Übungen zur Vorlesung
Betriebssysteme I
Wintersemester 2001/02

AG BS
Jan Peleska, Aliki Tsiolakis

Blatt 4

Scheduling

Aufgabe 1: Implementierung eines Schedulers 100%

Teilaufgabe 1: Ringpuffer mit variabler Länge

Programmieren Sie ein C-Bibliothek, welche die Punkt-zu-Punkt Kommunikation über Telegramme variabler Länge zwischen Threads über Ringpuffer ermöglicht. Die Bibliothek sollte folgende Funktionen enthalten:

```
/*
  Erzeuge einen FIFO Ringpuffer der Gesamtgröße buffersize Bytes,
  der für die Kommunikation von Thread sourceThId nach Thread
  targetThId zu verwenden ist. Der Puffer wird dynamisch erzeugt. Die
  Verwaltungsinformation (Zeiger auf den Pufferanfang und Puffergröße,
  source/targe thread ids, Read/Write-Indices) wird in einer ebenfalls
  dynamisch zu allokierenden Struktur struct Rb_handle_t registriert
  und an den Aufrufer zurückgegeben.
*/
struct Rb_handle_t *initRb(int  targetThId,
                           int  sourceThId,
                           size_t  buffersize);

/*
  Schreibe ein neues Element "item" der Länge "itemsize" in den Ring-
  puffer. Der Befehl schlägt fehl, wenn eine der folgenden
  Bedingungen erfuellt ist:
    - targetThId/sourceThId passen nicht zum handle
    - Es ist nicht mehr genügend Platz im  Puffer, um den Item der
      Länge itemsize Bytes im Ringbpuffer unterzubringen.
*/
int writeRb(struct Rb_handle_t  *handle,
            int  targetThId,
            int  sourceThId,
            const void  *item,
            size_t  itemsize);

/*
  Analog für Lesen des ersten Elementes aus dem Ringpuffer.
  Bei Aufruf trägt man in itemsize die Länge des vom Anwendungs-
  programm zur Verfügung gestellten Puffers ein. Bei Rückkehr der
  Funktion ist dort die Länge der tatsächlich nach item kopierten
  Daten enthalten.
```

```
  Der Rückgabewert ist
       1   - falls erfolgreich und vollständig nach item
             kopiert wurde
       0   - falls nichts im Puffer war
       -1  - falls nicht das ganze Telegramm nach item passte.
             In diesem Fall wird item komplett gefüllt und der
             Rest kann mit einem nachfolgenden Aufruf abgeholt
             werden. Er geht also nicht verloren.
      -2 ... bei Fehlern analog zu writeRb().
 */
int readRb(struct Rb_handle_t  *handle,
           int  targetThId,
           int  ourceThId,
           void  *item,
           size_t  *itemsize);

/*
  Deallokiere alle zum Ringpuffer gehörigen Puffer.
*/
int freeRb(struct Rb_handle_t  *handle);
```

Teilaufgabe 2: Nicht-präemptives priorisiertes User-Space Scheduling

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

```
/*
  Registriere einen neuen User-Thread mit Identifikation threadId beim
  Scheduler, indem ein Funktionspointer auf die Thread-Funktion zur
  Initialisierung, die Thread-Main-Funktion zur zyklischen
  Verarbeitung und die Funktion zur Terminierung als Parameter mit-
  gegeben werden.
  prio ist eine statische Priorität: Der Scheduler aktiviert die
  Thread-Main-Funktion in jedem (prio+1)-tem Scheduling-Zyklus. 0 ist
  also die beste Priorität.
  Der Scheduler registriert den Thread und ruft danach sofort die
  Initialisierungsfunktion auf.
*/
int registerThread(int  threadId,
                   int  (*myThreadInit)(int threadId),
                   int  (*myThread)(int threadId, void *context),
                   int  (*myThreadTerminate)(int threadId, void *context),
                   int  prio);

/*
  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 ent-
  sprechenden 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.
*/
int registerContext(int threadId, void *context);

/*
  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.
*/
int schedActivateUserThreads(void);

/*
  Der folgende Aufruf bewirkt, dass der Scheduler alle Terminierungs-
  funktionen aufruft.
*/
int schedTerminateUserThread(void);
```

Die Verwendung im Anwendungsprogramm ist folgendermaßen:

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

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

void t1Terminate(int threadId) {
  ...
}
//... weitere Thread-Funktionen


int main() {

   registerThread(1,t1Init,t1,t1Terminate,0);
   registerThread(2,...);
```

```
 while ( ! terminationCondition ) {
    schedActivateUserThreads();
 }

 schedTerminateUserThread();
}
/*-------------------------------------------------*/
```

Die `terminationCondition` könnte beispielsweise von einem Signal-Handler gesetzt werden.

Teilaufgabe 3: Anwendung von Teilaufgabe 2

Schreiben Sie einen Prozess, der 4 User Threads `T1`, `T2`, `T3`, `T4` mit der Technik von Teilaufgabe (2) enthält, die auf folgende Weise miteinander über die Ringpuffer von Teilaufgabe (1) kommunizieren (sogenannte Pipeline-Verarbeitung): `T1` → `T2` → `T3` → `T4`

- **T1** liest UDP/IP-Telegramme mit der NO-WAIT-Option von einem externen Client-Programm. (Für NO-WAIT benötigt man den `fcntl()`-System Call.) Wenn nichts im socket ist, gibt er die Kontrolle an den Scheduler zurück, andernfalls schreibt er den Eingabepuffer als Textstring in den Ringpuffer `T1` → `T2`. Falls der Textstring zu lang für den Puffer ist (d.h., writeRb einen Fehler liefert), wird der aktuelle Input verworfen.
- **T2** liest den Eingabe-Ringpuffer. Falls Daten vorhanden sind, wandelt er alle Kleinbuchstaben in Großbuchstaben (andere Zeichen bleiben unverändert) und gibt das Resultat in den Ringpuffer `T2` → `T3`.
- **T3** liest den Eingabepuffer und setzt alle Zeichen, die im zuvor erhaltenen Telegramm an derselben Stelle vorhanden waren, auf Blank. Das Ergebnis schreibt er in den Ringpuffer `T3` → `T4`.
- **T4** gibt das Ergebnis auf dem Bildschirm aus.

Das **externe Client-Programm** liest von `stdin` und schreibt diesen Text-String auf den UDP-Socket, der von Thread `T1` gelesen wird.

Geben Sie für die Aufgabe eine **schriftliche Lösung** ab, die den Lösungsansatz, aufgetretene Probleme, Testfälle und -ergebnisse sowie den gut dokumentierten Sourcecode enthält. **Zusätzlich** schicken Sie den (dokumentierten) Sourcecode per Email an `tsio@informatik.uni-bremen.de`.

Abgabe: Bis Freitag, 11. Januar 2002.

Sowohl bei der schriftlichen Lösung als auch im Source-Code die Gruppennummer und die Namen aller Gruppenmitglieder nicht vergessen!