Multi-Threading unter Linux

Dieses Hilfsblatt beschreibt einige Grundlagen zur Behandlung von Pthreads in dem Umfang, wie sie zur Bearbeitung der Praktikumsaufgabe benötigt werden. Die Pthread Library hat einen höheren Funktionsumfang. Schauen Sie sich die benutzten Systemaufrufe zusätzlich in der Dokumentation an!

- http://www.opengroup.org/onlinepubs/009695399/functions/xsh_chap02_09.html

Soll die Pthread Library benutzt werden, muss
`#include <pthread.h>` eingebunden werden.

Erzeugen eines Threads

In einem POSIX konformen System werden Threads mit dem `pthread_create()` Befehl erzeugt und gestartet.

Syntax

```
int pthread_create(pthread_t *restrict thread,
const pthread_attr_t *restrict attr,
void *(*start_routine)(void*), void *restrict arg);
```

Parameter

- `thread`: Ein Handle auf den Thread. Im Erfolgsfall wird hier die ID des Threads hineingeschrieben.
- `attr`: Attribute des Threads. Sollen keine speziellen Attribute benutzt werden (was im Rahmen des Praktikums der Fall ist), kann hier NULL übergeben werden.
- `start_routine`: Adresse der Funktion, die als Thread gestartet wird. Die Funktion muss vom Typ: `void * func(void* arg)` sein (kein Rückgabewert, generische Parameterliste).
- `arg`: Zeiger auf Argumentenstruktur für die Funktion `start_routine`. Rückgabewert
- 0: Der Thread wurde erfolgreich erzeugt und gestartet.
- Fehlernummer: ein Fehler ist aufgetreten.

Beenden eines Threads

Zum Beenden eines Threads in einer in einem Thread laufenden Funktion wird der Befehl `pthread_exit()` benutzt.

Syntax

```
void pthread_exit(void *value_ptr);
```

Parameter

- `value_ptr`: wird an den Erzeuger zurückgegeben. Der Zeiger darf nicht auf lokale Daten zeigen! Die Daten müssen auf dem Heap liegen, der Erzeuger ist dafür verantwortlich, den Speicher wieder frei zu geben.

Rückgabewert Von hier gibt es kein zurück! Wird ein Thread mit `return value_ptr;` verlassen, wird implizit die `pthread_exit()` Funktion aufgerufen.

Warten auf einen Thread

Soll der Rückgabewert eines Threads ausgewertet werden, so muss der Aufrufer auf das `pthread_exit()` (oder das `return`) des Threads warten. Diese Synchronisierung erfolgt mit dem `pthread_join()` Befehl.

Syntax

```
int pthread_join(pthread_t thread, void **value_ptr);
```

Parameter

- `thread`: Der Thread Handle, der beim Erzeugen des Threads beschrieben wurde.
- `value_ptr`: Hier wird der Zeiger aus `pthread_exit()` abgelegt. Werden keine Rückgabewerte erwartet, kann hier NULL übergeben werden.

Rückgabewert

- 0: Kein Fehler
- Fehlernummer: Ein Fehler ist aufgetreten.

Achtung: wird der Hauptthread (z.B. `main()`) eines Prozesses mit `return` verlassen, werden auch alle ”Unterthreads” beendet. Deshalb sollt auch dann auf die Threads gewartet werden, wenn kein Rückgabewert erwartet wird.

Beispiel

```
void* funcThread(void *arg)
{
  struct ThreadArgumente *arguments = (ThreadArgumente *)arg;
  struct RetValues *ret = NULL
  /* Do whatever has to be done. */
  return (void *)ret;
  /* or:
  pthread_exit((void *)ret);
  */
}
int main(int argc, char* argv[])
  {
  struct ThreadArgumente arguments;
  struct RetValues *ret = NULL
  pthread_t thread;
  int status = 0;
  /* write something in arguments... */
  status = pthread_create (&thread,
  NULL,
  funcThread,
  &arguments);
  /* do what else has to be done */
  status = pthread_join(thread, &ret);
  /* maybe something else has to be done */
  return 0;
}
```

Synchronisation zwischen Threads

Bitte beachten Sie, dass einige Funktionen der C-Standardbibliothek auch interne Strukturen (statische Variablen) verwalten. Diese Funktionen sind nicht reentrant (nicht thread save). Benutzen mehrere Threads diese Funktionen, sind die Ergebnisse nicht determiniert, sondern es entsteht eine Race Condition. Aus diesem Grund gibt es für viele Funktionen eine reentrante Implementierung, die in einem Prozess bei Multi-Threading benutzt werden muss. Ein Beispiel für eine nicht reentrante Funktion ist `strtok()`. Hier gibt es die reentrante Version `strtok_r()`. Greifen mehrere Threads auf gemeinsame Daten zu, muss der Zugriff auf diese Daten geregelt sein, um Race Conditions zu vermeiden. In der Pthread Library gibt es Mutexe (mutual exclusion, gegenseitiger Ausschluss) und Bedingungsvariablen (conditions variables) zur Synchronisation der Zugriffe.

Mutex

Ein Mutex serialisiert den Zugriff auf kritische Ressourcen und realisiert damit einen wechselseitigen Ausschluss. Nur ein Thread kann zu einer bestimmten Zeit seinen durch einen Mutex geschützten kritischen Bereich betreten. Ein Mutex ist eine Variable vom Typ `pthread_mutex_t`. Vor der Benutzung muss der Mutex initialisiert werden.

Syntax

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
const pthread_mutexattr_t *restrict attr);
```

oder:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Parameter

- `mutex`: Zeiger auf den zu initialisierenden Mutex.
- `attr`: Zeiger auf eine Struktur mit Attributen. Sollen keine speziellen Attribute benutzt werden (wie hier im Rahmen des Praktikums) kann ein NULL Zeiger übergeben werden.

Rückgabewert

- 0: Kein Fehler.
- Fehlernummer: Ein Fehler ist aufgetreten.

Aus Gründen der Wartbarkeit und Lesbarkeit des Codes sollte ein Mutex mit den kritischen Daten verknüpft sein (siehe Beispiel unten). Ein Mutex kann durch die Funktion `pthread_mutex_lock()` gesperrt und durch `pthread_mutex_unlock()` wieder freigegeben werden.

Syntax

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Parameter

- `mutex`: Zeiger auf den zu sperrenden bzw. freizugebenden Mutex.

Rückgabewert

- 0: Kein Fehler.
- Fehlernummer: Ein Fehler ist aufgetreten.

Wird der Mutex nicht mehr benötigt, muss er zerstört werden, da er Ressourcen im Betriebssystem belegt. Hierzu muss die `pthread_mutex_destroy()` Funktion benutzt werden. Ein Mutex darf nicht mehr gesperrt sein, wenn er zerstört wird.

Syntax

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Parameter

- `mutex`: Zeiger auf den zu zerstörenden Mutex.

Beispiel (ohne weitere Fehlerbehandlung)

```
/* in header file or at beginning of C file */
typedef struct tag_t
{
  pthread_mutex_t *mutex;
  struct MyCriticalData data;
} Tag_t;
/* somewhere at the beginning of a function */
void foo()
{
  int status = 0;
  Tag_t protected;
  protected->mutex = (pthread_mutex_t *)malloc(
  sizeof(pthread_mutex_t));
  status = pthread_mutex_init(protected->mutex, NULL);
  /* do something not critical */
  status=pthread_mutex_lock(protected->mutex);
  /* do something critical with protected->data */
  status=pthread_mutex_unlock(protected->mutex);
  /* do something else not critical */
  status = pthread_mutex_destroy(protected->mutex);
  free(protected->mutex);
}
```

Bedingungsvariablen
Mit Bedingungsvariablen können komplexere Synchronisationen (z.B. Barrieren) aufgebaut werden. Die Bedingungsvariable gibt Auskunft über den Zustand gemeinsam genutzter Daten. Bedingungsvariablen sind mit Mutexen verknüpft und werden immer mit diesen gemeinsam verwendet. Ein Thread kann auf eine Bedingung warten, die dann von einem anderen Thread signalisiert werden muss. Zur Benutzung siehe auch das Beispiel unten. Ebenso wie Mutexe sind auch Bedingungsvariablen Datentypen, die im Betriebssystem angelegt, initialisiert und nach Benutzung wieder zerstört werden müssen. Erzeugen, Initialisieren und Zerstören der Bedingungsvariablen erfolgt analog zu den Mutexen.

Syntax

```
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_init(pthread_cond_t *restrict cond,
const pthread_condattr_t *restrict attr);
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Benutzung der Bedingungsvariablen
Ein Thread, der auf eine Bedingung wartet, muss die `pthread_cond_wait()` Funktion (oder `pthread_cond_timedwait()`) aufrufen. Der Aufruf ist nur in einem kritischen Bereich erlaubt, d.h. es muss vorher ein Mutex gesperrt worden sein. In einem atomaren Zugriff wird der Mutex freigegeben und der Thread geht in den blockiert Zustand über. In diesem Zustand wartet er, bis die Bedingung erfüllt ist.

Syntax

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex,
const struct timespec *restrict abstime);
int pthread_cond_wait(pthread_cond_t *restrict cond,
pthread_mutex_t *restrict mutex);
```

Parameter

- `cond`: Bedingungsvariable, die signalisiert warden muss.
- `mutex`: Der mit der Bedingungsvariable verknüpfte Mutex.
- `abstime`: Zeit, die gewartet wird. Erfolgt keine Signalisierung innerhalb der gegebenen Zeit, kann der Thread weiterlaufen.

Rückgabewert

- 0: Kein Fehler.
- ETIMEDOUT: Die Zeit abstime ist abgelaufen, ohne dass eine Signalisierung erfolgt ist.
- Fehlernummer: Ein Fehler ist aufgetreten.

Signalisierung der Bedingung
Der signalisierende Thread kann einen wartenden Thread mit `pthread_cond_signal()` oder an alle wartenden Threads `pthread_cond_broadcast()` signalisieren. Warten mehrere Threads auf eine Bedingung, ist nicht definiert, welcher Thread nach `pthread_cond_signal()` weiterläuft (in der Regel ist dies der erste Prozess, der blockierte). Achtung: Die Aufrufe werden nicht gespeichert. Erfolgt eine Signalisierung bevor der wait-Aufruf erfolgt, geht die Signalisierung verloren.

Syntax

```
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Parameter

- `cond`: Zeiger auf die Bedingungsvariable.

  Rückgabewert

- 0: Kein Fehler.
- Fehlernummer: Ein Fehler ist aufgetreten.

Beispiel

Zwei Threads, die über zwei Bedingungsvariablen synchronisiert sind. Bitte beachten Sie die Variablen `sync1` und `sync2` und ihre Benutzung. Die `while`-Schleifen im Code verhindern ein ungewolltes Verlassen der Synchronisationsbedingungen, denn zwischen dem Signalisieren der Bedingung und dem Entsperren des durch die Bedingung blockierten Threads kann ein anderer Thread den Zustand des Systems schon wieder verändert haben. `sync1` und `sync2` sind die Variablen, auf die sich die Threads synchronisieren.

```
/* Structure with required data */
typedef struct tag_t
{
  pthread_mutex_t *mut;
  pthread_cond_t *cond1;
  pthread_cond_t *cond2;
  int sync1;
  int sync2
  struct MyCriticalData data;
} Tag_t;
void* thread1(void* args)
{
  Tag_t *tag = (Tag_t *)args;
  int aCondition = 0;
  /* do something */
  /* now we reach a critical section: */
  status=pthread_mutex_lock(tag->mut);
  while(tag->sync1)
  {
    pthread_cond_wait (tag->cond1, tag->mut);
  }
```

```
  /* do something with tag->data */
  /* calculate aCondition */
  if (aCondition) tag->sync2 = 0;
  pthread_mutex_unlock(tag->mut);
  pthread_cond_signal(tag->cond2);
  /* do something more */
}
void* thread2(void* args)
{
  Tag_t *tag = (Tag_t *)args;
  int aCondition = 0;
  /* do something */
  /* now we reach a critical section: */
  status=pthread_mutex_lock(tag->mut);
  while(tag->sync2)
  {
    pthread_cond_wait (tag->cond2, tag->mut);
  }
  /* do something with tag->data */
  /* calculate aCondition */
  if (aCondition) tag->sync1 = 0;
  pthread_mutex_unlock(tag->mut);
  pthread_cond_signal(tag->cond1);
  /* do something more */
}
```

Komprimieren mit miniz

Das Projekt miniz ist eine verlustfreie, leistungsstarke Bibliothek zur Komprimierung. Nähere Details sind der offiziellen Projektseite zu entnehmen:

- https://code.google.com/p/miniz/

In diesem Beispiel wird ein String unter Zuhilfenahme eines Wrappers komprimiert, seine Länge und seine Länge nach der Kompression ausgegeben. Alle benötigten Funktionalitäten befinden sich in der Wrapper-Datei miniz.h.

Auszug aus der Dokumentation von miniz.h:

Es wird eine Instanz der `struct Result` auf dem Heap erzeugt. Dann wird der String komprimiert und das Ergebnis in `Result` im Feld `data` gespeichert (dafür wird neuer Heap-Speicher alloziert). Die Länge des komprimierten Inhalts wird im Feld `length` gespeichert.

Es wird ein Pointer auf o.g. `Result` zurückgeliefert oder `NULL`, wenn kein Speicher mehr reserviert werden konnte oder das Komprimieren fehlschlug.

Wird `Result` nicht mehr gebraucht, so muss zuerst das Feld `data` und danach `Result` selbst durch `free()` freigegeben werden.

```
#include "miniz.h"

int main(void)
{
  const char *string = "Mich kann man gut kürzen kürzen
      kürzen kürzen kürzen kürzen kürzen kürzen kürzen.";
  Result *result = compress_string(string);

  printf("Gekürzt/aufgebläht von %d auf %d Bytes.", strlen(string), result->length);
  free(result->data);
  free(result);
  return 0;
}
```

Verwendung der Queue

Im folgenden Beispiel werden zwei Instanzen einer selbstdefinierten `struct Student` (bestehend aus Name und Alter) in eine Queue eingehängt. Danach wird der am Anfang der Queue stehende Student herausgenommen und sein Name ausgegeben. Um die Queue näher zu verstehen und ihre Funktionen kennenzulernen, schauen Sie bitte in die Datei `queue.h`.

```
#include "queue.h"

typedef struct student
{
    const char *name;
    int alter;
} Student;

static Queue queue;

int main(void)
{
  queue = queue_create();

  Student *a = (Student *) malloc(sizeof(Student));
  Student *b = (Student *) malloc(sizeof(Student));

  if (!a || !b)
  {
    perror("Es konnte kein Speicher reserviert werden.");
    exit(-1);
  }

  a->name = "Bert";
  a->alter = 27;

  b->name = "Schnabelbus";
  b->alter = 79;

  queue_insert(queue, a);
  queue_insert(queue, b);

  // Vorderstes Element aus der Queue nehmen, löschen und ausgeben
  Student *student = queue_head(queue);
  queue_delete(queue);
  printf("%s", student->name); //-> Bert
```

```
  // Den von student verbrauchten Speicher freigeben
  // (der andere Student befindet sich noch in der Queue)
  free(student);

  return 0;
}
```