Bitte bearbeiten Sie nur eine der beiden Aufgaben 23–24. Welche Aufgabe Sie bearbeiten, ermitteln Sie wie folgt:

1. Finden Sie die kleinste Matrikelnummer `matr` in Ihrer Gruppe (falls Sie die Aufgaben zu zweit bearbeiten).
2. Wenn die Nummer ungerade ist, bearbeiten Sie Aufgabe 23.
3. Wenn die Nummer gerade ist, bearbeiten Sie Aufgabe 24.

Zur Abgabe:

- Falls Sie die Aufgabe nicht vollständig lösen konnten, ergänzen Sie im Kopf der Lösungsdatei (oder in einem separaten Dokument, das Sie mit abgeben) Hinweise dazu, welche Teile fehlen, und beschreiben Sie, welche Probleme Sie bei der Lösungsfindung hatten.
- Beachten Sie, dass der Code ausführlich dokumentiert sein soll.
- Erzeugen Sie ein Verzeichnis `sp-ss2013-`*`name`* (wobei *`name`* der Nachname eines der Gruppenmitglieder ist). In dieses Verzeichnis kopieren Sie Ihre gut dokumentierte Lösungsdatei. Die Datei muss im Kopf als Kommentar die Namen und Matrikelnummern aller Gruppenmitglieder enthalten. (Ein Template finden Sie im Aufgabenarchiv als `abgabe.c`.)
- Erstellen Sie im Verzeichnis außerdem eine Datei `abstract-`*`name`*`.txt`, in der Sie kurz einen Vorschlag für das Thema Ihres Projektvortrags aufschreiben. (Wenn Sie eine Gruppe sind, erstellen Sie jeweils eine Datei `abstract-`*`name`*`.txt` für jedes Mitglied.)
- Wechseln Sie dann in den Ordner, der `sp-ss2013-`*`name`* enthält, und erstellen Sie mit

  `tar czf sp-ss2013-`*`name`*`.tgz sp-ss2013-`*`name`*`/`

  ein `tgz`-Archiv, das Sie mir per Mail zuschicken (`h.g.esser@gmx.de`). Ich schicke Ihnen eine Eingangsbestätigung. Falls Sie keine erhalten sollten, fragen Sie bitte nach.
- Die **Deadline** für das Verschicken dieser Mail ist der 28.06.2013, 18:00 Uhr.
- Bereiten Sie einen ca. zehn Minuten dauernden Vortrag vor, in dem Sie Ihren Lösungsansatz beschreiben. Wenn Sie mit einem Partner zusammen gearbeitet haben, werden Sie gemeinsam einen ca. 15 Minuten dauernden Vortrag halten, bei dem Sie sich abwechseln. Nach jedem Vortrag ist ca. fünf Minuten Zeit für Fragen der anderen Teilnehmer (und von mir).

Auf der Webseite finden Sie das Aufgabenarchiv `sp-ss2013-ue12.tgz`. mit Unterordnern `23` und `24` für die jeweiligen Teilaufgaben; Sie benötigen nur den Unterordner, der zu Ihrer oben festgelegten Aufgabe gehört.

Für alle Aufgaben verwenden Sie den modularisierten `ohmthread`-Code, den Sie in Aufgabe 22 erzeugt haben.

Hinweis: Den Alternativtermin für den 09.07. legen wir heute fest.

23. Synchronisation mit Mutexen

Mutexe werden zum Locking und damit für die Synchronisation verwendet. Im Unterordner `23/` des Aufgabenarchivs finden Sie ein Beispiel `pthread-beispiel.c`, das die `pthread`-Bibliothek verwendet und zwei Threads startet, die *nicht* synchronisiert sind. Die beiden Threads verändern eine globale Variable (einer zählt sie hoch, der andere zählt sie runter), und am Ende sollte der Wert 0 herauskommen – das passiert aber nicht. Stattdessen erhalten Sie bei mehreren Programmausführungen unterschiedliche, von 0 abweichende Werte, weil durch Thread-Wechsel zu ungünstigen Zeiten „lost updates" auftreten, also Änderungen an der globalen Variable verloren gehen.

Solche Probleme löst man, indem man „kritische Bereiche" erkennt und durch einen Mutex schützt. Das Programm `pthread-mutex.c` verwendet einen solchen Mutex, um das Problem zu beheben: Es initialisiert die Variable `mutex` mit `pthread_mutex_init()` und schützt dann die Zugriffe auf die globale Variable, indem es vor dem Zugriff mit `pthread_mutex_lock()` eine Sperre errichtet, die es nach dem Zugriff mit `pthread_mutex_unlock()` wieder aufhebt. Dadurch arbeitet diese Version korrekt.

Hat ein Thread erfolgreich einen Mutex „gelockt" und versucht dann (nach einem Thread-Wechsel) ein zweiter Thread, *denselben* Mutex zu locken, wird er blockiert, bis der erste Thread den Mutex wieder freigibt. Für jeden Mutex ist also eine Warteschlange notwendig. Gibt es in der Warteschlange mehr als einen Thread, der auf die Freigabe des Mutex wartet, wird nur der erste Thread aus dieser Warteschlange aufgeweckt (und er kann dann weiter arbeiten).

Die Datei `ohmthread-beispiel.c` enthält eine auf die `ohmthread`-Funktionen portierte Variante von `pthread-beispiel.c` – damit hier der Fehler auftritt, müssen Sie den Scheduler mit einem kleinen Quantum arbeiten lassen: Auf dem Testrechner „funktionierte" es mit

```
#define QUANTUM   100     // Quantum in mikrosec. fuer Scheduler
```

Sie müssen eventuell mit dem Wert experimentieren. Das Ziel ist, dass während der Programmausführung zahlreiche Thread-Wechsel stattfinden.

Wichtig bei Mutexen ist, dass der Zugriff auf die Mutex-Variable nicht unterbrechbar ist. Betriebssysteme stellen dazu Mechanismen bereit, die einen Prozesswechsel unterbinden. Dieses Verhalten können Sie simulieren, indem Sie den Scheduler vorübergehend deaktivieren (mehr dazu weiter unten).

Mutexe führen Sie nun im `ohmthread`-Code ein; Sie bilden dazu die `pthread`-Mutex-Funktionen

```
pthread_mutex_init()    - create a mutex
pthread_mutex_lock()    - lock a mutex
pthread_mutex_trylock() - attempt to lock a mutex without blocking
pthread_mutex_unlock()  - unlock a mutex
```

nach.

1. Definieren Sie in der Header-Datei mit `typedef` einen neuen Mutex-Typ: Ein `char`-Wert reicht für den Mutex-Wert aus, außerdem müssen Sie für den Mutex eine Warteschlange verwalten – dafür brauchen Sie nur einen Integer-Wert `queue` (Warteschlange), der die Thread-ID des ersten wartenden Threads enthält. Sie können also

   ```
   typedef struct {
     char m;
     int queue;
   } ohmthread_mutex_t;
   ```

   schreiben.

2. Um den Scheduler vorübergehend aus- (und wieder ein-) schalten zu können, definieren Sie eine globale Variable `SCHEDULER_ACTIVE`, die Sie bei der Initialisierung des Threading auf 1 (true) setzen. Schreiben Sie zwei Makros `DISABLE` und `ENABLE`, welche diese Variable auf 0 bzw. 1 setzen. Am Anfang des Schedulers (also Ihres Alarm-Handlers) prüfen Sie den Wert von `SCHEDULER_ACTIVE`: Wenn dieser 0 ist, kehren Sie aus dem Handler sofort zurück – dann finden keine Thread-Wechsel mehr statt.

Testen Sie, ob das neue Feature funktioniert, indem Sie in einer Thread-Funktion `DISABLE;` und nach einer (nicht unendlichen) Schleife `ENABLE;` aufrufen – Sie müssen die Definitionen von `DISABLE` und `ENABLE` auch in die Header-Datei eintragen und dort außerdem

```
extern int SCHEDULER_ACTIVE;
```

deklarieren, damit Ihr Programm diese Variablen bzw. Makros kennt.

3. Implementieren Sie eine Funktion

   ```
   void ohmthread_mutex_init (ohmthread_mutex_t *mutex);
   ```

   welche in einer (als Pointer) übergebenen Mutex-Variable den Wert auf 0 und den Warteschlangeneintrag auf 0 setzt. (0 soll bedeuten: Es gibt noch keine Einträge in der Warteschlange.) Die entsprechende `pthread`-Funktion ist etwas komplexer, aber die zusätzlichen Features benötigen wir hier nicht.

4. Implementieren Sie die Funktionen

   ```
   void ohmthread_mutex_lock (ohmthread_mutex_t *mutex);
   void ohmthread_mutex_unlock (ohmthread_mutex_t *mutex);
   ```

   wie folgt:

   Die Lock-Funktion schaltet den Scheduler aus und liest dann den in der Mutex-Variable gespeicherten Wert. Wenn dieser 0 ist (was bedeutet: nicht gelockt), setzt sie den Wert auf 1 (sperrt also diesen Mutex), aktiviert den Scheduler wieder und kehrt mit `return` zurück.

   Ist der Wert hingegen 1, muss der aktuelle Thread blockiert und in die Warteschlange für diesen Mutex eingetragen werden. Das Blockieren ist einfach: Dazu definieren Sie in der Header-Datei einen neuen Thread-Zustand, z. B. mit

   ```
   #define THREAD_WAITMUTEX     4
   ```

   und setzen diesen Wert im TCB; mit `ENABLE; raise(SIGALRM);` können Sie später (wie in der Funktion `ohmthread_join`) dafür sorgen, dass ein anderer Thread aktiviert wird.

   Für das Eintragen in die Warteschlange benötigen Sie eine Schleife: Wenn `mutex->queue` gleich 0 ist, gibt es noch keinen Eintrag in der Warteschlange, und Sie können einfach `mutex->queue` auf `tid` setzen. Anderenfalls schauen Sie in den TCB mit der Nummer `mutex->queue` und dort in das Feld `next`: Das haben wir bisher nicht verwendet, es soll beim Einsatz von Warteschlangen immer die Thread-ID des nächsten Threads in der Warteschlange enthalten. Sie springen also über `t = threads[t].next`; jeweils zum nächsten Thread in der Warteschlange. Wenn Sie dabei einen Thread finden, dessen TCB im `next`-Feld den Wert 0 hat, sind Sie am Ende angekommen und hängen den aktuellen Thread hinten an (`threads[t].next = tid; threads[tid].next = 0;`).

   Die Unlock-Funktion ist einfacher gestrickt. Hier gibt es zwei Fälle: Wenn die Warteschlange leer ist, setzen Sie den Mutex-Wert auf 0 zurück und sind fertig. Gibt es mindestens einen Eintrag in der Warteschlange, lassen Sie den Mutex-Wert unverändert (auf 1), wecken den ersten Thread in der Warteschlange (d. h.: `state`-Feld im TCB auf `THREAD_ACTIVE` setzen) und entfernen ihn aus der Warteschlange. Auch in dieser Funktion halten Sie am Anfang mit `DISABLE` den Scheduler an und aktivieren ihn vor jedem `return` wieder mit `ENABLE`.

5. **Bonusaufgabe:** Lesen Sie die Manpage zur Funktion `pthread_mutex_trylock()` durch – mit dieser Funktion kann ein (`pthread`-) Thread *versuchen*, einen Mutex zu sperren. Die Funktion kehrt in jedem Fall zurück (blockiert also nie), und der Rückgabewert verrät, ob der Aufruf erfolgreich war (0) oder nicht (`EBUSY`=16, definiert in `/usr/include/asm-generic/errno-base.h`). Programme, die diese Funktion verwenden, müssen also auf jeden Fall den Rückgabewert überprüfen und dürfen nur dann den „kritischen Bereich“ betreten, wenn der Aufruf erfolgreich war.

   Implementieren Sie eine `ohmthread`-Variante dieser Funktion. Sie können dabei im Wesentlichen den Code von `ohmthread_lock()` verwenden und müssen nur den Code fürs Blockieren durch das Erzeugen eines Fehlercodes ersetzen.

24. Signale zwischen Threads

In dieser Aufgabe bilden Sie die Prozess-internen Signal-Handler nach. Ein Thread soll in der Lage sein, Thread-eigene Signal-Handler zu installieren, andere Threads sollen dann mit der Funktion `ohmthread_kill()` ein Signal an einen solchen Thread schicken können. Die zu implementierenden Funktionen sind

```
ohmthread_signal (int signo, void... handler)
```

```
ohmthread_kill (pthread_t thread, int signo)
```

Das Verhalten soll dabei wie folgt sein:

- Das Eintragen eines Signal-Handlers hat zunächst keine Auswirkung; im Thread-Control-Block wird nur für die Signalnummer die Adresse der Handler-Funktion vermerkt.
- Wenn ein Thread `ohmthread_kill (thread, signo)` aufruft, soll im Thread-Control-Block des Threads `thread` vermerkt werden, dass dieses Signal gesendet wurde.
- Die wichtigste Änderung findet im Scheduler statt: Jedesmal wenn dieser einen neuen Thread auswählt, prüft er, ob für diesen unbehandelte Signale vorliegen. Wenn ja, springt er nicht wie gewohnt in die Thread-Funktion, sondern ruft den Signal-Handler auf. Nach Abarbeiten des Signal-Handlers prüfen Sie, ob noch weitere Signale zu verarbeiten sind.

Im Detail:

1. Definieren Sie in `ohmthread.c` einen neuen Signal-Handler-Typ

   ```
   typedef void (*sighandler_t) (int);
   ```

   (dieser Typ beschreibt eine Funktion, die ein `int`-Argument akzeptiert) und erweitern Sie die TCB-Struktur `tcb_t` um zwei Einträge `sighandler_t sighandlers[32]` und `unsigned int sigpending`. Der erste Eintrag speichert die Adressen der Signal-Handler, der zweite ist ein 32 Bit großer Integer-Wert, dessen einzelne *Bits* den aktuellen Zustand für jedes potenzielle Signal enthalten (0 = Signal nicht empfangen; 1 = Signal empfangen).

2. Schreiben Sie eine Funktion

   ```
   sighandler_t ohmthread_signal (int sig, sighandler_t func);
   ```

   die eine Signalnummer `sig` und einen Signal-Handler `func` als Argumente akzeptiert. Sie soll in den eigenen TCB in `sighandlers[sig]` die Funktion `func` eintragen. (Die Syntax ist an die „normale“ Funktion `signal()` angelehnt.)

3. Schreiben Sie eine Funktion

   ```
   int ohmthread_kill (pthread_t thr, int sig);
   ```

   die prüft, ob `thr` für einen gültigen Thread steht – wenn ja, soll sie in dessen TCB in `sigpending` das Bit Nummer `sig` setzen. Um in einer normalen Integer-Variable `x` das Bit Nummer `i` zu setzen, können Sie den Befehl

   ```
   x = x | (1 << i);
   ```

   verwenden. (<< ist der Links-Shift-Operator, `1<<i = 2^i`, also z. B. `1<<5 = 2^5 = 32`).

   Den Befehl können Sie noch verkürzen, indem Sie statt `x = x |` … die Variante `x |=` … verwenden – dabei müssen Sie `x` nicht zweimal nennen.

4. Nehmen Sie die Signaturen der beiden neuen `ohm_thread`-Funktionen in die Header-Datei `ohmthread.h` auf.

5. Prüfen Sie, dass diese Funktionen korrekt arbeiten, indem Sie ein Programm schreiben, das zwei Threads erzeugt. Im ersten Thread läuft eine Funktion, die zunächst mit

   ```
   int i;
   for (i=0; i<50; i++) usleep (10000);
   ```

   ein wenig wartet und dann mit `ohmthread_kill()` das Signal Nummer 9 an den zweiten

Thread schickt. Danach tut sie in einer Endlosschleife nichts mehr. Der zweite Thread soll in einer Endlosschleife immer wieder den Wert von `sigpending` in seinem TCB ausgeben. Zu Beginn sollte hier immer 0 erscheinen; nachdem der erste Thread das Signal geschickt hat, sollte dieser Wert auf 512 (=2^9) umspringen. Das Programm wird sich nur kompilieren lassen, wenn Sie die Definition von `tcb_t` und die beiden Zeilen

```
#define MAXTHREADS 20        // bis zu 20 Threads
extern tcb_t threads[];
```

in Ihr Programm übernehmen. (Diese Typdeklaration, die Konstante und die Variable sind ja nicht Teil der Header-Datei, weil es eigentlich nicht vorgesehen ist, dass Threads auf ihren TCB zugreifen.) Wenn Sie Ihr Programm `24-threads.c` nennen, können Sie das Makefile im Unterordner `24/` des Aufgabenarchivs verwenden.

6. Jetzt kommt der schwere Teil, in dem Sie den Scheduler anpassen.

   Im Alarm-Handler (also im Scheduler) gibt es eine Stelle, an der Sie die Bedindung `(tid== oldtid)` überprüfen (um zu sehen, ob überhaupt ein Thread-Wechsel stattfindet); falls die Bedingung erfüllt ist, wird an dieser Stelle der Alarm-Handler verlassen. Den neuen Code fügen Sie *vor* diesem Test ein, denn ein Thread kann auch ein Signal an sich selbst schicken.

   Testen Sie nun zunächst, ob `threads[tid].sigpending == 0` ist: Falls ja, gibt es hier nichts zu tun, denn das bedeutet, dass alle Bits in `sigpending` gleich 0 sind (es also keine wartenden Signale gibt).

   Falls der Wert `!= 0` ist, testen Sie die einzelnen Bits: Wenn `sigpending & (1<<i) != 0` gilt, ist das `i`-te Bit in `sigpending` gesetzt. Dann prüfen Sie, ob der Thread einen Signal-Handler für dieses Signal registriert hat (also `sighandlers[i] != NULL`). Ist das der Fall, setzen Sie das Bit auf 0 zurück (das geht mit `sigpending &= ~(1<<sig)`, denn `1<<sig` hat das `sig`-te Bit gesetzt, und ~ kippt alle Bits) und rufen den Signal-Handler auf.

   Tipp: Wenn `sig` die Signalnummer ist, erledigen Sie den Aufruf mit

   ```
   threads[tid].sighandlers[sig](sig)
   ```

   denn `sighandlers[sig]` ist der Signal-Handler, und als Argument übergeben Sie ihm die Signal-Nummer. (Dadurch können Sie später einen Signal-Handler für mehrere Signale nutzen.)

7. Um den neuen Code zu testen, verwenden Sie folgenden Signal-Handler:

   ```
   void sigfunction (int sig) {
     printf ("[%d] in sigfunction (sig=%d)\n", tid, sig);
   };
   ```

   und tragen ihn in der Thread-Funktion des zweiten Threads mit `ohmthread_signal (9, sigfunction);` ein.

   In der ersten Thread-Funktion ergänzen Sie nach dem Aufruf `ohmthread_kill(3,9)` noch die Zeile

   ```
   printf ("[%d] Signal verschickt\n", tid);
   ```

   Wenn Sie Ihr Programm nun testen, sollten Sie in der Ausgabe die Zeilen

   ```
   [2] Signal verschickt
   [3] in sigfunction (sig=9)
   ```

   finden.

8. **Bonusaufgabe:** Wenn ein Thread sich selbst ein Signal schickt, wird dieses nicht sofort zugestellt, der Thread läuft zunächst normal hinter dem `ohmthread_kill()`-Aufruf weiter. Woran liegt das? Beheben Sie das Problem.