Hochschule RheinMain
Fachbereich DCSM - Informatik
Prof. Dr. Reinhold Kröger
Prof. Dr. Martin Gergeleit

Betriebssysteme und Rechnerarchitektur
WS 2015/16
LV 3142

Übungsblatt 4
Bearbeitungszeit 2 Wochen
Abgabetermin: 14.12.2015, 4:00 Uhr

Viele Betriebssysteme bieten eine Thread-Abstraktion als Programmierschnittstelle zur Entwicklung von nebenläufigen Anwendungen an. In der UNIX-Welt hat sich dabei das 1995 erstmals standardisierte Pthread-API (POSIX Threads) durchgesetzt. Als „leichtgewichtige Prozesse" konzipiert, teilen sich mehrere Threads einen UNIX-Prozess und damit denselben Adressraum, geöffnete Dateien, Sockets, usw. In dieser Übung wird die Synchronisation mittels Semaphoren, Mutexen und Condition-Variablen zwischen Pthreads angewendet (vgl. Kap. 5.2.2- 5.2.4).

Die meisten UNIX-Systeme bieten zwei unterschiedliche Schnittstellen zur Synchronisation an, das ältere *System V Interprocess Communication* (IPC) API und das neuere POSIX-API. Beide Schnittstellen bieten Semaphoren, Shared Memory Segmente (Übungsblatt 6) und Message Queues (nicht besprochen) mit ähnlicher Funktionalität an. In diesem Praktikum soll die jeweilige POSIX-Variante benutzt werden. Beachten Sie bitte, dass die POSIX-Varianten immer einen Unterstrich im Namen haben, z. B. `sem_init()` oder `shm_open()`!

POSIX-Semaphoren unterscheiden zwischen *Named* und *Unnamed* Semaphoren. Ein Named Semaphor stellt ein persistentes Objekt im `/dev/shm`-Dateisystem dar und existiert so lange, bis es explizit wieder gelöscht wird. Weitere Prozesse können dann auf das Semaphor mit `sem_open()` zugreifen. Unnamed-Semaphoren werden dagegen mit `sem_init()` als Variable im Speicher des aufrufenden Prozesses angelegt und automatisch gelöscht, wenn dieser Prozess terminiert. Die include-Datei `<semaphore.h>` deklariert folgende Aufrufe:

`int sem_init(sem_t *sem, int pshared, unsigned int value);`	Erzeugen eines Semaphor-Objektes mit dem Startwert `value`. Das Flag `pshared` zeigt an, ob das Semaphor zwischen mehreren Prozessen geteilt werden kann (Wert ungleich 0), ansonsten nur im aufrufenden Prozess nutzbar.
`int sem_wait(sem_t *sem);`	Semaphor-Operation P() (Dekrementieren und evtl. Blockieren).
`int sem_post(sem_t *sem);`	Semaphor-Operation V() (Inkrementieren).
`int sem_destroy(sem_t *sem);`	Semaphor-Objekt zerstören.
`int sem_trywait(sem_t *sem);`	P()-Operation nur, falls Aufrufer nicht blockiert.
`int sem_getvalue(sem_t *sem, int *sval);`	Gibt den aktuellen Wert des Semaphors in einem integer zurück, auf das `sval` zeigt.

Die Manual Pages `pthreads` und `sem_overview` bieten einen Überblick über die vorhandenen Funktionen. Weitere Informationen zur Programmierung mit Pthreads sind

unter http://www.llnl.gov/computing/tutorials/pthreads/ oder im Lehrbuch Butenhof: "Programming with POSIX Threads", Addison-Wesley, 1997, verfügbar.

Aufgabe 4.1 (Einführendes Beispiel für wechselseitigen Ausschluss und Pthreads, Abgabe):

In dieser Aufgabe wird ein Semaphor zur Sicherung des wechselseitigen Ausschlusses mehrerer Threads in der Benutzung eines gemeinsamen Betriebsmittels eingesetzt. Das Betriebsmittel ist das Bildschirmfenster, in dem die Threads gestartet werden und in das sie jeweils Ausgaben (über ihre Standardausgabe) durchführen. Zunächst erfolge die Ausgabe der Threads unsynchronisiert.

(a) Schreiben Sie ein Programm `gibaus.c`, dass zuerst insgesamt NUM_THREADS=6 Threads startet und danach auf deren Beendigung wartet. Jeder Thread gebe zyklisch ITERATIONS=5 mal eine Zeile mit jeweils ANZ_ZEICHEN=10 eindeutigen Zeichen aus, z.B. 'A' für den ersten Thread, 'B' für den zweiten usw., damit können Sie die Ausgaben der Threads später unterscheiden. Nach der Ausgabe eines Zeichens lege sich der Thread für eine zufällige Zeit zwischen 100 und 500 Millisekunden schlafen, bevor er das nächste Zeichen (oder "\n") ausgibt. Hinweis: nach der Ausgabe eines Zeichens müssen Sie den Ausgabepuffer mit Hilfe der Funktion `fflush(stdout)` leeren, da `printf()` standardmäßig ganze Zeilen ausgibt. Was beobachten Sie bzgl. der Ausgabe der verschiedenen Threads?

(b) Erweitern Sie das Programm aus (a) zu einem Programm `gibaus_sem.c`, das sicherstellt, dass ein Thread die Zeichen **einer Zeile** atomar ausgibt (und damit ausschließlich vollständige Zeilen mit gleichen Zeichen entstehen). Nutzen Sie dazu ein POSIX-Semaphor.

(c) Ändern Sie das Programm aus (b) zu einem Programm `gibaus_mutex.c` ab, das anstelle eines POSIX-Semaphors ein Pthread-Mutex benutzt. Können Sie einen Unterschied im Verhalten feststellen?

Aufgabe 4.2 (Philosophen-Problem mit Mutexen und Condition-Variablen, Abgabe):

In dieser Übung wird das Problem der speisenden Philosophen betrachtet (vgl. Abschnitt 5.3.2 der Vorlesung). Dazu ist eine Lösung mittels Threads, Mutexen und Condition-Variablen aus dem POSIX-Standard zu entwickeln.

(a) Erstellen Sie eine Lösung für das bekannte Problem der speisenden Philosophen. Gehen Sie von der korrekten Lösung des Philosophenproblems aus, wie sie in der Vorlesung (Folien 5-63ff) für N Philosophen vorgestellt wurde. Dabei soll jeder Philosoph als POSIX Thread realisiert werden und RUNDEN=10 mal speisen. Nachdem alle Philosophen fertig sind, soll sich das Programm beenden und vorher alle benutzten Ressourcen wieder freigegeben werden. Für die Durchsetzung des wechselseitigen Ausschlusses benutzen Sie bitte Mutexe, und für die Blockierung der Philosophen Condition-Variablen. Die Zeitdauern aller Philosophen in den jeweiligen Zuständen seien durch Zufallszahlengeneratoren im Bereich von 1 bis 10 Sekunden bestimmt. Organisieren Sie die Ausgabe des Programms so, dass die relevanten Zustände der Philosophen sichtbar werden und alle Ausgaben eines Philosophen untereinander in einer Gruppe von Spalten erscheinen. Hinweis: Sie können bei der Entwicklung mit Hilfe der Funktion `assert()` (engl. Zusicherung) prüfen, ob sich Ihre Philosophen wirklich im korrekten Zustand befinden!

(b) Testen Sie die Lösung für N=1, 2, 3, 5 und 8 Philosophen. Überprüfen Sie Ihre Lösung auf Korrektheit.

(c) Speichern Sie die Ausgaben Ihrer endgültigen Version von (b) für mindestens 5 Denken/Essen-Zyklen in der Datei `philosophen.txt` und geben Sie diese Datei mit ab.

Aufgabe 4.3 (Synchronisations-Problem, Abgabe):

Für ein Blech seiner berühmten Zwergenplätzchen benötigt der Bäckermeister-Zwerg folgende Backzutaten: 2 kg Mehl, 5 Eier und 1 Liter Milch.

Damit das Backen reibungslos verläuft, versorgen ihn drei weitere Zwerge mit den jeweiligen Zutaten:

– der Eier-Zwerg bringt 1 Ei pro Minute,
– der Milch-Zwerg bringt 1 Liter Milch alle zwei Minuten,
– der Mehl-Zwerg bringt 1 kg Mehl alle drei Minuten.

Da die Bäckerei, wie bei Zwergen üblich, sehr klein ist, kann nur ein Zwerg zur gleichen Zeit die Backstube betreten: entweder ein Lieferzwerg, der seine Zutaten schnell auf dem Küchentisch ablegt und dann davon eilt, um weitere Zutaten zu holen, oder der Bäckermeister, der, wenn alle Zutaten abgeliefert wurden, mit dem Backen beginnt und nach 5 Minuten mit einem Blech herrlich duftender Plätzchen wieder hinauskommt. Es darf höchstens die jeweilige Zutatenmenge auf dem Küchentisch sein, die für exakt ein Blech benötigt wird.

Da Plätzchenbacken eine anstrengende Tätigkeit ist, hält der Bäckermeister in der Zwischenzeit ein Schläfchen vor der Backstube. Er wird von den Liefer-Zwergen immer dann geweckt, wenn der Vorrat einer Zutat für den nächsten Backvorgang aufgefüllt wurde. Der Bäckermeister schaut dann nach, ob die anderen Zutaten auch vollständig sind, damit er mit dem Backen anfangen kann. Ansonsten verlässt er die Backstube wieder und schläft weiter. Ebenso warten die Liefer-Zwerge vor der Backstube, wenn für weitere Vorräte auf dem Küchentisch kein Platz mehr ist, bis der Bäcker mit dem Backen des nächsten Blechs fertig ist.

(a) Modellieren Sie dieses Problem im Zeitraffer (Sekunden statt Minuten) in einem Programm `baeckerei.c` mit je einem Thread pro Zwerg. Die angelieferten Zutaten können dabei über Variablen gezählt werden. Nutzen Sie zum gegenseitigen Ausschluss aus der Backstube ein Mutex und zur Synchronisation der Liefer-Zwerge und des Bäckermeisters zwei Condition-Variablen für die Zustände *Vorrat aufgefüllt* und *Vorrat aufgebraucht*. Lassen Sie die Zwerge ihre aktuellen Zustände und Tätigkeiten (vor der Backstube warten, Backstube betreten, Backstube verlassen, Backen, Wecken, Warten) auf der Konsole in unterschiedlichen Spalten ausgeben:

```
                          Bäcker: es fehlen noch Zutaten, warte ...
Eier: Anlieferung
Eier: es sind 1 Ei(er) in der Backstube
Eier: Backstube verlassen
        Milch: Anlieferung
        Milch: es sind 1 Liter Milch in der Backstube
        Milch: wecke Bäcker
        Milch: Backstube verlassen
                          Bäcker: es fehlen noch Zutaten, warte ...
Eier: Anlieferung
Eier: es sind 2 Eier in der Backstube
Eier: Backstube verlassen
```

```
                Mehl: Anlieferung
                Mehl: es sind 1 kg Mehl in der Backstube
                Mehl: Backstube verlassen
Eier: Anlieferung
...
```

Simulieren Sie den zeitlichen Ablauf für 10 Bleche Plätzchen!

(b) Würde Ihre Lösung auch mit mehreren Bäckern funktionieren (natürlich unter der Annahme, dass nur ein Bäcker zu einem Zeitpunkt in der Küche backen kann)? Wie müssten Sie Ihr Programm anpassen?

Bewertung:

Aufgabe	Kriterien	Punkte
4.1 (a)	Threads erstellen und joinen	2
	Ausgaben	1
4.1 (b)	Korrekte Verwendung der Semaphoren	2
4.1 (c)	Korrekte Mutex-Verwendung	2
4.2 (a)	Threads erstellen und joinen	1
	Synchronisierung korrekt	4
	Variable Anzahl Philosophen	1
4.2 (b, c)	Test und Dokumentation	1
4.3 (a)	Threads erstellen und joinen	1
	Ausgaben	1
	Synchronisierung mit Mutexen und Condition-Variablen	4
4.3 (b)	Mehrere Bäcker (Zusatzpunkt)	1 (opt)
	Abzüge bei fehlender Return-Code-Behandlung	(-3)
	Abzüge bei Compiler-Warnungen	(-2)
	Abzüge Lesbarkeit / Kommentare	(-2)
	Gesamt	**20 / 21**