Systeme I:
Betriebssysteme

Kapitel 5
Nebenläufigkeit und wechselseitiger Ausschluss

Maren Bennewitz

Version 12.12.2012

Nachtrag zu letzter Vorlesung

- Hauptspeicher reicht nur für begrenzte Anzahl von Prozessen
- Swapping: Prozesse werden auf der Festplatte ausgelagert (z.B. wartende Prozesse)
- Konzept des virtuellen Speichers: Programme können auch laufen, wenn sich nur ein Teil von ihnen im Hauptspeicher befindet

Nachtrag zu letzter Vorlesung

- Betriebssystem stellt jedem Prozess einen Adressraum zur Verfügung
- Virtueller Speicher eines Prozesses: Sein Hauptspeicherbereich + Bereiche in einer Auslagerungsdatei auf der Festplatte
- Virtuelle Adresse: Ein Ort im Speicher
- Betriebssystem verwaltet Speicher und Zugriff (Paging, Kapitel 8)

Nachtrag zu letzter Vorlesung

- Scheduling: Zuweisung von CPU-Zeit (Kapitel 7)
- Welcher Prozess wird wann ausgeführt (für Ein- und Mehrprozessorsysteme)

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 Prozesses können parallel auf mehreren Prozessoren ausgeführt werden
- Sie können aber auch „pseudo-parallel" auf einem Prozessor ausgeführt werden (siehe letztes Kapitel)

Nebenläufigkeit

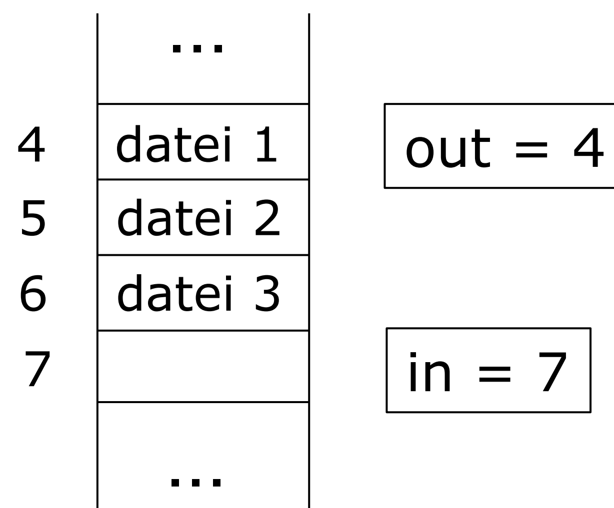
- Betriebssystem muss Ressourcen der aktiven Prozesse verwalten
- Bei Zugriff auf gemeinsame Ressourcen muss wechselseitiger Ausschluss garantiert werden
- Die Korrektheit des Ergebnisses muss unabhängig von der Ausführungsgeschwindigkeit sein

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 in einer Datei zugänglich für alle Prozesse:
  - *out*: Nummer der nächsten zu druckenden Datei
  - *in*: Nummer des nächsten freien Eintrags

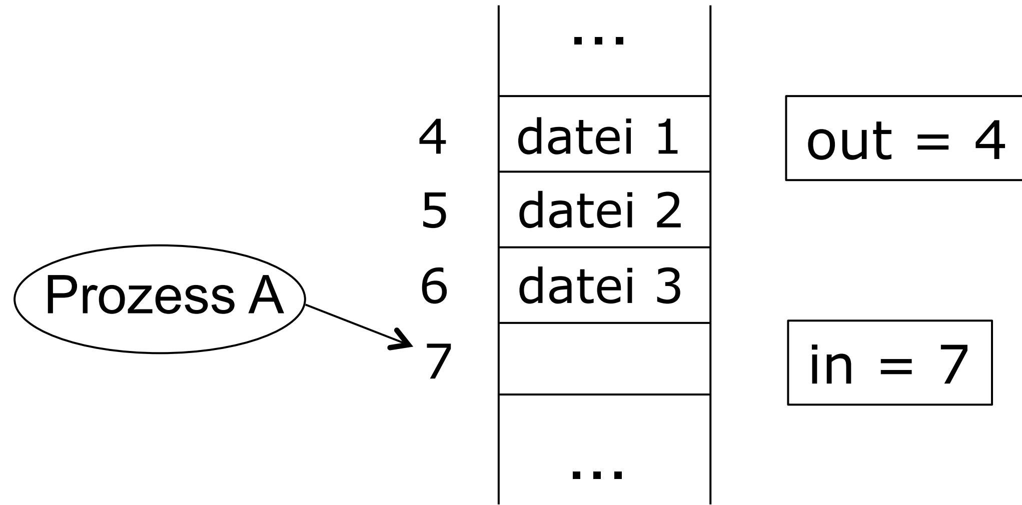
Beispiel: Druckerpuffer

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



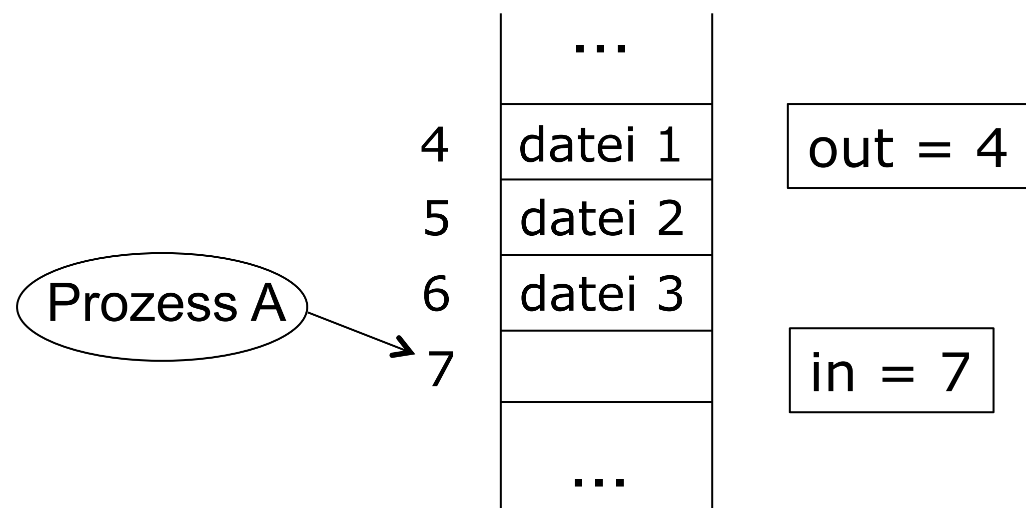
Beispiel: Druckerpuffer

- Prozess A liest *in* aus und speichert 7



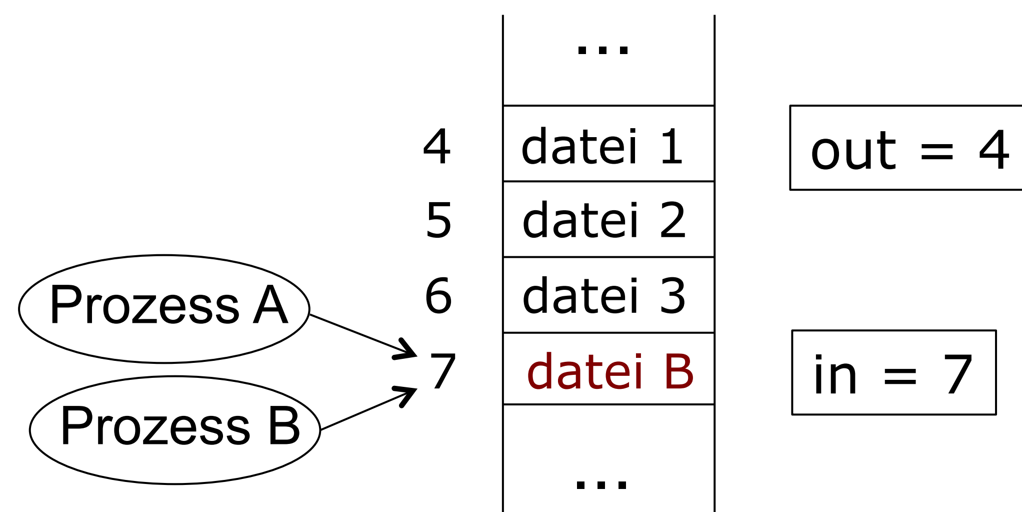
Beispiel: Druckerpuffer

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



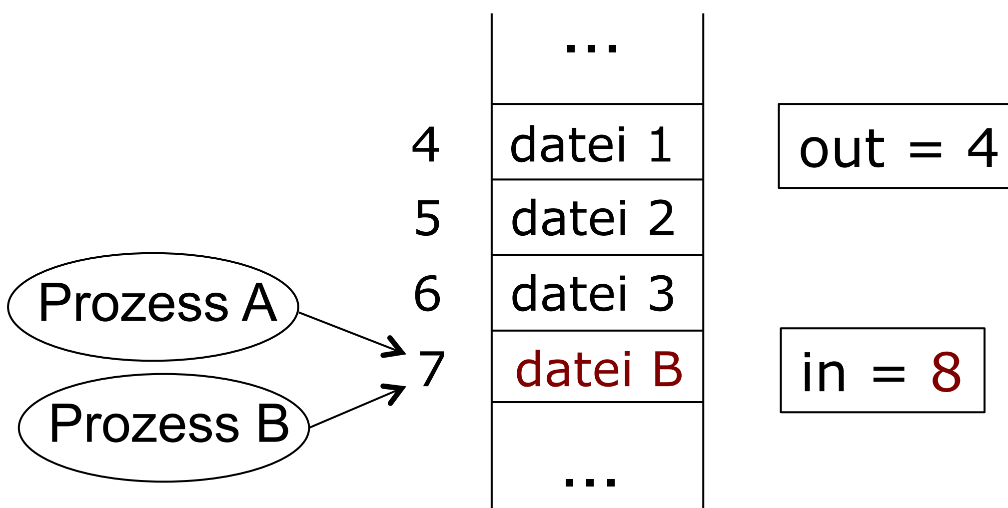
Beispiel: Druckerpuffer

- 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



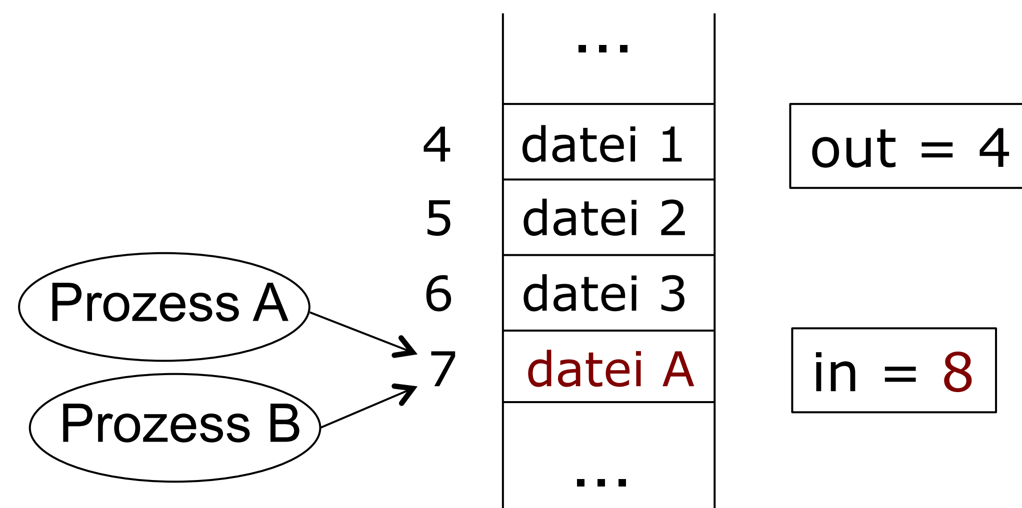
Beispiel: Druckerpuffer

- 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 diesen als freien Slot gemerkt hatte
- Dateiname von Prozess B wird überschrieben!



Critical Regions (1)

- „Race Conditions“:
  - Zwei oder mehr Prozesse lesen oder beschreiben gleichen gemeinsamen Speicher
  - Endergebnis hängt davon ab, wer wann läuft
- Benötigt: Wechselseitiger Ausschluss von Prozessen („mutual exclusion“)
- Verbiete Prozessen gleichzeitig mit einem anderen die gemeinsam genutzten Daten zu lesen oder zu beschreiben

Critical Regions (2)

- Kritische Regionen: Teile des Programms, in denen auf gemeinsam benutzten 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 einen effizienten Ablauf zu gewährleisten bei gemeinsam genutzten Daten

Anforderungen an wechselseitigen Ausschluss

- Keine zwei Prozesse dürfen gleichzeitig in ihren kritischen Regionen sein
- Es dürfen keine Annahmen über Geschwindigkeit und Anzahl der CPUs 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 Mehrprozessorsystemen nicht zum Erfolg
- Mehrere Prozesse werden gleichzeitig ausgeführt, wechselseitiger Ausschluss kann nicht garantiert werden

Lösungen für wechselseitigen Ausschluss

- Software-Lösungen: Verantwortlichkeit bei Prozessen, Anwendungsprogramme 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 seinen kritischen Bereich 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
  - Weiterer Prozess startet, setzt Sperre auf 1
  - 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
- Aktives Warten (Verschwendung von CPU Zeit)

```
/* 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 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 (schneller) 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 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 keine gute Idee!
- Prozess 0 wird von einem Prozess blockiert, ohne dass dieser in seiner kritischen Region ist (verletzt 3. Anforderung, Folie 17)

```
/* 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 */
 }
```

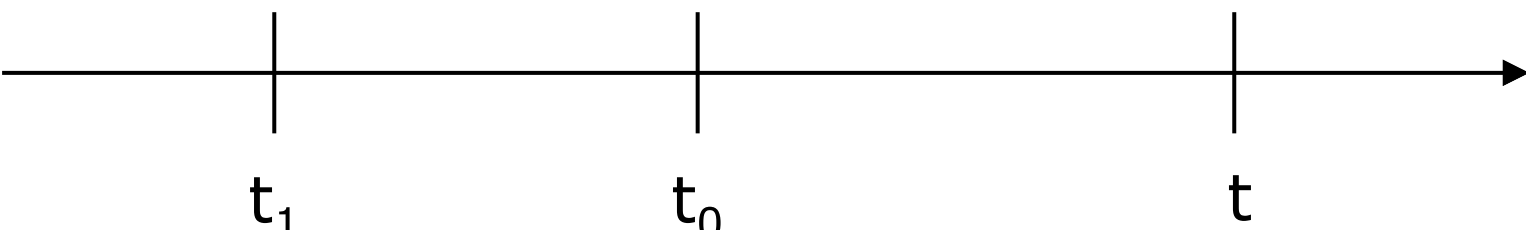
Wechselseitiger Ausschluss

Satz:

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

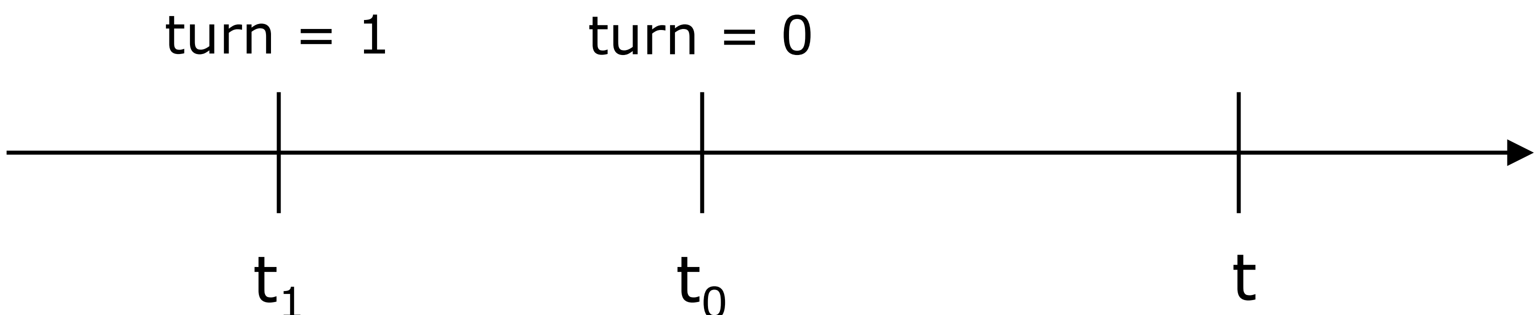
Beweis durch Widerspruch

- Annahme: Es gibt einen Zeitpunkt t, zu dem beide Prozesse in kritischen Regionen sind
- t_1: Der letzte Zeitpunkt < t, zu dem Prozess 1 die solange-Schleife verlassen hat und in seinen kritischen Bereich gegangen ist
- t_0: Der letzte Zeitpunkt < t, zu dem Prozess 0 die solange-Schleife verlassen hat und in seinen kritischen Bereich 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. Bereich
- Prozess 1 ist aber zwischen t_1 und t permanent in seinem kritischen Bereich
- *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. Bereich
- Prozess 1 ist aber zwischen t_1 und t permanent in seinem kritischen Bereich
- *turn* = 0 kann nicht zwischen t_1 und t ausgeführt worden sein!
- Prozess 0 kann nicht in kritischem Bereich sein
- Widerspruch zur Annahme!
- Es gibt also keinen Zeitpunkt t, zu dem beide Prozesse in ihren kritischem Bereichen sind

Versuch 1b: Analyse

- Vorteil: Wechselseitiger Ausschluss ist garantiert
- Nachteil: Aktives Warten („Busy waiting") führt zu Verschwendung von Rechenzeit
- Grund: Nur abwechselnder Zugriff auf kritischen Bereich
- Beispiel: Prozess 0 ist schnell, Prozess 1 hat einen sehr langen nicht kritischen Abschnitt

Feedback zur Vorlesung

- Fragebögen
- Bitte Vorder- und Rückseite ausfüllen
- Vielen Dank!

Nächster Versuch

- Vorher wurde der Name des Prozesses gespeichert, der an der Reihe ist
- Besser: Speichere Zustandsinformation der Prozesse (in seiner kritischer Region?)
- 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
- 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 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 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 Erfolg?

```
/* 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 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

- 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)

4. Versuch

- Nicht ausreichend: Status-Flag setzen ohne den Status anderer Prozesse zu überprüfen
- Füge hinzu: Möglichkeit, das Flag zurückzusetzen, um anderem Prozess Vorrang zu lassen
- „Nichtstun“ in Schleife wird ersetzt durch zeitweilige Zurücknahme der Anforderung
- Dadurch: Es ist anderen Prozessen erlaubt, in die kritische Region einzutreten

4. Versuch

- Führt das zu Erfolg?

```
/* Prozess 0 */
wiederhole
 {
   flag[0] := true;
   solange (flag[1] = true)
    {
      flag[0] := false;
      /* zufäll.Verzögerung. */;
      flag[0] := true;
    }
   /* kritische Region */
   flag[0] := false;
   /* nicht kritische Region */
 }
```

```
/* Prozess 1 */
wiederhole
 {
   flag[1] := true;
   solange (flag[0] = true)
    {
      flag[1] := false;
      /* zufäll.Verzögerung. */;
      flag[1] := true;
    }
   /* kritische Region */
   flag[1] := false;
   /* nicht kritische Region */
 }
```

4. Versuch

- Beide Prozesse setzen ihre Flags auf *true*

```
/* Prozess 0 */
wiederhole
 {
   flag[0] := true;
   solange (flag[1] = true)
    {
      flag[0] := false;
      /* zufäll.Verzögerung. */;
      flag[0] := true;
    }
   /* kritische Region */
   flag[0] := false;
   /* nicht kritische Region */
 }
```

```
/* Prozess 1 */
wiederhole
 {
   flag[1] := true;
   solange (flag[0] = true)
    {
      flag[1] := false;
      /* zufäll.Verzögerung. */;
      flag[1] := true;
    }
   /* kritische Region */
   flag[1] := false;
   /* nicht kritische Region */
 }
```

4. Versuch

- Beide Prozesse überprüfen das jeweils andere Flag

```
/* Prozess 0 */
wiederhole
 {
   flag[0] := true;
   solange (flag[1] = true)
    {
      flag[0] := false;
      /* zufäll.Verzögerung. */;
      flag[0] := true;
    }
   /* kritische Region */
   flag[0] := false;
   /* nicht kritische Region */
 }
```

```
/* Prozess 1 */
wiederhole
 {
   flag[1] := true;
   solange (flag[0] = true)
    {
      flag[1] := false;
      /* zufäll.Verzögerung. */;
      flag[1] := true;
    }
   /* kritische Region */
   flag[1] := false;
   /* nicht kritische Region */
 }
```

4. Versuch

- Beide Prozesse setzen ihre Flags wieder auf *false*

```
/* Prozess 0 */
wiederhole
 {
   flag[0] := true;
   solange (flag[1] = true)
    {
      flag[0] := false;
      /* zufäll.Verzögerung. */;
      flag[0] := true;
    }
   /* kritische Region */
   flag[0] := false;
   /* nicht kritische Region */
 }
```

```
/* Prozess 1 */
wiederhole
 {
   flag[1] := true;
   solange (flag[0] = true)
    {
      flag[1] := false;
      /* zufäll.Verzögerung. */;
      flag[1] := true;
    }
   /* kritische Region */
   flag[1] := false;
   /* nicht kritische Region */
 }
```

4. Versuch

- Beide Prozesse setzen ihre Flags dann wieder auf *true*

```
/* Prozess 0 */
wiederhole
 {
   flag[0] := true;
   solange (flag[1] = true)
    {
      flag[0] := false;
      /* zufäll.Verzögerung. */;
      flag[0] := true;
    }
   /* kritische Region */
   flag[0] := false;
   /* nicht kritische Region */
 }
```

```
/* Prozess 1 */
wiederhole
 {
   flag[1] := true;
   solange (flag[0] = true)
    {
      flag[1] := false;
      /* zufäll.Verzögerung. */;
      flag[1] := true;
    }
   /* kritische Region */
   flag[1] := false;
   /* nicht kritische Region */
 }
```

4. Versuch

- Wahrscheinlichkeit für Deadlock ist klein, wenn die zufälligen Verzögerungen streuen
- Aber es kann theoretisch sein, dass die zufälligen Verzögerungen so verteilt sind, dass die Schleifen nicht verlassen werden
- Keine absolute Garantie für Deadlock-Freiheit
- Dynamische Verklemmung („Livelock"): Es gibt Ausführungssequenzen, die erfolgreich ablaufen, aber keine Garantie

Versuch 4: Analyse

- Vorteile:
  - Auch nicht-alternierender Zugriff auf kritische Region ist erlaubt
  - Wegen zufälliger Verzögerung nur geringe Wahrscheinlichkeit für Deadlock
- Nachteile:
  - Wieder aktives Warten
  - Deadlock nach wie vor möglich!
  - Unsaubere Lösung!
  - Verhalten ist schlecht vorhersagbar

Versuch 5: Richtige Lösung

- Statusbeobachtung reicht nicht aus
- Kombiniere vorigen Ansatz mit einer Reihenfolge für beide Prozesse
- Gemeinsame Variablen:
  turn, flag[0], flag[1]
- Initialisierung:
  flag[0] := *false*; flag[1] := *false*; *turn*: beliebig
- Variable *turn* bestimmt, welcher Prozess auf seiner Anforderung bestehen darf
- Petersons Algorithmus, 1981

Versuch 5: Peterson-Algorithmus

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

/* Prozess 0 */	/* Prozess 1 */

wiederhole
{
flag[0] := true;
turn := 1;
solange (flag[1] = true
und turn = 1)
tue nichts;
/* kritische Region */
flag[0] := false;
/* nicht kritische Region */
}
	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 kritischen Regionen sind

Beweis durch Widerspruch

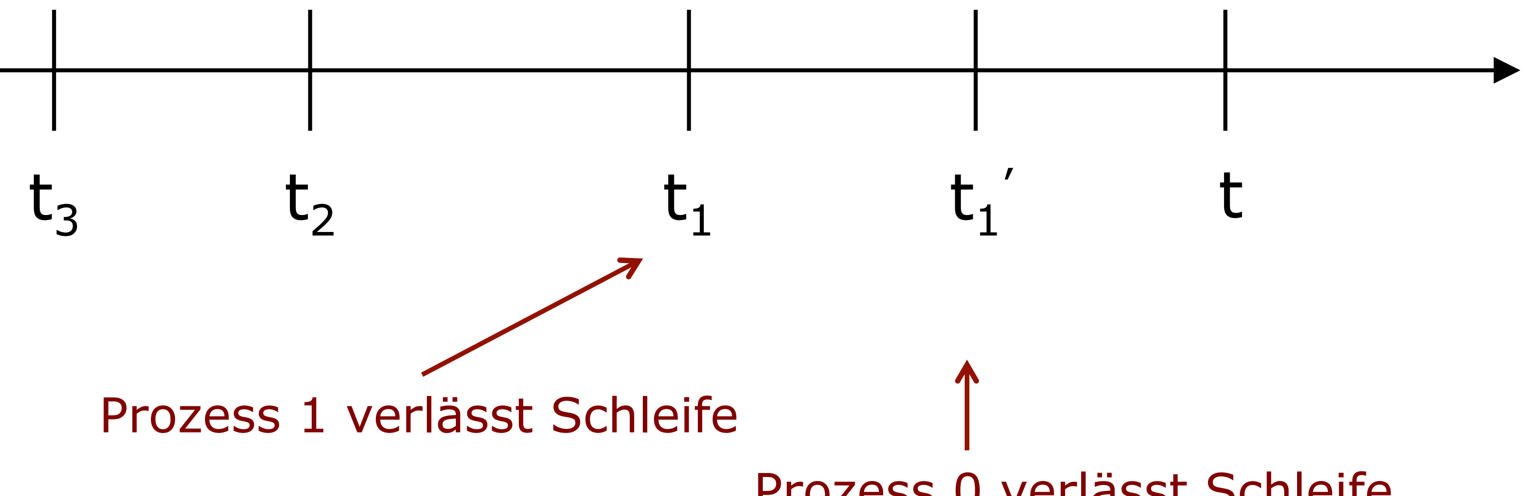
- t_1: Der letzte Zeitpunkt $<$ t, zu dem Prozess 1 die solange-Schleife verlässt
- t_1': Der letzte Zeitpunkt $<$ t, zu dem Prozess 0 die solange-Schleife verlässt
- t_2: Der letzte Zeitpunkt $<$ t, zu dem Prozess 1 *turn*:= 0 ausführt
- t_2': Der letzte Zeitpunkt $<$ t, zu dem Prozess 0 *turn*:= 1 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 */	/* Prozess 1 */
wiederhole	wiederhole
{	{
flag[0] := true; ← t_3'	flag[1] := true; ← t_3
turn := 1; ← t_2'	turn := 0; ← t_2
solange (flag[1] = true	solange (flag[0] = true
und turn = 1)	und turn = 0)
tue nichts; ← t_1'	tue nichts; ← t_1
/* kritische Region */ ← t	t → /* kritische Region */
flag[0] := false;	flag[1] := false;
/* nicht kritische Region */	/* nicht kritische Region */
}	}

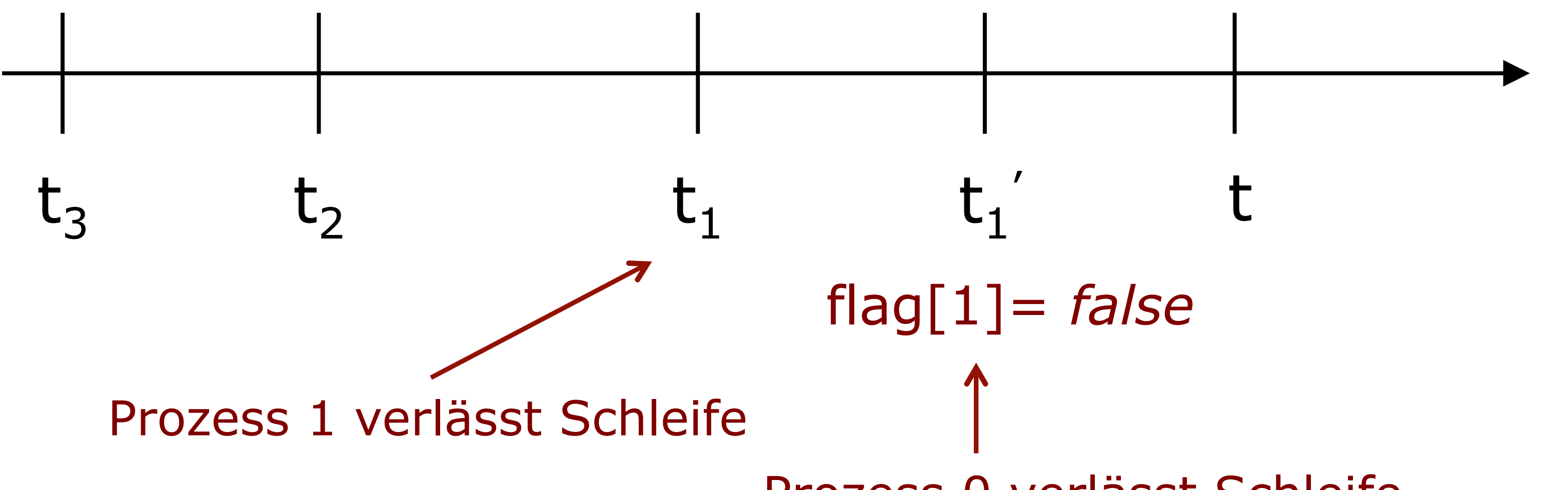
Beweis durch Widerspruch

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



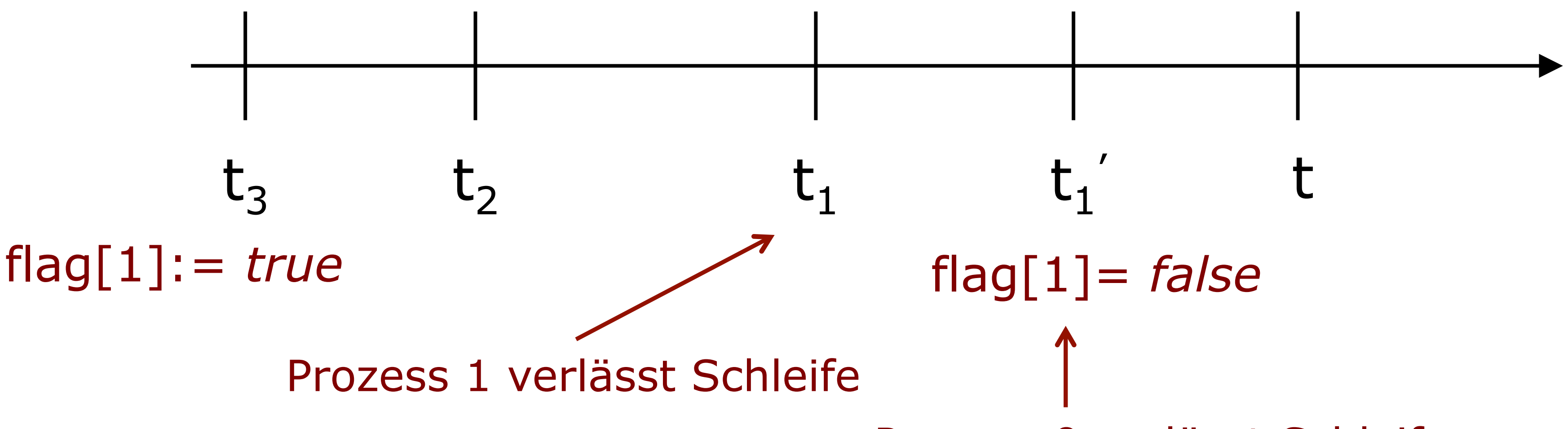
Beweis durch Widerspruch

- 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**: flag[1] = *false* zum Zeitpunkt t_1'



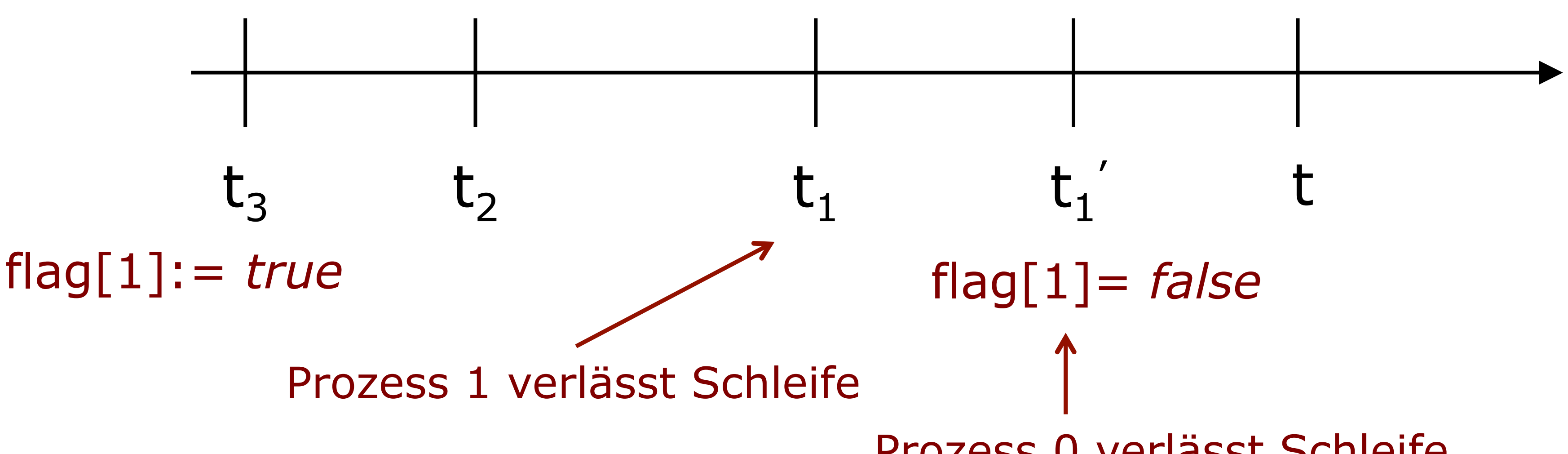
Beweis durch Widerspruch

- 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**: flag[1] = *false* zum Zeitpunkt t_1'
- Zum Zeitpunkt t_3 wurde flag[1]:=*true gesetzt*



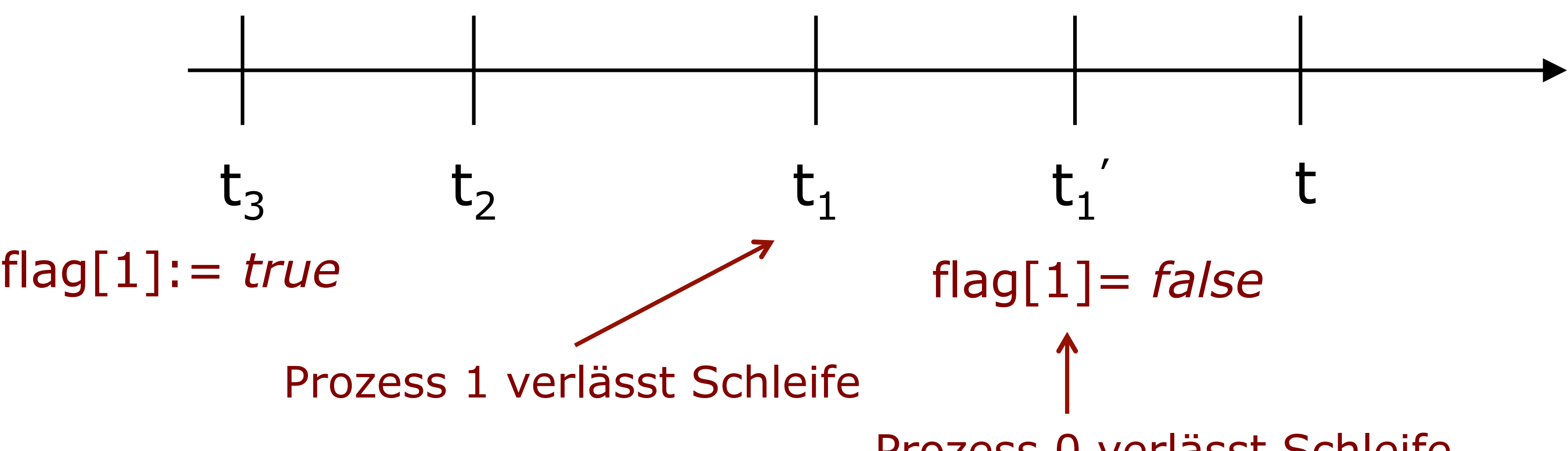
Beweis durch Widerspruch

- Es müsste zwischen t_3 und t_1' flag[1]:= *false* gesetzt worden sein



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)



Beweis durch Widerspruch

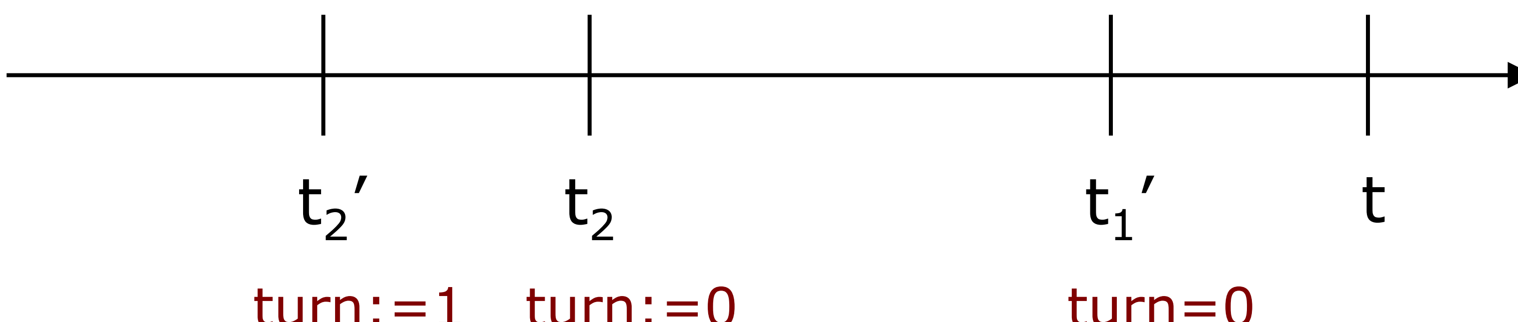
- Es müsste zwischen t_3 und t_1' flag[1]:= *false* gesetzt worden sein
- flag[1]:= *false* passiert aber nur bei der Initialisierung und nach der kritischen Region von Prozess 1
- Zwischen t_3 und t_1' erreicht Prozess 1 aber nicht das Ende der kritischen Region! (entspr. Annahme)
- Prozess 0 kann nicht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

Beweis durch Widerspruch

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

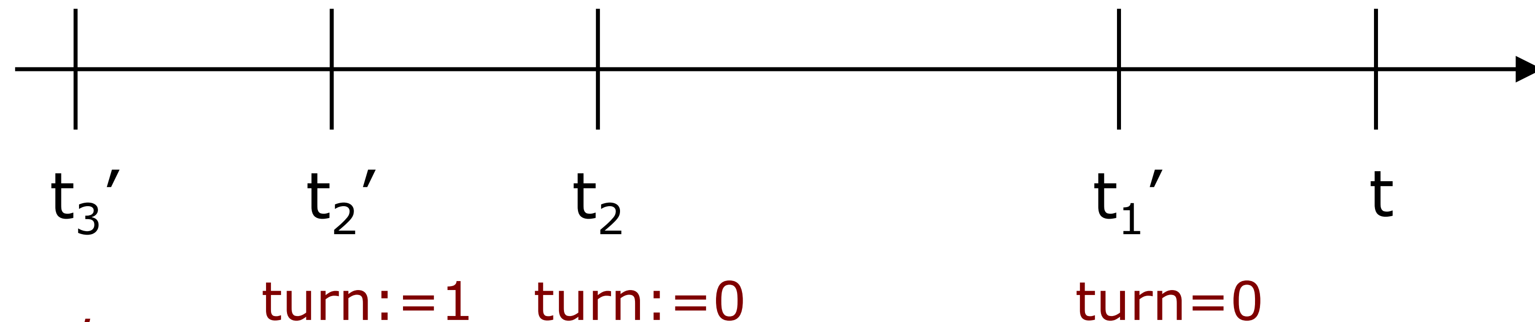
Beweis durch Widerspruch

- **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

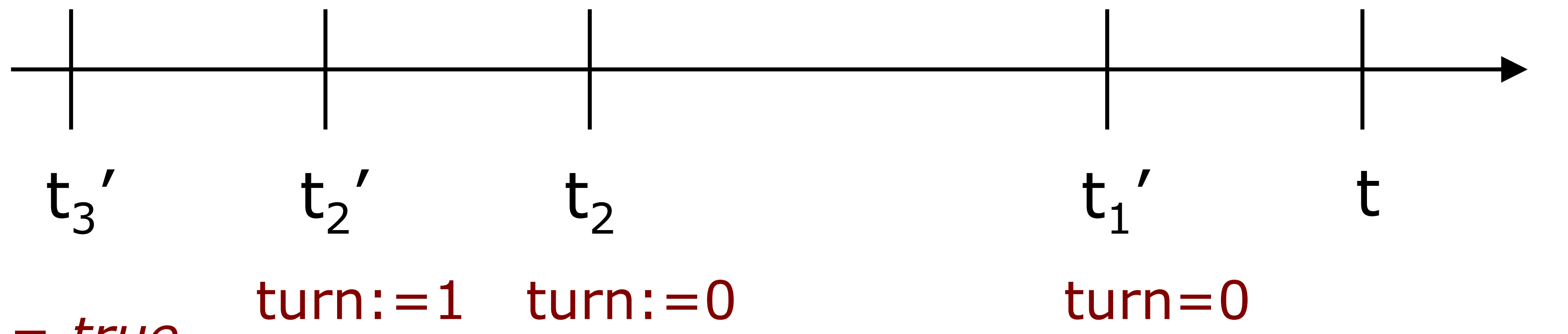
- Zum Zeitpunkt t_3’ wurde flag[0]:=*true gesetzt*



flag[0]:= *true*

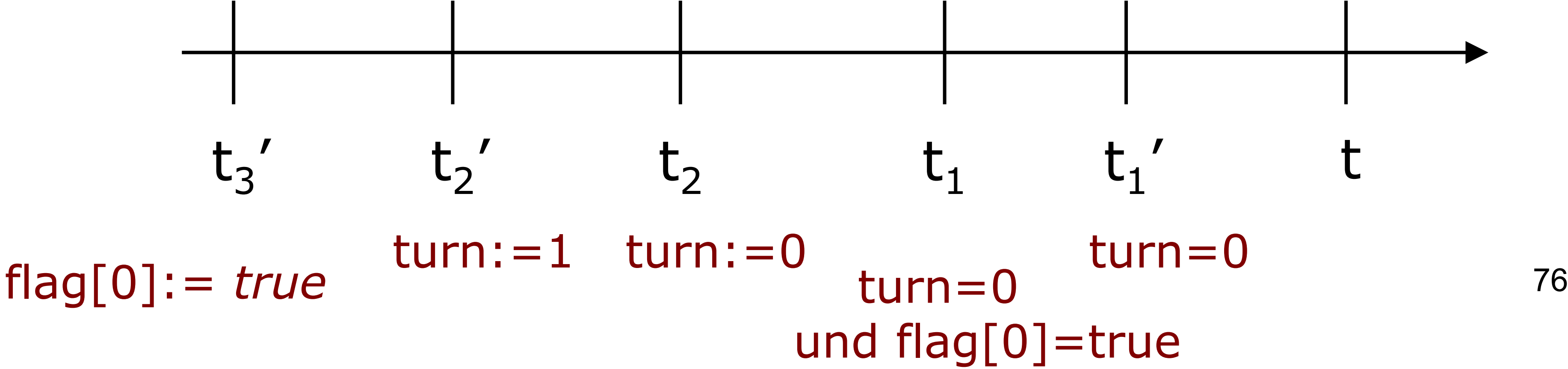
Beweis durch Widerspruch

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



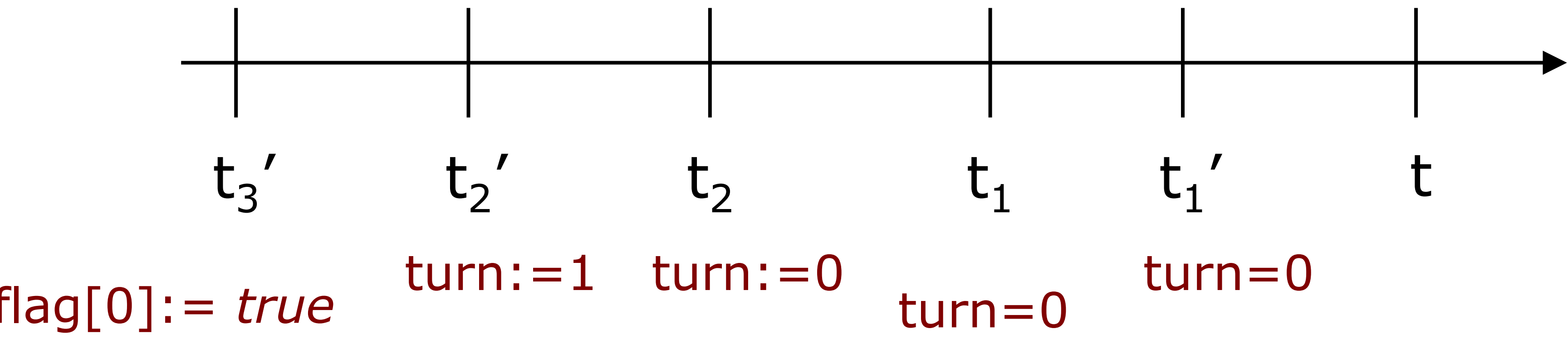
Beweis durch Widerspruch

- Zum Zeitpunkt t_3' wurde flag[0]:=*true gesetzt*
- Zwischen t_3' und t wird flag[0]:=*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 flag[0]=*true* und *turn*=0!
- Also: Prozess 1 kann zum Zeitpunkt t_1 die Schleife nicht verlassen haben und in seinem kritischem Bereich sein
- Widerspruch zur Annahme!
- Es gibt also keinen Zeitpunkt t, zu dem beide Prozesse in ihren kritischem Bereichen sind



Peterson-Algorithmus: Keine gegenseitige Blockierung

- Annahme: Prozess 0 ist in Schleife blockiert
- Also: flag[1]=true und turn=1

/* Prozess 0 */	/* Prozess 1 */

wiederhole
{
flag[0] := true;
turn := 1;
solange (flag[1] = true
und turn = 1)
tue nichts;
/* kritische Region */
flag[0] := false;
/* nicht kritische Region */
}
	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 flag[1]=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!
  - Keine Deadlocks können auftreten! (*turn* kann nicht gleichzeitig 0 und 1 sein)
- Nachteil: Wieder aktives Warten
- Es gibt eine Verallgemeinerung auf *n* Prozesse (wesentlich komplizierter!)

Übung: Versuch 5, Variante 2

- Funktioniert das?

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

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

Übung: Versuch 5, Variante 2

- Funktioniert das?
- Ja! Führt keinen Fehler ein.

/* Prozess 0 */	/* Prozess 1 */

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

Übung: Versuch 5, Variante 3

- Funktioniert das?

/* Prozess 0 */	/* Prozess 1 */

wiederhole
{
flag[0] := true;
turn := 0; ← statt 1
solange (flag[1] = true
und turn = 1) ← orig.
tue nichts;
/* kritische Region */
flag[0] := false;
/* nicht kritische Region */
}
	wiederhole
	{
	flag[1] := true;
	turn := 1; ← statt 0
	solange (flag[0] = true
	und turn = 0) ← orig.
	tue nichts;
	/* kritische Region */
	flag[1] := false;
	/* nicht kritische Region */
	}

Übung: Versuch 5, Variante 3

- Funktioniert das?
- Nein: Wechselseitiger Ausschluss nicht mehr garantiert!

/* Prozess 0 */	/* Prozess 1 */

wiederhole
{
flag[0] := true;
turn := 0; ← statt 1
solange (flag[1] = true
und turn = 1) ← orig.
tue nichts;
/* kritische Region */
flag[0] := false;
/* nicht kritische Region */
}
	wiederhole
	{
	flag[1] := true;
	turn := 1; ← statt 0
	solange (flag[0] = true
	und turn = 0) ← orig.
	tue nichts;
	/* kritische Region */
	flag[1] := false;
	/* nicht kritische Region */
	}

Wechselseitiger Ausschluss in Software: Zusammenfassung

- Wechselseitiger Ausschluss ist in Software schwer zu realisieren
- Alles was einfacher ist als Petersons Algorithmus ist höchstwahrscheinlich falsch
- Fehler durch kritische Wettläufe, subtile Fehler
- Formale Beweise sind unabdingbar!
- 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;
   /* 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:
  - Einführung atomarer Operationen
  - Hardware garantiert atomare Ausführung
- Testen und Setzen zusammen bilden eine atomare Operation:
  - Befehl TSL (Test and Set Lock)
  - Da TSL ein einziger Befehl ist, kann ein Prozesswechsel nicht zwischen Testen und Setzen erfolgen

Wechselseitiger Ausschluss in Hardware (2)

- Gemeinsame Sperrvariable LOCK
- Befehl TSL RX, LOCK mit Speicheradresse von LOCK und Register RX:
  - Inhalt von LOCK wird in RX eingelesen
  - Gleichzeitig: 1 wird an der Speicheradresse von LOCK abgelegt
  - RX = Speicher[LOCK]; Speicher[LOCK] := 1
- Lesen und Setzen ist *ein* Befehl, atomare Ausführung

Wechselseitiger Ausschluss in Hardware (3)

Prozesse, die Zugriff auf den kritischen Abschnitt erhalten wollen, führen folgende Befehle aus:

```
enter_region:
    TSL RX, LOCK          // Kopiere Lock-Inhalt und setze Lock
    CMP RX, #0            // Hat die Sperrvariable den Wert 0?
    JNE enter_region      // Wenn nein, schon gesperrt, Schleife
    ...                   // Fortfahren, Betreten des krit. Bereichs
```

Wechselseitiger Ausschluss in Hardware (3)

Alternative: Befehl XCHG (alle Prozessoren der Intel x-86-Serie benutzen es)

```
enter_region:
    MOVE RX, #1        // Speichere 1 im Register
    XCHG RX, LOCK      // Tausche Inhalte von Register+Sperre
    CMP RX, #0         // Hat die Sperrvariable den Wert 0?
    JNE enter_region   // Wenn nein, schon gesperrt, Schleife
     ...               // Fortfahren, Betreten des krit. Bereichs
```

Wechselseitiger Ausschluss in Hardware (4)

Wenn ein Prozesse, seinen kritischen Bereich verlässt, setzt er die Sperrvariable wieder zurück auf 0:

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

Wechselseitiger Ausschluss in Hardware: Analyse (1)

- Vorteile:
  - Bei beliebiger Anzahl von Prozessen und sowohl Ein- wie auch Mehrprozessorsystemen anwendbar
  - Kann für die Überprüfung mehrere kritischer Regionen eingesetzt werden (jeweils eigene Variable)
  - Nicht-alternierender Zugriff auf den kritischen Abschnitt
  - Wechselseitiger Ausschluss garantiert
  - Kein Deadlock (bei Nichtberücksichtigung von Prioritäten)
- Nachteil: Aktives Warten wie vorher

Wechselseitiger Ausschluss in Hardware: Analyse (2)

- Bei Prioritäten von Prozessen, können diese Lösungen trotzdem zu Deadlocks führen (auch bei Petersons 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 einem Deadlock kommen
- Wenn ein Prozess in seiner kritischen Region oder nach Setzen seines Flags ausfällt
- Andere Prozesse warten ewig

Zwischenstand

- Sowohl Software- wie auch Hardwarelösungen weisen Nachteile auf
- Wechselseitiger Ausschluss muss im Betriebssystem integriert sein
- Besser: Prozesse blockieren statt warten
- Interprozesskommunikation durch Systemaufrufe statt Verschwendung von Rechenzeit

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 (Betriebssystem verwaltet die Warteschlange)
- 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

- Ein Mutex m besteht aus einer binären Lock-Variable $lock_m$, einer Warteschlange $queue_m$ und er besitzt eine eindeutige id_m
- Vor Eintritt in die kritische Region wird die Funktion mutex_lock(m) aufgerufen
- 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($lock_m$) führt TSL-Befehl aus und liefert *false* genau dann, wenn die Lock-Variable $lock_m$ vorher 1 war

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

Nur zur Veranschaulichung; in Wirklichkeit TSL-Befehl

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

Mutex: Warteschlange

- 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
- Das Betriebssystem verwaltet die Warteschlange

mutex_unlock

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

```
function mutex_unlock(mutex m)
 {
   lock_m = 0;
   wakeup(id_m);
   return;
 }
```

Initialisierung (1)

- Zu einem kritischen Bereich gibt es im Programmcode einen sog. Key (Zeichenkette)
- Zur Initialisierung des Mutex gibt es einen Betriebssystemaufruf init_lock(key)
- Beim ersten Aufruf von init_lock(key) durch einen Prozess, wird
  - eine eindeutige lock-Variable reserviert
  - eine eindeutige Nummer id vergeben

Initialisierung (2)

- id ist der Rückgabewert von init_lock(key)
- Rufen weitere Prozesse init_lock(key), wird lediglich id zurückgeliefert
- Im Benutzerprogramm kann id dann benutzt werden
- Alle Aufrufe von mutex_lock und mutex_unlock benutzen die Nummer id als Parameter für sleep und wakeup

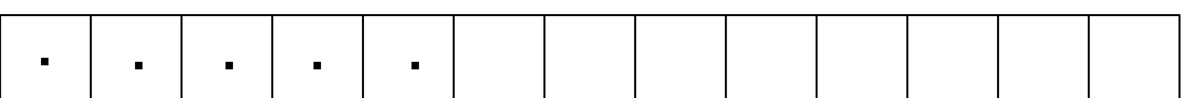
Mutexe: Zusammenfassung

- Einfachste Möglichkeit um wechselseitigen Ausschluss mit Systemfunktionen zu garantieren
- Zwei Zustände: gesperrt / nicht gesperrt
- Kein aktives Warten, CPU wird abgegeben
- Der Prozess, der den Mutex gesperrt hat, gibt ihn auch wieder frei

Das Erzeuger-Verbraucher-Problem (1)

Ein typisches Problem bei nebenläufiger Datenverarbeitung (z.B. Drucker, Webserver)

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

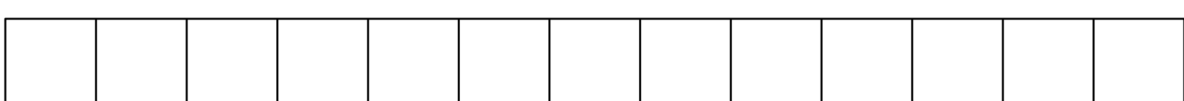


Das Erzeuger-Verbraucher-Problem (2)

- 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 Erzeuger-Verbraucher-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 Erzeuger-Verbraucher-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);                    // fügen Objekt in Puffer ein
       count = count + 1;
       wenn (count = 1)                      // wenn Puffer vorher leer
           wakeup(consumer_id);              // wecke Konsumenten
     }
 }
```

Das Erzeuger-Verbraucher-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 Erzeuger-Verbraucher-Problem – Eine Idee

Ist diese Lösung korrekt?

```
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 Erzeuger-Verbraucher-Problem – Fehlersituation (1)

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

Das Erzeuger-Verbraucher-Problem – Eine Idee

Ist diese Lösung korrekt?

```
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 Erzeuger-Verbraucher-Problem – Fehlersituation (2)

2. Fehlersituation mit zwei Verbrauchern und einem schlafenden Erzeuger

- Puffer ist voll (count=MAX_BUFFER)
- 1. Verbraucher entnimmt Objekt, zählt count runter: count=MAX_BUFFER-1, wird dann unterbrochen
- 2. Verbraucher entnimmt ebenfalls Objekt, zählt count runter: count=MAX_BUFFER-2
- Aufwecken des Erzeugers geht verloren!

Das Erzeuger-Verbraucher-Problem – Eine Idee

Ist diese Lösung korrekt?

```
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 Erzeuger-Verbraucher-Problem – Fehlersituation (3)

3. Fehlersituation mit je einem Verbraucher und Erzeuger

- Puffer ist leer
- „wenn (count=0)“ wird ausgeführt, Unterbrechung vor sleep(consumer_id)
- Dann wird komplett Erzeuger ausgeführt, Aufruf „wakeup(consumer_id)“ hat keinen Effekt
- Aufwecken geht verloren!

Das Erzeuger-Verbraucher-Problem – Fehlersituation (3)

- Dann wird Verbraucher weiter ausgeführt, „sleep(consumer_id)“
- Situation: Verbraucher schläft, obwohl Puffer nicht leer!
- Evtl. noch weitere Erzeuger-Aufrufe, aber weil der Puffer nie mehr leer ist, gibt es nie mehr den Aufruf „wakeup(consumer_id)“
- Analog bei Unterbrechung nach „wenn (count = MAX_BUFFER)“ in Erzeuger

Das Erzeuger-Verbraucher-Problem – Zusammenfassung

- Diese Idee ist keine Lösung!
- Problem:

  „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?
  - Letzteres würde fast immer zum Überprüfen der Warteschlage führen (auch wenn keine Prozesse drin sind)
  - Dies wäre also meistens überflüssig, zu viele Systemaufrufe
- Warum nicht um alles eine kritische Region?
  - Blockieren möglich
  - Falls sleep ausgeführt wird (bei leerem Buffer), kann anderer Prozess (producer) nichts dazu tun, dass ein Aufwecken später stattfindet

Das Erzeuger-Verbraucher-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: keine Weckrufe sind bisher gespeichert, nächster Prozess legt sich schlafen
- Wert > 0: frei, nächster Prozess darf fortfahren

Semaphor: down/up Operationen

- Initialisiere Zähler des Semaphors: $count_s = 1$
- down-Operation:
  - Verringere den Wert von $count_s$ um 1
  - Wenn $count_s < 0$, blockiere den aufrufenden Prozess (Warteschlange), sonst fahre fort

  atomare Operation

- up-Operation:
  - Erhöhe den Wert von $count_s$ um 1
  - Wenn $count_s <= 0$, wecke einen der blockierten Prozesse auf

  atomare Operation

  - Bedeutung: Wenn $count_s < 0$ vor Erhöhen: Es gibt mindestens einen wartenden Prozess

Semaphore: Unteilbarkeit

- Wenn einmal eine Operation eines Semaphors begonnen wurde: Kein anderer Prozess kann auf das Semaphor zugreifen
- Unentbehrlich um Synchronisations-probleme zu vermeiden

Semaphore

Ein Semaphor s hat drei Komponenten:

- Integer-Variable $count_s$
- Warteschlange $queue_s$
- Lock-Variable $lock_s$

Wechselseitiger Ausschluss mit Semaphoren (1)

Voraussetzungen:

- Es existiert ein Semaphor s
- $count_s$ ist auf 1 initialisiert
- n Prozesse sind gestartet, konkurrieren um kritischen Abschnitt

```
/* 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: $count_s$=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: $count_s=0$
- 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: $count_s = -1$

```
/* 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: $count_s$=-1
- 2. Prozess wird schlafengelegt + eingefügt in $queue_s$

```
/* 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: $count_s$=-1
- 2. Prozess wird schlafengelegt + eingefügt in $queue_s$
- Analog für 3. Prozess: down(s) führt zu $count_s$=-2, Prozess wird schlafengelegt + eingefügt in $queue_s$

```
/* 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
- $count_s$<=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
- $count_s$<=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: $count_s = 0$
- $count_s <= 0$: 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: $count_s=0$
- $count_s<=0$: 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: $count_s=m$, $m>1$
- Wenn $count_s>=0$, dann Wert von $count_s$:

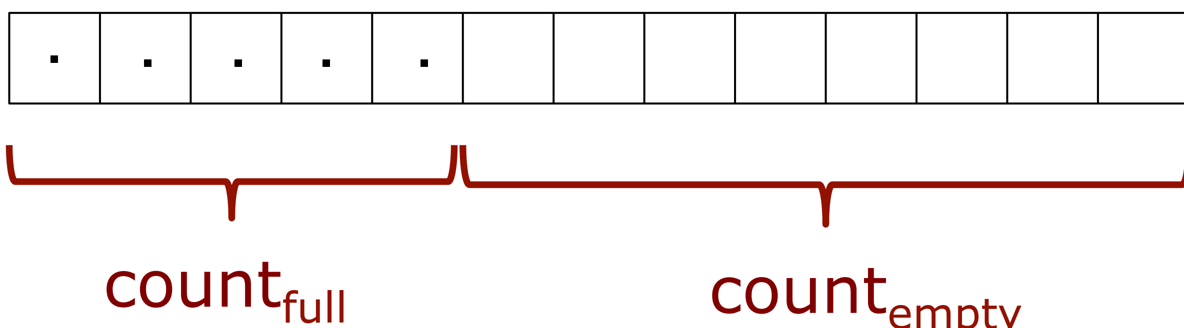
  Anzahl der Prozesse, die down(s) ausführen können ohne zu blockieren (ohne dass zwischenzeitlich up(s) ausgeführt wird)
- $count_s < 0$, dann gilt:

  $|count_s|$ ist die Anzahl der wartenden Prozesse in $queue_s$

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:
  - $count_{empty}$ = MAX_BUFFER
  - $count_{full}$ = 0

Produzenten/Konsumenten-Problem mit Semaphoren (2)

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

Produzenten/Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; count_mutex = 1;              // mutex für kritische Bereiche
semaphore empty; count_empty = MAX_BUFFER;     // zählt freie Plätze
semaphore full; count_full = 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; $count_{mutex}$ = 1; // mutex für kritische Bereiche
semaphore empty; $count_{empty}$ = MAX_BUFFER; // zählt freie Plätze
semaphore full; $count_{full}$ = 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);
    }
 }
```

kritischer Bereich

Produzenten/Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; count_mutex = 1;              // mutex für kritische Bereiche
semaphore empty; count_empty = MAX_BUFFER;     // zählt freie Plätze
semaphore full; count_full = 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);
    }
 }
```

kritischer Bereich

Produzenten/Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; count_mutex = 1;                   // mutex für kritische Bereiche
semaphore empty; count_empty = MAX_BUFFER;     // zählt freie Plätze
semaphore full; count_full = 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

Produzenten/Konsumenten-Problem mit Semaphoren (3)

```
semaphore mutex; count_mutex = 1;               // mutex für kritische Bereiche
semaphore empty; count_empty = MAX_BUFFER;      // zählt freie Plätze
semaphore full; count_full = 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; count_mutex = 1;             // mutex für kritische Bereiche
semaphore empty; count_empty = MAX_BUFFER;    // zählt freie Plätze
semaphore full; count_full = 0;               // zählt belegte Plätze
```

```
Prozedur consumer
 {
  …
  wiederhole
   {
→   down(full);   zähle runter; überprüfe, ob noch 1 belegter Platz; 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; count_mutex = 1;              // mutex für kritische Bereiche
semaphore empty; count_empty = MAX_BUFFER;     // zählt freie Plätze
semaphore full; count_full = 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; count_mutex = 1;                  // mutex für kritische Bereiche
semaphore empty; count_empty = MAX_BUFFER;         // zählt freie Plätze
semaphore full; count_full = 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)

- Das vorgestellte Verfahren funktioniert für eine Anzahl $m>1$ von Prozessen, wenn die Operationen atomar 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 seinen kritischen Bereich 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
- Zweites Schlafenlegen des Produzenten ist eigentlich unnötig

Produzenten/Konsumenten-Problem mit Semaphoren (7)

Frage: Funktioniert das immer noch, wenn in Prozedur consumer

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

nein!

```
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 (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 den kritischen Bereich, 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(mutex_s);
    count_s = count_s - 1;
    wenn (count_s < 0)
      {
        setze diesen Prozess in
                        queue_s;
        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;
        Schreibe Prozess P in Liste
        der bereiten Prozesse
      }
    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 aufgrund des Mutex
- Zumindest für binäre Semaphore ist die Verwendung von Mutexen in Semaphoraufrufen etwas aufwändig! (Mutexe alleine reichen aus)
- 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(lock_s) = false)
      tue nichts;
  count_s = count_s - 1;
  wenn (count_s < 0)
    {
      setze diesen Prozess in
                           queue_s;
      blockiere den Prozess und
      führe unmittelbar vor
      Abgabe des Prozessors noch
      lock_s = 0 aus
    }
  sonst
    lock_s = 0
  }
```

```
up(semaphore s)
  {
  solange (testset(lock_s) = false)
      tue nichts;
  count_s = count_s + 1;
  if (count_s <= 0)
    {
      entferne einen Prozess P aus
                                  queue_s;
      Schreibe Prozess P in Liste
      der bereiten Prozesse
    }
  lock_s = 0;
  }
```

Freigabe des kritischen Bereichs

Implementierung von Semaphoren: Versuch 2 Analyse

- Aktives Warten, aber nicht so gravierend: Beschränkt auf up und down, diese sind relativ kurz
- down und up sind nicht wirklich atomar, aber trotzdem stören sich verschiedene Aufrufe von down und up nicht aufgrund der solange-Schleife
- Semaphoren blockieren Prozesse: Keine CPU-Zeit für wartende Prozesse
- Implementierung auch für Multiprozessoren geeignet
- Für den Benutzer sind nur abstrakte Semaphoren sichtbar, keine Implementierungsdetails

Zusammenfassung

- CPU (einzelne CPU oder Multiprozessor) wird von mehreren Prozessen geteilt
- Verwaltung gemeinsamer Ressourcen bei Multiprogramming ist notwendig und ist nicht trivial
- Subtile Fehler möglich, formale Beweise nötig
- Verschiedene Konzepte für wechselseitigen Ausschluss, 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 zu derselben gemeinsamen Ressource 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
- Prozess, der Mutex sperrt, muss ihn auch wieder freigeben