

Betriebssysteme

Kap. 3: Interprozesskommunikation (IPC)

3.1: Einführung

Stand: WS 12/13 (15.01.13)

Prof. Dr. Wolfgang Küchlin

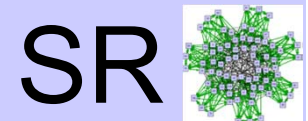
Dipl.-Inform., Dr. sc. techn. (ETH)

**Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften**

Universität Tübingen

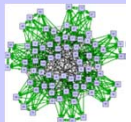
**Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)**

**Wolfgang.Kuechlin@uni-tuebingen.de
<http://www-sr.informatik.uni-tuebingen.de>**



Teil I: Einführung

- IPC = Synchronisation und Kommunikation
- Grundtypen von Zugriffskonflikten
- Grundtyp der Kommunikation



Interaktion mehrerer Akteure

➤ Interaktion

- zwischen zwei Akteuren (= *Threads of control*)
- erfordert Austausch von Information (=Kommunikation)
- Kommunikation durch
 - gemeinsamen Speicher (→ Zugriffskonflikte)
 - Nachrichtenaustausch (→ Vermittler)

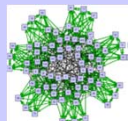
➤ Interaktion = Synchronisation + Kommunikation

➤ Synchronisation

- Zur Vermeidung von (Zugriffs-)Konflikten zw. den Akteuren

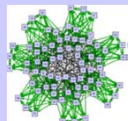
➤ Kommunikation

- Zum gemeinsamen Lösen von Problemen durch Partner



Concurrency / Nebenläufigkeit

- Aktivitäten heißen **nebenläufig (concurrent)**, wenn sie *logisch gleichzeitig* ablaufen
- Concurrent Processes laufen auf einem sequentiellen Rechner in *beliebiger* (unvorhersehbarer) Verschachtelung ab
- Concurrent Processes können auf einem parallelen Rechner echt gleichzeitig ablaufen
- Aus logischer Sicht ergibt sich kein Unterschied
 - die grundsätzlichen Probleme sind gleich
 - **Synchronisation** und **Kommunikation** nötig



Synchronisation und Serialisierung

- Grundproblem: Zugriffskonflikte
 - Beim nebenläufigen rw-Zugriff auf gemeinsame Variablen können Effekte (Artefakte) auftreten, die bei beliebiger Hintereinander-Ausführung der Zugriffe unmöglich wären.
 - Das Ergebnis des Codes kann von der genauen zeitlichen Ausführung abhängen, d.h. vom Wettlauf der Akteure um den Zugriff auf die gemeinsame Variable (*race condition*).
- Ziel: Serialisierbarkeit durch Synchronisation
 - Die Ausführung kritischer Codestücke soll zwischen den Akteuren so **synchronisiert** werden, dass das Ergebnis nicht vom timing der Ausführung abhängt. Es sollen nur solche Effekte auftreten können, die auch bei einer **Serialisierung** (Hintereinander-Ausführung) möglich wären.



Race conditions

➤ Race conditions

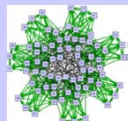
- beim Zugriff auf gemeinsame Variablen
- gemeinsames Lesen problemlos

➤ Grundproblem

- Update: erst Lesen, dann (verändert) Schreiben
- Update kann unterbrochen werden

➤ 2 Problemtypen

- Ausfall
 - einer der Updates verpufft wirkungslos
- Inkonsistenz
 - zusammengesetzte Datenstruktur nach Update inkonsistent

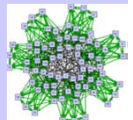
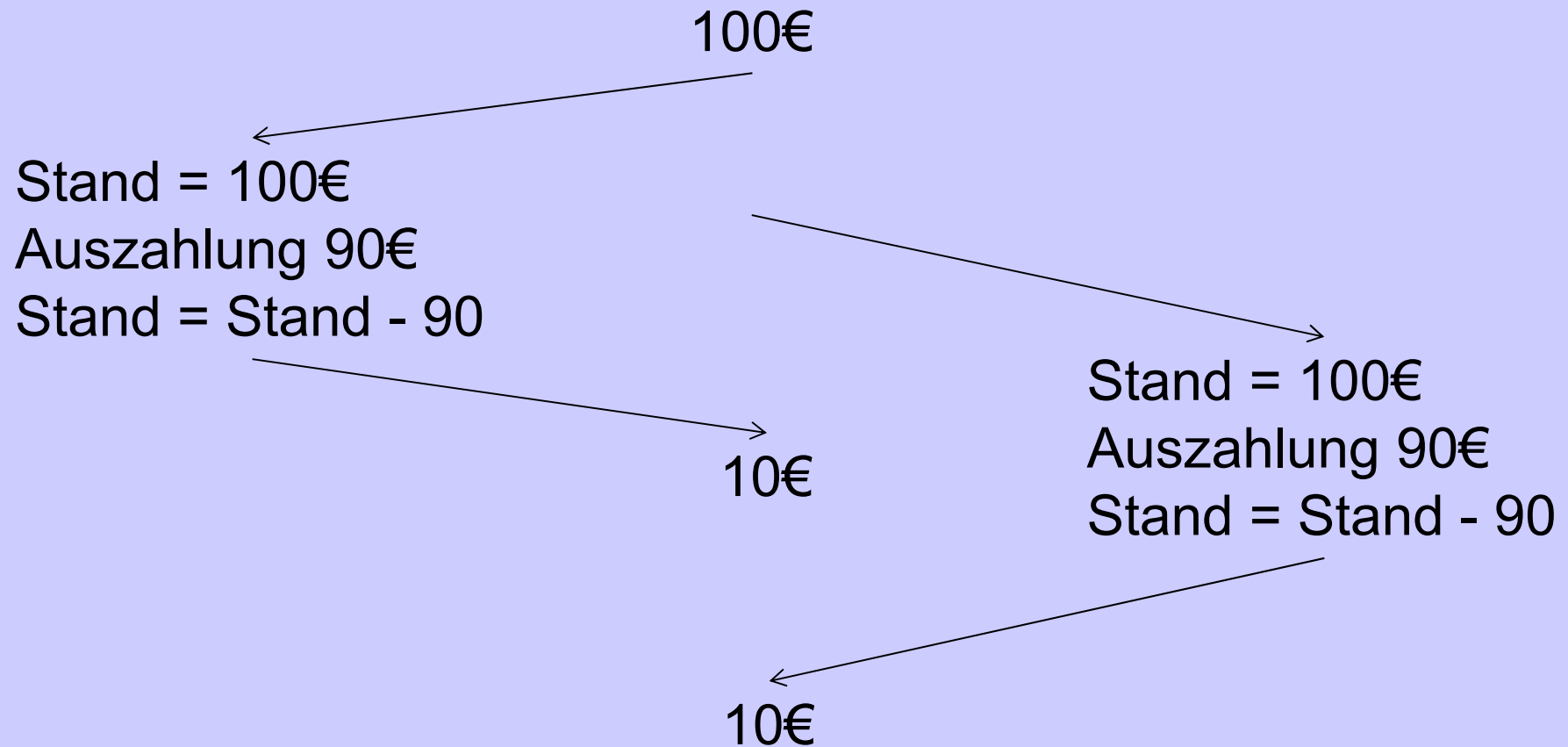


Ausfall bei Zugriff auf Bank-Konto

Automat 1

Kontostand

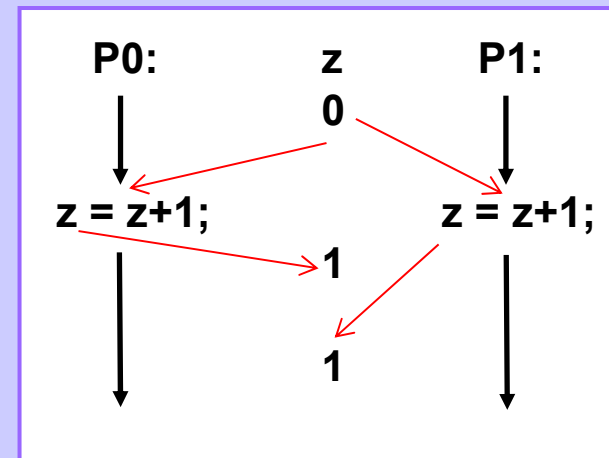
Automat 2



Ausfall bei Inkrement von Variablen

➤ Beispiel zum Typ: Ausfall

- Zwei Prozesse zählen auf gleicher (globaler) Variablen
- $z = z + 1$ wird zerlegt in
 - Laden in Register
 - Inkrementieren des Registers
 - Speichern des Registers
 - i.A. auch bei $z++$



- Inkrements gehen bei Prozesswechsel zwischen „Laden“ und „Speichern“ verloren. → **Ausfall**
- Problemtyp **Ausfall: mindestens 2 writer**



Race condition bei Zuweisung

- Ergebnis einer Operation hängt von zeitlichen Zufälligkeiten ab (gemeinsames Rennen zum Ziel)

```
int z=0; // globale Variable
```

```
int P1()
```

```
{z=1;
```

```
return(z);
```

```
}
```

```
int P2()
```

```
{z=2;
```

```
return(z);
```

```
}
```

- Ergebnis von P1 / P2 ist je nach dem:

1 / 1 oder 1 / 2 oder **2 / 2**

Artefakte



Increment Race

- Ergebnis einer Inkrement / Dekrement Operation hängt von zeitlichen Zufälligkeiten ab

```
int z=0;
```

```
int P1()
```

```
{z++;
```

```
return(z);
```

```
}
```

```
int P2()
```

```
{z++;
```

```
return(z);
```

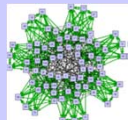
```
}
```

- Ergebnis von P1 / P2 ist je nach dem:
1 / 1 oder **1 / 2** oder **2 / 1** oder **2 / 2**
- Ein Inkrement kann verloren gehen.



Zugriffskonflikt mit Inkonsistenz

- Gemeinsame Datenstruktur ist zusammengesetzt
- Größer als 1 Wort
 - long (bei 32 bit Architektur)
 - double (bei 32 bit Architektur)
 - Record / Objekt / File etc.
- Beispiel: gemeinsames Objekt x , $x.a = x.b = 1$.
 - P1 liest $y_a = x.a (=1)$.
 - P2 schreibt $x.a = 0$; $x.b = 0$.
 - P1 liest $y_b = x.b (=0)$.
 - P1 hat inkonsistente Kopie von x
- Konflikt schon bei 1 Schreiber und 1 Leser.



Beispiel: Inkonsistenz

```

writer (a)
int a[2];
{ a[0] = 1;
  a[1] = 1;
}

```

```

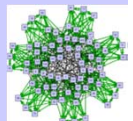
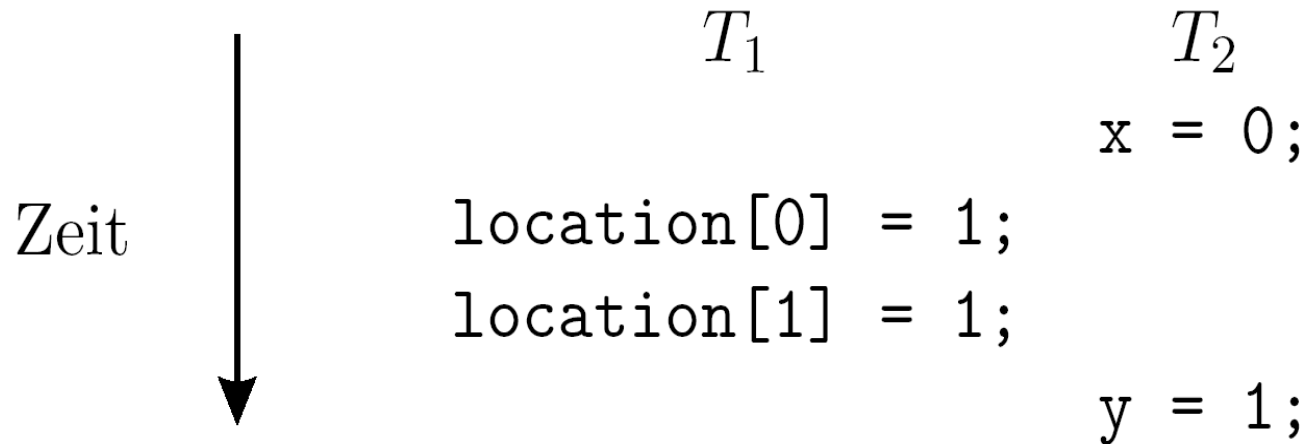
reader (a)
int a[2];
{ x = a[0];
  y = a[1];
}

```

```

int location[2];
fork(writer,location); fork(reader,location);

```



Beispiel: Update eines Datensatzes

- Schlüsselpaar für kryptographisches Verfahren wird nach jedem Gebrauch mit neuen zufällig gewählten Schlüsseln erneuert

A

Schlüssel für Kodierung

B

Schlüssel für Dekodierung

P0:

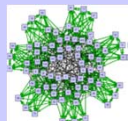
```
r = random();  
A = create_A( r );  
B = create_B ( r );
```

**P1:**

```
r = random();  
A = create_A( r );  
B = create_B ( r );
```



- Möglichkeit, dass A und B von verschiedenen r stammen (und somit nicht zusammenpassen) → **Inkonsistenz**

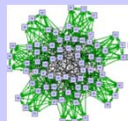


Bereinigung von Zugriffskonflikten

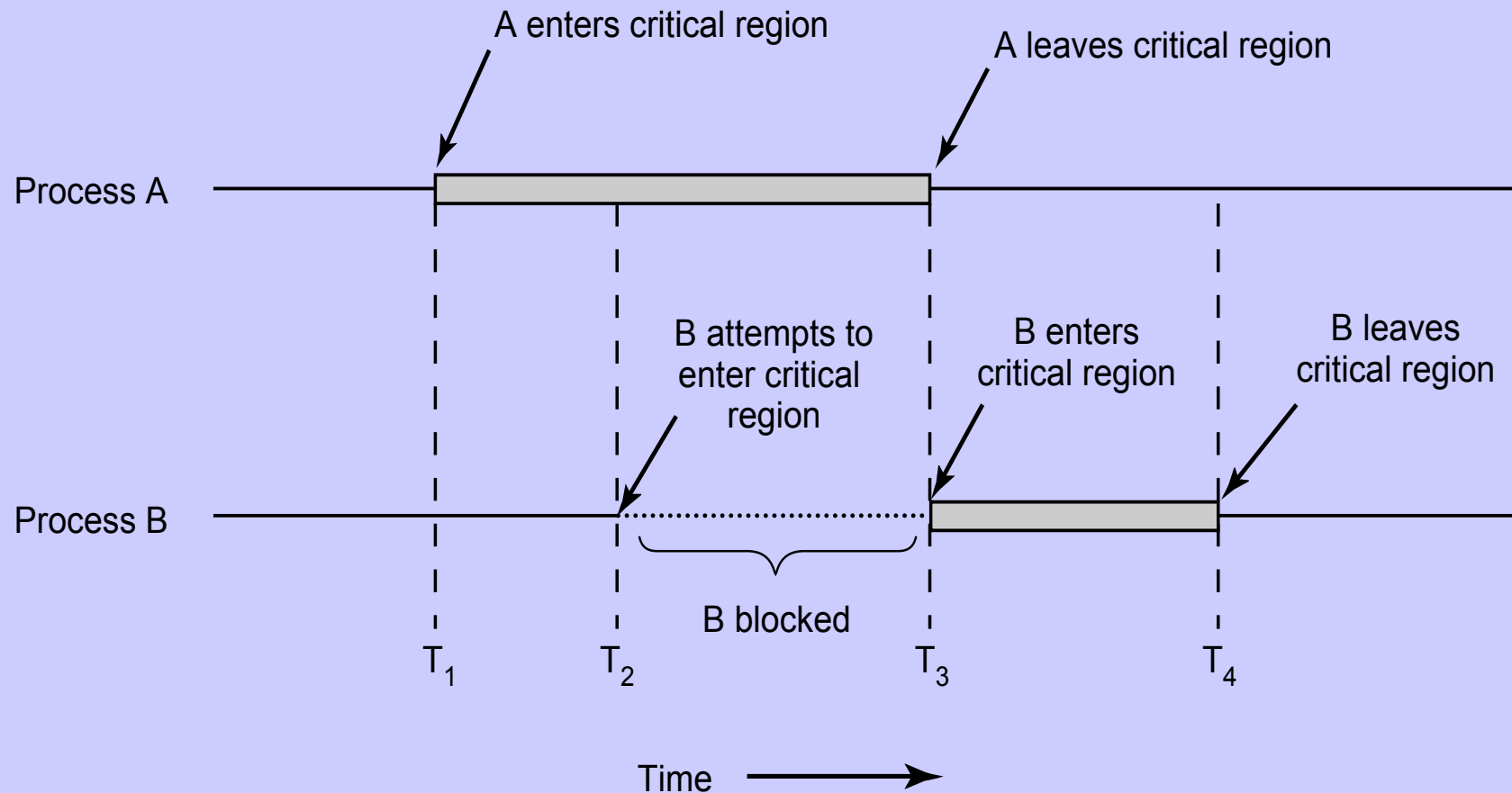
- kritische Abschnitte (KA)
 - Codesequenzen, die zu Zugriffskonflikten führen können
- kritische Ressourcen (KR)
 - kritische Abschnitte
 - gemeinsame Datenstrukturen
- Synchronisation durch *code* oder *data locking*
- Hilfsmittel für locking: Schlossvariable (*mutual exclusion lock, mutex*)

```
void mutex_lock(m)
    mutex_t m;
```

```
void mutex_unlock(m)
    mutex_t m;
```



Kritische Abschnitte



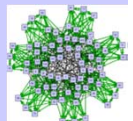
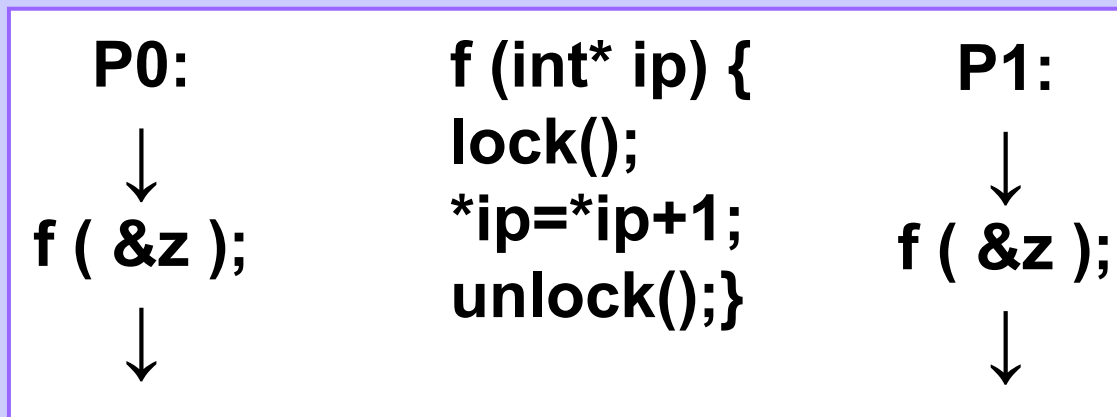
Tanenbaum, Abbildung 2-19: Mutual exclusion using critical regions

Wolfgang Küchlin, WSI und STZ OIT, Uni Tübingen



Zugriffskonflikte: Lösungsansatz

- Allgemein: Schutz von kritischen Ressourcen
 - Schutz von Code-Sequenzen
 - Kritische Bereiche/Abschnitte (**Critical Section, Critical Region**)
 - Lösung mit **Code Locking** (Lock gehört logisch zum Code)

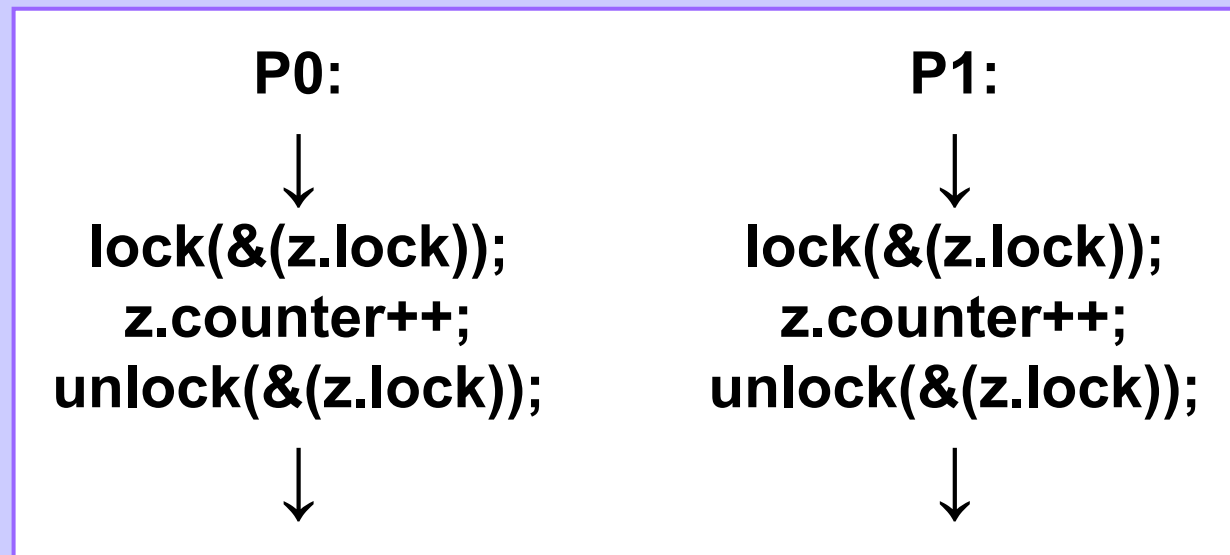


Zugriffskonflikte: Lösungsansätze (II)

- Schutz von Daten: Kritische Daten (**Critical Data**)
- Lösung mit **Data Locking** (Lock gehört logisch zu den Daten)

- Daten mit Lock:

```
struct lockedcounter {  
    mutex_t lock;  
    int counter;  
};
```



Granularität

➤ Wichtiger Gesichtspunkt für Effizienz:

Granularität (*grain size*)

- Größe der geschützten Ressource

➤ Locking kann sein:

- **grob-granular** (*coarse grained*)

- Akteure halten sich lange in den KR auf
- Bsp.: Datei-Ebene

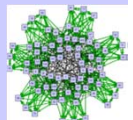
- **mittel-granular** (*medium grained*)

- Bsp.: Gruppe von Records

- **feingranular** (*fine grained*)

- Bsp.: ein Record oder einzelne Komponenten eines Record

➤ Antagonismus zwischen Granularität und Parallelität.



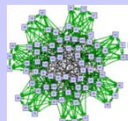
Wechselseitiger Ausschluss: Desiderata

n Prozesse mit kritischen Bereichen

- Gegenseitiger Ausschluss garantiert, dass max. 1 Prozess in einem kritischen Bereich ist.

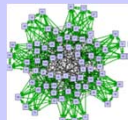
Gesuchtes Verfahren soll erfüllen

1. **[Korrektheit]**
Nie dürfen 2 Prozesse gleichzeitig in einem gemeinsamen kritischen Abschnitt sein.
2. **[Allgemeinheit, Portabilität]**
Es dürfen keine Annahmen über Geschwindigkeit oder Anzahl der CPUs in die Lösung eingehen.
3. **[Effizienz]**
Kein Prozess, der außerhalb eines kritischen Abschnitts läuft, darf einen anderen Prozess aufhalten (blockieren).
4. **[Fairness]**
Kein Prozess muss je ewig darauf warten, einen kritischen Abschnitt ausführen zu dürfen. (**Starvation**)



Lösung auf Stufe Betriebssystem

- Deaktivieren und Maskieren von Unterbrechungen in kritischen Abschnitten.
 - Elegant!
 - Funktioniert (effizient) nur bei 1-Prozessor-Maschinen
 - oder asymmetrischen Multiprozessoren, bei denen nur Master-Prozessor Interrupts bearbeitet.
 - Anfällig auf Probleme (blockieren, etc.)
 - Für User Prozesse ungeeignet
 - Nebenläufigkeit durch abwechselndes Scheduling nach System Calls nach wie vor möglich
 - Blockieren aller Ressourcen durch User-Prozess wäre möglich
 - Alle Nachteile von single-tasking BS
 - In Spezialfällen dennoch möglich (z.B. Mikrocontroller)



Software-Lösungen für Lock/Unlock

P1:	$p = 0$	P2:
while (...) {		while (...) {
...		...
while (p != 0);		while (p != 0);
p=1;		p=2;
... (krit. Bereich)		... (krit. Bereich)
p=0;		p=0;
...		...
}		}

➤ Probleme

- nicht korrekt
 - bei Unterbrechung zwischen Test und Zuweisung
- Ineffizient durch Aktives Warten (**Busy Waiting**)



Versuch II (Strict Alternation)

```
P1:
while (1) {
    while (turn != 1); //wait
    critical_section();
    turn = 2;
    rest1();
}
```

```
P2:
while (1) {
    while (turn != 2); //wait
    critical_section();
    turn = 1;
    rest2();
}
```

➤ Probleme

- Strikt abwechselnde Nutzung des kritischen Bereichs, die while (...) werden damit synchronisiert.
- Verletzt:
 - Allgemeinheit, Effizienz, evtl. auch Starvation
- Aktives Warten (**Busy Waiting**)



Versuch III (Setzen eigenes und Testen fremdes Flag)

```
P1:
bereit[1] = FALSE;
while ( ... ) {
    ...
    bereit[1] = TRUE;
    while (bereit[2] ); //wait
    ... (krit. Bereich)
    bereit[1] = FALSE;
    ...
}
```

```
P2:
bereit[2] = FALSE;
while ( ... ) {
    ...
    bereit[2] = TRUE;
    while (bereit[1] ); //wait
    ... (krit. Bereich)
    bereit[2] = FALSE;
    ...
}
```

- Korrekt, keine strikte Abwechslung vorgeschrieben
- Probleme
 - Deadlock bei gleichzeitigem `bereit[.] = TRUE`
 - Effizienz: Prozess, der nach `bereit[.]` blockiert, blockiert auch Partner
 - Aktives Warten (**Busy Waiting**)
 - nicht allgemein



Versuch IV (Testen fremdes und Setzen eigenes Flag)

```
P1:
bereit[1] = FALSE;
while ( ... ) {
    ...
    bereit[1] = TRUE;
    while (bereit[2] ); //wait
    ... (krit. Bereich)
    bereit[1] = FALSE;
    ...
}
```

```
P2:
bereit[2] = FALSE;
while ( ... ) {
    ...
    bereit[2] = TRUE;
    while (bereit[1] ); //wait
    ... (krit. Bereich)
    bereit[2] = FALSE;
    ...
}
```

- Probleme
 - Nicht korrekt bei gleichzeitigem while (bereit[.])
- III/IV gemischt
 - Ebenfalls nicht korrekt bei Unterbrechung nach Test und vor dem Setzen.



Versuch V (Lösung von Peterson)

```
void enter_region (process)
int process;
{ int other = 1 - process;          /* other process */
  interested [process] = 1;
  loser = process;
  while (loser == process && interested [other]) /* wait */;
}
void leave_region (process)
int process;
{ interested [process] = 0; }
```

- Globale var. „loser“: es verliert, wer diese zuletzt setzt (kein deadlock).
- Ist korrekt, erfüllt unsere Anforderungen
- Lösung nicht allgemein, da sie nur für 2 Prozesse funktioniert.
- Lösung nur für Prozessnummer aus 1 Byte portabel.



Hardware-Unterstützung: TSL

- Die meisten Prozessoren bieten atomare **Test-und-Set** Instruktion TSL (*Test and Set Lock*) für Mutex-Implementierung.
- Die TSL Instruktion: tsl register, lock
 - Aktionen:
 - register = lock;
 - lock = 1;
 - Speicherbus wird während der Ausführung gesperrt
→ Lesen und Schreiben laufen atomar ab
- Semantik:

Alter Wert	Return Wert	Neuer Wert
1	1	1
0	0	1
- SPARC: TSL wird mit LDSTUB implementiert (Atomic Load-Store Unsigned-Byte)



Hardware-Unterstützung: TSL

- Für die folgenden Beispiele verpacken wir die TSL-Instruktion in eine C-Funktion

- `int TSL(int* lock)`

```
{// store 1 in register;
```

```
// tsl register, lock
```

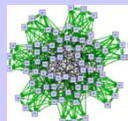
```
// return register;
```

```
}
```

- Dies ist möglich, da mittels „asm“-Direktiven Assembler-Anweisungen in C-Code eingebettet werden können.

- Sei lockp ein pointer auf eine Lockvariable.

Nach einem Aufruf `oldval = TSL(lockp)` gilt `*lockp == 1` und der Wert von `oldval` ist der alte Wert von `*lockp` vor dem Aufruf.



Mutex als Spin-Lock auf Basis TSL

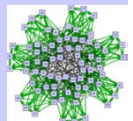
- ***spin lock***: wechselseitiger Ausschluss mit aktivem Warten

```
while ( TSL(a) == 1 );  
      :   (krit. Bereich)  
*a = 0;
```

mutex_lock(&a); als spin lock
krit. Bereich;
mutex_unlock(&a);

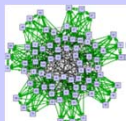
➤ Probleme

- TSL-Operation belastet Prozessor und Bus
- Nicht fair, Verhungern ist möglich
- Aktives Warten (teuer!)



Mutex als Spin-Lock

```
void mutex_lock (lock);  
int * lock;  
{ while (TSL (lock))      /* wait */; }  
  
void mutex_unlock (lock);  
int * lock;  
{ * lock = 0; }
```



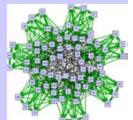
Mutex als Spin-Lock

➤ Mutex für Multiprozessoren

- bisherige Lösung bei Multiprozessorsystemen problematisch
- tsl führt wegen Schreiben ständig zur Invalidierung in allen Caches → lokale Kopie lock_p der Schlossvariable für jeden Prozessor

```
void mutex_lock (lock_p)
char* lock_p;
{
    while ( TSL(lock_p)) {
        while (* lock_p)          /* wait on local cache copy */ ;
    }
}
```

- *burst of activity* in Folge von mutex_unlock



Mutex als Spin-Lock

➤ Warten an lokaler Kopie (um Bus zu entlasten)

```
while ( TSL(a) == 1 )  
    while (*a == 1 );  
    : (krit. Bereich)  
    *a = 0;
```

Hauptspeicher-Zugriff
Cache Zugriff

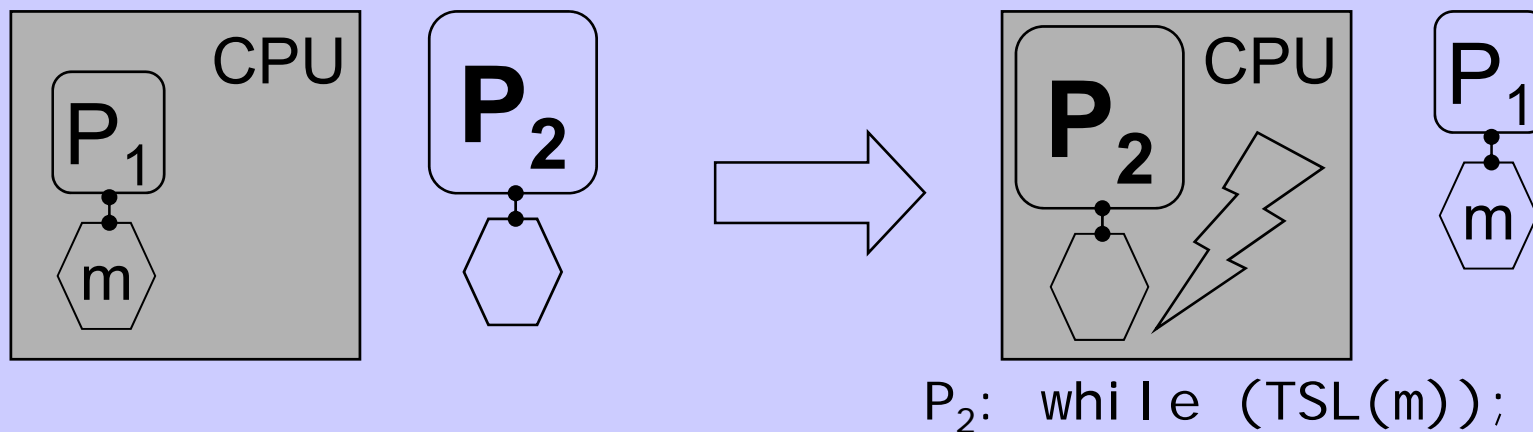
➤ Probleme

- Nicht fair, Verhungern ist möglich
- Prioritäts-Inversion:
 - Deadlock möglich, falls aktiv wartender Prozess höhere Priorität hat, als derjenige, der das Lock hält
 - Lösung durch (temporäre) Vererbung der Priorität (priority inheritance). Der Prozess, der das Lock hält, bekommt die Priorität des Wartenden, bis er das Lock zurückgibt



Prioritäts-Inversion

- Prioritäts-Inversion durch aktives Warten
- Beispiel:
 - P_1 Prozess niederer Priorität
 - P_2 Prozess hoher Priorität, aber P_2 ist blockiert
 - P_1 hält Lock m
 - Wird P_2 deblockiert, so wird P_1 verdrängt und P_2 muss auf P_1 warten.



Mutex mit passivem Warten

➤ Mutex mit Spin Limit

- In C Threads daher statt `while (TSL(m));`;
üblicherweise:

```
for (i=0; i < mutex_spin_limit; i++)  
    if (! TSL(m)) return;
```

- Falls das `mutex_spin_limit` überschritten,
nachfolgend passives Warten (→ sleep)

➤ Passives Warten

- Einketten in Warteliste am Lock `m` (*sleep on the lock*)
- Aufwecken der auf `m` Wartenden als Seiteneffekt von `mutex_unlock(m)`. Alle führen erneut `mutex_lock(m)` aus.
- passives Warten benötigt das Betriebssystem!

