Technische Universität München
Fakultät für Informatik
Dr. M. Schneider, G. Bauer,
Dr. M. Pizka, Dr. B. Rumpe,
Dr. R. Steinbrüggen

WS 2002/2003
Lösungsvorschläge zu Blatt 7
20. Dezember 2002

Übungen zur Vorlesung Einführung in die Informatik III

Aufgabe 25 Monitor

(a) Die Definition des Monitors schließt die gleichzeitige Ausführung der Prozeduren „increment“, „decrement“ und „get“ aus. Auch eine mehrmalige gleichzeitige Ausführung jeder dieser Prozeduren ist ausgeschlossen. Man beachte, dass laut Monitordefinition während des Wartens im „wait“ eine andere Prozedur des Monitors aufgerufen werden kann.

(b) In der Lösung von Aufgabe 20 wird z.B. das Boolesche Semaphor `sx` zur Benachrichtigung eingesetzt. Ein Beispiel für die Verwendung zum gegenseitigen Ausschluss wäre:

```
⌈  sema bool sn:= true;
   var int n := 0;
   ⟦⌈  P(sn); n := n+1; V(sn);
       ‖
       P(sn); n := n-1; V(sn); ⌋⟧
⌋
```

Beim Monitor kann der gegenseitige Ausschluss durch verschiedene Prozeduren erreicht werden. Die Benachrichtigung kann mit Signalen umgesetzt werden.

(c)
```
⌈  monitor X = /* Counter */
   ⌈  signal si, sd, sp;
      var bool i, d, p = false, false, false
      var int n = 0;
      proc next_inc = ():          ⌈ i := true; si.signal ⌋
      proc next_dec = ():          ⌈ d := true; sd.signal ⌋
      proc increment = ():         ⌈ if ¬ i then si.wait fi;
                                     i := false; n := n+1; p := true; sp.signal ⌋
      proc decrement = ():         ⌈ if ¬ d then sd.wait fi;
                                     d := false; n := n-1; p := true; sp.signal ⌋
      proc wait_produced = ():     ⌈ if ¬ p then sp.wait fi;
                                     p := false ⌋
      proc get = (var int v):      ⌈ v := n ⌋
   ⌋

   monitor Y = ⌈ /* identisch mit X */ ⌋
```

```
    var int x, y, z := 0, 0, 0;
    X.next_inc(); Y.next_dec(); /* start */

    [[
    while true do X.increment() od
    ||
    while true do Y.decrement() od
    ||
    while true do X.wait_produced(); Y.wait_produced();
                  X.get(x); Y.get(y); z := x+y;
                  X.next_inc(); Y.next_dec() od
    ]]
]
```

Aufgabe 26 Monitore in Java

a) Für die Realisierung der Signal „wait“ Operation werden hier `while`-Schleifen und die Java Signalisierungs-Methode `notifyAll()` eingesetzt. `notifyAll()` ist notwendig, da mehrere Threads in unterschiedlichen `wait()`-Aufrufen auf verschiedene Ereignisse warten können, z.B. auf `inc` und auf `produced`. Ein einfaches `notify()` könnte in dieser Situation den falschen Thread aktivieren. Mit einer `while`-Schleife kann dies verhindert werden: Die wartenden Threads stellen selbst fest, ob „ihr“ Ereignis eingetreten ist oder nicht und blockieren sich ggf. erneut. `notifyAll()` gewährleistet, dass die Nachricht den gewünschten Thread erreicht.

```
public class Counter {
    private int n=0;
    private boolean si,sd,sp;
    private boolean i = false, d = false, p = false;

    public synchronized void increment() {
            awaitNextInc(); n++; produced();
    }

    public synchronized void decrement() {
            awaitNextDec(); n–; produced();
    }

    public synchronized int get() {
            return n;
    }

    private void produced() {
            p = true; sp = true; notifyAll();
    }

    private void awaitNextInc() {
            if(!i) {
```

```
            si=false;
            while(!si) {
                try {wait();} catch(InterruptedException e) { }}
            }
        i = false;
    }

    private void awaitNextDec() {
        if(!d) {
            sd=false;
            while(!sd) {
                try {wait();} catch(InterruptedException e) { }}
            }
        d = false;
    }

    public synchronized void waitProduced() {
        if(!p) {
            sp=false;
            while(!sp) {
                try {wait();} catch(InterruptedException e) { }}
            }
        p = false;
    }

    public synchronized void nextInc() {
        i = true; si = true; notifyAll();
    }

    public synchronized void nextDec() {
        d = true; sd = true; notifyAll();
    }
}
```

b)

```
public static void main( String[] arguments )     {
    final Counter x = new Counter();
    final Counter y = new Counter();

    new Thread() {
          public void run() { while(true) x.increment(); }
          }.start();
    new Thread() {
          public void run() { while(true) y.decrement(); }
          }.start();
    new Thread() {
          public void run() {
              int z=0;
              for( int i = 0; i<50 ; i++ )  {
                  x.waitProduced(); y.waitProduced();
                  z = x.get() + y.get();
                  System.out.println(z+"="+x.get()+"+"+y.get());
```

```
                    x.nextInc(); y.nextDec();
                }
            }}.start();
      x.nextInc();y.nextDec();
    }

   Das Ablaufprotokoll:
   0=1+-1
   0=2+-2
   0=3+-3
   0=4+-4
   ...
   0=49+-49
   0=50+-50
```

(P) `synchronized` kann und muß bei einigen Methoden wie `nextInc()` und `nextDec()` entfallen. Dies ist notwendig, damit eine Verschachtelung von Monitoren verhindert wird, da dies zu einer Verklemmung führen würde. Der Vorteil dieser Lösung liegt im geringeren Synchronisationsaufwand während der Berechnung, da weniger Threads unnötigerweise aktiviert werden.

```
public class Counter  {
      private int n=0;
      private Object si = new Object(); private boolean i=false;
      private Object sd = new Object(); private boolean d=false;
      private Object sp = new Object(); private boolean p=false;

      public void increment()  {
         awaitNextInc(); synchronized(this){n++;} produced();
      }

      public void decrement()  {
         awaitNextDec(); synchronized(this){n--;} produced();
      }

      public synchronized int get()  {
          return n;
      }

      private void produced() {
        synchronized(sp)  {
           p=true;sp.notifyAll()
        }
      }

      private void awaitNextInc() {
        synchronized(si)  {
           if(!i){try{ si.wait();}catch(InterruptedException e){}}
           i = false;
        }
      }
```

```
    private void awaitNextDec() {
      synchronized(sd) {
         if(!d){try{ sd.wait();}catch(InterruptedException e){}}
         d = false;
      }
    }

    public void waitProduced()  {
       synchronized(sp) {
          if(!p){try{sp.wait();}catch(InterruptedException e){}}
          p = false;
       }
    }

    public void nextInc()   {
       synchronized(si)  {
          i = true;si.notifyAll();
       }
    }

    public void nextDec()   {
       synchronized(sd)  {
          d = true;sd.notifyAll();
       }
    }

    public static void main( String[] arguments )
    { /* wie in b) */ }
}
```

Aufgabe 27 Leser-Schreiber-Problem

(a) – mit Hilfe von **await** Wir verwenden eine Boolesche Variable `ws`, die angibt ob gerade ein Schreiber schreibt, und eine natürlichzahlige Variable `rs`, die die Anzahl der Leser angibt.

Gegenseitiger Ausschluss von Lesern und Schreibern wird dadurch erreicht, dass die Schreiber warten müssen, bis die Anzahl der Leser Null ist und umgekehrt.

Gegenseitiger Ausschluss der Schreiber wird dadurch erreicht, dass sobald ein Schreiber seinen kritischen Bereich betritt, die Eintrittsbedingung für andere Schreiber falsch wird.

```
var bool ws : = false;
var nat rs : = 0;

proc Writer =
  ⌈ var m x
    while true do
      produce(x);
```

```
      await ¬ ws ∧ rs = 0 then ws := true endwait ;
      write(x);
      await true then ws := false endwait
    od ⌋

proc Reader =
  ⌈ var m y
    while true do
      await ¬ ws then rs := rs + 1 endwait ;
      read(y);
      await true then rs := rs - 1 endwait
      consume(y);
    od ⌋
```

– mit Hilfe von Semaphoren

  Wir verwenden eine Boolesche Semaphore `ws`, die angibt ob gerade ein Schreiber schreibt, und eine natürlichzahlige Variable `rs`, die die Anzahl der Leser angibt.Wir verwenden ausserdem eine Boolesche Semaphore `mutex`, für den exklusiven Zugriff auf die Variable `rs`.

  Jeder Schreiber belegt zuerst die Semaphore `ws` und ein Leser belgt `ws` genau dann, wenn er der erste Leser war. Ein Schreiber gibt `ws` frei, nahcdem er geschrieben hat, Ebenso gibt der letzte Leser `ws` wieder frei. Dadurch wird der gegenseitige Ausschluss von Lesern und Schreibern und der gegenseitige Ausschluss von zwei Schreibern erreicht, aber mehrere Leser zugleich zugelassen.

```
sema bool mutex, ws : = true, true;
var int rs : = 0;

proc Writer =
  ⌈ var m x
    while true do
      produce(x);
      P(ws); write(x); V(ws)
    od ⌋

proc Reader =
  ⌈ var m y
    while true do
      P(mutex); rs := rs + 1; if rs = 1 then P(ws) fi ; V(mutex);
      read(y);
      P(mutex); rs := rs - 1; if rs = 0 then V(ws) fi ; V(mutex);
      consume(y);
    od ⌋
```

– mit Hilfe von Monitoren

  Wir verwenden eine Boolesche Variable `ws`, die angibt ob gerade ein Schreiber schreibt, und eine natürlichzahlige Variable `rs`, die die Anzahl der Leser angibt.

  Es werden Monitor-Prozeduren `start_read`, `end_read`, `start_write`, und `end_write` eingeführt. `start_read` und `start_write` werden von Lesern bzw. Schreibern aufge-

rufen, die zu lesen bzw. schreiben wünschen. `end_read` und `end_write` werden von Lesern bzw. Schreibern aufgerufen, die mit Lesen bzw. Schreiben fertig sind. Durch das Monitorkonzept wird gewährleistet, dass diese Prozeduren nur unter gegenseitigem Auschluss ausgeführt werden. Das Signal `rw_enable` zeigt an, dass gerade niemand liest oder schreibt. Falls es einen Schreiber gibt, legt sich ein Leser Prozess solange schlafen, bis der Schreiber fertig mit Schreiben ist und er das Signal `rw_enable` erhält. Dann kann kein Schreiber mehr zu schreiben beginnen, solange bis der letzte Leser durch das Signal `rw_enable` angibt, dass wieder gelesen/geschrieben werden kann.

```
monitor ReadersWriter =
  ⌈ var int rs := 0;
    var bool ws := false;
    signal rw_enable;

    proc start_read =
      ⌈ if ws then rw_enable.wait fi ;
        rs := rs + 1;  ⌋

    proc end_read =
      ⌈ rs := rs - 1;
        if rs = 0 then rw_enable.signal fi ;  ⌋

    proc start_write =
      ⌈ if ws ∨ 0 < rs then rw_enable.wait fi ;
        ws := true;  ⌋

    proc end_write =
      ⌈ ws := false;
        rw_enable.signal;  ⌋

proc Writer =
  ⌈ var m x;
    produce(x);
    start_write();
    write(x);
    end_write();  ⌋

proc Reader =
  ⌈ var m y;
    start_read();
    read(y);
    end_read();
    consume(y);  ⌋
```

(b) Ein Problem bei diesen Lösungen ist, dass es zu unfairen Abläufen kommen kann. Es ist z.B. möglich, dass es zu jedem Zeitpunkt mindestens einen Leser gibt. Dann kommen wartende Schreiber nie an die Reihe, d.h. es kommt zu einer Aushungerung der Schreiber.

Eine Lösung, um zu verhindern, dass einer der beiden Typen von Prozessen das Betriebsmit-

tel vollständig für sich beansprucht, wurde von Hoare (1974) vorgeschlagen: Einem neuen Leser wird es nicht gestattet, mit dem Lesen zu beginnen, falls es einen wartenden Schreiber gibt; Alle Leser, die auf die Beendigung der Schreiboperation warten, sollen vor dem nächsten Leser Vorrang haben.

Das lässt sich für die Semaphor-Lösung mit der Einführung von Warteschlangen realisieren.

Für die Monitore wurde dazu von Hoare eine Erweiterung um an Bedingungen geknüpfte Warteschlagen vorgeschlagen. Dieses Prinzip ist im folgenden Monitor umgesetzt:

```
monitor ReadersWriter =
  ⌈ var int rs := 0;
    var bool ws := false;
    signal r_enable, w_enable;

    proc start_read =
      ⌈ if ws ∨ w_enable.queue then r_enable.wait fi ;
        rs := rs + 1;
        r_enable.signal;  ⌋

    proc end_read =
      ⌈ rs := rs - 1;
        if rs = 0 then w_enable.signal fi ;  ⌋

    proc start_write =
      ⌈ if ws ∨ 0 < rs then w_enable.wait fi ;
        ws := true;  ⌋

    proc end_write =
      ⌈ ws := false;
        if r_enable.queue then r_enable.signal else w_enable.signal fi ;  ⌋

proc Writer =
  ⌈ var m x;
    produce(x);
    start_write();
    write(x);
    end_write();  ⌋

proc Reader =
  ⌈ var m y;
    start_read();
    read(y);
    end_read();
    consume(y);  ⌋
```