

## Leseprobe

»Linux-Unix-Programmierung« deckt alle Bereiche ab, die für die Entwicklung von Anwendungen für die Linux-Welt wichtig sind, C-Kenntnisse vorausgesetzt. Diese Leseprobe enthält das vollständige Kapitel 12, »Threads«. Außerdem können Sie das gesamte Inhaltsverzeichnis und den Index einsehen.

-  **Kapitel 12: »Threads«**
-  **Inhaltsverzeichnis**
-  **Index**
-  **Die Autoren**
-  **Leseprobe weiterempfehlen**

Jürgen Wolf, Klaus-Jürgen Wolf  
**Linux-Unix-Programmierung –  
Das umfassende Handbuch**

1.435 Seiten, gebunden, 4. Auflage 2016  
49,90 Euro, ISBN 978-3-8362-3772-7

 [www.rheinwerk-verlag.de/3854](http://www.rheinwerk-verlag.de/3854)

# 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 -pthread
```

Der zusätzliche Parameter `-pthread` setzt gleichzeitig verschiedene Präprozessoreinstellungen und fügt die Pthreads-Bibliothek der Linkliste hinzu. Und wenn Sie in der Literatur einmal `-lpthread` 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 Duplikation 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 verkapelt – 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.<sup>1</sup>

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.

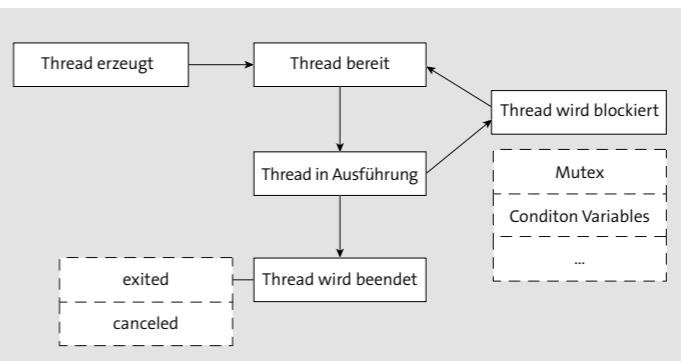


Abbildung 12.1 Zustände von Threads

- ▶ *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.

<sup>1</sup> »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 ausführlich 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));
    sprintf (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 fortgefahrene. 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_getdetachstate( pthread_attr_t *attribute,
                                int detachstate );

```

```
int pthread_attr_setdetachstate( 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_getdetachstate()` können Sie das `detached`-Attribut erfragen, und mit `pthread_attr_setdetachstate()` 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 Policies*):

- ▶ `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, unvorseehbare 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.

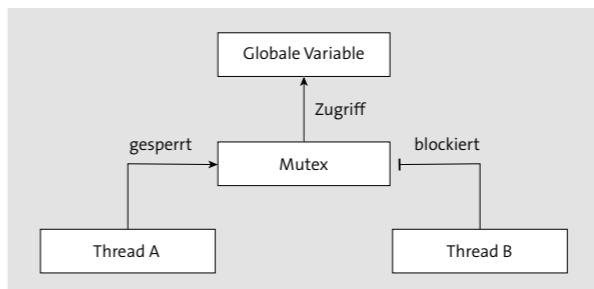


Abbildung 12.2 Nur ein Thread kann einen Mutex sperren.

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_cond_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 `pthread_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_broadcast()` verwenden.

Natürlich gibt es auch noch eine Funktion, mit der Sie – im Gegensatz zu `pthread_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 ());
    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 -lpthread
$ ./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_unlock()\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 ());
}
```

```
    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);
    /* Semaphor 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

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

->Semaphore (Wert: 0)
->Semaphore 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 das-selbe 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 Ein-treffen 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 unabrechbar einrichtet und gleich danach den alten Zustand wiederherstellt:

```
int oldstate;

/* Thread als unabrechbar 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 unabrechbar 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 ein 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) über gibt. 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 `pthread_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` (reentrant 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 der selben Funktion in mehreren Threads jedes Mal überschrieben wird). Tabelle 12.1 enthält eine kurze Liste einiger gängiger reentrant Funktionen, deren genauere Syntax und Verwendung Sie bitte der entsprechenden Man-Page entnehmen.

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

Tabelle 12.1 Nicht thread-sichere Funktionen und die Alternativen

Nicht thread-sicher	thread-sicher	Bedeutung
<code>getpwnam</code>	<code>getpwnam_r</code>	Eintrag in <code>/etc/passwd</code> erfragen (via Loginname)
<code>getgrgid</code>	<code>getgrgid_r</code>	Eintrag in <code>/etc/group</code> erfragen (via GID)
<code>getgrnam</code>	<code>getgrnam_r</code>	Eintrag in <code>/etc/group</code> 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 `pthread_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(SIG_BLOCK, &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(SIG_BLOCK, &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(SIG_BLOCK, &sigmask, (sigset_t *)0);

    /* 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_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 (SIG\_BLOCK) 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.

## Auf einen Blick

1	Einführung .....	25
2	Laufzeitumgebungen .....	29
3	Dynamische Daten in C .....	47
4	E/A-Funktionen .....	57
5	Attribute von Dateien und Verzeichnissen .....	145
6	Zugriff auf Systeminformationen .....	169
7	Devices – eine einfache Verbindung zur Hardware .....	193
8	System- und Benutzerdateien .....	219
9	Dämonen, Zombies und Prozesse .....	241
10	Signale .....	325
11	IPC – Interprozesskommunikation .....	349
12	Threads .....	439
13	Populäre Programmiertechniken unter Linux .....	505
14	Netzwerkprogrammierung .....	533
15	Abgesicherte Netzwerkverbindungen .....	663
16	Relationale Datenbanken: MySQL, PostgreSQL und SQLite .....	687
17	GTK+ .....	839
18	Werkzeuge für Programmierer .....	983
19	Shell-Programmierung .....	1075

# Inhalt

<b>1 Einführung</b>	25
1.1 Anforderung an den Leser .....	25
1.2 Das Betriebssystem .....	26
1.3 Schreibkonventionen .....	26
1.4 Notationsstil .....	27
1.5 Weitere Hilfen .....	28
<b>2 Laufzeitumgebungen</b>	29
2.1 Historisches .....	29
2.1.1 Von UNIX ... .....	29
2.1.2 ... zu Linux .....	31
2.2 Distributionen und ihre Unterschiede .....	33
2.3 Die GNU-Toolchain .....	34
2.3.1 Das GNU Build System .....	34
2.3.2 Die GNU Compiler Collection (GCC) .....	34
2.3.3 GNU Binutils (binutils) .....	35
2.3.4 GNU Make (gmake) .....	35
2.3.5 Der GNU Debugger (gdb) .....	35
2.4 Paketmanagement .....	36
2.5 Der Compiler GCC – eine kurze Einführung .....	37
2.5.1 GCC, erhöre uns – der Aufruf .....	37
2.5.2 Was befiehlst du, Meister? .....	38
2.5.3 Klassifikation der Dateitypen .....	41
2.6 POSIX, X/OPEN und ANSI C .....	42
2.6.1 POSIX .....	42
2.6.2 X/OPEN .....	44
2.6.3 ANSI C .....	44
2.6.4 Weitere Standards .....	45

<b>3 Dynamische Daten in C</b>	47	4.4.3 Formatierte Ausgabe .....	109
<b>3.1 Speicher anfordern</b>	47	4.4.4 Formatierte Eingabe .....	110
<b>3.2 Speicher verschieben und löschen</b>	49	4.4.5 Binäres Lesen und Schreiben .....	111
<b>3.3 Zeichenketten und -funktionen</b>	49	4.4.6 Zeichen- und zeilenweise Ein-/Ausgabe .....	112
3.3.1 strdup() und strndup() bzw. strdupa() und strndupa()	49	4.4.7 Status der Ein-/Ausgabe überprüfen .....	113
3.3.2 strcpy(), strncpy(), strcat() und strncat()	50	4.4.8 Stream positionieren .....	113
3.3.3 strchr() und strrchr()	50	4.4.9 Puffer kontrollieren .....	115
3.3.4 strpbrk()	50	4.4.10 Datei löschen und umbenennen .....	117
3.3.5 strtok() und strtok_r()	51	4.4.11 Temporäre Dateien erstellen .....	117
<b>3.4 Zeichenkodierung</b>	51	<b>4.5 Mit Verzeichnissen arbeiten</b> .....	118
3.4.1 Wide Characters	52	4.5.1 Ein neues Verzeichnis anlegen – mkdir()	119
3.4.2 UTF-8	52	4.5.2 In ein Verzeichnis wechseln – chdir(), fchdir()	120
<b>3.5 Müllsammler, Kanarienvögel und Sicherheit</b>	54	4.5.3 Ein leeres Verzeichnis löschen – rmdir()	122
<b>4 E/A-Funktionen</b>	57	4.5.4 Das Format eines Datei-Eintrags in struct dirent .....	122
<b>4.1 Elementare E/A-Funktionen</b>	57	4.5.5 Einen Verzeichnissstream öffnen – opendir()	124
<b>4.2 Filedescriptor</b>	58	4.5.6 Lesen aus dem DIR-Stream mit opendir() und Schließen des DIR-Streams mit closedir()	126
4.2.1 Verwaltung für offene Deskriptoren	59	4.5.7 Positionieren des DIR-Streams .....	129
<b>4.3 Funktionen, die den Filedescriptor verwenden</b>	61	4.5.8 Komplettes Verzeichnis einlesen – scandir()	130
4.3.1 Datei öffnen – open()	61	4.5.9 Ganze Verzeichnisbäume durchlaufen – nftw()	135
4.3.2 Anlegen einer neuen Datei – creat()	67	<b>4.6 Fehlerbehandlung</b> .....	140
4.3.3 Datei schließen – close()	68	<b>4.7 Temporäre Dateien und Verzeichnisse</b> .....	143
4.3.4 Schreiben von Dateien – write()	68	<b>4.8 Ausblick</b> .....	143
4.3.5 Lesen von Dateien – read()	72		
4.3.6 Schreib-/Lesezeiger positionieren – lseek()	74		
4.3.7 Duplizieren von Filedeskriptoren – dup() und dup2()	77		
4.3.8 Ändern oder Abfragen der Eigenschaften eines Filedeskriptors – fcntl()	79		
4.3.9 Record Locking – Sperren von Dateien einrichten	85		
4.3.10 Multiplexing E/A – select()	96		
4.3.11 Unterschiedliche Operationen – ioctl()	99		
4.3.12 Lesen und Schreiben mehrerer Puffer – writev() und readv()	100		
4.3.13 Übersicht zu weiteren Funktionen, die den Filedescriptor verwenden	102		
<b>4.4 Standard-E/A-Funktionen</b>	106	<b>5 Attribute von Dateien und Verzeichnissen</b> .....	145
4.4.1 Der FILE-Zeiger	106	<b>5.1 Struktur stat</b> .....	145
4.4.2 Öffnen und Schließen von Dateien	107	5.1.1 Dateiart und Zugriffsrechte einer Datei erfragen – st_mode .....	146
		5.1.2 User-ID-Bit und Group-ID-Bit – st_uid und st_gid .....	150
		5.1.3 Inode ermitteln – st_ino .....	151
		5.1.4 Linkzähler – st_nlink .....	151
		5.1.5 Größe der Datei – st_size .....	156
		5.1.6 st_atime, st_mtime, st_ctime .....	158
		<b>5.2 Erweiterte Attribute</b> .....	161
		5.2.1 Access Control Lists (Zugriffskontrolllisten) .....	161
		5.2.2 Erweiterte benutzerdefinierte Attribute .....	167

<b>6 Zugriff auf Systeminformationen</b>	169
<b>6.1 Das /sys-Dateisystem (sysfs)</b>	169
<b>6.2 Das /proc-Dateisystem (procfs)</b>	171
<b>6.3 Informationen aus dem /proc-Verzeichnis herausziehen</b>	171
<b>6.4 Hardware- bzw. Systeminformationen ermitteln</b>	173
6.4.1 CPU-Informationen – /proc/cpuinfo	174
6.4.2 Geräteinformationen – /proc/devices	175
6.4.3 Speicherauslastung – /proc/meminfo	175
6.4.4 Weitere Hardware-Informationen zusammengefasst	175
<b>6.5 Prozessinformationen</b>	179
6.5.1 /proc/\$pid/cmdline	180
6.5.2 /proc/\$pid/environ	180
6.5.3 /proc/self	181
6.5.4 /proc/\$pid/fd/	182
6.5.5 /proc/\$pid/statm	183
<b>6.6 Kernel-Informationen</b>	184
6.6.1 /proc/locks	190
6.6.2 /proc/modules	190
<b>6.7 Filesysteme</b>	191
6.7.1 /proc/mounts	191
<b>6.8 Weiterführendes</b>	192
<b>7 Devices – eine einfache Verbindung zur Hardware</b>	193
<b>7.1 Die Gerätedateitypen</b>	193
<b>7.2 Die Gerätedateinummern</b>	194
<b>7.3 Historisches</b>	195
<b>7.4 Zugriff auf die Gerätedateien</b>	196
<b>7.5 Gerätenamen</b>	197
<b>7.6 Spezielle Gerätedateien</b>	200
<b>7.7 Gerätedateien in der Praxis einsetzen</b>	201
7.7.1 CD auswerfen und wieder schließen	203
7.7.2 CD-ROM-Fähigkeiten	204

<b>7.7.3 Audio-CD abspielen – komplett und einzelne Tracks – Pause, Fortfahren und Stopp</b>	205
<b>7.7.4 Den aktuellen Status der Audio-CD ermitteln</b>	209
<b>7.7.5 Das komplette Listing</b>	211
<b>8 System- und Benutzerdateien</b>	219
<b>8.1 Die Datei /etc/passwd</b>	219
8.1.1 Die Datei /etc/passwd auswerten	221
8.1.2 getpwuid und getpwnam – einzelne Abfrage von /etc/passwd	221
8.1.3 getpwent, setpwent und endpwent – komplette Abfrage von /etc/passwd	223
<b>8.2 Die Datei /etc/shadow</b>	225
8.2.1 Die Datei /etc/shadow auswerten	227
8.2.2 getspent, setspent und endsent – komplette Abfrage von /etc/shadow	228
<b>8.3 Die Datei /etc/group</b>	231
8.3.1 Die Datei /etc/group auswerten	232
8.3.2 getgrnam und getrggid – einzelne Einträge aus /etc/group abfragen	233
8.3.3 getgrent, setgrent und endgrent – alle Einträge in /etc/group abfragen	235
<b>8.4 uname – Informationen zum lokalen System erfragen</b>	235
<b>8.5 Das Verzeichnis /etc/skel und der Network Information Service (NIS)</b>	237
<b>8.6 Dateien für Netzwerkinformationen</b>	238
<b>8.7 Folgen für Entwickler</b>	238
8.7.1 Statisches Linken	238
8.7.2 Authentifizierung modernisiert	238
<b>9 Dämonen, Zombies und Prozesse</b>	241
<b>9.1 Was ist ein Prozess?</b>	241
<b>9.2 Prozesskomponente</b>	242
9.2.1 Prozessnummer (PID)	243
9.2.2 Prozessnummer des Vaterprozesses (PPID)	243
9.2.3 Benutzer- und Gruppennummer eines Prozesses (UID, EUID, GID, EGID)	243
9.2.4 Prozessstatus	244
9.2.5 Prozesspriorität	245
9.2.6 Timesharing-Prozesse	246

9.2.7	Echtzeitprozesse .....	250
9.2.8	Prozessauslagerung .....	251
9.2.9	Steuerterminal .....	252
<b>9.3</b>	<b>Prozesse überwachen – ps, top, kpm .....</b>	252
<b>9.4</b>	<b>Lebenszyklus eines Prozesses .....</b>	255
<b>9.5</b>	<b>Umgebungsvariablen eines Prozesses .....</b>	257
9.5.1	Einzelne Umgebungsvariablen abfragen .....	259
9.5.2	Umgebungsvariable verändern oder hinzufügen – putenv() und setenv() .....	260
9.5.3	Löschen von Umgebungsvariablen – unsetenv() und clearenv() .....	263
<b>9.6</b>	<b>Ressourcenlimits eines Prozesses .....</b>	265
9.6.1	Mehr Sicherheit mit Ressourcenlimits .....	268
9.6.2	Core-Dateien .....	269
<b>9.7</b>	<b>Prozesserkennung .....</b>	269
<b>9.8</b>	<b>Erzeugung von Prozessen – fork() .....</b>	271
9.8.1	Pufferung .....	274
9.8.2	Was wird vererbt und was nicht? .....	279
9.8.3	Einen Prozess mit veränderter Priorität erzeugen .....	280
<b>9.9</b>	<b>Warten auf einen Prozess .....</b>	282
<b>9.10</b>	<b>Die exec-Familie .....</b>	289
9.10.1	execl() .....	290
9.10.2	execve() .....	291
9.10.3	execv() .....	291
9.10.4	execle() .....	292
9.10.5	execlp() .....	292
9.10.6	execvp() .....	293
9.10.7	Kindprozesse mit einem exec-Aufruf überlagern .....	293
<b>9.11</b>	<b>Kommandoaufrufe aus dem Programm – system() .....</b>	295
<b>9.12</b>	<b>Dämonen bzw. Hintergrundprozesse .....</b>	296
9.12.1	Wie ein Prozess zum Dämon wird .....	297
9.12.2	Dämon, sprich mit uns! .....	297
9.12.3	Protokollieren von Dämonen – syslog() .....	298
9.12.4	syslog() in der Praxis .....	300
9.12.5	Den Dämon, den ich rief ... .....	303
<b>9.13</b>	<b>Rund um die Ausführung von Prozessen .....</b>	307
9.13.1	Einen Dämon beim Booten mit einem init-Skript starten .....	308
9.13.2	Darf es auch ein bisschen weniger init sein? (Startskript mit systemd) .....	316
9.13.3	Hintergrundprozesse und Jobkontrolle .....	317

9.13.4	Prozesse zeitgesteuert ausführen (cron-Jobs) .....	321
<b>9.14</b>	<b>Zusammenfassung und Ausblick .....</b>	324

## **10 Signale**

---

<b>10.1</b>	<b>Grundlagen zu den Signalen .....</b>	325
10.1.1	Signalmaske .....	327
10.1.2	Signale und fork() .....	328
10.1.3	Signale und exec .....	328
10.1.4	Übersicht über die Signale .....	328
<b>10.2</b>	<b>Das neue Signalkonzept .....</b>	331
10.2.1	Wozu ein »neues« Signalkonzept? .....	332
<b>10.3</b>	<b>Signalmenge initialisieren .....</b>	332
<b>10.4</b>	<b>Signalmenge hinzufügen oder löschen .....</b>	333
<b>10.5</b>	<b>Signale einrichten oder erfragen .....</b>	333
10.5.1	Einen Signalhandler einrichten, der zurückkehrt .....	338
<b>10.6</b>	<b>Signal an den eigenen Prozess senden – raise() .....</b>	340
<b>10.7</b>	<b>Signale an andere Prozesse senden – kill() .....</b>	340
<b>10.8</b>	<b>Zeitschaltuhr einrichten – alarm() .....</b>	341
<b>10.9</b>	<b>Prozesse stoppen, bis ein Signal eintritt – pause() .....</b>	342
<b>10.10</b>	<b>Prozesse für eine bestimmte Zeit stoppen – sleep() und usleep() .....</b>	342
<b>10.11</b>	<b>Signalmaske erfragen oder ändern – sigprocmask() .....</b>	343
<b>10.12</b>	<b>Prozess während einer Änderung der Signalmaske stoppen – sigsuspend() .....</b>	344
<b>10.13</b>	<b>Prozesse synchronisieren .....</b>	344

## **11 IPC – Interprozesskommunikation**

---

<b>11.1</b>	<b>Unterschiedliche Techniken zur Interprozesskommunikation (IPC) im Überblick .....</b>	349
11.1.1	(Namenlose) Pipes .....	350
11.1.2	Benannte Pipes (FIFO-Pipes) .....	351
11.1.3	Message Queue (Nachrichtenspeicher) .....	353
11.1.4	Semaphore .....	353
11.1.5	Shared Memory (gemeinsamer Speicher) .....	354
11.1.6	STREAMS .....	354

11.1.7 Sockets .....	355
11.1.8 Lock Files (Sperrdateien) .....	356
11.1.9 Dateisperren (Advisory File Locks) .....	356
11.1.10 Dateisperren (Record Locking) .....	356
<b>11.2 Neuere Techniken .....</b>	<b>357</b>
11.2.1 D-Bus und kdbus .....	357
11.2.2 Dateisystem-Beobachtung .....	357
<b>11.3 Gründe für IPC .....</b>	<b>358</b>
<b>11.4 Pipes .....</b>	<b>359</b>
11.4.1 Eigenschaften von Pipes .....	359
11.4.2 Pipes einrichten – pipe() .....	359
11.4.3 Eigenschaften von elementaren E/A-Funktionen bei Pipes .....	363
11.4.4 Standard-E/A-Funktionen mit pipe .....	364
11.4.5 Pipes in einen anderen Prozess umleiten .....	366
11.4.6 Ein Filterprogramm mithilfe einer Pipe erstellen .....	369
11.4.7 Einrichten einer Pipe zu einem anderen Prozess – popen() .....	372
11.4.8 Mail versenden mit Pipes und Sendmail .....	374
11.4.9 Drucken über eine Pipe mit lpr .....	378
11.4.10 Benannte Pipes – FIFOs .....	383
<b>11.5 System-V-Interprozesskommunikation .....</b>	<b>400</b>
11.5.1 Gemeinsamkeiten der System V-UNIX-Mechanismen .....	400
11.5.2 Ein Objekt einrichten, eine Verbindung herstellen und das Objekt wieder löschen .....	401
11.5.3 Datenaustausch zwischen nicht verwandten Prozessen .....	402
<b>11.6 Semaphore .....</b>	<b>402</b>
11.6.1 Lebenszyklus eines Semaphors .....	403
11.6.2 Ein Semaphore öffnen oder erstellen – semget() .....	406
11.6.3 Abfragen, ändern oder löschen der Semaphormenge – semctl() .....	407
11.6.4 Operationen auf Semaphormengen – semop() .....	409
11.6.5 Semaphore im Vergleich mit Sperren .....	411
<b>11.7 Message Queues .....</b>	<b>411</b>
11.7.1 Eine Message Queue öffnen oder erzeugen – msgget() .....	412
11.7.2 Nachrichten versenden – msgsnd() .....	413
11.7.3 Eine Nachricht empfangen – msgrcv() .....	414
11.7.4 Abfragen, ändern oder löschen einer Message Queue – msgctl() .....	414
<b>11.8 Shared Memory .....</b>	<b>424</b>
11.8.1 Ein Shared-Memory-Segment erstellen oder öffnen – shmget() .....	424
11.8.2 Ein Shared-Memory-Segment abfragen, ändern oder löschen – shmctl() .....	425
11.8.3 Ein Shared-Memory-Segment anbinden (attach) – shmat() .....	426

11.8.4 Ein Shared-Memory-Segment loslösen – shmdt() .....	427
11.8.5 Client-Server-Beispiel – Shared Memory .....	427
<b>11.9 Das Dateisystem überwachen .....</b>	<b>435</b>

## **12 Threads**

---

<b>12.1 Unterschiede zwischen Threads und Prozessen .....</b>	<b>439</b>
<b>12.2 Scheduling und Zustände von Threads .....</b>	<b>440</b>
<b>12.3 Die grundlegenden Funktionen zur Thread-Programmierung .....</b>	<b>442</b>
12.3.1 pthread_create – einen neuen Thread erzeugen .....	442
12.3.2 pthread_exit – einen Thread beenden .....	443
12.3.3 pthread_join – auf das Ende eines Threads warten .....	444
12.3.4 pthread_self – die ID von Threads ermitteln .....	444
12.3.5 pthread_equal – die ID von zwei Threads vergleichen .....	450
12.3.6 pthread_detach – einen Thread unabhängig machen .....	452
<b>12.4 Die Attribute von Threads und das Scheduling .....</b>	<b>453</b>
<b>12.5 Threads synchronisieren .....</b>	<b>459</b>
12.5.1 Mutex .....	462
12.5.2 Condition-Variablen (Bedingungsvariablen) .....	471
12.5.3 Semaphore .....	483
12.5.4 Weitere Synchronisationstechniken im Überblick .....	486
<b>12.6 Threads abbrechen (canceln) .....</b>	<b>487</b>
<b>12.7 Erzeugen von Thread-spezifischen Daten (TSD-Data) .....</b>	<b>492</b>
<b>12.8 pthread_once – Codeabschnitt einmal ausführen .....</b>	<b>495</b>
<b>12.9 Thread-safe (thread-sichere Funktionen) .....</b>	<b>498</b>
<b>12.10 Threads und Signale .....</b>	<b>499</b>
<b>12.11 Zusammenfassung und Ausblick .....</b>	<b>504</b>

## **13 Populäre Programmietechniken unter Linux**

---

<b>13.1 Reguläre Ausdrücke .....</b>	<b>505</b>
13.1.1 Ermittlung der aktuellen Version der SystemRescueCd im Netz .....	506
13.1.2 Syntax .....	506
13.1.3 Beispiele .....	507

13.1.4 Programmieren von regulären Ausdrücken mit pcre .....	509
13.1.5 In Datenbanken .....	511
<b>13.2 getopt: Kommandozeilenoptionen auswerten .....</b>	<b>513</b>
<b>13.3 Capabilities: Wenn root zu mächtig wäre .....</b>	<b>517</b>
13.3.1 Prozess-Capabilities .....	518
13.3.2 Datei-Capabilities .....	522
<b>13.4 Lokalisierung mit gettext .....</b>	<b>522</b>
<b>13.5 mmap(): Dateien als Speicherbereich .....</b>	<b>526</b>
<b>13.6 Codedokumentation automatisch erzeugen lassen mit Doxygen .....</b>	<b>530</b>
<b>14 Netzwerkprogrammierung .....</b>	<b>533</b>
<b>14.1 Einführung .....</b>	<b>533</b>
<b>14.2 Aufbau von Netzwerken .....</b>	<b>534</b>
14.2.1 ISO/OSI und TCP/IP – die Referenzmodelle .....	534
14.2.2 Das World Wide Web (Internet) .....	537
<b>14.3 TCP/IP – Aufbau und Struktur .....</b>	<b>539</b>
14.3.1 Datenübertragungsschicht .....	539
14.3.2 Internetschicht .....	539
14.3.3 Transportschicht (TCP, UDP) .....	540
14.3.4 Anwendungsschicht .....	541
<b>14.4 TCP-Sockets .....</b>	<b>543</b>
<b>14.5 Das Kommunikationsmodell für Sockets .....</b>	<b>544</b>
<b>14.6 Grundlegende Funktionen zum Zugriff auf die Socket-Schnittstelle .....</b>	<b>545</b>
14.6.1 Ein Socket anlegen – socket() .....	545
14.6.2 Verbindungsaufbau – connect() .....	547
14.6.3 Socket mit einer Adresse verknüpfen – bind() .....	550
14.6.4 Auf Verbindungen warten – listen() und accept() .....	550
14.6.5 Senden und Empfangen von Daten (1) – write() und read() .....	551
14.6.6 Senden und Empfangen von Daten (2) – send() und recv() .....	552
14.6.7 Verbindung schließen – close() .....	553
<b>14.7 Aufbau eines Clientprogramms .....</b>	<b>553</b>
14.7.1 Zusammenfassung Clientanwendung und Quellcode .....	556
<b>14.8 Aufbau des Serverprogramms .....</b>	<b>558</b>
14.8.1 Zusammenfassung: Serveranwendung und Quellcode .....	559

<b>14.9 IP-Adressen konvertieren, manipulieren und extrahieren .....</b>	<b>563</b>
14.9.1 inet_aton(), inet_pton() und inet_addr() .....	564
14.9.2 inet_ntoa() und inet_ntop() .....	565
14.9.3 inet_network() .....	566
14.9.4 inet_netof() .....	566
14.9.5 inet_lnaof() .....	567
14.9.6 inet_makeaddr() .....	567
<b>14.10 Namen und IP-Adressen umwandeln .....</b>	<b>570</b>
14.10.1 Nameserver .....	571
14.10.2 Informationen zum Rechner im Netz – gethostbyname und gethostbyaddr .....	571
14.10.3 Service-Informationen – getservbyname() und getservbyport() .....	576
<b>14.11 Der Puffer .....</b>	<b>580</b>
<b>14.12 Standard-E/A-Funktionen verwenden .....</b>	<b>581</b>
14.12.1 Pufferung von Standard-E/A-Funktionen .....	582
<b>14.13 Parallele Server .....</b>	<b>582</b>
<b>14.14 Synchrones Multiplexing – select() .....</b>	<b>598</b>
<b>14.15 POSIX-Threads und Netzwerkprogrammierung .....</b>	<b>620</b>
<b>14.16 Optionen für Sockets setzen bzw. erfragen .....</b>	<b>625</b>
14.16.1 setsockopt() .....	625
14.16.2 getsockopt() .....	626
14.16.3 Socket-Optionen .....	626
<b>14.17 UDP .....</b>	<b>630</b>
14.17.1 Clientanwendung .....	632
14.17.2 Serveranwendung .....	633
14.17.3 recvfrom() und sendto() .....	633
14.17.4 bind() verwenden oder weglassen .....	638
<b>14.18 Unix-Domain-Sockets (IPC) .....</b>	<b>639</b>
14.18.1 Die Adressstruktur von Unix-Domain-Sockets .....	639
14.18.2 Lokale Sockets erzeugen – socketpair() .....	643
<b>14.19 Multicast-Socket .....</b>	<b>645</b>
14.19.1 Anwendungsgebiete von Multicast-Verbindungen .....	653
<b>14.20 Nichtblockierende I/O-Sockets .....</b>	<b>653</b>
<b>14.21 Etwas zu Streams, Raw Socket und TLI und XTI .....</b>	<b>656</b>
14.21.1 Raw Socket .....	657
14.21.2 TLI und XTI .....	657
14.21.3 RPC (Remote Procedure Call) .....	658

<b>14.22 Netzwerksoftware nach IPv6 portieren .....</b>	659
14.22.1 Konstanten .....	659
14.22.2 Strukturen .....	659
14.22.3 Funktionen .....	660
<b>14.23 Sicherheit .....</b>	660
<b>15 Abgesicherte Netzwerkverbindungen</b>	663
<b>15.1 Grundlagen .....</b>	663
15.1.1 Schlüsselgenerierung .....	664
15.1.2 Bibliotheken und Protokollversionen .....	665
15.1.3 Schlüsselgenerierung in der Praxis .....	665
<b>15.2 Server .....</b>	666
<b>15.3 Client .....</b>	671
<b>15.4 Referenz OpenSSL .....</b>	675
15.4.1 BIO: Session eröffnen und schließen .....	675
15.4.2 ERR: Alles über Fehlermeldungen .....	678
15.4.3 OpenSSL-Konfiguration .....	679
15.4.4 SSL: Operationen auf SSL-Verbindungen .....	679
15.4.5 SSL_CTX und andere: Schlüssel und ihre Optionen .....	682
<b>16 Relationale Datenbanken: MySQL, PostgreSQL und SQLite</b>	687
<b>16.1 Relationales Datenbankdesign .....</b>	688
<b>16.2 Datenhaufen und indizierte Listen .....</b>	690
<b>16.3 SQL-Systeme im Überblick .....</b>	691
<b>16.4 Vorbereitungen .....</b>	695
16.4.1 MySQL installieren und starten .....	696
16.4.2 PostgreSQL installieren und starten .....	699
16.4.3 SQLite installieren und starten .....	701
<b>16.5 Beispiele in SQL .....</b>	701
16.5.1 Tabellen anlegen .....	701
16.5.2 Schlüsselfelder .....	702
16.5.3 Indizes .....	703

16.5.4 Tabellen umbenennen und ändern .....	704
16.5.5 Daten einfügen, ändern und löschen .....	708
16.5.6 Daten importieren .....	711
16.5.7 Daten ausgeben .....	711
16.5.8 NULL ist 0 oder undefiniert? .....	713
16.5.9 Unscharfe Suche .....	714
<b>16.6 MySQL-C-Programmierschnittstelle .....</b>	714
16.6.1 Verbindung mit dem MySQL-Server aufbauen .....	716
16.6.2 Aufgetretene Fehler ermitteln – mysql_errno() und mysql_error() .....	719
16.6.3 Schließt die Verbindung zum Server – mysql_close() .....	720
16.6.4 Erstes Beispiel .....	720
16.6.5 Verschiedene Informationen ermitteln .....	725
16.6.6 Datenbanken, Tabellen und Felder ausgeben (MYSQL_RES) .....	729
16.6.7 Ergebnismenge zeilenweise bearbeiten (MYSQL_ROW) .....	731
16.6.8 Ergebnismenge spaltenweise einlesen (und ausgeben) – MYSQL_FIELD .....	733
16.6.9 Ein Beispiel .....	738
16.6.10 Ergebnismenge – weitere Funktionen .....	745
16.6.11 Befehle an den Server – mysql_query() und mysql_real_query() .....	746
16.6.12 Weitere Funktionen .....	750
16.6.13 Veraltete Funktionen .....	756
<b>16.7 Beispiel eines Newsystems mit MySQL .....</b>	757
16.7.1 Die Headerdatei my_cgi.h .....	758
16.7.2 (Pseudo-)Planung .....	764
16.7.3 Datenbank und Tabellen anlegen .....	764
16.7.4 MySQL-Clients mit GUI .....	786
16.7.5 Randnotiz .....	786
<b>16.8 Neue SQL-Funktionen für die Shell – MySQL erweitern .....</b>	787
<b>16.9 MySQL-Funktionen mit der UDF-Schnittstelle entwerfen .....</b>	788
16.9.1 UDF-Sequenzen .....	790
16.9.2 UDF_INIT-Struktur .....	790
16.9.3 UDF_ARGS-Struktur .....	791
16.9.4 Rückgabewert .....	793
16.9.5 Benutzerdefinierte Funktionen erstellen .....	793
16.9.6 Benutzerdefinierte Funktion kompilieren, installieren und ausführen .....	795
<b>16.10 PostgreSQL – Konfiguration .....</b>	798
16.10.1 Konfigurationsdateien bei PostgreSQL (postgresql.conf, pg_hba.conf) .....	798
16.10.2 Crashkurs: PostgreSQL .....	799
16.10.3 PostgreSQL-C-API – libpq .....	807
16.10.4 Umgebungsvariablen und Passworddatei .....	830
16.10.5 PostgreSQL und Threads .....	831

16.10.6 Ausblick .....	832
16.10.7 Funktionsüberblick .....	833
<b>17 GTK+</b>	<b>839</b>
<b>17.1 Was ist GTK+? .....</b>	<b>839</b>
17.1.1 Was sind GDK und Glib? .....	840
17.1.2 Schnittstellen von GTK+ zu anderen Programmiersprachen .....	841
17.1.3 GTK+ und GNOME .....	841
17.1.4 Die GTK+-Versionen 1.2, 2.x und 3.x .....	841
17.1.5 GTK+ – Aufbau dieses Kapitels .....	842
<b>17.2 GTK+-Anwendungen übersetzen .....</b>	<b>843</b>
<b>17.3 Eine Einführung in die Glib-Bibliothek .....</b>	<b>843</b>
17.3.1 Datentypen .....	844
17.3.2 Routinen .....	845
17.3.3 Assertions-Funktionen .....	847
17.3.4 Speicherverwaltung .....	849
17.3.5 Stringbearbeitung .....	852
17.3.6 Selbstverwaltender Stringpuffer .....	857
17.3.7 Timer .....	860
17.3.8 Dynamische Arrays .....	862
17.3.9 Listen, Hashtabellen und binäre Bäume .....	866
17.3.10 Ausblick zur Glib .....	867
<b>17.4 Grundlagen der GTK+-Programmierung .....</b>	<b>867</b>
17.4.1 Die Umgebung initialisieren .....	868
17.4.2 Widgets erzeugen und gegebenenfalls die Attribute setzen .....	868
17.4.3 Eine Callback-Funktion einrichten, um Events abzufangen .....	870
17.4.4 Eine GTK+-Anwendung beenden .....	873
17.4.5 Die hierarchische Anordnung der Widgets definieren .....	874
17.4.6 Widgets anzeigen .....	875
17.4.7 Signale und Events abfangen und bearbeiten – (Events-)Verarbeitungsschleife .....	876
17.4.8 GTK+ und Umlaute (Zeichenkodierung) .....	877
<b>17.5 Fenster – GtkWindow .....</b>	<b>878</b>
17.5.1 Dialogfenster (Dialogboxen) .....	881
17.5.2 GtkMessageDialog .....	886
<b>17.6 Anzeige-Elemente .....</b>	<b>886</b>
17.6.1 Text – GtkLabel .....	889

17.6.2 Trennlinie – GtkSeparator .....	892
17.6.3 Grafiken – GtkImage .....	893
17.6.4 Statusleiste – GtkStatusbar .....	893
17.6.5 Fortschrittsbalken – GtkProgressBar .....	894
<b>17.7 Behälter .....</b>	<b>894</b>
17.7.1 Boxen – GtkBox .....	894
17.7.2 Aufteilungen, Register und Button-Box .....	896
17.7.3 Tabellen – GtkTable .....	903
17.7.4 Ausrichtung – GtkAlignment .....	907
<b>17.8 Buttons und Toggled-Buttons .....</b>	<b>908</b>
17.8.1 Buttons allgemein .....	915
17.8.2 Radio-Buttons (GtkRadioButton) .....	915
17.8.3 GtkRadioButton, GtkCheckButton und GtkToggleButton .....	916
17.8.4 Signale für Buttons (GtkButton) .....	917
<b>17.9 Dateneingabe .....</b>	<b>917</b>
17.9.1 Textfelder – GtkEntry .....	924
17.9.2 Schieberegler – GtkScale .....	925
17.9.3 Zahlenfelder – GtkSpinButton .....	926
17.9.4 Einstellungen – GtkAdjustment .....	927
17.9.5 GtkEditable .....	928
<b>17.10 Menü und Toolbar .....</b>	<b>929</b>
17.10.1 Menü – GtkItemFactory .....	934
17.10.2 Toolbar – GtkToolbar .....	940
17.10.3 Optionsmenü – GtkOptionsMenu .....	943
17.10.4 Combo-Boxen – GtkCombo .....	944
<b>17.11 Mehrzeiliger Text .....</b>	<b>948</b>
17.11.1 Text(editor) – GtkTextView, GtkTextBuffer .....	956
17.11.2 Scrollendes Fenster – GtkScrolledWindow .....	960
<b>17.12 Auswählen (Selection) .....</b>	<b>962</b>
17.12.1 Dateiauswahl – GtkFileSelection .....	972
<b>17.13 Events .....</b>	<b>973</b>
<b>17.14 Weitere Widget- und GTK+-Elemente im Überblick .....</b>	<b>979</b>
17.14.1 Bäume und Listen .....	980
17.14.2 Lineale .....	980
17.14.3 Zwischenablage (Clipboard) .....	980
17.14.4 Drag & Drop .....	981
17.14.5 Stock Items – Repertoire-Einträge .....	981
17.14.6 Signale .....	981
17.14.7 Ressource-Files .....	981

17.14.8 Widget-Entwicklung .....	981
17.14.9 GDK .....	981
<b>18 Werkzeuge für Programmierer</b>	<b>983</b>
<b>18.1 Der Compiler GCC</b> .....	<b>984</b>
18.1.1 Standardgebrauch des GCC .....	986
18.1.2 Linken von Programmbibliotheken .....	986
18.1.3 Dateien, die GCC kennt .....	988
18.1.4 Ausgabedateien bei jedem einzelnen Schritt der Übersetzung erstellen .....	988
18.1.5 Noch mehr Optionen .....	989
18.1.6 Optionen für Warnmeldungen .....	989
18.1.7 Präprozessor-Optionen .....	990
18.1.8 Debuggen und Profiling .....	991
18.1.9 Optimierungsflags .....	991
18.1.10 Architektur- und Submodell-Flags .....	992
18.1.11 Statisches vs. dynamisches Linken .....	993
<b>18.2 Make</b> .....	<b>994</b>
18.2.1 Make im Zusammenspiel .....	994
18.2.2 Die Rolle des Makefiles .....	995
18.2.3 Aufbau des Makefiles .....	997
18.2.4 Make zur Installation verwenden .....	1010
18.2.5 Make-Optionen .....	1011
18.2.6 Ausblick .....	1011
<b>18.3 Das Makefile portabel machen</b> .....	<b>1011</b>
18.3.1 Die Autotools .....	1013
<b>18.4 Bibliotheken erstellen</b> .....	<b>1019</b>
18.4.1 Allgemeines zu dynamischen Bibliotheken .....	1024
18.4.2 Dynamisches Nachladen von Bibliotheken .....	1026
<b>18.5 RPM</b> .....	<b>1029</b>
18.5.1 Einführung in RPM .....	1029
18.5.2 Verzeichnisse, die RPM benötigt .....	1032
18.5.3 Ein eigenes RPM-Paket erstellen .....	1032
18.5.4 Sources .....	1032
18.5.5 Die Spec-Datei .....	1034
18.5.6 Paket erstellen .....	1037
18.5.7 Das Paket installieren .....	1040
<b>18.6 Versionskontrollsysteme</b> .....	<b>1040</b>

<b>18.7 Zeitmessung an Programmen</b> .....	<b>1042</b>
18.7.1 Einfache Zeitmessung mit TIME – Laufzeit von Prozessen .....	1042
18.7.2 Profiling mit GPROF – Laufzeit von Funktionen .....	1043
18.7.3 Analyse mit GCOV .....	1047
<b>18.8 Debuggen mit GDB und DDD</b> .....	<b>1050</b>
<b>18.9 STRACE – Systemaufrufe verfolgen</b> .....	<b>1062</b>
<b>18.10 Memory Leaks und unerlaubte Speicherzugriffe</b> .....	<b>1064</b>
18.10.1 efence .....	1065
18.10.2 valgrind .....	1068
<b>18.11 Ausblick</b> .....	<b>1072</b>
<b>19 Shell-Programmierung</b> .....	<b>1075</b>
<b>19.1 Was ist eine Shell und was kann sie?</b> .....	<b>1075</b>
<b>19.2 Was sind Shellskripte und wann ist ihr Einsatz sinnvoll?</b> .....	<b>1076</b>
<b>19.3 Wann brauche ich keine Shellskripte?</b> .....	<b>1077</b>
<b>19.4 Welche Schwierigkeiten sind typisch für Shellskripte?</b> .....	<b>1078</b>
<b>19.5 Verschiedene Shelltypen</b> .....	<b>1079</b>
19.5.1 Erweiterungen der Bourne-Shell (sh) .....	1079
19.5.2 Erweiterung der C-Shell (csh) .....	1080
19.5.3 Welche Shell sollte man kennen? .....	1081
<b>19.6 Shellskripts ausführen</b> .....	<b>1081</b>
19.6.1 Shellskript im Hintergrund ausführen .....	1082
19.6.2 Die Bedeutung der Subshell .....	1083
19.6.3 Die ausführende Shell festlegen .....	1085
19.6.4 Shellskript beenden .....	1091
<b>19.7 Vom Shellskript zum Prozess</b> .....	<b>1092</b>
<b>19.8 Einen Datenstrom (Stream) umleiten</b> .....	<b>1093</b>
19.8.1 Die Standardausgabe umleiten .....	1093
19.8.2 Die Standardfehlerausgabe umleiten .....	1094
19.8.3 Die Standardausgabe mit der Standardfehlerausgabe verknüpfen .....	1095
19.8.4 Die Standardeingabe umleiten .....	1096
19.8.5 Pipes .....	1097
19.8.6 Standardausgabe in zwei Richtungen mit tee .....	1099
19.8.7 Zusammenfassung der verschiedenen Umlenkungen .....	1100

<b>19.9 Ersatzmuster (Namen-Expansion) zur Suche verwenden .....</b>	1101
19.9.1 Beliebige Zeichenfolge * (Asterisk) .....	1101
19.9.2 Beliebiges Zeichen ? .....	1102
19.9.3 Zeichenbereiche einschränken .....	1102
19.9.4 Brace Extensions (nur Bash und Korn-Shell) .....	1104
19.9.5 Tilde-Expansion (nur Bash und Korn-Shell) .....	1104
19.9.6 Zusammenfassung zu den Ersatzmustern .....	1104
<b>19.10 Variablen .....</b>	1105
19.10.1 Zahlen .....	1109
19.10.2 Zeichenketten .....	1115
19.10.3 Arrays (nur Bash und Korn-Shell) .....	1121
19.10.4 Variablen exportieren .....	1123
19.10.5 Die Umgebungsvariablen (Shellvariablen) .....	1127
19.10.6 Auto-Variablen der Shell .....	1128
<b>19.11 Quotings .....</b>	1130
19.11.1 Single und Double Quotes .....	1130
19.11.2 Kommandosubstitution (Back Quotes) .....	1132
<b>19.12 Kommandozeilenargumente .....</b>	1133
19.12.1 Kommandozeilenparameter \$1 bis \$9 .....	1133
19.12.2 Variable Anzahl von Parametern auswerten .....	1135
19.12.3 Die Anzahl der Argumente überprüfen .....	1135
19.12.4 Der Befehl shift .....	1136
19.12.5 Argumente für die Kommandozeile setzen – set .....	1137
19.12.6 Kommandozeilenoptionen auswerten .....	1139
<b>19.13 Kontrollstrukturen .....</b>	1140
19.13.1 Die if-Anweisung .....	1141
19.13.2 Die else-Alternative für die if-Verzweigung .....	1145
19.13.3 Mehrfache Alternative mit elif .....	1145
19.13.4 Das Kommando test .....	1146
19.13.5 Dateistatus ermitteln .....	1151
19.13.6 Ausdrücke logisch verknüpfen .....	1154
19.13.7 Verzweigungen mit case .....	1157
19.13.8 Schleifen .....	1159
<b>19.14 Terminal-Eingabe und -Ausgabe .....</b>	1166
19.14.1 Ausgabe .....	1166
19.14.2 Eingabe .....	1170
19.14.3 Umlenkungen mit exec und File-Deskriptoren erzeugen .....	1184
19.14.4 Named Pipes .....	1188
19.14.5 Menü mit select (nur Bash und Korn-Shell) .....	1189

<b>19.15 Funktionen .....</b>	1192
19.15.1 Funktionsaufruf .....	1192
19.15.2 Externe Funktionen verwenden .....	1193
19.15.3 Parameterübergabe .....	1193
19.15.4 Rückgabewert aus einer Funktion .....	1194
<b>19.16 Signale .....</b>	1197
19.16.1 Signale senden .....	1197
19.16.2 Signale in einem Shellskript abfangen – trap .....	1198
19.16.3 Signal-Handler einrichten .....	1199
19.16.4 Signale ignorieren oder zurücksetzen .....	1200
<b>19.17 Prozess- und Skriptausführung .....</b>	1201
19.17.1 Auf Prozesse warten .....	1201
19.17.2 Hintergrundprozess hervorholen .....	1201
19.17.3 Jobverwaltung .....	1202
19.17.4 Explizite Subshell verwenden .....	1204
19.17.5 Kommunikation zwischen Shellskripten .....	1205
19.17.6 Skripte zeitgesteuert ausführen .....	1209
<b>19.18 Ein Shellskript bei der Ausführung .....</b>	1209
<b>Anhang</b>	1211
<b>A Sicherheit unter Linux .....</b>	1213
<b>B Funktionsreferenz .....</b>	1239
<b>C Linux-Unix-Kommandoreferenz .....</b>	1321
Index .....	1405

# Index

_Exit .....	1258	/proc (Forts.)
_Exit() .....	1258	/proc/filesystems .....
_IOFBF .....	116	/proc/ide/ .....
_IOLBF .....	116	/proc/interrupts .....
_IONBF .....	116	/proc/ioports .....
_POSIX_CHOWN_RESTRICTED .....	1272	/proc/loadavg .....
_POSIX_JOB_CONTROL .....	1272	/proc/locks .....
_POSIX_NO_TRUNC .....	1272	/proc/meminfo .....
_POSIX_PRIORITY_SCHEDULING .....	249	/proc/modules .....
_POSIX_SAVED_IDS .....	1272	/proc/mounts .....
_POSIX_SOURCE .....	1273	/proc/mtab .....
_POSIX_THREAD_PRIORITY_SCHEDULING .....	455	/proc/net/ .....
_POSIX_VERSION .....	1272	/proc/scsi .....
_XOPEN_VERSION .....	1273	/proc/scsi/scsi .....
? (Shell) .....	1102	/proc/self .....
[ (Kommando) .....	1147	/proc/stat .....
* (Shell) .....	1101	/proc/sys/dev/cdrom/autoeject .....
/dev-Verzeichnis .....	193	/proc/sys/kernel/ctrl-alt-del .....
/etc/group .....	231	/proc/sys/kernel/osrelease .....
<i>Struktur</i> .....	232	/proc/sys/kernel/ostype .....
/etc/gshadow .....	232	/proc/sys/kernel/version .....
/etc/hosts .....	571	/proc/uptime .....
/etc/inittab .....	310	/proc/version .....
/etc/ld.so.conf .....	1026	Filesystem .....
/etc/passwd .....	219	Hardware-/Systeminformationen .....
/etc/resolv.conf .....	571	Informationen abfragen .....
/etc/services .....	576	Kernelinformationen .....
/etc/shadow .....	220, 225	Prozessinformationen .....
<i>Struktur</i> .....	227	/proc-Verzeichnis .....
/etc/shells .....	220	/usr/bin .....
/etc/skel .....	237	/usr/lib/systemd/system/ .....
/etc/syslog.conf .....	301	/usr/local/bin .....
/proc .....		& (Hintergrundprozess) .....
/proc/\$pid/cmdline .....	180	<> (Shell) .....
/proc/\$pid/cwd .....	183	<arpa/inet.h> .....
/proc/\$pid/environ .....	180	<assert.h> .....
/proc/\$pid/exe .....	183	<complex.h> .....
/proc/\$pid/fd/ .....	182	<ctype.h> .....
/proc/\$pid/maps .....	184	<dirent.h> .....
/proc/\$pid/root .....	183	<dlfcn.h> .....
/proc/\$pid/stat .....	183	<errno.h> .....
/proc/\$pid/statm .....	183	<fenv.h> .....
/proc/\$pid/status .....	183	<float.h> .....
/proc/apm .....	176	<glib.h> .....
/proc/cpuinfo .....	174	<inttypes.h> .....
/proc/devices .....	175	<iso646.h> .....
/proc/dma .....	175	<libpq-fe.h> .....
		1315

<limits.h> ..... 61, 1244  
 <limits.h> (POSIX) ..... 1271 AF\_INET6 ..... 545  
 <locale.h> ..... 1245 AF\_INIX ..... 545  
 <math.h> ..... 1248 AF\_IPX ..... 545  
 <math.h> C99 ..... 1268 AF\_LOCAL ..... 545  
 <netdb.h> ..... 1307, 1308 AF\_NETLINK ..... 545  
 <netinet/in.h> ..... 564, 1305 AF\_PACKET ..... 545  
 <pthread.h> ..... 1297 AF\_UNIX ..... 545  
 <regex.h> ..... 133 AF\_X25 ..... 545  
 <setjmp.h> ..... 1249 afio ..... 1367  
 <signal.h> ..... 1250 AIX ..... 26, 31  
 <stdarg.h> ..... 110, 1251 alarm() ..... 341, 1293  
 <stdbool.h> ..... 1267 alias ..... 1402  
 <stddef.h> ..... 1252 alphasort() ..... 130  
 <stdint.h> ..... 1268 ANSI C ..... 42, 1239  
 <stdio.h> ..... 106, 1253 ANSI C99 ..... 1262  
 <stdlib.h> ..... 1257 AppArmor ..... 517  
 <string.h> ..... 1259 apropos ..... 1400  
 <sys/mman.h> ..... 1276 ARG\_MAX ..... 1272  
 <sys/socket.h> ..... 1302 arp ..... 1385  
 <sys/stat.h> ..... 145, 1283 arpa ..... 1306  
 <tgmath.h> ..... 1270 Array  
     *Shell* ..... 1121  
 <time.h> ..... 159, 1042, 1261 AS ..... 985  
 <unistd.h> ..... 1272 asctime() ..... 1261  
 <wchar.h> ..... 1263 ash ..... 1080  
 <wctype.h> ..... 1264 A-Shell ..... 1080  
 ~ (Shell) ..... 1104 asin() ..... 1248  
 \$! ..... 1129 asnprintf() ..... 110  
 \$? ..... 1129 asprintf() ..... 110  
 \${@} ..... 1129 Assembler ..... 985  
 \${\*} ..... 1129 assert ..... 1240  
 \${#} ..... 1129, 1135 at ..... 1346  
 \${\$} ..... 1129 AT&T ..... 30  
 \${0} ..... 1129 atan() ..... 1248  
 \${1} ..... 1129 atan2() ..... 1248  
 \${1, \$2, ..., \$9} ..... 1133 atexit() ..... 868, 1257  
 \${0\_EXCL} ..... 118 atof() ..... 1257  
 atoi() ..... 1257  
 atol() ..... 1257  
 atoll() ..... 1258  
 Atomare Operation ..... 338  
 Atomare Variable ..... 338  
 Ausführrechte ..... 1221  
 Ausgaben verknüpfen, Shell ..... 1095  
 AUTOCONF ..... 1072  
 AUTOMake ..... 1072  
 awk ..... 1120

**A**

Abdeckungsanalyse ..... 1047  
 abort() ..... 1257  
 abs() ..... 1258  
 accept() ..... 551, 1303  
 access() ..... 1286  
 acos() ..... 1248  
 Advisory Locking ..... 87  
 AF\_APPLETALK ..... 545  
 AF\_ATMPCV ..... 545  
 AF\_AX25 ..... 545  
 AF\_INET ..... 545

**B**

Back Quotes ..... 1132  
 badblocks ..... 1355  
 basename ..... 1341  
 Bash ..... 1080  
 bash ..... 1080  
 batch ..... 1347  
 bc ..... 1403  
     *wissenschaftliches Rechnen* ..... 1112  
 Benutzerdatenbank ..... 219  
 bg ..... 319, 1348  
     *Shell* ..... 1203  
 Bibliotheken ..... 1019  
     *dynamisch nachladen* ..... 1026  
     *dynamische* ..... 1024  
     *erstellen* ..... 1019  
     *Shared Librarys* ..... 1024  
 Big Endian ..... 555  
 BigData ..... 695  
 Binäres Lesen und Schreiben ..... 111  
 bind() ..... 550, 1303  
     *verwenden oder weglassen* ..... 638  
 binutils ..... 35  
 Blockgröße ..... 72, 156  
 BoehmGC ..... 48, 55  
 Bourne-Again-Shell ..... 1080  
 Bourne-Shell ..... 1079  
 Brace Extension, Shell ..... 1104  
 BSD ..... 26, 30  
     *FreeBSD* ..... 26, 30  
     *NetBSD* ..... 30  
     *OpenBSD* ..... 30  
 bsearch() ..... 1258  
 btowc() ..... 1241  
 Buffer Overflow ..... 113, 1064, 1222  
 bunzip2 ..... 1366  
 bzipcat ..... 1321  
 bzip2 ..... 1366

**C**

C99 ..... 1262  
 cached Inodes ..... 72  
 cal ..... 1380  
 calloc() ..... 1257  
 Capabilities ..... 517  
 case, Shell ..... 1157  
 cat ..... 1322  
 cd ..... 1341  
 ceil() ..... 1249  
 Certification Authority ..... 664  
 cfdisk ..... 1356  
 CHAR\_BIT ..... 1244  
 CHAR\_MAX ..... 1244  
 CHAR\_MIN ..... 1244  
 chdir() ..... 120, 1281  
 chgrp ..... 1322  
 CHILD\_MAX ..... 1272  
 chmod ..... 1324  
 chmod() ..... 105, 147, 1286  
 chown ..... 1325  
 chown() ..... 1286  
 chroot() ..... 1226  
 cksum ..... 1322  
 clamav ..... 1215  
 CLang ..... 35  
 clear ..... 1397  
 clearenv() ..... 264, 1287  
 clearerr() ..... 113, 1256  
 clock() ..... 1261  
 close() ..... 68, 553, 1273  
 closedir() ..... 126, 128, 1282  
 closelog() ..... 300  
 close-on-exec-Flag ..... 77, 81, 368  
 cmp ..... 1325  
 Codepoints ..... 52  
 comm ..... 1325  
 Compiler  
     *GCC* ..... 37, 984  
 complex ..... 1264  
 compress ..... 1367  
 Condition-Variablen ..... 471  
 connect() ..... 547, 1303  
 Content Syndication ..... 1214  
 Context Switch ..... 486  
 Copy-on-write-Verfahren ..... 272  
 copysign() ..... 1269  
 Core Dump ..... 1234  
 cos() ..... 1248  
 cosh() ..... 1249  
 cp ..... 1326  
 cpio ..... 1367  
 creat() ..... 67  
 cron ..... 1348  
 cron-Job ..... 321  
 csh ..... 1079  
 C-Shell  
     *Erweiterung* ..... 1080  
 csplit ..... 1327  
 ctime() ..... 1261  
 ctype ..... 1240

cut .....	1327
<i>Shell</i> .....	1115
<b>D</b>	
<i>Dämon</i>	
<i>cron</i> .....	296
<i>div. Netzwerkdämonen</i> .....	297
<i>Drucker (lpd, cupsd)</i> .....	297
<i>erzeugen</i> .....	303
<i>klogd</i> .....	297
<i>syslogd</i> .....	297
<i>xinetd</i> .....	296
<i>Dämonprozess</i> .....	296
<i>dash</i> .....	1080
<i>date</i> .....	1381
<i>Datei</i>	
<i>beschneiden</i> .....	103
<i>binäres Lesen</i> .....	111
<i>binäres Schreiben</i> .....	111
<i>Dateiart ermitteln</i> .....	146
<i>Eigentümer</i> .....	150
<i>Größe ermitteln</i> .....	156
<i>Gruppeneigentümer</i> .....	150
<i>lesen</i> .....	72
<i>Link</i> .....	151
<i>löschen</i> .....	117
<i>mehrere Puffer lesen</i> .....	100
<i>mehrere Puffer schreiben</i> .....	100
<i>neu anlegen</i> .....	62, 67
<i>öffnen</i> .....	61, 107
<i>positionieren</i> .....	74
<i>schließen</i> .....	68, 107
<i>schreiben</i> .....	68
<i>sperren</i> .....	85
<i>Status ermitteln (Shell)</i> .....	1151
<i>temporäre erstellen</i> .....	117, 1225
<i>umbenennen</i> .....	117
<i>Zeitstempel (Inode-Änderung)</i> .....	158
<i>Zeitstempel (Lesezugriff)</i> .....	158
<i>Zeitstempel (Schreibzugriff)</i> .....	158
<i>Zugriffsrechte</i> .....	63, 1219
<i>Zugriffsrechte erfragen</i> .....	146
<i>Zugriffsrechte verändern</i> .....	105
<i>Dateisperren</i> .....	356
<i>Dateitabelleneintrag</i> .....	60
<i>Dateizeiger positionieren</i> .....	113
<i>Datenbankprogrammierung</i> .....	687, 694
<i>Datenbanksystem</i>	
<i>Attribut und Attributname</i> .....	688
<i>Beziehungstypen</i> .....	689
<i>Datenbanksystem (Forts.)</i>	
<i>Domäne</i> .....	688
<i>Entität</i> .....	688
<i>Grad der Relation</i> .....	688
<i>Kardinalität</i> .....	688
<i>Normalisierungstheorie</i> .....	689
<i>Relation</i> .....	688
<i>relational</i> .....	688
<i>SQL</i> .....	691
<i>SQL-Server</i> .....	691
<i>Tupel</i> .....	688
<i>DB2</i> .....	693
<i>DBL_DIG</i> .....	1243
<i>DBL_EPSILON</i> .....	1244
<i>DBL_MANT_DIG</i> .....	1243
<i>DBL_MAX</i> .....	1244
<i>DBL_MAX_10_EXP</i> .....	1244
<i>DBL_MAX_EXP</i> .....	1244
<i>DBL_MIN</i> .....	1244
<i>DBL_MIN_10_EXP</i> .....	1243
<i>DBL_MIN_EXP</i> .....	1243
<i>dd</i> .....	1356
<i>dd_rescue</i> .....	1359
<i>DDD</i> .....	1051
<i>DDL-Element</i> .....	708
<i>Deadlocks</i> .....	358
<i>Debuggen</i>	
<i>DDD</i> .....	1051
<i>GDB</i> .....	1051
<i>Define-Makro</i> .....	40
<i>dev</i> .....	193
<i>Devices</i> .....	193
<i>Dezimalzahlen vergleichen, Shell</i> .....	1109, 1147
<i>df</i> .....	1353
<i>DIFF</i> .....	1073
<i>diff</i> .....	1328
<i>diff3</i> .....	1328
<i>difftime()</i> .....	1261
<i>dig</i> .....	1389
<i>DIR</i> .....	125
<i>dircmp</i> .....	1341
<i>dirent</i>	
<i>-Struktur</i> .....	1282
<i>dirfd()</i> .....	125
<i>dirname</i> .....	1341
<i>div()</i> .....	1258
<i>dlclose()</i> .....	1026
<i>dlerror()</i> .....	1026
<i>dlopen()</i> .....	1026
<i>dlsym()</i> .....	1026
<i>dmesg</i> .....	1382

<i>dos2unix</i> .....	1329
<i>DoS-Attacken</i> .....	268
<i>Double Quotes</i> .....	1131
<i>DOXYGEN</i> .....	1073
<i>Doxxygen</i> .....	530
<i>DoxyWizard</i> .....	531
<i>Druckerbefehle</i> .....	1383
<i>DT_BLK</i> .....	123
<i>DT_CHR</i> .....	123
<i>DT_DIR</i> .....	123
<i>DT_FIFO</i> .....	123
<i>DT_REG</i> .....	123
<i>DT SOCK</i> .....	123
<i>DT UNKNOWN</i> .....	123
<i>DTTOIF()</i> .....	124
<i>du</i> .....	1354
<i>dump</i> .....	1370
<i>dump2fs</i> .....	1359
<i>dup()</i> .....	77, 368, 1273
<i>dup2()</i> .....	77, 368, 1273
<i>Dynamische Daten</i> .....	47
<b>E</b>	
<i>E2BIG</i> .....	1243
<i>e2fsck</i> .....	1359
<i>EACCES</i> .....	1242
<i>EBADF</i> .....	1242
<i>echo, Shell</i> .....	1166
<i>ECONTR</i> .....	1242
<i>ECURDIR</i> .....	1242
<i>EDOM</i> .....	1242
<i>EEXIST</i> .....	1243
<i>EFAULT</i> .....	1243
<i>efence</i> .....	1065
<i>EGID</i> .....	243
<i>Eingabe, Shell</i> .....	1170
<i>Einschränkungsmaske</i>	
<i>setzen und abfragen</i> .....	66
<i>EINVACC</i> .....	1242
<i>EINVAL</i> .....	1243
<i>EINVDAT</i> .....	1242
<i>EINVDRV</i> .....	1242
<i>EINVENV</i> .....	1242
<i>EINVFMT</i> .....	1242
<i>EINVFNC</i> .....	1242
<i>EINVMEM</i> .....	1242
<i>Elementare E/A-Funktionen</i> .....	57
<i>elif, Shell</i> .....	1145
<i>else, Shell</i> .....	1145
<i>EMFILE</i> .....	1242
<i>endgent()</i> .....	235
<i>endpwent()</i> .....	223
<i>endspent()</i> .....	228
<i>ENMFILE</i> .....	1242
<i>ENODEV</i> .....	1243
<i>ENOENT</i> .....	140, 1242
<i>ENOEXEC</i> .....	1243
<i>ENOFILE</i> .....	1242
<i>ENOMEM</i> .....	1243
<i>ENOPATH</i> .....	1242
<i>ENOTSAM</i> .....	1242
<i>env</i> .....	1403
<i>ERANGE</i> .....	1242
<i>errno</i> .....	62, 140, 1241
<i>Ersatzmuster, Shell</i> .....	1101
<i>ESUCCESS</i> .....	140
<i>etc</i> .....	219, 220, 225, 227, 231, 232, 237
<i>EUID</i> .....	243
<i>EWOULDBLOCK</i> .....	654
<i>EXDEV</i> .....	1243
<i>exec, Shell</i> .....	1184
<i>execl()</i> .....	289, 1289
<i>execle()</i> .....	289, 1289
<i>execlp()</i> .....	289, 1289
<i>execv()</i> .....	289, 1289
<i>execve()</i> .....	289
<i>execvp()</i> .....	289, 1289
<i>exit</i> .....	1342
<i>Shellkommando</i> .....	1091
<i>exit()</i> .....	1257
<i>exp()</i> .....	1249
<i>expand</i> .....	1329
<i>export</i> .....	1123
<i>EZERO</i> .....	1242
<b>F</b>	
<i>F_DUPFD</i> .....	80
<i>F_GETFD</i> .....	80
<i>F_GETFL</i> .....	81
<i>F_GETLK</i> .....	89
<i>F_GETOWN</i> .....	82
<i>F_RDLCK</i> .....	90
<i>F_SETFD</i> .....	81
<i>F_SETFL</i> .....	82
<i>F_SETLK</i> .....	90
<i>F_SETLKW</i> .....	91
<i>F_SETOWN</i> .....	83
<i>F_UNLCK</i> .....	90
<i>F_WRLCK</i> .....	90
<i>fabs()</i> .....	1249

fchdir()	120, 1281	FLT_DIG	1243
chmod()	105, 1286	FLT_EPSILON	1244
fchown()	1286	FLT_MANT_DIG	1243
fclose()	107, 1253	FLT_MAX	1244
fcntl()	79, 87, 88, 364, 1274	FLT_MAX_10_EXP	1244
FD_CLR()	96, 600, 1275	FLT_MAX_EXP	1244
<i>Sicherheit</i>	1237	FLT_MIN	1244
FD_ISSET()	96, 600, 1275	FLT_MIN_10_EXP	1243
FD_SET()	96, 600, 1275	FLT_MIN_EXP	1243
<i>Sicherheit</i>	1237	FLT_RADIX	1243
FD_ZERO()	96, 600, 1275	Flushing, fflush()	116
fdisk	1360	Flusskontrolle	358
fdopen()	105, 107, 1274	fma()	1269
feclearexcept()	1265	fmax()	1269
fegetenv()	1266	fmin()	1269
fegetexceptflag()	1265	fmod()	1249
fegetround()	1266	fold	1331
Fehlerbehandlung	140	fopen()	107, 1253
feholdexcept()	1266	fork()	271, 583, 1288
fenv	1265	Formatierte Ausgabe	109
feof()	111, 113, 1256	Formatierte Eingabe	110
feraiseexcept()	1265	for-Schleife, Shell	1159
ferror()	111, 113, 1256	fprintf()	109, 1254
fesetenv()	1266	fputc()	112, 1255
fesetexceptflag()	1265	fputs()	112, 1255
fesetround()	1266	fread()	111, 1256
fetestexcept()	1265	free	1355
feupdateenv()	1266	Free Software Foundation	31
fflush()	115, 275, 1253	free()	1257
fg	319, 1348	freopen()	107, 1253
<i>Shell</i>	1201	frexp()	1249
fgetc()	112, 1255	fscanf()	111, 1254
fgetpos()	114, 1256	fsck	1362
fgets()	112, 1255	fseek()	74, 114, 1256
FIFO-Pipes	351, 383	fsetpos()	114, 1256
<i>Kommunikation nicht verwandter Prozesse</i>	391	fstat()	104, 146, 1284
file	1329	fsync()	102
File-Deskriptor		ftell()	114, 1256
<i>Shell</i>	1184	ftok()	400
Filedeskriptor	58	FTP	541
<i>/proc/\$pid/fd/</i>	182	ftruncate()	103
duplicieren	77, 80	fts()	135
Eigenschaften abfragen	79	FTW_D	136
Eigenschaften ändern	79	FTW_DNR	136
fileno()	105, 1274	FTW_F	136
Filesystem	191	FTW_NS	136
FILE-Zeiger	106	FTW_SL	136
find	1330	ftw()	135, 139, 1283
finger	1342	Funktion, Shell	1192
float	1243	Funktionsreferenz	1239
floor()	1249	ANSI C	1239

## Funktionsreferenz (Forts.)

C99	1262
elementare E/A-Funktionen	1271
Verzeichnisfunktionen	1281
fwrite()	111, 1256

**G**

g_array_append_vals()	862
g_array_free()	862
g_array_index()	862
g_array_insert_vals()	862
g_array_new()	862
g_arrayprepend_vals()	862
g_array_remove_index()	862
g_assert()	847
g_error()	846
g_filename_from_utf8()	972
g_free()	849
g_list_append()	944
g_malloc()	849
g_malloc0()	849
g_memdup()	850
g_message()	846
g_new()	851
g_new0()	851
g_object_get()	916, 944
g_object_new()	869, 880
g_object_set()	916, 944
g_print()	845
g_printerr()	845
g_realloc()	849
g_renew()	851
g_return_if_fail()	848
g_return_val_if_fail()	848
g_set_print_handler()	846
g_set_printerr_handler()	846
g_signal_connect_swapped()	939
g_signal_connect()	872
g_snprintf()	854
G_STR_DELIMITERS	855
g_strcasecmp()	852
g_strchomp()	855
g_strchug()	855
g_strconcat()	854
g_strdelimit()	855
g_strdown()	852
g_strdup_printf()	854
g_strdup_vprintf()	854
g_strdup()	854
g_strerror()	846
GCOV	1047
GDB	991, 1051
<i>ausführen</i>	1052
<i>Ausführung fortsetzen</i>	1056
<i>Datenausgabe</i>	1058
<i>Einzelschritte</i>	1056
<i>Haltepunkte setzen</i>	1054
<i>Haltepunkte verwalten</i>	1056
<i>Inhalt einer Datei anzeigen</i>	1052
<i>Programm übersetzen</i>	1051
<i>Variablen prüfen</i>	1060
<i>Variablen verändern</i>	1060
gdb	35

GDK ..... 840  
*Eventmaske* ..... 976  
*Events* ..... 973  
*Xlib* ..... 840  
GdkPixbuf ..... 893  
Gerätedatei ..... 193  
*/dev-Verzeichnis* ..... 193  
*Block Device* ..... 193  
*Character Device* ..... 193  
*Minor-Nummer* ..... 195  
*Namen* ..... 197  
*spezielle* ..... 200  
GETALL ..... 409  
getc() ..... 112, 1255  
getchar() ..... 112, 1255  
getcwd() ..... 122, 1281  
getdtablesize() ..... 1231  
getegid() ..... 269, 1288  
getenv() ..... 259, 1258, 1287  
geteuid() ..... 269, 1288  
getgid() ..... 269, 1288  
getgrent() ..... 235  
gethostbyaddr() ..... 571, 1308  
gethostbyname() ..... 571, 1308  
gethostbyname2() ..... 1308  
GETNCNT ..... 409  
getopts, Shell ..... 1139  
getpeername() ..... 1305  
GETPID ..... 409  
getpid() ..... 269, 1287  
getppid() ..... 269, 1287  
getpriority() ..... 247  
getpwent() ..... 223  
getpwnam() ..... 221  
getpwuid() ..... 221  
getrlimit() ..... 265, 1287  
getrusage() ..... 268, 1287  
gets() ..... 112, 1255  
getservbyname() ..... 558, 576, 1308  
getservbyport() ..... 576, 1309  
getsockname() ..... 1305  
getsockopt() ..... 626, 1304  
getspent() ..... 228  
getspnam() ..... 230  
gettext ..... 523  
getuid() ..... 269, 1288  
GETVAL ..... 409  
GETZCNT ..... 409  
GID ..... 243  
GIMP ..... 839  
Gleitpunktzahlen  
*Kategorie (C99)* ..... 1270  
*Makros (C99)* ..... 1269  
*-Umgebung* ..... 1265  
*vergleichen* ..... 1269  
Glib ..... 840  
*UTF-8* ..... 971  
Glib-Bibliothek ..... 843  
*Array (dynamisch)* ..... 862  
*Assertions-Funktionen* ..... 847  
*Ausgabe* ..... 845  
*binäre Bäume* ..... 866  
*Datentypen* ..... 844  
*Hashtabellen* ..... 866  
*Listen* ..... 866  
*Makros zur Speicherverwaltung* ..... 851  
*Quarks* ..... 852  
*Speicherblöcke kopieren* ..... 850  
*Speicherklumpen* ..... 851  
*Speicherverwaltung* ..... 849  
*Stringbearbeitung* ..... 852  
*Stringpuffer (dynamisch)* ..... 857  
*Strings kopieren* ..... 854  
*Timer* ..... 860  
glIBC ..... 517, 524  
gmake ..... 35  
gmtime() ..... 1261  
GNOME-Desktop ..... 841  
GNU Binutils ..... 35  
GNU Build System ..... 34  
GNU Compiler Collection ..... 34  
GNU Debugger ..... 35  
GNU Make ..... 35  
GNU-Assembler ..... 985  
GNU-Projekt ..... 31  
GNU-Toolchain ..... 34  
GPROF ..... 991, 1043  
*Aufrufgraph-Profil* ..... 1046  
*flaches Profil* ..... 1044  
*Profil anlegen* ..... 1044  
*strukturiertes Profil* ..... 1046  
groupadd ..... 1343  
groupdel ..... 1343  
Group-ID-Bit ..... 150  
groupmod ..... 1343  
groups ..... 1343  
grp.h ..... 232  
Gruppendatenbank ..... 231  
gtk\_adjustment\_clamp\_page() ..... 928  
gtk\_adjustment\_get\_value() ..... 927  
gtk\_adjustment\_new() ..... 927

gtk\_adjustment\_set\_value() ..... 927  
gtk\_box\_pack\_defaults() ..... 875  
gtk\_box\_pack\_start\_defaults() ..... 895  
gtk\_combo\_set\_popdown\_strings() ..... 944  
gtk\_container\_add() ..... 874  
gtk\_dialog\_new\_with\_buttons() ..... 884  
gtk\_dialog\_run() ..... 885  
gtk\_exit() ..... 876  
gtk\_file\_selection\_get\_selections() ..... 972  
GTK\_HSCALE ..... 917  
gtk\_init() ..... 868  
gtk\_item\_factory\_create\_items() ..... 935, 938  
gtk\_item\_factory\_create\_new() ..... 935  
gtk\_item\_factory\_get\_widget() ..... 938  
gtk\_item\_factory\_new() ..... 938  
gtk\_item\_factory\_path\_from\_widget() ..... 938  
gtk\_main\_quit() ..... 876  
gtk\_main() ..... 876  
gtk\_menu\_append() ..... 943  
gtk\_menu\_item\_new\_with\_label () ..... 943  
gtk\_menu\_shell\_append() ..... 943  
gtk\_message\_dialog\_new() ..... 942  
gtk\_notebook\_append\_page\_menu() ..... 901  
gtk\_notebook\_insert\_page\_menu() ..... 901  
gtk\_notebook\_prepend\_page\_menu() ..... 901  
gtk\_notebook\_remove\_page() ..... 901  
gtk\_option\_menu\_get\_history() ..... 944  
gtk\_option\_menu\_set\_history() ..... 944  
gtk\_pack\_box\_end() ..... 895  
gtk\_pack\_box\_start() ..... 895  
gtk\_paned\_add1() ..... 900  
gtk\_paned\_add2() ..... 900  
gtk\_paned\_pack1() ..... 900  
gtk\_paned\_pack2() ..... 900  
gtk\_statusbar\_get\_context() ..... 893  
gtk\_statusbar\_pop() ..... 893  
gtk\_statusbar\_push() ..... 893  
gtk\_statusbar\_remove() ..... 893  
gtk\_statusbar\_set\_has\_resize\_grip() ..... 893  
gtk\_table\_attach\_defaults() ..... 905  
gtk\_table\_attach() ..... 905  
gtk\_text\_buffer\_apply\_tag\_by\_name() ..... 958  
gtk\_text\_buffer\_copy\_clipboard() ..... 958  
gtk\_text\_buffer\_create\_tag() ..... 956  
gtk\_text\_buffer\_cut\_clipboard() ..... 958  
gtk\_text\_buffer\_delete\_selection() ..... 958  
gtk\_text\_buffer\_get\_bounds() ..... 973  
gtk\_text\_buffer\_get\_end\_iter() ..... 973  
gtk\_text\_buffer\_get\_selection\_bounds() ..... 958  
gtk\_text\_buffer\_get\_text() ..... 973  
gtk\_text\_buffer\_insert() ..... 973  
gtk\_text\_buffer\_paste\_clipboard() ..... 958  
gtk\_text\_view\_get\_buffer() ..... 956  
gtk\_toolbar\_append\_space() ..... 942  
gtk\_toolbar\_insert\_stock() ..... 941  
GTK\_TYPE\_ACCEL\_GROUP ..... 939  
GTK\_TYPE\_ADJUSTMENT ..... 917, 927  
*Signale* ..... 928  
GTK\_TYPE\_BUTTON ..... 908  
*Eigenschaften* ..... 915  
GTK\_TYPE\_CHECK\_BUTTON ..... 908  
*Eigenschaften* ..... 915, 916  
GTK\_TYPE\_CLIPBOARD ..... 949  
GTK\_TYPE\_COMBO ..... 945  
*Eigenschaften* ..... 945  
GTK\_TYPE\_ENTRY ..... 917, 924  
*Eigenschaften* ..... 924  
GTK\_TYPE\_FILE\_SELECTION ..... 972  
*Eigenschaften* ..... 972  
GTK\_TYPE\_HBOX ..... 895  
*Eigenschaften* ..... 895  
GTK\_TYPE\_HBUTTON\_BOX ..... 902  
GTK\_TYPE\_HPANED ..... 896, 899  
*Eigenschaften* ..... 900  
GTK\_TYPE\_HSCALE ..... 925  
*Eigenschaften* ..... 925  
*Signale* ..... 926  
GTK\_TYPE\_HSEPARATOR ..... 892  
GTK\_TYPE\_IMAGE ..... 893  
GTK\_TYPE\_LABEL ..... 889  
*Eigenschaften* ..... 889  
GTK\_TYPE\_MENU ..... 943  
GTK\_TYPE\_NOTEBOOK ..... 896, 901  
*Eigenschaften* ..... 902  
GTK\_TYPE\_OPTION\_MENU ..... 943  
GTK\_TYPE\_PROGRESS\_BAR ..... 894  
GTK\_TYPE\_RADIO\_BUTTON ..... 908, 915  
*Eigenschaften* ..... 915, 916  
GTK\_TYPE\_RANGE ..... 925  
GTK\_TYPE\_SCROLLED\_WINDOW ..... 960  
*Eigenschaften* ..... 960  
GTK\_TYPE\_SPIN\_BUTTON ..... 917, 926  
*Eigenschaften* ..... 926  
GTK\_TYPE\_TABLE ..... 903, 905  
*Eigenschaften* ..... 907  
GTK\_TYPE\_TEXT\_BUFFER ..... 948  
GTK\_TYPE\_TEXT\_ITER ..... 949  
GTK\_TYPE\_TEXT\_MARK ..... 949  
GTK\_TYPE\_TEXT\_TAG ..... 949  
GTK\_TYPE\_TEXT\_TAG\_TABLE ..... 949  
GTK\_TYPE\_TEXT\_VIEW ..... 948, 956

GTK\_TYPE\_TOGGLED\_BUTTON ..... 908  
*Eigenschaften* ..... 915, 916  
 GTK\_TYPE\_TOOLBAR ..... 929, 940  
*Eigenschaften* ..... 940  
 GTK\_TYPE\_TOOLTIP ..... 942  
 GTK\_TYPE\_VBOX ..... 895  
*Eigenschaften* ..... 895  
 GTK\_TYPE\_VBUTTON\_BOX ..... 902  
 GTK\_TYPE\_VPANED ..... 896, 899  
*Eigenschaften* ..... 900  
 GTK\_TYPE\_VSCALE ..... 917, 925  
*Eigenschaften* ..... 925  
*Signale* ..... 926  
 GTK\_TYPE\_VSEPARATOR ..... 892  
 GTK\_TYPE\_WINDOW ..... 878, 880  
*Eigenschaften* ..... 880  
`gtk_widget_add_accelerator()` ..... 939  
`gtk_widget_destroy()` ..... 876  
`gtk_widget_hide_all()` ..... 876  
`gtk_widget_hide()` ..... 876  
`gtk_widget_set_event()` ..... 976  
`gtk_widget_show_all()` ..... 875  
`gtk_widget_show()` ..... 875  
`gtk_window_add_accel_group()` ..... 939  
 GTK+ ..... 839  
*GNOME* ..... 841  
*übersetzen* ..... 843  
 GTK+ Widget, Textlabel-Eigenschaften ..... 889  
 GTK+-Bibliothek  
*Anwendung beenden* ..... 873  
*Anzeige-Elemente* ..... 886  
*Callback-Funktion* ..... 870  
*Container* ..... 875  
*Eigenschaften von Widgets* ..... 869  
*Events* ..... 870, 876, 973  
*Grafiken* ..... 893  
*Grundlagen* ..... 867  
*initialisieren* ..... 868  
*Klassen-Baum* ..... 871  
*Pango* ..... 890  
*Repertoire* ..... 915  
*Ressource-Files* ..... 981  
*Signal für Buttons* ..... 917  
*Signale* ..... 870, 876, 981  
*Stock Items* ..... 981  
*Stock-Element* ..... 941  
*Umlaute* ..... 877  
*UTF-8* ..... 877  
*Verarbeitungsschleife* ..... 876  
*Verarbeitungsschleife beenden* ..... 876  
*Widget erzeugen* ..... 868

*GTK+-Bibliothek (Forts.)*  
*Widgets packen* ..... 875

GTK+-Widget  
*Abkürzungsbuchstabe* ..... 940  
*Accelerator* ..... 935, 939, 940  
*anzeigen* ..... 875  
*auswählen* ..... 962  
*Bäume und Listen* ..... 980  
*Behälter* ..... 875, 894, 896, 903  
*Box* ..... 894  
*Button* ..... 908  
*Button-Box* ..... 902  
*Checkbutton* ..... 908  
*Combobox* ..... 944  
*Container* ..... 875  
*Dateiauswahl* ..... 962, 972  
*Dateneingabe* ..... 917  
*Dialogbox* ..... 881  
*Drag & Drop* ..... 981  
*Eigenschaften* ..... 869  
*Eigenschaften abfragen* ..... 916  
*Eigenschaften verändern* ..... 916  
*erzeugen* ..... 868  
*Farbauswahl* ..... 962  
*Fenster* ..... 878  
*Fenster-Eigenschaften* ..... 880  
*Fortschrittsbalken* ..... 894  
*Grafiken anzeigen* ..... 893  
*hierarchische Anordnung* ..... 874  
*Lineal* ..... 980  
*löschen* ..... 876  
*Menü* ..... 929, 934  
*Notizbuch* ..... 901  
*Optionsmenü* ..... 943  
*packen* ..... 875  
*Radio-Buttons* ..... 908, 915  
*Schiebeleiste* ..... 899  
*Schieberegler* ..... 918, 925  
*Schriftauswahl* ..... 962  
*scrollendes Fenster* ..... 960  
*Signale für Buttons* ..... 917  
*Statusleiste* ..... 893  
*Stellgröße* ..... 927  
*Tabelle* ..... 903  
*Tastatur-Shortcuts* ..... 939  
*Text (mehrzeilig)* ..... 948  
*Text(editor)* ..... 956  
*Textdarstellung (Eigenschaften)* ..... 959  
*Textfelder* ..... 918, 924  
*Textlabel* ..... 889  
*Text-Tags (Eigenschaften)* ..... 957

GTK+-Widget (Forts.)  
*Text-Widget-System* ..... 948  
*Toolbar* ..... 929, 940  
*Toolipp* ..... 942  
*Trennlinie* ..... 892  
*Umschaltbutton* ..... 908  
*Unterstriche* ..... 940  
*verstecken* ..... 876  
*Zahlenfelder* ..... 918, 926  
*Zwischenablage* ..... 980  
*GtkAccelGroup* ..... 939  
*GtkAdjustment* ..... 917, 927  
*GtkBox* ..... 894  
*GtkButton* ..... 908, 917  
*GtkCellRenderer* ..... 980  
*GtkCellRendererPixbuf* ..... 980  
*GtkCellRendererText* ..... 980  
*GtkCellRendererToggle* ..... 980  
*GtkCheckButton* ..... 916  
*GtkCheckButtons* ..... 908  
*GtkClipboard* ..... 949, 980  
*GtkColorSelection* ..... 962  
*GtkCombo* ..... 944  
*GtkDialog* ..... 881  
*GtkEditable* ..... 928  
*GtkEntry* ..... 917, 924, 926, 928, 944  
*GtkFileSelection* ..... 962, 972  
*GtkFontSelection* ..... 962  
*GtkHBox* ..... 894  
*GtkHButtonBox* ..... 902  
*GtkHPaned* ..... 896, 899  
*GtkHScale* ..... 917, 925  
*GtkImage* ..... 886, 893  
*GtkItemFactory* ..... 929, 934  
*GtkItemFactoryEntry* ..... 934  
*-Struktur* ..... 935  
*GtkLabel* ..... 886, 889, 940  
*GtkListStore* ..... 980  
*GtkMenu* ..... 929, 943  
*GtkMenuBar* ..... 929  
*GtkMessageDialog* ..... 886, 942  
*GtkNotebook* ..... 896, 901  
*GtkOptionsMenu* ..... 943  
*GtkProgressBar* ..... 894  
*GtkRadioButton* ..... 908, 915, 916  
*GtkRange* ..... 925  
*GtkRuler* ..... 980  
*GtkScale* ..... 917  
*GTK-Schnittstellen* ..... 841  
*GtkScrolledWindow* ..... 960  
*GtkSeparator* ..... 886, 892

**H**

*halt* ..... 1382  
*Handshake* ..... 671  
*head* ..... 1332  
*Heartbleed* ..... 684  
*horror()* ..... 573  
*High Level* ..... 58  
*Hintergrundprozess* ..... 317, 1082  
*hervorholen* ..... 1201  
*Höhere Ebene* ..... 58  
*host* ..... 1389  
*Host Byte Order* ..... 555  
*HOST\_NOT\_FOUND* ..... 574  
*hostname* ..... 1385  
*HP-UX* ..... 26, 31  
*htonl()* ..... 555, 1305  
*htons()* ..... 555, 1305  
*HTTP* ..... 542  
*HTTPS* ..... 542  
*hypot()* ..... 1269

**I**

*I18n* ..... 44  
*iconv* ..... 52  
*id* ..... 1344

if_Shell .....	1141
ifconfig .....	1386
IFS_Shellvariable .....	1175
IFTODT() .....	124
inet_addr() .....	564
inet_aton() .....	556, 564, 1306
inet_lnaof() .....	567, 1307
inet_makeaddr() .....	567, 1307
inet_netof() .....	566, 1307
inet_network() .....	566, 1306
inet_ntoa() .....	565, 1307
inet_ntop() .....	565, 1307
inet_pton() .....	564, 1307
info .....	1400
Ingres .....	692
init .....	308
init-Skript .....	308
Inline-Eingabeumleitung .....	1172
InnoDB .....	692
Inode ermitteln .....	151
INT_MAX .....	1245
Integer-Arithmetik, Shell .....	1109
Integertypen .....	1268
Interprozesskommunikation .....	
benannte Pipes .....	383
benannte Pipes (FIFO-Pipe) .....	351
FIFO-Pipes .....	383
Lock Files .....	356
Message Queue .....	353, 411
namenlose Pipes .....	350, 359
Pipes .....	359
Record Locking .....	356
Semaphore .....	353, 402
Shared Memory .....	354, 424, 435
Sockets .....	355
STREAMS .....	354
System-V-IPC .....	400
Unix-Domain-Sockets .....	639
Interprozesskommunikation (IPC) .....	349
inttypes .....	1267
ioctl() .....	99, 196
IP_ADD_MEMBERSHIP .....	628, 646
IP_DROP_MEMBERSHIP .....	628, 647
IP_HDRINCL .....	627
IP_MULTICAST_IF .....	647
IP_MULTICAST_IF .....	628
IP_MULTICAST_LOOP .....	628, 647
IP_MULTICAST_TTL .....	628, 647
IP_OPTIONS .....	627
IP_RECVSTADDR .....	627
IP_RECVIF .....	627
IP_TOS .....	628
IP_TTL .....	628
IPC_CREAT .....	407, 412, 425
IPC_EXCL .....	407, 412, 425
IPC_INFO .....	409, 415, 426
IPC_PRIVATE .....	407, 412, 425
IPC_RMD .....	415
IPC_RMID .....	409, 426
IPC_SET .....	409, 415, 426
IPC_STAT .....	409, 415, 425
IPv4 portieren nach IPv6 .....	659
IPv6_ADD_MEMBERSHIP .....	629
IPv6_ADDRFORM .....	628
IPv6_CHECKSUM .....	628
IPv6_DROP_MEMBERSHIP .....	629
IPv6_DSTOPTS .....	628
IPv6_HOPLIMIT .....	628
IPv6_HOPOPTS .....	629
IPv6_MULTICAST_HOPS .....	629
IPv6_MULTICAST_IF .....	629
IPv6_MULTICAST_LOOP .....	629
IPv6_NEXTHOP .....	629
IPv6_PKTINFO .....	629
IPv6_PKTOPTIONS .....	629
isalnum() .....	1240
isalpha() .....	1240
isascii() .....	1241
iscntrl() .....	1240
isdigit() .....	1240
isgraph() .....	1240
islower() .....	1240
iso646 .....	1263
isprint() .....	1241
ispunct() .....	1241
isspace() .....	1241
isupper() .....	1241
isxdigit() .....	1241

**J**

jmp_buf .....	1249
jobs .....	1348
jobs, Shell .....	1203
Jobverwaltung .....	317
bg .....	319
fg .....	319
Shell .....	1202
JSON .....	695

**K**

Kanarienvögel .....	55
Keep Alive .....	619
Kernelinformationen .....	184
Kernel-Level .....	58
kill .....	1348
Shell .....	1197
kill() .....	326, 340, 1293
killall .....	1348
Kommando .....	
adduser .....	1345
afio .....	1367
alias .....	1402
apropos .....	1400
arp .....	1385
at .....	1346
badblocks .....	1355
basename .....	1341
batch .....	1347
bc .....	1112, 1403
bg .....	1203, 1348
bunzip2 .....	1366
bzcat .....	1321
bzip2 .....	1366
cal .....	1380
cat .....	1322
cd .....	1341
cfdisk .....	1356
chgrp .....	1322
chmod .....	1324
chown .....	1325
cksum .....	1322
clear .....	1397
cmp .....	1325
comm .....	1325
compress .....	1367
cp .....	1326
cpio .....	1367
cron .....	1348
csplit .....	1327
cut .....	1115, 1327
date .....	1381
dd .....	1356
dd_rescue .....	1359
df .....	1353
diff .....	1328
diff3 .....	1328
dig .....	1389
dircmp .....	1341
dirname .....	1341

## Kommando (Forts.)

dmesg .....	1382
dos2unix .....	1329
Drucker- .....	1383
du .....	1354
dump .....	1370
dump2fs .....	1359
e2fsck .....	1359
echo .....	1166
env .....	1403
exec .....	1184
exit .....	1342
expand .....	1329
export .....	1123
fdisk .....	1360
fg .....	1201, 1348
file .....	1329
find .....	1330
finger .....	1342
fold .....	1331
free .....	1355
fsck .....	1362
getopts .....	1139
groupadd .....	1343
groupdel .....	1343
groupmod .....	1343
groups .....	1343
gunzip .....	1372
gzip .....	1372
halt .....	1382
head .....	1332
host .....	1389
hostname .....	1385
id .....	1344
ifconfig .....	1386
info .....	1400
jobs .....	1203, 1348
kill .....	1197, 1348
killall .....	1348
last .....	1344
less .....	1332
let .....	1110
ln .....	1332
logname .....	1344
logout .....	1342
ls .....	1333
mail .....	1387
mailx .....	1387
man .....	1401
md5 .....	1322
md5sum .....	1322

## Kommando (Forts.)

<i>mesg</i>	1397
<i>mkdir</i>	1341
<i>mksfs</i>	1363
<i>mknod</i>	1188
<i>mkswap</i>	1364
<i>more</i>	1333
<i>mount</i>	1365
<i>mt</i>	1373
<i>mv</i>	1333
<i>netstat</i>	1389
<i>newgrp</i>	1344
<i>nice</i>	1349
<i>nl</i>	1334
<i>nohup</i>	1349
<i>nslookup</i>	1389
<i>od</i>	1335
<i>pack</i>	1374
<i>parted</i>	1366
<i>passwd</i>	1344
<i>paste</i>	1116, 1335
<i>pcat</i>	1335
<i>pgrep</i>	1350
<i>ping</i>	1390
<i>Postscript</i>	1402
<i>printenv</i>	1403
<i>printf</i>	1168
<i>prtvtoc</i>	1366
<i>ps</i>	1349
<i>pstree</i>	1351
<i>pwd</i>	1104, 1342
<i>rcp</i>	1391
<i>read</i>	1170
<i>reboot</i>	1382
<i>renice</i>	1351
<i>reset</i>	1397
<i>restore</i>	1370
<i>r-Kommmandos</i>	1391
<i>rlogin</i>	1391
<i>rm</i>	1335
<i>rmdir</i>	1342
<i>rsh</i>	1391
<i>rsync</i>	1394
<i>rwho</i>	1391
<i>scp</i>	1392
<i>set</i>	1137
<i>setterm</i>	1398
<i>shift</i>	1136
<i>shutdown</i>	1382
<i>sleep</i>	1351
<i>sort</i>	1335

## Kommando (Forts.)

<i>split</i>	1336
<i>ssh</i>	1391
<i>ssh-keygen</i>	1393
<i>stty</i>	1398
<i>su</i>	1351
<i>sudo</i>	1352
<i>sum</i>	1322
<i>swap</i>	1355
<i>swapoff</i>	1366
<i>swapon</i>	1366
<i>sync</i>	1366
<i>tac</i>	1337
<i>tail</i>	1188, 1337
<i>tar</i>	1374
<i>tee</i>	1099, 1338
<i>test</i>	1146
<i>time</i>	1353
<i>top</i>	1353
<i>touch</i>	1338
<i>tr</i>	1117, 1338
<i>traceroute</i>	1395
<i>trap</i>	1198
<i>tty</i>	1399
<i>type</i>	1339
<i>typeset</i>	1111
<i>ufsdump</i>	1370
<i>ufsrestore</i>	1370
<i>umask</i>	1339
<i>umount</i>	1365
<i>unalias</i>	1402
<i>uname</i>	1381
<i>uncompress</i>	1367
<i>uniq</i>	1339
<i>unix2dos</i>	1340
<i>unpack</i>	1374
<i>unzip</i>	1379
<i>uptime</i>	1382
<i>useradd</i>	1345
<i>userdel</i>	1345
<i>usermod</i>	1345
<i>uudecode</i>	1388
<i>uuencode</i>	1388
<i>wait</i>	1201
<i>wall</i>	1396
<i>wc</i>	1340
<i>whatis</i>	1402
<i>whereis</i>	1340
<i>who</i>	1346
<i>whoami</i>	1346
<i>write</i>	1396

## Kommando (Forts.)

<i>zcat</i>	1341
<i>zip</i>	1379
<i>zless</i>	1341
<i>zmore</i>	1341
Kommandoausführung überprüfen, Shell	1141
Kommandosubstitution	1132
Kommandozeilenargument, Shell	1133
Kommandozeilenoptionen auswerten, Shell	1139
Kommunikationsmodell, Socket	544
Korn-Shell	1079
<i>ksh</i>	1079

## L

L10N	44
<i>labs()</i>	1258
<i>last</i>	1344
Laufzeitumgebung	29
<i>lchown()</i>	1286
lconv-Struktur	1247
LDBL_DIG	1243
LDBL_EPSILON	1244
LDBL_MANT_DIG	1243
LDBL_MAX	1244
LDBL_MAX_10_EXP	1244
LDBL_MAX_EXP	1244
LDBL_MIN	1244
LDBL_MIN_10_EXP	1243
LDBL_MIN_EXP	1243
<i>ldconfig</i>	1025
<i>ldiv()</i>	1258
<i>less</i>	1332
<i>let</i>	1110
LEX	1072
<i>libpq-fe</i>	1315
LibreOffice Base	708
<i>limits</i>	61, 1244, 1271
Link	151
abfragen	153
Dangling Symlink	153
harter	152
symbolischer	152
LINK_MAX	1272
<i>link()</i>	153
Linker	985
Linux	31
Geschichte	31
Torvalds	32
<i>listen()</i>	550, 1303

## Listing

<i>add_db.c</i>	773
<i>addr.c</i>	567
<i>admin.c</i>	770
<i>backward.c</i>	75, 160
<i>cdrom.c</i>	212
<i>child.c</i>	272
<i>child2.c</i>	275
<i>child3.c</i>	276
<i>child4.c</i>	278
<i>client_msq.c</i>	420
<i>client_shm.c</i>	432
<i>client.c (FIFO-Pipes)</i>	396
<i>client.c (Multicast)</i>	648
<i>client.c (TCP)</i>	556
<i>client.c (TCP/linear)</i>	586
<i>client.c (UDP)</i>	636
<i>cpy_file_mmap.c</i>	1278
<i>cpy_file.c</i>	73
<i>daemon.c</i>	305
<i>dup_fd.c</i>	78
<i>dynamisch.c</i>	1027
<i>environ1.c</i>	258
<i>environ2.c</i>	258
<i>eventloop1.c</i>	248
<i>eventloop2.c</i>	249
<i>exec_child.c</i>	294
<i>exec1.c</i>	290
<i>exec2.c</i>	291
<i>exec3.c</i>	291
<i>exec4.c</i>	292
<i>exec5.c</i>	292
<i>exec6.c</i>	293
<i>fifo_buf.c</i>	389
<i>fifo1.c</i>	385
<i>fifo2.c</i>	386
<i>fifo4.c</i>	389
<i>file_size.c</i>	157
<i>file_times.c</i>	158
<i>filetest.c</i>	141
<i>filetest2.c</i>	142
<i>filter.c</i>	369
<i>find_dir.c</i>	133
<i>ftwalk.c</i>	136
<i>get_env.c</i>	259
<i>glib1.c</i>	846
<i>glib2.c</i>	848
<i>glib3.c</i>	850
<i>glib4.c</i>	853
<i>glib5.c</i>	855
<i>glib6.c</i>	858

Listing (Forts.)	
<i>glib7.c</i>	860
<i>glib8.c</i>	863
<i>gtk1.c</i>	878
<i>gtk2b.c</i>	890
<i>gtk3.c</i>	896
<i>gtk3b.c</i>	903
<i>gtk4.c</i>	908
<i>gtk5.c</i>	918
<i>gtk6.c</i>	929
<i>gtk6b.c</i>	945
<i>gtk7.c</i>	949
<i>gtk8.c</i>	962
<i>gtk9.c</i>	977
<i>index_news.c</i>	777
<i>kernelinf.c</i>	185
<i>key_ftok.c</i>	401
<i>list_wd.c</i>	127
<i>logging.c</i>	302
<i>login.c</i>	767
<i>Login.html</i>	765
<i>make_file.c</i>	64
<i>memory.c</i>	172, 1065
<i>msq_header.h</i>	416
<i>my_cgi.h</i>	758
<i>my_find.c</i>	295
<i>my_getpid.c</i>	181
<i>my_limit.c</i>	267
<i>my_link.c</i>	154
<i>my_programm.c</i>	293
<i>my_setlocale.c</i>	1246
<i>my_stat.c</i>	148
<i>mychdir.c</i>	121
<i>myequal.c</i>	1020
<i>myequal.h</i>	1020
<i>myinfo.c</i>	176
<i>mymkdir.c</i>	119
<i>mysql1.c</i>	720
<i>mysql2.c</i>	722
<i>mysql3.c</i>	727
<i>mysql4.c</i>	738
<i>non_block.c</i>	654
<i>offsetof()</i>	1252
<i>openCD.c</i>	99
<i>ping_pong.c</i>	338
<i>pipe1.c</i>	362
<i>pipe2.c</i>	365
<i>pipe3.c</i>	367
<i>pipe3b.c</i>	371
<i>play_fd.c</i>	83
<i>poll_stdin_time.c</i>	97
Listing (Forts.)	
<i>polling_fifo.c</i>	392
<i>popen1.c</i>	373
<i>popen2.c</i>	375
<i>postgre1.c</i>	813
<i>postgre3.c</i>	817, 823
<i>printme.c</i>	378
<i>printme2.c</i>	380
<i>prio_child.c</i>	280
<i>prio.c</i>	247
<i>proz_dat.c</i>	270
<i>pserver.c (TCP/parallel)</i>	589
<i>put_env.c</i>	260
<i>put_env2.c</i>	261
<i>reverse.c</i>	793
<i>scan_dir.c</i>	131
<i>search_db.c</i>	781
<i>sem.c</i>	403
<i>sender.c</i>	394
<i>server_msq.c</i>	418
<i>server_shm.c</i>	430
<i>server.c (FIFO-Pipes)</i>	397
<i>server.c (Multicast)</i>	647
<i>server.c (TCP)</i>	560
<i>server.c (TCP/linear)</i>	583
<i>server.c (UDP)</i>	634
<i>set_env.c</i>	263
<i>shm_header.h</i>	427
<i>sig_sync.c</i>	345
<i>sig.c</i>	334
<i>sperre.c</i>	92
<i>strxcat.c</i>	1251
<i>summe.c</i>	794
<i>testlist.c</i>	1023
<i>thserver.c (TCP/Thread)</i>	621
<i>trash.c</i>	85
<i>ugid.c</i>	150
<i>unset_env.c</i>	264
<i>waise.c</i>	282
<i>wait.c</i>	285
<i>waitpid.c</i>	288
<i>write_file.c</i>	68
<i>write_vec.c</i>	101
<i>zombie.c</i>	283
Little Endian	555
ln	1332
Locale	524
locale.h	1245
localeconv()	1247
localtime()	1261
Lock File	356

Lock Files	356
lockf()	87, 95
Locking	833
LOG_ALERT	298
LOG_AUTH	298
LOG_AUTHPRIV	298
LOG_CONS	300
LOG_CRIT	298
LOG_CRON	299
LOG_DAEMON	299
LOG_DEBUG	298
LOG_EMERG	298
LOG_ERR	298
LOG_FTP	298
LOG_INFO	298
LOG_KERN	299
LOG_LOCAL0	299
LOG_LPR	299
LOG_MAIL	299
LOG_NDELAY	300
LOG_NEWS	299
LOG_NOTICE	298
LOG_PERROR	300
LOG_PID	300
LOG_SYSLOG	299
LOG_USER	299
LOG_UUCP	299
LOG_WARNING	298
log()	1249
log10()	1249
log2()	1269
logb()	1269
Logischer Operator, Shell	1154
logname	1344
logout	1342
LONG_MAX	1245
longjmp()	1249
Low Level	57
ls	1333
lseek()	74, 1273
lstat()	146, 1284
<b>M</b>	
mail	1387
mailx	1387
Make	994
<i>Abhangigkeit</i>	1000
<i>Abkurzungen</i>	1004
<i>Anwendungen installieren</i>	1010
<i>implizite Regeln</i>	1007
Make (Forts.)	
<i>Kommentare</i>	999
<i>Makefile</i>	997
<i>Makros</i>	1004
<i>Makros (Übersicht)</i>	1005
<i>Musterregeln</i>	1009
<i>Tabulator</i>	999
<i>Variablen</i>	1006
<i>Ziel (Target)</i>	998
<i>Makefile</i>	997
<i>malloc()</i>	1257
<i>man</i>	1401
<i>Mandatory Locking</i>	87, 95
<i>mit Linux</i>	88
<i>MariaDB</i>	692
<i>math</i>	1248, 1268
<i>MAX_CANON</i>	1272
<i>MAX_INPUT</i>	1271
<i>MB_LEN_MAX</i>	1245
<i>MD5</i>	674
<i>md5</i>	1322
<i>md5sum</i>	1322
<i>memchr()</i>	1259
<i>memcmp()</i>	1259
<i>memcpy()</i>	49, 1259
<i>memlockall()</i>	1280
<i>memmove()</i>	49, 1259
<i>Memory Leaks</i>	54, 1064
<i>efence</i>	1065
<i>valgrind</i>	1068
<i>Memory Management Unit → MMU</i>	
<i>memset()</i>	49, 1259
<i>mesg</i>	1397
<i>Message Queue</i>	353, 411
<i>ndern</i>	414
<i>erfragen</i>	414
<i>erzeugen</i>	412
<i>lschen</i>	414
<i>Nachricht empfangen</i>	414
<i>Nachricht versenden</i>	413
<i>ffnen</i>	412
<i>struct msqid_ds</i>	415
<i>mkdir</i>	1341
<i>mkdir()</i>	119, 1281
<i>mkfifo</i>	1188
<i>mkfifo()</i>	385, 1294
<i>mkfs</i>	1363
<i>mknod</i>	1188
<i>mknod()</i>	388
<i>mkstemp()</i>	117
<i>mkswap</i>	1364

mktemp() ..... 117  
 mktime() ..... 1261  
 mlock() ..... 1280  
 mmap() ..... 1276  
 MMU ..... 526  
 modf() ..... 1249  
 more ..... 1333  
 mount ..... 1365  
 MSG\_WAITALL ..... 580  
 msgctl() ..... 414, 1295  
 msgget() ..... 412, 1295  
 msgrcv() ..... 414, 1295  
 msgsnd() ..... 413, 1295  
 msync() ..... 1278  
 mt ..... 1373  
 Multicast-Socket ..... 645  
 Multicast-Socket, Anwendungsgebiete ..... 653  
 Multics ..... 29  
 Multiplexing ..... 96, 1274  
 Multiplexing I/O ..... 598  
 Multiversion Concurrency Control → MVCC  
 munlock() ..... 1280  
 munlockall() ..... 1280  
 munmap() ..... 1278  
 Mutexe ..... 462  
 mv ..... 1333  
 MVCC ..... 692  
 my\_ulonglong ..... 1315  
 MyISAM ..... 692  
 MySQL ..... 1314  
 MySQL ..... 691  
     Daten ändern ..... 710  
     Daten ausgeben ..... 711  
     Daten einfügen ..... 708  
     Daten importieren ..... 711  
     Daten löschen ..... 710  
     Index ..... 703  
     NULL ..... 713  
     Schlüsselfelder ..... 702  
     Subquery ..... 709  
     UDF-Schnittstelle ..... 787  
     unscharfe Suche ..... 714  
 MySQL C-API ..... 715  
     Benutzer wechseln ..... 750  
     Datenbanknamen abfragen ..... 729  
     Ergebnismenge ..... 745  
     Ergebnismenge bearbeiten ..... 731  
     Fehlerbehandlung ..... 719  
     Fehlercodes ..... 719  
     Feldcursor abfragen ..... 737  
     Feldcursor platzieren ..... 737  
 MySQL C-API (Forts.)  
     Informationen abfragen ..... 725  
     my\_ulonglong ..... 732  
     MYSQL\_FIELD ..... 733  
     MYSQL\_FIELD\_OFFSET ..... 736  
     MYSQL\_RES ..... 729  
     MYSQL\_ROW ..... 731  
     MySQL-Objekt initialisieren ..... 717  
     Server-Threads ermitteln ..... 730  
     spaltenweise abarbeiten ..... 733  
     SQL-Befehle an den Server ..... 746  
     Tabellennamen abfragen ..... 730  
     veralte Funktionen ..... 756  
     Verbindung herstellen ..... 717  
     Verbindung schließen ..... 720  
     weitere Funktionen ..... 750  
     Zeilencursor abfragen ..... 732  
     Zeilencursor platzieren ..... 732  
     zeilenweise abarbeiten ..... 731  
     Zugangsdaten ..... 757  
 mysql\_affected\_rows() ..... 749, 1309  
 mysql\_change\_user() ..... 750, 1310  
 mysql\_character\_set\_name() ..... 1310  
 mysql\_close() ..... 720, 1309  
 mysql\_connect() ..... 719, 1309  
 mysql\_create\_db() ..... 1310  
 mysql\_data\_seek() ..... 745, 1310  
 mysql\_debug() ..... 756, 1310  
 mysql\_drop\_db() ..... 1310  
 mysql\_dump\_debug\_info() ..... 756, 1310  
 mysql\_eof() ..... 1310  
 mysql\_errno() ..... 719, 1310  
 mysql\_error() ..... 719, 1310  
 mysql\_escape\_string() ..... 751, 1311  
 mysql\_fetch\_field\_direct() ..... 736, 1311  
 mysql\_fetch\_field() ..... 735, 1311  
 mysql\_fetch\_fields() ..... 736  
 mysql\_fetch\_lengths() ..... 746  
 mysql\_fetch\_row() ..... 731, 1311  
 MYSQL\_FIELD ..... 1315  
 mysql\_field\_count() ..... 737, 1311  
 MYSQL\_FIELD\_OFFSET ..... 1315  
 mysql\_field\_seek() ..... 737, 1311  
 mysql\_field\_tell() ..... 737, 1311  
 mysql\_free\_result() ..... 730, 750, 1311  
 mysql\_get\_client\_info() ..... 725, 1312  
 mysql\_get\_host\_info() ..... 726, 1312  
 mysql\_get\_proto\_info() ..... 726, 1312  
 mysql\_get\_server\_info() ..... 726, 1312  
 mysql\_info() ..... 726, 1312  
 mysql\_init() ..... 717, 1312

mysql\_insert\_id() ..... 751, 1312  
 mysql\_kill() ..... 1312  
 mysql\_list\_dbs() ..... 729, 1312  
 mysql\_list\_fields() ..... 730, 1312  
 mysql\_list\_processes() ..... 730, 1312  
 mysql\_list\_tables() ..... 1313  
 mysql\_num\_fields() ..... 731, 737, 1313  
 mysql\_num\_rows() ..... 732, 1313  
 mysql\_options() ..... 752, 1313  
 mysql\_ping() ..... 754, 1313  
 mysql\_query() ..... 746, 1313  
 mysql\_real\_connect() ..... 717, 719, 1313  
 mysql\_real\_escape\_string() ..... 751, 1310  
 mysql\_real\_query() ..... 746, 1313  
 mysql\_reload() ..... 1313  
 MYSQL\_RES ..... 1315  
 MYSQL\_ROW ..... 1315  
 MYSQL\_ROW\_OFFSET ..... 732  
 mysql\_row\_seek() ..... 732, 1314  
 mysql\_row\_tell() ..... 732, 1314  
 mysql\_select\_db() ..... 755, 1314  
 mysql\_shutdown() ..... 755, 1314  
 mysql\_stat() ..... 1314  
 mysql\_store\_result() ..... 737, 747, 1314  
 mysql\_thread\_id() ..... 755, 1314  
 mysql\_thread\_safe() ..... 1314  
 mysql\_use\_result() ..... 749, 1314  
 MySQL-Befehl  
     ALTER TABLE ..... 705  
     benutzerdefinierte Funktionen ..... 793  
     CREATE FUNCTION ..... 797  
     DELETE ..... 710  
     DROP FUNCTION ..... 797  
     eigene Funktionen schreiben ..... 787  
     erweitern in C ..... 787  
     INSERT INTO ..... 708  
     LOAD DATA INFILE ..... 711  
     MODIFY ..... 707  
     SELECT ..... 711  
     UPDATE ..... 710  
**N**  
 NAME\_MAX ..... 1272  
 Named Pipes  
     Shell ..... 1188  
 Namen-Expansion  
     Shell ..... 1101  
 Nameserver ..... 571  
 nanosleep() ..... 342, 1293  
 netdb ..... 1307, 1308  
 netinet ..... 1305  
 netstat ..... 1389  
 Network Byte Order ..... 555  
 Netzwerkprogrammierung ..... 533  
     Adressfamilie ..... 545  
     auf Verbindung warten ..... 550  
     Bibliotheken ..... 665  
     Big Endian ..... 555  
     Clientanwendung ..... 553  
     Clientanwendung (UDP) ..... 632  
     Daten empfangen ..... 551, 552, 633  
     Daten senden ..... 551, 552, 633  
     Datenformat ..... 597  
     Grundlagen ..... 663  
     Host Byte Order ..... 555  
     IPv4 nach IPv6 ..... 659  
     Konverter-Funktionen ..... 563  
     Little Endian ..... 555  
     Multicast-Socket ..... 645  
     Network Byte Order ..... 555  
     nichtblockierendes Socket ..... 653  
     OpenSSL ..... 665  
     parallele Server ..... 582  
     Protokollfamilie ..... 545  
     Pufferung ..... 580  
     Raw Socket ..... 657  
     RPC ..... 658  
     Schlüsselgenerierung ..... 664  
     Serveranwendung ..... 558  
     Serveranwendung (UDP) ..... 633  
     Socket ..... 543  
     Socket an Port binden ..... 550  
     Socket anlegen ..... 545  
     Socket-Optionen ..... 625  
     Socketschnittstelle ..... 545  
     Socket-Typen ..... 546  
     systemabhängig ..... 555  
     Threads ..... 620  
     TLI ..... 657  
     UDP ..... 630  
     Verbindung schließen ..... 553  
     Verbindungsauflauf (Client) ..... 547  
     Verbindungswünsche abarbeiten ..... 551  
     XTI ..... 657  
 Netzwerktechnik  
     Anwendungsschicht ..... 536  
     Bitübertragungsschicht ..... 534  
     Darstellungsschicht ..... 535  
     DNS ..... 538  
     Internet (www) ..... 537  
     Kommunikationsmodell ..... 544

Netzwerktechnik (Forts.)	
<i>Kommunikationsschicht</i>	535
<i>Netzklasse</i>	566
<i>Netzwerknummer</i>	566
<i>Ports</i>	541
<i>Protokolle</i>	541
<i>Referenzmodell</i>	534
<i>RFC</i>	541
<i>Sicherungsschicht</i>	535
<i>TCP/IP</i>	539
<i>TCP/IP-Schichtenmodell</i>	536
<i>Transportschicht</i>	535
<i>Vermittlungsschicht</i>	535
<i>newgrp</i>	1344
<i>nftw()</i>	139, 1283
<i>NGROUPS_MAX</i>	1271
<i>nice</i>	1349
<i>Nice-Wert</i>	520
<i>Nichtblockierendes Socket</i>	653
<i>NIS</i>	237
<i>nl</i>	1334
<i>NLS</i>	44
<i>No eXecute</i>	349
<i>NO_ADDRESS</i>	574
<i>NO_DATA</i>	574
<i>NO_RECOVERY</i>	574
<i>nohup</i>	1349
<i>NoSQL</i>	695
<i>nslookup</i>	1389
<i>ntohl()</i>	555, 1306
<i>ntohs()</i>	555, 1306
<i>NTP</i>	542
<b>O</b>	
<i>O_ACCMODE</i>	82
<i>O_APPEND</i>	61, 62, 74
<i>O_ASYNC</i>	70, 82
<i>O_CREAT</i>	62, 67
<i>O_EXCL</i>	63
<i>O_NDELAY</i>	363, 364
<i>O_NOCTTY</i>	63
<i>O_NONBLOCK</i>	63, 96, 363, 364, 388
<i>O_RDONLY</i>	62
<i>O_RDWR</i>	62
<i>O_SYNC</i>	63, 70, 102
<i>O_TRUNC</i>	63, 67, 104
<i>O_WRONLY</i>	62
<i>od</i>	1335
<i>off_t</i>	75
<i>offsetof()</i>	1252
<i>OPEN_MAX</i>	61, 68, 97, 1272
<i>open()</i>	61, 1273
<i>opendir()</i>	125, 1282
<i>openlog()</i>	299
<i>OpenSSL</i>	665
<i>Fehlermeldungen</i>	678
<i>Heartbleed</i>	684
<i>Konfiguration</i>	679
<i>Referenz</i>	675
<i>OPENSSL_config()</i>	674
<i>Oracle</i>	693
<i>OSI-Schichtenmodell</i>	534
<i>Out-of-Memory-Killer</i>	47
<i>Overcommitment</i>	47
<b>P</b>	
<i>pack</i>	1374
<i>Paketmanagement</i>	36
<i>Pango</i>	890
<i>parted</i>	1366
<i>PASS_MAX</i>	1271
<i>passwd</i>	219, 1344
<i>Struktur</i>	221
<i>Passworddatei</i>	220, 225
<i>paste</i>	1335
<i>Shell</i>	1116
<i>PATCH</i>	1073
<i>PATH_MAX</i>	1272
<i>pause()</i>	332, 342, 1293
<i>pcat</i>	1335
<i>pclose()</i>	1294
<i>pdksh</i>	1080
<i>perror()</i>	62, 1256
<i>PF_APPLETALK</i>	545
<i>PF_ATMPVC</i>	545
<i>PF_AX25</i>	545
<i>PF_INET</i>	545
<i>PF_INET6</i>	545
<i>PF_IPX</i>	545
<i>PF_LOCAL</i>	545
<i>PF_NETLINK</i>	545
<i>PF_PACKET</i>	545
<i>PF_UNIX</i>	545
<i>PF_X25</i>	545
<i>pgrep</i>	1350
<i>PID</i>	243
<i>PID_MAX</i>	1272
<i>PID-Lock</i>	81
<i>ping</i>	1390
<i>PIPE_BUF</i>	369, 388, 1272

<i>Pipe, Shell</i>	1097
<i>pipe()</i>	359, 1294
<i>Pipes</i>	359
<i>Drucken mit lpr</i>	378
<i>Eigenschaften</i>	359
<i>elementare Ein-/Ausgabe</i>	363
<i>Filterprogramm</i>	369
<i>Mail versenden</i>	374
<i>Standard-Ein-/Ausgabe</i>	364
<i>umleiten</i>	366
<i>Pipes (benannt)</i>	351, 383
<i>Pipes (namenlos)</i>	350
<i>PIPESTATUS, Shell</i>	1143
<i>POODLE-Angriff</i>	665
<i>POP3</i>	542
<i>popen()</i>	372, 1230, 1294
<i>Positionsparameter, Shell</i>	1133
<i>POSIX</i>	42
<i>POSIX_CHOWN_RESTRICTED</i>	1272
<i>POSIX_JOB_CONTROL</i>	1272
<i>POSIX_NO_TRUNC</i>	1272
<i>POSIX_PRIORITY_SCHEDULING</i>	249
<i>POSIX_SAVED_IDS</i>	1272
<i>POSIX_SOURCE</i>	1273
<i>POSIX_VERSION</i>	1272
<i>PostgreSQL</i>	692
<i>pgpass</i>	831
<i>Daten ausgeben</i>	805
<i>Daten hinzufügen</i>	804
<i>Datenbank verwenden</i>	801
<i>Datentypen</i>	800
<i>grafische Frontends</i>	803
<i>Konfigurationsdateien</i>	798
<i>Passworddatei</i>	830
<i>pg_hba.conf</i>	798
<i>phpPgAdmin</i>	803
<i>postgresql.conf</i>	798
<i>Server starten/stoppen</i>	799
<i>Tabelle anlegen</i>	804
<i>Tabelle löschen</i>	804
<i>Umgebungsvariablen</i>	830
<i>unscharfe Suche</i>	806
<i>PostgreSQL C-API</i>	807
<i>Anfrage an den Server</i>	814
<i>Anwendung übersetzen</i>	807
<i>Ergebnis einer Anfrage</i>	816
<i>Fehlerbehandlung</i>	811
<i>Informationen zur Verbindung</i>	810
<i>NULL</i>	822
<i>Rückgabe einer Anfrage auslesen</i>	821
<i>Status der Verbindung</i>	810
<i>PostgreSQL C-API (Forts.)</i>	
<i>Status einer Anfrage</i>	815
<i>Statuszeichenkette einer Anfrage ermitteln</i>	822
<i>Threads</i>	831
<i>Verbindung herstellen</i>	808
<i>Verbindung schließen</i>	812
<i>Verbindung wiederaufnehmen</i>	813
<i>PostgreSQL-Befehl</i>	
<i>CREATE TABLE</i>	804
<i>DELETE FROM</i>	805
<i>DROP TABLE</i>	804
<i>INSERT INTO</i>	804
<i>SELECT</i>	805
<i>UPDATE</i>	805
<i>Postscript-Kommandos</i>	1402
<i>pow()</i>	1249
<i>PPID</i>	243
<i>PQbackendPID()</i>	812, 1317
<i>PQbinaryTuples()</i>	817, 1318
<i>PQclear()</i>	817, 1318
<i>PQcmdStatus()</i>	822, 1319
<i>PQcmdTuples()</i>	822, 1319
<i>PQconnectdb()</i>	808, 1315
<i>PQconnectPoll()</i>	808, 1316
<i>PQconnectStart()</i>	808, 1316
<i>PQdb()</i>	812, 1316
<i>PQerrorMessage()</i>	811
<i>PQexec()</i>	814, 1317
<i>PQfinish()</i>	812, 1317
<i>PQfmod()</i>	816, 1318
<i>PQfname()</i>	816, 1318
<i>PQfnnumber()</i>	816, 1318
<i>PQfsize()</i>	816, 1318
<i>PQftype()</i>	816, 1318
<i>PQgetisnull()</i>	822, 1319
<i>PQgetlength()</i>	822, 1319
<i>PQgetssl()</i>	812, 1317
<i>PQgetvalue()</i>	821, 1319
<i>PQhost()</i>	812, 1316
<i>PQnfields()</i>	816, 1318
<i>PQntuples()</i>	816, 1318
<i>PQoidStatus()</i>	823, 1319
<i>PQoidValue()</i>	823, 1319
<i>PQoptions()</i>	812, 1316
<i>PQpass()</i>	812, 1316
<i>PQport()</i>	812, 1316
<i>PQreset()</i>	813, 1317
<i>PQresetPoll()</i>	813, 1317
<i>PQresetStart()</i>	813, 1317
<i>PQresStatus()</i>	815, 1317
<i>PQresultErrorMessage()</i>	816, 1318

PQresultStatus() ..... 815, 1317  
 PQsetdb() ..... 808, 1316  
 PQsetdbLogin() ..... 808, 1315  
 PQsocket() ..... 812, 1316  
 PQstatus ..... 810  
 PQstatus() ..... 1316  
 PQtty() ..... 812, 1316  
 PQuser() ..... 812, 1316  
 Präprozessor ..... 984  
 printenv ..... 1403  
 printf  
    *Shell* ..... 1168  
 printf() ..... 109, 1254  
 proc ..... 171, 173, 174, 175, 176, 179, 180,  
      181, 182, 183, 184, 190, 191  
 Process Control Block ..... 60  
 Profiling ..... 991, 1042  
    *Laufzeit einzelner Codezeilen* ..... 1047  
    *Laufzeit von Funktionen* ..... 1043  
    *Laufzeit von Prozessen* ..... 1042  
 Prozess ..... 241  
    *Ausführung* ..... 307  
    *Auslagerung* ..... 251  
    *Benutzernummer (UID, EUID)* ..... 243  
    *cron-Job* ..... 321  
    *Dämon-* ..... 296, 308  
    *Deadlocks* ..... 358  
    *Eltern-* ..... 272  
    *Erzeugung* ..... 271  
    *Flusskontrolle* ..... 358  
    *Gruppennummer (GID, EGID)* ..... 243  
    *Hintergrund-* ..... 317  
    *init* ..... 256, 308  
    *init-Skript* ..... 308  
    *Jobverwaltung* ..... 317  
    *Kenndaten* ..... 242  
    *Kind-* ..... 272  
    *Kind überlagern* ..... 293  
    *Kommando ps* ..... 252  
    *komplett ersetzen* ..... 289  
    *Lebenszyklus* ..... 255  
    *Limits* ..... 265  
    *paralleler Server (TCP)* ..... 582  
    *Priorität* ..... 245  
    *Priorität verändern* ..... 280  
    *Prozesserkennung* ..... 269  
    *Prozessnummer (PID)* ..... 243  
    *Prozessnummer des Vaters (PPID)* ..... 243  
    *Pufferung* ..... 274  
    *Race Condition* ..... 358  
    *Runlevel* ..... 309  
 Prozess (Forts.)  
    *Scheduling-Priorität abfragen* ..... 246  
    *Scheduling-Priorität verändern* ..... 246  
    *Startup-Skripte* ..... 310  
    *Status* ..... 244  
    *Steuerterminal* ..... 252  
    *stoppen* ..... 342, 344  
    *stoppen (Zeit)* ..... 342  
    *suspendieren* ..... 342, 344  
    *suspendieren (Zeit)* ..... 342  
    *Swapping* ..... 251  
    *synchronisieren (Signale)* ..... 344  
    *Timesharing* ..... 246  
    *überwachen* ..... 252  
    *Umgebungsvariablen* ..... 257  
    *Vererbung* ..... 279  
    *warten* ..... 282  
    *Warteschleife* ..... 248  
    *zeitgesteuert ausführen* ..... 321  
    *Zugriffsdisziplin* ..... 358  
    *Zustände* ..... 244  
 Prozesse  
    *Signale* ..... 328  
 Prozesstabelleneintrag ..... 60, 326  
 prtctoc ..... 1366  
 ps ..... 1349  
 pstree ..... 1351  
 pthread ..... 1297  
 pthread\_attr\_destroy() ..... 453, 1298, 1299  
 pthread\_attr\_getdetachstate() ..... 453  
 pthread\_attr\_getdetachstate() ..... 1298  
 pthread\_attr\_getinheritsched() ..... 459, 1298  
 pthread\_attr\_getschedparam() ..... 458  
 pthread\_attr\_getschedpolicy() ..... 458, 1298  
 pthread\_attr\_init() ..... 453, 1298, 1299  
 pthread\_attr\_setdetachstate() ..... 453  
 pthread\_attr\_setdetachstate() ..... 1299  
 pthread\_attr\_setinheritsched() ..... 459, 1299  
 pthread\_attr\_setschedparam() ..... 458  
 pthread\_attr\_setschedpolicy() ..... 458, 1299  
 PTHREAD\_CANCEL\_ASYNCHRONOUS ..... 487  
 PTHREAD\_CANCEL\_DEFERRED ..... 487  
 PTHREAD\_CANCEL\_DISABLE ..... 487  
 pthread\_cancel() ..... 487, 1301  
 pthread\_cleanup\_pop() ..... 443  
 pthread\_cleanup\_push() ..... 443  
 pthread\_cond\_broadcast() ..... 471, 1301  
 pthread\_cond\_destroy() ..... 476, 1301  
 pthread\_cond\_init() ..... 476, 1301  
 PTHREAD\_COND\_INITIALIZER ..... 471  
 pthread\_cond\_signal() ..... 471, 1301

pthread\_cond\_timedwait() ..... 471  
 pthread\_cond\_timewait() ..... 1301  
 pthread\_cond\_wait() ..... 471, 1301  
 pthread\_condattr\_destroy() ..... 482  
 pthread\_condattr\_init() ..... 482  
 PTHREAD\_CREATE\_DETACHED ..... 453, 454  
 PTHREAD\_CREATE\_JOINABLE ..... 454  
 pthread\_create() ..... 442, 1297  
 pthread\_detach() ..... 452, 1298  
 pthread\_equal() ..... 450, 1298  
 pthread\_exit() ..... 443, 1297  
 PTHREAD\_EXPLICIT\_SCHED ..... 459  
 pthread\_getschedparam() ..... 455  
 pthread\_getspecific() ..... 493  
 PTHREAD\_INHERIT\_SCHED ..... 459  
 pthread\_join() ..... 444, 1297  
 pthread\_key\_create() ..... 493  
 pthread\_key\_delete() ..... 493  
 pthread\_kill() ..... 499  
 pthread\_mutex\_destroy() ..... 467, 1299  
 PTHREAD\_MUTEX\_ERRORCHECK\_NP ..... 471  
 PTHREAD\_MUTEX\_FAST\_NP ..... 471  
 pthread\_mutex\_init() ..... 467  
 PTHREAD\_MUTEX\_INITIALIZER ..... 463  
 pthread\_mutex\_lock() ..... 463, 1299  
 PTHREAD\_MUTEX\_RECURSIVE\_NP ..... 471  
 pthread\_mutex\_trylock() ..... 463, 1299  
 pthread\_mutex\_unlock() ..... 463, 1299  
 pthread\_mutexattr\_destroy() ..... 470, 1300  
 pthread\_mutexattr\_getkind\_np() ..... 1300  
 pthread\_mutexattr\_gettype() ..... 470, 1300  
 pthread\_mutexattr\_init() ..... 470, 1300  
 pthread\_mutexattr\_setkind\_np() ..... 1300  
 pthread\_mutexattr\_settype() ..... 470, 1300  
 PTHREAD\_ONCE\_INIT ..... 496  
 pthread\_once() ..... 495  
 pthread\_self() ..... 444, 1297  
 pthread\_setcancelstate() ..... 487, 1301  
 pthread\_setcanceltype() ..... 487, 1302  
 pthread\_setschedparam() ..... 455  
 pthread\_setspecific() ..... 493  
 pthread\_sigmask() ..... 499  
 pthread\_testcancel() ..... 487  
 Public-Domain-Korn-Shell ..... 1080  
 Puffer ..... 70, 72  
    *kontrollieren* ..... 115  
 Puffereinstellung  
    *ANSIC* ..... 116  
    *SVR4* ..... 116  
 Pufferüberlauf ..... 1064, 1222  
 putc() ..... 112, 1255  
 putchar() ..... 112, 1255  
 putenv() ..... 260, 1287  
 puts() ..... 112, 1255  
 pwd ..... 1104, 1342  
 pwd.h ..... 221

**Q**

qsort() ..... 1258  
 Quota ..... 520  
 Quotings, Shell ..... 1130

**R**

Race Condition ..... 332, 358, 460, 1224  
 raise() ..... 326, 340, 1293  
 rand() ..... 1257  
 Raw Socket ..... 657  
 rcp ..... 1391  
 read, Shell ..... 1170  
 read() ..... 72, 363, 552, 1273  
 readdir\_r() ..... 128  
 readdir() ..... 126, 1282  
 readlink() ..... 153  
 readv() ..... 100  
 realloc() ..... 1257  
 reboot ..... 1382  
 Record Locking ..... 85, 356  
    *Exclusive Locks* ..... 86  
    *Shared Locks* ..... 86  
 recv() ..... 553, 1303  
 recvfrom() ..... 633, 1304  
 Referenz ..... 1239  
 regcomp() ..... 133  
 regerror() ..... 133  
 regex ..... 133  
 regex\_t ..... 133  
 regexec() ..... 133  
 regfree() ..... 133  
 remove() ..... 117, 1254  
 rename() ..... 117, 1254  
 renice ..... 1351  
 Replikation ..... 692  
 reset ..... 1397  
 Ressourcenlimits ..... 265  
 restore ..... 1370  
 rewind() ..... 114, 1256  
 rewinddir() ..... 129, 1282  
 RFC ..... 541  
 rint() ..... 1269  
 rkhunter ..... 1215

r-Kommandos ..... 1391  
 RLIM\_INFINITY ..... 265  
 RLIMIT\_CORE ..... 266  
 RLIMIT\_CPU ..... 266  
 RLIMIT\_DATA ..... 266  
 RLIMIT\_FSIZE ..... 266  
 RLIMIT\_LOCKS ..... 266  
 RLIMIT\_MEMLOCK ..... 266  
 RLIMIT\_NOFILE ..... 266  
 RLIMIT\_NPROC ..... 266  
 RLIMIT\_RSS ..... 266  
 RLIMIT\_STACK ..... 266  
 rlogin ..... 1391  
 rm ..... 1335  
 rmdir ..... 1342  
 rmdir() ..... 117, 122, 1281  
 Rootkit ..... 1213  
 round() ..... 1269  
 RPC ..... 658  
 RPM ..... 1029  
*benötigte Komponenten* ..... 1031  
*Build-Abschnitt* ..... 1036  
*Einführung* ..... 1029  
*Files-Sektion* ..... 1037  
*Install-Abschnitt* ..... 1036  
*Paket erstellen* ..... 1037  
*Paket installieren* ..... 1040  
*Patches* ..... 1031  
*Präambel* ..... 1034  
*Prep-Abschnitt* ..... 1036  
*Sourcecode* ..... 1031, 1032  
*Specfile* ..... 1031, 1034  
*Verzeichnisse* ..... 1032  
 rsetlimit() ..... 61  
 rsh ..... 1391  
 rsync ..... 1394  
 Runlevel ..... 309  
 rwho ..... 1391

**S**

S\_IRGRP ..... 64, 147, 1285  
 S\_IROTH ..... 64, 147, 1285  
 S\_IRUSR ..... 64, 147, 1285  
 S\_IWGRP ..... 64  
 S\_IWXO ..... 64  
 S\_IRWXU ..... 64  
 S\_ISBLK() ..... 147, 1284  
 S\_ISCHR() ..... 147, 1284  
 S\_ISDIR() ..... 147, 1284  
 S\_ISFIFO ..... 362, 385

S\_ISFIFO() ..... 147, 1284  
 S\_ISGID ..... 64  
 S\_ISLINK() ..... 147, 1284  
 S\_ISREG() ..... 147, 1284  
 S\_ISSOCK() ..... 147, 1284  
 S\_ISUID ..... 64  
 S\_ISVTX ..... 64  
 S\_IWGRP ..... 64, 147, 1285  
 S\_IWOTH ..... 64, 147, 1285  
 S\_IWUSR ..... 64, 147, 1285  
 S\_IXGRP ..... 64, 147, 1285  
 S\_IXOTH ..... 64, 147, 1285  
 S\_IXUSR ..... 64, 147, 1285  
 SA\_NOCLDSTOP ..... 334  
 SA\_NOCLDWAIT ..... 334  
 SA\_NODEFER ..... 334  
 SA\_NOMASK ..... 334  
 SA\_ONESHOT ..... 334  
 SA\_ONSTACK ..... 334  
 SA\_RESETHAND ..... 334  
 SA\_RESTART ..... 334  
 SA\_SIGINFO ..... 334  
 scalb() ..... 1269  
 scandir() ..... 130, 1282  
 scanf() ..... 111, 1254  
 SCHAR\_MAX ..... 1244  
 SCHAR\_MIN ..... 1244  
 SCHED\_FIFO ..... 454  
 SCHED\_OTHER ..... 454  
 SCHED\_RR ..... 454  
 sched\_setpriority() ..... 342  
 sched\_yield() ..... 248  
 Schleife, Shell ..... 1159  
 Schlüsselfelder ..... 702  
 Schlüsselgenerierung ..... 664  
 SCO ..... 30  
 scp ..... 1392  
 sed ..... 1119  
 SEEK\_CUR ..... 75, 114  
 SEEK\_END ..... 75, 114  
 SEEK\_SET ..... 75, 114  
 seekdir() ..... 129, 1282  
 select, Shell ..... 1189  
 select() ..... 96, 342, 599, 1274, 1293  
*Sicherheit* ..... 1237  
 SELinux ..... 517  
 sem\_destroy() ..... 483  
 sem\_getvalue() ..... 483  
 sem\_init() ..... 483  
 sem\_post() ..... 483  
 sem\_trywait() ..... 483

SEM\_UNDO ..... 410  
 sem\_wait() ..... 483  
 Semaphore ..... 353, 402  
*abfragen* ..... 407  
*ändern* ..... 407  
*erstellen* ..... 406  
*Lebenszyklus* ..... 403  
*löschen* ..... 407  
*öffnen* ..... 406  
*Operationen* ..... 409  
*struct sembuf* ..... 410  
*Threads* ..... 483  
*Vergleich mit Sperren* ..... 411  
 semctl() ..... 403, 408, 1295  
 semget() ..... 403, 406, 1295  
 semop() ..... 403, 409, 1296  
 send() ..... 552, 1303  
 Sendmail ..... 374  
 sendto() ..... 633, 1303  
 Sequencing ..... 543  
 set  
*Shell* ..... 1137  
 SETALL ..... 409  
 setbuf() ..... 115, 1254  
 setbuffer() ..... 116  
 setegid() ..... 1288  
 setenv() ..... 262, 1287  
 seteuid() ..... 1288  
 setgid() ..... 270, 1288  
 setgrent() ..... 235  
 Set-group-ID-Bit ..... 64  
 setjmp ..... 1249  
 setjmp() ..... 1249  
 setlinebuf() ..... 116  
 setlocale() ..... 1245  
 setpriority() ..... 247, 280  
 setpwent() ..... 223  
 setregid() ..... 1288  
 setreuid() ..... 1288  
 setrlimit() ..... 265, 1287  
 setsid() ..... 303  
 setsockopt() ..... 562, 625, 651, 1304  
 setspent() ..... 228  
 setterm ..... 1398  
 setuid() ..... 270, 1288  
 Set-user-ID-Bit ..... 64  
 SETVAL ..... 409  
 setvbuf() ..... 115, 1254  
 sh ..... 1079  
 SHA1-Algorithmus ..... 674  
 shadow.h ..... 227

Shared Librarys ..... 1024  
 Shared Memory ..... 354, 424, 435  
*abfragen* ..... 425  
*ändern* ..... 425  
*erstellen* ..... 424, 436  
*löschen* ..... 425  
*öffnen* ..... 424, 436  
*Segment loslösen* ..... 427  
*Segment anbinden* ..... 426  
*shmctl()* ..... 425  
 Shebang-Zeile ..... 1086  
 Shell ..... 1075  
*Array* ..... 1121  
 A-Shell ..... 1080  
*auf Prozess warten* ..... 1201  
*Ausgabe* ..... 1166  
*Auto-Variable* ..... 1128  
 Bash ..... 1080  
 Bourne-Shell ..... 1079  
*case* ..... 1157  
 C-Shell ..... 1079  
*dash* ..... 1080  
*Dateistatus ermitteln* ..... 1151  
*Datenstrom umleiten* ..... 1093  
*Dezimalzahl* ..... 1109  
*Dezimalzahlen vergleichen* ..... 1147  
*Eingabe* ..... 1170  
*einzelnes Zeichen einlesen* ..... 1179  
*elif-Anweisung* ..... 1145  
*else-Alternative* ..... 1145  
*festlegen* ..... 1085  
*for-Schleife* ..... 1159  
*Funktion* ..... 1192  
*if-Anweisung* ..... 1141  
*IFS (Shell-Variable)* ..... 1175  
*Jobverwaltung* ..... 1202  
*Kommandosubstitution* ..... 1132  
*Kommandozeilenargument* ..... 1133  
 Korn-Shell ..... 1079  
*Leerzeichen* ..... 1131  
*logischer Operator* ..... 1154  
*Menü mit select* ..... 1189  
*Quotings* ..... 1130  
*rechnen* ..... 1112  
*Schleife* ..... 1159  
*Signal* ..... 1197  
 tcsh ..... 1080  
*test* ..... 1146  
*Typ* ..... 1079  
*Umgebungsvariable* ..... 1127  
*until-Schleife* ..... 1164

Shell (Forts.)	
<i>Variable</i>	1105
<i>Variableninterpolation</i>	1107
<i>while-Schleife</i>	1163
<i>Zeichenkette</i>	1115
<i>Zeichenketten vergleichen</i>	1149
<i>Zeilenumbruch</i>	1131
<i>Z-Shell</i>	1080
<i>Shell-Programmierung</i>	1075
<i>Shellskript</i>	1076
<i>ausführen</i>	1081
<i>beenden</i>	1091
<i>Datenstrom umleiten</i>	1093
<i>im Hintergrund ausführen</i>	1082
<i>Kommunikation</i>	1205
<i>ohne Subshell</i>	1084
<i>Shebang-Zeile</i>	1086
<i>Subshell</i>	1083
<i>synchronisieren</i>	1207
<i>Variable</i>	1105
<i>shift</i>	
<i>Shell</i>	1136
<i>SHM_LOCK</i>	426
<i>SHM_UNLOCK</i>	426
<i>shmat()</i>	426, 1296
<i>shmctl()</i>	1296
<i>shmdt()</i>	1296
<i>shmget()</i>	424, 436, 1296
<i>SHMMAX</i>	425
<i>SHMMIN</i>	425
<i>SHRT_MAX</i>	1245
<i>SHRT_MIN</i>	1245
<i>shutdown</i>	1382
<i>shutdown()</i>	1304
<i>Sicherheit</i>	1213
<i>Ausführrecht</i>	1221
<i>chroot()</i>	1226
<i>Core Dump</i>	1234
<i>Filedeskriptoren</i>	1231
<i>Logfiles</i>	1218
<i>popen()</i>	1230
<i>Race Condition</i>	1224
<i>select()</i>	1237
<i>Socketdeskriptoren</i>	1231
<i>SQL Injection</i>	1235
<i>SUID-Bit</i>	1219
<i>Superuser</i>	1215
<i>syslog()</i>	1218
<i>system()</i>	1230
<i>temporäre Dateien</i>	1225
<i>Trojaner</i>	1214
<i>Sicherheit (Forts.)</i>	
<i>UmgangsvARIABLEn</i>	1227
<i>Viren</i>	1214
<i>Zugriffsrechte</i>	1219, 1230
<i>sig_atomic_t</i>	338
<i>SIG_BLOCK</i>	343, 499
<i>SIG_SETMASK</i>	343, 499
<i>SIG_UNBLOCK</i>	343, 499
<i>sigaction()</i>	333, 1290
<i>sigaddset()</i>	333, 1289
<i>sigdelset()</i>	333, 1289
<i>sigemptyset()</i>	332, 1289
<i>SIGEMT</i>	329
<i>sigfillset()</i>	332, 1289
<i>sighandler_t</i>	333
<i>SIGIOT</i>	329
<i>sigismember()</i>	1289
<i>SIGKILL</i>	329, 499
<i>signal</i>	1250
<i>Signal, Shell</i>	1197
<i>signal()</i>	1250
<i>Signale</i>	
<i>benutzerdefinierte</i>	326
<i>einrichten</i>	333
<i>erfragen</i>	333
<i>exec-Aufruf</i>	328
<i>fork()</i>	328
<i>Gerätesignale</i>	326
<i>neues Signalkonzept</i>	331
<i>pending signal</i>	326
<i>Prozess suspendieren</i>	344
<i>Prozesse synchronisieren</i>	344
<i>senden</i>	340
<i>SIGABRT</i>	329
<i>SIGALRM</i>	330
<i>SIGBUS</i>	329
<i>SIGCHLD</i>	257, 328, 330, 589
<i>SIGCLD</i>	330
<i>SIGCONT</i>	330, 341
<i>SIGFPE</i>	326, 329
<i>SIGHUP</i>	303, 329
<i>SIGILL</i>	326, 329
<i>SIGINT</i>	303, 329
<i>SIGIO</i>	82, 330
<i>SIGLOST</i>	331
<i>Signalhandler</i>	338
<i>Signalmaske</i>	327, 343
<i>Signalmenge</i>	332
<i>SIGPIPE</i>	331, 361
<i>SIGPOLL</i>	330
<i>SIGPROF</i>	330

<i>Signale (Forts.)</i>	
<i>SIGQUIT</i>	329
<i>SIGSEGV</i>	326, 329
<i>SIGSTOP</i>	327, 330
<i>SIGSYS</i>	329
<i>SIGTERM</i>	329
<i>SIGTRAP</i>	329
<i>SIGTSTP</i>	330
<i>SIGTTIN</i>	330
<i>SIGTOU</i>	330
<i>SIGURG</i>	82, 330
<i>SIGUSR1</i>	331, 345
<i>SIGUSR2</i>	331, 345
<i>SIGVTALRM</i>	330
<i>SIGWINCH</i>	303, 331
<i>SIGXCPU</i>	331
<i>SIGXFSZ</i>	331
<i>Systemsignale</i>	326
<i>Threads</i>	499
<i>Zeitschaltuhr</i>	341
<i>Signalhandler</i>	333
<i>Signalmaske</i>	
<i>ändern</i>	343
<i>erfragen</i>	343
<i>Signalmenge</i>	
<i>hinzufügen</i>	333
<i>initialisieren</i>	332
<i>löschen</i>	333
<i>sigpending()</i>	1290
<i>sigprocmask()</i>	343, 1290
<i>sigset_t</i>	332
<i>SIGSTOP</i>	499
<i>sigsuspend()</i>	344, 1290
<i>sigtimedwait()</i>	499
<i>sigwait()</i>	499
<i>sigwaitinfo()</i>	499
<i>sin()</i>	1248
<i>Single Quotes</i>	1131
<i>sinh()</i>	1249
<i>size_t</i>	68
<i>sleep</i>	1351
<i>sleep()</i>	342, 1293
<i>Smack</i>	517
<i>SMTP</i>	542
<i>SNMP</i>	542
<i>snprintf()</i>	109
<i>SO_BROADCAST</i>	626
<i>SO_DEBUG</i>	626
<i>SO_DONTROUTE</i>	626
<i>SO_ERROR</i>	627
<i>SO_KEEPALIVE</i>	627
<i>SO_LINGER</i>	627
<i>SO_OOBINLINE</i>	627
<i>SO_RCVBUF</i>	627
<i>SO_RCVTIMEO</i>	627
<i>SO_REUSEADDR</i>	627
<i>SO_REUSEPORT</i>	627
<i>SO_SNDBUF</i>	627
<i>SO SNDTIMEO</i>	627
<i>SO_TYPE</i>	627
<i>SO_USELOOPBACK</i>	627
<i>SOCK_DGRAM</i>	546
<i>SOCK_PACKET</i>	546
<i>SOCK_RAW</i>	546
<i>SOCK_RDM</i>	546
<i>SOCK_STREAM</i>	546
<i>Socket</i>	543
<i>Adressfamilie</i>	545
<i>auf Verbindung warten</i>	550
<i>Clientanwendung</i>	553
<i>Daten empfangen</i>	551, 552
<i>Daten senden</i>	551, 552
<i>Kommunikationsmodell</i>	544
<i>lauschen</i>	550
<i>lokales</i>	643
<i>Multicast-</i>	645
<i>Multiplexing</i>	598
<i>Nameserver</i>	571
<i>nichtblockierendes</i>	653
<i>Optionen abfragen</i>	625
<i>Optionen setzen</i>	625
<i>parallele Server</i>	582
<i>Port binden</i>	550
<i>Protokollfamilie</i>	545
<i>Pufferung</i>	580, 582
<i>Raw</i>	657
<i>select()</i>	599
<i>Serveranwendung</i>	558
<i>sockaddr</i>	549
<i>sockaddr_in</i>	548
<i>sockaddr_un</i>	639
<i>Socket-Deskriptor</i>	547
<i>Socket-Typen</i>	546
<i>Standard-E/A-Funktionen</i>	581
<i>Struktur</i>	548
<i>UDP</i>	630
<i>Unix-Domain-</i>	639
<i>Verbindung schließen</i>	553
<i>Verbindungsauflauf (Client)</i>	547
<i>Verbindungswünsche abarbeiten</i>	551
<i>Webserver</i>	609
<i>Socket Credentials</i>	520

socket() ..... 545, 1303  
 Socket-Deskriptor ..... 547  
 socketpair() ..... 643, 1304  
 Sockets ..... 355  
     Threads ..... 620  
 Solaris ..... 26, 31  
 sort ..... 1335  
 sparse ..... 75  
 Spearfishing ..... 1214  
 Speicherklumpen ..... 851  
 Sperrdateien ..... 356  
 split ..... 1336  
 sprintf() ..... 109, 1254  
 SQL ..... 687, 691  
 SQL Injection ..... 1235  
     Optionen ..... 1064  
 SQLite ..... 694  
 sqrt() ..... 1249  
 srand() ..... 1257  
 sscanf() ..... 111, 1254  
 SSH ..... 542  
 ssh ..... 1391  
 ssh-keygen ..... 1393  
 ssize\_t ..... 68  
 Stallman, Richard ..... 31  
 Standardausgabe  
     stdout ..... 59  
     STDOUT\_FILENO ..... 59  
 Standardausgabe umleiten, Shell ..... 1093  
 Standard-E/A-Funktionen ..... 106  
     Socket ..... 581  
 Standardeingabe  
     stdin ..... 59  
     STDIN\_FILENO ..... 59  
 Standardeingabe umleiten, Shell ..... 1096  
 Standardfehlerausgabe  
     stderr ..... 59  
     STDERR\_FILENO ..... 59  
 Standardfehlerausgabe umleiten, Shell ..... 1094  
 Startup-Skripte erstellen ..... 310  
 stat (Struktur) ..... 145  
     st\_atime ..... 158  
     st\_blksize ..... 156  
     st\_blocks ..... 156  
     st\_ctime ..... 158  
     st\_gid ..... 150  
     st\_ino ..... 151  
     st\_mode ..... 146  
     st\_mtime ..... 158  
         explizit verwenden ..... 1204  
     st\_nlink ..... 151  
     st\_size ..... 156  
     st\_uid ..... 150  
 stat() ..... 146, 1284  
 stat-Struktur ..... 1283  
 stdarg ..... 110, 1251  
 stdbool ..... 1267  
 stddef ..... 1252  
 stderr, Shell ..... 1094  
 stdin, Shell ..... 1096  
 stdint ..... 1268  
 stdio ..... 106, 1253  
 stdlib ..... 1257  
 stdout, Shell ..... 1093  
 Steuerterminal ..... 252  
 Sticky-Bit ..... 64  
 STRACE ..... 1062  
 strcat() ..... 50, 1259  
 strchr() ..... 50, 1260  
 strcmp() ..... 1260  
 strcoll() ..... 1261  
 strcpy() ..... 50, 1259  
 strcspn() ..... 1260  
 strdup() ..... 49  
 strdupa() ..... 49  
 Systemcalls  
     STRACE ..... 1062  
 systemd ..... 33  
     /usr/lib/systemd/system/ ..... 316  
     Ersatz für init ..... 316  
     Startskripte ..... 316  
 Systeminformationen ..... 171  
 System-V-IPC ..... 400  
     Gemeinsamkeiten ..... 400  
     ipcrm ..... 406  
     ipcs ..... 406  
     Message Queue ..... 411  
     Semaphore ..... 402  
     Shared Memory ..... 424, 435

sum ..... 1322  
 Superuser ..... 1215  
 SVR4 ..... 45  
 swap ..... 1355  
 swapoff ..... 1366  
 swapon ..... 1366  
 symlink() ..... 153  
 Symmetric Multiprocessing ..... 693  
 sync ..... 1366  
 Synchronisieren, Shellskripts ..... 1207  
 sys/mman ..... 1276  
 sys/socket ..... 1302  
 sys/stat ..... 145, 1283  
 syslog() ..... 298, 1218  
 system() ..... 295, 1230, 1258, 1289  
 Systemcalls ..... 58  
     STRACE ..... 1062  
 tac ..... 1337  
 tail ..... 1337  
     Shell ..... 1188  
 tan() ..... 1248  
 tanh() ..... 1249  
 tar ..... 1374  
 TCP\_KEEPALIVE ..... 629  
 TCP\_MAXRT ..... 629  
 TCP\_MAXSEG ..... 629  
 TCP/IP Aufbau ..... 539  
 tcsh ..... 1080  
 tee ..... 1099, 1338  
 telldir() ..... 129, 1282  
 Telnet ..... 541  
 Temporäre Datei ..... 1225  
     erstellen ..... 117  
 test, Shell ..... 1146  
 tgmath ..... 1270  
 Threading ..... 833

**T**

tac ..... 1337  
 tail ..... 1337  
     Shell ..... 1188  
 tan() ..... 1248  
 tanh() ..... 1249  
 tar ..... 1374  
 TCP\_KEEPALIVE ..... 629  
 TCP\_MAXRT ..... 629  
 TCP\_MAXSEG ..... 629  
 TCP/IP Aufbau ..... 539  
 tcsh ..... 1080  
 tee ..... 1099, 1338  
 telldir() ..... 129, 1282  
 Telnet ..... 541  
 Temporäre Datei ..... 1225  
     erstellen ..... 117  
 test, Shell ..... 1146  
 tgmath ..... 1270  
 Threading ..... 833

Thread-Programmierung ..... 439  
 Threads  
     abbrechen ..... 487  
     Attribute ..... 453  
     Barrier ..... 486  
     Bedingungsvariablen ..... 471  
     beenden ..... 443  
     canceln ..... 487  
     Conditions Variablen (dynamisch) ..... 476  
     Condition-Variablen ..... 471  
     Condition-Variablen (statisch) ..... 471  
     Condition-Variablen-Attribute ..... 482  
     Daemon- ..... 452  
     einmalig ausführen ..... 495  
     erzeugen ..... 442  
     Exit-Handler einrichten ..... 443  
     ID ermitteln ..... 444  
     lösen ..... 452  
     Mutex-Attribute ..... 470  
     Mutexe ..... 462  
     Mutexe (dynamisch) ..... 467  
     Mutexe (statisch) ..... 463  
     Netzwerkprogrammierung ..... 620  
     Prozesse ..... 439  
     Rückgabewert ..... 448  
     RW-Locks ..... 486  
     Scheduling ..... 440, 453  
     Semaphore ..... 483  
     Signale ..... 499  
     Spinlocks ..... 486  
     synchronisieren ..... 459  
     thread-sicher ..... 498  
     Thread-spezifische Daten ..... 492  
     TSD-Daten ..... 492  
     Typen-Casting ..... 447  
     vergleichen ..... 450  
     warten ..... 444  
     Zustände ..... 440

Tilde-Expansion, Shell ..... 1104  
 TIME ..... 1042  
 time ..... 159, 1261, 1353  
 time() ..... 1261  
 tmpfile() ..... 117, 1254  
 tmpnam() ..... 117, 1254  
 tm-Struktur ..... 1262  
 tolower() ..... 1241  
 Toolkit ..... 839  
 top ..... 1353  
 touch ..... 1338  
 toupper() ..... 1241

tr ..... 1338  
*Shell* ..... 1117  
traceroute ..... 1395  
Transaktion ..... 692  
trap, *Shell* ..... 1198  
Trigger ..... 692  
Trojaner ..... 1214  
trunc() ..... 1269  
truncate() ..... 104  
Trusted Computing ..... 1215  
TRY\_AGAIN ..... 574  
tty ..... 1399  
type ..... 1339  
typeset  
  *Shell* ..... 1111

unpack ..... 1374  
unsetenv() ..... 263, 1287  
until-Schleife, *Shell* ..... 1164  
unzip ..... 1379  
uptime ..... 1382  
userdel ..... 1345  
User-ID-Bit ..... 150  
User-Level ..... 58  
usermod ..... 1345  
USHRT\_MAX ..... 1245  
usleep() ..... 342, 1293  
usr ..... 983  
UTF-8 ..... 52, 877, 971  
utime() ..... 159  
uudecode ..... 1388  
uuencode ..... 1388

**U**

UCHAR\_MAX ..... 1244  
UDP ..... 630  
  *Clientanwendung* ..... 632  
  *Serveranwendung* ..... 633  
ufsdump ..... 1370  
ufsrestore ..... 1370  
UID ..... 243  
UID\_MAX ..... 1272  
UINT\_MAX ..... 1245  
ulimit() ..... 61  
ULONG\_MAX ..... 1245  
umask ..... 1339  
umask() ..... 66, 1286  
Umgebungsvariable  
  *Shell* ..... 1127  
Umgebungsvariablen ..... 257, 1227  
  *einzeln abfragen* ..... 259  
  *hinzufügen* ..... 260  
  *löschen* ..... 263  
  *verändern* ..... 260  
umount ..... 1365  
unalias ..... 1402  
uname ..... 1381  
uname() ..... 235  
uncompress ..... 1367  
ungetc() ..... 112, 1255  
uniq ..... 1339  
unistd ..... 1272  
Unix ..... 29  
  *Geschichte* ..... 29  
unix2dos ..... 1340  
Unix-Domain-Sockets ..... 639  
unlink() ..... 117

vasprintf() ..... 110  
vasnprintf() ..... 110  
vasprintf() ..... 110  
versionsort() ..... 130  
Verzeichnis ..... 118  
  *Arbeitsverzeichnis ermitteln* ..... 122  
  *komplett einlesen* ..... 130  
  *lesen* ..... 126  
  *löschen* ..... 122  
  *neu anlegen* ..... 119  
  *öffnen* ..... 124  
  *positionieren* ..... 129  
  *schließen* ..... 126  
  *Verzeichnisbäume durchlaufen* ..... 135  
  *wechseln* ..... 120  
vfprintf() ..... 109, 1255  
View ..... 692  
Views ..... 692  
Viren ..... 1214  
v-node-Tabelle ..... 60  
vprintf() ..... 109, 1255  
vsnprintf() ..... 109  
vsprintf() ..... 109, 1255

**V**

wait, *Shell* ..... 1201  
wait() ..... 257, 284, 1288  
waitpid() ..... 284, 287, 589, 1288  
wall ..... 1396  
Wayland ..... 839  
wc ..... 1340  
wchar ..... 1263  
WCONTINUED ..... 288  
WCOREDUMP() ..... 285  
wctob() ..... 1241  
wctype ..... 1264  
Webserver, Apache ..... 758  
Werkzeuge ..... 983  
whatis ..... 1402  
whereis ..... 1340  
while-Schleife, *Shell* ..... 1163  
who ..... 1346  
whoami ..... 1346  
Wide Characters ..... 52  
WIFEXITED() ..... 285  
WIFSIGNALLED() ..... 285  
WIFSTOPPED() ..... 285  
WNOHANG ..... 288, 594  
WNOWAIT ..... 288  
Wrapper ..... 71  
write ..... 1396  
write() ..... 68, 364, 552, 1273  
writev() ..... 100  
WSTOPSIG() ..... 285  
WTERMSIG() ..... 285  
WUNTRACED ..... 288

zcat ..... 1341  
Zeichen einlesen, *Shell* ..... 1179  
Zeichencodierung ..... 52  
Zeichenkette, *Shell* ..... 1115  
Zeichenketten vergleichen, *Shell* ..... 1149  
Zeichenkodierung ..... 51  
Zeichenweise E/A ..... 112  
Zeilenweise E/A ..... 112  
Zeitmessung  
  *GCOV* ..... 1042  
  *GPROF* ..... 1042  
  *Laufzeit einzelner Codezeilen* ..... 1047  
  *Laufzeit von Funktionen* ..... 1043  
  *Laufzeit von Prozessen* ..... 1042  
  *Profiling* ..... 1042  
  *TIME* ..... 1042  
zgrep ..... 1341  
zip ..... 1379  
zless ..... 1341  
zmore ..... 1341  
zsh ..... 1080  
Z-Shell ..... 1080  
Zugriffsdisziplin ..... 358  
Zugriffsrechte ..... 1219  
  *erfragen* ..... 146

**X**

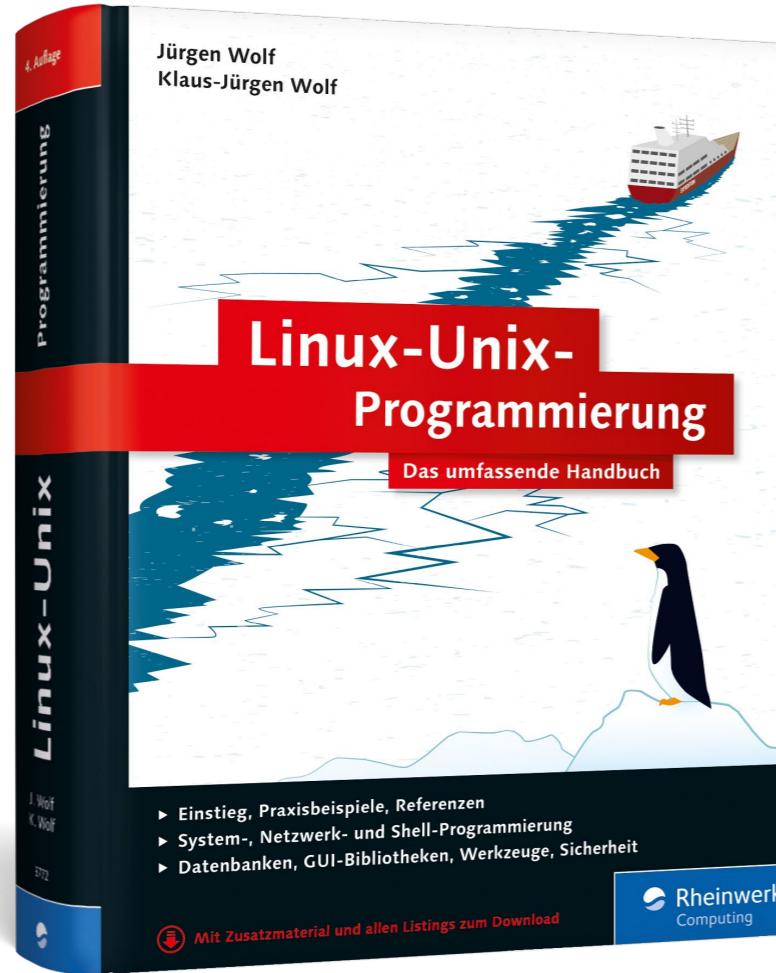
X/OPEN ..... 42  
Xenix ..... 31  
XOPEN\_VERSION ..... 1273

**Y**

YACC ..... 1072

**Z**

zcat ..... 1341  
Zeichen einlesen, *Shell* ..... 1179  
Zeichencodierung ..... 52  
Zeichenkette, *Shell* ..... 1115  
Zeichenketten vergleichen, *Shell* ..... 1149  
Zeichenkodierung ..... 51  
Zeichenweise E/A ..... 112  
Zeilenweise E/A ..... 112  
Zeitmessung  
  *GCOV* ..... 1042  
  *GPROF* ..... 1042  
  *Laufzeit einzelner Codezeilen* ..... 1047  
  *Laufzeit von Funktionen* ..... 1043  
  *Laufzeit von Prozessen* ..... 1042  
  *Profiling* ..... 1042  
  *TIME* ..... 1042  
zgrep ..... 1341  
zip ..... 1379  
zless ..... 1341  
zmore ..... 1341  
zsh ..... 1080  
Z-Shell ..... 1080  
Zugriffsdisziplin ..... 358  
Zugriffsrechte ..... 1219  
  *erfragen* ..... 146



Jürgen Wolf, Klaus-Jürgen Wolf  
**Linux-Unix-Programmierung –  
Das umfassende Handbuch**

1.435 Seiten, gebunden, 4. Auflage 2016  
49,90 Euro, ISBN 978-3-8362-3772-7



[www.rheinwerk-verlag.de/3854](http://www.rheinwerk-verlag.de/3854)



**Jürgen Wolf** ist Softwareentwickler und Autor aus Leidenschaft, er programmiert seit Jahren auf Linux- und Unix-Systemen. Aus seiner Feder stammen vielbeachtete Titel zu C/C++ und zur Linux-Programmierung.



**Klaus-Jürgen Wolf** ist seit 1989 im IT-Umfeld tätig, vorwiegend als Entwickler (C, C++, Python, Java) und als Administrator (Linux/UNIX). Der ausgewiesene Spezialist aus dem Bereich der Linux- und Unix-Sicherheit hat dieses Standardwerk grundlegend aktualisiert und um weitere wertvolle Praxisbeispiele aus zahlreichen Projekten erweitert.

*Wir hoffen sehr, dass Ihnen diese Leseprobe gefallen hat. Sie dürfen sie gerne empfehlen und weitergeben, allerdings nur vollständig mit allen Seiten. Bitte beachten Sie, dass der Funktionsumfang dieser Leseprobe sowie ihre Darstellung von der E-Book-Fassung des vorgestellten Buches abweichen können. Diese Leseprobe ist in all ihren Teilen urheberrechtlich geschützt. Alle Nutzungs- und Verwertungsrechte liegen beim Autor und beim Verlag.*

*Teilen Sie Ihre Leseerfahrung mit uns!*

