Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
SS 05

AG Betriebssysteme FB3
Kirsten Berkenkötter, Jan Peleska

Ausgabe: 9.5.2005
Abgabe: 23.5.2005

Übungszettel 2

Hinweise

Die Abgabe erfolgt als Ausdruck am Ende der Vorlesung und als E-Mail an *kirsten@tzi.de*. **Auf jeden Fall** sollten alle CSP-Dateien auch in elektronischer Form (als E-Mail-Attachment) abgegeben werden. Verwendet für Aufgabe 1 und 2 jeweils eigene Dateien. Zur vollständigen Lösung der Aufgabe gehören Spezifikation, Verifikation und Dokumentation (in Latex). Der Betreff der E-Mail sollte folgendes Aussehen haben:
BS1 Abgabe x Gruppe y.

Bitte immer die Namen aller Gruppenmitglieder und die Gruppennummer angeben!

Aufgabe 1: Petersons Algorithmus – zwei Prozesse

Petersons Algorithmus zur Realisierung von *mutual exclusion* ist im Folgenden in ANSI C angegeben; er besteht aus zwei Prozeduren, die von den Prozessen aufgerufen werden, die auf die kritische Region zugreifen wollen.

```
#define N     2                                 //number of processes

int turn;                                       //whose turn is it?
int interested[N];                              //process with pid in {0..N-1}
                                                //interested (== 1) or not (== 0)

void enter_region(int process)
{
  int other = 1 - process;                      //pid of the other process

  interested[process] = 1;                      //process is interested
  turn = process;                               //set turn flag to pid

  while(interested[other] && turn == process)  //wait, until critical region is free
    ;

  //now do some nice critical stuff
}

void leave_region(int process)
{
  interested[process] = 0;                      //process is not interested any more
}
```

a) Gebt (in FDR-Syntax) eine CSP-Spezifikation dieses Algorithmus an, indem ihr zwei Prozesse spezifiziert, die möglichst direkte Umsetzungen der beiden Prozeduren *enter_region* und *leave_region* sind.

b) Weist mit Hilfe von FDR nach, dass der Algorithmus auch tatsächlich *mutual exclusion* garantiert. Verwendet dazu einen Watchdog.

Aufgabe 2: Petersons Algorithmus – N Prozesse

a) Die Verallgemeinerung von Petersons Algorithmus für zwei Prozesse (s. Aufgabe 1) mit einem Array *interested[N]* für N=3 ist keine Lösung für *mutual exclusion*. Begründet diese Aussage und illustriert sie an einem Beispiel.

b) Die Verallgemeinerung von Petersons Algorithmus für N Prozesse sieht in ANSI C folgendermaßen aus:

```
void enter_region(int process)
{
  int i, j;
  for(i = 0; i < N; i++)
  {
    interested[process] = i+1;
    turn[i] = process;

    for(j = 0; j < N; j++)
    {
      if(j != process)
      {
        while(interested[j] >= interested[process] && turn[i] == process)
          ;
      }
    }
  }
}

void leave_region(int process)
{
  interested[process] = 0;
}
```

Erläutert, warum und wie dieser Prozess *mutual exclusion* für beliebig viele Prozesse garantiert.

c) Erweitert eure CSP-Spezifikation aus Aufgabe 1) entsprechend des verallgemeinerten Algorithmus. Zeigt für N=3, dass der Algorithmus das gewünschte Verhalten garantiert.

Aufgabe 3: Non-Blocking Write Protocol

Das Non-Blocking Write Protocol (NBW) ist für den Einsatz in Echtzeitsystemen gedacht, um zeitintensive Operationen wie z.B. bei Semaphoren durch die Verwaltung einer Warteschlange zu vermeiden. Der schreibende Prozess wird hier nie blockiert; der lesende Prozess überprüft, ob konsistente Daten vorliegen. Dazu wird das *Concurrency Control Field* (CCF) verwendet, eine einfache Integer-Variable, die vom schreibenden Prozess vor und nach dem Schreiben inkrementiert wird. Hat das CCF einen ungeraden Wert, ist gerade ein Scheibvorgang aktiv, und der lesende Prozess bricht sofort ab. Hat das CCF vor und nach Beginn eines Lesevorgangs unterschiedliche Werte, sind die gelesenen Daten inkonsistent, da in der Zwischenzeit geschrieben wurde.

Dieser Algorithmus ist nur effektiv, wenn die Zeit zwischen zwei Schreiboperationen deutlich größer ist als die Zeit für einen Schreib-oder Lesevorgang. Dies ist in der Regel bei Echtzeitsystemen der Fall.

Im Folgenden ist der Code in ANSI C angegeben:

```
int CCF = 0;                                    //concurrency control field

void write()
{
  int CCF_old = CCF;                            //get old value of CCF

  CCF = CCF_old+1;                              //increment CCF before write
  //write data
  CCF = CCF_old+2;                              //increment CCF after write
}

void read()
{
  int CCF_begin, CCF_end;                       //get CCF before and after reading

  start: CCF_begin = CCF;                       //value of CCF before reading
         if(CCF_begin % 2)                      //value of CCF is odd
           goto start;                          //writer is active, try again
         //read data
         CCF_end = CCF;                         //value of CCF after reading
         if(CCF_begin != CCF_end)               //check if writer war active while reading
   goto start;
}
```

a) Gebt (in FDR-Syntax) eine CSP-Spezifikation dieses Algorithmus an, in der die beiden Prozeduren *write* und *read* möglichst genau umgesetzt sind. Nehmt an, dass CCF maximal den Wert 8 erreichen kann (den modulo-Operator verwenden).

b) Weist mit Hilfe von FDR nach, dass der Algorithmus tatsächlich funktioniert. Modelliert dazu einen Watchdog-Prozess, der das korrekte Verhalten von lesendem und schreibendem Prozess überprüft, sowie einen Speed-Prozess, der die lesenden und schreibenden Prozesse koordiniert.