Betriebssysteme

Kap. 3: Interprozesskommunikation (IPC)
3.1: Einführung

Stand: WS 08/09 (7.1.09)

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

SR 

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

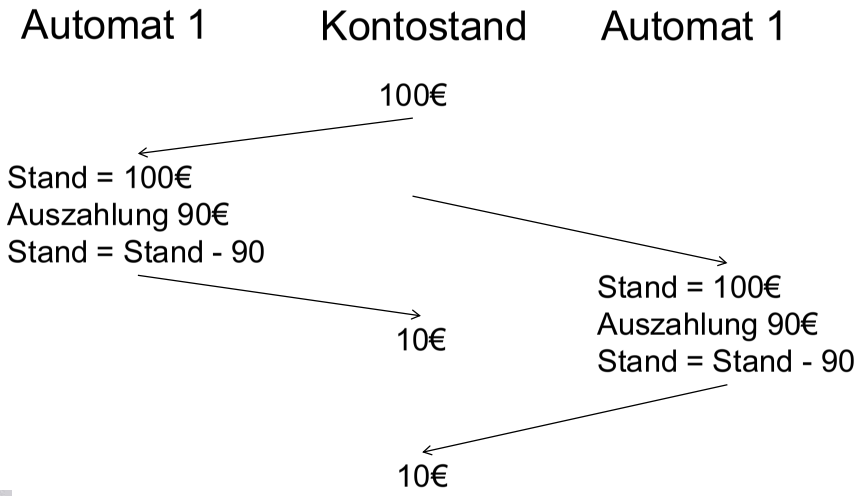
Synchronisation und Serialisierung

- Grundproblem: Zugriffskonflikte
  - Beim nebenläufigen 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 Zugriffe soll zwischen den Akteuren so **synchronisiert** werden, dass das Ergebnis nicht vom timing der Zugriffe abhängt. Es sollen nur solche Effekte auftreten können, die auch bei einer **Serialisierung** (beliebige Hintereinander-Ausführung) möglich wären.

Race conditions

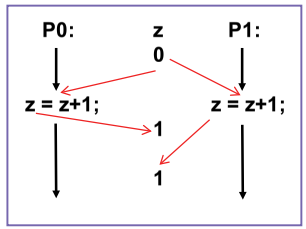
- Race conditions
  - beim Zugriff auf gemeinsame Variablen
  - gemeinsames Lesen problemlos
- Grundproblem
  - Update: erst Lesen, dann (verändert) Schreiben
  - Update kann unterbrochen werden
- 2 Problemtypen
  - Ausfall
    - einer der Updates verpufft wirkungslos
  - Inkonsistenz
    - zusammengesetzte Datenstruktur nach Update inkonsistent

Ausfall bei Zugriff auf Bank-Konto



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

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

```
int P1()                int P2()
{static int z;          {static int z;
z=1;                    z=2;
return(z);              return(z);
}                       }
```

- Egebnis von P1 / P2 ist je nach dem:
  1 / **1** oder 1 / 2 oder **2** / 2

Artefakte

Increment Race

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

```
int P1()                int P2()
{static int z;          {static int z;
z++;                    z++;
return(z);              return(z);
}                       }
```

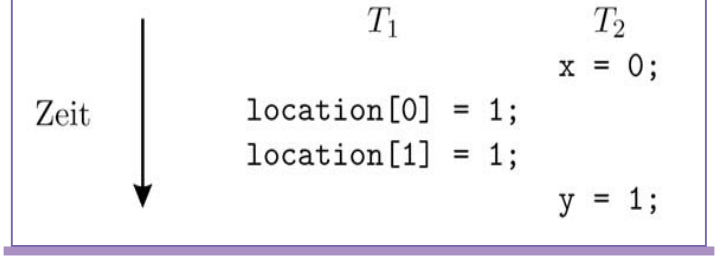
- Egebnis von P1 / P2 ist je nach dem:
  1 / 1 oder 1 / 2 oder 2 / 1 oder **2 / 2**
- Ein Inkrement kann verloren gehen.

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 ya = x.a (=1).
  - P2 schreibt x.a = 0; x.b = 0.
  - P1 liest yb = x.b (=0).
    → P1 hat inkonsistente Kopie von x
- Konflikt schon bei 1 Schreiber und 1 Leser.

Beispiel: Inkonsistenz

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



Beispiel: Update eines Datensatzes

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

A Schlüssel für Kodierung **B** Schlüssel für Dekodierung

```
P0:                          P1:
 ↓                            ↓
r = random();                r = random();
A = create_A( r );           A = create_A( r );
B = create_B ( r );          B = create_B ( r );
 ↓                            ↓
```

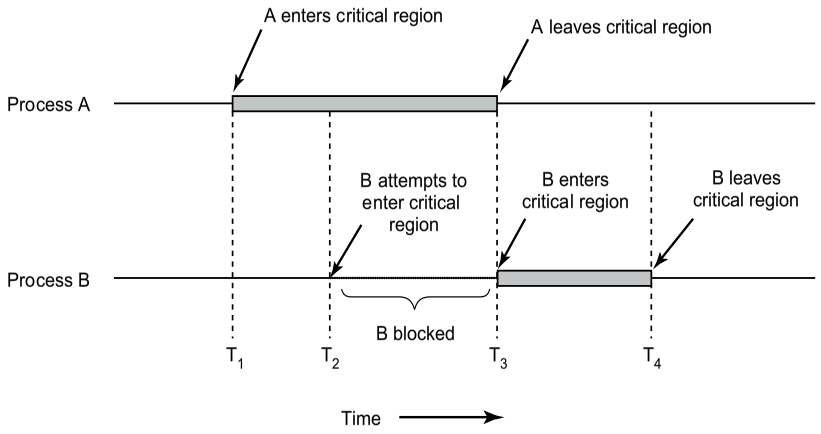
- Möglichkeit, dass A und B von verschiedenen r stammen (und somit nicht zusammenpassen) → **Inkonsistenz**

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

Kritische Abschnitte



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

Zugriffskonflikte: Lösungsansatz

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

```
P0:                 f (int* ip) {          P1:
 ↓                  lock();                 ↓
f ( &z );           *ip=*ip+1;             f ( &z );
 ↓                  unlock();}              ↓
```

Zugriffskonflikte: Lösungsansätze (II)

- Schutz von Daten: Kritische Daten (**Critical Data**)
- Lösung mit **Data Locking**
  - Daten mit Lock:

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

P0:	P1:
↓	↓
lock(&z.lock);	**lock(&z.lock);**
z.counter++;	**z.counter++;**
unlock(&z.lock);	**unlock(&z.lock);**
↓	↓

Granularität

- Wichtiger Gesichtspunkt für Effizienz: **Granularität** (*grain size*)
- 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 nur bei 1-Prozessor-Maschinen
    - oder asymmetrischen Multiprozessoren, bei denen nur Master-Prozessor Interrupts bearbeitet.
  - Anfällig auf Probleme (blockieren, etc.)
  - Für User Prozesse ungeeignet
    - Blockieren aller Ressourcen durch User-Prozess 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
  - Aktives Warten (**Busy Waiting**)

Versuch II (Strict Alternation)

```
      P1:                                P2:
while (1) {                        while (1) {
   while (turn != 1); //wait          while (turn != 2); //wait
   critical_section();                critical_section();
   turn = 2;                          turn = 1;
   rest1();                           rest2();
}                                  }
```

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

Versuch III (Setzen eigenes und Testen fremdes Flag)

```
      P1:                                P2:
bereit[1] = FALSE;                 bereit[2] = FALSE;
while ( ... ) {                    while ( ... ) {
   ...                                ...
   bereit[1] = TRUE;                  bereit[2] = TRUE;
   while (bereit[2] ); //wait         while (bereit[1] ); //wait
   ... (krit. Bereich)                ... (krit. Bereich)
   bereit[1] = FALSE;                 bereit[2] = FALSE;
   ...                                ...
}                                  }
```

- Korrekt, keine strikte Abwechslung vorgeschrieben
- Probleme
  - Deadlock bei gleichzeitigem bereit[.] = TRUE
  - Aktives Warten (**Busy Waiting**)
  - nicht allgemein

Versuch IV (Testen fremdes und Setzen eigenes Flag)

```
      P1:                                P2:
bereit[1] = FALSE;                 bereit[2] = FALSE;
while ( ... ) {                    while ( ... ) {
   ...                                ...
   bereit[1] = TRUE;                  bereit[2] = TRUE;
   while (bereit[2] ); //wait         while (bereit[1] ); //wait
   ... (krit. Bereich)                ... (krit. Bereich)
   bereit[1] = FALSE;                 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; }
```

- Ist korrekt, erfüllt unsere Anforderungen
- Lösung nicht allgemein, da sie nur für 2 Prozesse funktioniert.
- Lösung nur für Prozessnummer aus 1 Byte portabel.

Hardware unterstützte Lösung: TSL

- Die meisten Prozessoren bieten gegenseitigen Ausschluss (**Mutex**) in Form von TSL (*Test and Set Lock*).
- 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)

Mutex als Spin-Lock

- *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

- 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 wartender Prozess höhere Priorität hat, als derjenige, der das Lock hält
    - Lösung durch (temporäre) Vererbung der Priorität. Der Prozess, der das Lock hält, bekommt die Priorität des Wartenden

Mutex als Spin-Lock

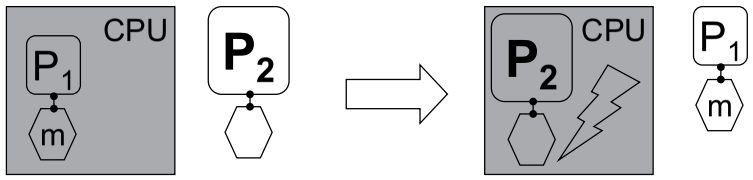
- Mutex für Multiprozessoren
  - bisherige Lösung bei Multiprozessorsystemen problematisch
  - `tsl` führt wegen Schreiben ständig zur Invalidierung in allen Caches → lokale Kopie `lock_p` der Schlossvariable für jeden Prozessor

```
            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`

Prioritäts-Inversion

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



P_2: `while (TSL(m));`

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 benötigt das Betriebssystem!