Multi-Threading unter Unix / Linux

Dieses Hilfsblatt beschreibt einige Grundlagen zur Behandlung von **Pthreads** in dem Umfang, wie sie zur Bearbeitung der Praktikumsaufgabe benötigt werden. Die Pthread Library, die Threads in portable Weise zur Verfügung, 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

Im Lernraum finden Sie ein Beispielprojekt (**`prosumer.c`**), welches ein einfaches Erzeuger-Verbrauchersystem mit Hilfe der Pthread Library implementiert.

Mit C++-11 wurden sehr ähnliche Konzepte zur Thread-Behandlung bei C++ in objekt-orientierter Form integriert. Das Laufzeitsystem unter Linux greift zur Umsetzung auf die Pthread Bibliothek zurück. Auch hierzu befindet sich im Lernraum ein Beispielprojekt (**`prosumer.cpp`**).

Grundlegendes

Soll die Pthread Library benutzt werden, muss

```
#include <pthread.h>
```

eingebunden werden. Beim Kompilieren und Binden per gcc muss die Option **`-pthread`** angegeben werden.

Für C++ müssen beim g++ die Optionen **`-std=c++11 -pthread`** verwendet werden. Für die Erzeugung von Threads muss

```
#include <thread>
```

inkludiert werden. Zur Prozess-Synchronisation werden in der Regel noch

```
#include <mutex>
#include <atomic>
#include <condition_variable>
```

benötigt. Die folgende Auflistung der Programmierkonstrukte fokussiert die Realisierung in C. Die entsprechenden Mechanismen in C++ sind farblich gekennzeichnet.

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.

In C++-11 wird ein Thread über den Konstruktor der Klasse **thread** angelegt:

```
template < class Function, class ... Args >
explicit thread (Function && f, Args &&... args);
```

Die Funktion **f** wird dazu mit den Argumenten **args** aufgerufen.

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.

In C++-11 ruft der Destruktor eine Methode **terminate()** auf.

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 ThreadArgs* arguments = (ThreadArgs *) 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 ThreadArgs  arguments;
  struct RetValues*  ret = NULL
  pthread_t  thread;
  int status = 0;
  /* deliver data via 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;
}
```

In C++-11 kann zum Warten auf einen Thread die Methode **join()** auf dem betreffenden Thread aufgerufen werden.

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 jeweils eine reentrante Realisierung, die in einem Prozess bei Multi-Threading benutzt werden sollte.

Ein Beispiel für eine **nicht reentrante** Funktion ist **`strtok()`**. Hier gibt es die reentrante Version **`strtok_r()`**.

Greifen mehrere Threads nebenläufig auf gemeinsame Daten zu, so muss der Zugriff auf diese Daten geregelt werden, 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.

Mutex erzeugen

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).

In C++-11 wird ein Mutex per Konstruktor **`std::mutex`** erzeugt.

Mutex sperren und freigeben

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.

In C++-11 wird ein Mutex per Methode **lock()** gesperrt und per **unlock()** wieder frei gegeben.

Mutex löschen

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 löschenden 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 concurrent 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);
}
```

In C++-11 wird der Mutex per Destruktor gelöscht. Achtung: Bei auf dem Stack angelegten Variablen wird dieser aufgerufen, wenn die Variable aus dem Gültigkeitsbereich herausfällt.

Bedingungsvariablen:

Mit Bedingungsvariablen können komplexere Synchronisationen (z. B. Barrieren) aufgebaut werden. Ein Thread kann auf eine Bedingung warten, deren Erfüllung von einem anderen Thread signalisiert wird. Die Bedingungsvariable gibt Auskunft über den Zustand gemeinsam genutzter Daten. Bedingungsvariablen sind mit Mutexen verknüpft und werden immer mit diesen gemeinsam verwendet: so kann der kritische Abschnitt, in dem die Bedingung abgefragt wird, solange für einen anderen Thread freigegeben werden, bis die Bedingung erfüllt ist.

Ebenso wie Mutexe werden auch Bedingungsvariablen durch entsprechende Datentypen realisiert. Die Variablen werden im Betriebssystem initialisiert und müssen nach Benutzung wieder zerstört werden.

Erzeugen, Initialisieren und Zerstören der Bedingungsvariablen erfolgt analog zu den Mutexen.

Syntax

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

int pthread_cond_destroy(pthread_cond_t *cond);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Nutzung von 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*, der durch einen Mutex kontrolliert wird. In einem atomaren Zugriff wird der referenzierte Mutex freigegeben und der aktuelle Thread ‚blockiert'. In diesem Zustand wartet der Thread, bis die Bedingung erfüllt ist, d. h. ein entsprechendes Signal von einem anderen Prozess geschickt wurde.

Syntax

```
int pthread_cond_wait(pthread_cond_t *restrict cond,
       pthread_mutex_t *restrict mutex);

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

Parameter

- **cond**: Bedingungsvariable, auf die gewartet wird.
- **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.

In C++-11 kann eine Bedingungsvariable per **std::condition_variable()** erzeugt werden. Eine solche Variable ist nicht kopierbar.

Signalisierung von Bedingungen

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_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(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.

```
/* struct containing required attributes */
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 */
}
```

In C++-11

- kann auf eine Bedingungsvariable vom Typ **`condition_variable`** per Methode **`wait()`** gewartet werden, die einen Booleschen Wert zurückgibt (**`true`**, wenn Bedingung erfüllt).
- Mit **`notify_one()`** kann ein Thread, der auf die Bedingung wartet, deblockiert werden.
- Mit **`notify_all()`** werden alle wartenden Prozesse deblockiert.