5. Synchronisation (2)

5. Synchronisation
5.3 Synchr.-Methoden

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)

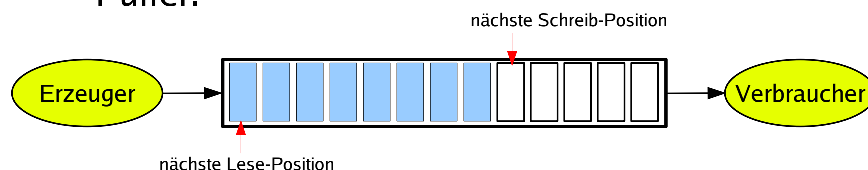
- Aktives Warten (busy waiting):
  - Ausführen einer Schleife, bis eine Variable einen bestimmten Wert annimmt.
  - Der Thread ist bereit und belegt die CPU.
  - Die Variable muss von einem anderen Thread gesetzt werden.
    - (Großes) Problem, wenn der andere Thread endet.
    - (Großes) Problem, wenn 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...

```
VERBRAUCHER          ERZEUGER

n=read(count);       ..
-------------------------------------------
..                   produce_item();
..                   n=read(count);
..                   /* n=0 */
..                   n=n+1;
..                   write(n,count);
..        <------    wake(VERBRAUCHER);
-------------------------------------------
/* n=0 */            ..
sleep();             ..
```


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 synchronisierende 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 positiv 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 $\leq$ 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 $\geq$ 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++;
}
```

Semaphore (5): Beispiel

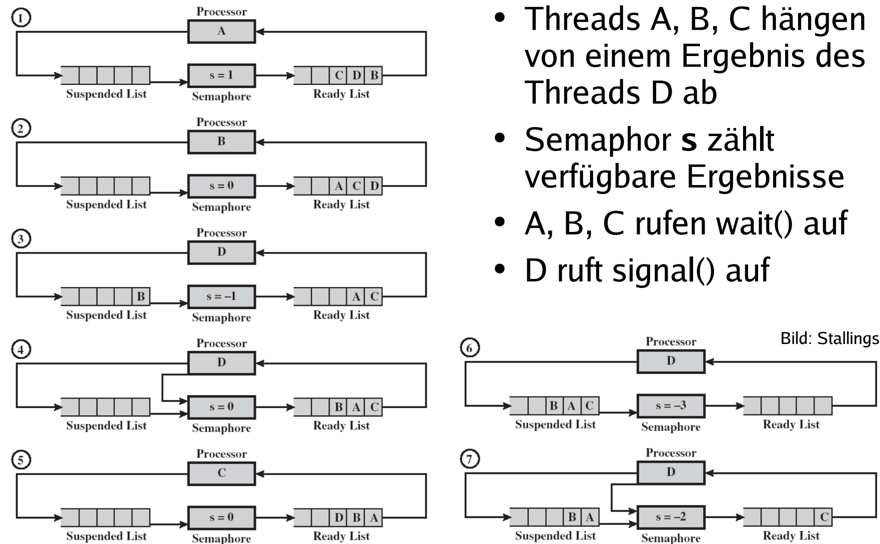


Bild: Stallings

- Threads A, B, C hängen von einem Ergebnis des Threads D ab
- Semaphor **s** zählt verfügbare Ergebnisse
- A, B, C rufen wait() auf
- D ruft signal() auf

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

Erzeuger-Verbraucher-Problem mit Semaphoren und Mutexen

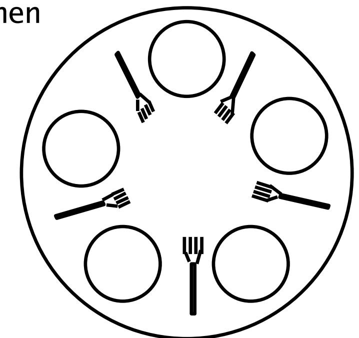
```
typedef int semaphore;
semaphore mutex = 1;        // Kontrolliert Zugriff auf Puffer
semaphore empty = N;        // Zählt freie Plätze im Puffer
semaphore full = 0;         // Zählt belegte Plätze im Puffer

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
  }
}
```


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

Philosophenproblem (1)

- Fünf Philosophen an einem Tisch (Dijkstra 1965)
  - Jeder Philosoph hat vor sich einen Teller Spaghetti.
  - Zwischen je zwei Tellern liegt eine Gabel.
  - Jeder Philosoph wechselt ab zwischen Denken und Essen.
  - Zum Essen benötigt ein Philosoph die beiden Gabeln (bzw. Eßstäbchen) rechts und links von seinem Teller.



- Aufgabe: – Philosophen nicht verhungern lassen
  – und maximale Parallelität erreichen

Philosophenproblem (2)

- Erste Idee:

```
#define  N  5                  // Anzahl der Philosophen
philosopher (int i) {          // i: Nummer des Philosophen (0...N-1)
  while (TRUE)  {
    think();                   // Der Philosoph denkt
    take_fork(i);              // Linke Gabel aufnehmen
    take_fork( (i+1)%N );      // Rechte Gabel aufnehmen (%=modulo)
    eat();                     // Der Philosoph isst
    put_fork (i);              // Linke Gabel zurücklegen
    put_fork ( (i+1)%N );      // Rechte Gabel zurücklegen
  }
}
```

  - take_fork(i) blockiert den Thread, bis die Gabel frei ist
  - Lösung ist falsch: Deadlock, wenn alle Philosophen gleichzeitig die linke Gabel nehmen

Philosophenproblem (3)

- 1. Korrekturversuch:
  - Jeder Philosoph prüft (nach dem Aufnehmen der linken Gabel), ob die rechte verfügbar ist
  - Ist die rechte Gabel nicht da, legt er seine linke zurück, wartet kurz und nimmt sie wieder
- Wenn die Wartezeit zufällig ist, könnte das „oft" funktionieren (nicht gut genug)
- Bei gleicher Wartezeit könnte Endlosschleife enstehen (alle Phil. nehmen linke Gabel, legen sie wieder hin, nehmen sie wieder etc.) ➞ „starvation"

Philosophenproblem (4)

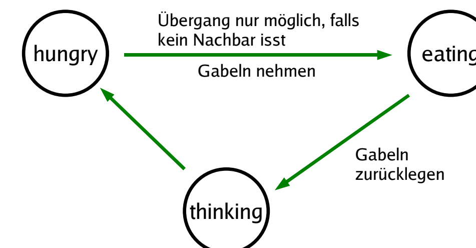
- 2. Korrekturversuch:
  - Ganzer Block vom Aufnehmen der ersten Gabel bis zum Ablegen aller Gabeln durch Mutex schützen, also

```
while (TRUE)  {
    think();
    wait (mutex);       // Anfang kritischer Bereich
    take_fork(i);
    take_fork( (i+1)%N );
    eat();
    put_fork (i);
    put_fork ( (i+1)%N );
    signal (mutex);     // Ende kritischer Bereich
  }
```

  - OK, aber nicht effizient: Es kann immer nur ein Philosoph essen – fünf Gabeln reichen für zwei

Philosophenproblem (5)

- Korrekte Lösung:
  - Zustände der Philosophen in Array state[ ] speichern



  - Semaphor sem[i] für jeden Philosophen: blockiert, wenn nicht beide Gabeln verfügbar sind

Philosophenproblem (6)

```
#define N 5                 // Anzahl der Philosophen
#define LEFT  (i-1)%N       // Nummer des linken Nachbarn
#define RIGHT (i+1)%N       // Nummer des rechten Nachbarn
#define THINKING 0          // der Philosoph denkt
#define HUNGRY   1          // Philosoph versucht, Gabeln zu nehmen
#define EATING   2          // der Philosoph isst

typedef int semaphore;      // Semaphore sind besondere "int"
int       state[N];         // Vektor für Zustände
semaphore mutex=1;          // Semaphor zum gegenseitigen Ausschluss
                            // beim Zugriff auf den Vektor state
semaphore sem[N]={0};       // ein Semaphor pro Philosoph

philosopher (int i)         // i: welcher Philosoph (0 bis N-1)
{
  while (TRUE)  {           // Endlosschleife
    think ();               // der Philosoph denkt
    take_forks (i);         // nimm beide Gabeln auf oder blockiere
    eat ();                 // iss die Nudeln
    put_forks (i);          // lege beide Gabeln zurück
  }
}
```


Philosophenproblem (7)

```
take_forks (int i) {      // i: welcher Philosoph (0 bis N-1)
  wait (mutex);           // kritischen Abschnitt betreten
  state[i] = HUNGRY;      // der Philosoph ist hungrig
  test (i);               // versuche beide Gabeln zu bekommen
  signal (mutex);         // Kritischen Abschnitt verlassen
  wait (sem[i]);          // Blockieren, falls Gabeln nicht frei
}

put_forks (int i) {       // i: welcher Philosoph (0 bis N-1)
  wait (mutex);           // kritischen Abschnitt betreten
  state[i] = THINKING;    // der Philosoph beendet das Essen
  test (LEFT);            // Testen, ob linker Phil. essen will + kann
  test (RIGHT);           // Testen, ob rechter Phil. essen will + kann
  signal (mutex);         // Kritischen Abschnitt verlassen
}

test (int i) {            // Testen, ob Philosoph i essen will + kann
  if ( state[i]     == HUNGRY &&
       state[LEFT]  != EATING &&
       state[RIGHT] != EATING ) {
    state[i] = EATING;
    signal (sem[i]);      // Philosoph i kann jetzt essen, darum wecken
  }
}
```


Beispiel: Phil. 2 und 3 wollen essen

```
// i=2, sem[2]=0

think();
take_forks (2);
   wait (mutex);
   state[2] = HUNGRY;
   test (2);
      state[2] == HUNGRY?  yes
      state[1] != EATING?  yes
      state[3] != EATING?  yes
      ->
      state[2] = EATING;
      signal (sem[2]);   // sem[2]=1
   signal (mutex);
   wait (sem[2]);   // Sem. ist 1, wird 0
eat ();
put_forks (2);
   wait (mutex);
   state[2] = THINKING;
   test (1);  // ggf. andere aufwecken
   test (3);
   signal (mutex);
loop...
```

```
// i=3, sem[3]=0

think();
take_forks (3);
   wait (mutex);
   state[3] = HUNGRY;
   test (3);
      state[3] == HUNGRY?  yes
      state[2] != EATING?  NO !!
      state[4] != EATING?  yes
      -> nichts tun (kein signal()-Aufruf)
   signal (mutex);
   wait (sem[3]);   // ist 0, blockieren!

   [blockiert, solange Phil. 2 isst]

eat ();
put_forks (3);
   wait (mutex);
   state[3] = THINKING;
   test (2);  // ggf. andere aufwecken
   test (4);
   signal (mutex);
loop...
```