Hochschule RheinMain
Fachbereich DCSM - Informatik
Prof. Dr. Robert Kaiser
Sebastian Flothow
Alexander Schönborn
Daniel Schultz

Betriebssysteme

WS 2020/21
LV 3122

Aufgabenblatt 3
Bearbeitungszeit 2 Wochen
Abgabetermin: 21.12.2020, 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 diesem Aufgabenblatt wird die Synchronisation mittels **Semaphoren** und **Mutexen** zwischen Pthreads angewendet (vgl. Kap. 5.2.2-5.2.4).

`int pthread_create(`
` pthread_t *thread,`
` const pthread_attr_t *attr,`
` void *(*func)(void *arg),`
` void *arg);`
	Erzeugen eines Threads. Der erzeugte Thread führt die angegebene Funktion `func` aus und bekommt das angegebene Argument `arg` übergeben. Der Wert `NULL` für `attr` erzeugt einen Thread mit Default-Attributen.

Return der Thread-Funktion, oder: `void pthread_exit(void *retval);`	Terminieren des aufrufenden Threads, mit Rückgabewert der Funktion oder in `retval`.

`int pthread_join(`
` pthread_t thread,`
` void **retval);`
	Auf die Beendigung eines Threads warten. Der Rückgabewert des Threads wird in `retval` zurückgegeben.

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 (dieses Praktikum), Shared Memory Segmente (Aufgabenblatt 7) 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_destroy(sem_t *sem);`	Semaphor-Objekt zerstören.
`int sem_wait(sem_t *sem);`	Semaphor-Operation $P()$ (Dekrementieren und evtl. Blockieren).
`int sem_post(sem_t *sem);`	Semaphor-Operation $V()$ (Inkrementieren).

Äquivalente Funktionen für Mutexe sind in `<pthread.h>` definiert:

`int pthread_mutex_init( pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`	Erzeugen eines Mutex-Objektes im ungesperrten Zustand. Der Wert `NULL` für `attr` erzeugt ein Mutex mit Default-Attributen.
`int pthread_mutex_destroy( pthread_mutex_t *mutex);`	Mutex-Objekt zerstören.
`int pthread_mutex_lock( pthread_mutex_t *mutex);`	Mutex-Objekt sperren. Blockiert solange das Mutex von einem anderen Thread gesperrt ist.
`int pthread_mutex_unlock( pthread_mutex_t *mutex);`	Mutex-Objekt freigeben.

Condition-Variablen erlauben die Signalisierung von Ereignissen:

`int pthread_cond_init( pthread_cond_t *cond, const pthread_condattr_t *attr);`	Erzeugen einer Condition-Variable.
`int pthread_cond_destroy( pthread_cond_t *cond);`	Condition-Variable zerstören.
`int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex);`	Ein zuvor gesperrtes Mutex-Objekt freigeben und dann auf der Condition-Variable warten. Nach dem Aufwachen wird das Mutex-Objekt wieder gesperrt.
`int pthread_cond_signal( pthread_cond_t *cond);`	Einen wartenden Thread wecken.
`int pthread_cond_broadcast( pthread_cond_t *cond);`	Alle wartenden Threads wecken.

Da das Warten auf einer Condition-Variablen aus anderen Gründen unterbrochen und ein wartender Thread fälschlicherweise geweckt werden kann (engl. *spurious wakeups*), muss immer geprüft werden, ob nach dem Aufwachen die Aufweckbedingung erfüllt ist. Das Grundmuster für das sichere Zusammenspiel von Mutexen und Condition-Variablen sieht daher wie folgt aus:

```
                    pthread_mutex_t m;
                    pthread_cond_t cv;
                    Bool tuwas = FALSE;

// Wartender Thread                 // Signalisierender Thread
pthread_mutex_lock(&m);             pthread_mutex_lock(&m);
...                                 ...
```

```
while (!tuwas)                          tuwas = TRUE;
    pthread_cond_wait(&cv, &m);         pthread_cond_signal(&cv);
...                                     ...
pthread_mutex_unlock(&m);               pthread_mutex_unlock(&m);
```

In diesem Beispiel wird der wartende Thread auf der linken Seite und der aufweckende Thread auf der rechten Seite dargestellt. Gemeinsame Variablen sind in der Mitte oberhalb der Codefragmente dargestellt. Alle Operationen auf den Condition-Variable finden innerhalb des durch das Mutex geschützen Bereiches statt. Beachten Sie, dass `pthread_cond_wait` das umschliessende Mutex intern vor dem Warten freigibt und nach dem Aufwecken wieder lockt. Ebenfalls ist die Variable `tuwas` in diesem Beispiel nur ein Platzhalter für die Aufweckbedingung. Die Aufweckbedingung in Ihren Programmen hängt später ganz konkret von der Problemstellung ab!

Die Manual Pages `pthreads` und `sem_overview` bieten einen Überblick über die vorhandenen Funktionen. Weitere Informationen zur Programmierung mit Pthreads sind im Lehrbuch Butenhof, „Programming with POSIX Threads“, Addison-Wesley, 1997, verfügbar, oder online: `https://computing.llnl.gov/tutorials/pthreads/`

Aufgabe 3.1 (Wechselseitiger Ausschluss):

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`, das zuerst insgesamt sechs Threads startet und danach auf deren Beendigung wartet. Diese Threads werden dann im Folgenden parallel ausgeführt. Jeder Thread gebe zyklisch fünf mal eine Zeile mit jeweils zehn eindeutigen Zeichen aus, z.B. 'A' für den ersten Thread, 'B' für den zweiten usw., damit können Sie die Ausgaben der Threads später unterscheiden können. 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.

   Was beobachten Sie bezüglich der Ausgabe der verschiedenen Threads?

   Hinweis: Nach der Ausgabe eines Zeichens müssen Sie den Ausgabepuffer mit Hilfe der Funktion `fflush(stdout)` leeren, da `printf()` standardmäßig ganze Zeilen ausgibt!

b) Erweitern Sie das Programm aus (a) zu einem Programm `gibaus_sem.c`, das sicherstellt, dass ein Thread die Zeichen **einer Zeile** exklusiv (ohne Störungen der anderen Threads) ausgibt, damit ausschließlich vollständige Zeilen mit gleichen Zeichen entstehen.

   Nutzen Sie dazu ein POSIX-Semaphor.

   Achtung: MacOS unterstützt keine Unnamed Semaphoren. Bitte lösen Sie diese Aufgabe auf einem Linux-System!

c) Ändern Sie das Programm aus (b) zu einem Programm `gibaus_mutex.c` ab, das anstelle eines POSIX-Semaphors ein Pthread-Mutex benutzt.

Aufgabe 3.2 (Zwergenbäckerei):

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 wieder 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. Benutzen Sie zur Signalisierung der schlafenden Lieferzwerge eine Condition-Variable *„Vorrat aufgebraucht“* und zur Signalisierung des schlafenden Bäckermeisters eine Condition-Variable *„Vorrat aufgefüllt“*. Lassen Sie die Zwerge ihre aktuellen Zustände und Tätigkeiten (vor der Backstube schlafend, Backstube betreten wollen, Backstube betreten haben, Backstube verlassen, Anliefern, Backen, Wecken) 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 Ei(er) 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!

Hinweis: Sie können bei der Entwicklung mit Hilfe der Funktion `assert(<bedingung>)` (engl. Zusicherung) prüfen, ob sich die Zwerge tatsächlich in einem korrekten Zustand befinden!

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)?

Passen Sie Ihr Programm für einen zweiten Bäcker an.