Vorbereitung

- Booten Sie den Rechner unter Linux und melden Sie sich mit Ihrem Account an (Passwort ist evtl. die Matrikelnummer). Öffnen Sie dann ein Terminalfenster (Konsole, xterm etc.).
- Legen Sie (falls noch nicht geschehen) in Ihrem Home-Verzeichnis ein Unterverzeichnis `bspraktikum` an, das geht mit dem Befehl
  `mkdir bspraktikum`
- Wechseln Sie in das neue Verzeichnis:
  `cd bspraktikum`
- Laden Sie das Archiv mit den Aufgaben-Dateien herunter, das geht mit
  `wget http://fhm.hgesser.de/bs-ss2008/prakt02.tgz`
- Entpacken Sie das Archiv mit `tar`:
  `tar xzf prakt02.tgz`
- Dadurch entsteht ein neues Unterverzeichnis `prakt02`, in das Sie hinein wechseln:
  `cd prakt02`

1. Zombie-Prozess erzeugen

Übersetzen Sie das Programm `fork-zombie.c` mit folgendem Befehl:

```
gcc -o fork-zombie fork-zombie.c
```

und führen Sie es anschließend aus:

```
./fork-zombie
```

Betrachten Sie anschließend in einem zweiten Terminalfenster mit

```
ps -o pid,ppid,cmd,state -C fork-zombie
```

einen Teil der Prozessliste. Über die Process ID (PID) und die Parent Process ID (PPID) können Sie herausfinden, welcher der beiden Prozesse der Vater und welcher der Sohn ist.

[☛ Lösungsblatt]

a) Welcher Prozess (Vater oder Sohn) ist der Zombie?

b) Wie kommt es hier dazu, dass ein Zombie-Prozess entsteht?

2. Posix-Threads

Betrachten Sie das aus der Vorlesung bekannte Programm `thread-beispiel.c`, das Sie mit dem Kommando

```
gcc -lpthread -o thread-beispiel thread-beispiel.c
```

übersetzen können. (Über die Option `-lpthread` binden Sie die pthread-Bibliothek zum Programm hinzu.)

a) Führen Sie das Programm aus.
[☛ Lösungsblatt] Notieren Sie, welche Informationen es ausgibt (nicht den vollen Text, nur eine Beschreibung).

b) Öffnen Sie ein zweites Terminalfenster und beobachten Sie dort die Prozess- und Thread-Liste mit dem Befehl
`ps -Lf -C thread-beispiel`

Sie können für eine regelmäßige Anzeige sorgen, indem Sie das Kommando mit `watch` ausführen; der richtige Befehl heißt dann

```
watch -n1 "ps -Lf -C thread-beispiel"
```

In der Ausgabe sehen Sie für (bis zu) drei Threads die Prozess-ID (PID), Parent Process ID (PPID) und Thread ID (LWP, Light Weight Process ID). Wenn die Ausgabe zu schnell verschwindet, ändern Sie in der Definition von `thread_function1` und `thread_function2` die Bedingung `i<10` in (z. B.) `i<30`.

[☛ Lösungsblatt] Stellen Sie fest, welche Prozess-IDs in der Spalte PPID eingetragen sind – was sagt Ihnen das über Vater-Sohn-Verhältnisse zwischen dem „Haupt-Thread“ und den von ihm erzeugten Threads?

c) Öffnen Sie ggf. noch ein drittes Terminalfenster und schicken Sie versuchsweise einem der Threads (unter Angabe seiner Thread-ID aus der Spalte LWP!) ein STOP-Signal (`kill -STOP` *`LWPID`*).
[☛ Lösungsblatt] Was geschieht dann mit dem Thread, dem Sie das Signal schicken, und was geschieht mit den übrigen Threads?
Schicken Sie anschließend einem anderen (!) Thread das CONT(inue)-Signal (`kill -CONT` *`LWPID`*).
[☛ Lösungsblatt] Was geschieht dann mit den Threads?

3. Globale Variable im Thread

Die Datei `zwei-threads.c` ähnelt der Datei `thread-beispiel.c`, enthält aber nur ein Gerüst für den Start der beiden Threads, außerdem fehlen die Verzögerungen. Dafür wird am Anfang eine globale Integer-Variable `i` definiert.

a) Füllen Sie die beiden Thread-Funktionen mit Inhalt: Eine der Funktionen soll die globale Variable initialisieren und dann zwei Sekunden warten (`sleep`); die andere soll zunächst kurz warten und dann den Wert der Variable ausgeben (`printf("i=%d\n", i);`).
[☛ Lösungsblatt] Welchen Wert hatten Sie der Variable zugewiesen und welcher wird ausgegeben? Was sagt Ihnen das über die Speichernutzung der Threads?

```
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>  /* printf() */

int i;

void *thread_function1(void *arg) {
  return 0;
}
void *thread_function2(void *arg) {
  return 0;
}
int main(void) {
  pthread_t mythread1;
  pthread_t mythread2;
  /* Threads erzeugen */
  if ( pthread_create( &mythread1, NULL, thread_function1, NULL) ) {
    printf("Fehler bei Thread-Erzeugung."); abort();
  }
  if ( pthread_create( &mythread2, NULL, thread_function2, NULL) ) {
    printf("Fehler bei Thread-Erzeugung ."); abort();
  }
  /* Threads wieder einsammeln */
  if ( pthread_join ( mythread1, NULL ) ) {
    printf("Fehler beim Join."); abort();
  }
  if ( pthread_join ( mythread2, NULL ) ) {
    printf("Fehler beim Join."); abort();
  }
  exit(0);
}
```

b) Passen Sie Ihr Programm an: Vor den Thread-Definitionen initialisieren Sie die Variable bereits auf 0 (`int i=0;`). Im ersten Thread soll nun 1000 mal der Wert 1 addiert werden, im zweiten Thread 1000 mal der Wert 1 abgezogen werden – geben Sie im Hauptprogramm (vor `exit(0);`) den finalen Wert von `i` aus. Dieser sollte 0 sein. Wenn Sie das Programm starten, erhalten Sie mit großer Wahrscheinlichkeit auch den Wert 0. Lassen Sie es über eine Schleife ca. 500 bis 1500 mal laufen und prüfen Sie die Ausgaben. (Tipp: Lesen Sie die Manpage zu `sort`, insbesondere zur Option `-u`.)
[☛ Lösungsblatt] War das Ergebnis immer 0? Wenn nein, warum nicht?

Lösungsblatt – Übung 2

Punkte:

	Name	**Matrikelnummer**
Teilnehmer 1		
Teilnehmer 2		