Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
WS 2013/2014

AG Betriebssysteme FB3
Jan Peleska, Florian Lapschies

Abgabe: 23.01.2014

Übungszettel 5

Aufgabe 1: Bibliothek für priorisiertes User-Space Scheduling

Implementiert eine Scheduler-Bibliothek (*scheduler.c/scheduler.h*) auf Basis von *setjmp()/longjmp()* in C. Als Basis könnt ihr die Beispiele aus Vorlesung und Übung verwenden. Die Datei *mangling.h* kann unverändert verwendet werden.

Die folgenden Funktionen sollen zur Verfügung gestellt werden:

Starten der User-Space-Threads

```
int ut_sched_run_tasks(struct ut_sched_task *tasks,
                       unsigned int nb_tasks);
```

Mit der Funktion *ut_sched_run_tasks()* werden *nb_tasks*-viele User-Space-Threads gestartet. Die einzelnen Threads werden dabei über das Array *tasks* spezifiziert.

```
struct ut_sched_task {
  unsigned int id;
  unsigned int lwp_id;
  unsigned int priority;
  void (*entry_function)(void *);
  void *argument;
  size_t stack_size;
};
```

Die Einträge dieses Arrays sind vom Typ *struct ut_sched_task*, dessen einzelnen Felder die folgende Bedeutung haben:

- *id* ist ein Bezeichner mit dem der User-Space-Thread eindeutig referenziert werden kann.
- *lwp_id* legt fest, auf welchem LWP der User-Space-Thread laufen soll.
- *priority* legt die Priorität fest, mit der der User-Space-Thread auf einem LWP laufen soll.
- *entry_function* stellt die Einstiegsfunktion für den zu startenden User-Space-Thread dar.
- *argument* ist ein Parameter mit der die Einstiegsfunktion aufgerufen werden soll.
- *stack_size* ist die Größe des Stacks, den der User-Space-Thread nutzen soll.

Die Funktion überprüft zunächst, ob die Thread-Bezeichner eindeutig sind. Sind diese es nicht wird ein Fehler zurückgegeben. Ansonsten werden für die User-Space-Threads die Kontexte angelegt und mit *pthread_create()* so viele LWPs, wie es verschiedene *lwp_id*-Werte in dem Array gibt, erzeugt. Auf jedem der LWPs läuft dann eine Instanz des User-Space-Schedulers, der

die Threads mit der zugehörigen *lwp_id* schedulet. Der Scheduler aktiviert einen User-Space-Thread mit der Priorität *priority* in jedem *priority*-tem Scheduling-Zyklus. Hat jedoch mindestens ein User-Space-Thread auf dem LWP die Priorität 0, werden in jedem Zyklus ausschließlich die Threads mit der Priorität 0 nacheinander aktiviert. Wenn sich alle User-Space-Threads auf dem LWPs beendet haben, terminiert auch der LWP. *ut_sched_run_tasks()* wiederum wartet mit *pthread_join()* auf die Terminierung aller LWPs und kehrt erst dann zurück.

Die Rückgabewerte sind wie folgt:

- -2: nicht genug Speicher vorhanden
- -1: fehlerhafte Parameter
- 0: erfolgreicher Aufruf

Achtet darauf, dass im Falle eines Fehlers der Speicher, der von dieser Funktion angelegt wurde, wieder freigegeben wird und dass alle bis zum Auftreten des Fehlers erzeugten Pthreads beendet werden. Somit ist sichergestellt, dass beim Eintreten eines Fehlers durch diese Funktion keine Memory Leaks und unnütze Pthreads hinterlassen werden.

Damit die einzelnen Pthreads erst mit dem Schedulen anfangen, wenn die Erstellung aller anderen Pthreads erfolgreich war, sollten diese zunächst auf ein Flag pollen, welches die folgenden Situationen kodiert:

- *ut_sched_run_tasks()* hat die Erstellung der Pthreads noch nicht abgeschlossen und es muss weiter gewartet werden.
- Es ist ein Fehler beim Erstellen aufgetreten, so dass sich der Thread beenden muss.
- Das Erstellen der Pthreads wurde erfolgreich abgeschlossen, so dass das Scheduling beginnen kann.

Freiwillige Abgabe der CPU

```
void ut_sched_yield()
```

Mit der Funktion *ut_sched_yield()* gibt ein User-Space-Thread freiwillig die CPU ab, so dass der nächste auf dem LWP läuft.

änderung der Priorität

```
int ut_sched_prio(unsigned int id, unsigned int prio)
```

Die Funktion *sched_prio()* ändert die Priorität des angegebenen User-Space-Threads.

Die Rückgabewerte sind wie folgt:

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

Aufgabe 2: Ringpuffer mit Threads und Scheduling

In dieser Aufgabe soll die Schedulingbibliothek in einem Anwendungsprogramm verwendet werden.

Schreibt einen Programm, das drei User-Space-Threads *TF*, *TK* und *TD* mit der Bibliothek aus Aufgabe 1 verwaltet. Die Threads laufen dabei lediglich auf einem LWP und kommunizieren miteinander über einen Ringpuffer.

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

  TK läuft bei der ersten Aktivierung zunächst mit der Priorität 0. Sollte *TK* innerhalb von fünf Sekunden kein Zeichen von *stdin* lesen können, wird die Priorität von *TD* und die von *TK* selbst auf 1 gesetzt und mit *ut_sched_yield()* die CPU freiwillig abgegeben. Sobald wieder ein neues Zeichen von *TK* gelesen wird, wird die Priorität von *TD* und *TK* wieder auf 0 zurückgesetzt. Wenn das Ende der Eingabe erreicht ist, beendet sich *TK*.

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

Wenn sich sowohl *TF*, als auch *TK* beendet haben, beendet sich schließlich auch *TD*, nachdem er die restlichen Einträge des Ringpuffer ausgegeben hat. Somit kehrt *ut_sched_run_tasks()* zurück und das Programm beendet sich.

Der Ringpuffer soll sieben Einträge fassen können.

Hinweise

- Alle in dieser Aufgabe geforderten Programme sollen mit einem Makefile übersetzt werden können. In diesem Makefile sollen Abhängigkeiten des Build-Prozesses richtig erfasst sein. Des weiteren soll das Makefile auch eine Regel `clean` enthalten, die alle Komplilate wieder löscht.

- Achtet darauf, dass Ihr die Rückgaben aller Systemaufrufe auf mögliche Fehler überprüft. Im ungewünschten Fehlerfall soll eine erklärende Ausgabe auf `stderr` erfolgen und das Programm dann mit `exit(EXIT_FAILURE)` (siehe `man 3 exit`) beendet werden. Viele Funktionen setzen im Fehlerfall die Variable `errno` (siehe `man 3 errno`), über die

der genaue Fehler identifiziert werden kann. Hilfreich für Fehlerausgaben ist die Funktion `strerror()` (siehe `man 3 strerror`) zur Formatierung von `errno`. Studiert auf jeden Fall die entsprechenden man-Pages, um zu erfahren welche Rückgabewerte Fehler kennzeichnen und welche Fehler auftreten können.

- Achtet darauf, dass Ihr allen Speicher, den Ihr mit `malloc()` alloziert habt, auch zum Programmende wieder mit `free()` freigegeben habt. Bei der Behandlung von Fehlerfällen, bei denen das Programm mit einem Fehlercode beendet wird, darf darauf verzichtet werden. Hilfreich zum Aufdecken von Speicherlecks und Speicherzugriffsfehlern ist das Tool `valgrind`.
- Die Abgabe erfolgt als Ausdruck zu Beginn der Übung und zusätzlich elektronisch über das Git Repository.
- Die Dokumentation der Aufgabenlösung ist in LaTeX anzufertigen.