Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
SS 04

AG Betriebssysteme FB3
Kirsten Berkenkötter, Jan Peleska

Ausgabe: 3.5.2004
Abgabe: 17.5.2004

Übungszettel 1

Hinweise

Die Abgabe erfolgt als E-Mail an *trie@tzi.de*. **Auf jeden Fall** sollten die CSP-Spezifikationen auch in elektronischer Form (als E-Mail-Attachment) abgegeben werden.

Bitte immer die Namen aller Gruppenmitglieder und die Gruppennummer angeben!

Aufgabe 1: Petersons Algorithmus

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 FALSE   0
#define TRUE    1
#define N       2                 /* number of processes                    */

int turn;                         /* whose turn is it?                      */
int interested[N];                /* all values initially 0 (FALSE)         */

void enter_region (int process)   /* process: who is entering (0 or 1)      */
{
   int other;                     /* number of other process                */
   other = 1 - process;           /* the opposite of process                */
   interested[process] = TRUE;    /* show that you are interested           */
   turn = process;                /* set flag                               */
   while (turn == process && interested[other] == TRUE); /* null statement   */

void leave_region (int process)   /* process: who is leaving (0 or 1)       */
{
   interested[process] = FALSE;   /* indicate departure from critical region */
}
```

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. Definiert dazu eine geeignete Abstraktion des erwarteten Verhaltens. Dokumentiert das Ergebnis geeignet.

c) Die Verallgemeinerung von Petersons Algorithmus mit einem Array *interested[N]* für N=3 ist keine Lösung für *mutual exclusion*. Begründet diese Aussage.

Aufgabe 2: 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 ein *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;                /* das CCF ist mit 0 initialisiert                */

void write()                /* der schreibende Prozess wird nie blockiert     */
{
   int CCF_old = CCF;       /* alten Wert des CCF merken                      */
   CCF = CCF_old + 1;       /* CCF vor dem Schreiben inkrementieren           */
   //Write data structure
   CCF = CCF_old + 2;       /* CCF nach dem Schreiben nochmal inkrementieren  */
}

void read()                 /* der schreibende Prozess prüft Konsistenz       */
{
   int CCF_begin, CCF_end;
   start: CCF_begin = CCF;  /* Wert des CCF vor dem Lesen merken              */
          if((CCF_begin % 2) != 0) /* Ungerades CCF -> Schreibvorgang laeuft  */
             goto start;    /* Lesen sofort abbrechen                         */
          //Read data structure
          CCF_end = CCF;    /* Wert des CCF nach dem Lesen merken             */
          if (CCF_end != CCF_begin) /* Überprüfung auf Inkonsistenz           */
             goto start;    /* Inkonsistente Daten, nochmal lesen             */
}
```

a) Gebt (in FDR-Syntax) eine CSP-Spezifikation dieses Algorithmus an, in der die beiden Prozeduren *write* und *read* möglichst genau umgesetzt sind. Nehmetan, 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.