Lehrstuhl Connected Mobility
Fakultät für Informatik
Technische Universität München

Grundlagen von Betriebssystemen und Systemsoftware

WS 2018/19

Übung 5: Threads und Synchronisation

zum 19. November 2018

- Die **Hausaufgaben** für dieses Übungsblatt müssen **spätestens** am **Sonntag, den 25.11.2018 um 23:59** in Moodle hochgeladen worden sein.
- Alle C-Programme müssen mit den folgenden Flags kompiliert werden:
  `gcc -Wall -o <progname> <prog.c>`.
- Hierzu stellen wir für jedes Übungsblatt jeweils ein Makefile bereit, das **nicht** verändert werden darf, um sicherzustellen, dass Ihre Abgabe auch korrekt von uns getestet und bewertet werden kann. **Ihr Programmcode muss zwingend mit dem Makefile kompilierbar sein. Anderenfalls kann die Abgabe nicht bewertet werden.** Wenn Sie zwischenzeitlich Änderungen vornehmen wollen, um etwa bestimmte Teile mittels `#ifdef` einzubinden und zu testen, dann kopieren Sie am besten das Makefile und modifizieren Ihre Kopie. Mit `make -f <Makefile>` können Sie dann eine andere Datei zum Übersetzen verwenden.
- Die Abgabe der Programme erfolgt als Archivdatei, die die verschiedenen Quelldateien (.{c|h}) umfasst und **nicht** in Binärform. Nicht kompilierfähiger Quellcode wird **nicht gewertet**.
- Um die Abgabe zu standardisieren enthält das Makefile ein Target "submit" (`make submit`), was dann eine Datei blatt05.tgz zum Hochladen erzeugt.
- Damit die richtigen Dateien hochgeladen und ausgeführt werden, geben wir bei allen Übungen die jeweils zu verwendenden Dateinamen für den Quellcode und auch für das Executable an.
- Die Tests werden auf diesem Aufgabenblatt mittels Python-Programmen durchgeführt. Zum Ausführen der Tests geben Sie `python xyz_test.pyc` auf der Konsole ein. Ihre ausführbaren Programme müssen sich im selben Verzeichnis wie die Testprogramme befinden.

1 Vorbereitung auf das Tutorium

Was ist die Rolle der folgenden POSIX- (Portable Operating System Interface) bzw. Bibliotheksfunktionen? Erläutern Sie kurz die Parameter und den Rückgabewert jeder Funktion. [1]

- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);`
- `int pthread_mutex_lock(pthread_mutex_t *mutex);`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex);`
- `int pthread_mutex_trylock(pthread_mutex_t *mutex);`
- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
- `int pthread_cond_destroy(pthread_cond_t *cond);`
- `int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec *restrict abstime);`
- `int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);`
- `int pthread_cond_broadcast(pthread_cond_t *cond);`
- `int pthread_cond_signal(pthread_cond_t *cond);`
- `int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr);`
- `int sprintf(char *restrict buffer, const char *restrict format, ...);`
- `int open(const char *path, int oflag, ...);`
- `int close(int filedes);`

2 Tutoraufgaben

2.1 Thread-Safety für Funktionen

Viele Bibliotheksfunktionen in C sind in ihrer jeweils ursprünglichen Form nicht geeignet, (ohne Schutz durch Semaphore oder andere Synchronisationsmechanismen) von mehreren Threads verwendet zu werden.

Erläutern Sie das Problem anhand der folgenden Beispielfunktionen. Welche Regels gelten für sogenannte reentrante Funktionen? Welche Rolle spielen hier `static`-Variablen in Funktionen?

- `void srand (unsigned seed); int rand (void);`

  →

  `int s_rand (unsigned *seed);`
- `char *strtok(char *restrict str, const char *restrict sep);`

  →

  `char *strtok_r(char *restrict str, const char *restrict sep, char **restrict lasts);`

[1] Erläuterungen und Beispiele der einzelnen Funktionen finden Sie auch in den **man**-Pages auf Ihrer Linux-VM.

2.2 Speisende Philosophen

Wir haben uns mit dem Problem der speisenden Philosophen beschäftigt und dabei in der Vorlesung eine verklemmungs- und verhungerungsfreie Lösung kennengelernt, die allerdings zu jedem Zeitpunkt nur maximal einen Philosophen essen lässt, obwohl zwei gleichzeitige Esser möglich sind. Wie und warum funktioniert der folgende Algorithmus?

```
#define N            5                  /* number of philosophers */
#define LEFT         (i+N–1)%N          /* number of i's left neighbor */
#define RIGHT        (i+1)%N            /* number of i's right neighbor */
#define THINKING     0                  /* philosopher is thinking */
#define HUNGRY       1                  /* philosopher is trying to get forks */
#define EATING       2                  /* philosopher is eating */

typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N–1 */
{
    while (TRUE) {                      /* repeat forever */
        think( );                       /* philosopher is thinking */
        take_forks(i);                  /* acquire two forks or block */
        eat( );                         /* yum-yum, spaghetti */
        put_forks(i);                   /* put both forks back on table */
    }
}

void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = HUNGRY;                  /* record fact that philosopher i is hungry */
    test(i);                            /* try to acquire 2 forks */
    up(&mutex);                         /* exit critical region */
    down(&s[i]);                        /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
    down(&mutex);                       /* enter critical region */
    state[i] = THINKING;                /* philosopher has finished eating */
    test(LEFT);                         /* see if left neighbor can now eat */
    test(RIGHT);                        /* see if right neighbor can now eat */
    up(&mutex);                         /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

2.3 Parallele Programmierung mit POSIX-Threads

In dieser Aufgabe werden wir ein minimales Programmierbeispiel mit POSIX-Threads durchlaufen.

a) Erstellen Sie eine Datei `minithread.c` und inkludieren Sie zunächst alle nötigen Header für Ein-/Ausgabe (`stdio.h`), Threads (`pthread.h`), sowie Systemfunktionen wie `exit` (`stdlib.h`). Vergessen Sie nicht, vor dem Inkludieren irgendeiner Header-Datei das Makro `_GNU_SOURCE` zu definieren. Welchen Zweck verfolgt dies (`man feature_test_macros`) und was drücken Sie als Programmierer damit aus?

b) Schreiben Sie eine Funktion `thread_func`, die genau der Signatur des `start_routine` Parameters von `pthread_create` folgt. Verwenden Sie die `printf`-Funktion, um das an `thread_func` übergebene Argument als String auf die Kommandozeile auszugeben. Verwenden Sie als Format dafür `Hello, my name is %s!\n`. Beenden Sie nach der Ausgabe den Thread mithilfe von `pthread_exit(NULL)`.

c) Definieren Sie in Ihrer `main`-Funktion ein Array `names` auf folgende Weise:

```
const char *names[5] = { "up", "you", "give", "gonna", "never" };
```

Erstellen Sie ein Array `tid` gleicher Länge des Typs `pthread_t`. Dieses Array wird die IDs der erstellten Threads enthalten. Legen Sie außerdem Platz für eine Zählervariable `i` und einen Rückgabewert `ret` (Typ `int`) an.

d) Rufen Sie `pthread_create` innerhalb einer Schleife so auf, dass der erstellte Thread jeweils einen Eintrag aus dem `names` Array als Argument erhält und die Ausführung mit der Funktion `thread_func` beginnt. Speichern Sie die IDs der erstellten Threads in Array `tid`. Wählen Sie den Parameter `attr` als `NULL`, sodass der Thread mit Standardattributen erstellt wird. Vergessen Sie nicht, den Rückgabewert der Funktion `pthread_create` zu prüfen. Durchlaufen Sie dabei das `names` Array rückwärts.

e) Jetzt, da alle Threads gestartet wurden, kann der Hauptthread die CPU freigeben, indem er `pthread_yield` aufruft. Fügen Sie diesen Aufruf hinter Ihrer Schleife in Ihren Code ein.

f) Abschließend sollte auf die Beendigung der erstellten Threads mithilfe von `pthread_join` gewartet werden. Rufen Sie dazu `pthread_join` innerhalb einer weiteren Schleife auf und verwenden Sie als Argument jeweils einen Eintrag des `tid`-Arrays. Vergessen Sie nicht, den Rückgabewert der Funktion zu prüfen.

2.4 Wechselseitiger Ausschluss

Um korrekte Berechnungen von nebenläufigen Prozessen zu gewährleisten, sind Maßnahmen zur Vermeidung von Konflikten zu ergreifen, die durch den konkurrierenden Zugriff auf gemeinsam benutzte Objekte entstehen. In den frühen Jahren der Informatik wurde versucht, dem Programmierer ein explizit nutzbares Werkzeug an die Hand zu geben, um parallele Prozesse in **kritischen Bereichen (critical region/section)** wechselseitig auszuschließen. Das Betriebssystem muss dazu eine geeignete Basis zur Verfügung stellen. In einem ersten Ansatz verwenden wir als Basis atomares Lesen und atomares Schreiben von Integer-Variablen. Eine Realisierungsmöglichkeit des **wechselseitigen Ausschlusses (mutual exclusion)** besteht nun darin, eine globale Datenstruktur zur Verfügung zu stellen, deren Wert von den Prozessen getestet wird. In Abhängigkeit von diesem Wert beginnt der jeweils testende Prozess mit der Ausführung seiner kritischen Phase oder er testet weiter. Dies ist eine Lösung des wechselseitigen Ausschlusses mit sogenanntem **aktivem Warten (busy waiting)**.

2.4.1 Begriffe

Klären Sie in diesem Zusammenhang die Bedeutung folgender Begriffe:

a) Race condition

b) Mutual exclusion

c) Critical region/section

d) Busy waiting

e) Deadlock

2.4.2 Synchronisationskonzepte in Programmiersprachen

Mit Semaphoren haben Sie ein Konzept zur Prozesssynchronisation kennengelernt. Beschreiben Sie kurz die zentralen Aspekten eines weiteren Synchronisationsprimitivs, welches Sie aus der Programmiersprache Java kennen.

2.4.3 Fragestellungen

a) Welche Probleme können bei der Realisierung des wechselseitigen Ausschlusses mit aktivem Warten auftreten?

b) Welchen Einfluss hat die jeweilige Rechnerarchitektur auf die Realisierung vom wechselseitigen Ausschluss? Denken Sie an Single-/Multiprozessor-Umgebungen.

c) Benötigt man zur Realisierung wiederum wechselseitig ausgeschlossene Operationen?

2.4.4 Erzeuger–Verbraucher-Problem, kritische Bereiche und Semaphore

Wir betrachten das folgende, stark vereinfachte[2] Beispiel für ein Erzeuger–Verbraucher-Problem: Eine Datei soll über ein Netzwerk auf einen Computer transferiert werden. Die Netzwerkkarte N des Computers empfängt blockweise Datenpakete und legt diese im Puffer (Buffer) B (Kapazität: n Datenblöcke) ab, von wo aus sie nach und nach entnommen und auf die Festplatte F gespeichert werden. Um wechselseitigen Ausschluss zu erreichen, sei folgender Lösungsversuch mit der Semaphore `wa` (wechselseitiger Ausschluss) als Pseudocode gegeben:

Beachten Sie: Die Pseudocode-Operation zum Deklarieren eines Semaphors mit Namen `name` mit Startwert `n` laute: `name(n);` Die Pseudocode-Operationen zum Aufrufen von P (down) und V (up) auf der Semaphore mit Namen `name` lauten: `down(name)` und `up(name)`

```
Deklaration:
wa(1);

Netzwerkkarte N:
while(true) {
    <empfange Datenblock>;
```

[2] Beachten Sie, dass dieses Beispiel aus didaktischen Gründen von reellen, hardwareseitigen Vorgängen abweicht und das Szenario auf ein einfacheres Problem reduziert.

```
    down(wa);
    <schreibe Datenblock in B>;
    up(wa);
}

Festplatte F:
while(true) {
    down(wa);
    <entnimm Datenblock aus B, falls vorhanden, sonst warte>;
    up(wa);
    <schreibe Datenblock auf Festplatte>;
}
```

2.4.5 Verklemmungen identifizieren

Laufen beide Prozesse verklemmungsfrei? Welche Probleme können auftreten?

2.4.6 Verkemmungsfreiheit mittels Semaphoren

Geben Sie eine verbesserte Version an, in der keine Probleme mehr auftreten, indem Sie zwei weitere Semaphoren geeignet deklarieren und geeignete Aufrufe von `down` und `up` einfügen.

2.4.7 Reihenfolge der Lock-Operationen der Semaphore

Welche Probleme treten auf, wenn Sie in Ihrer verbesserten Lösung die Reihenfolge der `down`-Operationen für `wa` und Ihrer beiden zusätzlichen Semaphoren permutieren?

3 Hausaufgaben

3.1 Synchronisation beim Ressourcenzugriff: Mutex (6 Punkte)

Wenn mehrere Prozesse oder Threads sich die Standardausgabe ("stdout") des Terminals teilen, können sich die Ausgaben überlagern und schwer lesbar werden. Effektiv stellt stdout eine gemeinsam genutzte Ressource dar, die die Prozesse oder Threads sinnvollerweise nur in kompletten Ausgabeblöcken nutzen sollen, die nicht unterbrochen werden. Wir wollen das an einem praktischen Beispiel veranschaulichen.

Nehmen Sie als Ausgangspunkt Ihr Programm `threadit` vom letzten Übungsblatt (inkl. der Listenverwaltung). Lassen Sie maximal 10 Threads zu. Die Anzahl der zu startenden Threads soll wieder auf der Kommandozeile (Parameter `-n`) übergeben werden. Erweitern Sie Ihre Threadfunktion so, dass diese die laufende Nummer des Threads k (beginnend bei 0) als Parameter übergeben bekommt und dann folgende Aktionen durchführt:

- die Datei *k*`.txt` mittels `open()` öffnet,
- den Inhalt der Datei in Einheiten von 64 Zeichen mittels `read()` liest und die laufende Nummer i der Einheit (beginnend bei 0) festhält,
- einen Ausgabepräfix erzeugt, der k und i der Ausgabe voranstellt, und zwar im Format `"[%02d] %03d"`, wobei erst k und dann i ausgegeben werden soll, gefolgt von einem Tab `\t`,
- dann die 64 Zeichen gefolgt von einem Newline ausgibt, wobei angenommen werden kann, dass alle Zeichen in der Datei darstellbar sind.

Verwenden Sie drei Mal den `write()`-Systemaufruf: 1. für den Präfix, 2. für die Einheit von 64 (oder weniger) Bytes aus der Datei und 3. für das Newline. (Das ist nicht effizient, dient aber der Übung.)

Wir stellen hierfür Beispieldateien bereit, die `0.txt`, `2.txt`, ..., `9.txt` benannt sind und sich im aktuellen Arbeitsverzeichnis von blatt05 befinden sollen.

Starten Sie das Programm wiederholt, und betrachen Sie die Ausgabe. Kann man in allen Fällen alle Zeilen gut lesen?

Stellen Sie nun zwei sinnvolle Gruppierungen bereit, die (nicht gleichzeitig) mit der Option `-l` (Line) oder `-f` (File) ausgewählt werden.

- `-l` fasst alle drei `write()`-Anweisungen in einem kritischen Abschnitt zusammen, so dass immer nur eine ganze Zeile von einem Thread atomar ausgegeben werden kann.
- `-f` fasst alle Ausgaben einer Datei zu einem kritischen Abschnitt zusammen, so dass immer nur der gesamte Dateiinhalt am Stück ausgegeben wird.

Schützen Sie diese kritischen Abschnitt gegen gleichzeitiges Betreten durch mehrere Threads, indem Sie ein Mutex benutzen.

Syntax: `syncem [-n <# Threads>] [-l|-f]`

Quellen: `syncem.c list.c list.h`

Executable: `syncem`

Beispielausgaben:

Ohne Synchronisation

```
$ ./syncem -n 10
[00] 000        0000000000000000000000000000000000000000000000000000000000000000000000000000
[01] 000        1111111111111111111111111111111111111111111111111111111111111111111111111111[00] 001
0000000000000000000000000000000000
[01] 001        11111111111111111111111111111111
[02] 000        22222222222222222222222222222222222222222222222222222222222222222222
[02] 001        22222222222222222222222222222222
[03] 000        33333333333333333333333333333333333333333333333333333333333333333333333333
[03] 001        333333333333333333333333333333333
[04] 000        [05] 000        4444444444444444444444444444444444444444444444444444444444444444444455555555555555555

[05] 001        [04] 001        5555555555555555555555555555555554444444444444444444444444444444444

[06] 000        66666666666666666666666666666666666666666666666666666666666666
[06] 001        6666666666666666666666666666666
[07] 000        77777777777777777777777777777777777777777777777777777777777777
[08] 000        [09] 000        88888888888888888888888888888888888888888888888888888888888888888888888888888888999999999999999999
[09] 001        99999999999999999999999999999999

[08] 001        888888888888888888888888888888888
[07] 001        7777777777777777777777777777777777
```

Synchronisation pro Zeile

```
$ ./syncem -n 10 -l
[00] 000        0000000000000000000000000000000000000000000000000000000000000000000000000000
[02] 000        22222222222222222222222222222222222222222222222222222222222222222222
[03] 000        33333333333333333333333333333333333333333333333333333333333333333333333333
[04] 000        44444444444444444444444444444444444444444444444444444444444444444444
[00] 001        00000000000000000000000000000000
[05] 000        55555555555555555555555555555555555555555555555555555555555555555555
[06] 000        66666666666666666666666666666666666666666666666666666666666666666666
[01] 000        11111111111111111111111111111111111111111111111111111111111111111111
[07] 000        77777777777777777777777777777777777777777777777777777777777777777777
[08] 000        88888888888888888888888888888888888888888888888888888888888888888888
[09] 000        99999999999999999999999999999999999999999999999999999999999999999999
[02] 001        22222222222222222222222222222222
[03] 001        333333333333333333333333333333333
[04] 001        44444444444444444444444444444444
[05] 001        55555555555555555555555555555555
[06] 001        66666666666666666666666666666666
[01] 001        11111111111111111111111111111111
[07] 001        77777777777777777777777777777777
[08] 001        88888888888888888888888888888888
[09] 001        99999999999999999999999999999999
```

Synchronisation pro Thread

```
$ ./syncem -n 10 -f
[02] 000        2222222222222222222222222222222222222222222222222222222222222222
[02] 001        22222222222222222222222222222222
[00] 000        0000000000000000000000000000000000000000000000000000000000000000
[00] 001        00000000000000000000000000000000
[01] 000        1111111111111111111111111111111111111111111111111111111111111111
[01] 001        11111111111111111111111111111111
[03] 000        3333333333333333333333333333333333333333333333333333333333333333
[03] 001        33333333333333333333333333333333
[04] 000        4444444444444444444444444444444444444444444444444444444444444444
[04] 001        44444444444444444444444444444444
[05] 000        5555555555555555555555555555555555555555555555555555555555555555
[05] 001        55555555555555555555555555555555
[06] 000        6666666666666666666666666666666666666666666666666666666666666666
[06] 001        66666666666666666666666666666666
[07] 000        7777777777777777777777777777777777777777777777777777777777777777
[07] 001        77777777777777777777777777777777
[08] 000        8888888888888888888888888888888888888888888888888888888888888888
[08] 001        88888888888888888888888888888888
[09] 000        9999999999999999999999999999999999999999999999999999999999999999
[09] 001        99999999999999999999999999999999
```

3.2 Synchronisation beim Ressourcenzugriff: Conditional Variables (4 Punkte)

Schreiben Sie ein Progamm `syncorder`, welches Ihr Programm `syncem` um eine weitere Option `-o` ergänzt und dafür sorgt, dass die Ausgaben der Threads in aufsteigender Reihenfolge der laufenden Thread-Nummern erfolgt:

- `-o` fasst alle `write()`-Anweisungen eines Threads in einem kritischen Abschnitt zusammen und garantiert die Ausgabe alle Threads in aufsteigender Reihenfolge nach Threadnummer k.

Ersetzen Sie hierzu den Systemaufruf `write()` durch eine von Ihnen geschriebene Funktion `write_buffer()`, die wie folgt definiert ist:

- `int write_buffer (long thread, char *buffer, int len);`

Die Funktion erhält als ersten Parameter die Laufnummer des Threads und danach (analog zu `write()`) einen Pointer auf den Puffer und die Pufferlänge. Die Funktion soll prüfen, ob der aufrufende Thread mit seiner Ausgabe an der Reihe ist. Falls ja, soll sie den Puffer mittels `write()` auf STDOUT ausgeben. Falls nicht, soll sie den Thread mit Hilfe einer Conditional Variable schlafen legen. Vergessen Sie nicht, ihn wieder aufzuwecken, wenn der aktuelle berechtigte Thread alle seine Ausgaben abgeschlossen hat.

Um zu erkennen, dass ein Thread mit seinen Ausgaben fertig ist, ruft er die Funktion `write_buffer` mit seiner Thread-Nummer, dem Pointer NULL und der Länge 0 auf.

Syntax: `syncorder [-n <# Threads>] [-l|-f|-o]`

Quellen: `syncorder.c list.c list.h`

Executable: `syncorder`

Beispielausgabe:

```
$ ./syncem -n 10 -o
[00] 000        00000000000000000000000000000000000000000000000000000000000000000000
[00] 001        00000000000000000000000000000000000
[01] 000        11111111111111111111111111111111111111111111111111111111111111111111
[01] 001        11111111111111111111111111111111
[02] 000        22222222222222222222222222222222222222222222222222222222222222222222
[02] 001        22222222222222222222222222222222
[03] 000        33333333333333333333333333333333333333333333333333333333333333333333
[03] 001        33333333333333333333333333333333
[04] 000        44444444444444444444444444444444444444444444444444444444444444444444
[04] 001        44444444444444444444444444444444
[05] 000        55555555555555555555555555555555555555555555555555555555555555555555
[05] 001        55555555555555555555555555555555
[06] 000        66666666666666666666666666666666666666666666666666666666666666666666
[06] 001        66666666666666666666666666666666
[07] 000        77777777777777777777777777777777777777777777777777777777777777777777
[07] 001        77777777777777777777777777777777
[08] 000        88888888888888888888888888888888888888888888888888888888888888888888
[08] 001        88888888888888888888888888888888
[09] 000        99999999999999999999999999999999999999999999999999999999999999999999
[09] 001        99999999999999999999999999999999
```