Übungen zur Synchronisation und zu Deadlocks

Laden Sie das Programmarchiv zur aktuellen Übung mit

```
wget http://swf.hgesser.de/b1-ss2017/prakt/swf-bs1-ss2017-uebung04.tgz
```

von der Webseite herunter, entpacken Sie es mit `tar xzf swf-bs1-ss2017-uebung04.tgz` und wechseln Sie in den neuen Ordner `ue04/`.

- Aufgaben 1, 2 und 4: **in Heimarbeit**;
  Besprechung am 16.06. (Nr. 1, 2) bzw. 22.06. (Nr. 4)
- Aufgabe 3: während der Vorlesung

Teil A: Synchronisation

1. Nullsummenspiel mit und ohne Mutexe

a) Die Datei `adders.c` enthält ein kleines Programm, das 100 Threads erzeugt, welche alle dieselbe globale Variable `glob` verändern. Lesen Sie das Programm und finden Sie heraus, was es tut. Die Funktion `waste_time()` ist nur dazu da, etwas Zeit zu verschwenden, damit der Scheduler zu einem anderen Thread umschaltet. Was sollte am Ende des Hauptprogramms als Startwert und Endwert ausgegeben werden?

b) Übersetzen Sie das Programm mit

```
gcc -pthread -o adders adders.c
```

und testen Sie es mehrfach. Hierbei sollten Sie unterschiedliche Endwerte erhalten – ein Hinweis darauf, dass das Programm nicht korrekt arbeitet. Um das Programm automatisiert 100x auszuführen und nur verschiedene Endwerte auszugeben, können Sie z. B. in der Shell eine Schleife programmieren und die Ausgaben mit `sort -u` sortieren:

```
for i in $( seq 1 100 ); do ./adders | grep Endwert; done | sort -u
```

c) In einem ersten Schritt beheben Sie das Problem, dass die Variable `local` gar nicht lokal definiert ist: Entfernen Sie die Deklaration von `local` am Anfang des Programms und fügen Sie lokale Deklarationen in den Funktionen `add()` und `sub()` ein (am Anfang: `int local = glob;` statt `local = glob;`).

Kompilieren und testen Sie das Programm erneut.

d) Das Programm arbeitet immer noch fehlerhaft, weil der Zugriff auf `glob` parallel erfolgt. Der ganze Code in den Funktionen `add()` und `sub()` ist jeweils ein kritischer Abschnitt – Sie brauchen einen Mutex, um die kritischen Abschnitte zu schützen. In der Programmversion `adder-mutex.c` ist dieser Mutex bereits deklariert (`pthread_t mutex;`) – er muss aber noch

- in `main()` initialisiert werden, bevor die Threads erzeugt werden,
- und in `add()` und `sub()` jeweils ganz am Anfang (vor dem Auslesen von `glob`) gelockt und ganz am Ende (nach dem Zurückschreiben von `glob`) unlockt werden.

Für die Initialisierung, das Locking und Unlocking benötigen Sie drei Bibliotheksfunktionen. Lesen Sie dazu die Manpages zu `pthread_mutex_init()`, `pthread_mutex_lock()` und `pthread_mutex_unlock()`. (Falls diese nicht installiert sind, verwenden Sie die Manpage-Dateien im Ordner `ue04/`, die Sie z. B. mit `man ./pthread_mutex_init.3posix.gz` anzeigen können.) Das zweite Argument `attr` von `pthread_mutex_init()` können Sie auf `NULL` setzen.

Kompilieren und testen Sie das neue Programm mit

```
gcc -pthread -o adders-mutex adders-mutex.c
for i in $( seq 1 100 ); do ./adders-mutex | grep Endwert; done | sort -u
```

– jetzt sollte es korrekt arbeiten und immer 0 zurück geben.

2. Condition Variables

Neben Mutexen und Semaphoren gibt es mit **Condition Variables** noch ein drittes, häufig verwendetes Tool zur Synchronisation. Das Ziel beim Einsatz dieser Variablen ist, dass ein Thread sich schlafen legt, weil er darauf wartet, dass eine bestimmte Bedingung (*condition*) erfüllt ist – z. B., dass in einem Puffer wieder Platz zum Schreiben ist. Ein anderer Thread kann dann über diese Variable ein Signal an den schlafenden Prozess senden, wenn er weiß, dass die Bedingung inzwischen erfüllt ist (z. B., weil er Platz im Puffer geschaffen hat): Dadurch wird der schlafende Thread aufgeweckt.

a) Lesen Sie die kurze Beschreibung von Condition Variables auf der Seite

`https://computing.llnl.gov/tutorials/pthreads/#ConditionVariables`

Ergänzend können Sie auch die Manpages zu `pthread_cond_init()`, `pthread_cond_wait()` und `pthread_cond_signal()` (bis einschließlich zum Abschnitt „Waiting and Signaling on Condition Variables“) lesen. Beim Aufruf von `pthread_cond_wait` wird auch ein Mutex als zweites Argument benötigt. Machen Sie sich das Zusammenspiel von Condition Variable und Mutex klar.

b) Unter

`http://www.cs.nmsu.edu/~jcook/Tools/pthreads/pc.c`

finden Sie eine Implementierung des Erzeuger-Verbraucher-Problems (*producer consumer problem*) mit Hilfe von Condition Variables. Lesen Sie den Quellcode und überlegen Sie sich, wie die Signalisierung zwischen den Erzeuger- und Verbraucher-Threads funktioniert: Wann legen sich die Threads schlafen und wie werden sie wieder aufgeweckt?

Teil B: Deadlocks

3. Deadlocks (Theorie)

Es gebe fünf Prozesse P_1, P_2, P_3, P_4 und P_5 sowie sechs Ressourcen R_1, R_2, R_3, R_4, R_5, R_6. Es gelten dabei die Belegungen und Anforderungen auf der rechten Seite.

Zeichnen Sie den Ressourcen-Zuordnungsgraph für dieses Szenario und leiten Sie daraus ab, ob sich die fünf Prozesse im Deadlock-Zustand befinden. Begründen Sie Ihre Antwort.

- P_1 hat R_2 belegt und fordert R_5 an.
- P_2 hat R_4 belegt und fordert R_2 und R_3 an.
- P_3 hat R_3 belegt und fordert R_1 an.
- P_4 hat R_1 belegt und fordert R_6 an.
- P_5 hat R_5 belegt und fordert R_4 an.

4. Deadlocks (Praxis)

Betrachten Sie das auf Seite 4 abgedruckte Programm, das Sie auch als `deadlock.c` im Quellcode-Archiv zu diesem Aufgabenblatt finden.

a) Ohne aktiviertes Cheating läuft dieses Programm in einen Deadlock. Woran liegt das und wie könnten Sie das Problem lösen?

b) Testen Sie das Programm in der VM – einmal mit und einmal ohne aktiviertes Cheating.

c) Schalten Sie das Cheating wieder aus und korrigieren Sie die Synchronisation (wie in Teil **a)**).

d) Welchen Nachteil hat die korrekte Implementierung?

```
// deadlock.c
#include <stdio.h>
#include <pthread.h>

pthread_t       thr1,  thr2;
pthread_mutex_t res_A, res_B;

void *thread1 (void *args) {
  pthread_mutex_lock (&res_A);    printf ("thread1: locked A\n");
  sleep (1);  // Thread-Wechsel durch Scheduler ermöglichen
  pthread_mutex_lock (&res_B);    printf ("thread1: locked B\n");

  printf ("thread1: critical!\n"); sleep (1);

  pthread_mutex_unlock (&res_A);  printf ("thread1: unlocked A\n");
  pthread_mutex_unlock (&res_B);  printf ("thread1: unlocked B\n");
}

void *thread2 (void *args) {
  pthread_mutex_lock (&res_B);    printf ("thread2: locked B\n");
  sleep (1);  // Thread-Wechsel durch Scheduler ermöglichen
  pthread_mutex_lock (&res_A);    printf ("thread2: locked A\n");

  printf ("thread2: critical!\n"); sleep (1);

  pthread_mutex_unlock (&res_A);  printf ("thread2: unlocked A\n");
  pthread_mutex_unlock (&res_B);  printf ("thread2: unlocked B\n");
}

// Folgende Zeile auskommentieren, um Cheating zu aktivieren
// #define CHEATING

int main () {
  pthread_mutex_init(&res_A, NULL);
  pthread_mutex_init(&res_B, NULL);

  // Threads erzeugen
  pthread_create (&thr1, NULL, thread1, NULL);
  pthread_create (&thr2, NULL, thread2, NULL);

  #ifdef CHEATING
    // Cheating
    sleep (5);
    printf ("main: Nach 5 Sekunden: Cheat (unlocking A)...\n");
    pthread_mutex_unlock (&res_A);    // illegal!
  #endif

  // Threads einsammeln
  pthread_join (thr1, NULL);
  pthread_join (thr2, NULL);

  printf ("Fertig\n");
}
```