3 Interprozesskommunikation

IPC

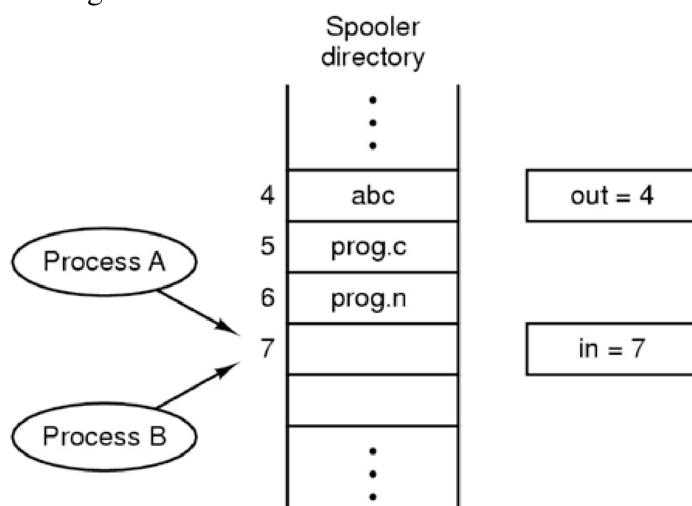
- Wie werden Informationen weitergereicht?
- Verhindern, dass sich Prozesse ins Gehege kommen
- Was passiert bei Abhängigkeiten?

Informationsaustauschmethoden

- Pipes (i. Allg. unidirektional)
- Sockets (Netzverbindungen, aber auch lokal)
- Gemeinsame Dateien/Verzeichnisse
- Shared Memory (explizit gemeinsam genutzter Speicher)
- Message Passing

Kritische Abschnitte

Race Condition: Eine Situation, bei der zwei Prozesse/Threads auf einen gemeinsamen Speicher zugreifen und bei der das Ergebnis von der Ausführungsreihenfolge der Zugriffe abhängt.

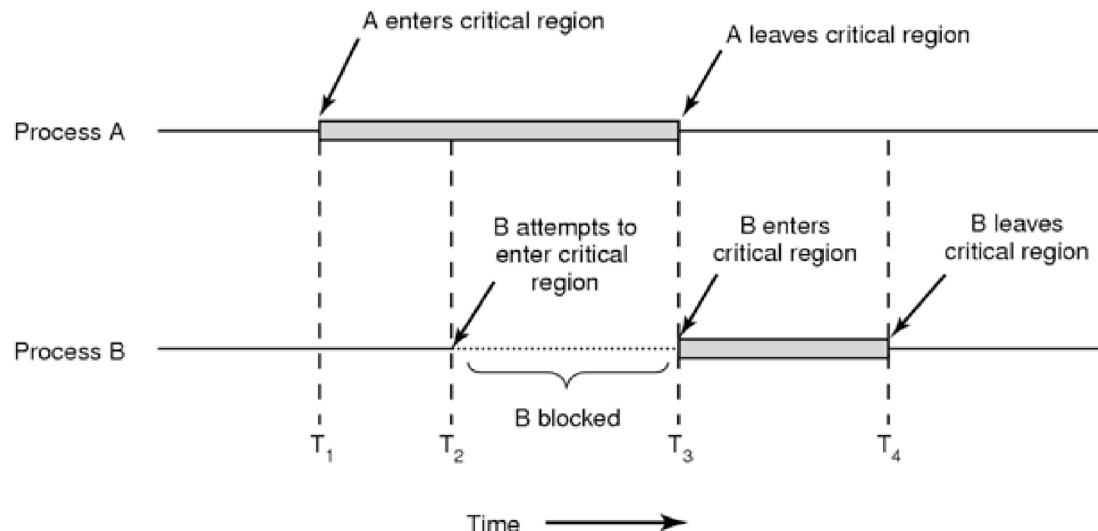


Gemeinsamer Zugriff auf den Wert „in“ in einer Druckerwarteschlange.

```
drucke( dateiname ) {
    i := lese_wert_aus_datei( in );
    i := i + 1;
    schreibe_wert_in_datei( in, i );
    schreibe_in_warteschlange( dateiname, i );
}
```

Benötigt: Wechselseitiger Ausschluss (mutual exclusiuon)

Kritischer Abschnitt: Der Teil des Programms, in denen auf gemeinsam benutzten Speicher zurückgegriffen wird.



Wechselseitiger Ausschluss mit aktivem Warten

Methode1: Unterbrechungen ausschalten

- Nach Betreten des kritischen Bereiches alle Unterbrechungen ausschalten
- Nach Verlassen, alle Unterbrechungen einschalten

Kann das System blockieren durch Prozesse, die Unterbrechungen nicht wieder einschalten

Methode 2: Variable Sperren

- Erhören/verringern einer Sperrvariable

Verlagert das Problem nur => keine Lösung

Methode 3: Die TSL-Anweisung

- Test and Set Lock
- Erfordert Unterstützung in Maschinensprache
- TSL lädt den Inhalt eines Speicherwortes in ein Register und schreibt an diese Adresse einen von Null verschiedenen Wert.
- Die Anweisung ist atomar, d.h. sie ist nicht Unterbrechbar. Der Speicherbus wird dabei gesperrt (wichtig bei mehreren CPUs)
- Verbraucht unnötig Rechenzeit („spinlock“)

```
enter_region:
    TSL REGISTER,LOCK            | copy lock to register and set lock to 1
    CMP REGISTER,#0              | was lock zero?
    JNE enter_region             | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0                 | store a 0 in lock
    RET | return to caller
```

Prioritätsumkehrproblem: B im kritischen Bereich hat niedrigere Prio als A, der auf B wartet. Beide kommen nie zum Zug.

Sleep and Wakeup

Zwei Methoden: sleep und wakeup

Das Erzeugs-Verbraucher-Problem: Zwei Prozesse sind über einen Puffer beschränkter Länge verbunden. Einer erzeugt Informationen und der andere entnimmt sie (siehe ls | grep).
Lösung: Wenn einer nicht arbeiten kann, wird er schlafen gelegt.

Situation: Puffer leer und Verbraucher testet, ob count == 0. Scheduler unterbricht. Erzeuger füllt Puffer und weckt Verbraucher. Danach wechselt der Scheduler zum Verbraucher und dieser legt sich schlafen.
(konkretes problem: if(count==0) und sleep() sind getrennte Operationen, zwischen denen ein Taskwechsel stattfinden kann(!))

Problem: Weckruf geht verloren.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Semaphoren

Variablenart Semaphore, welche die Anzahl der Weckrufe speichert.

Zwei Operationen auf s: up und down

```
down(s):
  if(s == 0) then
    sleep;
  s := s-1

up(s):
  s:= s +1;
  if( Warteliste für s ist nicht leer ) then
    wecke einen schlafenden Prozess
```

up und down sind atomar (!). Implementierung meist mittels TSL (ohne aktives Warten!).

Drei Semaphoren: Zum zählen der vollen/leeren Plätze sowie eine zum alleinigen Ausführen des kritischen Bereiches.

Ein Mutex ist ein Semaphor, welches nur 0 oder 1 sein kann.

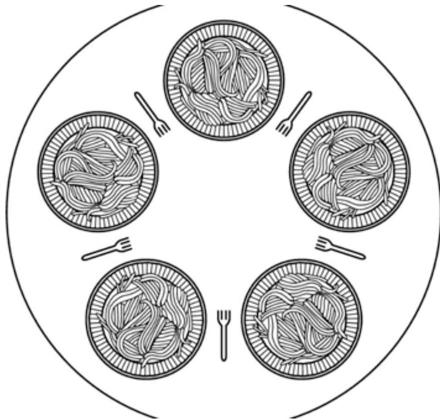
Semaphore meist im Kernel als Systemaufruf, Mutex im User-Modus.

Java: Monitor = Kapselung eines kritischen Bereiches.

```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                      /* TRUE is the constant 1 */
        item = produce_item( );         /* generate something to put in buffer */
        down(&empty);                   /* decrement empty count */
        down(&mutex);                   /* enter critical region */
        insert_item(item);              /* put new item in buffer */
        up(&mutex);                     /* leave critical region */
        up(&full);                      /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                      /* infinite loop */
        down(&full);                    /* decrement full count */
        down(&mutex);                   /* enter critical region */
        item = remove_item( );          /* take item from buffer */
        up(&mutex);                     /* leave critical region */
        up(&empty);                     /* increment count of empty slots */
        consume_item(item);             /* do something with the item */
    }
}
```

Problem der speisenden Philosophen

- Dient zur Veranschaulichung der Probleme bei der IPC
- Viele Lösungsalgorithmen



5 Philosophen die Denken oder Essen. Zum essen brauchen sie 2 Gabeln.

```
#define N 5                        /* number of philosophers */

void philosopher(int i)            /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                /* philosopher is thinking */
          take_fork(i);            /* take left fork */
          take_fork((i+1) % N);    /* take right fork; % is modulo operator */
          eat( );                  /* yum-yum, spaghetti */
          put_fork(i);             /* put left fork back on the table */
          put_fork((i+1) % N);     /* put right fork back on the table */
     }
}
```

Offensichtliche Lösung des Problems

- kann zu „deadlock“ führen, wenn alle gleichzeitig eine Gabel aufnehmen
- Modifikation: Philosoph legt gabel hin, wenn andere Gabel belegt. Kann zu „Aushungern“ (starvation) führen.
- Modifikation 2: Greifen/Hinlegen und dann zufällige Zeit warten (fuktioniert so z.B. bei Ethernet). Keine Funktionsgarantie.

Die Probleme tauchen auf, *obwohl* es keine kritischen Bereiche gibt!

```
#define N           5           /* number of philosophers */
#define LEFT        (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT       (i+1)%N     /* number of i's right neighbor */
#define THINKING    0           /* philosopher is thinking */
#define HUNGRY      1           /* philosopher is trying to get forks */
#define EATING      2           /* philosopher is eating */
typedef int semaphore;          /* semaphores are a special kind of int */
int state[N];                   /* array to keep track of everyone's state */
semaphore mutex = 1;            /* mutual exclusion for critical regions */
semaphore s[N];                 /* one semaphore per philosopher */

void philosopher(int i)         /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {              /* repeat forever */
        think( );               /* philosopher is thinking */
        take_forks(i);          /* acquire two forks or block */
        eat( );                 /* yum-yum, spaghetti */
        put_forks(i);           /* put both forks back on table */
    }
}

void take_forks(int i)          /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = HUNGRY;          /* record fact that philosopher i is hungry */
    test(i);                    /* try to acquire 2 forks */
    up(&mutex);                 /* exit critical region */
    down(&s[i]);                /* block if forks were not acquired */
}

void put_forks(i)               /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);               /* enter critical region */
    state[i] = THINKING;        /* philosopher has finished eating */
    test(LEFT);                 /* see if left neighbor can now eat */
    test(RIGHT);                /* see if right neighbor can now eat */
    up(&mutex);                 /* exit critical region */
}

void test(i)                    /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

Lösungsalgorithmus für das Philosophen-Problem.