Einführung in die Threadprogrammierung

Florian Mayer

Februar 2014

Zusammenfassung

Dieses Dokument ist als eine Einführung in die Welt der Threadprogrammierung zu verstehen. Dabei wird in den Beispielen konsequent C als Programmiersprache verwendet und davon ausgegangen, dass der Leser ein wesentliches Verständnis für die Funktionsweise von unixoiden Betriebssystemen besitzt.

1 Allgemeines

1.1 Kernel- und User-Threads

Es wird zwischen sog. User- und Kernel-Threads unterschieden. Der POSIX Standard Pthreads definiert einen Satz von C-Tpyen und Funktionen, die plattformunabhängiges Programmieren mit Kernel-Threads bieten. Kernel-Threads werden vom Betriebssystem verwaltet. Hierbei besitzt ein Prozess entweder einen oder mehrere Threads, die jeweils gescheduled ablaufen. Jeder Prozess hat nach wie vor seinen eigenen Speicherbereich, geöffnete Dateien, ein Datensegment und ein Codesegment. Wenn der Prozess nur einen Kernel-Thread besitzt, verfügt er nur über einen Befehlszähler und ein Stacksegment. Wenn hingegen mehrere Kernel-Threads innerhalb des Prozesses "laufen", wird für jeden Thread ein separater/es Befehlszähler bzw. Stacksegment mitgeführt. Einen Prozess mit n Threads liegt nennt man Task.
User-Threads oder Userlevel-Threads sind keine Threads im herkömmlichen Sinne. Sie ermöglichen zwar die "parallele" Abarbeitung von Anweisungen, allerdings ist die Funktionalität dabei nicht direkt im Kernel implementiert. Der Kontextwechsel zwischen den einzelnen User-Threads übernimmt dabei eine User-Thread-Bibliothek, die somit auch über einen eigenen Scheduler verfügen muss, der nicht im Kernel, sondern im Userland ausgeführt wird. Folglich weiß das Betriebssystem nichts von der Existenz von User-Threads. Duch das Fehlen des priviligierten Prozessor-Modus kann nur kooperatives Scheduling betrieben werden. Kernel-Threads werden hingegen präemptiv gescheduled.

1.2 Implementierungen

- User-Threads
  - Linux: LinuxThreads (nicht mehr weiterentwickelt)
  - Windows: seit Windows 98 Fibers
  - JVM verwendet Green Threads, falls der JVM-Host keine Kernel-Threads bietet
- Kernel-Threads
  - Linux: Native Posix Thread Library (NPTL)
  - Windows: Funktionen der win32/64-API

1.3 Platformunabhängigkeit

Nahezu alle modernen UNIX basierten Betriebssysteme halten sich an den POSIX-Standard. Die Impelemntierungen der PThreads-Library Unterscheiden sich allerdings stark im Hinblick auf Standardwerte. Beispielsweise beträgt die größe des Stacks für Thread-Routinen auf Plattform A 2^13 Byte, auf Plattform B jedoch lediglich 2^10 Byte. Den vollständigen POSIX-Standard kann man auf der Webseite opengroups.org einsehen.
(http://www.opengroup.org/onlinepubs/007904975/toc.htm)

2 Einführung in POSIX-Threads

Die Bezeichner aller Funktionen haben das Präfix "pthread_". Man kann die Funktionen in folgende vier Gruppen einteilen:

1. Thread Management
2. Wechselseitiger Ausschluss
3. Bedingungsvariablen
4. Synchronsiationsfunktionen

2.1 Thread-Management

2.1.1 Funktionen

- `int pthread_create(pthread_t *thread, const pthread_attr_t *attribute, void *(*function)(void *), void *args);`
  Dient der Erstellung von Threads. Die Funktion gibt 0 zurück, wenn sie erfolgreich war, ansonsten einen Fehlercode ungleich 0. Returnstatements in einer in einem Thread ausgeführten Funktion beenden den jeweiligen Thread. Der Aufruf von `void exit(x);` in einem Thread beendet den gesamten Task, also alle Threads, die zum Prozess.

- `void pthread_exit(void *return);`
  Führt zur Beendigung eines aufrufenden Threads.

- `void pthread_join(pthread_t *thread, void **thread_return);`
  Routine zum Warten auf die Beendigung eines gestarteten Threads.

- `pthread_t pthread_self();`
  Ist die PThreads-Analogie zu `int getid();`

2.2 Sourcecodebeispiel

```
1  #include <pthread.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5
6  void *printNum(void *num){
7      for(;;){}
8      return NULL;
9  }
10
11 int main(int argc, char ** argv){
12     int i;
13     pthread_t thread[8];
14
```

```
15      /* creation of eight threads */
16      for(i=0; i<8; i++){
17          /* first NULL means that we want to use the standard attributes for
                our threads
18              second NULL means that we do not need parameters */
19          pthread_create(thread+i, NULL, printNum, (void *) NULL);
20      }
21
22      /* Threads won't ever return */
23      for(i=0; i<8; i++){
24          pthread_join(thread[i], NULL);
25      }
26
27      return EXIT_SUCCESS;
28  }
```

2.3 Wechselseitiger Ausschluss (Mutual Exclusion)

2.3.1 Funktionen

Pthreads stellt einen Mechanisumus für Mutexe bereit. Die entsprechenden Funktionen sind mit pthreads_mutex geprefixt. Die wichtigesten Funktionen sind:

- `int pthread_mutex_init(pthreads_mutex_t *mutex, pthread_mutexattr_t *attr);`
  Initialisert einen Mutex, attr kann dabei zur genaueren Konfiguration verwendet werden. Setzen von attr = NULL, wenn die Standardwerte genügen.

- `int pthread_mutex_destroy(pthread_mutex_t *mutex);`
  Entfernt einen erstellten Mutex aus dem System. Gibt 0 zurück, wenn der Aufruf erfolgreich war, ansosnten wird ein Errorcode zurückgegeben. Mögliche Codes sind z.B.
  - EBUSY: Mutex ist noch nicht wieder freigegeben.
  - EINVAL: Es wurde eine Adresse übergeben, die keinen gültigen Mutex referenziert.
  - ...

- `int pthread_mutex_trylock(pthread_mutex_t *m);`
  Versucht den Mutex zu schließen. Gibt 0 zurück, wenn der Mutex erfolgreich geschlossen werden konnte, ansosnten wird ein Error-Code zurückgegeben. Wichtige Errorcodes sind:
  - EINVAL: Der Pointer auf den Mutex ist ungültig
  - EBUSY: Mutex konnte nicht geschlossen werden, da er zum Zeitpunkt des Funktionsaufrufs schon geschlossen war

- `int pthread_mutex_timedlock(pthread_mutex_t *restrict m, const struct timespec *restrict timeout);`
  Verhält sich wie pthread_mutex_lock(...), mit dem Unterschied, dass nur timeout Zeitheinheiten blockiert wird. Wenn es nicht möglich war den Lock zu schließen, wird ETIMEDOUT zurückgegeben.

- `int pthread_mutex_lock(pthread_utext_t *restrict m);`
  Schließt einen Mutex. Wenn er bereits geschlossen war, blockiert die Funktion solange, bis der Thread, der den Mutex hält, diesen wieder freigibt. Auch hier wird 0 bei Erfolg und ein Errorcode bei einem Fehler zurückgegeben. Errorcodes sind z.B.
  - EINVAL: Falsche Parameter

- `int pthread_mutex_unlock(pthread_mutex_t *m);`
  Gibt einen geschlossenen Mutex wieder frei

- `int pthread_mutexattr_init(pthread_mutex_t *m);`
  Initialisiert ein Attributobjekt vom Typ pthread_mutexattr_t

- `int pthread_mutexattr_destroy(pthread_mutex_t *m);`
  Zerstört ein Mutexattributobjekt

2.3.2 Sourcecodebeispiel für die Synchonisierung mithilfe von Mutexen

```
1  /* Creates one(!) file pointer every thread has access to.
2      Synchronisation happens with the mutex sync which is initialized
3      dynamically in the main function. */
4  #include <pthread.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7
8  #define FPATH "resource"
9
10 /* parameters */
11 struct f {
12     FILE *fptr;
13     pthread_mutex_t *mut;
14     int tid;
15 };
16
17 /* routine executed by threads */
18 void *writeSth(void *data){
19     struct f temp = *((struct f *) data);
20     pthread_mutex_lock(temp.mut);
21     fprintf(temp.fptr, "Ich␣bin␣der␣Thread␣nummer:␣%d\n", temp.tid);
22     pthread_mutex_unlock(temp.mut);
23     return NULL;
24 }
25
26 int main(int argc, char ** argv){
27     int i;
28     struct f data[8];
29     FILE *file = fopen(FPATH, "w+");
30     pthread_t thread[8];
31
32     /* create the mutex */
33     pthread_mutex_t sync;
34     pthread_mutex_init(&sync, NULL);
35
36     /* starts threads and passes arguments */
37     for(i=0; i<8; i++){
38         data[i].fptr = file;
39         data[i].tid = i;
40         data[i].mut = &sync;
41
42         pthread_create(thread+i, NULL, writeSth, (void *) (data+i));
43     }
44
45     /* waits for all threads */
46     for(i=0; i<8; i++){
47         pthread_join(thread[i], NULL);
48     }
49
50     /* free the resources taken by the mutex */
51     pthread_mutex_destroy(&sync);
52     fclose(file);
```

```
53      return EXIT_SUCCESS;
54  }
```

2.4 Bedingungsvariablen

2.4.1 Funktionen für Bedingungsvariablen

Locks verhalten sich wie Semaphore, die mit dem Wert eins initialisiert wurden. Nur ein Thread kann bei dieser Art der Synchronsiation gleichzeitig in den kritischen Abschnitt. Bei komplexeren Problemen wie z.B. dem Erzeuger-Verbraucher-Problem ist es jedoch notwendig bis zu n Threads gleichzeitig in einem zu synchronisierenden Abschnitt arbeiten zu lassen. Dies wird durch die Kombination von Mutexen und Bedingungsvariablen ermöglicht. Bei der Verwendung geht man nach den folgenden Schritten vor:

1. Initialisierung von globalem Speicher, dessen Bearbeitung synchronisiert werden muss.
2. Initialisierung einer Bedingungsvariable. (Umgesetzt als Objekt der Pthreads-Bibliothek)
3. Initialisierung eines Mutexes mit Bezug auf die Bedingungsvariable Erstellung der Threads.

Pthreads bietet folgende Funktionen für die Erstellung, Bearbeitung und Vernichtung von Bedingungsvariablen:

- `int pthreads_cond_init(pthread_cond_t *c, const pthread_condattr_t *restrict a)`
  Initialisert ein zuvor deklariertes Objekt vom Typ pthread_cond. Dabei kann ein Attribut übergeben. Die Initialisierung der Attribute erfolgt wie weiter unten beschrieben.
- `int pthread_cond_destroy(pthread_cond_t *c)`
  Gibt die von c benötigten Ressourcen wieder frei.
- `int pthread_condattr_init(pthread_condattr_t *c);`
  Initialisert ein zuvor deklariertes Objekt vom Typ pthread_condattr_t. Für Bedingungsvariablen ist nur das Attribut "process-shared" definiert. Wird eine Bedingungsvariable mit diesem Attribut initialisiert, so ist sie Prozessübergreifend sichtbar.
- `int pthread_condattr_destroy(pthread_condattr_t *c);`
  Entfernt ein Bedingungsvariablenattribut.
- `int pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);`
  Diese Routine sollte nur aufgerufen werden, wenn Mutex m gesperrt ist. Beim Aufruf wird der Mutex automatisch frei und der aktuelle Thread blockiert. Die Blockade hält solange an, bis das Signal (Pro Instanz einer Bedingungsvariable gibt es eine Signalquelle, die mit ihr assoziiert ist) empfangen wird. Der Mutex m wird, nachdem die Funktion zurückgegeben hat, automatisch gesperrt.
- `int pthread_cond_signal(pthread_cond_t*c);`
  Wird verwendet, um einen gerade mit `pthread_cond_wait(c,m);` wartenden Thread aufzuwecken. Die Funktion sollte aufgerufen werden, wenn der Mutex m gesperrt ist. Außerdem muss nach dem Aufruf zusätzlich der Mutex entsperrt werden.
- `int pthread_cond_broadcast(pthread_cont_t *c);`
  Wird wie die vorherige Funktion verwendet, mit der Ausnahme, dass nun beliebig viele Wartende Threads angesprochen werden können.