Übungen zur Vorlesung
Betriebssysteme I
Studiengang Informatik
WS 2011/2012

AG Betriebssysteme FB3
Prof. Dr. Jan Peleska, Florian Lapschies

Abgabetermin: 01.12.2011

Übungszettel 2

1 Die Speisenden Philosophen mit Semaphoren und Shared Memory

In Vorlesung wurde Euch das Konzept von Semaphoren vorgestellt. Ein gängiges illustratives Fallbeispiel für die Nutzung von Semaphoren ist das sogenannte *Philosophenproblem*:

Fünf Philosophen sitzen gemeinsam an einem runden Tisch. Vor jedem Philosophen liegt jeweils ein großer Teller voller Nudeln. Zwischen zwei nebeneinander sitzenden Philosophen liegt jeweils eine Gabel, so dass es also am Tisch genauso viele Gabeln wie Philosophen gibt. Jeder Philosoph *denkt* entweder oder er *isst*. Zum Essen nimmt sich ein Philosoph die beiden Gabeln zu seiner Seite. Nach dem Essen legt die Gabel wieder auf dem Tisch und denkt daraufhin wieder. Da in diesem Szenario die Ressource Gabeln begrenzt sind, können nie alle Philosophen gleichzeitig essen.

In dieser Aufgabe soll das Problem der Speisenden Philosophen nun in C mit UNIX System V Semaphoren und Shared Memory programmiert werden. Die einzelnen Philosophen sollen dabei als Prozesse nachempfunden werden, die auf einer gemeinsamen Datenstruktur im Shared Memory operieren:

```
1 #include <inttypes.h>
2
3 #define MAX_PHILS 5
4
5 typedef struct SHMData {
6   uint32_t nextPhilId;
7   uint32_t data[MAX_PHILS];
8 } SHMData_t;
```

Jedem Philosophen wird eine ID $philId \in \{0, \ldots, MAX\_PHILS\}$ vom Typ `uint32_t` zugeordnet, mit der dieser eindeutig identifiziert werden kann. Die nächste zu vergebene ID wird im Shared Memory im Feld `nextPhilId` gespeichert. Ein Philosoph mit der ID $philId$ operiert dann auf den Datenfeldern `data[philId]` und `data[(philId+1)%MAX_PHILS]`. Damit Philosophen exklusiv mit ihren beiden Datenfeldern arbeiten können, wird jeden Datenfeld i eine Semaphore $forkSem_i$ zugeordnet, über die der Zugriff geregelt wird. Des weiteren benötigen wir eine weitere Semaphore $couterSem$ für den Zugriff auf $nextPhilId$. Insgesamt werden also $MAX\_PHILS + 1$ viele Semaphoren benötigt.

1.1 Anlegen des Shared Memories und der Semaphoren

Nach dem Start legt ein Philosoph zunächst das Shared-Memory für die Datenstruktur `SHMData_t` mit den Zugriffsrechten `rw-------` an, wenn es nicht bereits existiert. Dann fügt er das Shared Memory seinem Addressraum hinzu. Das neu erstellte Shared Memory muss nicht extra initialisiert werden, da es bereits mit Null-Bytes gefüllt ist.

Danach müssen die Semaphoren erstellt und deren Zähler vom ersten Philosophen initialisiert werden. Hierzu ruft ein Philosophenprozess `semget()` zunächst mit den Flags `IPC_CREAT` und `IPC_EXCL` auf, um die Semaphoren zu erstellen, wenn es bislang noch kein anderer gemacht hat. Wenn dieser Aufruf Erfolg hat, ist der Philosoph der erste und initialisiert alle $MAX\_PHILS + 1$ Semaphoren mit `semctl()` und dem Befehl `SETALL` auf einen initialen Wert von 1. Schlägt der Aufruf fehl und ist dabei auch noch *errno* auf `EEXIST` gesetzt, ist der Prozess nicht der erste und die Semaphoren werden mit einem erneutem Aufruf von `semget()` ohne das Flag `IPC_EXCL` zur Verfügung gestellt. Ebenso wie das Shared Memory sollen auch die Semaphoren die Zugriffsrechte `rw-------` bekommen.

1.2 Zugriff auf die gemeinsame Datenstruktur

Nachdem das Shared Memory und die Semaphoren angelegt sind, wird einem Philosophen die ID $philId$ aus `nextPhilId` zugeordnet und danach $nextPhilId$ inkrementiert. Sollte es jetzt mehr als MAX_PHILS viele Prozesse geben, wird der Prozess mit einem Fehler beendet. Um den gegenseitigen Ausschluss während des Zugriffs auf $nextPhilId$ zu sichern wird dieser kritische Abschnitt mit der Semaphore $counterSem$ gesichert. Sobald dem Prozess seine ID zugewiesen wurde, schreibt er $philId$ nach `data[philId]`.

Danach wird folgende Endlosschleife vollzogen:

- Der Prozess macht mit einem einzigen `semop()` Aufruf ein Down auf $forkSem_i$ und $forkSem_{(i+1)\ \%\ MAX\_PHILS}$.
- Die Werte von `data[i]` und `data[(i+1)%MAX_PHILS]` werden vertauscht.
- Der Prozess legt sich mit `sleep()` für eine Sekunde schlafen.
- Der Prozess macht mit einem einzigen `semop()` Aufruf ein Up auf $forkSem_i$ und $forkSem_{(i+1)\ \%\ MAX\_PHILS}$.

Zur besseren Lesbarkeit kapselt Ihr bitte die Up- und Down-Operationen auf $forkSem$ und $counterSem$ in den Funktionen `void lockCounter()`, `void unlockCounter()`, `void lockForks()` und `void unlockForks()`.

1.3 Freigeben des Shared Memories und der Semaphoren

IPC Ressourcen sind solange im System vorhanden, bis sie explizit gelöscht werden. Daher sollen die Philosophen selbst dafür sorgen, dass das Shared Memory und die Semaphoren freigegeben werden, wenn der Benutzer `Cntl+C` in der Konsole drückt. Hierzu soll mit `signal()` (siehe `man 3 signal`) nach der Vergabe der ID ein Signal-Handler für `SIGINT` registriert werden. Im Handler sollen mit `semctl()` bzw. `shmctl()` und dem Befehl `IPC_RMID` diese Ressourcen freigegeben werden. Da nachfolgende Zugriffe der anderen Philosophen auf die Semaphoren nach dessem Entfernen fehlschlagen würden, muss bei einem Fehler von `semop()` der Wert `errno` unterschieden werden. Hat `errno` den Wert `EIDRM` wird das Programm mit `EXIT_SUCCESS` beendet, ansonsten mit `EXIT_FAILURE`.

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 am Ende der Übung und zusätzlich elektronisch über das Subversion Repository.
- Die Dokumentation der Aufgabenlösung ist in LaTeX anzufertigen.