5. Synchronisation (1)

5. Synchronisation

Einführung (1)

- Es gibt Prozesse (oder Threads oder Kernel-Funktionen) mit gemeinsamem Zugriff auf bestimmte Daten, z. B.
  - Threads des gleichen Prozesses: gemeinsamer Speicher
  - Prozesse mit gemeinsamer Memory-Mapped-Datei
  - Prozesse / Threads öffnen die gleiche Datei zum Lesen / Schreiben
  - SMP-System: Scheduler (je einer pro CPU) greifen auf gleiche Prozesslisten / Warteschlangen zu

Einführung (2)

- Synchronisation: Probleme mit „gleichzeitigem“ Zugriff auf Datenstrukturen
- Beispiel: Zwei Prozesse erhöhen einen Zähler

```
erhoehe_zaehler( )
{
  w=read(Adresse);
  w=w+1;
  write(Adresse,w);
}
```

Ausgangssituation: w=10

P1:	P2:
w=read(Adresse); // 10	
w=w+1; // 11	
	w=read(Adresse); // 10
	w=w+1; // 11
	write(Adresse,w); // 11
write(Adresse,w); // 11 !!	

Ergebnis nach P1, P2: w=11 – nicht 12!

Einführung (3)

- Gewünscht wäre eine der folgenden Reihenfolgen:

Ausgangssituation: w=10

P1:	P2:
w=read(Adr); // 10	
w=w+1; // 11	
write(Adr,w); // 11	
	w=read(Adr); // 11
	w=w+1; // 12
	write(Adr,w); // 12

Ergebnis nach P1, P2: w=12

Ausgangssituation: w=10

P1:	P2:
	w=read(Adr); // 10
	w=w+1; // 11
	write(Adr,w); // 11
w=read(Adr); // 11	
w=w+1; // 12	
write(Adr,w); // 12	

Ergebnis nach P1, P2: w=12

Einführung (4)

- Ursache: `erhoehe_zaehler()` arbeitet nicht **atomar**:
  - Scheduler kann die Funktion unterbrechen
  - Funktion kann auf mehreren CPUs gleichzeitig laufen
- Lösung: Finde alle Code-Teile, die auf gemeinsame Daten zugreifen, und stelle sicher, dass immer nur ein Prozess auf diese Daten zugreift (gegenseitiger Ausschluss, mutual exclusion)

Einführung (5)

- Analoges Problem bei Datenbanken:

```
exec sql CONNECT ...
exec sql SELECT kontostand INTO $var FROM KONTO
         WHERE kontonummer = $knr
$var = $var - abhebung
exec sql UPDATE Konto SET kontostand = $var
         WHERE kontonummer = $knr
exec sql DISCONNECT
```

  Bei parallelem Zugriff auf gleichen Datensatz kann es zu Fehlern kommen
- Definition der (Datenbank-) **Transaktion**, die u.a. **atomar und isoliert** erfolgen muss

Einführung (6)

Race Condition:

- Mehrere parallele Threads / Prozesse nutzen eine gemeinsame Ressource
- Zustand hängt von Reihenfolge der Ausführung ab
- Race: die Threads liefern sich „ein Rennen“ um den ersten / schnellsten Zugriff

Einführung (7)

Warum Race Conditions vermeiden?

- Ergebnisse von parallelen Berechnungen sind nicht eindeutig (d. h. potenziell falsch)
- Bei Programmtests könnte (durch Zufall) immer eine „korrekte“ Ausführreihenfolge auftreten; später beim Praxiseinsatz dann aber gelegentlich eine „falsche“.
- Race Conditions sind auch Sicherheitslücken

Einführung (8)

Race Condition als Sicherheitslücke

- Wird von Angreifern genutzt
- Einfaches Beispiel:

```
read(command)
f=open("/tmp/script","w")
write(f,command)
f.close()
chmod("/tmp/script","a+x")
system("/tmp/script")
```

Angreifer ändert Dateiinhalt vor dem chmod; Programm läuft mit Rechten des Opfers

Einführung (9)

- Idee: Zugriff via Lock auf einen Prozess (Thread, ...) beschränken:

```
erhoehe_zaehler( ) {
  flag=read(Lock);
  if (flag == LOCK_UNSET) {
    set(Lock);
    /*  Anfang des „kritischen Bereichs“  */
    w=read(Adresse);  w=w+1;
    write(Adresse,w);
    /*  Ende des „kritischen Bereichs“  */
    release(Lock);
  };
}
```

- Problem: Lock-Variable nicht geschützt

Einführung (10)

- Nicht alle Zugriffe auf Daten sind problematisch:
  - Gleichzeitiges Lesen von Daten stört nicht
  - Prozesse, die „disjunkt“ sind (d.h.: die keine gemeinsamen Daten haben) können ohne Schutz zugreifen
- Sobald mehrere Prozesse/Threads/... gemeinsam auf ein Objekt zugreifen – und mindestens einer davon schreibend –, ist das Verhalten des Gesamtsystems **unvorhersehbar** und **nicht reproduzierbar**.

Inhaltsübersicht: Synchronisation

Kritische Abschnitte (1)

- Programmteil, der auf gemeinsame Daten zugreift
  - Müssen nicht verschiedene Programme sein: auch mehrere Instanzen des gleichen Programms!
- Block zwischen erstem und letztem Zugriff
- Nicht den Code schützen, sondern die Daten
- Formulierung: kritischen Bereich „betreten“ und „verlassen“

Kritische Abschnitte (2)

- Anforderung an parallele Threads:
  - Es darf maximal ein Thread gleichzeitig im kritischen Abschnitt sein
  - Kein Thread, der außerhalb kritischer Bereiche ist, darf einen anderen blockieren
  - Kein Thread soll ewig auf das Betreten eines kritischen Bereichs warten
  - Deadlocks sollen vermieden werden (z. B.: zwei Prozesse sind in verschiedenen krit. Bereichen und blockieren sich gegenseitig)

Gegenseitiger Ausschluss

- Tritt nie mehr als ein Thread gleichzeitig in den kritischen Bereich ein, heißt das „**gegenseitiger Ausschluss**“ (englisch: mutual exclusion, kurz: mutex)
- Es ist Aufgabe der Programmierer, diese Bedingung zu garantieren
- Das Betriebssystem bietet Hilfsmittel, mit denen gegenseitiger Ausschluss durchgesetzt werden kann, schützt aber nicht vor Programmier-fehlern

Programmtechnische Synchr. (1)

1. Versuch: Lock-Variable (wie in Einführung)

- Lock-Variable auf *false* initialisiert
- Prozess, der krit. Bereich betreten will, prüft lock==*false* – wenn Bedingung erfüllt ist:
  - lock=*true* setzen,
  - Bereich betreten und wieder verlassen
  - lock=*false* (zurück)setzen

```
while ( lock ) {
  /* warten */
} ;
lock=true;
kritischer_bereich();
lock=false;
```

- Verschiebt Problem nur auf die Lock-Variable

Programmtechnische Synchr. (2)

2. Versuch: Nächsten Prozess speichern

- Lock-Variable *turn* legt fest, welcher Prozess als nächster den krit. Bereich betreten darf:

```
while (true) {
  while (turn != 1) {
    /* warten */
  };
  kritischer_bereich();
  turn=2;
}
```

```
while (true) {
  while (turn != 2) {
    /* warten */
  };
  kritischer_bereich();
  turn=1;
}
```

- Verhindert Race Conditions
- Aber: kritischer Bereich kann nur abwechselnd betreten werden

Programmtechnische Synchr. (3)

3. Versuch: Für jeden Thread separate Variable, die „Thread ist in krit. Bereich“ anzeigt

```
while (true) {
  C1=true;
  while (C2) {
    /* wait */
  };
  kritischer_bereich();
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    /* wait */
  };
  kritischer_bereich();
  C2=false;
}
```

- Verhindert Race Conditions
- Deadlock tritt auf, wenn beide gleichzeitig den kritischen Bereich betreten wollen

Programmtechnische Synchr. (4)

4. Versuch (Dekker): Kombination aus Lock-Variablen und wechselnder Reihenfolge

```
while (true) {
  C1=true;
  while (C2) {
    if (turn != 1) {
      C1=false;
      while (turn != 1) {
        /* wait */
      };
      C1=true;
    };
  };
  kritischer_bereich();
  turn=2;
  C1=false;
}
```

```
while (true) {
  C2=true;
  while (C1) {
    if (turn != 2) {
      C2=false;
      while (turn != 2) {
        /* wait */
      };
      C2=true;
    };
  };
  kritischer_bereich();
  turn=1;
  C2=false;
}
```


Programmtechnische Synchr. (5)

Alternative: Petersons Algorithmus

```
C1=true;
turn=2;
while (C2 && turn==2)
  /* warten */;
kritischer_abschnitt();
C1=false;
```

```
C2=true;
turn=1;
while (C1 && turn==1)
  /* warten */;
kritischer_abschnitt();
C2=false;
```

Programmtechnische Synchr. (6)

Petersons Algorithmus –
gegenseitiger Ausschluss gewährt:

- Wenn P_1 C_1 auf *true* setzt, kann P_2 seinen kritischen Bereich nicht mehr betreten
- War P_2 schon im kritischen Bereich, dann war C_2 schon *true*, d.h., P_1 durfte nicht in seinen kritischen Bereich

Programmtechnische Synchr. (7)

Petersons Algorithmus –
keine gegenseitige Blockade:

Angenommen, P1 ist in der While-Schleife blockiert, d. h.: C2=true und turn=2 (P1 kann den krit. Bereich betreten, wenn eine der Bedingungen nicht mehr gilt, also entweder C2=false oder turn=1 wird)

Dann nur 2 Möglichkeiten:

- P2 wartet auf Einlass in den krit. Bereich -> das kann nicht sein, denn mit turn=2 darf P2 in seinen kritischen Bereich
- P2 nutzt wiederholt den krit. Bereich, monopolisiert Zugang zu ihm -> das kann auch nicht sein, weil P2 vor dem Betreten die turn-Variable auf 1 setzt (und damit P1 den Vortritt lassen würde)

Test-and-Set-Lock (TSL) (1)

- Maschineninstruktion (z.B. mit dem Namen TSL = Test and Set Lock), die atomar eine Lock-Variable liest und setzt, also ohne dazwischen unterbrochen werden zu können.

```
enter:
  tsl register, flag  ; Variablenwert in Register kopieren und
                          ; dann Variable auf 1 setzen
  cmp register, 0     ; War die Variable 0?
  jnz enter           ; Nicht 0: Lock war gesetzt, also Schleife
  ret

leave:
  mov   flag, 0       ; 0 in flag speichern: Lock freigeben
  ret
```


Test-and-Set-Lock (TSL) (2)

- TSL muss zwei Dinge leisten:
  - Interrupts ausschalten, damit der Test-und-Setzen-Vorgang nicht durch einen anderen Prozess unterbrochen wird
  - Im Falle mehrerer CPUs den Speicherbus sperren, damit kein Prozess auf einer anderen CPU (deren Interrupts nicht gesperrt sind!) auf die gleiche Variable zugreifen kann

Aktives und passives Warten (1)

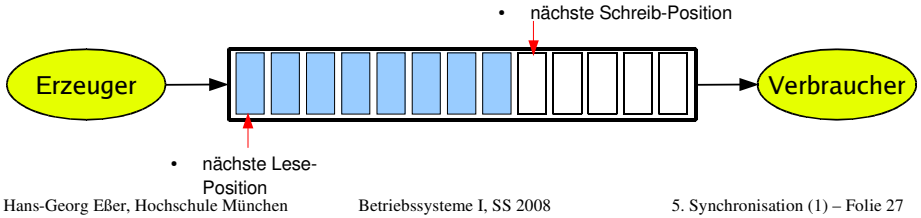
- Aktives Warten (busy waiting):
  - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt.
  - Der Thread ist bereit und belegt die CPU.
  - Die Variable muss von einem anderen Thread gesetzt werden.
    - (Großes) Problem, wenn der andere Thread endet.
    - (Großes) Problem, wenn der andere Thread – z. B. wegen niedriger Priorität - nicht dazu kommt, die Variable zu setzen.

Aktives und passives Warten (2)

- Passives Warten (sleep and wake):
  - Ein Thread blockiert und wartet auf ein Ereignis, das ihn wieder in den Zustand „bereit“ versetzt.
  - Der blockierte Thread verschwendet keine CPU-Zeit.
  - Ein anderer Thread muss das Eintreten des Ereignisses bewirken.
    - (Kleines) Problem, wenn der andere Thread endet.
  - Bei Eintreten des Ereignisses muss der blockierte Thread geweckt werden, z. B.
    - explizit durch einen anderen Thread,
    - durch Mechanismen des Betriebssystems.

Erzeuger-Verbraucher-Problem (1)

- Beim Erzeuger-Verbraucher-Problem (producer consumer problem, bounded buffer problem) gibt es zwei kooperierende Threads:
  - Der Erzeuger speichert Informationen in einem beschränkten Puffer.
  - Der Verbraucher liest Informationen aus diesem Puffer.



Erzeuger-Verbraucher-Problem (2)

- **Synchronisation**
  - **Puffer nicht überfüllen:**
    Wenn der Puffer voll ist, muss der Erzeuger warten, bis der Verbraucher eine Information aus dem Puffer abgeholt hat, und erst dann weiter arbeiten.
  - **Nicht aus leerem Puffer lesen:**
    Wenn der Puffer leer ist, muss der Verbraucher warten, bis der Erzeuger eine Information im Puffer abgelegt hat, und erst dann weiter arbeiten.

Erzeuger-Verbraucher-Problem (3)

- Realisierung mit passivem Warten:
  - Eine gemeinsam benutzte Variable „count“ zählt die belegten Positionen im Puffer.
  - Wenn der Erzeuger eine Information einstellt und der Puffer leer war (count == 0), weckt er den Verbraucher;
    bei vollem Puffer blockiert er.
  - Wenn der Verbraucher eine Information abholt und der Puffer voll war (count == max), weckt er den Erzeuger;
    bei leerem Puffer blockiert er.

Erzeuger-Verbraucher-Problem mit sleep / wake

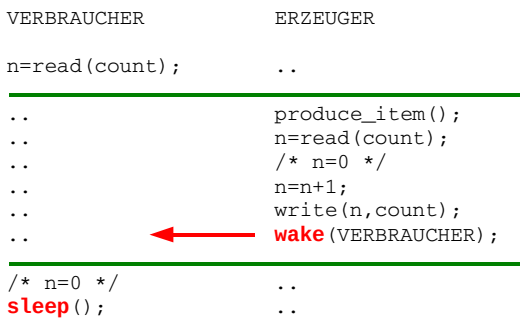
```
#define  N  100                        // Anzahl der Plätze im Puffer
int  count = 0;                        // Anzahl der belegten Plätze im Puffer

producer () {
  while (TRUE) {                       // Endlosschleife
    produce_item (item);               // Erzeuge etwas für den Puffer
    if (count == N) sleep();           // Wenn Puffer voll: schlafen legen
    enter_item (item);                 // In den Puffer einstellen
    count = count + 1;                 // Zahl der belegten Plätze inkrementieren
    if (count == 1) wake(consumer);    // war der Puffer vorher leer?
  }
}

consumer () {
  while (TRUE) {                       // Endlosschleife
    if (count == 0) sleep();           // Wenn Puffer leer: schlafen legen
    remove_item (item);                // Etwas aus dem Puffer entnehmen
    count = count - 1;                 // Zahl der belegten Plätze dekrementieren
    if (count == N-1) wake(producer); // war der Puffer vorher voll?
    consume_item (item);               // Verarbeiten
  }
}
```


Deadlock-Problem bei sleep / wake (1)

- Das Programm enthält eine race condition, die zu einem Deadlock führen kann, z. B. wie folgt:
  - Verbraucher liest Variable count, die den Wert 0 hat.
  - Kontextwechsel zum Erzeuger.
  - Erzeuger stellt etwas in den Puffer ein, erhöht count und weckt den Verbraucher, da count vorher 0 war.
  - Verbraucher legt sich schlafen, da er für count noch den Wert 0 gespeichert hat (der zwischenzeitlich erhöht wurde).
  - Erzeuger schreibt den Puffer voll und legt sich dann auch schlafen.

Deadlock-Problem bei sleep / wake (2)

- **Problemursache:**
  Wakeup-Signal für einen – noch nicht – schlafenden Prozess wird ignoriert
- Falsche Reihenfolge
- Weckruf „irgendwie“ für spätere Verwendung aufbewahren...



Deadlock-Problem bei sleep / wake (3)

- Lösungsmöglichkeit: Systemaufrufe *sleep* und *wake* verwenden ein „wakeup pending bit“:
  - Bei *wake()* für einen nicht schlafenden Thread dessen wakeup pending bit setzen.
  - Bei *sleep()* das wakeup pending bit des Threads überprüfen – wenn es gesetzt ist, den Thread nicht schlafen legen.

  Aber: Lösung lässt sich nicht verallgemeinern (mehrere zu synchronisiernende Prozesse benötigen evtl. zusätzliche solche Bits)

Semaphore (1)

Ein **Semaphor** ist eine Integer- (Zähler-) Variable, die man wie folgt verwendet:

- Semaphor hat festgelegten Anfangswert N („Anzahl der verfügbaren Ressourcen“).
- Beim Anfordern eines Semaphors (P- oder **Wait**-Operation):
  - Semaphor-Wert um 1 erniedrigen, falls er >0 ist,
  - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert 0 ist.

Semaphore (2)

- Bei Freigabe eines Semaphors (V- oder **Signal**-Operation):
  - einen Thread aus der Warteschlange wecken, falls diese nicht leer ist,
  - Semaphor-Wert um 1 erhöhen (wenn es keinen auf den Semaphor wartenden Thread gibt)
- Code sieht dann immer so aus:

```
wait (&sem);
/* Code, der die Ressource nutzt */
signal (&sem);
```


Semaphore (3)

- Variante: Negative Semaphor-Werte
  - Semaphor zählt Anzahl der wartenden Threads
  - Anfordern (WAIT):
    - Semaphor-Wert um 1 erniedrigen (~~falls er positiv ist~~)
    - Thread blockieren und in eine Warteschlange einreihen, wenn der Semaphor-Wert ≤ 0 ist.
  - Freigabe (SIGNAL):
    - Thread aus der Warteschlange wecken (falls nicht leer)
    - Semaphor-Wert um 1 erhöhen (~~wenn es keinen auf den Semaphor wartenden Thread gibt~~)

Semaphore (4)

Standard-Variante: Semaphor kann nur Werte ≥ 0 annehmen

```
wait (sem) {
  if (sem>0)
    sem--;
    else BLOCK_CALLER;
}
```

```
signal (sem) {
  if (P in QUEUE(sem)) {
    wakeup (P);
    remove (P, QUEUE);
  }
  else sem++;
}
```

Variante: Semaphor auch negativ, speichert Größe der Warteschlange

```
wait (sem) {
  if (sem<1)
    BLOCK_CALLER;
  sem--;
}
```

```
signal (sem) {
  if (P in QUEUE(sem)) {
    wakeup (P);
    remove (P, QUEUE); }
  sem++;
}
```


Mutexe (1)

- **Mutex:** boolesche Variable (true/false), die den Zugriff auf gemeinsam genutzte Daten synchronisiert
  - true: Zugang erlaubt
  - false: Zugang verboten
- **blockierend:** Ein Thread, der sich Zugang verschaffen will, während ein anderer Thread Zugang hat, blockiert ➞ Warteschlange
- Bei Freigabe:
  - Warteschlange enthält Threads ➞ einen wecken
  - Warteschlange leer: Mutex auf true setzen

Mutexe (2)

- **Mutex** (**mut**ual **ex**clusion) = **binärer Semaphor**, also ein Semaphor, der nur die Werte 0 / 1 annehmen kann

```
wait (mutex) {
  if (mutex==1)
    mutex=0;
    else BLOCK_CALLER;
}
```

```
signal (mutex) {
  if (P in QUEUE(mutex)) {
    wakeup (P);
    remove (P, QUEUE);
  }
  else mutex=1;
}
```

- Neue Interpretation: wait ➞ lock
  signal ➞ unlock
- Mutexe für exklusiven Zugriff (kritische Bereiche)

Blockieren?

- Betriebssysteme können Mutexe und Semaphoren **blockierend** oder **nicht-blockierend** implementieren
- blockierend:
  wenn der Versuch, den Zähler zu erniedrigen, scheitert
  ➞ warten
- nicht blockierend:
  wenn der Versuch scheitert
  ➞ vielleicht etwas anderes tun

Atomare Operationen

- Bei Mutexen / Semaphoren müssen die beiden Operationen wait() und signal() **atomar** implementiert sein:

  Während der Ausführung von wait() / signal() darf kein anderer Prozess an die Reihe kommen

Warteschlangen

- Mutexe / Semaphore verwalten Warteschlangen (der Prozesse, die schlafen gelegt wurden)
- Beim Aufruf von signal() muss evtl. ein Prozess geweckt werden
- Auswahl des zu weckenden Prozesses ist ein ähnliches Problem wie die Prozess-Auswahl im Scheduler
  - FIFO: **starker** Semaphor / Mutex
  - zufällig: **schwacher** Semaphor / Mutex

Erzeuger-Verbraucher-Problem mit Semaphoren und Mutexen

```
typedef int semaphore;
semaphore mutex = 1;          // Kontrolliert Zugriff auf Puffer
semaphore empty = N;          // Zählt freie Plätze im Puffer
semaphore full = 0;           // Zählt belegte Plätze im Puffer

producer() {
  while (TRUE) {              // Endlosschleife
    produce_item(item);       // Erzeuge etwas für den Puffer
    wait (empty);             // Leere Plätze dekrementieren bzw. blockieren
    wait (mutex);             // Eintritt in den kritischen Bereich
    enter_item (item);        // In den Puffer einstellen
    signal (mutex);           // Kritischen Bereich verlassen
    signal (full);            // Belegte Plätze erhöhen, evtl. consumer wecken
  }
}

consumer() {
  while (TRUE) {              // Endlosschleife
    wait (full);              // Belegte Plätze dekrementieren bzw. blockieren
    wait (mutex);             // Eintritt in den kritischen Bereich
    remove_item(item);        // Aus dem Puffer entnehmen
    signal (mutex);           // Kritischen Bereich verlassen
    signal (empty);           // Freie Plätze erhöhen, evtl. producer wecken
    consume_entry (item);     // Verbrauchen
  }
}
```