Betriebssystem-Entwicklung

7. Synchronisierung

Michael Schöttner

Vorschau

Heinrich Heine
Universität Düsseldorf

- Ausgangssituation
- Wettlaufsituationen
- Kritischer Abschnitt
- Wechselseitiger Ausschluss
- Test&Set Maschinen-Instruktion
- Semaphore

Ausgangssituation

Heinrich Heine
Universität Düsseldorf

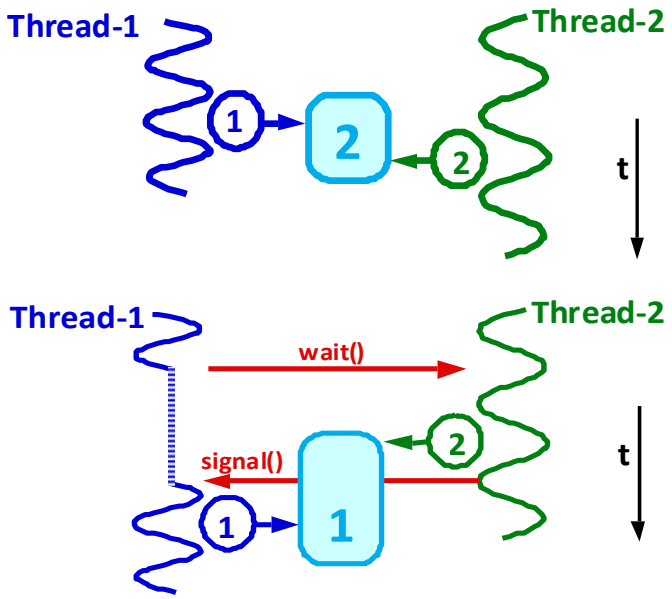
- Interrupts können jederzeit auftreten (sofern diese nicht maskiert sind)
  - Also nicht erst zum Ende einer Quelltextzeile, sondern beispielsweise mitten in der Berechnung eines komplexen Ausdrucks
  - Erfolgen im Interrupt-Handler Zugriffe auf Ressourcen, die auch von Threads genutzt werden, können Wettlaufsituationen (engl. race conditions) entstehen

- Präemptives Multithreading:
  - Hier erfolgt die Umschaltung auf einen anderen Thread mithilfe des Timer-Interrupts, sodass das Gleiche wie oben beschrieben gilt

- Wichtig, bei einem Multicore-Rechner reicht es nicht die Interrupts auf einem Core zu maskieren, da diese dann einfach auf einem anderen Core behandelt werden!

Wettlaufsituation (engl. race condition)

- Wenn nebenläufige Threads auf gemeinsame Variablen schreiben, so ist das Ergebnis nicht deterministisch.
- Synchronisierung der nebenläufigen Ausführung lässt das Resultat deterministisch werden.



Kritischer Abschnitt

Heinrich Heine
Universität Düsseldorf

- = engl. critical region: Programmabschnitte, die auf gemeinsame Variablen zugreifen und deshalb einer Synchronisierung bedürfen.
  - Wechselseitiger Ausschluss gewünscht → max. 1 Thread im kritischen Abschnitt
  - Keine Annahmen bezüglich CPU-Geschwindigkeit, #Cores, ...
  - Fairness: Wartezeit für Eintritt in kritischen Abschnitt muss begrenzt sein.
  - Keine Verklemmungen → Fortschritt garantiert

- Unterschiedliche Programmabschnitte können dieselben Variablen nutzen
  → Nicht Programmabschnitt, sondern Variablen werden geschützt.

Fehlerhafte Lösung für einen krit. Abschnitt

- Wird während Prüfen des Flags umgeschaltet, so können beide Threads den kritischen Abschnitt betreten.
- Problem. Testen und Setzen des Flags geschieht nicht atomar.

```
int busyFlag = 0;
```

Thread 1

```
...
while (busyFlag);


busyFlag = 1;

/* Anweisungen */

busyFlag = 0;
...
```

Umschalten

Thread 2

```
...

while (busyFlag);
busyFlag = 1;


/* Anweisungen */

busyFlag = 0;
...
```

Synchronisierung

- Beispiel Ein Thread inkrementiert jeweils zwei globale Variablen
  - Einmal mit Synchronisierung und einmal ohne.
- Erzeugt der Haupt-Thread mehrere Threads die `my_thread` nebenläufig ausführen, so können wir Lost-Update-Probleme sehen

statischer Mutex

```
#define COUNT  100000

long sync=0;
long asyn=0;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *my_thread(void *param) {
  long zwisch;
  int count = COUNT;

   while ( count-- > 0 ) {
      zwisch=asyn+1; asyn=zwisch;

      pthread_mutex_lock( &mutex );
      zwisch = sync+1;
      sync   = zwisch;
      pthread_mutex_unlock( &mutex );
   }

   printf("asyn: %ld\n", asyn);
   printf("sync: %ld\n", sync);
   return NULL;
}
```

Lost-update Problem

Heinrich Heine
Universität Düsseldorf

- Wenn eine Inkrementierung unterbrochen wird und der andere Thread ebenfalls inkrementiert, kann eine Inkrementierung verloren gehen
  → „i++;“ erfolgt evt. nicht atomar

- Beispiel-Ausgabe des Programms (siehe vorherige Seite):

```
asyn: 18213815
sync: 18761335
asyn: 19452479
sync: 20000000
```

- Ausnahmsweise und je nach Last und „Laune“ des Schedulers im Betriebssystem läuft das Programm auch ohne Verlust:

```
asyn: 19843320
sync: 19843320
asyn: 20000000
sync: 20000000
```

Wechselseitiger Ausschluss

- = engl. mutual exclusion; löst das Problem des kritischen Abschnitts
- Basis hierfür ist eine atomare Test-&-Set-Instruktion:
  - Prinzip: Test = return Speicherwort; Set = setze Speicherwort auf true
  - Bieten alle Prozessoren für Desktop und Server-Betriebssysteme
    - Test&Set verhindert Interrupts am eigenen Core,
      sowie potentielle Zugriffe durch andere Cores oder Busmaster-Geräte
- Abstrakte Implementierung einer Sperre mithilfe von Test&Set

```
int  lock = 0;          /* 0=not locked, 1=locked */

void acquire () {
    while  Test_And_Set (lock_var ) ;  /*  busy polling */
}

void release () {
        lock = 0;
}
```

Test&Set in x86-Assembler

Heinrich Heine
Universität Düsseldorf

```
/*  if old == *ptr then
        *ptr := _new
    return prev

    (function inlining improves speed by avoiding functions calls;
     gnu99 requires 'static' to be combined with 'inline')
*/
static inline unsigned long CAS unsigned long *ptr,
                                unsigned long old,
                                unsigned long _new) {
    unsigned long prev

    /*  AT&T/UNIX assembly syntax
        The 'volatile' keyword after 'asm' indicates that the instruction has important side-effects.
        GCC will not delete a volatile asm if it is reachable.
     */
    asm volatile "lock;"                                  /* prevent race conditions with other cores    */
                 "cmpxchg %1, %2;"                        /* q=64-bit, %1 = _new; %2 = *ptr, constraints */
                  : "=a"(prev)                            /* output: =a: RAX -> prev (%0)                */
                  : "r"(_new), "m"(*ptr), "a"(old)        /* input = %1, %2, %3                          */
                                                          /* (r=register, m=memory, a=accumlator = rax   */
                  : "memory");                            /* ensures assembly block will not be moved by gcc */
    return prev;
}
```

```
unsigned long lock = 0;
unsigned long *ptr = &lock;

void acquire_spinlock(int t) {
    while (CAS(ptr, 0, t) != t) ;
}

void free_spinlock() {
    lock = 0;
}
```

Semaphore

Heinrich Heine
Universität Düsseldorf

- Semaphor: (Wortbedeutung aus dem Griechischen)
  - optischer Telegraph; Mast mit Armen, durch deren Verstellung Zeichen zur Nachrichtenübermittlung weitergeleitet werden; schon im Altertum bekannt
  - Im Eisenbahnwesen auch Bezeichnung für ein Hauptsignal; in der Schifffahrt
  - Zur Anzeige von Windrichtung und Windstärke von der Küste aus.

- Hier besondere Variablen zur Synchronisierung zwischen Threads
  - Im Gegensatz zu Test&Set hier **kein** Busy-Polling

Semaphore

- Variable zur Synchronisierung mit folgenden **atomaren** Operationen:
  - vorgeschlagen durch E. Djikstra, 1968
  - binäre Semaphore mit Werten 0 oder 1
  - zählende Semaphore mit Werten 0 .. n
  - Initialisieren: InitSem(semVar)
  - "Passieren"?: P(semVar)
  - "Vreigeben": V(semVar)

```
if (semVar > 0)
        semVar = semVar - 1
else {
        warten auf V( semVar )
}
```

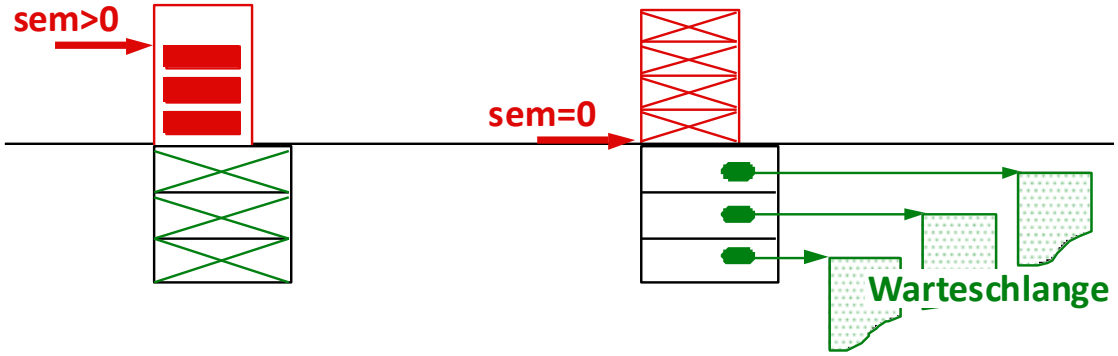
```
if ( Thread wartet auf semVar )
        anstossen( Thread )
else
        semVar = semVar + 1;
```

- Originalsprache Holländisch ...

Semaphore

- Funktionen P&V werden intern mithilfe der Test & Set-Instruktionen realisiert
  - Entweder direkt durch den Compiler oder das Betriebssystem

- Semaphore haben intern eine Queue zur Verwaltung wartender Threads.
  - Einträge verweisen auf Thread-Control-Block (TCB)
  - alle wartenden Threads sind blockiert → kein Busy-Waiting



Leser/Schreiber mit Semaphore

Heinrich Heine
Universität Düsseldorf

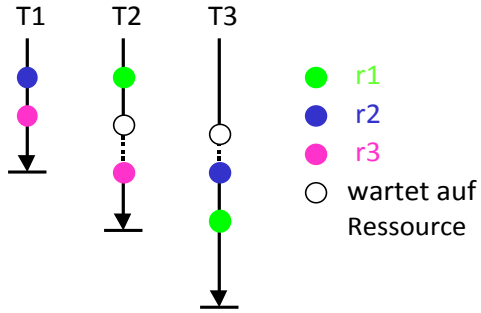
- N Leser oder 1 Schreiber parallel erlaubt.

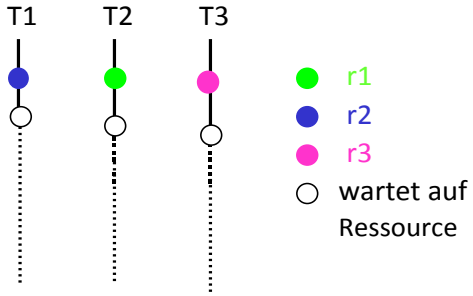
```
int       readcount=0;   /* Anzahl aktiver Leser */
semaphore mutex=1, wrt=1;

Leser() {
  P(mutex);              /* mit Leser sync. */
  readcount++;
  if (readcount==1)      /* kein Leser aktiv? */
     P(wrt);             /* Schreiber blockieren */
  V(mutex);

  reading();

  P(mutex);              /* mit Leser sync. */
  readcount --;
  if (readcount==0)      /* kein Leser mehr aktiv? */
     V(wrt);             /* Schreiber zulassen */
  V(mutex);
}
```

```
Schreiber() {
  P(wrt);     /* nur 1 Schreiber */
  writing();
  V(wrt);
}
```

Verklemmung (engl. deadlock)

- Zwei oder mehrere Threads machen keinen Fortschritt, weil sie Ressourcen besitzen, die von einem anderen Thread benötigt würden.
- Drei Threads verlangen erst eine Ressource und später noch eine zweite.
  - Günstiger Verlauf → alle 3 Threads terminieren:
  - Ungünstig, kein Thread terminiert
    → Verklemmung ist möglich, aber nicht zwingend.





Bedingungen für eine Verklemmung

Heinrich Heine
Universität Düsseldorf

1. **Wechselseitiger Ausschluss (engl. mutual exclusion)**
   Betroffene Ressource ist nicht gemeinsam nutzbar.
2. **Halten & Warten (engl. hold and wait)**
   Wartender Thread besitzt Ressource und wartet auf weitere.
3. **Keine Verdrängung (engl. no preemption)**
   Ressourcen können einem Thread nicht entzogen werden.
4. **Zirkuläre Wartesituation (engl. circular wait)**

Sobald alle vier Bedingungen erfüllt sind, liegt eine Verklemmung vor

Sonderfälle bei einem Mutex

Heinrich Heine
Universität Düsseldorf

- Rekursive Funktionen, welche einen Mutex verwenden
  - Es gibt Mutex-Implementierungen die es erlauben, dass der gleiche Thread mehrfach den gleichen Mutex erhält, beispielsweise in rekursiven Funktionen
    - Intern wird dazu ein Zähler geführt (wie bei einer Semaphore).
    - Der Mutex wird erst beim letzten Unlock-Aufruf freigegeben, wenn der interne Zähler auf 0 fällt
  - Falls die Mutex-Implementierung dies nicht erlaubt, führt dies u.U. zu einer Verklemmung mit nur einem Mutex

- Interrupts und Threads verwenden den gleichen Mutex
  - Beispielsweise für den Zugriff auf den Seicher-Allokator
  - Auch hier kann eine Verklemmung mit nur einem Mutex entstehen
    - Ein Thread hält den Mutex und wird vor der Freigabe durch einen Interrupt unterbrochen
    - Der Interrupt bekommt den Mutex dann nicht und hängt, wodurch der andere Thread auch nicht weiterlaufen kann