Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
WS 07/08

AG Betriebssysteme FB3
Kirsten Berkenkötter, Jan Peleska

Ausgabe: 15.1.2008
Abgabe: 29.1.2008

Übungszettel 4

Hinweise

Die Abgabe erfolgt als Ausdruck am Ende der Vorlesung und als E-Mail an *kirsten@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 (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: Nicht-präemptives priorisiertes User-Space Scheduling

Implementiert eine Scheduler-Bibliothek (*scheduler.c/scheduler.h*) auf Basis von *setjmp()/longjmp()* in C. Als Basis könnt ihr das Beispiel aus der Vorlesung verwendet. Beachtet aber folgende Änderungen: a) es sollen keine LWPs unterstützt werden; b) es sollen keine Arrays zur Verwaltung der Threads eingesetzt werden; c) das Scheduling ist priorisiert. Die Datei *mangling.h* kann unverändert verwendet werden.

Die folgenden Funktionen sollen zur Verfügung gestellt werden:

Registrieren eines Threads

```
int schedRegisterUserThread(schedFuncPtr_t func,
                            void *args,
                            unsigned int argsize,
                            char *name,
                            unsigned int prio);
```

Ein neuer User-Thread wird beim Scheduler registriert, indem ein Funktionspointer *func* auf die Thread-Funktion übergeben wird. Ausserdem werden die Parameter als void–Pointer *args* und die Größe dieses Pointers *argsize* benötigt. *name* ist der Name des Threads, der zur Identifizierung dient. *prio* ist eine statische Priorität: Der Scheduler aktiviert die Thread-Funktion in jedem (prio+1)tem Scheduling-Zyklus. *0* ist also die höchste Priorität.

Der Scheduler registriert den Thread, indem er ihn in einer Ringliste abspeichert. In dieser Liste befinden sich ausschließlich Daten der registrierten Threads, nicht des Schedulers selber.

Die Rückgabewerte sind wie folgt:

- -2: nicht genug Speicher vorhanden
- -1: fehlerhafte Parameter
- 0: erfolgreiche Registrierung

Entfernen eines Threads

```
void schedUnregisterUserThread(char *name);
```

Ein Thread wird nach Beendigung aus der Ringliste entfernt. Dies kann zum einen durch erfolgreiche Beending des Threads mit *return* erfolgen (impliziter Aufruf von *schedUnregisterUserThread()*) oder durch einen expliziten Aufruf von *schedUnregisterUserThread()* von aussen, z.B. durch einen Signalhandler.

In dieser Funktion werden alle allokierten Speicherbereiche freigegeben und der Thread aus der Ringliste entfernt. Der zu löschende Thread wird mit Hilfe seines Namens *name* identifiziert.

Scheduling

```
void schedActivateUserThread();
```

Die registrierten Threads werden nach dem oben beschriebenen Prinzip der statischen Priorität geschedult. Wenn alle Threads beendet sind (oder keine vorhanden), beendet sich der Scheduler.

Freiwillige Abgabe der CPU

```
void schedYield()
```

Die function *schedYield()* sorgt dafür, dass ein korrekter Kontextwechsel stattfindet.

Hilfsfunktion

Starten und Beenden von Threads

```
void schedWrapper()
```

Die Funktion *schedWrapper()* sorgt dafür, dass ein Thread korrekt gestartet und beendet wird. Sie wird intern im Scheduler verwendet.

Aufgabe 2: Ringpuffer mit Threads und Scheduling

In dieser Aufgabe sollen der Ringpuffer von Übungsblatt 1 und die Schedulingbibliothek in einem Anwendungsprogramm verwendet werden.

Schreibt 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 1 kommunizieren (sogenannte Pipeline–Verarbeitung):

T1 → T2 → T3 → T4

- *T1* liest Text–Strings von der Standardeingabe. Wenn eine Sekunde lang nichts eingegeben wird, gibt er die Kontrolle an den Scheduler zurück, andernfalls schreibt er den Eingabepuffer als Textstring in den Ringpuffer *RB12*. Verwendet *select()*, um nicht–blockierendes Lesen von der Standardeingabe zu gewährleisten.

- *T2* liest den Eingabepuffer *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.

Alle Threads erfüllen ihre jeweilige Aufgabe permanent in einer while–Schleife. Nach jedem Durchlauf wird die CPU freiwillig abgegeben, ebenso.

Für die Verwendung der Ringpuffer gilt folgendes:

- Das Schreiben in einen Ringpuffer erfolgt in einer while–Schleife. Wenn ein Fehler auftritt (falsche Parameter, ...), beendet der Thread sich mit return. Wenn nicht geschrieben werden kann, weil der Puffer gerade voll ist, wird die CPU freiwillig abgegeben. Wenn das Schreiben erfolgreich war, wird die while–Schleife verlassen.
- Das Lesen aus einem Ringpuffer erfolgt in einer while–Schleife. Wenn ein Fehler auftritt (falsche Parameter, ...), beendet der Thread sich mit return. Wenn nicht gelesen werden kann, weil der Puffer gerade leer ist, wird die CPU freiwillig abgegeben. Wenn das Lesen erfolgreich war, wird die while–Schleife verlassen.

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