Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
SS 05

AG Betriebssysteme FB3
Kirsten Berkenkötter, Jan Peleska

Ausgabe: 23.5.2005
Abgabe: 13.6.2005

Übungszettel 3

Hinweise

Die Abgabe erfolgt als Ausdruck am Ende der Vorlesung und als E-Mail an *kirsten@tzi.de*. **Auf jeden Fall** sollten alle Quelldateien auch in elektronischer Form (als E-Mail-Attachment) abgegeben werden. 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: Monitore mit Semaphoren – das Konzept

a) Beschreibt, wie ein Monitor mit Hilfe von Semaphoren umgesetzt werden kann. Zeigt den Ablauf für die folgenden Funktionen eines Monitors:

- Ausführen einer Funktion des Monitors
- *WAIT(condition)*
- *NOTIFY(condition)*
- *NOTIFYALL(condition)*

Hierbei gilt, dass *NOTIFY* einen (beliebigen) wartenden Prozess freigibt und *NOTIFYALL* alle wartenden Prozesse, wenn die assoziierte *condition* wahr wird.

b) Gebt (in FDR–Syntax) eine CSP–Spezifikation dieses Algorithmus für das Producer/Consumer–Problem mit einem Producer und einem Consumer an. Auf die Funktion *NOTIFYALL* kann in diesem Fall natürlich verzichtet werden, die anderen drei Funktionen sollen als CSP–Prozesse umgesetzt werden. Als Grundlage dient die CSP–Spezifikation des Producer/Consumer–Problems mit Semaphoren aus der Vorlesung. Im Monitor sollen sich die Prozesse *INSERT* und *REMOVE* befinden, die zum Einfügen, bzw. Löschen eines Eintrags in einen Puffer dienen.

c) Weist mit Hilfe von FDR nach, dass

- das System deadlock– und livelock–frei ist.
- sich immer nur ein Prozess im Monitor befindet.

Aufgabe 2: Monitore mit Semaphoren – Realisierung in C

Um einen Monitor in C zu realisieren, wird zunächst eine Bibliothek angelegt, welche die dazu benötigten Funktionen realisiert. Dann wird ein konkreter Monitor für das Producer/Consumer–Problem angelegt, der die entsprechenden Daten und Funktionen kapselt. Dieser konkrete Monitor kann dann verwendet werden.

Gegeben sind:

- *producer.c* – Der Prozess für den Producer.
- *consumer.c* – Der Prozess für den Consumer.
- *Makefile* – Makefile für alle Prozesse
- *runProdConX* – Testläufe, die hinterher funktionieren sollen!

Diese Dateien sollen nicht mehr verändert werden (mit Ausnahme des Makefile, wo *#define VERBOSE* gesetzt werden kann).

a) Bibliothek für Monitorfunktionen: *monitorlib.c/monitorlib.h*

Generell gilt: wenn *#define VERBOSE* gilt, werden Statusinformationen des Monitors ausgegeben (Anfragen und Freigeben von Semaphoren, Aufruf von Funktionen, etc.), um den korrekten Ablauf verfolgen zu können. Wenn Funktionen fehlschlagen (z.B. *semget()*), wird immer mit *perror()* eine Meldung ausgegeben und dem aufrufenden Prozess das Signal *SIGTERM* geschickt.

Monitor erzeugen

```
int createMonitor()
```

Legt alle Verwaltungsstrukturen für den Monitor in einem Shared Memory an und liefert dessen ID zurück.

Funktionen im Monitor registrieren

```
void registerMonitorFunction(monitor_t *mon,
                             void* (*func)(void *args),
                             int pid)
```

Registriert eine Funktion im Monitor. Übergeben wird der Zeiger auf das Shared Memory, der Zeiger auf die Funktion und die PID des aufrufenden Prozesses. Die ID wird benötigt, da jeder Prozess seinen eigenen Speicherbereich hat und der Zeiger auf die Funktion nur innerhalb dieses Speicherbereichs gültig ist.

Monitor freigeben

```
void freeMonitor(monitor_t *mon)
```

Gibt die belegten Ressourcen frei.

Funktion des Monitors ausführen

```
void* requestMonitorFunction(monitor_t *mon,
                             void* (*func)(void *args),
                             void *args,
                             int pid)
```

Führt eine Funktion des Monitors aus. Übergeben wird der Zeiger auf das Shared Memory, der Zeiger auf die auszuführende Funktion, ein Zeiger auf die Argumente der Funktion, sowie die PID des aufrufenden Prozesses zur korrekten Identifikation der Funktion. Zurückgegeben wird ein Zeiger auf das Ergebnis der Funktion.

WAIT(condition)

```
void waitCondition(monitor_t *mon, int n)
```

Implementiert die *WAIT*–Funktion eines Monitors. Übergeben wird der Zeiger auf das Shared Memory, sowie die Nummer der *condition*.

NOTIFY(condition)

```
void notifyCondition(monitor_t *mon, int n)
```

Implementiert die *NOTIFY*–Funktion eines Monitors. Übergeben wird der Zeiger auf das Shared Memory, sowie die Nummer der *condition*.

NOTIFYALL(condition)

```
void notifyAllCondition(monitor_t *mon, int n)
```

Implementiert die *NOTIFYALL*–Funktion eines Monitors. Übergeben wird der Zeiger auf das Shared Memory, sowie die Nummer der *condition*.

b) Monitor für das Producer/Consumer–Problem: *prodcon.c/prodcon.h*

Wie auch bei den Monitorfunktionen gilt: Wenn Funktionen fehlschlagen (z.B. *semget()*), wird immer mit *perror()* eine Meldung ausgegeben und dem aufrufenden Prozess das Signal *SIGTERM* geschickt.

Öffnen des Monitors

```
void openMonitor()
```

Anlegen des konkreten Monitors für das Producer/Consumer–Problem (Registrieren der Funktionen, etc.).

Schließen des Monitors

```
void closeMonitor()
```

Schließen des konkreten Monitors (Freigeben aller Ressourcen).

Einfügen eines Elements in den Puffer

```
static void insertItemMonitor(int *item)
void insertItem(int item)
```

Die erste Funktion ist die Funktion, die dem Monitor gehört. Hier wird das Einfügen eines Eintrags in den Puffer erledigt. Wenn # *ONE* aktiviert ist, wird *NOTIFY* verwendet, sonst *NOTIFYALL*. Diese Funktion ist nicht von außen sichtbar.

Die zweite Funktion dient als Wrapper und wird von *producer.c* aufgerufen.

Löschen eines Elements aus dem Puffer

```
static int *removeItemMonitor()
int removeItem()
```

Die erste Funktion ist die Funktion, die dem Monitor gehört. Hier wird das Entfernen eines Eintrags aus dem Puffer erledigt. Wenn # *ONE* aktiviert ist, wird *NOTIFY* verwendet, sonst *NOTIFYALL*. Diese Funktion ist nicht von außen sichtbar.

Die zweite Funktion dient als Wrapper und wird von *consumer.c* aufgerufen.

c) *NOTIFY* vs. *NOTIFYALL*

Macht die Verwendung von *NOTIFY* und *NOTIFYALL* einen Unterschied? Wenn ja, begründet, warum. Wenn nein, gebt ein Beispiel für eine sinnvolle Anwendung der beiden Funktionen.