Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
SS 04

AG Betriebssysteme FB3
Kirsten Berkenkötter, Jan Peleska

Ausgabe: 5.7.2004
Abgabe: drei Tage vor dem Fachgespräch, spätestens am 19.7.2004

Übungszettel 4

Hinweise

Die Abgabe erfolgt als E-Mail an *trie@tzi.de*. **Auf jeden Fall** sollten alle C-Dateien auch in elektronischer Form (als E-Mail-Attachment) abgegeben werden. Zur vollständigen Lösung der Aufgabe gehören Programm, Test und Dokumentation.

Bitte immer die Namen aller Gruppenmitglieder und die Gruppennummer angeben!

Aufgabe 1: Nicht-präemptives priorisiertes User-Space Scheduling

Implementiert eine Scheduler-Bibliothek in C, welche folgende Funktionen zur Verfügung stellt:

Registrieren eines Threads

Ein neuer User-Thread mit Identifikation *threadId* wird beim Scheduler registriert, indem ein Funktionspointer auf die Thread-Funktion zur Initialisierung, die Thread-Main-Funktion zur zyklischen Verarbeitung und die Funktion zur Terminierung als Parameter mitgegeben werden.

prio ist eine statische Priorität: Der Scheduler aktiviert die Thread-Main-Funktion in jedem (prio+1)tem Scheduling-Zyklus. *0* ist also die höchste Priorität.

Der Scheduler registriert den Thread und ruft danach sofort die Initialisierungsfunktion auf. Bei Erfolg gibt die Funktion die *ID* des Thread zurück, ansonsten *-1*.

```
int registerThread(int threadId,
                   void (*myThreadInit)(int threadId),
                   void (*myThread)(int threadId, void *context),
                   void (*myThreadTerminate)(int threadId,
                                             void *context),
                   int prio);
```

Thread-Kontext

Bei zyklischer Aktivierung und Terminierung durch den Scheduler erhält jeder User-Thread nur seine Thread-Id und einen Pointer auf seine Zustandsdaten als Eingangsdaten.

Die weiteren Inputs besorgt er sich selber durch Lesen der entsprechenden Schnittstelle. Zustandsinformationen legt er sich in einer bei der Initialisierung dynamisch zu allokierenden Struktur an. Diese wird mit *registerContext()* beim Scheduler registriert, so dass dieser bei jeder

zyklischen Thread-Aktivierung und bei der Thread-Terminierung den Zeiger auf den Kontext als Parameter mitgeben kann.

Der Rückgabewert ist im Erfolgsfall *0*, sonst *-1*.

```
int registerContext(int threadId, void *context);
```

Thread-Aktivierung

Der folgende Aufruf führt dazu, dass der Scheduler alle User-Thread-Main-Funktionen(mit passender Id und passendem Context-Pointer als Parameter) aufruft, die laut *prio* im aktuellen Zyklus aufzurufen sind.

```
void schedActivateUserThreads(void);
```

Thread-Terminierung

Der folgende Aufruf bewirkt, dass der Scheduler alle Terminierungsfunktionen aufruft.

Bei Erfolg ist der Rückgabewert *0*, sonst-*1*.

```
int schedTerminateUserThread(void);
```

Anwendungsbeispiel

Die Verwendung im Anwenderprogramm ist folgendermaßen:

```
void t1Init(int threadId)
{
  //...
  registerContext(myContextPointer);
  //...
}

void t1(int threadId, void *c)
{
  //...
}

void t1Terminate(int threadId, void *c)
{
  //...
}

//weitere Thread-Funktionen

int main()
{
  int result;

  result = registerThread(1, t1Init, t1, t1Terminate, 0);
  //...
```

```
  result = registerThread(2, t2Init, t2, t2Terminate, 2);

  //...

  //Die terminateCondition sollte durch Signal-Handler gesetzt werden.
  while(! terminateCondition)
  {

    schedActivateUserThreads();
  }

  result = schedTerminateUserThreads();
  //...
}
```

Aufgabe 2: Ringpuffer mit Threads und Scheduling

In dieser Aufgabe sollen der Ringpuffer von Übungsblatt 2 und die Schedulingbibliothek in einem Anwendungsprogramm verwendet werden.

Schreibt Sie einen Prozess, der vier User-Threads *T1*, *T2*, *T3* und *T4* mit der Technik von Aufgabe 1 verwendet, die auf folgende Weise miteinander über die Ringpuffer von Übungsblatt 2 kommunizieren (sogenannte Pipeline-Verarbeitung):

T1 → T2 → T3 → T4

- *T1* liest Text-Strings von der Standardeingabe. Wenn nichts eingegeben wird, gibt er die Kontrolle an den Scheduler zurück, andernfalls schreibt er den Eingabepuffer als Textstring in den Ringpuffer *RB12*. Falls der Textstring zu lang für den Puffer ist (d.h. *writeRb()* liefert einen Fehler zurück), wird der aktuelle Input verworfen. Verwendet *select()*, um nicht-blockierendes Lesen von der Standardeingabe zu gewährleisten.
- *T2* liest den Eingabe-Ringpuffer *RB12*. Falls Daten vorhanden sind, wandelt er alle Kleinbuchstaben in Großbuchstaben (andere Zeichen bleiben unverändert) und gibt das Resultat in den Ringpuffer *RB23*.
- *T3* liest den Eingabepuffer *RB23* und setzt alle Zeichen, die im zuvor erhaltenen Telegramm an derselben Stelle vorhanden waren, auf Leerzeichen. Das Ergebnis schreibt er in den Ringpuffer *RB34*.
- *T4* liest aus *RB34* und gibt das Ergebnis auf dem Bildschirm aus.

Der Prozess (und seine Threads) sollen bei Auftreten der Signale SIGTERM und SIGINT beendet werden.