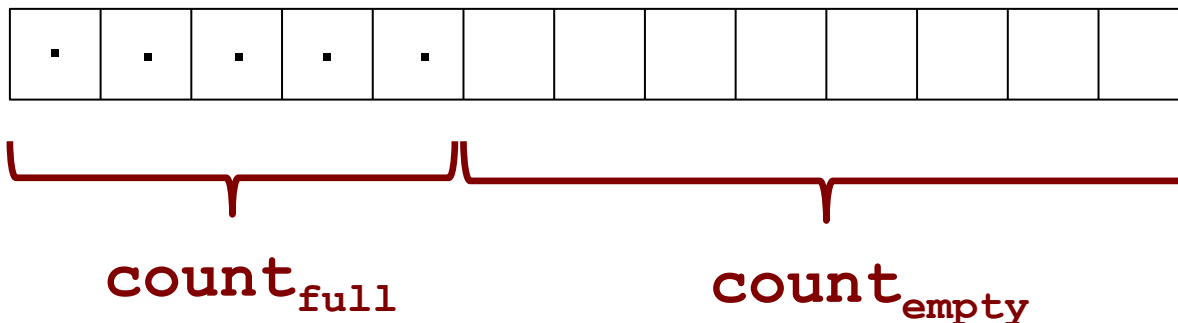

Semaphoren für Produzenten-Konsumenten-Problem

- Auf 1 initialisierte Semaphoren heißen *binäre Semaphoren* (vgl. Mutex)
- Behandlung mehrfach nutzbarer Ressourcen möglich durch Initialisierung: $\text{count}_s = m, m > 1$
- Wenn $\text{count}_s \geq 0$:
 - | count_s | entspr. Anzahl der Prozesse, die $\text{down}(s)$ ausführen können ohne zu blockieren (ohne zwischenzeitliches $\text{up}(s)$)
- $\text{count}_s < 0$, dann gilt:
 - | count_s | Anzahl der wartenden Prozesse in queue_s

Produzenten-Konsumenten-Problem mit Semaphoren (1)

Drei verschiedene Semaphore werden benötigt:

- `exclu`: für wechselseitigen Ausschluss
- `empty`: zählt freie Plätze
- `full`: zählt belegte Plätze



Produzenten-Konsumenten-Problem mit Semaphoren (2)

- Puffer ist anfangs leer, kein Prozess in kritischer Region:
 - $\text{count}_{\text{empty}} = \text{MAX_BUFFER}$
 - $\text{count}_{\text{full}} = 0$
 - $\text{count}_{\text{exclu}} = 1$

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 `down(full)` aus; wenn $\text{count}_{\text{full}} < 0$: Blockiere
 - Immer wenn etwas hinzugefügt werden soll: führe `down(empty)` aus; wenn $\text{count}_{\text{empty}} < 0$: blockiere
 - In `up` werden evtl. schlafende Prozesse geweckt

Produzent mit Semaphoren (1)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen
semaphore empty; count_empty = 4; // zählt freie Plätze
semaphore full; count_full = 0; // zählt belegte Plätze
```

Buffer

--	--	--	--

Prozedur **producer**

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

Produzent mit Semaphoren (2)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen  
semaphore empty; count_empty = 3; // zählt freie Plätze  
semaphore full; count_full = 0; // zählt belegte Plätze
```

Buffer


--	--	--	--

Prozedur **producer**

```
{  
  wiederhole versuche, kritische Region zu betreten  
  {  
    item = produce_item(); // produziere nächstes Objekt  
    down(empty);  
    → down(exclu);  
    insert_item(item); // füge Objekt in Puffer ein  
    up(exclu);  
    up(full);  
  }  
}
```

Produzent mit Semaphoren (3)

```
semaphore exclu; count_exclu = 0; // "mutex"-Sem. für krit. Regionen  
semaphore empty; count_empty = 3; // zählt freie Plätze  
semaphore full; count_full = 0; // zählt belegte Plätze
```


Buffer 

Prozedur **producer**

```
{  
  wiederhole Objekt in Puffer, kritische Region freigeben  
  {  
    item = produce_item(); // produziere nächstes Objekt  
    down(empty);  
    down(exclu);  
    insert_item(item); // füge Objekt in Puffer ein  
    → up(exclu);  
    up(full);  
  }  
}
```

Produzent mit Semaphoren (4)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen  
semaphore empty; count_empty = 3; // zählt freie Plätze  
semaphore full; count_full = 0; // zählt belegte Plätze
```

Buffer 


Prozedur **producer**

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

Belegte Plätze aktualisieren,
ggf. blockierte Konsumenten aufwecken

Produzent mit Semaphoren (5)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen
semaphore empty; count_empty = 3; // zählt freie Plätze
semaphore full; count_full = 1; // zählt belegte Plätze
```

Buffer 


Prozedur **producer**

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

→

Konsument mit Semaphoren (1)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen
semaphore empty; count_empty = 3; // zählt freie Plätze
semaphore full; count_full = 1; // zählt belegte Plätze
```

Buffer 


Prozedur **consumer**

```
{
  wiederhole
  {
    → down(full);
    down(exclu);
    item = remove_item(); // entferne Objekt aus Puffer
    up(exclu);
    up(empty);
    consume_item(item); // konsumiere Objekt
  }
}
```

zähle runter; überprüfe, ob ≥ 1 Platz belegt; ggf. sleep

Konsument mit Semaphoren (2)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen
semaphore empty; count_empty = 3; // zählt freie Plätze
semaphore full; count_full = 0; // zählt belegte Plätze
```

Buffer 

Prozedur **consumer**

```
{
  wiederhole
  {
    down(full);
    → down(exclu);
    item = remove_item(); // entferne Objekt aus Puffer
    up(exclu);
    up(empty);
    consume_item(item); // konsumiere Objekt
  }
}
```

versuche, kritische Region zu betreten

Konsument mit Semaphoren (3)

```
semaphore exclu; count_exclu = 0; // "mutex"-Sem. für krit. Regionen
semaphore empty; count_empty = 3; // zählt freie Plätze
semaphore full; count_full = 0; // zählt belegte Plätze
```

Buffer

--	--	--	--

Prozedur **consumer**

```
{
  wiederhole
  {
    down(full);
    down(exclu);
    → item = remove_item(); // entferne Objekt aus Puffer
    up(exclu);
    up(empty);
    consume_item(item); // konsumiere Objekt
  }
}
```

Entferne Objekt, gib kritische Region frei

Konsument mit Semaphoren (4)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen
semaphore empty; count_empty = 3; // zählt freie Plätze
semaphore full; count_full = 0; // zählt belegte Plätze
```

Buffer

--	--	--	--

Prozedur **consumer**

```
{
  wiederhole
  {
    down(full);
    down(exclu);
    item = remove_item(); // entferne Objekt aus Puffer
    up(exclu);
    → up(empty);
    consume_item(item); // konsumiere Objekt
  }
}
```

Signalisiere neuen freien Platz,
ggf. Produzenten aufwecken

Konsument mit Semaphoren (5)

```
semaphore exclu; count_exclu = 1; // "mutex"-Sem. für krit. Regionen
semaphore empty; count_empty = 4; // zählt freie Plätze
semaphore full; count_full = 0; // zählt belegte Plätze
```

Buffer

--	--	--	--

Prozedur **consumer**

```
{
  wiederhole
  {
    down(full);
    down(exclu);
    item = remove_item(); // entferne Objekt aus Puffer
    up(exclu);
    up(empty);
    → consume_item(item); // konsumiere Objekt
  }
}
```

Produzenten-Konsumenten-Problem mit Semaphoren

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

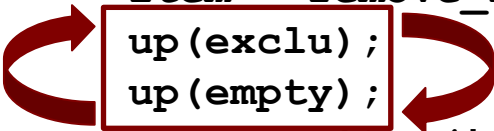
Reihenfolge der Operationen (1)

Frage: Funktioniert das immer noch, wenn in
Prozedur consumer

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

Prozedur **consumer**

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



ja!

Reihenfolge der Operationen (2)

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(exclu)` erst später auf, erst dann wird Produzent wieder aufgeweckt

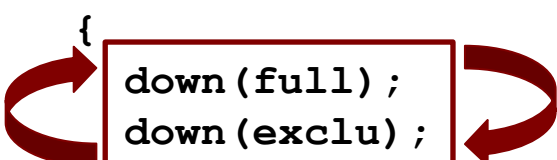
Reihenfolge der Operationen (3)

Frage: Funktioniert das immer noch, wenn in
Prozedur consumer

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

Prozedur **consumer**

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



nein!

Problem bei vertauschter Reihenfolge (1)

Konsument 1

→ `down(exclu);`
`down(full);`

```
count_exclu = 1
count_empty = 4
count_full = 0
```

`down(exclu);`
→ `down(full);`

```
count_exclu = 0
count_empty = 4
count_full = 0
```

`down(exclu);`
→ `down(full);`

```
count_exclu = 0
count_empty = 3
count_full = 0
```

Produzent 1

→ `down(empty);`
`down(exclu);`

→ `down(empty);`
`down(exclu);`

`down(empty);`
→ `down(exclu);`

→ Deadlock

Problem bei vertauschter Reihenfolge (2)

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**
- **mutex_s** innerhalb des Semaphors, Pseudocode:

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

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

Implementierung von Semaphoren: Analyse Versuch 1

- **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: Analyse Versuch 2

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

Mutex und Semaphor

- Mutex stets für wechselseitigen Ausschluss

```
mutex_lock(m)  
...  
mutex_unlock(m)
```

P0

- Semaphor (u.a.) zwei Anwendungen
 - Wechselseitiger Ausschluss (binäre Semaphore)

```
down(exclu);  
...  
up(exclu);
```

P0

- Signalisierung (z.B. Kapazität, ggf. auch binär)

```
up(empty);
```

P0

```
down(empty);
```

P1

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