Kapitel 12
Threads

Neben den Prozessen existiert noch eine andere Form der Programmausführung, die Linux unterstützt – die Threads, die auch als »leichtgewichtige« Prozesse bekannt sind.

Mit der Thread-Programmierung können Sie Anwendungen schreiben, die erheblich schneller und parallel ablaufen, und zwar gleichzeitig auf verschiedenen Prozessorkernen eines Computers. In diesem Kapitel erhalten Sie einen Einblick in die Thread-Programmierung unter Linux und erfahren, wie Sie diese Kenntnisse in der Praxis einsetzen können.

Hinweis

Die Beispiele im Buch verwenden *Pthreads* und sind kompatibel zum plattformübergreifenden POSIX-Standard. Früher gab es noch andere verbreitete Standards, darunter die sogenannten »Linux-Threads«, die BSD-Threads, aber auch mehr kooperative Ansätze, wie die GNU Portable Threads. Diese Ansätze waren zum Teil völlig unterschiedlich und werden auch heute manchmal noch verwendet, gelten aber als überholt.

Dennoch bleibt es Ihnen nicht erspart, Ihre Programme etwas anders zu übersetzen, und zwar mit diesem Compileraufruf:

```
$ gcc -o thserver thserver.c  -pthreads
```

Der zusätzliche Parameter `-pthreads` setzt gleichzeitig verschiedene Präprozessoreinstellungen und fügt die Pthreads-Bibliothek der Linkliste hinzu. Und wenn Sie in der Literatur einmal `-lpthreads` sehen: Das umfasst dann nur die Änderung in der Linkliste. Beachten Sie bitte, dass nicht unbedingt jede Plattform Pthreads unterstützt.

12.1 Unterschiede zwischen Threads und Prozessen

Prozesse haben wir ja bereits ausführlich erklärt. Sie wissen, wie Sie eigene Prozesse mittels `fork()` erstellen können, und in Kapitel 11 zur Interprozesskommunikation (IPC) haben Sie erfahren, wie man einzelne Prozesse synchronisiert. Der Aufwand, den Sie bei der Interprozesskommunikation getrieben haben, entfällt bei den Threads fast komplett.

Ein weiterer Nachteil bei der Erstellung von Prozessen gegenüber Threads ist der enorme Aufwand, der mit der Duplizierung des Namensraums einhergeht – mit den Threads haben Sie ihn nicht, da Threads in einem gemeinsamen Adressraum ablaufen. Somit stehen den einzelnen Threads dasselbe Codesegment, dasselbe Datensegment, der Heap und alle anderen Zustandsdaten zur Verfügung, die ein »gewöhnlicher« Prozess besitzt – was somit auch die Arbeit beim Austausch von Daten und bei der Kommunikation untereinander erheblich erleichtert. Weil aber kein Speicherschutzmechanismus zwischen den Threads vorhanden ist, bedeutet dies auch: Wenn ein Thread abstürzt, reißt er alle anderen Threads mit.

Im ersten Moment besteht somit vorerst gar kein Unterschied zwischen einem Prozess und einem Thread, denn letztendlich besteht ein Prozess mindestens aus einem Thread. Ferner endet ein Prozess, wenn sich alle Threads beenden. Somit ist der *eine* Prozess (dieser eine Prozess ist der erste Thread, auch *Main Thread* bzw. Haupt-Thread genannt) verantwortlich für die gleichzeitige Ausführung mehrerer Threads – da doch Threads auch nur innerhalb eines Prozesses ausgeführt werden. Der gravierende Unterschied zwischen den Threads und den Prozessen besteht darin, dass Threads unabhängige Befehlsfolgen innerhalb eines Prozesses sind. Man könnte auch sagen, Threads sind in einem Prozess gefangen oder verkapselt – im goldenen Käfig eingeschlossen.

Natürlich müssen Sie dabei immer Folgendes im Auge behalten: Wenn Threads denselben Adressraum verwenden, teilen sich alle Threads den statischen Speicher und somit auch die globalen Variablen miteinander. Ebenso sieht dies mit den geöffneten Dateien (z. B. Filedeskriptoren), Signalhandlern- und Einstellungen, der Benutzer- und der Gruppenkennung und dem Arbeitsverzeichnis aus. Daher sind auch in der Thread-Programmierung gewisse Synchronisationsmechanismen nötig und auch vorhanden.

12.2 Scheduling und Zustände von Threads

Auch bei der Thread-Programmierung ist (wie bei den Prozessen) der Scheduler im Betriebssystem aktiv, der bestimmt, wann welcher Thread Prozessorzeit erhält. Auch die E/A-Bandbreite wird durch einen Scheduler zugeteilt. Die Zuteilung kann wie schon bei den Prozessen prioritäts- und zeitgesteuert erfolgen, sich aber andererseits am Ressourcenverbrauch und an den Geräten orientieren, auf die gewartet wird. (Eine Maus sendet vielleicht nicht viele Zeichen, aber der Thread sollte nicht erst umständlich »aufwachen« müssen, wenn von dort Input kommt.)

Bei zeitgesteuerten Threads, also *Timesharing Threads*, bedeutet dies, dass jedem Thread eine bestimmte Zeit (des Prozessors oder der Prozessoren) zur Verfügung steht, ehe dieser automatisch unterbrochen wird und anschließend ein anderer Thread an der Reihe ist (präemptives Multitasking).

Hinweis

Linux unterstützt seit Jahren verschiedene Scheduler. Häufig will man auf Entwicklercomputern einen anderen Scheduler als auf den Servercomputern haben, da sich die grafische Oberfläche sonst »laggy« anfühlt.[1]

Je höher seine Priorität ist, desto mehr Rechenzeit bzw. E/A-Bandbreite bekommt ein Prozess (Thread) zugewiesen. Neben normalen Prioritäten gibt es auch noch besondere Prioritätsklassen für sogenannte Echtzeitanwendungen. Diese werden im Extremfall kooperativ »geschedult«. Das heißt, wenn der Prozess nicht freiwillig zum Kernel zurückkehrt, zum Beispiel indem er eine I/O-Operation anfordert, bekommen andere Prozesse keine Rechenzeit. Die Maschine steht dann theoretisch wie eingefroren. Unter Umständen kann es auch fatal ausgehen, wenn der Prozess höher priorisiert ist als die E/A-Prozesse des Kernels.

Anhand von Abbildung 12.1 können Sie die Zustände erkennen, in denen sich ein Thread befinden kann. Bei genauerer Betrachtung fällt auf, dass sich die Threads – abgesehen von den weiteren Unterzuständen – nicht wesentlich von den Prozessen unterscheiden.

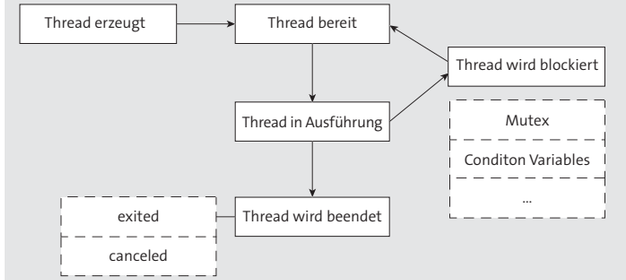

Abbildung 12.1 Zustände von Threads

- *bereit* – Der Thread wartet, bis ihm Prozessorzeit zur Verfügung steht, um seine Arbeit auszuführen.
- *ausgeführt* – Der Thread wird im Augenblick ausgeführt. Bei Multiprozessorsystemen können hierbei mehrere Threads gleichzeitig ausgeführt werden (pro CPU ein Thread).
- *wartet* – Der Thread wird im Augenblick blockiert und wartet auf einen bestimmten Zustand (z. B. Bedingungsvariable, Mutex-Freigabe etc).
- *beendet* – Ein Thread hat sich beendet oder wurde abgebrochen.

1 »Laggy« soll in diesem Zusammenhang hohe Latenz bedeuten, nicht unbedingt niedrige Geschwindigkeit. (Computerspieler wissen, was damit gemeint ist.)

12.3 Die grundlegenden Funktionen zur Thread-Programmierung

Hinweis

Einen Hinweis gleich zu Beginn der Thread-Programmierung – alle Funktionen aus der `pthread`-Bibliothek geben bei Erfolg `0`, ansonsten bei einem Fehler `-1` zurück.

12.3.1 pthread_create – einen neuen Thread erzeugen

Einen neuen Thread kann man mit der Funktion `pthread_create` erzeugen:

```
#include <pthread.h>

int pthread_create( pthread_t *thread,
                    const pthread_attr_t *attribute,
                    void *(*funktion)(void *),
                    void *argumente );
```

Wenn Sie die Funktion betrachten, dürfte Ihnen die Ähnlichkeit mit der Funktion `clone()` auffallen (siehe Manual Page), worauf sich `pthread_create()` unter Linux ja auch beruft. Jeder Thread bekommt eine eigene Identifikationsnummer (ID) vom Datentyp `pthread_t`, die in der Variablen des ersten Parameters `thread` abgelegt wird. Anhand dieser ID werden alle anderen Threads voneinander unterschieden.

Mit dem zweiten Parameter `attribute` können Sie bei dem neu zu startenden Thread Attribute setzen, z. B. die Priorität, die Stackgröße und noch einige mehr. Auf die einzelnen Attribute werden wir noch eingehen. Geben Sie hierfür `NULL` an, werden die Standardattribute für den Thread vorgenommen.

Mit dem dritten Parameter geben Sie die »Funktion« für einen Thread selbst an. Hierbei geben Sie die Anfangsadresse einer Routine an, die der Thread verwenden soll. Wenn sich die hier angegebene Funktion beendet, bedeutet dies auch automatisch das Ende des Threads.

Argumente, die Sie dem Thread mitgeben wollen, können Sie mit dem vierten Parameter, `argumente`, übergeben. Meistens wird dieser Parameter verwendet, um Daten an Threads zu übergeben. Hierzu wird in der Praxis häufig die Adresse einer Strukturvariablen herangezogen.

Nach dem Aufruf von `pthread_create()` kehrt diese Funktion sofort wieder zurück und fährt mit der Ausführung hinter `pthread_create()` fort. Der neu erzeugte Thread führt sofort asynchron seine Arbeit aus. Jetzt würden praktisch zwei Threads gleichzeitig ausgeführt: der Haupt-Thread und der neue Thread, der vom Haupt-Thread mit `pthread_create()` erzeugt wurde. Welcher der beiden Threads hierbei zunächst mit seiner Ausführung beginnt, ist nicht festgelegt. (Das ist dasselbe Verhalten wie bei `fork`.)

12.3.2 pthread_exit – einen Thread beenden

Beenden können Sie einen Thread auf unterschiedliche Weise. Meistens werden Threads mit der Funktion `pthread_exit()` beendet:

```
#include <pthread.h>

void pthread_exit( void * wert );
```

Diese Funktion beendet nur den Thread, indem Sie diese aufrufen. Mit dem Argument `wert` geben Sie den Exit-Status des Threads an. Diesen Status können Sie mit `pthread_join()` ermitteln (wie das geht, folgt in Kürze). Natürlich darf auch hierbei, wie eben C-üblich, der Rückgabewert kein lokales Speicherobjekt vom Thread sein, da dieses (wie eben bei Funktionen auch) nach der Beendigung des Threads nicht mehr gültig ist.

Neben der Möglichkeit, einen Thread mit `pthread_exit()` zu beenden, sind noch folgende Dinge zu beachten:

- Ruft ein beliebiger Thread die Funktion `exit()` auf, werden alle Threads, einschließlich des Haupt-Threads, beendet (also das komplette Programm). Genauso sieht dies aus, wenn Sie dem Prozess das Signal `SIGTERM` oder `SIGKILL` senden.
- Ein Thread, der mittels `pthread_create()` erzeugt wurde, lässt sich auch mit `return` **`[wert]`** beenden. Dies entspricht exakt dem Verhalten von `pthread_exit()`. Der Rückgabewert kann hierbei ebenfalls mit `pthread_join()` ermittelt werden.

Exit-Handler für Threads einrichten

Wenn Sie einen Thread beenden, können Sie auch einen Exit-Handler einrichten. Dies wird in der Praxis recht gern verwendet, um z. B. temporäre Dateien zu löschen, Mutexe freizugeben oder eben sonstige »Reinigungsarbeiten« durchzuführen. Ein solcher eingerichteter Exit-Handler wird dann automatisch bei Beendigung eines Threads mit z. B. `pthread_exit()` oder `return` ausgeführt. Das Prinzip ist ähnlich, wie Sie es von der Standardbibliotheksfunktion `atexit()` kennen sollten. Auch in diesem Fall werden bei mehreren Exit-Handlern die einzelnen Funktionen in umgekehrter Reihenfolge (da Stack) der Einrichtung ausgeführt.

Hier sehen Sie die Funktionen dazu:

```
#include <pthread.h>

void pthread_cleanup_push( void (*function)(void *),
                           void *arg );
void pthread_cleanup_pop( int exec );
```

Eine solche Funktion richten Sie also mit der Funktion `pthread_cleanup_push()` ein. Als ersten Parameter übergeben Sie dabei die Funktion, die ausführt werden soll, und als zweiten Parameter die Argumente für den Exit-Handler (falls nötig). Den zuletzt eingerichteten Exit-

Handler können Sie wieder mit der Funktion `pthread_cleanup_pop()` vom Stack entfernen. Geben Sie allerdings einen Wert ungleich `0` als Parameter `exec` an, so wird diese Funktion zuvor noch ausgeführt, was bei einer Angabe von `0` nicht gemacht wird.

Ein Umstand, der uns schon zur Weißglut gebracht hat, ist die Tatsache, dass die beiden Funktionen `pthread_cleanup_push()` und `pthread_cleanup_pop()` als Makros implementiert sind. Das wäre nicht so schlimm, wenn `pthread_cleanup_push()` eine sich öffnende geschweifte Klammer enthalten würde und `pthread_cleanup_pop()` eine sich schließende geschweifte Klammer. Das bedeutet: Sie müssen beide Funktionen im selben Anweisungsblock ausführen. Daher müssen Sie immer ein `_push` und ein `_pop` verwenden, auch wenn Sie wissen, dass eine `_pop`-Stelle nie erreicht wird.

12.3.3 pthread_join – auf das Ende eines Threads warten

Bevor Sie sich dem ersten Listing widmen können, benötigen Sie noch Kenntnisse zur Funktion `pthread_join()`:

```
#include <pthread.h>

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

`pthread_join()` hält den aufrufenden Thread (meistens den Haupt-Thread), der einen Thread mit `pthread_create()` erzeugt hat, so lange an, bis der Thread mit der ID `thread` vom Typ `pthread_t` beendet wurde. Der Exit-Status (bzw. Rückgabewert) des Threads wird an die Adresse von `thread_return` geschrieben. Sind Sie nicht am Rückgabewert interessiert, können Sie hier auch `NULL` verwenden. `pthread_join()` ist also das, was Sie bei den Prozessen mit `waitpid()` kennen.

Ein Thread, der sich beendet, wird eben so lange nicht »freigegeben« bzw. als beendeter Thread anerkannt, bis ein anderer Thread `pthread_join()` aufruft. Diesen Zusammenhang könnten Sie bei den Prozessen mit Zombie-Prozessen vergleichen. Daher sollten Sie für jeden erzeugten Thread einmal `pthread_join()` aufrufen, es sei denn, Sie haben einen Thread »abgehängt« (dazu folgt in Kürze mehr).

12.3.4 pthread_self – die ID von Threads ermitteln

Die Identifikationsnummer eines auszuführenden Threads können Sie mit der Funktion `pthread_self()` erfragen:

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Als Rückgabewert erhalten Sie die Thread-ID vom Datentyp `pthread_t`.

Hierzu erstellen wir nun ein einfaches Beispiel, das alle bisher vorgestellten Funktionen in klarer Weise demonstrieren soll:

```
/* thread1.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* insg. MAX_THREADS Threads erzeugen */
#define MAX_THREADS 3
#define BUF 255

/* Einfache Daten für die Wertübergabe an den Thread */
struct data {
   int wert;
   char msg[BUF];
};

/* Ein einfacher Exit-Handler für Threads, der  *
 * pthread_cleanup_push und pthread_cleanup_pop *
 * in der Praxis demonstrieren soll             */
static void exit_handler_mem( void * arg ) {
   printf("\tExit-Handler aufgerufen ...");
   struct data *mem =  (struct data *)arg;
   /* Speicher freigeben */
   free(mem);
   printf("Speicher freigegeben\n");
}

/* Die Thread-Funktion */
static void mythread (void *arg) {
   struct data *f = (struct data *)arg;
   /* Exit-Handler einrichten - wird automatisch nach *
    * pthread_exit oder Thread-Ende aufgerufen        */
   pthread_cleanup_push( exit_handler_mem, (void*)f );
   /* Daten ausgeben */
   printf("\t-> Thread mit ID:%ld gestartet\n",
      pthread_self());
   printf("\tDaten empfangen: \n");
   printf("\t\twert = \"%d\"\n", f->wert);
   printf("\t\tmsg  = \"%s\"\n", f->msg);
   /* Den Exit-Handler entfernen, aber trotzdem ausführen, *
    * da als Angabe 1 anstatt 0 verwendet wurde            */
   pthread_cleanup_pop( 1 );
```

```
   /* Thread beenden - Als Rückgabewert Thread-ID verwenden.
    * Alternativ kann hierfür auch:
    * return(void) pthread_self();
    * verwendet werden */
   pthread_exit((void *)pthread_self());
}

int main (void) {
   pthread_t th[MAX_THREADS];
   struct data *f;
   int i;
   static int ret[MAX_THREADS];
   /* Haupt-Thread gestartet */
   printf("\n-> Main-Thread gestartet (ID:%ld)\n",
      pthread_self());
   /* MAX_THREADS erzeugen */
   for (i = 0; i < MAX_THREADS; i++) {
      /* Speicher für Daten anfordern und mit Werten belegen*/
      f = (struct data *)malloc(sizeof(struct data));
      if(f == NULL) {
         printf("Konnte keinen Speicher reservieren ...!\n");
         exit(EXIT_FAILURE);
      }
      /* Zufallszahl zwischen 1 und 10 (Spezial) */
      f->wert =  1+(int) (10.0*rand()/(RAND_MAX+1.0));
      snprintf (f->msg, BUF, "Ich bin Thread Nr. %d", i+1);
      /* Jetzt Thread erzeugen */
      if(pthread_create(&th[i], NULL, &mythread, f) != 0) {
         fprintf (stderr, "Konnte Thread nicht erzeugen\n");
         exit (EXIT_FAILURE);
      }
   }
   /* Auf das Ende der Threads warten */
   for( i=0; i < MAX_THREADS; i++)
      pthread_join(th[i], &ret[i]);
   /* Rückgabewert der Threads ausgeben */
   for( i=0; i < MAX_THREADS; i++)
       printf("<-Thread %ld ist fertig\n", ret[i]);
   /* Haupt-Thread ist jetzt auch fertig */
   printf("<- Main-Thread beendet (ID:%ld)\n",
      pthread_self());
   return EXIT_SUCCESS;
}
```


Das Programm bei der Ausführung:

```
$ gcc -o thread1 thread1.c -pthread
$ ./thread1

-> Main-Thread gestartet (ID:-1209412512)
        -> Thread mit ID:-1209414736 gestartet
        Daten empfangen:
                wert = "9"
                msg  = "Ich bin Thread Nr. 1"
        Exit-Handler aufgerufen ... Speicher freigegeben
        -> Thread mit ID:-1217807440 gestartet
        Daten empfangen:
                wert = "4"
                msg  = "Ich bin Thread Nr. 2"
        Exit-Handler aufgerufen ... Speicher freigegeben
        -> Thread mit ID:-1226200144 gestartet
        Daten empfangen:
                wert = "8"
                msg  = "Ich bin Thread Nr. 3"
        Exit-Handler aufgerufen ...Speicher freigegeben
<-Thread -1209414736 ist fertig
<-Thread -1217807440 ist fertig
<-Thread -1226200144 ist fertig
<- Main-Thread beendet (ID:-1209412512)
```

Hinweis am Rande

Bevor sich jemand über die Warnmeldung des Compilers wundert, noch ein Satz zum Casten von `void *`: In der Programmiersprache C ist ein Casten von oder nach `void *` nicht nötig. Aber wenn die Warnmeldung Sie stört, können Sie dies gern trotzdem nachholen.

Dieses Beispiel demonstriert auch auf einfache Weise, wie Sie Daten an einen neu erzeugten Thread übergeben können (hier mit der Struktur `data`). Ebenfalls gezeigt wurde hier die Verwendung eines Exit-Handlers, der nur den im Haupt-Thread angeforderten Speicherbereich freigibt. Zugegeben, das ließe sich auch im Thread `mythread` einfacher realisieren, aber zu Anschauungszwecken sind solch einfache Codebeispiele immer noch am besten. Im Beispiel wurden außerdem drei Threads aus der Funktion `mythread` erzeugt, die im Prinzip alle dasselbe machen, nämlich eine einfache Ausgabe der Daten, die an die Threads übergeben wurden.

Hierbei müssen wir nochmals explizit darauf hinweisen, dass die Ausführung, in welcher Reihenfolge die Threads starten, nicht vorgegeben ist, auch wenn dies hier einen anderen

Anschein erweckt. Dazu werden Synchronisationsmechanismen erforderlich. Jeder Thread wurde hier mit `pthread_exit()` und der eigenen Thread-ID als Rückgabewert beendet. Genauso gut kann dies natürlich auch mit `return` gemacht werden. Der Rückgabewert von den einzelnen Threads wird im Haupt-Thread von `pthread_join()` erwartet und ausgegeben. Der Haupt-Thread beendet sich am Ende erst, wenn alle Threads fertig sind.

Würden Sie in diesem Beispiel `pthread_join()` weglassen, so würde sich der Haupt-Thread noch vor den anderen Threads beenden. Dies bedeutet, dass alle anderen Threads zwar noch laufen, aber auf nun nicht mehr gültige Strukturvariablen zugreifen würden.

Rückgabewert von Threads

Zwar sind wir schon auf den Rückgabewert von Threads eingegangen, aber hierbei wurden nur Thread-spezifische Daten zurückgegeben (hier die Thread-ID). Aber genauso wie schon bei der Wertübergabe an Threads können Sie hierbei auch ganze Strukturen zurückgeben, was in der Praxis auch häufig so der Fall ist.

Dazu zeigen wir ein ähnliches Beispiel wie schon *thread1.c*, nur dass jetzt die Daten der Struktur aus dem Thread zurückgegeben und im Haupt-Thread mit `pthread_join()` »abgefangen« und anschließend ausgegeben werden. Auf die Verwendung eines Exit-Handlers haben wir der Übersichtlichkeit zuliebe verzichtet.

```
/* thread2.c */
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* insg. MAX_THREADS Threads erzeugen */
#define MAX_THREADS 3
#define BUF 255

/* Einfache Daten für die Wertübergabe an den Thread */
struct data {
   int wert;
   char msg[BUF];
};

/* Die Thread-Funktion */
static void *mythread (void *arg) {
   struct data *f= (struct data *)arg;
   /* Zufallszahl zwischen 1 und 10 (Spezial) */
   f->wert =  1+(int) (10.0*rand()/(RAND_MAX+1.0));
   snprintf (f->msg, BUF, "Ich bin Thread Nr. %ld",
      pthread_self());
   /* Thread beenden - Als Rückgabewert Strukturdaten
```



```
    * verwenden - Alternativ auch pthread_exit( f );  */
   return arg;
}

int main (void) {
   pthread_t th[MAX_THREADS];
   int i;
   struct data *ret[MAX_THREADS];

   /* Haupt-Thread gestartet */
   printf("\n-> Main-Thread gestartet (ID:%ld)\n",
      pthread_self());
   /* Speicher reservieren */
   for (i = 0; i < MAX_THREADS; i++){
      ret[i] = (struct data *)malloc(sizeof(struct data));
      if(ret[i] == NULL) {
         printf("Konnte keinen Speicher reservieren ...!\n");
         exit(EXIT_FAILURE);
      }
   }
   /* MAX_THREADS erzeugen */
   for (i = 0; i < MAX_THREADS; i++) {
      /* Jetzt Thread erzeugen */
      if(pthread_create(&th[i],NULL,&mythread,ret[i]) !=0) {
         fprintf (stderr, "Konnte Thread nicht erzeugen\n");
         exit (EXIT_FAILURE);
      }
   }
   /* Auf das Ende der Threads warten */
   for( i=0; i < MAX_THREADS; i++)
      pthread_join(th[i], (void **)&ret[i]);

   /* Daten ausgeben */
   for( i=0; i < MAX_THREADS; i++) {
      printf("Main-Thread: Daten empfangen: \n");
      printf("\t\twert = \"%d\"\n", ret[i]->wert);
      printf("\t\tmsg  = \"%s\"\n", ret[i]->msg);
   }
   /* Haupt-Thread ist jetzt auch fertig */
   printf("<- Main-Thread beendet (ID:%ld)\n",
      pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread2 thread2.c -pthread
$ ./thread2

-> Main-Thread gestartet (ID:-1209412512)
Main-Thread: Daten empfangen:
                wert = "9"
                msg  = "Ich bin Thread Nr. -1209414736"
Main-Thread: Daten empfangen:
                wert = "4"
                msg  = "Ich bin Thread Nr. -1217807440"
Main-Thread: Daten empfangen:
                wert = "8"
                msg  = "Ich bin Thread Nr. -1226200144"
<- Main-Thread beendet (ID:-1209412512)
```

12.3.5 pthread_equal – die ID von zwei Threads vergleichen

Um einen Thread mit einem anderen Thread zu vergleichen, können Sie die Funktion `pthread_equal()` verwenden. Dies wird häufig getan, um sicherzugehen, dass nicht ein Thread gleich einem anderen ist. Ein Wert ungleich `0` wird zurückgegeben, wenn beide Threads gleich sind; und `0` wird zurückgegeben, wenn die Threads eine unterschiedliche Identifikationsnummer (ID) besitzen.

Das folgende Beispiel erzeugt drei Threads mit derselben »Funktion«. Hierbei soll jeder Thread wiederum eine andere Aktion ausführen. Im Beispiel ist die Aktion zwar nur die Ausgabe eines Textes, aber in der Praxis könnten Sie hierbei neue Funktionen aufrufen. Für die ersten drei Threads wird jeweils eine bestimmte Aktion festgelegt. Alle anderen Threads führen nur noch die `else`-Aktion aus. Dies ist beispielsweise sinnvoll, wenn Sie in Ihrer Anwendung Vorbereitungen treffen wollen (im Beispiel eben drei Vorbereitungen) wie »Dateien anlegen«, »Müll beseitigen«, »eine Server-Verbindung herstellen« und noch vieles mehr.

Sind diese Vorbereitungen getroffen, wird immer mit der gleichen Funktion fortgefahren. Damit der Vergleich von Threads mit `pthread_equal()` auch funktioniert, wurden die Thread-IDs, die beim Anlegen mit `pthread_create()` erzeugt worden sind, in globale Variablen gespeichert – und sind daher auch für alle Threads »sichtbar«.

Hier sehen Sie das Beispiel, dessen Ausgabe auch einiges erklärt.

```
/* thread3.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```



```
#include <pthread.h>
#define MAX_THREADS 5
#define BUF 255

/* Globale Variable mit Thread-IDs *
 * für alle Threads sichtbar       */
static pthread_t th[MAX_THREADS];

static void aktion(void *name) {
   while( 1 ) {
      if(pthread_equal(pthread_self(),th[0])) {
         printf("\t->(%ld): Aufgabe \"abc\" Ausführen \n",
            pthread_self());
         break;
      }
      else if(pthread_equal(pthread_self(),th[1])) {
         printf("\t->(%ld): Aufgabe \"efg\" Ausführen \n",
            pthread_self());
         break;
      }
      else if(pthread_equal(pthread_self(),th[2])) {
         printf("\t->(%ld): Aufgabe \"jkl\" Ausführen \n",
            pthread_self());
         break;
      }
      else {
         printf("\t->(%ld): Aufgabe \"xyz\" Ausführen \n",
            pthread_self());
         break;
      }
   }
   pthread_exit((void *)pthread_self());
}

int main (void) {
   int i;
   static int ret[MAX_THREADS];

   printf("->Haupt-Thread (ID:%ld) gestartet...\n",
      pthread_self());
   /* Threads erzeugen */
   for (i = 0; i < MAX_THREADS; i++) {
      if (pthread_create (&th[i],NULL,&aktion,NULL) != 0) {
```

```
            printf ("Konnte keinen Thread erzeugen\n");
            exit (EXIT_FAILURE);
         }
      }
      /* Auf die Threads warten */
      for (i = 0; i < MAX_THREADS; i++)
         pthread_join (th[i], &ret[i]);
      /* Rückgabe der Threads auswerten */
      for (i = 0; i < MAX_THREADS; i++)
         printf("\t<-Thread %ld mit Arbeit fertig\n", ret[i]);
      printf("->Haupt-Thread (ID:%ld) fertig ...\n",
         pthread_self());
      return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread3 thread3.c -pthread
$ ./thread3
->Haupt-Thread (ID:-1209412512) gestartet...
        ->(-1209414736): Aufgabe "abc" Ausführen
        ->(-1217807440): Aufgabe "efg" Ausführen
        ->(-1226200144): Aufgabe "jkl" Ausführen
        ->(-1234592848): Aufgabe "xyz" Ausführen
        ->(-1242985552): Aufgabe "xyz" Ausführen
        <-Thread -1209414736 mit Arbeit fertig
        <-Thread -1217807440 mit Arbeit fertig
        <-Thread -1226200144 mit Arbeit fertig
        <-Thread -1234592848 mit Arbeit fertig
        <-Thread -1242985552 mit Arbeit fertig
->Haupt-Thread (ID:-1209412512) fertig ...
```

Sie können daran erkennen, dass die ersten drei Threads jeweils »abc«, »efg« und »jkl« ausführen. Alle noch folgenden Threads führen dann »xyz« aus. Zugegeben, das lässt sich eleganter mit den Synchronisationsmechanismen der Thread-Bibliothek lösen, aber das Beispiel demonstriert den Sachverhalt der Funktion `pthread_equal()` recht gut.

12.3.6 pthread_detach – einen Thread unabhängig machen

Die Funktion `pthread_detach()` stellt das Gegenteil von `pthread_join()` dar. Mit ihr legen Sie fest, dass nicht mehr auf die Beendigung des Threads gewartet werden soll:


```
#include <pthread.h>

int pthread_detach( pthread_t thread );
```

Sie lösen hiermit praktisch den Thread mit der ID `thread` von der Hauptanwendung los. Sie können diesen Vorgang gern mit den Daemon-Prozessen vergleichen. Dass dieser Thread dann selbstständig ist, hat nichts Magisches an sich: Im Grunde »markieren« Sie den Thread damit nur, sodass bei seinem Beenden der Exit-Status und die Thread-ID gleich freigegeben werden. Ohne `pthread_detach()` würde dies erst nach einem `pthread_join`-Aufruf der Fall sein. Natürlich bedeutet die Verwendung von `pthread_detach()`, dass auch kein `pthread_join()` mehr auf das Ende des Threads reagiert.

> **Hinweis**
>
> Ein Thread, der mit `pthread_detach()` oder mit dem Attribut `PTHREAD_CREATE_DETACHED` von den anderen Threads losgelöst wurde, kann nicht mehr mit `pthread_join()` abgefangen werden. Der Thread läuft praktisch ohne äußere Kontrolle weiter.

Ein typischer Codeausschnitt, der zeigt, wie Sie einen Thread von den anderen loslösen können, sieht so aus:

```
pthread_t a_thread;
int ret;
...
/* Einen neuen Thread erzeugen */
ret = pthread_create( &a_thread, NULL, thread_function, NULL);
/* bei Erfolg den Thread abhängen ... */
if (ret == 0) {
   pthread_detach(a_thread);
}
```

12.4 Die Attribute von Threads und das Scheduling

Wie Sie bereits im vorigen Abschnitt erfahren haben, kann man auch das Attribut `PTHREAD_CREATE_DETACHED` zum Abhängen (*detach*) von Threads verwenden. Hierzu können Sie die folgenden Funktionen verwenden:

```
#include <pthread.h>

int pthread_attr_init( pthread_attr_t *attribute );
int pthread_attr_getdetachestate( pthread_attr_t *attribute,
                                  int detachstate );
```

```
int pthread_attr_setdetachestate( pthread_attr_t *attribute,
                                  int detachstate );
int pthread_attr_destroy( pthread_attr_t *attribute );
```

Mit der Funktion `pthread_attr_init()` müssen Sie zunächst das Attributobjekt `attr` initialisieren. Dabei werden auch gleich die voreingestellten Attribute gesetzt. Um beim Thema »detached« und »joinable« zu bleiben: Die Voreinstellung hier ist `PTHREAD_CREATE_JOINABLE`. Damit wird also der Thread nicht von den anderen losgelöst und erst freigegeben, wenn ein anderer Thread nach dem Exit-Status dieses Threads fragt (mit `pthread_join()`).

Mit der Funktion `pthread_attr_getdetachestate()` können Sie das `detached`-Attribut erfragen, und mit `pthread_attr_setdetachedstate()` wird es gesetzt. Neben dem eben erwähnten `PTHREAD_CREATE_JOINABLE`, das ja die Standardeinstellung eines erzeugten Threads ist, können Sie hierbei auch `PTHREAD_CREATE_DETACHED` verwenden.

Das Setzen von `PTHREAD_CREATE_DETACHED` entspricht exakt dem Verhalten der Funktion `pthread_detach()` (siehe Abschnitt 12.3.6) und kann auch stattdessen verwendet werden – da es erheblich kürzer ist. Benötigen Sie das Attributobjekt `attr` nicht mehr, können Sie es mit `pthread_attr_destroy()` löschen. Somit machen die Funktionen wohl erst Sinn, wenn Sie bereits mit `pthread_detach()` einen Thread ausgehängt haben und diesen eventuell wieder zurückholen (`PTHREAD_CREATE_JOINABLE`) müssen.

Bedeutend wichtiger im Zusammenhang mit den Attributen von Threads erscheint hier schon das Setzen der Prozessorzuteilung (Scheduling). Laut POSIX gibt es drei verschiedene solcher Prozesszuteilungen (*Scheduling Policys*):

- `SCHED_OTHER` – Die normale Priorität wie bei einem gewöhnlichen Prozess. Der Thread wird beendet: entweder wenn seine Zeit um ist und er wartet, bis er wieder am Zuge ist, oder wenn ein anderer Thread oder Prozess gestartet wurde, der mit einer höheren Priorität ausgestattet ist.
- Echtzeit (`SCHED_FIFO`) – Dies sind Echtzeitprozesse. Sie werden in jedem Fall `SCHED_OTHER`-Prozessen vorgezogen. Auch können sie nicht von normalen Prozessen unterbrochen werden. Es gibt drei Möglichkeiten, Echtzeitprozesse zu unterbrechen:
  - Der Echtzeitprozess wandert in eine Warteschlange und wartet auf ein externes Ereignis.
  - Der Echtzeitprozess verlässt freiwillig die CPU (z. B. mit `sched_yield()`).
  - Der Echtzeitprozess wird von einem anderen Echtzeitprozess mit einer höheren Priorität verdrängt.
- Echtzeit (`SCHED_RR`) – Dies sind Round-Robin-Echtzeitprozesse. Beim Round-Robin-Verfahren hat jeder Prozess die gleiche Zeitspanne zur Verfügung. Ist diese verstrichen, so kommt der nächste Prozess an die Reihe. Unter Linux werden diese Prozesse genauso behandelt wie die oben genannten Echtzeitprozesse, mit dem Unterschied, dass diese an das Ende der *Run Queue* gesetzt werden, wenn sie den Prozessor verlassen.

Jetzt haben wir Echtzeitoperationen ins Spiel gebracht. Daher sollten wir einen kurzen Exkurs machen, damit Sie die Echtzeitstrategie nicht mit »jetzt –gleich sofort« vergleichen. Die Abarbeitung von Daten in Echtzeit kann einfach nicht sofort ausgeführt werden, sondern auch hier muss man sich damit begnügen, dass diese innerhalb einer vorgegebenen Zeitspanne abgearbeitet werden. Allerdings müssen solche Echtzeitoperationen auch unterbrechbar sein, um auf plötzliche, unvorsehbare Ereignisse reagieren zu können.

Daher unterscheidet man hier zwischen »weichen« und »harten« Echtzeitanforderungen. Die Anforderungen hängen vom Anwendungsfall ab. So kann man bei einem Computerspiel jederzeit »weiche« Echtzeitanforderungen setzen – was bei Maschinenanforderungen wohl eher katastrophal sein kann. Bei Maschinen muss innerhalb einer vorgegebenen Zeit reagiert werden. Der Hauptbereich von Echtzeitanwendungen ist immer noch:

- Multimedia – Audio, Video
- Steuerung, Regelung – Maschinen-, Robotersteuerung

Damit eine solche Zuteilungsstrategie auch funktioniert, muss das System sie auch unterstützen. Dies ist gegeben, wenn in Ihrem Programm die Konstante `_POSIX_THREAD_PRIORITY_SCHEDULING` definiert ist. Beachten Sie außerdem, dass die Echtzeit-Zuteilungsstrategien `SCHED_FIFO` und `SCHED_RR` nur vom Superuser root ausgeführt werden können.

Die Zustellungsstrategie verändern bzw. erfragen können Sie mit den folgenden Funktionen:

```
int pthread_setschedparam( pthread thread, int policy,
                           const struct sched_param *param);
int pthread_getschedparam( pthread thread, int policy,
                           struct sched_param *param);
```

Mit diesen Funktionen setzen (`set`) oder ermitteln (`get`) Sie die Zustellungsstrategie eines Threads mit der ID `thread` vom Typ `pthread_t`. Die Strategie legen Sie mit dem Parameter `policy` fest. Hierbei kommen die bereits beschriebenen Konstanten `SCHED_OTHER`, `SCHED_FIFO` und `SCHED_RR` infrage.

Mit dem letzten Parameter der Struktur `sched_param`, die sich in der Headerdatei *<bits/sched.h>* befindet, legen Sie die gewünschte Priorität fest:

```
/* Struktur sched_param */
struct sched_param {
    int sched_priority;
};
```

Das folgende Beispiel soll Ihnen zeigen, wie einfach es ist, die Zuteilungsstrategie und die Priorität zu verändern. Sie finden hier zwei Funktionen: eine, mit der Sie die Strategie und die Priorität abfragen können, sowie eine weitere, mit der Sie diese Werte neu setzen kön-

nen. Allerdings benötigen Sie für das Setzen Superuser-Root-Rechte, was im Beispiel ebenfalls ermittelt wird.

```
/* thread4.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#define MAX_THREADS 3
#define BUF 255

/* Funktion ermittelt die Zuteilungsstrategie *
 * und Priorität eines Threads */
static void getprio( pthread_t id ) {
   int policy;
   struct sched_param param;

   printf("\t->Thread %ld: ", id);
   if((pthread_getschedparam(id, &policy, &param)) == 0  ) {
      printf("Zuteilung: ");
      switch( policy ) {
         case SCHED_OTHER :  printf("SCHED_OTHER; "); break;
         case SCHED_FIFO  :  printf("SCHED_FIFO; ");  break;
         case SCHED_RR    :  printf("SCHED_RR; ");    break;
         default          :  printf("Unbekannt; ");   break;
      }
      printf("Priorität: %d\n", param.sched_priority);
   }
}

/* Funktion zum Setzen der Zuteilungsstrategie *
 * und Priorität eines Threads */
static void setprio( pthread_t id, int policy, int prio ) {
   struct sched_param param;

   param.sched_priority=prio;
   if((pthread_setschedparam( pthread_self(),
                              policy, &param)) != 0  ) {
      printf("Konnte Zuteilungsstrategie nicht ändern\n");
      pthread_exit((void *)pthread_self());
   }
}
```



```
static void thread_prio_demo(void *name) {
   int policy;
   struct sched_param param;
   /*Aktuelle Zuteilungsstrategie und Priorität erfragen */
   getprio(pthread_self());
   /* Ändern darf hier nur der root */
   if( getuid() != 0 ) {
      printf("Verändern geht nur mit Superuser-Rechten\n");
      pthread_exit((void *)pthread_self());
   }
   /* Neue Zuteilungsstrategie und Priorität festsetzen */
   setprio(pthread_self(), SCHED_RR, 2);
   /* Nochmals abfragen, ob erfolgreich verändert ... */
   getprio(pthread_self());
   /* Thread-Ende */
   pthread_exit((void *)pthread_self());
}

int main (void) {
   int i;
   static int ret[MAX_THREADS];
   static pthread_t th[MAX_THREADS];

   printf("->Haupt-Thread (ID:%ld) gestartet ...\n",
      pthread_self());
   /* Threads erzeugen */
   for (i = 0; i < MAX_THREADS; i++) {
      if (pthread_create ( &th[i],NULL, &thread_prio_demo,
                           NULL) != 0) {
         printf ("Konnte keinen Thread erzeugen\n");
         exit (EXIT_FAILURE);
      }
   }
   /* Auf die Threads warten */
   for (i = 0; i < MAX_THREADS; i++)
      pthread_join (th[i], &ret[i]);
   /* Rückgabe der Threads auswerten */
   for (i = 0; i < MAX_THREADS; i++)
      printf("\t<-Thread %ld mit Arbeit fertig\n", ret[i]);
   printf("->Haupt-Thread (ID:%ld) fertig ...\n",
      pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread4 thread4.c -pthread
$ ./thread4
->Haupt-Thread (ID:-1209412512) gestartet...
        ->Thread -1209414736: Zuteilung: SCHED_OTHER; Priorität: 0
!!! Verändern geht nur mit Superuser-Rechten!!!
        ->Thread -1217807440: Zuteilung: SCHED_OTHER; Priorität: 0
!!! Verändern geht nur mit Superuser-Rechten!!!
        ->Thread -1226200144: Zuteilung: SCHED_OTHER; Priorität: 0
!!! Verändern geht nur mit Superuser-Rechten!!!
        <-Thread -1209414736 mit Arbeit fertig
        <-Thread -1217807440 mit Arbeit fertig
        <-Thread -1226200144 mit Arbeit fertig
->Haupt-Thread (ID:-1209412512) fertig ...
$ su
Password:********
# ./thread4
->Haupt-Thread (ID:-1209412512) gestartet ...
->Thread -1209414736: Zuteilung: SCHED_OTHER; Priorität: 0
->Thread -1209414736: Zuteilung: SCHED_RR; Priorität: 2
->Thread -1217807440: Zuteilung: SCHED_OTHER; Priorität: 0
->Thread -1217807440: Zuteilung: SCHED_RR; Priorität: 2
->Thread -1226200144: Zuteilung: SCHED_OTHER; Priorität: 0
->Thread -1226200144: Zuteilung: SCHED_RR; Priorität: 2
        <-Thread -1209414736 mit Arbeit fertig
        <-Thread -1217807440 mit Arbeit fertig
        <-Thread -1226200144 mit Arbeit fertig
->Haupt-Thread (ID:-1209412512) fertig ...
```

Selbiges (Zuteilungsstrategien und Priorität) können Sie übrigens mit folgenden Funktionen auch über Attributobjekte (`pthread_attr_t`) setzen bzw. erfragen:

```
#include <pthread.h>

/* Zuteilungsstrategie verändern bzw. erfragen */
int pthread_attr_setschedpolicy( pthread_attr_t *attr,
                                 int policy);
int pthread_attr_getschedpolicy( const pthread_attr_t *attr,
                                 int *policy);

/* Priorität verändern bzw. erfragen */
int pthread_attr_setschedparam(
```



```
   pthread_attr_t *attr, const struct sched_param *param );
int pthread_attr_getschedparam(
   const pthread_attr_t *attr, struct sched_param *param );
```

Wollen Sie außerdem festlegen bzw. abfragen, wie ein Thread seine Attribute (Zuteilungsstrategie und Priorität) vom Erzeuger-Thread übernehmen soll, stehen Ihnen folgende Funktionen zur Verfügung:

```
#include <pthread.h>

int pthread_attr_setinheritsched(
   pthread_attr_t *attr, int inheritsched );
int pthread_attr_getinheritsched(
   const pthread_attr_t *attr, int *inheritsched );
```

Mit den beiden Funktionen `phtread_attr_getinheritsched()` und `phtread_attr_setinheritsched()` können Sie abfragen bzw. festlegen, wie der Thread die Attribute vom »Eltern«-Thread übernimmt. Dabei gibt es zwei Möglichkeiten:

- `PTHREAD_INHERIT_SCHED` bedeutet, dass der Kind-Thread die Attribute (mitsamt Zuteilungsstrategie und der Priorität) des Eltern-Threads übernimmt.
- `PTHREAD_EXPLICIT_SCHED` bedeutet, eben nichts zu übernehmen, sondern das zu verwenden, was in `attr` als Zuteilungsstrategie und Priorität festgelegt ist. Wurden die Attribute des »Eltern«-Threads nicht verändert, so ist der Kind-Thread dennoch (logischerweise) mit denselben Attributen wie der »Eltern«-Thread ausgestattet – da es sich um die Standardattribute handelt.

12.5 Threads synchronisieren

In vielen Fällen – bei Threads eigentlich fast immer – werden mehrere parallel laufende Prozesse benötigt, die gemeinsame Daten verwenden und/oder austauschen. Das einfachste Beispiel ist: Ein Thread schreibt gerade etwas in eine Datei, während ein anderer Thread daraus etwas liest. Dasselbe Problem haben Sie auch beim Zugriff auf globale Variablen. Wenn mehrere Threads auf eine globale Variable zugreifen müssen und Sie keine Vorkehrungen dafür getroffen haben, ist nicht vorherzusagen, welcher Thread die Variable gerade bearbeitet. Sind in diesem Szenario z. B. mathematische Arbeiten auf mehrere Threads aufgeteilt, kann man mit fast hundertprozentiger Sicherheit sagen, dass das Ergebnis nicht richtig sein wird.

Hierfür sei folgendes einfaches Beispiel gegeben. Zwei Threads greifen auf eine globale Variable zu – hier auf einen geöffneten `FILE`-Zeiger. Ein Thread wird erzeugt, um etwas in diese Datei zu schreiben, und ein weiterer Thread soll sie wieder auslesen. Ein simples Beispiel, wie

es scheint, nur dass es hierbei schon zu Synchronisationsproblemen (*Race Conditions*) kommt. Aber testen Sie selbst:

```
/* thread5.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define MAX_THREADS 2
#define BUF 255
#define COUNTER 10000000

/* Globale Variable */
static FILE *fz;

static void open_file(const char *file) {
   fz = fopen( file, "w+" );
   if( fz == NULL ) {
      printf("Konnte Datei %s nicht öffnen\n", file);
      exit(EXIT_FAILURE);
   }
}

static void thread_schreiben(void *name) {
   char string[BUF];

   printf("Bitte Eingabe machen: ");
   fgets(string, BUF, stdin);
   fputs(string, fz);
   fflush(fz);
   /* Thread-Ende */
   pthread_exit((void *)pthread_self());
}

static void thread_lesen(void *name) {
   char string[BUF];
   rewind(fz);
   fgets(string, BUF, fz);
   printf("Ausgabe Thread %ld: ", pthread_self());
   fputs(string, stdout);
   fflush(stdout);
   /* Thread-Ende */
```



```
   pthread_exit((void *)pthread_self());
}

int main (void) {
   static pthread_t th1, th2;
   static int ret1, ret2;

   printf("->Haupt-Thread (ID:%ld) gestartet ...\n",
      pthread_self());
   open_file("testfile");
   /* Threads erzeugen */
   if (pthread_create( &th1, NULL,
                       &thread_schreiben, NULL)!=0) {
      fprintf (stderr, "Konnte keinen Thread erzeugen\n");
      exit (EXIT_FAILURE);
   }
   /* Threads erzeugen */
   if (pthread_create(&th2,NULL,&thread_lesen,NULL) != 0) {
      fprintf (stderr, "Konnte keinen Thread erzeugen\n");
      exit (EXIT_FAILURE);
   }

   pthread_join(th1, &ret1);
   pthread_join(th2, &ret2);

   printf("<-Thread %ld fertig\n", th1);
   printf("<-Thread %ld fertig\n", th1);
   printf("<-Haupt-Thread (ID:%ld) fertig ...\n",
      pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread5 thread5.c -pthread
$ ./thread5
->Haupt-Thread (ID:-1209412512) gestartet...
Bitte Eingabe machen: Ausgabe Thread -1217807440: Hallo, das ist ein Test
<-Thread -1209414736 fertig
<-Thread -1209414736 fertig
<-Haupt-Thread (ID:-1209412512) fertig ...
$ cat testfile
Hallo, das ist ein Test
```

Bei der Eingabe können Sie schon erkennen, dass der Thread `thread_lesen` schon mit seiner Ausgabe begonnen hat und sich schon wieder beendet hat, bevor Sie etwas von der Tastatur eingeben konnten. Wenn alles läuft, wie es soll, sollte hier folgende Ausgabe bei der Programmausführung entstehen:

```
$ ./thread5
->Haupt-Thread (ID:-1209412512) gestartet ...
Bitte Eingabe machen: Hallo Welt
Ausgabe Thread -1217807440: Hallo Welt
<-Thread -1209414736 fertig
<-Thread -1209414736 fertig
<-Haupt-Thread (ID:-1209412512) fertig ...
```

Zugegeben, als echter C-Guru würde Ihnen jetzt hier schon etwas einfallen. Zum Beispiel könnten Sie eine »pollende« Schleife mit einem `sleep()` um den Lese-Thread herumbauen, die immer wieder abfragt, ob `fgets()` etwas eingelesen hat. Das wäre allerdings nicht im Sinne des Erfinders, und falls Sie wirklich die Threads für Echtzeitanwendungen verwenden wollen, ist das wohl auch das Ende Ihrer Programmiererkarriere, wenn die Weichenschaltung einer U-Bahn »in einer pollenden Schleife« warten muss, bevor die Weiche gestellt werden kann!

Für solche Fälle gibt es einige Synchronisationsmöglichkeiten, die Ihnen die Thread-Bibliothek anbietet.

12.5.1 Mutexe

Wenn Sie mehrere Threads starten und diese quasi parallel ablaufen, können Sie nicht erkennen, wie weit welcher Thread gerade mit der Verarbeitung von Daten ist. Wenn mehrere Threads beispielsweise an ein und derselben Aufgabe abhängig voneinander arbeiten, wird eine Synchronisation erforderlich. Genauso ist dies erforderlich, wenn Threads globale Variablen oder die Hardware, z. B. die Tastatur (`stdin`), verwenden, da sonst ein Thread diese Variable einfach überschreiben würde, noch bevor sie verwendet wird.

Um Threads zu synchronisieren, haben Sie zwei Möglichkeiten: zum einen mit sogenannten *Locks*, die Sie in diesem Abschnitt zusammen mit den Mutexen durchgehen werden, und zum anderen mit einem *Monitor*. Mit dem Monitor werden sogenannte Condition-Variablen verwendet.

Hinweis

Der Begriff »Mutex« steht für *Mutual Exclusion* (gegenseitiger Ausschluss). Ein Mutex ist somit ohne Besitzer oder gehört genau einem Thread.

Die Funktionsweise von Mutexen ähnelt den Semaphoren bei den Prozessen. Genauer gesagt: Ein Mutex ist nichts weiter als ein Semaphor, was wiederum nur eine atomare Operation auf eine Variable ist. Trotzdem lässt sich ein Mutex aber wesentlich einfacher erstellen. Das Prinzip ist simpel (siehe Abbildung 12.2). Ein Thread arbeitet mit einer globalen oder statischen Variablen, die für alle anderen Threads von einem Mutex blockiert (gesperrt) wird. Benötigt der Thread diese Variable nicht mehr, gibt er diese frei.

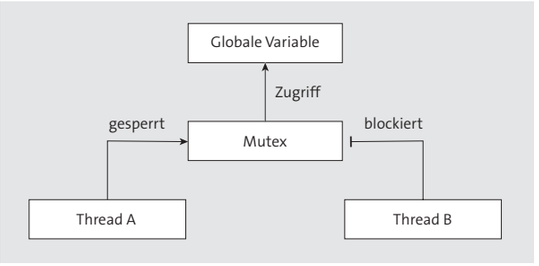

Abbildung 12.2 Nur ein Thread kann einen Mutex sperren.

Anhand dieser Erklärung dürfte auch klar sein, dass man selbst dafür verantwortlich ist, keinen Deadlock zu erzeugen. In folgenden Fällen könnten auch bei Threads Deadlocks auftreten:

- Threads können Ressourcen anfordern, obwohl sie bereits Ressourcen besitzen.
- Ein Thread gibt seine Ressource nicht mehr frei.
- Eine Ressource ist frei oder im Besitz eines »exklusiven« Threads.

Im Falle eines Deadlocks kann keiner der beteiligten Threads seine Arbeit mehr fortsetzen und somit ist meist keine normale Beendigung mehr möglich. Datenverlust kann die Folge sein.

Statische Mutexe

Um eine Mutex-Variable als statisch zu definieren, müssen Sie sie mit der Konstante `PTHREAD_MUTEX_INITIALIZER` initialisieren. Folgende Funktionen stehen Ihnen zur Verfügung, um Mutexe zu sperren und wieder freizugeben:

```
#include <pthread.h>

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

Mit `pthread_mutex_lock()` sperren Sie einen Mutex. Wenn hierbei z. B. ein Thread versucht, mit demselben Mutex ebenfalls eine Sperre einzurichten, so wird er so lange blockiert, bis der Mutex von einem anderen Thread wieder mittels `pthread_mutex_unlock()` freigegeben wird.

Die Funktion `pthread_mutex_trylock()` ist ähnlich wie `pthread_mutex_lock()`, nur dass diese Funktion den aufrufenden Thread nicht blockiert, wenn ein Mutex durch einen anderen Thread blockiert wird. `pthread_mutex_trylock()` kehrt stattdessen mit dem Fehlercode (`errno`) `EBUSY` zurück und macht mit der Ausführung des aufrufenden Threads weiter.

Das folgende Beispiel ist dasselbe, das Sie schon vom Listing *thread5.c* her kennen, nur dass jetzt das Synchronisationsproblem mithilfe eines Mutex behoben wird. Zuerst wird global der Mutex mit der Konstante `PTHREAD_MUTEX_INITIALIZER` statisch initialisiert, und anschließend werden im Beispiel die Sperren dort gesetzt und wieder freigegeben, wo dies sinnvoll ist.

```
/* thread6.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

#define MAX_THREADS 2
#define BUF 255
#define COUNTER 10000000

static FILE *fz;

/* Statische Mutex-Variable */
pthread_mutex_t fz_mutex=PTHREAD_MUTEX_INITIALIZER;

static void open_file(const char *file) {
   fz = fopen( file, "w+" );
   if( fz == NULL ) {
      printf("Konnte Datei %s nicht öffnen\n", file);
      exit(EXIT_FAILURE);
   }
}

static void thread_schreiben(void *name) {
   char string[BUF];

   printf("Bitte Eingabe machen: ");
   fgets(string, BUF, stdin);
   fputs(string, fz);
   fflush(fz);

   /* Mutex wieder freigeben */
```



```
   pthread_mutex_unlock( &fz_mutex );

   /* Thread-Ende */
   pthread_exit((void *)pthread_self());
}

static void thread_lesen(void *name) {
   char string[BUF];

   /* Mutex sperren */
   pthread_mutex_lock( &fz_mutex );
   rewind(fz);
   fgets(string, BUF, fz);
   printf("Ausgabe Thread %ld: ", pthread_self());
   fputs(string, stdout);
   fflush(stdout);

   /* Mutex wieder freigeben */
   pthread_mutex_unlock( &fz_mutex );

   /* Thread-Ende */
   pthread_exit((void *)pthread_self());
}


int main (void) {
   static pthread_t th1, th2;
   static int ret1, ret2;

   printf("->Haupt-Thread (ID:%ld) gestartet ...\n",
      pthread_self());
   open_file("testfile");

   /* Mutex sperren */
   pthread_mutex_lock( &fz_mutex );


   /* Threads erzeugen */
   if( pthread_create( &th1, NULL, &thread_schreiben,
                       NULL)!=0) {
      fprintf (stderr, "Konnte keinen Thread erzeugen\n");
      exit (EXIT_FAILURE);
   }
```

```
   /* Threads erzeugen */
   if(pthread_create(&th2,NULL, &thread_lesen, NULL) != 0) {
      fprintf (stderr, "Konnte keinen Thread erzeugen\n");
      exit (EXIT_FAILURE);
   }
   pthread_join(th1, &ret1);
   pthread_join(th2, &ret2);

   printf("<-Thread %ld fertig\n", th1);
   printf("<-Thread %ld fertig\n", th1);
   printf("<-Haupt-Thread (ID:%ld) fertig ...\n",
      pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread6 thread6.c -pthread
$ ./thread6
->Haupt-Thread (ID:-1209412512) gestartet ...
Bitte Eingabe machen: Hallo Welt mit Mutexen
Ausgabe Thread -1217807440: Hallo Welt mit Mutexen
<-Thread -1209414736 fertig
<-Thread -1209414736 fertig
<-Haupt-Thread (ID:-1209412512) fertig ...
```

Natürlich können Sie den Lese-Thread mit `pthread_mutex_trylock()` als eine nicht blockierende Mutex-Anforderung ausführen. Hierzu müssten Sie nur die Funktion `thread_lesen` ein wenig umändern. Hier ist ein möglicher Ansatz:

```
static void thread_lesen(void *name) {
   char string[BUF];

   /* Versuche Mutex zu sperren */
   while( (pthread_mutex_trylock( &fz_mutex )) == EBUSY) {
      sleep(10);
      printf("Lese-Thread wartet auf Arbeit ...\n");
      printf("Bitte Eingabe machen: ");
      fflush(stdout);
   }
   rewind(fz);
   fgets(string, BUF, fz);
   printf("Ausgabe Thread %ld: ", pthread_self());
   fputs(string, stdout);
```



```
   fflush(stdout);

   /* Mutex wieder freigeben */
   pthread_mutex_unlock( &fz_mutex );

   /* Thread-Ende */
   pthread_exit((void *)pthread_self());
}
```

Hierbei wird versucht, alle zehn Sekunden den Mutex zu sperren. Solange `EBUSY` zurückgegeben wird, ist der Mutex noch von einem anderen Thread gesperrt. Während dieser Zeit könnte der wartende Thread ja andere Arbeiten ausführen (es gibt immer was zu tun).

Hier sehen Sie das Programm bei der Ausführung mit `pthread_mutex_trylock()`:

```
$ gcc -o thread7 thread7.c -pthread
$ ./thread7
->Haupt-Thread (ID:-1209412512) gestartet ...
Bitte Eingabe machen: Lese-Thread wartet auf Arbeit ...
Bitte Eingabe machen: Lese-Thread wartet auf Arbeit ...
Bitte Eingabe machen: Hallo Mutex, Du bist frei
Ausgabe Thread -1217807440: Hallo Mutex, Du bist frei
<-Thread -1209414736 fertig
<-Thread -1209414736 fertig
<-Haupt-Thread (ID:-1209412512) fertig ...
```


Dynamische Mutexe

Wenn Sie Mutexe in einer Struktur verwenden wollen, was durchaus eine gängige Praxis ist, können Sie dynamische Mutexe verwenden. Dies sind dann Mutexe, für die zur Laufzeit mit z. B. `malloc()` Speicher angefordert wird. Für dynamische Mutexe stehen folgende Funktionen zur Verfügung:

```
#include <pthread.h>

int  pthread_mutex_init(
   pthread_mutex_t  *mutex,
   const  pthread_mutex_attr_t *mutexattr );
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Mit `pthread_mutex_init()` initialisieren Sie den Mutex `mutex`. Mit dem Parameter `mutexattr` können Sie Attribute für den Mutex verwenden. Wird hierbei `NULL` angegeben, werden die Standardattribute verwendet. Auf die Attribute von Mutexen gehen wir in Kürze ein. Freigeben können Sie einen solchen dynamisch angelegten Mutex wieder mit `pthread_mutex_destroy()`.

Hierzu zeigen wir nochmals dasselbe Beispiel wie eben mit *thread6.c*, nur mit dynamisch angelegtem Mutex.

```
/* thread8.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>
#define BUF 255

struct data {
   FILE *fz;
   char filename[BUF];
   pthread_mutex_t mutex;
};

static void thread_schreiben(void *arg) {
   char string[BUF];
   struct data *d=(struct data *)arg;

   printf("Bitte Eingabe machen: ");
   fgets(string, BUF, stdin);
   fputs(string, d->fz);
   fflush(d->fz);

   /* Mutex wieder freigeben */
   pthread_mutex_unlock( &d->mutex );
   /* Thread-Ende */
   pthread_exit((void *)pthread_self());
}

static void thread_lesen(void *arg) {
   char string[BUF];
   struct data *d=(struct data *)arg;

   /* Mutex sperren */
   while( (pthread_mutex_trylock( &d->mutex )) == EBUSY) {
      sleep(10);
      printf("Lese-Thread wartet auf Arbeit ...\n");
      printf("Bitte Eingabe machen: ");
      fflush(stdout);
   }
```



```
   rewind(d->fz);
   fgets(string, BUF, d->fz);
   printf("Ausgabe Thread %ld: ", pthread_self());
   fputs(string, stdout);
   fflush(stdout);
   /* Mutex wieder freigeben */
   pthread_mutex_unlock( &d->mutex );
   /* Thread-Ende */
   pthread_exit((void *)pthread_self());
}

int main (void) {
   static pthread_t th1, th2;
   static int ret1, ret2;
   struct data *d;

   /* Speicher für die Struktur reservieren */
   d = malloc(sizeof(struct data));
   if(d == NULL) {
      printf("Konnte keinen Speicher reservieren ...!\n");
      exit(EXIT_FAILURE);
   }

   printf("->Haupt-Thread (ID:%ld) gestartet ...\n",
      pthread_self());

   strncpy(d->filename, "testfile", BUF);
   d->fz = fopen( d->filename, "w+" );
   if( d->fz == NULL ) {
      printf("Konnte Datei %s nicht öffnen\n", d->filename);
      exit(EXIT_FAILURE);
   }

   /* Mutex initialisieren */
   pthread_mutex_init( &d->mutex, NULL );
   /* Mutex sperren */
   pthread_mutex_lock( &d->mutex );

   /* Threads erzeugen */
   if(pthread_create (&th1,NULL,&thread_schreiben,d) != 0) {
      fprintf (stderr, "Konnte keinen Thread erzeugen\n");
      exit (EXIT_FAILURE);
   }
```

```
   /* Threads erzeugen */
   if (pthread_create (&th2,NULL, &thread_lesen, d) != 0) {
      fprintf (stderr, "Konnte keinen Thread erzeugen\n");
      exit (EXIT_FAILURE);
   }
   pthread_join(th1, &ret1);
   pthread_join(th2, &ret2);

   /* Dynamisch angelegten Mutex löschen */
   pthread_mutex_destroy( &d->mutex );

   printf("<-Thread %ld fertig\n", th1);
   printf("<-Thread %ld fertig\n", th1);
   printf("<-Haupt-Thread (ID:%ld) fertig ...\n",
      pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung zu zeigen können wir uns hier sparen, da es exakt dem Beispiel *thread6.c* entspricht, nur dass hierbei eben ein dynamischer Mutex statt eines statischen verwendet wurde.

Mutex-Attribute

Mit den folgenden Funktionen können Sie Mutex-Attribute verändern oder abfragen:

```
#include <pthread.h>

int pthread_mutexattr_init( pthread_mutexattr_t *attr );
int pthread_mutexattr_destroy( pthread_mutexattr_t *attr );
int pthread_mutexattr_settype( pthread_mutexattr_t *attr,
                               int kind );
int pthread_mutexattr_gettype(
   const  pthread_mutexattr_t  *attr,   int *kind );
```

Mit dem Mutex-Attribut legen Sie fest, was passiert, wenn ein Thread versuchen sollte, einen Mutex nochmals zu sperren, obwohl dieser bereits mit `pthread_mutex_lock()` gesperrt wurde. Mit der Funktion `pthread_mutexattr_init()` initialisieren Sie zunächst das Mutex-Attributobjekt `attr`. Zunächst wird hierbei die Standardeinstellung (`PTHREAD_MUTEX_FAST_NP`) verwendet. Ändern können Sie dieses Attribut mit `pthread_mutexattr_settype()`. Damit setzen Sie die Attribute des Mutex-Attributobjekts auf `kind`. Folgende Konstanten können Sie hierbei für `kind` verwenden:

- `PTHREAD_MUTEX_FAST_NP` (Standardeinstellung) – `pthread_mutex_lock()` blockiert den aufrufenden Thread für immer. Also ein Deadlock.
- `PTHREAD_MUTEX_RECURSIVE_NP` – `pthread_mutex_lock()` blockiert nicht und kehrt sofort erfolgreich zurück. Wird ein Thread mit diesem Mutex gesperrt, so wird ein Zähler für jede Sperrung um den Wert 1 erhöht. Damit die Sperrung eines rekursiven Mutex aufgehoben wird, muss dieser ebenso oft freigegeben werden, wie er gesperrt wurde.
- `PTHREAD_MUTEX_ERRORCHECK_NP` – `pthread_mutex_lock()` kehrt sofort wieder mit dem Fehlercode `EDEADLK` zurück, also ähnlich wie mit `pthread_mutex_trylock()`, nur dass hier eben `EBUSY` zurückgegeben wird.

> **Hinweis**
>
> Da die Variablen hierbei mit dem Suffix `_NP` (*non-portable*) verbunden sind, sind sie nicht mit dem POSIX-Standard vereinbar und somit nicht für portable Programme geeignet.

12.5.2 Condition-Variablen (Bedingungsvariablen)

Bedingungsvariablen werden dazu verwendet, auf das Eintreffen einer bestimmten Bedingung zu warten bzw. die Erfüllung oder den Eintritt einer Bedingung zu zeigen. Bedingungsvariablen werden außerdem mit den Mutexen verknüpft. Dabei wird beim Warten auf eine Bedingung eine Sperre zu einem mit ihr verknüpften Mutex freigegeben (natürlich musste zuvor eine Sperre auf den Mutex erfolgt sein).

Andersherum sollte vor einem Eintreffen einer Bedingung eine Sperre auf den verknüpften Mutex erfolgen, sodass nach dem Warten auf diesen Mutex auch die Sperre auf den Mutex wieder vorhanden ist. Erfolgte keine Sperre vor dem Signal, wartet ein Thread wieder, bis eine Sperre auf den Mutex möglich ist.

Statische Bedingungsvariablen

Für die Bedingungsvariablen wird der Datentyp `pthread_cont_t` verwendet. Damit eine solche Bedingungsvariable überhaupt als statisch definiert ist, muss sie mit der Konstante `PTHREAD_COND_INITIALIZER` initialisiert werden.

Hier sehen Sie die Funktionen, mit deren Hilfe Sie mit Condition-Variablen operieren können:

```
#include <pthread.h>

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

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

Bevor Sie zunächst die Funktion `pthread_cond_wait()` verwenden, müssen Sie beim aufrufenden Thread den Mutex `mutex` sperren. Mit einem anschließenden `pthread_cond_wait()` wird der Mutex dann freigegeben, und der Thread wird mit der Bedingungsvariablen `cond` so lange blockiert, bis eine bestimmte Bedingung eintritt. Bei einem erfolgreichen Aufruf von `pthread_cond_wait()` wird auch für den Mutex automatisch die Sperre wieder eingerichtet – oder es herrscht einfach wieder derselbe Zustand wie vor dem `pthread_cond_wait`-Aufruf.

Threads, die auf die Bedingungsvariable `cond` warten, können Sie mit `phtread_cond_signal()` wieder aufwecken und weiter ausführen. Bei mehreren Threads, die auf die Bedingungsvariable `cond` warten, bekommt der Thread mit der höchsten Priorität den Zuschlag.

Wollen Sie hingegen alle Threads aufwecken, die auf die Bedingungsvariable `cond` warten, können Sie die Funktion `pthread_cond_signal()` verwenden.

Natürlich gibt es auch noch eine Funktion, mit der Sie – im Gegensatz zu `phtread_cond_wait()` – nur eine gewisse Zeit auf die Bedingungsvariable `cond` warten, bevor sie zum aufrufenden Thread zurückkehrt und wieder automatisch die Sperre von Mutex einrichtet. Das ist die Funktion `pthread_cond_timewait()`. Als Zeit können Sie hierbei `abstime` verwenden, womit Sie eine absolute Zeit in Sekunden und Nanosekunden angeben, die seit dem 1.1.1970 vergangen sind.

```
struct timespec {
   time_t tv_sec;   // Sekunden
   long   tv_nsec;  // Nanosekunden
};
```

Hierzu folgt ein recht einfaches Beispiel, das nur die Funktionalität von Bedingungsvariablen und vor allem deren Verwendung demonstriert:

```
/* thread9.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>


#define THREAD_MAX 3

pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```



```
static void *threads (void *arg) {
   printf("\t->Thread %ld wartet auf Bedingung\n",
      pthread_self());

   pthread_mutex_lock(&mutex);
   pthread_cond_wait(&cond, &mutex);

   printf("\t->Thread %ld hat Bedingung erhalten\n",
      pthread_self());

   printf("\t->Thread %ld: Sende wieder die "
          "Bedingungsvariable\n", pthread_self());
   pthread_cond_signal(&cond);
   pthread_mutex_unlock(&mutex);
   return NULL;
}

int main (void) {
   int i;
   pthread_t th[THREAD_MAX];

   printf("->Main-Thread %ld gestartet\n", pthread_self());
   for(i=0; i<THREAD_MAX; i++)
      if (pthread_create (&th[i],NULL, &threads, NULL)!=0) {
         printf ("Konnte keinen Thread erzeugen\n");
         exit (EXIT_FAILURE);
      }
   printf("->Main-Thread: habe soeben %d Threads erzeugt\n",
      THREAD_MAX);

   /* Kurz ruhig legen, damit der Main-Thread als Erstes die
    * Bedingungsvariable sendet */
   sleep(1);
   printf("->Main-Thread: Sende die Bedingungsvariable\n");
   pthread_cond_signal(&cond);

   for(i=0; i<THREAD_MAX; i++)
      pthread_join (th[i], NULL);
   printf("->Main-Thread %ld beendet\n", pthread_self());
   pthread_exit(NULL);
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread9 thread9.c -pthread
$ ./thread9
->Main-Thread -1209416608 gestartet
->Main-Thread: habe soeben 3 Threads erzeugt
        ->Thread -1209418832 wartet auf Bedingung
        ->Thread -1217811536 wartet auf Bedingung
        ->Thread -1226204240 wartet auf Bedingung
->Main-Thread: Sende die Bedingungsvariable
        ->Thread -1209418832 hat Bedingung erhalten
        ->Thread -1209418832: Sende wieder die Bedingungsvariable
        ->Thread -1217811536 hat Bedingung erhalten
        ->Thread -1217811536: Sende wieder die Bedingungsvariable
        ->Thread -1226204240 hat Bedingung erhalten
        ->Thread -1226204240: Sende wieder die Bedingungsvariable
->Main-Thread -1209416608 beendet
```

Sie sehen an diesem Beispiel, dass eine Kettenreaktion der weiteren Threads entsteht, sobald der Haupt-Thread eine Bedingungsvariable »sendet«. Hier werden die Threads so abgearbeitet, wie sie in der Queue angelegt wurden.

Im folgenden Beispiel wartet der Thread Nummer 2 auf die Condition-Variable von Thread 1. Thread 1 weist einem globalen Zahlenarray `werte` zehn Werte zu, die Thread 2 anschließend berechnet. Dies ist natürlich auch wieder ein primitives Beispiel und soll nur die Funktion von Condition-Variablen demonstrieren.

```
/* thread10.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

static int werte[10];
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

static void thread1 (void *arg) {
   int ret, i;

   printf ("\t->Thread %ld gestartet ...\n",
      pthread_self ());
   sleep (1);
   ret = pthread_mutex_lock (&mutex);
```



```
if (ret != 0) {
   printf ("Fehler bei lock in Thread:%ld\n",
      pthread_self());
   exit (EXIT_FAILURE);
}

/* Kritischer Codeabschnitt */
for (i = 0; i < 10; i++)
   werte[i] = i;
/* Kritischer Codeabschnitt Ende */

printf ("\t->Thread %ld sendet Bedingungsvariable\n",
   pthread_self());
pthread_cond_signal (&cond);

ret = pthread_mutex_unlock (&mutex);
if (ret != 0) {
   printf ("Fehler bei unlock in Thread: %ld\n",
      pthread_self ());
   exit (EXIT_FAILURE);
}
printf ("\t->Thread %ld ist fertig\n",pthread_self());
   }
   printf ("\t->Thread %ld ist fertig\n",pthread_self());
   pthread_exit ((void *) 0);
}

static void thread2 (void *arg) {
   int i;
   int summe = 0;

   printf ("\t->Thread %ld wartet auf Bedingungsvariable\n",
      pthread_self ());
   pthread_cond_wait (&cond, &mutex);
   printf ("\t->Thread %ld gestartet ...\n",
      pthread_self ());
   for (i = 0; i < 10; i++)
      summe += werte[i];
   printf ("\t->Thread %ld fertig\n",pthread_self());
   printf ("Summe aller Zahlen beträgt: %d\n", summe);
   pthread_exit ((void *) 0);
}

int main (void) {
   pthread_t th[2];
```

```
   printf("->Main-Thread %ld gestartet\n", pthread_self());

   pthread_create (&th[0], NULL, thread1, NULL);
   pthread_create (&th[1], NULL, thread2, NULL);

   pthread_join (th[0], NULL);
   pthread_join (th[1], NULL);

   printf("->Main-Thread %ld beendet\n", pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread10 thread10.c -pthread
$ ./thread10
->Main-Thread -1209416608 gestartet
        ->Thread -1209418832 gestartet ...
        ->Thread -1217811536 wartet auf Bedingungsvariable
        ->Thread -1209418832 sendet Bedingungsvariable
        ->Thread -1209418832 ist fertig
        ->Thread -1217811536 gestartet ...
        ->Thread -1217811536 fertig
Summe aller Zahlen beträgt: 45
->Main-Thread -1209416608 beendet
```

Hinweis

In diesem und auch in vielen anderen Beispielen haben wir das eine oder andere Mal auf eine Fehlerüberprüfung verzichtet, was Sie in der Praxis natürlich tunlichst vermeiden sollten. Allerdings würde ein »perfekt« geschriebenes Programm zu viele Buchseiten in Anspruch nehmen.

Dynamische Bedingungsvariablen

Natürlich können Sie Bedingungsvariablen auch dynamisch anlegen, wie dies häufig mit Datenstrukturen der Fall ist. Hierzu stehen Ihnen die folgenden Funktionen zur Verfügung:

```
#include <pthread.h>

int pthread_cond_init( pthread_cond_t *cond,
                       pthread_condattr_t *cond_attr );
int pthread_cond_destroy( pthread_cond_t *cond );
```


Mit `pthread_cond_init()` initialisieren Sie die Bedingungsvariable `cond` mit den über `attr` festgelegten Attributen (hierauf gehen wir im nächsten Abschnitt ein). Verwenden Sie für `attr` `NULL`, werden die standardmäßig voreingestellten Bedingungsvariablen verwendet. Freigeben können Sie die dynamisch angelegte Bedingungsvariable `cond` wieder mit der Funktion `pthread_cond_destroy()`.

Hier sehen Sie dasselbe Beispiel wie schon bei *thread10.c* zuvor, nur eben als dynamische Variante:

```
/* thread11.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

struct data {
   int werte[10];
   pthread_mutex_t mutex;
   pthread_cond_t cond;
};

static void thread1 (void *arg) {
   struct data *d=(struct data *)arg;
   int ret, i;

   printf ("\t->Thread %ld gestartet ...\n",
      pthread_self ());
   sleep (1);
   ret = pthread_mutex_lock (&d->mutex);
   if (ret != 0) {
      printf ("Fehler bei lock in Thread:%ld\n",
         pthread_self());
      exit (EXIT_FAILURE);
   }

   /* Kritischer Codeabschnitt */
   for (i = 0; i < 10; i++)
      d->werte[i] = i;
   /* Kritischer Codeausbschnitt Ende */

   printf ("\t->Thread %ld sendet Bedingungsvariable\n",
      pthread_self());
   pthread_cond_signal (&d->cond);
```

```
   ret = pthread_mutex_unlock (&d->mutex);
   if (ret != 0) {
      printf ("Fehler bei unlock in Thread: %ld\n",
         pthread_self ());
      exit (EXIT_FAILURE);
   }
   printf ("\t->Thread %ld ist fertig\n", pthread_self());
   pthread_exit ((void *) 0);
}

static void thread2 (void *arg) {
   struct data *d=(struct data *)arg;
   int i;
   int summe = 0;

   printf ("\t->Thread %ld wartet auf Bedingungsvariable\n",
      pthread_self ());
   pthread_cond_wait (&d->cond, &d->mutex);
   printf ("\t->Thread %ld gestartet ...\n",
      pthread_self ());
   for (i = 0; i < 10; i++)
      summe += d->werte[i];
   printf ("\t->Thread %ld fertig\n",pthread_self());
   printf ("Summe aller Zahlen beträgt: %d\n", summe);
   pthread_exit ((void *) 0);
}

int main (void) {
   pthread_t th[2];
   struct data *d;

   /* Speicher für die Struktur reservieren */
   d = malloc(sizeof(struct data));
   if(d == NULL) {
      printf("Konnte keinen Speicher reservieren ...!\n");
      exit(EXIT_FAILURE);
   }

   /* Bedingungsvariablen initialisieren */
   pthread_cond_init(&d->cond, NULL);

   printf("->Main-Thread %ld gestartet\n", pthread_self());
```



```
   pthread_create (&th[0], NULL, thread1, d);
   pthread_create (&th[1], NULL, thread2, d);

   pthread_join (th[0], NULL);
   pthread_join (th[1], NULL);

   /* Bedingungsvariable freigeben */
   pthread_cond_destroy(&d->cond);

   printf("->Main-Thread %ld beendet\n", pthread_self());
   return EXIT_SUCCESS;
}
```

Hierzu folgt noch ein typisches Anwendungsbeispiel. Wir simulieren ein Programm, das Daten empfängt, und erzeugen dabei zwei Threads. Jeder dieser beiden Threads wird mit `pthread_cond_wait()` in einen Wartezustand geschickt und wartet auf das Signal `pthread_cond_signal()` vom Haupt-Thread – ein einfaches Client-Server-Prinzip also. Der Haupt-Thread simuliert dann, er würde zwei Datenpakete an einen Client-Thread verschicken. Der Client-Thread simuliert anschließend, er würde die Datenpakete bearbeiten. Im Beispiel wurden statische Bedingungsvariablen verwendet. Die Ausgabe und der Ablauf des Programms sollten den Sachverhalt außerdem von selbst erklären:


```
/* thread12.c */
#define _MULTI_THREADED
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NUMTHREADS 2

static void checkResults (const char *string, int val) {
   if (val) {
      printf ("Fehler mit %d bei %s", val, string);
      exit (EXIT_FAILURE);
   }
}


static pthread_mutex_t dataMutex =
   PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t DatenVorhandenCondition =
   PTHREAD_COND_INITIALIZER;
static int DatenVorhanden = 0;
```

```
static int geteilteDaten  = 0;

static void *theThread (void *parm) {
   int rc;
   // Datenpaket in zwei Verarbeitungsschritten
   int retries = 2;

   printf ("\t->Client %ld: gestartet\n", pthread_self ());
   rc = pthread_mutex_lock (&dataMutex);
   checkResults ("pthread_mutex_lock()\n", rc);

   while (retries--) {
      while (!DatenVorhanden) {
         printf ("\t->Client %ld: Warte auf Daten ...\n",
            pthread_self ());
         rc = pthread_cond_wait ( &DatenVorhandenCondition,
                                  &dataMutex);
         if (rc) {
            printf ("Client %ld: pthread_cond_wait()"
                    " Fehler rc=%d\n", rc, pthread_self ());
            pthread_mutex_unlock (&dataMutex);
            exit (EXIT_FAILURE);
         }
      }
      printf("\t->Client %ld: Daten wurden gemeldet --->\n"
             "\t----> Bearbeite die Daten, solange sie "
             "geschützt sind (lock)\n", pthread_self ());
      if (geteilteDaten == 0) {
         DatenVorhanden = 0;
      }
   }//Ende while(retries--)

   printf ("Client %ld: Alles erledigt\n",
      pthread_self ());
   rc = pthread_mutex_unlock (&dataMutex);
   checkResults ("pthread_mutex_unlock()\n", rc);
   return NULL;
}

int main (int argc, char **argv) {
   pthread_t thread[NUMTHREADS];
   int rc = 0;
   // Gesamtanzahl der Datenpakete
```



```
int anzahlDaten = 4;
int i;
printf ("->Main-Thread %ld gestartet ...\n");
for (i = 0; i < NUMTHREADS; ++i) {
   rc=pthread_create (&thread[i], NULL, theThread, NULL);
   checkResults ("pthread_create()\n", rc);
}

/* Server-Schleife */
while (anzahlDaten--) {
   sleep (3); // Eine Bremse zum "Mitverfolgen"
   printf ("->Server: Daten gefunden\n");

   /* Schütze geteilte (shared) Daten und Flags */
   rc = pthread_mutex_lock (&dataMutex);
   checkResults ("pthread_mutex_lock()\n", rc);
   printf ("->Server: Sperre die Daten und gib eine "
           "Meldung an Consumer\n");
   ++geteilteDaten;     /* Füge "shared" Daten hinzu  */
   DatenVorhanden = 1;  /* ein vorhandenes Datenpaket */
   /* Client wieder aufwecken */
   rc = pthread_cond_signal (&DatenVorhandenCondition);
   if (rc) {
      pthread_mutex_unlock (&dataMutex);
      printf ("Server: Fehler beim Aufwecken von "
              "Client, rc=%d\n", rc);
      exit (EXIT_FAILURE);
   }
   printf("->Server: Gibt die gesperrten Daten"
          " wieder frei\n");
   rc = pthread_mutex_unlock (&dataMutex);
   checkResults ("pthread_mutex_lock()\n", rc);
}//Ende while(anzahlDaten--)

for (i = 0; i < NUMTHREADS; ++i) {
   rc = pthread_join (thread[i], NULL);
   checkResults ("pthread_join()\n", rc);
}
printf ("->Main-Thread ist fertig\n");
return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread12 thread12.c -pthread
$ ./thread12
->Main-Thread -1073743916 gestartet...
        ->Client -1209418832: gestartet
        ->Client -1209418832: Warte auf Daten ...
        ->Client -1217811536: gestartet
        ->Client -1217811536: Warte auf Daten ...
->Server: Daten gefunden
->Server: Sperre die Daten und gib eine Meldung an Consumer
->Server: Gibt die gesperrten Daten wieder frei
        ->Client -1209418832: Daten wurden gemeldet --->
        ----> Bearbeite die Daten, solange sie geschützt sind (lock)
        ->Client -1209418832: Daten wurden gemeldet --->
        ----> Bearbeite die Daten, solange sie geschützt sind (lock)
Client -1209418832: Alles erledigt
->Server: Daten gefunden
->Server: Sperre die Daten und gib eine Meldung an Consumer
->Server: Gibt die gesperrten Daten wieder frei
        ->Client -1217811536: Daten wurden gemeldet --->
        ----> Bearbeite die Daten, solange sie geschützt sind (lock)
        ->Client -1217811536: Daten wurden gemeldet --->
        ----> Bearbeite die Daten, solange sie geschützt sind (lock)
Client -1217811536: Alles erledigt
->Server: Daten gefunden
->Server: Sperre die Daten und gib eine Meldung an Consumer
->Server: Gibt die gesperrten Daten wieder frei
->Server: Daten gefunden
->Server: Sperre die Daten und gib eine Meldung an Consumer
->Server: Gibt die gesperrten Daten wieder frei
->Main-Thread ist fertig
```

Condition-Variablen-Attribute

Für die Attribute von Bedingungsvariablen stehen Ihnen folgende Funktionen zur Verfügung:

```
#include <pthread.h>

int pthread_condattr_init( pthread_condattr_t *attr );
int pthread_condattr_destroy( pthread_condattr_t *attr );
```


Allerdings machen diese Funktionen noch keinen Sinn, da Linux-Threads noch keine Attribute für Bedingungsvariablen anbieten. Diese Funktionen wurden dennoch implementiert, um den POSIX-Standard zu erfüllen.

12.5.3 Semaphore

Threads können auch mit Semaphoren synchronisiert werden. Wie Sie bereits in Kapitel 11, »IPC – Interprozesskommunikation«, erfahren haben, sind Semaphore nichts anderes als nicht negative Zählvariablen, die man beim Eintritt in einen kritischen Bereich dekrementiert und beim Verlassen wieder inkrementiert. Hierzu stehen Ihnen folgende Funktionen zur Verfügung:

```
#include <semaphore.h>

int sem_init( sem_t *sem, int pshared, unsigned int value );
int sem_wait( sem_t * sem );
int sem_trywait( sem_t * sem );
int sem_post( sem_t * sem );
int sem_getvalue( sem_t * sem, int * sval );
int sem_destroy( sem_t * sem );
```

Alle Funktionen geben bei Erfolg `0` oder bei einem Fehler `-1` zurück. Mit der Funktion `sem_init()` initialisieren Sie das Semaphor `sem` mit dem Anfangswert `value`. Geben Sie für den zweiten Parameter `pshared` einen Wert ungleich `0` an, kann das Semaphor gemeinsam von mehreren Prozessen und deren Threads verwendet werden. Wenn ein Wert gleich `0` verwendet wird, kann das Semaphor nur »lokal« für die Threads des aktuellen Prozesses verwendet werden.

Die Funktion `sem_wait()` wird zum Suspendieren eines aufrufenden Threads verwendet. `sem_wait()` wartet so lange, bis der Zähler `sem` einen Wert ungleich `0` besitzt. Sobald der Wert von `sem` ungleich `0` ist, also z. B. um 1 inkrementiert wurde, kann der suspendierende Thread mit seiner Ausführung fortfahren. Des Weiteren dekrementiert `sem_wait`, wenn diese Funktion »aufgeweckt« wurde, den Zähler des Semaphors wieder um 1. Im Gegensatz zu `sem_wait()` blockiert `sem_trywait()` nicht, wenn `sem` gleich `0` ist, und kehrt sofort mit dem Rückgabewert `-1` zurück.

Den Zähler des Semaphors `sem` können Sie mit der Funktion `sem_post()` um 1 erhöhen. Wollen Sie also einen anderen Thread aufwecken, der mit `sem_wait()` suspendiert wurde, müssen Sie nur `sem_post()` aus einem anderen Thread aufrufen. `sem_post()` ist eine nicht blockierende Funktion.

Wollen Sie überprüfen, welchen Wert das Semaphor gerade hat, können Sie die Funktion `sem_getvalue()` verwenden. Mit der Funktion `sem_destroy()` löschen Sie das Semaphor `sem` wieder.

Das folgende Beispiel entspricht dem Listing *thread10.c*, nur dass hier anstatt Bedingungsvariablen und Mutexen eben ein Semaphor verwendet wird. Mithilfe der Semaphore lässt sich eine Synchronisation (unserer Meinung nach) erheblich einfacher realisieren.

```
/* thread13.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>

static int werte[10];
sem_t sem;

static void thread1 (void *arg) {
   int ret, i, val;

   printf ("\t->Thread %ld gestartet ...\n",
      pthread_self ());

   /* Kritischer Codeabschnitt */
   for (i = 0; i < 10; i++)
      werte[i] = i;
   /* Kritischer Codeausschnitt Ende */

   /* Semaphor um 1 inkrementieren */
   sem_post(&sem);
   /* Aktuellen Wert ermitteln */
   sem_getvalue(&sem, &val);
   printf("\t->Semaphor inkrementiert (Wert: %d)\n", val);

   printf ("\t->Thread %ld ist fertig\n\n",pthread_self());
   pthread_exit ((void *) 0);
}

static void thread2 (void *arg) {
   int i;
   int summe = 0;

   /* Semaphor suspendiert, bis der Wert ungleich 0 ist */
   sem_wait(&sem);

   printf ("\t->Thread %ld gestartet ...\n",
```



```
      pthread_self ());
   for (i = 0; i < 10; i++)
      summe += werte[i];

   printf ("\t->Summe aller Zahlen beträgt: %d\n", summe);
   printf ("\t->Thread %ld fertig\n\n",pthread_self());
   pthread_exit ((void *) 0);
}

int main (void) {
   pthread_t th[2];
   int val;

   printf("->Main-Thread %ld gestartet\n", pthread_self());
   /* Semaphor initialisieren */
   sem_init(&sem, 0, 0);
   /* Aktuellen Wert abfragen */
   sem_getvalue(&sem, &val);
   printf("->Semaphor initialisiert (Wert: %d)\n\n", val);

   /* Mit Absicht andersherum */
   pthread_create (&th[1], NULL, thread2, NULL);
   pthread_create (&th[0], NULL, thread1, NULL);

   pthread_join (th[0], NULL);
   pthread_join (th[1], NULL);

   /* Aktuellen Wert abfragen */
   sem_getvalue(&sem, &val);
   printf("->Semaphor (Wert: %d)\n", val);
   /* Semphor löschen */
   sem_destroy(&sem);
   printf("->Semaphor gelöscht\n");
   printf("->Main-Thread %ld beendet\n", pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread13 thread13.c -pthread
$ ./thread13
->Main-Thread -1209416608 gestartet
->Semaphor initialisiert (Wert: 0)
```

```
        ->Thread -1217811536 gestartet ...
        ->Thread -1209418832 gestartet ...
        ->Summe aller Zahlen beträgt: 45
        ->Thread -1209418832 fertig

        ->Semaphor inkrementiert (Wert: 0)
        ->Thread -1217811536 ist fertig

->Semaphor (Wert: 0)
->Semaphor gelöscht
->Main-Thread -1209416608 beendet
```

12.5.4 Weitere Synchronisationstechniken im Überblick

Neben den hier vorgestellten Synchronisationsmechanismen bietet die `phtread`-Bibliothek Ihnen noch drei weitere an, auf die wir hier allerdings nur kurz eingehen:

- *RW-Locks* – Mit RW-Locks (Read-Write-Locks) können Sie es einrichten, dass mehrere Threads aus einem (shared) Datenbereich lesen, aber nur ein Thread zum selben Zeitpunkt darin etwas schreiben darf (*one-writer, many-readers*). Alle Funktionen dazu beginnen mit dem Präfix `pthread_rwlock_`.
- *Barrier* – Als Barrier bezeichnet man einen Punkt, der als (unüberwindbare) Barriere verwendet wird, die erst überwunden werden kann, wenn eine bestimmte Anzahl von Threads diese Barriere erreicht. Das funktioniert nach dem Prinzip der hohen Mauer bei den Pfadfindern, die man nur im Team (mit einer gewissen Anzahl von Personen) überwinden kann. Solange eine gewisse Anzahl von Threads nicht vorhanden ist, müssen eben alle Threads vor der Barriere warten. Soll z. B. ein bestimmter Thread erst ausgeführt werden, wenn viele andere Threads parallel mehrere Teilaufgaben erledigt haben, sind Barriers eine prima Synchronisationsmöglichkeit. Alle Funktionen zu den Barriers beginnen mit dem Präfix `pthread_barrier_`.
- *Spinlocks* – Spinlocks sind nur für Multiprozessorsysteme interessant. Das Prinzip ist dasselbe wie bei den Mutexen, nur dass – anders als bei den Mutexen – ein Thread, der auf einen Spinlock wartet, nicht die CPU freigibt, sondern eine sogenannte *Busy Loop* (Schleife) ausführt, bis der Spinlock frei ist. Dadurch bleibt Ihnen ein Kontextwechsel (*Context Switch*) erspart. Bei einem Kontextwechsel wird der Thread blockiert, und alle Informationen, die für das Weiterlaufen benötigt werden, müssen gespeichert werden. Wenn Sie viele Kontextwechsel in Ihrem Programm haben, ist dies eine Menge eingesparter Zeit, die man mit Spinlocks gewinnen kann. Alle Funktionen zu den Spinlocks beginnen mit dem Präfix `pthread_spin_`.

12.6 Threads abbrechen (canceln)

Wird ein Thread abgebrochen bzw. beendet, wurde bisher auch der komplette Thread beendet. Doch auch hierbei ist es möglich, auf eine Abbruchaufforderung zu reagieren. Hierzu sind drei Möglichkeiten vorhanden:

- `PTHREAD_CANCEL_DISABLE` – Damit legen Sie fest, dass ein Thread nicht abbrechbar ist. Dennoch bleiben Abbruchaufforderungen von anderen Threads nicht unbeachtet. Diese bleiben bestehen, und es kann gegebenenfalls darauf reagiert werden, wenn man den Thread mittels `PTHREAD_CANCEL_ENABLE` wieder in einen abbrechbaren Zustand versetzt.
- `PTHREAD_CANCEL_DEFERRED` – Diese Abbruchmöglichkeit ist die Standardeinstellung bei den Threads. Bei einem Abbruch fährt der Thread so lange fort, bis der nächste Abbruchpunkt erreicht wurde. Man spricht von einem »verzögerten« Abbruchpunkt. Einen solchen »Abbruchpunkt« stellten unter anderem Funktionen wie `pthread_cond_wait()`, `pthread_cond_timewait()`, `pthread_join()`, `pthread_testcancel()`, `sem_wait()`, `sigwait()`, `open()`, `close()`, `read()`, `write()` und noch viele weitere mehr dar.
- `PTHREAD_CANCEL_ASYNCHRONOUS` – Mit dieser Option wird der Thread gleich nach dem Eintreffen einer Abbruchaufforderung beendet. Hierbei handelt es sich um einen asynchronen Abbruch.

Im Folgenden sehen Sie die Funktionen, mit denen Sie einem anderen Thread einen Abbruch senden können, und Sie lernen, wie Sie die Abbruchmöglichkeiten selbst festlegen:

```
#include <pthread.h>

int pthread_cancel( pthread_t thread );
int pthread_setcancelstate( int state, int *oldstate );
int pthread_setcanceltype( int type, int *oldtype );
void pthread_testcancel( void );
```

Mit der Funktion `pthread_cancel()` schicken Sie dem Thread mit der ID `thread` eine Abbruchaufforderung. Ob der Thread gleich abbricht oder erst beim nächsten Abbruchpunkt, hängt davon ab, ob hier `PTHREAD_CANCEL_DEFERRED` (Standard) oder `PTHREAD_CANCEL_ASYNCHRONOUS` verwendet wird. Bevor sich der Thread beendet, werden noch – falls sie verwendet wurden – alle Exit-Handler-Funktionen ausgeführt.

Mit der Funktion `pthread_setcancelstate()` legen Sie fest, ob der Thread auf eine Abbruchaufforderung reagieren soll (`PTHREAD_CANCEL_ENABLE` = Default) oder nicht (`PTHREAD_CANCEL_DISABLE`). Im zweiten Parameter `oldstate` können Sie den zuvor eingestellten Wert für den Thread in der übergebenen Adresse sichern – oder, falls dieser Parameter nicht benötigt wird, `NULL` angeben.

Die Funktion `pthread_setcanceltype()` hingegen legt über den Parameter `type` fest, ob der Thread verzögert (`PTHREAD_CANCEL_DEFERRED` = Default) oder asynchron (`PTHREAD_CANCEL_`

ASYNCHRONOUS) beendet werden soll. Auch hier können Sie den alten Zustand des Threads in der Adresse oldtype sichern oder NULL verwenden.

Mit der Funktion pthread_testcancel() können Sie überprüfen, ob eine Abbruchaufforderung anliegt. Lag eine Abbruchbedingung vor, dann wird der Thread tatsächlich auch beendet. Sie können damit praktisch auch einen eigenen Abbruchpunkt festlegen.

Zunächst zeigen wir ein einfaches Beispiel zu pthread_cancel():

```
/* thread14.c */
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

pthread_t t1, t2, t3;
static int zufallszahl;

static void cancel_test1 (void) {
   /* Pseudo-Synchronisation, damit nicht ein Thread
      beendet wird, der noch gar nicht läuft. */
    sleep(1);
   if (zufallszahl > 25) {
      pthread_cancel (t3);
      printf ("(%d) : Thread %ld beendet %ld\n",
         zufallszahl, pthread_self(), t3);
   printf ("%ld zuende\n", pthread_self());
   pthread_exit ((void *) 0);

  }
}

static void cancel_test2 (void) {
   sleep(1);  // Pseudo-Synchronisation
   if (zufallszahl <= 25) {
      pthread_cancel (t2);
      printf ("(%d) : Thread %ld beendet %ld\n",
         zufallszahl, pthread_self(), t2);
     printf ("%ld zuende\n", pthread_self());
     pthread_exit ((void *) 0);

    }
}
```



```
static void zufall (void) {
   srand (time (NULL));
   zufallszahl = rand () % 50;
   pthread_exit (NULL);
}
int main (void) {
    if ((pthread_create (&t1, NULL, zufall, NULL)) != 0) {
       fprintf (stderr, "Fehler bei pthread_create ...\n");
       exit (EXIT_FAILURE);
    }
    if((pthread_create(&t2, NULL, cancel_test1, NULL))!=0) {
         fprintf (stderr, "Fehler bei pthread_create...\n");
         exit (EXIT_FAILURE);
    }
    if((pthread_create(&t3, NULL, cancel_test2, NULL))!=0) {
         fprintf (stderr, "Fehler bei pthread_create ...\n");
         exit (EXIT_FAILURE);
   }
   pthread_join (t1, NULL);
   pthread_join (t2, NULL);
   pthread_join (t3, NULL);
   return EXIT_SUCCESS;
}
```

Hier werden drei Threads erzeugt. Einer der Threads erzeugt eine Zufallszahl, die anderen zwei Threads reagieren entsprechend auf diese Zufallszahl. Je nachdem, ob die Zufallszahl kleiner bzw. größer als 25 ist, beendet der eine Thread den anderen mit pthread_cancel(). Wenn Sie das Programm ausführen, wird trotzdem, nach Beendigung eines der beiden Threads mit pthread_cancel(), zweimal das Folgende ausgegeben:

```
Thread n beendet
```

Wie kann das sein, wo Sie doch mindestens einen Thread beendet haben? Das ist die zweite Bedingung zur Beendigung von Threads, nämlich die Reaktion auf die Abbruchanforderungen. Die Standardeinstellung lautet hier ja PTHREAD_CANCEL_DEFERRED. Damit läuft der Thread noch bis zum nächsten Abbruchpunkt, in unserem Fall pthread_exit(). Wenn Sie einen Thread sofort abbrechen wollen bzw. müssen, müssen Sie mit pthread_setcanceltype() die Konstante PTHREAD_CANCEL_ASYNCHRONOUS setzen, z. B. in der main-Funktion mit:

```
if ((pthread_setcanceltype( PTHREAD_CANCEL_ASYNCHRONOUS,
                            NULL))!= 0) {
    fprintf(stderr, "Fehler bei pthread_setcanceltype\n");
    exit (EXIT_FAILURE);
}
```

In der Praxis kann man aber von asynchronen Abbrüchen abraten, da sie an jeder Stelle auftreten können. Wird z. B. `pthread_mutex_lock()` aufgerufen und tritt hier der Abbruch ein, nachdem der Mutex gesperrt wurde, hat man schnell einen Deadlock erzeugt. Einen asynchronen Abbruch sollte man in der Praxis nur verwenden, wenn die Funktion »asynchronsicher« ist, was mit `pthread_cancel()`, `pthread_setcancelstate()` und `pthread_setcanceltype()` nicht allzu viele Funktionen sind. Wenn Sie schon asynchrone Abbrüche verwenden müssen, dann eben immer, wenn ein Thread keine wichtigen Ressourcen beinhaltet, wie reservierten Speicherplatz (Memory Leaks), Sperren etc..

Ein besonders häufiger Anwendungsfall von `PTHREAD_CANCEL_DISABLE` sind kritische Codebereiche, die auf keinen Fall abgebrochen werden dürfen. Zum Beispiel ist dies sinnvoll bei wichtigen Einträgen in Datenbanken oder bei komplexen Maschinensteuerungen. Am besten realisiert man solche Codebereiche, indem man den kritischen Abschnitt als unabbrechbar einrichtet und gleich danach den alten Zustand wiederherstellt:

```
int oldstate;

/* Thread als unabbrechbar einrichten */
if ((pthread_setcancelstate( PTHREAD_CANCEL_DISABLE, &oldstate))!= 0) {
    fprintf(stderr, "Fehler bei pthread_setcancelstate\n");
    exit (EXIT_FAILURE);
}

/* ---------------------------------------- */
/* Hier kommt der kritische Codebereich rein */
/* ---------------------------------------- */

/* Alten Zustand des Threads wieder herstellen */
if ((pthread_setcancelstate(oldstat, NULL))!= 0) {
    fprintf(stderr, "Fehler bei pthread_setcancelstate\n");
    exit (EXIT_FAILURE);
}
```

Ein einfaches Beispiel hierzu:

```
/* thread15.c */
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

static void cancel_test (void) {
   int oldstate;
```



```
   /* Thread als unabbrechbar einrichten */
   if ((pthread_setcancelstate( PTHREAD_CANCEL_DISABLE,
                                &oldstate))!= 0) {
       printf("Fehler bei pthread_setcancelstate\n");
       exit (EXIT_FAILURE);
   }

   printf("Thread %ld im kritischen Codeabschnitt\n",
      pthread_self());
   sleep(5); // 5 Sekunden warten

   /* Alten Zustand des Threads wiederherstellen */
   if ((pthread_setcancelstate(oldstate, NULL))!= 0) {
       printf("Fehler bei pthread_setcancelstate\n");
       exit (EXIT_FAILURE);
   }

   printf("Thread %ld nach dem kritischen Codeabschnitt\n",
      pthread_self());
   pthread_exit ((void *) 0);
}

int main (void) {
   pthread_t t1;
   int *abbruch;

   printf("Main-Thread %ld gestartet\n", pthread_self());

   if((pthread_create(&t1, NULL, cancel_test, NULL)) != 0) {
      fprintf (stderr, "Fehler bei pthread_create ...\n");
      exit (EXIT_FAILURE);
   }
   /* Abbruchaufforderung an den Thread */
   pthread_cancel(t1);
   pthread_join (t1, &abbruch);
   if( abbruch == PTHREAD_CANCELED )
      printf("Thread %ld wurde abgebrochen\n", t1);
   printf("Main-Thread %ld beendet\n", pthread_self());
   return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread15 thread15.c -pthread
$ ./thread15
Main-Thread -1209416608 gestartet
Thread -1209418832 im kritischen Codeabschnitt
Thread -1209418832 nach dem kritischen Codeabschnitt
Thread -1209418832 wird abgebrochen
Main-Thread -1209416608 beendet
```

Ohne das Setzen von `PTHREAD_CANCEL_DISABLE` am Anfang des Threads `cancel_test` würde das Beispiel keine fünf Sekunden mehr warten und auch nicht mehr `Thread -1209418832 nach dem kritischen Codeabschnitt` ausgeben – am besten testen Sie dies, indem Sie das Verändern des Cancel-Status auskommentieren oder anstelle von `PTHREAD_CANCEL_DISABLE` die Konstante `PTHREAD_CANCEL_ENABLE` verwenden.

12.7 Erzeugen von Thread-spezifischen Daten (TSD-Data)

Bei einem Aufruf von Funktionen werden die lokalen Daten auf dem Stack abgelegt und auch wieder abgeholt. Bei den Threads kann man ja solch langlebige Daten entweder mit zusätzlichen Argumenten an die einzelnen Threads weitergeben oder aber in globalen Variablen speichern. Wenn Sie aber z. B. vorhaben, eine Bibliothek für den Multithread-Gebrauch zu schreiben, ist dies nicht mehr möglich, da man ja hierbei die Argumentzahl nicht mehr verändern kann, damit auch ältere Programme ohne Threads diese Bibliothek verwenden können. Und weil man auch nicht weiß, wie viele Threads die Bibliotheksfunktionen nutzen werden, kann man nun mal keine Aussage darüber machen, wie groß die globalen Daten sein sollen.

Um das Ganze vielleicht noch etwas einfacher zu erklären: Es ist einfach nicht möglich, dass globale und statische Variablen unterschiedliche Werte in den verschiedenen Threads haben können. Aus diesem Grund wurden sogenannte *Schlüssel* eingeführt – oder auch *thread-spezifische Daten* (kurz *TSD-Data*). Dabei handelt es sich um eine Art Zeiger, der immer auf die Daten verweist, die dem Thread gehören, der eben einen entsprechenden »Schlüssel« benutzt. Beachten Sie allerdings, dass hierbei immer mehrere Threads den gleichen »Schlüssel« benutzen – also hat nicht jeder Thread einen extra »Schlüssel«!

Dabei bekommt jeder Thread einen privaten Speicherbereich mit einem eigenen Schlüssel zugeteilt. Dies können Sie sich als ein Array von `void`-Zeigern vorstellen, auf die der Thread mit »seinem« Schlüssel zugreifen kann.

Hier sehen Sie die Funktionen, mit denen Sie Thread-spezifische Daten erzeugen können bzw. die mit Thread-spezifischen Daten arbeiten können.


```
#include <pthread.h>

int pthread_key_create(
   pthread_key_t *key, void (*destr_function) (void*) );
int pthread_key_delete( pthread_key_t key );
int pthread_setspecific(
   pthread_key_t key, const void *pointer );
void * pthread_getspecific(pthread_key_t key);
```

Mit `pthread_key_create()` erzeugen Sie einen neuen TSD-Schlüssel mit der Speicherstelle `key` des Schlüssels und geben als zweites Argument entweder `NULL` oder die Funktion an, um den Speicher der Daten wieder freizugeben; eine Exit-Handler-Funktion, wenn Sie so wollen.

Mit der Funktion `pthread_setspecific()` können Sie Daten mit dem TSD-Schlüssel assoziieren. Sie legen damit praktisch die TSD-Daten für den TSD-Schlüssel `key` über den Zeiger `pointer` fest.

Mit der Funktion `pthread_getspecific()` kann man die Daten aus dem TSD-Schlüssel auslesen.

Mit dem folgenden Beispiel werden `MAX_THREADS` Threads erzeugt, von denen jeder eine Thread-eigene Datei mittels TDS-Daten erzeugt. Im Beispiel wird nur protokolliert, dass der Thread gestartet und wieder beendet wurde. Zwischen den beiden Zeilen sollten Sie die eigentliche Arbeit des Threads eintragen. Fehler oder sonstige Meldungen dieser Arbeit können Sie ebenfalls wieder mit `thread_write` in die für den Thread vorgesehene Datei schreiben. Dass diese Funktion »nur« mit einem einfachen String aufgerufen werden kann, ist dem TSD-Schlüssel zu verdanken, der in der Funktion `thread_write` mittels `pthread_getspecific()` eingelesen wird – ein simples Grundgerüst eben, mit dem Sie ohne großen Aufwand Thread-eigene Logdateien verwenden können.

```
/* thread17.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 3

/* TSD-Datenschlüssel */
static pthread_key_t tsd_key;

/* Schreibt einen Text in eine Datei für
 * jeden aktuellen Thread */
void thread_write (const char* text) {
  /* TSD-Daten lesen */
  FILE* th_fp = (FILE*) pthread_getspecific (tsd_key);
  fprintf (th_fp, "%s\n", text);
```

```
}

/* Am Ende den Zeiger auf die Datei(en) schließen */
void thread_close (void* th_fp) {
  fclose ((FILE*) th_fp);
}

void* thread_tsd (void* args) {
  char th_fpname[20];
  FILE* th_fp;

  /* Einen Thread-spezifischen Dateinamen erzeugen */
  sprintf(th_fpname,"thread%d.thread",(int) pthread_self());
  /* Die Datei öffnen  */
  th_fp = fopen (th_fpname, "w");
  if( th_fp == NULL )
    pthread_exit(NULL);
  /* TSD-Daten zu TSD-Schlüssel festlegen */
  pthread_setspecific (tsd_key, th_fp);

  thread_write ("Thread wurde gestartet ...\n");

  /* Hier kommt die eigentliche Arbeit des Threads hin */

  thread_write("Thread ist fertig ...\n");
  pthread_exit(NULL);
}

int main (void) {
  int i;
  pthread_t threads[MAX_THREADS];

  /* Einen neuen TSD-Schlüssel erzeugen - Beim Ende eines
   * Threads wird die Funktion thread_close ausgeführt   */
  pthread_key_create (&tsd_key, thread_close);
  /* Threads erzeugen */
  for (i = 0; i < MAX_THREADS; ++i)
    pthread_create (&(threads[i]), NULL, thread_tsd, NULL);
  /* Auf die Threads warten  */
  for (i = 0; i < MAX_THREADS; ++i)
    pthread_join (threads[i], NULL);
  return EXIT_SUCCESS;
}
```


Das Programm bei der Ausführung:

```
$ gcc -o thread17 thread17.c -pthread
$ ./thread17
$ ls *.thread
thread-1209418832.thread  thread-1217811536.thread  thread-1226204240.thread
$ cat thread-1209418832.thread
Thread wurde gestartet ...

Thread ist fertig ...
```

12.8 pthread_once – Codeabschnitt einmal ausführen

In den Beispielen bisher haben Sie häufig mehrere Threads gestartet. Manchmal tritt aber eine Situation ein, in der man gewisse Vorbereitungen treffen muss – z. B. eine bestimmte Datei anlegen, weil mehrere Threads darauf zugreifen müssen oder man eine Bibliotheksroutine entwickelt. Wenn es nicht möglich ist, solche Vorbereitungen beim Start des Haupt-Threads zu treffen, sondern man dies in einem der Neben-Threads machen muss, dann benötigt man einen Mechanismus, mit dem eine Funktion exakt nur einmal ausgeführt wird, egal wie oft und von welchem Thread aus sie aufgerufen wurde.

Hier sehen Sie in einfaches Beispiel dafür, worauf wir hinauswollen:

```
/* thread18.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 3

void thread_once(void) {
   printf("Funktion thread_once() aufgerufen\n");
}

void* thread_func (void* args) {
   printf("Thread %ld wurde gestartet\n", pthread_self());
   thread_once();
   printf("Thread %ld ist fertig gestartet\n",
      pthread_self());
   pthread_exit(NULL);
}

int main (void) {
  int i;
```

```
  pthread_t threads[MAX_THREADS];
  /* Threads erzeugen */
  for (i = 0; i < MAX_THREADS; ++i)
    pthread_create (&(threads[i]), NULL, thread_func, NULL);
  /* Auf die Threads warten  */
  for (i = 0; i < MAX_THREADS; ++i)
    pthread_join (threads[i], NULL);
  return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread18 thread18.c -pthread
$ ./thread18
Thread -1209418832 wurde gestartet
Funktion thread_once() aufgerufen
Thread -1209418832 ist fertig gestartet
Thread -1217811536 wurde gestartet
Funktion thread_once() aufgerufen
Thread -1217811536 ist fertig gestartet
Thread -1226204240 wurde gestartet
Funktion thread_once() aufgerufen
Thread -1226204240 ist fertig gestartet
```

Beabsichtigt war in diesem Beispiel, dass die Funktion `thread_once` nur einmal aufgerufen wird. Aber wie das für Thread-Programme eben üblich ist, wird die Funktion bei jedem Thread aufgerufen. Hier kann man zwar mit den Thread-spezifischen Synchronisationen nachhelfen, aber die Thread-Bibliothek bietet Ihnen mit der Funktion `pthread_once()` eine Funktion an, die einen bestimmten Code nur ein einziges Mal ausführt:

```
#include <pthread.h>

pthread_once_t once_control = PTHREAD_ONCE_INIT;
int  pthread_once(
pthread_once_t  *once_control, void (*init_routine) (void));
```

Vor der Ausführung von `pthread_once()` muss man eine statische Variable mit der Konstante `PTHREAD_ONCE_INIT` initialisieren, bevor man diese an `pthread_once()` (erster Parameter) übergibt. Als zweites Argument übergeben Sie `pthread_once()`, die Funktion, die den einmal auszuführenden Code enthält. Wird dieser einmal auszuführende Code ausgeführt, wird dies in der Variablen `once_control` vermerkt, sodass diese Funktion kein zweites Mal mehr ausgeführt werden kann. Hierzu sehen Sie das Beispiel *thread18.c* nochmals mit `phtread_once()`:


```
/* thread19.c */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define MAX_THREADS 3

static pthread_once_t once = PTHREAD_ONCE_INIT;

void thread_once(void) {
   printf("Funktion thread_once() aufgerufen\n");
}

void* thread_func (void* args) {
   printf("Thread %ld wurde gestartet\n", pthread_self());
   pthread_once(&once, thread_once);
   printf("Thread %ld ist fertig gestartet\n",
      pthread_self());
   pthread_exit(NULL);
}

int main (void) {
  int i;
  pthread_t threads[MAX_THREADS];
  /* Threads erzeugen */
  for (i = 0; i < MAX_THREADS; ++i)
    pthread_create (&(threads[i]), NULL, thread_func, NULL);
  /* Auf die Threads warten  */
  for (i = 0; i < MAX_THREADS; ++i)
    pthread_join (threads[i], NULL);
  return EXIT_SUCCESS;
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread19 thread19.c -pthread
$ ./thread19
Thread -1209418832 wurde gestartet
Funktion thread_once() aufgerufen
Thread -1209418832 ist fertig gestartet
Thread -1217811536 wurde gestartet
Thread -1217811536 ist fertig gestartet
Thread -1226204240 wurde gestartet
Thread -1226204240 ist fertig gestartet
```

Jetzt wird die mit `pthread_once()` eingerichtete Funktion tatsächlich nur noch einmal ausgeführt.

12.9 Thread-safe (thread-sichere Funktionen)

Die Thread-Programmierung kann nur mit Bibliotheken realisiert werden, die als thread-sicher (*thread-safe*) gelten. Denn auch die Bibliotheken müssen die parallele Ausführung erlauben, um nicht ins Straucheln zu geraten. Mit der Einführung von Glibc 2.0 wurden die Linux-Threads in den Bibliotheken implementiert und müssen nicht extra besorgt werden. Somit ist die Funktion `strcmp()` – auch wenn sie schon über 15 Jahre alt ist – thread-sicher.

`readdir()` hingegen ist z. B. nicht thread-safe. Das Problem mit `readdir()` ist Folgendes: Wenn mehrere Threads denselben `DIR`-Zeiger verwenden, überschreiben sie immer den aktuellen Rückgabewert des Threads, den Sie zuvor erhalten haben. Als Alternative für Funktionen, die nicht als thread-sicher gelten (auch wenn es nicht sehr viele sind), wurden thread-sichere Schnittstellen eingeführt, die in der Regel an der Endung `_r` (reentrante Funktionen) zu erkennen sind. Die thread-sichere Alternative zu `readdir()` lautet somit `readdir_r()`.

Die Endung `_r` zeigt Ihnen außerdem nicht nur, dass die Funktion thread-sicher ist, sondern auch, dass diese Funktion keinen internen statischen Puffer verwendet (der beim Aufruf derselben Funktion in mehreren Threads jedes Mal überschrieben wird). Tabelle 12.1 enthält eine kurze Liste einiger gängiger reentranter Funktionen, deren genauere Syntax und Verwendung Sie bitte der entsprechenden Man-Page entnehmen.

Nicht thread-sicher	thread-sicher	Bedeutung
`getlogin`	`getlogin_r`	Loginname ermitteln
`ttyname`	`ttyname_r`	Terminalpfadname ermitteln
`readdir`	`readdir_r`	Verzeichniseinträge lesen
`strtok`	`strtok_r`	String anhand von Tokens zerlegen
`asctime`	`asctime_r`	Zeitfunktion
`ctime`	`ctime_r`	Zeitfunktion
`gmtime`	`gmtime_r`	Zeitfunktion
`localtime`	`localtime_r`	Zeitfunktion
`rand`	`rand_r`	(Pseudo-)Zufallszahlen generieren
`getpwuid`	`getpwuid_r`	Eintrag in */etc/passwd* erfragen (via UID)

Tabelle 12.1 Nicht thread-sichere Funktionen und die Alternativen

Nicht thread-sicher	thread-sicher	Bedeutung
`getpwnam`	`getpwnam_r`	Eintrag in */etc/passwd* erfragen (via Loginname)
`getgrgid`	`getgrgid_r`	Eintrag in */etc/group* erfragen (via GID)
`getgrnam`	`getgrnam_r`	Eintrag in */etc/group* erfragen (via Gruppenname)

Tabelle 12.1 Nicht thread-sichere Funktionen und die Alternativen (Forts.)

12.10 Threads und Signale

Signale lassen sich auch mit den Threads realisieren, nur muss man hierbei Folgendes beachten:

- Signale, die von der Hardware gesendet werden, bekommt immer der Thread, der das Hardware-Signal gesendet hat.
- Jedem Thread kann eine eigene Signalmaske zugeordnet werden. Allerdings gelten Signale, die mit `sigaction()` eingerichtet wurden, prozessweit für alle Threads.

Zur Verwendung von Signalen mit den Threads werden folgende Funktionen benötigt:

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

int pthread_sigmask( int how, const sigset_t  *newmask, sigset_t  *old_mask );
int pthread_kill( pthread_t thread, int signo );

int sigwait( const sigset_t *set, int *sig );
int sigwaitinfo( const sigset_t *set, siginfo_t *info );
int sigtimedwait( const  sigset_t  *set,  siginfo_t *info,
                  const struct timespec timeout );
```

Mit der Funktion `pthread_sigmask()` können Sie eine Thread-Signalmaske erfragen oder ändern. Im Prinzip entspricht diese Funktion der von `sigprocmask()`, nur eben auf Threads und nicht auf Prozesse bezogen. Abgesehen von den Signalen `SIGKILL` und `SIGSTOP` können Sie auch hierzu alle bekannten Signale verwenden. Schlägt die Funktion `pthread_sigmask()` fehl, wird die Signalmaske des Threads nicht verändert.

Wir empfehlen Ihnen, für die Funktion `pthread_sigmask()`(falls nötig) nochmals Kapitel 10, »Signale«, durchzulesen – da das Prinzip ähnlich wie zwischen den Prozessen funktioniert. Als erstes Argument für `how` wird eine Angabe erwartet, wie Sie die Signale verändern wollen. Mögliche Konstanten hierfür sind `SIG_BLOCK`, `SIG_UNBLOCK` und `SIG_SETMASK`. Als zweites Argu-

ment ist ein Zeiger auf einen Satz von Signalen nötig, der die aktuelle Signalmaske ergänzt, sie entfernt oder die Signalmaske ganz übernimmt. Hierfür kann auch `NULL` angegeben werden. Der dritte Parameter ist ein Zeiger auf die aktuelle Signalmaske. Hiermit können Sie entweder die aktuelle Signalmaske abfragen oder, wenn Sie mit dem zweiten Parameter eine neue Signalmaske einrichten, die alte Signalmaske sichern. Aber auch der dritte Parameter kann `NULL` sein. Wenn ein Thread einen weiteren Thread erzeugt, erbt dieser ebenfalls die Signalmaske. Wollen Sie also, dass alle Threads diese Signalmaske erben, sollten Sie vor der Erzeugung der Threads im Haupt-Thread die Signalmaske setzen.

Mit der Funktion `pthread_kill()` senden Sie dem Thread `thread` das Signal `signo`. Dazu möchten wir noch die Besonderheiten mit den Signalen `SIGKILL`, `SIGTERM` und `SIGSTOP` erläutern. Diese drei Signale gelten weiterhin prozessweit – senden Sie z. B. mit `phtread_kill()` das Signal `SIGKILL` an einen Thread, wird der komplette Prozess beendet, nicht nur der Thread. Ebenso sieht dies mit dem Signal `SIGSTOP` aus – hier wird der ganze Prozess (mit allen laufenden Threads) angehalten, bis ein anderer Prozess (nicht Thread) `SIGCONT` an den angehaltenen Prozess sendet.

Mit `sigwait()` halten Sie einen Thread so lange an, bis eines der Signale aus der Menge `set` gesendet wird. Die Signalnummer wird noch in `sig` geschrieben, bevor der Thread seine Ausführung fortsetzt. Wurde dem Signal ein Signalhandler zugeteilt, wird nichts in `sig` geschrieben.

Das folgende einfache Beispiel demonstriert Ihnen die Verwendung von Signalen in Verbindung mit Threads:

```
/* thread20 */
#include <stdio.h>
#include <pthread.h>
#include <signal.h>

pthread_t tid2;

void int_handler(int dummy) {
   printf("SIGINT erhalten von TID(%d)\n", pthread_self());
}

void usr1_handler(int dummy) {
   printf("SIGUSR1 erhalten von TID(%d)\n", pthread_self());
}

void *thread_1(void *dummy) {
   int sig, status, *status_ptr = &status;
   sigset_t sigmask;
```



```
   /* Kein Signal blockieren - SIG_UNBLOCK */
   sigfillset(&sigmask);
   pthread_sigmask(SIG_UNBLOCK, &sigmask, (sigset_t *)0);
   sigwait(&sigmask, &sig);

   switch(sig) {
      case SIGINT: int_handler(sig); break;
      default    : break;
   }
   printf("TID(%d) sende SIGINT an %d\n",
      pthread_self(), tid2);
   /* blockiert von tid2 */
   pthread_kill(tid2, SIGINT);
   printf("TID(%d) sende SIGUSR1 an %d\n",
      pthread_self(), tid2);
   /* nicht blockiert von tid2 */
   pthread_kill(tid2, SIGUSR1);

   pthread_join(tid2, (void **)status_ptr);
   printf("TID(%d) Exit-Status = %d\n", tid2, status);

   printf("TID(%d) wird beendet\n", pthread_self());
   pthread_exit((void *)NULL);
}

void *thread_2(void *dummy) {
   int sig;
   sigset_t sigmask;

   /* Alle Bits auf null setzen */
   sigemptyset(&sigmask);
   /* Signal SIGUSR1 nicht blockieren ... */
   sigaddset(&sigmask, SIGUSR1);
   pthread_sigmask(SIG_UNBLOCK, &sigmask, (sigset_t *)0);
   sigwait(&sigmask, &sig);

   switch(sig) {
      case SIGUSR1: usr1_handler(sig); break;
      default     : break;
   }
   printf("TID(%d) wird beendet\n", pthread_self());

   pthread_exit((void *)99);
```

```
}

int main(void) {
   pthread_t tid1;
   pthread_attr_t attr_obj;
   void *thread_1(void *), *thread_2(void *);
   sigset_t sigmask;
   struct sigaction action;

   /* Signalmaske einrichten - alle Signale im *
    * Haupt-Thread blockieren */
   sigfillset(&sigmask);       /* Alle Bits ein ...*/
   pthread_sigmask(SIG_BLOCK, &sigmask, (sigset_t *)0);

   /* Setup Signal-Handler für SIGINT & SIGUSR1 */
   action.sa_flags = 0;
   action.sa_handler = int_handler;
   sigaction(SIGINT, &action, (struct sigaction *)0);
   action.sa_handler = usr1_handler;
   sigaction(SIGUSR1, &action, (struct sigaction *)0);

   pthread_attr_init(&attr_obj);
pthread_attr_init(&attr_obj);
pthread_attr_setdetachstate( &attr_obj,
                             PTHREAD_CREATE_DETACHED );
pthread_create(&tid1, &attr_obj, thread_1, (void *)NULL);
printf("TID(%d) erzeugt\n", tid1);

pthread_attr_setdetachstate( &attr_obj,
                             PTHREAD_CREATE_JOINABLE);
pthread_create(&tid2, &attr_obj, thread_2, (void *)NULL);
printf("TID(%d) erzeugt\n", tid2);

sleep(1); // Kurze Pause ...

printf("Haupt-Thread(%d) sendet SIGINT an TID(%d)\n",
   pthread_self(), tid1);
pthread_kill(tid1, SIGINT);
printf("Haupt-Thread(%d) sendet SIGUSR1 an TID(%d)\n",
   pthread_self(), tid1);
pthread_kill(tid1, SIGUSR1);

printf("Haupt-Thread(%d) wird beendet\n",
   pthread_self());
```



```
   // Beendet nicht den Prozess!!!
   pthread_exit((void *)NULL);
}
```

Das Programm bei der Ausführung:

```
$ gcc -o thread20 thread20.c -pthread
$ ./thread20
TID(-1209418832) erzeugt
TID(-1217815632) erzeugt
Haupt-Thread(-1209416608) sendet SIGINT an TID(-1209418832)
Haupt-Thread(-1209416608) sendet SIGUSR1 an TID(-1209418832)
Haupt-Thread(-1209416608) wird beendet
SIGUSR1 erhalten von TID(-1209418832)
SIGINT erhalten von TID(-1209418832)
TID(-1209418832) sende SIGINT an -1217815632
TID(-1209418832) sende SIGUSR1 an -1217815632
SIGUSR1 erhalten von TID(-1217815632)
TID(-1217815632) wird beendet
TID(-1217815632) Exit-Status = 99
TID(-1209418832) wird beendet
```


Dieses Beispiel beinhaltet drei Threads (inklusive des Haupt-Threads). Im Haupt-Thread wird die Signalmaske so eingerichtet, dass alle Signale im Haupt-Thread blockiert (`SIGBLOCK`) werden. Als Nächstes werden Signalhandler für `SIGINT` und `SIGUSR1` eingerichtet. `thread_1` wird von den Threads abgehängt erzeugt (*detached*), und `thread_2` wird nicht von den anderen Threads abgehängt. Dann werden im Haupt-Thread die Signale `SIGINT` und `SIGUSR1` an den `thread_1` gesendet, und der Haupt-Thread beendet sich.

`thread_1` (da abgehängt, gerne auch *Daemon*-Thread genannt) hebt die Blockierung der Signale auf (`SIG_UNBLOCK`) und wartet auf Signale. Im Beispiel wurde ja bereits zuvor `SIGINT` und `SIGUSR1` vom Haupt-Thread gesendet, was die Ausgabe des Signalhandlers auch bestätigt. Sobald also `thread_1` seine Signale bekommen hat, sendet er die Signale `SIGINT` und `SIGUSR1` an `thread_2`, wartet (mittels `pthread_join()`), bis `thread_2` sich beendet, und gibt den Exit-Status von `thread_2` aus, bevor `thread_1` sich ebenfalls beendet.

`thread_1` hingegen hebt nur die Blockierung für `SIGUSR1` auf – alle anderen Signale werden ja durch die Weitervererbung des Haupt-Threads weiterhin blockiert. Anschließend wartet `thread_2` auf das Signal. Trifft `SIGUSR1` ein, wird der Signalhandler ausgeführt und der Thread mit einem Rückgabewert beendet (auf den `thread_1` ja mit `pthread_join()` wartet).

12.11 Zusammenfassung und Ausblick

Threads ist in den letzten Jahren zu einem ziemlichen Buzzword geworden, also ein Begriff, um den man nicht herumkommt, wenn man Vorträge hält. Die Programmierer sind in zwei Lager gespalten – in die Befürworter und die Gegner von Threads. Zugegeben, Threads haben je nach Anwendungsfall auch ihre Macken. Gerade in der Netzwerkprogrammierung sorgen Threads bezüglich der Skalierbarkeit immer wieder für Diskussionsstoff. Das Problem besteht in diesem Fall darin, dass es häufig ein hartes Limit der gleichzeitigen Threads gibt.

Wie dem auch sei, – die Autoren von namhafter Software (wie etwa von den Serveranwendungen BIND, Apache etc.) lassen sich nicht aufhalten, Thread-Unterstützung in ihre Software einzubauen – ganz einfach auch deswegen, weil Serverbetreiber es nicht einsehen, mehrere CPU-Kerne zu haben, die nichts tun, während die Maschine unter der Anzahl der Zugriffe fast zusammenbricht.

Ob Sie nun Threads in Ihre Software einbauen wollen oder nicht, müssen Sie letztendlich selbst entscheiden. Das hängt größtenteils auch von der Art der Software ab. Am besten wäre es sicherlich, wenn Sie Ihre Software mit Thread-Unterstützung anbieten – sprich, der Anwender kann diese bei Bedarf aktivieren bzw. testen.

Mit diesem Kapitel haben Sie auf jeden Fall ein fundiertes Polster an Wissen, sodass Sie Software mit Thread-Unterstützung schreiben können. Aber wie bei fast jedem Thema in diesem Buch gibt es immer noch einiges mehr, das wir zu den Threads noch hätten schreiben können. Aber immerhin haben wir im Vergleich zur ersten Auflage den Umfang des Kapitels bereits verdoppelt.