Vorbemerkung: Die Programme aus der letzten Übung stehen jetzt in 64-Bit-Versionen zur Verfügung, die ich erfolgreich auf zwei unterschiedlichen 64-Bit-Linux-Systemen übersetzen konnte; mit etwas Glück laufen sie auch auf den PCs im Übungsraum. Sie finden auf der Webseite ein zusätzliches Download-Paket mit den 64-Bit-Varianten der Programme (Link `ue09-64-bit-tgz`).

Falls die Programme auf den Übungsrechnern nicht laufen, können Sie einen meiner Server (auch ein 64-Bit-System) verwenden, das Passwort gebe ich bei Bedarf während der Übung bekannt. Sie loggen sich auf der Maschine mit dem Kommando `ssh syspro@46.38.243.41` ein (der Rechner hat keinen Namen) und legen dann mit `mkdir name` ein Unterverzeichnis an, in dem Sie arbeiten. In `/home/syspro/ue10/` finden Sie bereits die Dateien für diese Übung, die Sie dann mit `cp -r ue09 name/` in Ihren eigenen Unterordner kopieren.

18. Stack des Signal-Handlers

Im Aufgaben-Archiv `sp-ss2013-ue10.tgz` finden Sie die Datei `18-show-signal-stack.c`.

Wir wollen langfristig einen Signal-Handler als Scheduler nutzen, um zwischen den Threads (und ihren Stacks) hin und her zu schalten. Das Problem dabei ist, dass wir herausfinden müssen, wie wir aus dem Signal-Handler heraus auf die Daten des unterbrochenen Threads zugreifen können.

Betrachten Sie dazu das Programm `18-show-signal-stack.c`, das in `main()` zunächst einen Signal-Handler und einen Timer installiert und dann in einer Endlosschleife nichts mehr tut. Wenn Sie sich mit `objdump` den erzeugten Assembler-Code anzeigen lassen, finden Sie diese Endlosschleife in Form eines `jmp`-Befehls, der auf die Adresse dieses Befehls (auf sich selbst) zurück springt; auf einer Testmaschine war das die Adresse `0x40080f`, bei Ihnen kann es eine andere sein:

```
$ objdump -S a.out
00000000004007a9 <main>:
  4007a9:       55                      push   %rbp
  4007aa:       48 89 e5                mov    %rsp,%rbp
  [...]
  40080a:       e8 71 fd ff ff          callq  400580 <alarm@plt>
  40080f:       eb fe                   jmp    40080f <main+0x66>
  400811:       90                      nop
```

Hervorgehoben ist die Zeile mit dem `jmp`-Befehl.

a) Übersetzen Sie mit

```
gcc 18-show-signal-stack.c
```

das Programm und suchen Sie mit

```
objdump -S a.out
```

nach der Adresse, an der die Endlosschleife auf Ihrem Rechner „landet“. Jetzt können Sie das Programm auch starten. Es springt nach einer Sekunde Wartezeit in den Signal-Handler für das `SIGALRM`-Signal und gibt dann die ersten 50 64-bittigen Werte aus, die auf dem Stack liegen. Hier sollten Sie die im vorherigen Schritt notierte Adresse (im Beispiel `0x40080f`) wieder finden: Sie liegt auf dem Stack, denn der Signal-Handler merkt sich auf dem Stack, an welche Stelle im Programm er zurück springen muss, nachdem die Signalbehandlung abgeschlossen ist. Auf einem Testrechner sah das so aus:

```
$ ./a.out
pid = 5723
&main = 0x4007a9
alrm: sig = 14
```

```
alrm: esp = 0x7fff8a7c7870, ebp = 0x7fff8a7c78b0, eip = 0x400700
alrm: [00007fff8a7c7870] = [esp+   0] = 00007fff8a7c79c0
alrm: [00007fff8a7c7878] = [esp+   8] = 0000000e3bfaea74
[...]
alrm: [00007fff8a7c7960] = [esp+ 240] = 00007fff8a7c7d10
alrm: [00007fff8a7c7968] = [esp+ 248] = 000000000040080f
alrm: [00007fff8a7c7970] = [esp+ 256] = 0000000000000217     [...]
```

Die konkreten Adressen (Stack und Stack+Offset) können bei Ihnen wieder anders aussehen, aber `[esp+248]` ist ein guter Kandidat. Wenn Sie die Adresse an einem anderen Offset `[esp+`**`x`**`]` finden, dann merken Sie sich das **`x`**.

Bevor der Signal-Handler beendet wird, verändert er den Wert an der Adresse `esp+x`, so dass dort nicht mehr die ursprüngliche Rücksprungadresse, sondern die Adresse der Funktion `jump_test()` steht. Das erledigt der Befehl

```
write_memory (esp+248, &jump_test);
```

der als Makro implementiert ist und einen Assembler-`mov`-Befehl nutzt, um direkt in den Hauptspeicher zu schreiben.

b) Wenn Sie einen anderen Offset als 248 gefunden haben, müssen Sie diesen Befehl im Programm anpassen. Übersetzen Sie das Programm erneut und führen Sie es aus.

Durch die Änderung geht es nach der Rückkehr aus dem Signal-Handler nicht in der Endlosschleife in `main()`, sondern in der Funktion `jump_test()` weiter, welche schließlich `ENTERED jump_test()` ausgibt und dann eine eigene Endlosschleife startet.

Damit haben wir einen möglichen Mechanismus gefunden, um zwischen verschiedenen Threads hin und her zu schalten: Wenn wir den Signal-Handler regelmäßig aufrufen (lassen), können wir jeweils durch Manipulation des Stacks dafür sorgen, dass das Programm nach der Signalverarbeitung in einen anderen Thread zurück springt.

Leider eignet sich dieser Ansatz nicht besonders gut für eine Thread-Bibliothek, da sich der Offset (auf dem Stack, im Beispiel 248), der zur Rücksprungadresse führt, ändern kann.

c) Machen Sie sich noch mal klar, wie die Makros `read_memory` und `write_memory` funktionieren: Sie benutzen Cast-Operationen, um die Adresse in einen Pointer to `unsigned long` umzuwandeln, der Zugriff auf den Speicherinhalt erfolgt dann über den Dereferenzierungsoperator `*`.

19. Stack des Signal-Handlers

Signal-Handler können (wenn man sie nicht über `signal()`, sondern über `sigaction()` einträgt) zwei weitere Parameter (neben der Signalnummer) übernehmen, die formale Deklaration eines solchen Signal-Handlers sieht wie folgt aus:

```
void handler(int sig, siginfo_t *si, void *ptr);
```

Das zweite Argument interessiert (`si`) uns nicht, aber das dritte Argument `*ptr` zeigt auf eine Struktur vom Typ `ucontext_t`, und darin verbergen sich nützliche Daten. Wir suchen zunächst die Typdefinition und finden sie in `/usr/include/x86_64-linux-gnu/sys/ucontext.h`:

```
/* Userlevel context.  */
typedef struct ucontext {
    unsigned long int    uc_flags;
    struct ucontext      *uc_link;
    stack_t              uc_stack;
    mcontext_t           uc_mcontext;
    __sigset_t           uc_sigmask;
    struct _libc_fpstate __fpregs_mem;
  } ucontext_t;
```

Einige der enthaltenen Elemente sind selbst Strukturen, uns interessiert hier dasElement `uc_mcontext` (vom Typ `mcontext_t`). Der Typ `uc_mcontext` ist in derselben Header-Datei definiert:

```
/* Context to describe whole processor state.  */
typedef struct {
    gregset_t gregs;
    /* Note that fpregs is a pointer.  */
    fpregset_t fpregs;
    unsigned long __reserved1 [8];
} mcontext_t;
```

Und in dieser Struktur ist der Eintrag `gregs` ein Array von 64-Bit-Integer-Variablen:

```
typedef long int greg_t;          // Type for general register.
#define NGREG   23                // Number of general registers.
typedef greg_t gregset_t[NGREG];  // Container for all general
                                  // registers.
```

Hier ist Platz für die 23 allgemeinen Register einer 64-Bit-CPU, und diese lassen sich über Registernummern (`REG_RAX`, `REG_RBX` usw., in derselben Datei definiert) ansprechen.

Das bedeutet, dass man aus dem Signal-Handler auf die Registerinhalte zugreifen kann – und zwar auf deren Inhalte zum Zeitpunkt unmittelbar vor dem Aufruf des Handlers. Die drei wichtigen Register `RIP` (Instruction Pointer), `RSP` (Stack Pointer) und `RBP` (Base Pointer) sind über folgenden Code erreichbar:

```
static void handler(int sig, siginfo_t *si, void *ptr) {
  ucontext_t *uc = (ucontext_t *)ptr;  // erst den void-Ptr casten
  rip = uc->uc_mcontext.gregs[REG_RIP];
  rsp = uc->uc_mcontext.gregs[REG_RSP];
  rbp = uc->uc_mcontext.gregs[REG_RBP];       [...]
```

Wo liegt `gregs` eigentlich, und was bedeuten die Registerinhalte? Indem wir `RSP` auslesen und dann die Adresse von `gregs` anzeigen, erkennen wir:

```
rsp             =   7fff827b4a60
&gregs          = 0x7fff827b4ae8
&gregs[REG_RIP] = 0x7fff827b4b68
```

Die Differenz zwischen dem Stack-Pointer und dem `REG_RIP`-Eintrag von `gregs` ist 264 – in diesem Beispielprogramm ist der Offset nicht 248 (wie in Aufgabe 18), sondern 264.

Wir haben also die Stelle gefunden, an der auf dem Stack die Informationen über den vom Signal-Handler unterbrochenen Code liegen – und besser noch: Wir müssen nicht wie vorher den Offset erraten oder ausprobieren, sondern können über das dritte Argument der Handler-Funktion direkt darauf zugreifen.

a) Mit dem Programm `19a-context-test.c` können Sie die beschriebenen Funktionen prüfen: Es macht im Prinzip dasselbe wie das Programm aus Aufgabe 18, verwendet aber die alternative Funktion zum Eintragen eines Signal-Handlers, die auch Handler-Funktionen mit drei Argumenten (`sig`, `*si`, `*ptr`) zulässt. Lesen Sie den Code, übersetzen Sie das Programm und führen Sie es aus.

An dem Beispiel sehen Sie, dass das Umbiegen der Rücksprungadresse hier deutlich eleganter gelöst ist. Wir nutzen die Technik nun, um schrittweise Funktionen zu entwickeln, mit denen wir mehrere Threads erzeugen, zwischen denen der Prozess dann (über den Timer-Handler gesteuert) hin und her wechselt.

b) In dieser Aufgabe erzeugen Sie Ihre ersten „eigenen“ Threads (ohne Mithilfe der `pthread`-Bibliothek): Zunächst sind es nur zwei, zwischen denen das Programm hin und her wechselt.

In `19b-threads-01.c` sind bereits zwei Thread-Funktionen `test01()` und `test02()` definiert, die im Folgenden abwechselnd ausgeführt werden sollen; Sie geben jeweils in Endlosschleifen unterschiedliche Hallo-Welt-Meldungen aus, so dass man sie später unterscheiden kann.

Wir nutzen den Trick aus Ausgabe a) und ändern im Signal-Handler die Rücksprungadresse. Anders als in Aufgabe a) müssen wir aber jetzt auch den Fall verarbeiten, dass ein Thread bereits losgelaufen ist; wir können also nicht einfach jedesmal die Einsprungadresse von `test01` bzw. `test02` verwenden, sondern wir müssen uns vor einem Thread-Wechsel merken, wo es beim nächsten Mal im alten (unterbrochenen) Thread weiter geht, wenn wir ihn fortsetzen. Wir deklarieren dazu zwei Arrays `addresses[]` und `start[]`. In `addresses[]` merken wir uns bei jedem Kontext-Switch

die letzte aktuelle Adresse. `start[]` enthält die Einsprungadressen der beiden Funktionen und wird nur beim ersten Aufruf eines Threads benötigt. Da jeder Thread zusätzlich einen eigenen Stack benötigt, müssen wir diese Stacks zunächst allozieren und außerdem die Register `RSP` (Stack Pointer) und `RBP` (Base Pointer) bei jedem Thread-Wechsel sichern. Beim ersten Start eines Threads setzen wir beide Register auf das Ende des Stacks (`stackaddr+STACKSIZE`), denn Stacks wachsen ja von oben nach unten.

Das Programm enthält diverse Lücken (mit `// TO DO:` … kenntlich gemacht), hier müssen Sie Code ergänzen, um ein lauffähiges Programm zu erhalten. Stellen Sie das Programm fertig und testen Sie es, es sollte im Wechsel die beiden Threads ausführen, die jeweils ca. zehnmal ihre Hallo-Welt-Meldung ausgeben, bevor wieder zum anderen Thread gewechselt wird.

c) Jetzt verallgemeinern wie die Funktionalität und erlauben beliebig viele Threads (bis zu einer als `MAXTHREADS` definierten Maximalzahl), außerdem lagern wir die Initialisierung der Thread-Daten in eine Funktion `create_thread()` aus.

Die Programmdatei `19c-threads-02.c` enthält im Wesentlichen den Code aus Aufgabe b), es gibt zusätzlich drei globale Variablen `c1`, `c2`, `c3`, die von den Threads regelmäßig aktualisiert werden. Ein zusätzlicher Signal-Handler `ctrl_c_handler()` fängt die Tastenkombination Strg-C (Control-C) ab, gibt die Werte `c1`, `c2`, `c3` aus und beendet das Programm. Neben den schon bekannten Thread-Funktionen test01() und test02() gibt es ein paar weitere Funktionen (`rechnen()`, `vor_test02()`, `vor_vor_test02()`), mit denen das Programm prüft, ob Threads auch selbst weitere Funktionen aufrufen können. (Fehler an dieser Stelle würden auf Probleme mit dem Stack hinweisen.)

Wir haben außerdem bisher nur die Register RIP, RSP und RBP gesichert (was für unsere einfachen Testfunktionen ausgereicht hat), jetzt wollen wir alle Register sichern. Das ist sogar einfach als der vorherige Ansatz, weil wir einfach die ganze Struktur `uc->uc_mcontext.gregs` sichern können. Wir definieren drei neue Arrays:

```
mcontext_t  context[MAXTHREADS];
short       firstrun[MAXTHREADS];
void*       stacks[MAXTHREADS];
```

Das Sichern des aktuellen Kontext ist nun möglich, indem wir im Timer-Handler

```
context[tid] = uc->uc_mcontext;
```

ausführen: Strukturen können „in einem Rutsch“ kopiert werden. Allerdings enthält uc_mcontext als Element einen Zeiger, beim Kopieren durch einfaches Zuweisen wird nur der Zeiger selbst (die Speicheradresse) kopiert, nicht der dort liegende Inhalt. Das stört hier aber nicht (sondern wäre nur relevant, wenn wir z. B. einen Thread klonen wollten). Für den umgekehrten Weg verwenden wir analog

```
uc->uc_mcontext = context[tid];
```

Die Arrays `start[]`, `rsps[]` und `rbps[]` brauchen wir nur noch für die erste Ausführung eines Threads, es ist also weiterhin eine Fallunterscheidung nötig, ob ein Thread bereits gelaufen ist. Das Array `addresses[]` brauchen wir gar nicht mehr, die aktuellen RIP-Werte stehen in `gregs`.

Aufgabe: Ergänzen Sie die Funktionen `create_thread()`, `timer_handler()` und `main()`, so dass der Scheduler zwischen den in main() gestarteten fünf Threads hin und her wechselt. (Nicht jeder Thread führt eine separate Funktion aus!) Sie finden wieder verschiedene mit `// TO DO:` markierte Bereiche, in denen Sie Ergänzungen machen müssen. Einigen Code können Sie dabei aus Aufgabe b) leicht angepasst recyceln.

Kompilieren und testen Sie Ihr Programm. Es sollte ähnlich wie das Programm aus Aufgabe b) arbeiten. Wenn die Threads eine Weile (etwa eine halbe Minute) abwechselnd gelaufen sind, brechen Sie das Programm mit Strg-C ab – überprüfen Sie dann die abschließend ausgegebenen Werte in den Countern `c1`, `c2` und `c3` – sie sollten ungefähr gleich groß sein.

Ausblick: `create_thread()` leistet (in Verbindung mit dem Timer-Handler) schon fast dasselbe wie `pthread_create()` aus der POSIX-Bibliothek. Auf dem nächsten Übungsblatt werden Sie Aufgaben finden, mit denen Sie sich der Funktionalität von `pthread_create()` weiter annähern.