Nebenläufigkeit & Synchronisation

1. Abhängigkeit zwischen Prozessen

Die Koordination der Kooperation und Konkurrenz zwischen Prozessen wird Synchronisation genannt. Eine Synchronisation schränkt das Wettrennen unter den Prozessen ein, um dadurch entstehende Fehler zu vermeiden.

➩ Im Buchungssystem

➩ Beim Eisenbahnverkehr

1.1 Einseitige Synchronisation (*unilateral*)

✎ Sind zwei Aktivitäten A1 und A2 kausal voneinander abhängig, in dem Sinn, daß A1 die Voraussetzung für das richtige Ergebnis von A2 bildet, so muß A1 vor A2 ausgeführt werden. Die Notwendigkeit einer bestimmten Abfolge zweier Aktivitäten wird als einseitige Synchronisation (*unilateral*) oder Bedingungssynchronisation (*condition synchronization*), oder logische Synchronisation genannt.

$A1 \rightarrow A2$

$A1 \rightarrow A2 \wedge A2 \rightarrow A3 \Rightarrow A1 \rightarrow A3$ (transitiv)

✎ **Ereignisvariablen**

Die einseitige Synchronisation kann durch Ereignisvariablen (*event variables*) geregelt werden, für die zwei Operationen signal und wait definiert sind.

- Durch signal wird das Eintreten eines Ereignisses oder einer Bedingung angezeigt.
- Durch wait wird auf ein durch signal anzuzeigendes Ereignis gewartet.

➩ Java

```
public class BlockLock {
   private StringBuffer sb = new StringBuffer();
   public void getLock() {
      doSomething(sb);
      synchronized (sb) {
         try {
            sb.wait();
         } catch (Exception e) { }
      }
   }
   public void freeLock() {
      doSomethingElse(sb);
      synchronized (sb) {
         sb.notify();
      }
   }
```


1.2 Mehrseitige Synchronisation (*multilateral*)

✎ Sind zwei Aktivitäten A1 und A2 kausal voneinander abhängig, in dem Sinn, daß eine der beiden Aktivitäten verzögert werden muß, bis die andere zum Abschluß gekommen ist. Diese Arte von Synchronisation wird als mehrseitige Synchronisation (*multilateral*).

$A1 \leftrightarrow A2$

$A1 \leftrightarrow A2 \Leftrightarrow A2 \leftrightarrow A1$ (symmetrisch)

✎ Die Aktivitäten, die von einer mehrseitigen Synchronisation betroffen sind, stehen in einem gegenseitigen Ausschluß (*mutual exclusion*) miteinander.

✎ Anweisungen, deren Ausführung einen gegenseitigen Ausschluß erfordert, heißen kritische Abschnitte (*critical sections, critical regions*).

➩ kritische Abschnitte

```
k: critical(n);
-- n ist die Zahl der Prozesse, die maxmal nebenläufig
-- im kritischen Abschnitt arbeiten dürfen. k ist ein
-- kritischer Abschnitt der Ordnung n.
```

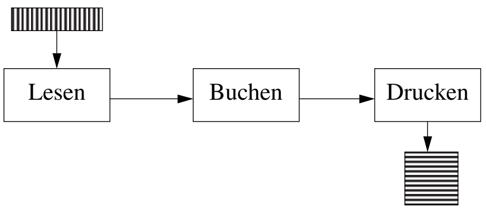
```
P1:                 P2:
...                 ...
crit k do           crit k do
  ...                 ...
end crit;           end crit;
...
```


✎ Sei Si die Menge der von der Aktivität Ai geschriebenen Variablen (Schreibmenge, *write set*), Li die der gelesenen Variablen (Lesemenge, *read set*). Dann darf A1 genau dann nicht zusammen mit A2 durchgeführt werden, wenn die folgende Bedingung erfüllt ist:

$$S1 \cap L2 \neq \{\} \vee L1 \cap S2 \neq \{\} \vee S1 \cap S2 \neq \{\}$$

➩ $x = x + 1 \leftrightarrow x = x + 1$
$v = x \leftrightarrow x = y$
$v = x \not\leftrightarrow w = x$ (konfliktfrei)

➩



eingang_puffer, ausgang_puffer

$S_{Lesen} = \{eingang\_puffer\}$ $L_{Lesen} = \{\}$

$S_{Buchen} = \{ausgang\_puffer\}$ $L_{Buchen} = \{eingang\_puffer\}$

$S_{Drucken} = \{\}$ $L_{Lesen} = \{ausgang\_puffer\}$

Lesen $\leftrightarrow$ Buchen und Buchen $\leftrightarrow$ Drucken

Lesen $\not\leftrightarrow$ Drucken

2 Synchronisation über gemeinsame Speicher

2.1 Schloßvariablen und -algorithmen

- Ein Schloßvariable (*lock variable*) regelt den Zugang zu einer Klasse kritischer Abschnitte, die von Prozessen aufgesperrt und verriegelt werden kann.
- Implementierungen der Zugriffsprozeduren von Schloßvariablen bezeichnen wir als Schloßalgorithmen (*lock algorithms*).

s : eine Schloßvariable

```
P:
  ...
  s.lock
    ...                -- kritischer Abschnitt
  s.unlock
  ...
```

✎ **Schloßalgorithmen mit unteilbaren Operationen**

➯ Unteilbare Maschineninstruktion: test_and_set(a,b)
- liest den Wert der Booleschen Variablen a
- kopiert a nach b
- setzt a auf true


```
bool besetzt = false;

void lock() {
  bool b;
  do test_and_set(besetzt,b) while (b);
}

void unlock() {
  besetzt = false;
```

✎ **Schloßalgorithmen für zwei Prozesse**

```
bool krit1 = false, krit2 = false;
int favorit = 1;

void lock1() {                    void lock2() {
  krit1 = true;                     krit2 = true;
  while (krit2) {                   while (krit1) {
    if (favorit!=1) {                 if (favoirt!=2); {
      krit1 = false;                    krit2 = false;
      while(favorit!=1);                while(favorit!=2);
      krit1 = true;                     krit2 = true;
    }                                 }
  }                                 }
}                                 }

void unlock1() {                  void unlock2() {
  favorit = 2;                      favorit = 1;
  krit1 = false;                    krit2 = false;
}                                 }
```


- Gegenseitiger Ausschluß:
  - “P1 betritt den kritischen Abschnitt” impliziert “P2 ist nicht im kritischen Abschnitt”.
  - “P1 ist im den kritischen Abschnitt” impliziert “P2 kommt nicht in den kritischen Abschnitt”.
- Verklemmungsfreiheit:
  - P1 und P2 bleiben nicht in ihrer äußeren Schleife hängen.
  - P1 und P2 bleiben nicht in ihrer inneren Schleife hängen.
  - P1 bleibt in der äußeren Schleife und P2 in der inneren Schleife hängen, oder umgekehrt.
- Aushungerungsfreiheit:
  - Ein Prozeß muß nicht warten, wenn der andere Prozeß den kritischen abschnitt nicht betreten will.
  - Wollen zwei Prozesse gleichzeitig den kritischen Abschnitt betreten, so gelangt mindestens einer umgehend hinein.
  - Wollen zwei Prozesse gleichzeitig ihren kritischen Abschnitt betreten, wird der bevorzugt, der nicht als letzter im kritischen Abschnitt war.

2.2 Synchronisation mit Semaphoren

✎ **Semaphoren**

- Ein Semaphor ist ein abstrakter Datentyp, dessen Objekte aus einem Zähler und einer Warteschlange bestehen. Er definiert zwei Operationen P für passieren und V für freigeben.
- Mit P-Operation wird der Wert des Semaphorzählers um eines erniedrigt. Erreicht er dabei einen negativen Wert, wird der die Operationsausführende Prozeß in die Warteschlange des Semaphors eingeordnet und blockiert.
- Mit V wird der Wert des Semaphorzähler um eins erhöht. Sollte der Zählerwert dann 0 oder negativ sein, wird der erste Prozeß aus der Warteschlange des Semaphors herausgeholt und in die Warteschlange der bereiten Prozesse eingereiht.

➯ Einseitige Synchronisation mit Semaphoren

```
semaphore s;
s.init(0);
void P1() {                       void P2() {
  . . . s.V(); . . . }              . . . s.P(); . . . }
```

➯ Gegenseitige Synchronisation mit Semaphoren

```
Sema s;
s.init(1);
void P1() {                       void P2() {
  . . .                             . . .
  s.P(); . . . ; s.V();             s.P(); . . .; s.V();
  . . . }                           . . . }
```

2.3 Synchronisation mit Monitoren

✎ Ein Monitor ist eine abstrakte Datenstruktur mit impliziten Synchronisationseigenschaften: Den Benutzern eines Monitors bleibt nicht nur die Implementierung der Zugriffsfunktionen auf gemeinsame Daten, sondern auch die Realisierung des gegenseitigen Ausschlusses für diese Funktionen verborgen.

➪

```
monitor Count {
int c = 0;
void increase()        void decrease()        int value()
  { c = c+1; }            { c = c - 1; }          { return c; }
}
```

```
P1:  . . . cnt.increase(); . . .
P2:  . . . cnt.increase(); . . .
P3:  . . . cnt.decrease(); . . .
```

➪ Java

```
public class Count {
   int c = 0;
   public synchronized void increase() {
      c = c+1;
   }
   public synchronized void decrease() {
      c = c-1;
   }
   public synchronized int value() {
      return(c);
   }
}
```