20. Verbesserungen der Thread-Funktionen

Auf der Webseite finden Sie das Aufgaben-Archiv `sp-ss2013-ue11.tgz`.

In dieser Aufgabe geht es darum, noch näher an die Funktionalität von `pthread_create()` heranzukommen. Dazu müssen wir aus dem Hauptprogramm auch einen Thread machen, denn in der bisherigen Lösung laufen nur noch die Threads, sobald der Alarm-Handler zum ersten Mal gestartet wurde.

Wir verlagern die Initialisierung des Signal-Handlers und des Timers in die `create_thread()`-Funktion: Diese prüft bei jedem Aufruf, ob es bereits Threads gibt – wenn nicht, wird das Threading initialisiert und für die aufrufende Funktion ein (zusätzlicher) erster Thread angelegt. Außerdem soll sich das Threading-System selbst um die Vergabe von Thread-IDs kümmern, so dass man beim Aufruf von `create_thread()` keine ID mehr angeben muss.

a) Schreiben Sie die folgenden zwei Funktionen:

`void initialize_threading()` – enthält (zunächst) die Initialisierung des Signal-Handlers und des Timers; entfernen Sie dann die entsprechenden Kommandos aus der `main()`-Funktion.

`int get_new_thread_id()` – erzeugt fortlaufende Thread-IDs (ab 0) und gibt diese zurück.

Passen Sie dann `create_thread()` so an, dass diese Funktion ohne eine Thread-ID aufgerufen wird; die Funktion nutzt dann `get_new_thread_id()`, um die zu verwendende ID festzulegen. Außerdem soll beim allerersten Aufruf von `create_thread()` am Ende der Thread-Erzeugung `initialize_threading()` aufgerufen werden. Sie können für den Test (Threading aktiv/nicht aktiv?) die bereits vorhandene Variable `thread_count` nutzen. Geben Sie der Funktion stattdessen ein erstes Argument vom Typ `pthread_t*`, über das sie die Thread-ID zurückgeben kann. (Unter Linux ist `pthread_t` als `unsigned long int` definiert.) Die Signatur der Funktion sieht danach wie folgt aus:

```
void create_thread (pthread_t *thr, void *(*function)(void *));
```

Passen Sie schließlich in `main()` die Aufrufe von `create_thread()` an (Sie müssen nun auch für jeden Thread eine Thread-Variable vom Typ `pthread_t` deklarieren und verwenden) und überprüfen Sie, dass Ihr Programm noch korrekt arbeitet.

b) *Thread Control Blocks*: Bisher verwenden wir verschiedene Arrays, um Informationen über Threads zu speichern. Diese sollen nun alle in einer Datenstruktur namens Thread Control Block zusammengefasst werden. Definieren Sie den Struct-Typ `tcb_t` und die Thread-Liste `threads` wie folgt:

```
typedef struct tcb {
  short    used;   // Wird dieser Eintrag benutzt?  (0/1)
  short    prev;   // vorheriger Thread (in Liste)
  short    next;   // nächster Thread (in Liste)
  ...  // TO DO: weitere Felder
} tcb_t;

tcb_t threads[MAXTHREADS];
```

Die beiden Elemente `prev` und `next` dienen später dazu, doppelt verkettete Listen (etwa: Liste der schlafenden Threads, Liste der ausführbereiten Threads) zu verwalten. Klassisch würde man hier Zeiger einsetzen, wir tragen stattdessen einen Index ins Array ein. `threads[0]` soll nicht verwendet werden, so dass die Informationen des ersten Threads in `threads[1]` gespeichert werden.

Passen Sie dann alle Zugriffe auf Thread-Informationen (z. B. die Arrays `stacks[]`, `start[]` usw.) an, so dass nur noch die neuen TCBs verwendet werden. Beim Initialisieren eines TCBs setzen Sie `prev` und `next` auf 0 (mit der vorläufigen Bedeutung: „es gibt keinen vorherigen bzw. nächsten Thread“).

c) *Argumente für die Thread-Funktionen*: Bisher können Sie den Thread-Funktionen keine Argumente übergeben. In dieser Aufgabe ändern Sie das, die Funktion `create_thread` soll darum ein weiteres Argument `void *arg` akzeptieren, das auf das Argument (bzw. die Argumente) zeigt. Damit sieht die Signatur nun wie folgt aus:

```
void create_thread (pthread_t *thr, void *(*function)(void *), void *arg);
```

Wie wir schon gesehen haben, werden bei Funktionsaufrufen die ersten sechs Argumente nicht über den Stack, sondern in Registern übergeben – das ist die „x64 calling convention“. Für Thread-Funktionen wird immer nur ein Argument vom Typ `void *` übergeben (hinter dem sich dann mehrere echte Argumente verbergen können), und die aufgerufene Funktion erwartet dieses erste Argument im Register `rdi`, siehe http://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/.

Um dieses Verhalten für unsere Threads zu aktivieren, ergänzen Sie den TCB um ein weiteres Element `rdi`, das Sie beim Thread-Erzeugen (an derselben Stelle, an der Sie `rsp` und und `rbp` setzen) mit `(unsigned long)arg` füllen. Wenn Sie dann zum ersten Mal im Scheduler (also im Alarm-Handler) zum neuen Thread wechseln, kopieren Sie `rdi` an die richtige Stelle in `uc->uc_mcontext.gregs`.

Benennen Sie `create_thread()` in `ohmthread_create()` um; wir werden in Analogie zu den `pthread_*()`-Funktionen im Folgenden alle Thread-relevanten Funktionen `ohmthread_*()` nennen.

Testen Sie den neuen Mechanismus der Parameter-Übergabe, indem Sie die Funktion `test04()`:

```
void *test04 (void *arg) {
  int value = (int)*(int*)arg;
  for (;;) {
    printf ("Argument-Test, tid = %d, arg = %d\n", tid, value);
    usleep (10000);
  };
}
```

ergänzen und aus dem Hauptprogramm zusätzlich mit

```
int arg1 = 42;  int arg2 = 129;
pthread_t t6, t7;
ohmthread_create (&t6, test04, &arg1);
ohmthread_create (&t7, test04, &arg2);
```

zwei weitere Threads starten, welche `test04()` starten und die Aufrufargumente 42 und 129 übergeben.

d) `ohmthread_exit()`: POSIX-Threads kann man mit einem Aufruf von `pthread_exit()` explizit beenden oder einfach mit `return` verlassen; im zweiten Fall wird automatisch ebenfalls `pthread_exit()` aufgerufen. In dieser Teilaufgabe implementieren Sie eine `ohmthread_exit()`-Funktion und manipulieren den Stack eines neuen Threads so, dass beim Verlassen mit `return` ebenfalls `ohmthread_exit()` aufgerufen wird. Damit können Sie Thread-Programme genau wie mit der `pthread`-Bibliothek schreiben.

`pthread_exit()` akzeptiert einen Rückgabewert (vom Typ `void *`), den andere Thread-Funktionen mit `pthread_join()` auswerten können, und auch ein mit `return` zurück gegebener Wert wird hier berücksichtigt. Dasselbe soll für `ohmthread_exit()` bzw. den `return`-Aufruf in einer Thread-Funktion gelten. Die Funktion hat also folgende Signatur:

```
void ohmthread_exit (void *value_ptr);
```

Um diese Funktion zu implementieren, sind folgende Schritte nötig:

– Erweitern Sie den TCB um zwei Felder `short state` (Zustand des Threads) und `void *retval` (Rückgabewert des beendeten Threads). Definieren Sie die folgenden drei Konstanten für die Thread-Zustände:

```
#define THREAD_NO_SUCH_STATE 0
#define THREAD_ACTIVE        1
#define THREAD_EXIT          2
```

– Bei der Initialisierung eines Threads setzen Sie dessen `state` auf `THREAD_ACTIVE`; beim Aufruf von `ohmthread_exit()` wird der Wert auf `THREAD_EXIT` gesetzt. Wir behalten zunächst den TCB für diesen Thread; in diesem Zustand ist er mit einem Zombie-Prozess vergleichbar. (Zur Erinnerung: Ein Zombie-Prozess ist ein beendeter Prozess, dessen Rückgabewert noch nicht vom Vaterprozess ausgelesen wurde.)

– `ohmthread_exit()` trägt den Rückgabewert in den TCB ein und ruft als letzten Punkt mit `raise(SIGALRM)` den Alarm-Handler (unseren Scheduler) auf, damit ein anderer Thread ausgewählt wird.

– Da das Programm nie wieder in einen beendeten Thread springen soll, müssen Sie im Alarm-Handler beim Bestimmen der nächsten Thread-ID noch prüfen, ob der jeweilige Thread im Zustand `THREAD_ACTIVE` ist – wenn nicht, geht die Suche weiter. Das kann (im Moment) im Extremfall dazu führen, dass alle Threads mit `ohmthread_exit()` beendet wurden: Dann findet der Scheduler keinen auszuführenden Thread und läuft in einer Endlosschleife weiter. Später werden wir dieses Verhalten noch überarbeiten. Wenn der Scheduler feststellt, dass er denselben Thread auswählt, der gerade unterbrochen wurde, dann soll er einfach mit `return` zurück kehren; dazu muss er sich die beim Aufruf gültige Thread-ID (z. B. in einer lokalen Variable `oldtid`) merken.

Danach fehlt noch die Behandlung des `return`-Aufrufs in einer Thread-Funktion. Hierzu ist der Stack zu manipulieren: Auf dem Stack muss die Rücksprungadresse liegen, an der es weiter geht, nachdem die laufende Thread-Funktion mit `return` verlassen wurde. Sie ist vom Typ `unsigned long`, also 8 Byte groß: Erniedrigen Sie beim Erzeugen des Threads zunächst den Stack Pointer und den Base Pointer (die Werte sind ja identisch, sie stehen in den Feldern `rsp` und `rbp`) um 8 Bytes und schreiben Sie dann die Adresse `&ohmthread_exit` an dieser Stelle in den Speicher. Dafür können Sie z. B. das aus Aufgabe 18 bekannte Makro

```
#define write_memory(addr, value) (*(unsigned long *)(addr) = \
  ((unsigned long)value))
```

verwenden.

Da die Thread-Funktionen den Rückgabetyp `void *` haben (der auch dem Argumenttyp von `ohmthread_exit()` entspricht), können Sie hier keine Zahlen übergeben. Das funktioniert bei den POSIX-Threads (`pthread`) genauso. Für Tests können Sie einen Rückgabewert nach `(void *)` casten.

Bauen Sie probeweise in `ohmthread_exit()` eine Debug-Ausgabe ein, die den Rückgabewert ausgibt, z. B.

```
void ohmthread_exit (void *retval) {
  ...
  printf ("DEBUG: ohmthread_exit (%ld)\n", (unsigned long)retval);
  ...
```

und testen Sie, ob Werte bei Verlassen eines Threads mit explizitem Aufruf von `pthread_exit()` sowie mit dem impliziten Aufruf (über `return`) korrekt angezeigt werden. Sie werden feststellen, dass es beim Verlassen über `return` nicht funktioniert.

Achten Sie auch darauf, dass Sie irgendwann mit `free()` den Stack des Threads freigeben müssen.

e) *Trick für Thread-Ende mit* `return`: Wird eine Thread-Funktion mit `return (void*)retval;` verlassen, wird sie zwar korrekt beendet, aber der Rückgabewert geht verloren. Das liegt daran, dass der Wert im falschen Register liegt: Die Funktion `ohmthread_exit()` erwartet den Wert im Register `rdi`, dieses wird aber nicht belegt, weil die Funktion nicht auf klassische Weise aufgerufen wird – über den Stack-Trick springt das Programm ja direkt an den Anfang dieser Funktion. Zu diesem Zeitpunkt liegt der Rückgabewert im Register `rax`, was zunächst nicht weiter hilft.

Eine Lösung verwendet eine kleine Assembler-Quellcodedatei, die nur die folgenden Zeilen enthält:

```
.globl call_ohmthread_exit
.text
call_ohmthread_exit:
  mov %rax, %rdi
  call ohmthread_exit
```

Diese stellt eine kleine Funktion `call_ohmthread_exit` zur Verfügung, die einfach den Inhalt von `rax` nach `rdi` kopiert und dann die Funktion `ohmtread_exit` (aus dem C-Programm) aufruft. Die beiden Quelldateien müssen separat übersetzt werden, was sich mit folgendem Makefile erreichen lässt:

```
20d-threads-06: 20d-threads-06.c call_ohmthread_exit.o
      gcc 20d-threads-06.c -lpthread call_ohmthread_exit.o -o $@
%.o: %.s
      gcc -c $<
all: 20d-threads-06
```

Wenn Sie `make` eingeben, wird alles übersetzt. (Die Einrückungen in der zweiten und vierten Zeile sind Tabulatoren.) Dann muss nur noch in der Funktion `ohmthread_create()` anstelle von `&ohmthread_exit` die Adresse `&call_ohmthread_exit` auf den Stack geschrieben werden – danach funktioniert alles wie gewünscht. Sie finden die beiden Dateien `Makefile` und `call_ohmthread_exit.s` im Quellcode-Archiv `sp-ss2013-ue11.tgz`.

Diese Lösung hätten Sie auch selbst finden können, aber nur mit erhöhtem Aufwand: Wenn Sie z. B. in die Funktion `test04()` am Ende den Aufruf `ohmthread_exit((void*)42)` einbauen, das Programm übersetzen und den für `test04()` generierten Assembler-Code anschauen, finden Sie am Ende die Sequenz

```
400c15:       bf 2a 00 00 00          mov    $0x2a,%edi
400c1a:       e8 e1 fb ff ff          callq  400800 <ohmthread_exit>
400c1f:       c9                      leaveq
400c20:       c3                      retq
```

Dabei ist `0x2a` = 42, also das Argument, und der Befehl unmittelbar vor dem Aufruf `call ohmtread_exit` kopiert ihn ins Register `edi`.

`ohmtread_exit()` erwartet das Argument im Register `rdi` (was bereits aus Aufgabe **c)** bekannt ist). Daraus ergibt sich die Kopieroperation `mov %rax, %rdi` in der Assembler-Datei.

f) *Hauptprogramm als separater Thread*: Es fehlt noch die Konvertierung des Hauptprogramms in einen eigenen Thread. Das ist aber auch kein großes Problem: Wenn der Alarm-Handler zum ersten Mal aufgerufen wird, kommt er ja gerade aus dem Hauptprogramm – dann sichert er einfach (vor den sonstigen Aktivitäten) den aktuellen Zustand im bisher nicht verwendeten TCB Nummer 1, die erzeugten Threads verwenden TCBs 2, 3 usw. Ändern Sie dazu die erste regulär (von `ohmthread_create()`) vergebene Thread-ID auf 2, damit die Nr. 1 frei bleibt.

Damit die Änderungen funktionieren, müssen Sie im Timer-Handler die Auswahl des nächsten Threads anpassen; Sie können z. B. durch das ganze TCB-Array (also von 1 bis `MAXTHREADS-1`) laufen und dann wieder von vorne beginnen:

```
for (;;) {
  // Nur bereiten Thread auswaehlen
  tid = (tid+1) % MAXTHREADS;    // naechste Thread-ID bestimmen
  if (tid == 0) tid = 1;         // tid 0: nicht verwendet
  if (threads[tid].state == THREAD_ACTIVE) break;    // gefunden!
}
```

Testen Sie, ob Ihr Hauptprogramm nun auch weiterläuft; bauen Sie dazu z. B. die folgende Schleife in `main()` ein:

```
for (;;) {
  printf ("main() lebt auch noch, tid = %d\n", tid);
  usleep (10000);
}
```

Die Meldungen von `main()` sollten zunächst mit `tid = -1` erfolgen (d. h., das Thread-System wurde noch nicht initialisiert), nach dem ersten Timer-Interrupt ändert sich die Ausgabe dann auf `tid = 1` (die Thread-ID des Hauptprogramms).

Ausblick: Im nächsten Teil der Projektaufgaben ergänzen wir eine `ohmthread_join()`-Funktion und verlagern den Thread-Code in eine eigene Bibliothek.