– Lösung zur Praktikumsaufgabe 11 –

Thema: *Threads*

1. a) Listing 1 zeigt eine beispielhafte Lösung.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

void* thread_function(void *arg)
{
  printf("Hello, thread!\n");
  pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
  int ret;
  pthread_t thid;

  ret = pthread_create(&thid, NULL, &thread_function, NULL);
  if (ret != 0) {
    perror("pthread_create");
    exit(EXIT_FAILURE);
  }
  ret = pthread_join(thid, NULL);
  if (ret != 0) {
    perror("pthread_join");
    exit(EXIT_FAILURE);
  }
  exit(EXIT_SUCCESS);
}
```

Listing 1: „Hello, world!“ mit POSIX-Threads (`hello-pthread.c`)

b)* Es muss darauf geachtet werden, dass bei der Rückgabe eines Wertes mit Zeigern gearbeitet wird. Da der Thread nach `pthread_exit()` vernichtet wird, ist ein Zeiger auf lokale Variablen des Threads nicht mehr gültig. Stattdessen kann mit Zeigern auf globale oder Heap-Variablen gearbeitet werden. Listing 2 zeigt eine mögliche Lösung.

2. Listing 3 zeigt den entsprechenden Code.

   Der Wert der globalen Variable ist nicht mehr 0 und auch nicht vorherzusehen.

```
$ ./2threads-cs-wrong
x is 63967256.
$ ./2threads-cs-wrong
x is -28157540.
$ ./2threads-cs-wrong
x is 72472397.
```

```c
/*

*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

int answer;

void* thread_function(void *arg)
{
  printf("%s\n", arg);

  answer = 42;
  pthread_exit(&answer);
}


int main (int argc, char *argv[])
{
  int ret;
  void *result;
  long c;
  pthread_t thid;

  ret = pthread_create(&thid, NULL, &thread_function, "Hello, world!");
  if (ret != 0) {
    perror("pthread_create");
    exit(EXIT_FAILURE);
  }

  ret = pthread_join(thid, &result);
  if (ret != 0) {
    perror("pthread_join");
    exit(EXIT_FAILURE);
  }

  printf("Die Antwort auf die Frage lautet: %d.\n", * (int*) result);

  exit(EXIT_SUCCESS);
}
```

Listing 2: Parameterübergabe an und Resultatwert von Threads (`paramthread.c`)

Der Grund ist, dass sich beide Threads beim schreibenden Zugriff auf die Variable „ins Gehege“ kommen. Die schreibenden Zugriffsoperationen sind so genannte *kritische Abschnitte* bezüglich der Variablen. Da diese nicht geschützt sind, gehen viele Wertaktualisierungen der gemeinsam genutzten Variablen verloren. Was es damit genau auf sich hat, erlernen Sie in der Lehrveranstaltung „Betriebssysteme 2“.

Auf Maschinen mit mehreren Cores genügen schon viel weniger als 100 Millionen Iterationen, um den Fehler zu provozieren.

```c
/*

*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define ITERATIONS 100000000

long x;

void* thread_function(void *arg)
{
  unsigned long c;

  for (c=0; c<ITERATIONS; c++) {
    x = x - 1;
  }

  pthread_exit(NULL);
}


int main (int argc, char *argv[])
{
  int ret;
  unsigned long c;
  pthread_t thid;

  ret = pthread_create(&thid, NULL, &thread_function, NULL);
  if (ret != 0) {
    perror("pthread_create");
    exit(EXIT_FAILURE);
  }

  for (c=0; c<ITERATIONS; c++) {
    x = x + 1;
  }

  ret = pthread_join(thid, NULL);
  if (ret != 0) {
    perror("pthread_join");
  }

  printf("x is %ld.\n", x);

  exit(EXIT_SUCCESS);
}
```

Listing 3: Kritischer Abschnitt bezüglich einer Variablen, unmanaged (`2threads-cs-wrong.c`)

3.* Der kritische Abschnitt kann beispielsweise mit einem Mutex geschützt werden, der den Eintritt eines Threads verhindert, so lange noch der andere Thread im kritischen

Abschnitt arbeitet. Damit wird wechselseitiger Ausschluss garantiert. Listing 4 zeigt die entsprechende Implementation.

Diese Korrektheit gibt es aber nicht umsonst. Durch die Vielzahl Kollisionen und die nötigen Systemein- und -austritte erhöht sich die Abarbeitungszeit drastisch, wie das folgende Abarbeitungsprotokoll demonstriert (CPU: AMD Ryzen 5):

```
$ time ./2threads-cs-wrong
x is -36802564.

real 0m0,724s
user 0m1,436s
sys 0m0,001s
$ time ./2threads-cs-correct
x is 0.

real 0m9,126s
user 0m8,389s
sys 0m9,533s
```

```c
/*

*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define ITERATIONS 100000000

long x;

pthread_mutex_t mymutex = PTHREAD_MUTEX_INITIALIZER;

void* thread_function(void *arg)
{
  long c;

  for (c=0; c<ITERATIONS; c++) {
    pthread_mutex_lock(&mymutex);
    x = x - 1;
    pthread_mutex_unlock(&mymutex);
  }

  pthread_exit(NULL);
}


int main (int argc, char *argv[])
{
  int ret;
  long c;
  pthread_t thid;

  ret = pthread_create(&thid, NULL, &thread_function, NULL);
  if (ret != 0) {
    perror("pthread_create");
    exit(EXIT_FAILURE);
  }

  for (c=0; c<ITERATIONS; c++) {
    pthread_mutex_lock(&mymutex);
    x = x + 1;
    pthread_mutex_unlock(&mymutex);
  }

  ret = pthread_join(thid, NULL);
  if (ret != 0) {
    perror("pthread_join");
  }

  printf("x is %ld.\n", x);

  exit(EXIT_SUCCESS);
}
```

Listing 4: Schutz des kritischer Abschnittes bezüglich einer Variablen mittels eines Mutexes, (`2threads-cs-correct.c`)