Übungen zur Vorlesung
**Betriebssysteme I**
Studiengang Informatik
WS 09/10

AG Betriebssysteme FB3
Jan Peleska, Florian Lapschies

Ausgabe: 14.12.2009
Abgabe: 07.01.2010

# Übungszettel 4

## 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 verwenden. 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 einen Thread mit der Priorität *prio* in jedem *prio*-tem Scheduling-Zyklus. Hat jedoch mindestens ein Thread die Priorität 0, werden in jedem Zyklus ausschließlich die Threads mit der Priorität 0 nacheinander aktiviert.

Der Scheduler registriert den Thread, indem er ihn in einer Ringliste abspeichert.

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 Beendigung 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 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.

### Änderung der Priorität

```
int schedPrio(char *name, unsigned int prio)
```

Die Funktion *schedPrio()* ändert die Priorität des angegebenen Threads.

Die Rückgabewerte sind wie folgt:

- -1: fehlerhafte Parameter
- 0: erfolgreiche Änderung der Priorität

### 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 soll die Schedulingbibliothek in einem Anwendungsprogramm verwendet werden.

Schreibt einen Prozess, der drei User–Threads *TF*, *TK* und *TD* mit der Bibliothek aus Aufgabe 1 verwaltet, die über einen Ringpuffer miteinander kommunizieren.

- *TF* liest in einer Schleife jeweils ein einzelnes Zeichen aus einer Datei und schreibt dieses Zeichen dann als Hexadezimal–String in den Ringpuffer. Am Ende jeden Schleifendurchlaufs wird mit *schedYield()* die CPU freiwillig abgegeben. Kann der String nicht in den Ringpuffer geschrieben werden, weil dieser voll ist, wird *schedYield()* aufgerufen und es wird bei der nächsten Aktivierung erneut versucht. Wenn das Ende der Datei erreicht ist, beendet sich *TF*.

  *TF* läuft stets mit der Priorität 2.

- *TD* liest in einer Schleife einen einzelnen Eintrag aus dem Ringpuffer und gibt ihn auf *stdout* aus. Am Ende jeden Schleifendurchlaufs oder wenn der Ringpuffer leer ist wird mit *schedYield()* die CPU freiwillig abgegeben.

  *TD* läuft bei der ersten Aktivierung zunächst mit der Priorität 0.

- *TK* liest in einer Schleife ein einzelnes Zeichen von *stdin* und schreibt dieses als String in den Ringpuffer. Am Ende jeden Schleifendurchlaufs wird mit *schedYield()* die CPU freiwillig abgegeben.

  *TK* läuft bei der ersten Aktivierung zunächst mit der Priorität 0. Sollte *TK* innerhalb von drei Sekunden kein Zeichen von *stdin* lesen können, wird die Priorität von *TD* und die von *TK* selbst auf 1 gesetzt und mit *schedYield()* die CPU freiwillig abgegeben. Sobald wieder ein neues Zeichen von *TK* gelesen wird, wird die Priorität von *TD* und *TK* wieder auf 0 zurückgesetzt.

  Im Ergebnis werden also keine neuen Zeichen aus der Datei gelesen, solange Eingaben auf *stdin* kommen. Verwendet hierbei *select()*, um nicht–blockierendes Lesen von der Standardeingabe zu gewährleisten.

Der Ringpuffer entspricht dem einfachem Ringpuffer, so wie er euch in der Vorlesung vorgestellt wurde: Im Ringpuffer werden lediglich Zeiger auf Strings gespeichert und die Zeichenketten selber liegen dann im Heap. Der Ringpuffer soll 30 Einträge groß sein.

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

### Hinweise

Die Abgabe erfolgt als Ausdruck am Ende der Vorlesung und zusätzlich elektronisch über das Subversion Repository. Die Dokumentation der Aufgabenlösung ist in LaTeX anzufertigen. Bitte vergesst nicht die Namen aller Gruppenmitglieder mitanzugeben.