Systemnahe Programmierung mit C und Linux

Das umfassende Handbuch

DIE LESEPROBE

Kapitel 4
Zugriff auf Systeminformationen

Das /proc-Pseudo-Filesystem enthält eine Menge Informationen zu Ihrem System. Ob Sie nun Informationen zum aktuellen Prozess, zur Hardware in Ihrem System oder zum Kernel benötigen, all dies finden Sie im /proc-Filesystem.

Da der Kernel den kompletten Rechner verwaltet, besitzt er auch eine Menge Informationen über das System, auf dem er läuft. Diese Informationen werden vom Kernel in Pseudodateien und Pseudoverzeichnissen abgelegt. All diese Dateien liegen im */proc*-Verzeichnis. Auf die Einträge im */proc*-Verzeichnis können Sie wie auf gewöhnliche Dateien zugreifen, meistens allerdings nur lesend. Das */proc*-Pseudo-Dateisystem belegt keinerlei Plattenplatz und befindet sich im Hauptspeicher. Sie können sich gerne einen Überblick über das */proc*-Verzeichnis verschaffen:

```
$ ls -l /proc | less
```

Auf alle diese Informationen können Sie fast ohne weiteres lesend zugreifen. Wollen Sie z. B. die aktuelle Kernel-Version Ihres Systems ermitteln und ausgeben lassen, können Sie auf die Datei */proc/version* zugreifen:

```
$ cat /proc/version
Linux version 2.6.27-7-generic (buildd@palmer)
(gcc version 4.3.2 (Ubuntu 4.3.2-1ubuntu11))
#1 SPM Tue Jan 4 19:33:20 UTC 2009
```

Auf diese Weise erhalten Sie die aktuelle Kernel-Version, das Datum, an dem der Kernel kompiliert wurde, und die Version des zugehörigen Compilers als String zurück. Das letzte Beispiel zeigt z. B. an, dass der Kernel schon sehr alt ist und dringend ein Update benötigt (Sie werden sich wahrscheinlich im Laufe Ihrer Programmier-Laufbahn noch wundern, wie oft Ihre Kunden noch uralte Software installiert haben).

4.1 Informationen aus dem /proc-Verzeichnis herausziehen

Als Programmierer dürfte Sie in erster Linie interessieren, wie Sie mit einem C-Programm an diese Informationen kommen. Als Beispiel soll hier die Speicherauslastung Ihres Systems

dienen. Es soll lediglich ermittelt werden, wie viel RAM insgesamt auf Ihrem System zur Verfügung steht. Diese Informationen erhalten Sie aus */proc/meminfo*:

```
$ cat /proc/meminfo
MemTotal:       514296 kB
MemFree:         10428 kB
Buffers:        122940 kB
Cached:         120112 kB
SwapCached:       1152 kB
Active:         337336 kB
Inactive:       122512 kB
HighTotal:           0 kB
HighFree:            0 kB
LowTotal:       514296 kB
LowFree:         10428 kB
SwapTotal:      409616 kB
SwapFree:       385128 kB
...
```

Uns interessiert der Wert `MemTotal`, der auf dem Raspberry Pi im letzten Beispiel (Raspberry Pi 2B+) 512 MB beträgt. Der einfachste Weg, diesen Wert auszulesen, ist es, */proc/meminfo* mit `fopen()` zu öffnen, in einen Puffer einzulesen und nach der Stringfolge »MemTotal« zu suchen. Anschließend kann der Wert mit `sscanf()` aus dem Puffer in eine dafür vorgesehene Variable übergeben werden. Das C-Programm hierzu sieht wie folgt aus:

```
/* memory.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

long int get_mem_total (void) {
    FILE *fp;
    char buffer[1024];
    size_t bytes_read;
    char *match;
    long mem_tot;

    if((fp = fopen("/proc/meminfo", "r")) == NULL) {
        perror("fopen()");
        exit(EXIT_FAILURE);
    }
```

```
    bytes_read = fread (buffer, 1, sizeof (buffer), fp);
    fclose (fp);
    if (bytes_read == 0 || bytes_read == sizeof (buffer))
        return 0;
    buffer[bytes_read] = '\0';
    /* Suchen nach der Stringfolge "MemTotal" */
    match = strstr (buffer, "MemTotal");
    if (match == NULL) /* nicht gefunden */
        return 0;
    sscanf (match, "MemTotal: %ld", &mem_tot);
    return (mem_tot/1024); /* 1 MB = 1024 KB */
}

int main (void) {
    long memory = get_mem_total();
    if(memory == 0)
        printf("Konnte RAM nicht ermitteln\n");
    else
        printf("Vorhandener Arbeitsspeicher: %ld MB\n" ,memory);
    return EXIT_SUCCESS;
}
```

Listing 4.1 Ermittelt die Größe des vorhandenen Arbeitsspeichers in Bytes.

Vielleicht wundern Sie sich, dass Ihr Pi nur 512 MB freien Speicher zur Verfügung hat, obwohl doch mindestens 2 GB installiert sind. Dies liegt aber daran, dass immer noch RAM für den Framebuffer (also die Grafikhardware) abgezweigt wird. Das heißt, dass beim Pi die GPU keinen eigenen RAM besitzt und deshalb je nach Framebuffer-Größe, die Sie in der Datei *config.txt* angegeben haben, auch eine unterschiedliche Größe freier Speicher vorhanden ist. Außerdem belegt der Kernel immer Speicher, auf den Sie nicht zugreifen können.

Bei PCs ist allerdings oft sehr viel mehr Speicher installiert, auf einem guten Gaming-PC sogar mehr als 4 GB. Dies kann ein Problem werden, wenn versucht wird, die verfügbare RAM-Größe (in Bytes gemessen) in ein `unsigned long int` vom Typ `uint_32` unterzubringen. Dies kann z. B. der Fall sein, wenn Sie ein 32-Bit-Linux benutzen und Ihr Compiler dann auch für `mem_tot` 32 Bits benutzt. Abhilfe kann dann die Verwendung des Datentyps `long long int` sein – wenn dies aber alles nichts hilft, ist ein Update auf ein 64-Bit-Linux nötig.

Sie können nun so oder so ähnlich bei allen Informationen vorgehen, die Sie im */proc*-Verzeichnis finden. Sie sollten allerdings beachten, dass sich der Name des Eintrags im */proc*-Verzeichnis bei neueren Kernel-Versionen ändern kann. Bedenken Sie dies, wenn Sie Ihr Programm auf dem neusten Stand halten müssen.

4.2 Hardware-/Systeminformationen ermitteln

Viele Einträge im */proc*-Dateisystem erlauben Ihnen, Informationen zur Hardware des Systems zu erhalten. Das sind vor allem interessante Informationen für Systemadministratoren, aber auch für den Programmierer, besonders von Spielen (Sie können hier in der Tat eine Menge tunen). Es folgen nun einige häufig verwendete Einträge im Überblick.

Hinweis zur Hardwareprogrammierung

Bei diesem Buch handelt es sich nicht um ein Buch zur Erklärung von Hardware. Sollten Sie keinerlei Kenntnisse vom Innenleben Ihres PC haben, dann wäre zusätzliche Literatur recht sinnvoll. Allerdings müssen wir Sie an dieser Stelle warnen: Das Programmieren auf reiner Hardwareebene (meistens dann auch in Assembler) ist das Schwierigste, was Sie im Endeffekt tun können. Dies gilt selbstverständlich auch für die Programmierung von Betriebssystemen. Wir sagen nicht, dass es unmöglich ist, dass Sie ein Betriebssystem programmieren, aber dies ist etwas, wofür Sie sich schon ein paar Jahre Zeit nehmen sollten.

4.2.1 CPU-Informationen – /proc/cpuinfo

/proc/cpuinfo enthält Informationen zur CPU (also zum Prozessor) oder zu den CPUs, die auf Ihrem System laufen. Dabei werden mehrere Kerne oft als separate CPUs gesehen. Ausgegeben wird hier also die Anzahl der Prozessorkerne. Ist der Wert von `processor` wie im nächsten Beispiel `0`, so kann dies bedeuten, dass es sich um ein Single-Prozessor-System handelt. Es kann aber auch bedeuten, dass Ihr Prozessortreiber nur einen Kern erkennt (dass also etwas nicht stimmt).

Hinweis zu Multiprozessorsystemen

Auf SMP (Multi-CPU-Maschinen) ist der Wert von `processor` nicht direkt `0`, sondern es wird für jeden Prozessor einzeln `processor`, `vendor_id` etc. angezeigt, weshalb »auf 0 gesetzt« nicht ganz korrekt ist.

Weitere Daten finden Sie zu der CPU-Familie, dem Modell, der Frequenz, der Revision und noch einigem mehr. Ein Ausschnitt der Daten auf einem älteren System:

```
$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 2
model name      : Intel(R) Pentium(R) 4 CPU 1.60GHz
stepping        : 4
cpu MHz         : 1594.865
cache size      : 512 KB
```

```
fdiv_bug        : no
hlt_bug         : no
f00f_bug        : no
coma_bug        : no
fpu             : yes
fpu_exception   : yes
cpuid level     : 2
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 sep mtrr pge mca cmov pat
  pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm
bogomips        : 3185.04
```

Wie Sie an dieser Stelle erkennen, wird bei Single-Prozessor-Systemen auch nur eine CPU angezeigt und eben der Eintrag `GenuineIntel`. Bei einer CPU der Core-Reihe (i3/i5/i7) würde dort z. B. `Core i5` stehen, und unter `model name` würde viermal `Intel(R) Core i5` erscheinen, also für jeden Kern ein Eintrag. Allerdings kann die Sache bei 8-Kern-Prozessoren anders aussehen, weil dort oft zwei Kerne zu einer Einheit zusammengefasst werden, um die Arbeit von Game-Engines zu beschleunigen. Deshalb kann es sein, dass beispielsweise für den Ryzen 3000 auch nur 4 Einträge erscheinen (statt 8 oder 16). Für mehr Informationen zur CPU sei die Dokumentation zu `cpuid` des Intel Architecture Software Developer's Manual empfohlen, allerdings sollten Sie darauf achten, immer die aktuellste Version herunterzuladen.

4.2.2 Geräteinformationen in /proc/devices

Verwechseln Sie */proc/devices* nicht mit dem Verzeichnis */dev*, denn dies sind zwei unterschiedliche Paar Schuhe. Während in */dev* die Kommunikationspfade (also die Datenkanäle) der Geräte in Form von Dateien eingeblendet werden, werden in */proc/devices* die Informationen zu allen verfügbaren Geräten (zeichen- und blockorientiert) wie der Festplatte, dem Diskettenlaufwerk oder der seriellen und parallelen Schnittstelle aufgelistet. Das heißt, Sie holen sich die Informationen über ein Gerät aus */proc/devices*, kommunizieren mit dem Gerät aber über */dev*.

4.2.3 Die Speicherauslastung mit /proc/meminfo ermitteln

In dieser Pseudodatei kann der aktuelle Speicherstatus ausgelesen werden. Angezeigt werden der vorhandene und der belegte Speicher (sowohl der physische als auch der Swap-Speicher). Ebenfalls lässt sich ermitteln, wie viel davon für geteilten Speicher (Shared Memory), Puffer und Caches belegt ist. *Swap-Speicher* bedeutet, dass ein Teil des Speichers auf die Festplatte ausgelagert wird, wenn zu wenig RAM vorhanden ist. Allerdings kann man diese Aussage nicht mehr im Allgemeinen machen, weil moderne Betriebssysteme inzwischen auch dann Seiten auslagern, wenn bestimmte Voraussetzungen hierfür vorliegen. Diese im Voraus zu kennen, ist inzwischen fast unmöglich, was viele Hardwareexperten (uns selbst ein-

geschlossen) inzwischen dazu veranlasst, aus Performance-Gründen die Abschaffung des Swap-Speichers zu fordern. Dies ist aber nur unsere ganz persönliche Sichtweise, die auch keinen weiteren Einfluss auf dieses Buch hat.

4.2.4 Weitere Hardwareinformationen in zusammengefasster Form

Es gibt natürlich noch eine Menge mehr Informationen zur Hardware im */proc*-Verzeichnis, die allerdings zum Teil auch von der Konfiguration des Systems abhängen. In Tabelle 4.1 sehen Sie eine kleine Zusammenfassung zu gängigen Einträgen im */proc*-Verzeichnis, die nicht nur auf dem Pi, sondern z. B. auch auf dem PC gültig sind:

Verzeichniseintrag	Bedeutung
/proc/dma	Liste von belegten DMA-Kanälen, die für einen Treiber reserviert sind, und der Name des Treibers. *DMA* (*Direct Memory Access*) ist ein Zugriffsverfahren, mit dem unabhängig von der CPU auf den Hauptspeicher zugegriffen werden kann. DMA wird vor allem von Soundhardware, Grafikhardware, Netzwerk- und USB-Karten und Festplatten benutzt, um eine schnelle Übertragung großer Datenblöcke zu gewährleisten.
/proc/interrupts	Liste der benutzten Interrupts, eventuell mit Anzahl der ausgelösten Interrupts seit Systemstart, Typ des Interrupts und Angabe, welche Module diesen Interrupt verwenden. Interrupts werden von unterschiedlicher Hardware auch unterschiedlich verwendet und sehen z. B. beim Pi anders aus als auf dem PC. Die Programmierung von sogenannten *Interrupt-Handlern* ist nicht trivial und benötigt separate Literatur.
/proc/ioports	Eine Liste aller belegten I/O-Ports für Ein-/Ausgabe-Geräte wie die Festplatte, Ethernet, Soundkarte, Modem, USB etc. (gemappte Hardware-Speicherbereiche befinden sich in */proc/iomem*). I/O-Ports sind Eingabekanäle für Geräte. I/O-Ports finden vor allem auf PCs Verwendung. Dort gibt es sogar spezielle Prozessorbefehle für den Zugriff auf diese Ports.
/proc/stat	Allgemeine Informationen über den Status der Prozessoren. Diese Informationen werden vom Programm *procinfo* verwendet und übersichtlich aufgelistet. Allerdings ist *procinfo* manchmal nicht von Haus aus installiert und muss mit `apt-get -install procinfo` nachinstalliert werden.

Tabelle 4.1 Systeminformationen im /proc-Verzeichnis

Verzeichniseintrag	Bedeutung
/proc/uptime	Anzahl der Sekunden, seitdem das System gestartet ist, und wie lange davon die CPU mit *Nichtstun* (HLT, Leerlauf = engl. *idle*) verbracht hat
/proc/scsi/	Unterverzeichnis mit Informationen zu SCSI-Geräten. Hier können aber auch externe USB-Laufwerke (vor allem beim Pi) erscheinen, die in USB verpackte SCSI-Kommandos zur Kommunikation benutzen.
/proc/scsi/scsi	Sofern Sie eine SCSI-Schnittstelle besitzen (also eine SCSI-Karte), finden Sie hier eine Auflistung aller SCSI-Geräte. Hier erscheinen aber keine externen USB-Geräte, die zur Kommunikation in USB verpackte SCSI-Kommandos benutzen.
/proc/ide/	Unterverzeichnis mit Informationen zu IDE-Geräten
/proc/usb	Unterverzeichnis mit Informationen zu USB-Geräten. Hier können auch externe USB-Laufwerke erscheinen, sogar solche, die eigentlich das SCSI-Protokoll verwenden und deshalb in */proc/scsi* erscheinen müssten (natürlich kann dies dann auch z. B. zu Problemen bei der Erkennung von externen Brennern führen).
/proc/apm	Informationen zum Advanced Power Management (ein Text wie »AC offline« bedeutet, dass ein Notebook im Batteriebetrieb läuft)
/proc/mounts	Liste aller gemounteten Dateisysteme. *Gemountet* bedeutet, dass ein Dateisystem erfolgreich in den Verzeichnisbaum eingehängt wurde und nun verwendet werden kann (siehe dazu auch `man mount`).
/proc/net/	Unterverzeichnis zu Netzwerkinformationen
/proc/loadavg	durchschnittliche Systemauslastung (eine Minute, drei Minuten, fünf Minuten, aktive Prozesse/Anzahl Prozesse und zuletzt benutzte `PID`)

Tabelle 4.1 Systeminformationen im /proc-Verzeichnis (Forts.)

Das folgende Listing soll Ihnen demonstrieren, wie einfach es ist, sich nützliche Informationen zur Hardware ausgeben zu lassen. Das Beispiel stellt eine einfache Schnittstelle dar, die Sie bei Bedarf erweitern können. Damit haben Sie nun schon die Grundlage für einen kleinen Systemmonitor zur Verfügung. Soll Ihr Monitor ein Konsolenprogramm werden, dann müssten Sie die Ausgabe benutzerfreundlich anpassen – ebenfalls dann, wenn Sie nur einzelne Informationen ausgeben wollen. Natürlich können Sie über das Programm auch ein grafisches Frontend ziehen, denn auch die Systemprogramme von KDE oder GNOME machen nichts anderes, als die Daten aus dem */proc*-Filesystem zu lesen und entsprechend auszugeben. Wie Sie GUIs erstellen, werden Sie im weiteren Verlauf auch noch lernen.

```
/* myinfo.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#define BUF 4096 /* anpassen */

enum { CPU, DEV, DMA, INT, IOP, MEM, VERS, SCSI, EXIT };

const char *info[] = {
    "/proc/cpuinfo", "/proc/devices", "/proc/dma",
    "/proc/interrupts", "/proc/ioports", "/proc/meminfo",
    "/proc/version", "/proc/scsi/scsi"
};

void get_info (int inf) {
    FILE *fp;
    char buffer[BUF];
    size_t bytes_read;

    if((fp = fopen(info[inf], "r")) == NULL) {
        perror("fopen()");
        return;
    }
    bytes_read = fread (buffer, 1, sizeof (buffer), fp);
    fclose (fp);
    if (bytes_read == 0 || bytes_read == sizeof (buffer))
        return;
    buffer[bytes_read] = '\0';

    printf("%s",buffer);
    printf("Weiter mit ENTER");
    getchar();
    return;
}

int main (void) {
    int auswahl;
    do {
        printf("Wozu benötigen Sie Informationen?\n\n");
        printf("-%d- Prozessor\n", CPU);
        printf("-%d- Geräte\n", DEV);
        printf("-%d- DMA\n", DMA);
```

```
        printf("-%d- Interrupts\n", INT);
        printf("-%d- I/O-Ports\n", IOP);
        printf("-%d- Speicher\n", MEM);
        printf("-%d- Version\n", VERS);
        printf("-%d- SCSI\n", SCSI);
        printf("-%d- Programmende\n", EXIT);
        printf("\nIhre Auswahl : ");
        do{ scanf("%d",&auswahl); } while(getchar() != '\n');

        switch(auswahl) {
            case CPU : get_info(CPU);   break;
            case DEV : get_info(DEV); break;
            case DMA : get_info(DMA); break;
            case INT : get_info(INT); break;
            case IOP : get_info(IOP); break;
            case MEM : get_info(MEM); break;
            case VERS: get_info(VERS); break;
            case SCSI: get_info(SCSI); break;
            case EXIT: printf("Bye\n"); break;
            default  : printf("Falsche Eingabe?\n");
        }
    } while(auswahl != EXIT);
    return EXIT_SUCCESS;
}
```

Listing 4.2 Zeigt einige Systeminformationen an, die Sie aus einem Menü auswählen können.

Das Programm gibt in der Konsole z. B. folgende Informationen aus (hier ist dies wieder unser altes »SCSI-Schätzchen«, das wir gerne für Testzwecke benutzen):

```
$ ./myinfo
Wozu benötigen Sie Informationen?

-0- Prozessor
-1- Geräte
-2- DMA
-3- Interrupts
-4- I/O-Ports
-5- Speicher
-6- Version
-7- SCSI
-8- Programmende
```

```
Ihre Auswahl : 7
Attached devices:
Host: scsi0 Channel: 00 Id: 00 Lun: 00
  Type:   Direct-Access    ANSI  SCSI revision: 02
Host: scsi2 Channel: 00 Id: 00 Lun: 00
  Vendor: MATSHITA Model: DVD-RAM UJ-850S  Rev: 1.60
  Type:   CD-ROM            ANSI  SCSI revision: 05
Weiter mit ENTER
```

Hinweis zum Löschen des Bildschirms

Wenn Sie den Bildschirm in Ihrer Konsole jedes Mal löschen wollen, wenn das Menü neu aufgebaut wird, dann verwenden Sie am besten noch vor dem ersten `printf()`-Befehl, der das Menü ausgibt, das Kommando **`system("clear")`**, das ein `clear`-Kommando an Ihre Konsole sendet. Weitere Fragen zum Löschen des Bildschirms in einem Terminal beantworten wir in Kapitel 13, »Terminal-E/A und Benutzerschnittstellen für die Konsole«.

4.3 Prozessinformationen auslesen

Linux ist ein System, das auf so ziemlich allen Prozessoren läuft, selbst auf einem alten 6502-System kann Linux installiert werden. Auch auf alten »Retrohandys« lässt sich Linux mittlerweile einrichten, wenn auch eben nicht mit dem Komfort von modernen Smartphones. Deshalb ist es auch nicht verwunderlich, wenn Sie neben Hardwareinformationen auch zahlreiche weitere Informationen zu den auf Ihrem Prozessor ausgeführten Programmen (den sogenannten *Prozessen*) erhalten können. Auch Tools wie `ps` oder `top` nutzen diese Möglichkeit, um Ihnen Informationen zu einem Prozess zu liefern, aber Sie können natürlich auch selbst entscheiden, was Sie sehen wollen. Den Prozessen und ihrer Verwaltung wird noch ein Kapitel gewidmet, deshalb fassen wir uns an dieser Stelle noch etwas kurz. Nun zu den Prozessinformationen: Das */proc*-Verzeichnis enthält für jeden Prozess ein eigenes Unterverzeichnis. Der Name des Unterverzeichnisses entspricht der Prozess-ID. Lassen Sie sich am besten einmal das */proc*-Verzeichnis in der folgenden Weise ausgeben:

```
$ ls -l /proc | less
dr-xr-xr-x    3 root     root     0 2003-11-13 23:46 1
dr-xr-xr-x    3 root     root     0 2003-11-13 23:46 12
dr-xr-xr-x    3 root     root     0 2003-11-13 23:46 1201
dr-xr-xr-x    3 root     root     0 2003-11-13 23:46 1278
dr-xr-xr-x    3 bin      root     0 2003-11-13 23:46 1303
...
dr-xr-xr-x    4 root     root     0 2003-11-13 23:47 tty
```

```
-r--r--r--    1 root     root     0 2003-11-13 23:47 uptime
-r--r--r--    1 root     root     0 2003-11-13 23:47 version
dr-xr-xr-x    3 root     root     0 2003-11-13 23:47 video
```

All diese hier aufgelisteten Verzeichnisse werden als *Prozessverzeichnisse* bezeichnet, da sie sich auf die Prozess-ID (`PID`) beziehen und Informationen zu diesem Prozess enthalten. Eigentümer und Gruppe eines jeden solchen `PID`-Verzeichnisses werden auf die ID des Benutzers, der den Prozess ausführt, gesetzt (eine nachträgliche Änderung über `setuid()` ändert übrigens nur den Eigentümer des Verzeichnisses, nicht den der Dateien). Nach Beendigung eines Prozesses wird der Eintrag im */proc*-Verzeichnis automatisch wieder gelöscht. Während der Ausführung eines Prozesses können Sie jedoch aus dem Verzeichnis die folgenden nützlichen Informationen erhalten:

```
#include <unistd.h>
int main(void) { setuid(25); while(1); }
```

Auf die folgende Weise können Sie Ihr Programm nun im Hintergrund ausführen (mit `./a.out &`):

```
# ./a.out &
[1] 2275
# ls -l /proc/2275/
insgesamt 0
-r--r--r--  1 pinguin  users   0 2004-08-14 02:36   cmdline
-r--------  1 pinguin  users   0 2004-08-14 02:36   environ
lrwxrwxrwx  1 pinguin  users   0 2004-08-14 02:36   exe->./a.out
dr-x------  2 pinguin  users   0 2004-08-14 02:36
fd-rw------ 1 pinguin  users   0 2004-08-14 02:36   mapped_base
-r--r--r--  1 pinguin  users   0 2004-08-14 02:36   maps
-rw-------  1 pinguin  users   0 2004-08-14 02:36   mem
-r--r--r--  1 pinguin  users   0 2004-08-14 02:36   mounts
lrwxrwxrwx  1 pinguin  users   0 2004-08-14 02:36   root -> /
-r--r--r--  1 pinguin  users   0 2004-08-14 02:36   stat
-r--r--r--  1 pinguin  users   0 2004-08-14 02:36   statm
-r--r--r--  1 pinguin  users   0 2004-08-14 02:36   status
```

Gilt auch für `seteuid()` und `setreuid()`.

4.3.1 Das Verzeichnis /proc/$pid/cmdline

Angenommen, Sie finden ein Verzeichnis mit der ID 2167 und wollen wissen, was in der Kommandozeile des Prozesses steht. In diesem Fall fragen Sie `cmdline` wie folgt ab:

```
$ cat /proc/2167/cmdline
anjuta
```

Der Prozess mit der ID 2167 ist also das Programm *Anjuta* bei der Ausführung. Wollen Sie alle Prozesse ausgeben lassen, können Sie dies folgendermaßen machen:

```
# cat /proc/[0-9]*/cmdline
```

Da die Ausgabe allerdings zu wünschen übriglässt, sollten Sie hierfür das Kommando `strings` einsetzen:

```
# strings -f /proc/[0-9]*/cmdline
```

Auf diese Weise bekommen Sie alle zu den Prozess-IDs gehörenden Namen angezeigt. Mit `cmdline` können Sie sich auch eine einzelne Zeile des Prozesses ausgeben lassen, in der der Name (oder auch das Kommando) des Programms mit allen Argumenten enthalten ist.

Falls `strings` bei Ihnen nicht funktioniert, können Sie auch das folgende Kommando verwenden:

```
# grep -Ha "" /proc/[0-9]*/cmdline | tr '\0' " "
```

4.3.2 Das Verzeichnis /proc/$pid/environ

Jeder Prozess hat außer der `PID` auch eine Prozessumgebung. Welche Umgebungsvariablen für welchen Prozess gesetzt sind, können Sie mit `environ` im */proc*-Verzeichnis der Prozess-ID erfragen. Hier soll weiterhin der Prozess mit der ID 2167, die Entwicklungsumgebung *Anjuta*, beobachtet werden:

```
$ strings -f /proc/2167/environ
/proc/2167/environ: LESSKEY=/etc/lesskey.bin
/proc/2167/environ: MANPATH=/usr/local/man:/usr/share/man:/..
/proc/2167/environ: INFODIR=/usr/local/info:/usr/share/...
/proc/2167/environ: NNTPSERVER=news
...
/proc/2167/environ: KDE_FULL_SESSION=true
/proc/2167/environ: KDE_MULTIHEAD=false
/proc/2167/environ: SESSION_MANAGER=local/linux:/tmp/.IC...
/proc/2167/environ: KDE_STARTUP_ENV=linux;1068759934;528706;1890
```

Wenn es auch hier wieder nicht mit `strings` klappen sollte, dann sollten Sie es mit Folgendem probieren:

```
# tr '\0' '\n' < /proc/2167/environ
```

4.3.3 Das Verzeichnis /proc/self

Wollen Sie die aktuelle Prozess-ID Ihres eigenen Programms ermitteln, so können Sie den Eintrag */proc/self* auswerten. Bei diesem Eintrag handelt es sich um einen symbolischen Link des aktuell laufenden Prozesses. Das aktuelle Programm sollten Sie aber nicht mit der Shell verwechseln (versuchen Sie es selbst mit `ls -dl /proc/self` und `ls -dl /proc/$$`). Diesen symbolischen Link können Sie mit der Funktion `readlink()` auslesen. Es folgt ein Listing, das demonstriert, wie Sie die Prozess-ID des aktuell laufenden Programms ermitteln können:

```
/* my_getpid.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int main (void) {
    char buf[64];
    int pid;

    memset(buf, '\0', sizeof(buf));
    readlink("/proc/self", buf, sizeof(buf) - 1);
    sscanf(buf, "%d", &pid);
    printf("Meine Prozess-ID lautet: %d\n", pid);
    return EXIT_SUCCESS;
}
```

Listing 4.3 Demonstriert die Funktion readlink(), mit der z. B. die PID des laufenden Programmes ermittelt werden kann. Eine Alternative ist getpid().

Das Programm bei seiner Ausführung:

```
$ gcc my_getpid.c -o my_getpid
$ ./my_getpid
Meine Prozess-ID lautet: 2256
```

Hinweis zu getpid()

Das Beispiel stellt natürlich nur eine Demonstration dar. In der Praxis werden Sie auf die Funktion `getpid()` für die Ermittlung der Prozess-ID des laufenden Programms zurückgreifen.

4.3.4 Das Verzeichnis /proc/$pid/fd/

Ein weiterer Eintrag in */proc* ist *fd*, ein Unterverzeichnis, das alle Einträge von Filedeskriptoren enthält, die ein Prozess geöffnet hat. Jeder Eintrag ist hier ein symbolischer Link, der auf eine bestimmte Datei lesend oder schreibend zurückgreift. Natürlich sind hierin auch immer die Standard-Filedeskriptoren (`0`, `1`, `2`) enthalten (es sei denn, sie wurden geschlossen, was viele Dämonen [Hintergrundprozesse] tun). Öffnen Sie nun eine Konsole, und geben Sie das Kommando `ps` ein:

```
$ ps
 PID TTY          TIME CMD
 1988 pts/1    00:00:00 bash
 2827 pts/1    00:00:00 ps
```

Wollen Sie jetzt wissen, welche Filedeskriptoren in der Bash zurzeit geöffnet sind, können Sie die folgende Zeile verwenden:

```
$ ls -l /proc/1988/fd/
insgesamt 0
lrwx------ 1 pinguin   users   64 2003-11-14 00:50 0->/dev/pts/1
lrwx------ 1 pinguin   users   64 2003-11-14 00:50 1->/dev/pts/1
lrwx------ 1 pinguin   users   64 2003-11-14 00:50 2->/dev/pts/1
```

Sie erkennen die drei Standard-Filedeskriptoren als symbolischen Link auf dem Pseudoterminal *pts/1* wieder. Sie können jetzt ohne weiteres auf diese Filedeskriptoren zugreifen:

```
$ echo "Ein Beweis gefällig?" > /proc/1988/fd/1
Ein Beweis gefällig?
```

Hier wurde die Standardausgabe von `echo` auf die eigentliche Standardausgabe des Pseudoterminals umgeleitet.

4.3.5 Das Verzeichnis /proc/$pid/statm

Informationen zur Speicherbelegung eines Prozesses werden in `statm` hinterlegt. Wollen Sie z. B. die Speicherbelegung der Entwicklungsumgebung *Anjuta* ermitteln, gehen Sie folgendermaßen vor:

```
$ cat /proc/2167/statm
2671 2670 1678 302 0 2368 992
```

Sie bekommen dabei sieben verschiedene Zahlen zurückgegeben. Die Bedeutung der einzelnen Zahlen von links nach rechts ist die folgende:

- gesamte Programmgröße in Kilobyte
- Größe von zugeteilten Speicherteilen in Kilobyte (sogenannte Seiten)

- Anzahl von »Shared Pages«
- Anzahl der vom Programmcode benutzten Seiten
- Anzahl der vom Stack/Daten benutzten Seiten
- Anzahl der von dem Bibliotheksprogrammcode benutzten Seiten
- Anzahl von unsauberen Seiten (also Seiten, die noch nicht wieder freigegeben wurden)

Es gibt noch weitere nützliche Informationen zu den Prozessen, die Sie in Tabelle 4.2 aufgelistet vorfinden. `$pid` ersetzen Sie bitte durch eine echte, gültige Prozess-ID.

Verzeichniseintrag	Bedeutung
/proc/$pid/status	Darin finden Sie die formatierten Informationen, die Sie ebenfalls mit */proc/$pid/stat* (allerdings nicht formatiert) ermitteln können. Dies sind Informationen wie die Prozess-ID, die reale und effektive User- und Group-ID, Speicherverbrauch oder Bitmasken.
/proc/$pid/stat	Siehe */proc/$pid/status*. Dies ist eine etwas kompaktere Datei, die sich besser als *status* mit `sscanf()` oder `fscanf()` verwenden lässt.
/proc/$pid/cwd	ein symbolischer Link zum aktuellen Arbeitsverzeichnis des Prozesses
/proc/$pid/exe	ein Verweis auf die ausführbare Programmdatei
/proc/$pid/root	ein Link zum Root-Verzeichnis, das als Wurzelverzeichnis für den Prozess gilt (siehe `chroot()`)
/proc/$pid/maps	Enthält Speicher-Mappings zu den verschiedenen laufenden Dateien und Bibliotheken, die mit diesem Prozess zusammenhängen. Die Datei kann sehr lang werden, wenn ein umfangreicher Prozess ausgeführt wird.

Tabelle 4.2 Prozessinformationen in den individuellen PID-Verzeichnissen

4.4 Kernel-Informationen in /proc

Viele Einträge im */proc*-Verzeichnis geben auch Informationen zum gerade laufenden Kernel aus. Der Ort der Kernel-Informationen ist unterschiedlich, viele liegen im */proc*-Verzeichnis selbst und weitere in den Unterverzeichnissen wie */proc/sys* oder */proc/sys/kernel*.

Hierzu einige häufig benötigte Informationen, die in viele Anwendungen integriert sind:

```
$ cat /proc/sys/kernel/ostype
Linux (Debian)
$ cat /proc/sys/kernel/osrelease
4.6.28-arm-eabi
```

```
$ cat /proc/sys/kernel/version
#1 SMP Tue Jan 4 19:33:20 UTC 2023
$ cat /proc/sys/kernel/ctrl-alt-del
0
```

Zuerst wird abgefragt, was für ein Betriebssystem genau hier läuft (`ostype`), dann die Version des Kernels (`osrelease`) und wann dieser Kernel kompiliert wurde (`version`). Bei der letzten Abfrage wird überprüft, ob `ctrl-alt-del` gesetzt ist. Damit können Sie beeinflussen, ob `init` bei der Tastenkombination Strg+Alt+Entf eine Aktion ausführen soll (dies ist hier nicht der Fall, da der Raspberry Pi [ARM-Kernel 4.6.28-arm-eabi] diese Funktion nicht benutzt). Steht `ctrl-alt-del` auf `1`, wird bei diesem Tastendruck sofort ein Reboot angeordnet (eigentlich nur bei PC-Linuxen). Steht `ctrl-alt-del` allerdings auf `0`, wird `init` dazu angehalten, das System ordentlich herunterzufahren (funktioniert allerdings nicht wirklich korrekt auf dem Pi). Wollen Sie diesen Parameter verändern, müssen Sie sich als Superuser einloggen. Dazu können Sie `su` (bzw. `sudo`) wie folgt benutzen:

```
$ su
Password:********
# echo 1 >> /proc/sys/kernel/ctrl-alt-del
# exit
exit
$ cat /proc/sys/kernel/ctrl-alt-del
1
```

Ein weiteres Beispiel: Sie haben eine CD-ROM eingelegt und gemountet. Nachdem Sie die Daten von der CD-ROM gelesen haben, werfen Sie die CD-ROM ordnungsgemäß mit `umount` wieder aus und fahren anschließend das System herunter. Jetzt benötigen Sie aber die CD-ROM für einen Bekannten. Somit müssen Sie den PC leider nochmals hochfahren. Wäre doch nett, wenn beim Aushängen der CD-ROM mit `umount` die CD ausgeworfen wird, oder? Für solche Fälle ist der Eintrag */proc/sys/dev/cdrom/autoeject* zuständig. Der Wert ist mit der Voreinstellung `0` versehen. Sie können nun folgendermaßen vorgehen, um den Wert auf `1` zu setzen (funktioniert leider nicht immer mit modernen USB-Laufwerken, die in */dev/usb* erscheinen):

```
$ mount /media/cdrom
$ cat /proc/sys/dev/cdrom/autoeject
0
$ su
Password:********
# echo 1 >> /proc/sys/dev/cdrom/autoeject
# exit
exit
```

```
$ cat /proc/sys/dev/cdrom/autoeject
1
$ umount /media/cdrom
```

Beim Aushängen des CD-ROM-Laufwerkes müsste jetzt die CD ausgeworfen werden. Dieses nette Feature zeigt allerdings schon recht früh seine Schattenseiten, z. B. wenn die Hardwareerkennung oder ein Installationsprogramm das Gerät mehrmals öffnet und schließt. Manche Laufwerke können dadurch sogar Schaden nehmen.

Viele Kernel-Parameter wurden übrigens bewusst so implementiert, dass sie den Bedürfnissen der Anwender angepasst werden können, um ein höchstes Maß an Flexibilität zu erreichen. Sie können sich also im Endeffekt an dieser Stelle nicht das System zerschießen. Da hier in der Regel auch Superuser-Rechte benötigt werden, sind diese Einstellungen meistens auch für den Systemadministrator vorgesehen. Daher ergibt es relativ wenig Sinn, wenn Sie in Ihrer Anwendung, die Sie schreiben, versuchen, die Kernel-Parameter zu verändern. Dies würde bedeuten, dass Ihre Anwendung im Superuser-Modus laufen müsste, und das sollten Sie sowieso nur bei Installationsprogrammen zulassen. Bedenken Sie, dass das auch nicht immer möglich ist und auch nicht immer möglich sein sollte.

Für administrative Zwecke soll hierfür jedoch nun eine solche Anwendung erstellt werden. Aber wie schon erwähnt, Sie benötigen Superuser-Rechte (oft auch *root-Rechte* genannt) für dieses Programm. Versucht hingegen ein normaler Anwender, etwas zu verändern, bekommt er die Meldung »permission denied« zurückgegeben. Es folgt das Beispiel mit anschließender Erläuterung.

```
/* kernelinfo.c */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd. h.>
#include <string.h>
#define BUF 4O96

enum { EJECT, FILE_MAX, SHARED_MAX  };
enum { CDINFO, OS, RELEASE, VERSION };

static const char *sys[] = {
  /* bei umount CD auswerfen */
  "/proc/sys/dev/cdrom/autoeject",
  /* max. Anzahl geöffneter Dateien pro Prozess */
  "/proc/sys/fs/file-max",
  /* max. Größe geteilter Speicher (Shared Memory) */
```

```
  "/proc/sys/kernel/shmmax"
};

static const char *info[] = {
  "/proc/sys/dev/cdrom/info",   /* Infos zur CD-ROM        */
  "/proc/sys/kernel/ostype",    /* Welches Betriebssystem? */
  "/proc/sys/kernel/osrelease"  /* Kernel-Version          */
  "/proc/sys/kernel/version"    /* Kernel von wann?        */
};

char *get_info (char *inf) {
    FILE *fp;
    char buffer[BUF];
    size_t bytes_read;

    fp = fopen (inf, "r");
    if (fp == NULL) {
        perror("fopen()");
        return NULL;   /* Fehler beim Öffnen */
    }
    bytes_read = fread (buffer, 1, sizeof (buffer), fp);
    fclose (fp);
    if (bytes_read == 0 || bytes_read == sizeof (buffer))
        return NULL;
    buffer[bytes_read] = '\0';
    return buffer;
}


void set_sys (const char *sys, unsigned long set) {
    FILE *fp;
    char buf[32];

    fp = fopen (sys, "w");
    if (fp == NULL) {
        perror ("fopen()");
        printf ("Weiter mit ENTER\n");
        getchar ();
        return;
    }
    sprintf(buf, "%ld", set);
    fprintf (fp, "%s", buf);
    fclose (fp);
```

```
    return;
}

int main (int argc, char **argv) {
    int auswahl;
    unsigned int file_max;
    unsigned long shared_max;

    do {
        printf ("Aktueller Zustand\n");
        printf ("Betriebssystem : %s", get_info (info[OS]));
        printf ("Kernel-Version  : %s",
            get_info (info[RELEASE]));
        printf ("Datum           : %s",
            get_info (info[VERSION]));
        printf ("-----------------------------------\n");
        printf ("Verändern können Sie Folgendes ...\n");
        printf ("-0- Bei \"umount\"  CD auswerfen"
            " Aktuell: %s", get_info (sys[EJECT]));
        printf ("-1- Max. Anzahl geöffneter Dateien pro"
            " Prozess    Aktuell: %s",
               get_info (sys[FILE_MAX]));
        printf ("-2- Max. Größe des geteilten Speichers"
            " (KB)     Aktuell %s",
               get_info (sys[SHARED_MAX]));
        printf ("-----------------------------------\n");
        printf ("Informationen bekommen Sie zu ...\n");
        printf ("-3- CD-ROM\n");
        printf ("-----------------------------------\n");
        printf ("-4- ENDE\n");
        printf ("Ihre Auswahl bitte (0-4) : ");

        do { scanf ("%d", &auswahl); } while (getchar ()!='\n');
        switch (auswahl) {
            case EJECT:
                if (strncmp ("0", get_info (sys[EJECT]),1) == 0)
                    set_sys (sys[EJECT], 1); /* setzen */
                else /* zurücksetzen */
                    set_sys (sys[EJECT], 0);
                break;
                case FILE_MAX:
                    printf ("Welcher Wert soll gesetzt werden: ");
                    do{ scanf("%d", &file_max); } while (getchar()!='\n');
```

```
                set_sys (sys[FILE_MAX], file_max);
            break;
            case SHARED_MAX:
                printf ("Welcher Wert soll gesetzt werden: ");
                do { scanf("%ld", &shared_max); }
                while (getchar()!='\n');
                set_sys (sys[SHARED_MAX], shared_max);
            break;
            case 3:
                printf ("%s", get_info (info[CDINFO]));
                printf ("Weiter mit ENTER\n");
                getchar();
            break;
            case 4:
                printf("Programm wird beendet\n");
            break;
            default:
                printf ("Unbekannte Eingabe\n");
        }
    } while (auswahl != 4);
    return EXIT_SUCCESS;
}
```

Listing 4.4 Mit diesem Listing können einige (unkritische) Kernel-Bootparameter geändert werden.

Das Programm gibt bei der Ausführung (auf dem PC) Folgendes aus:

```
$ gcc -o kernelinf kernelinf.c
$ ./kernelinf
Aktueller Zustand
Betriebssystem : Linux (Debian 64 Bit)
Kernelversion  : 4.6.28-generic
Datum          : #1 SMP Tue Jun 8 19:33:20 UTC 2022
-----------------------------------
Verändern können Sie Folgendes ...
-0- Bei "umount"  CD auswerfen Aktuell: 0
-1- Max. Anzahl geöffneter Dateien pro Prozess  Aktuell: 49594
-2- Max. Größe des geteilten Speichers (KB)    Aktuell 33554432
-----------------------------------
Informationen bekommen Sie zu ...
-3- CD-ROM
-----------------------------------
-4- ENDE
Ihre Auswahl bitte (0-4) :
```

Neben dem Auswerfen der CD-ROM bei `umount` können hier noch die maximale Anzahl gleichzeitig geöffneter Dateien pro Prozess und die maximale Größe des geteilten Speichers (Shared Memory) verändert werden. Diese Veränderungen können aber wie bereits erwähnt nur mit speziellen Rechten vorgenommen werden. Informationen zum CD-ROM-Laufwerk (was es alles kann, z. B. lesen, schreiben, Blöcke einer DVD+R/W löschen, oder ob es überhaupt ein DVD-Laufwerk ist) können allerdings wieder für alle Anwender aufgelistet werden. Eine Ausnahme sind allerdings Blu-Ray-Laufwerke, denn diese erscheinen oft entweder gar nicht oder aber als normale DVD-(+R/W-)Laufwerke. Dies liegt aber einfach daran, dass es unter Linux manchmal nicht für alle Geräte aktuelle Treiber gibt.

Bitte beachten Sie, dass diese Art von Veränderungen der Kernel-Parameter nur für den laufenden Betrieb gültig sind. Sobald Sie das System herunterfahren und wieder starten, werden die Werte wieder auf ihren Ursprungszustand eingestellt. Wollen Sie eine dauerhafte Veränderung bewirken, müssen Sie sich die Datei *sysctl.conf* im Verzeichnis */etc* vornehmen. Diese Datei können Sie auch mit dem gleichnamigen Kommando `sysctl` verändern. Wollen Sie z. B. mit `sysctl` die `autoeject`-Variable verändern, gehen Sie wie folgt vor (*root*-Rechte):

```
# sysctl -w dev.cdrom.autoeject=0
dev.cdrom.autoeject = 0
# sysctl -w dev.cdrom.autoeject=1
dev.cdrom.autoeject = 1
```

Auf die Angaben von */proc/sys* können Sie dabei verzichten, und anstelle eines Slashs wird hier ein Punkt verwendet.

Wollen Sie wissen, mit welchem Befehl der Kernel beim Booten gestartet wurde, können Sie ihn mit `cmdline` erfragen. Hier können Sie wertvolle Informationen erhalten, z. B. mittels der folgenden Zeile:

```
$ cat /proc/cmdline
root=/dev/hda6 vga=0x0314 hdc=ide-scsi hdclun=0 splash=silent
```

Der Eintrag `vga=0x0314` bedeutet beispielsweise, dass Ihre Grafikkarte in einem VGA-Modus gestartet wird, der Parameter `0x0314` wird in diesem Fall an eine Initialisierungsroutine des BIOS übergeben, um den Framebuffer (also den Speicher, in den Ihre Grafikdaten geschrieben werden) einzurichten. Meistens bedeutet ein solcher Eintrag, dass entweder Ihr Linux veraltet ist oder aber Ihre Grafikkarte keine Grafikbeschleunigung verwendet (dies kann vor allem Ihren Desktop stark ausbremsen). Bei neueren Linux-Versionen fehlt der Eintrag `vga=…` dagegen entweder vollständig, oder es wird stattdessen ein Eintrag wie `gpu=…` benutzt (dieser wird dann später an die `modprobe`-Routine übergeben, um die Grafikhardware zu initialisieren). Allerdings gilt dies nicht für den Raspberry Pi, denn er verwaltet die Grafikhardware mittels der Datei *config.txt*.

Auch die folgende Zeile sollte Ihnen zu denken geben:

```
auto BOOT_IMAGE=2.6.4-HX ro root=301
```

Hier werden ein sehr altes Kernel-Image (2.6.4) verwendet und ein Parameter (`BOOT_IMAGE`), der auf den Bootloader LILO hinweist. LILO ist inzwischen veraltet und sollte durch den GRUB-Bootloader ersetzt werden (im Endeffekt sollten Sie das oben genannte System möglichst vollständig ersetzen). An dieser Stelle gehen wir übrigens nicht grundlos auf ältere Systeme ein, denn auch hier kommen Sie früher oder später mit stark veralteten Systemen in Berührung. Dies hat mehrere Gründe: Zum einen sind einige Anwender einfach »update-faul« und ändern nichts, solange nicht sehr starke Probleme auftreten. Zum anderen wissen sie auch oft einfach nicht, wie man Updates macht, selbst wenn sie Linux regelmäßig benutzen.

Zu den verschiedenen Bootoptionen sei hier noch der Verweis auf die *bootparam*-Manual-Page gegeben. Es gibt noch weitaus mehr zu den Kernel-Informationen zu sagen, aber das würde den Rahmen des Buches bei weitem sprengen und am Thema der Linux-Programmierung vorbeigehen.

4.4.1 Das Verzeichnis /proc/locks

Im Verzeichnis */proc/locks* werden alle Dateien angezeigt, die im Augenblick vom Kernel gesperrt werden (vergleiche Kapitel 2, Funktion `lockf()`). Diese Ausgabe enthält interne Kernel-Debugging-Daten und kann daher logischerweise je nach System stark variieren. Ein Beispiel wäre:

```
1: FLOCK ADVISORY WRITE 807 03:05:308731 0 EOF c2ac0 c0248 c2a220
2: POSIX ADVISORY WRITE 708 03:05:308720 0 EOF c2a1c c2ac4 c02548
```

In der ersten Spalte besitzt jeder Lock eine einmalige Zahl gefolgt vom Typ des Locks. Mögliche Ausgaben sind `FLOCK` (meist bei älteren UNIX-Datei-Locks mit dem Systemaufruf `flock()`) und `POSIX` (bei neueren POSIX-Locks unter Linux mit dem Systemaufruf `lockf()`, den Sie hier auch stets verwenden sollten).

Der Wert `ADVISORY` in der dritten Spalte bedeutet, dass ein weiterer Datenzugriff für andere Benutzer möglich ist, aber keine weiteren Lock-Versuche mehr erlaubt sind. Ein anderer möglicher Wert ist `MANDATORY`, was bedeutet, dass keine weiteren Datenzugriffe mehr möglich sind, solange der Lock vorhanden ist. In der vierten Spalte befinden sich die Lese- oder Schreibrechte (`READ`, `WRITE`) des Eigentümers. Die fünfte Spalte enthält die `PID` des Lock-Eigentümers. Die ID der gelockten Datei finden Sie in der sechsten Spalte mit folgendem Format:

```
MAJOR-DEVICE:MINOR-DEVICE:INODE-NUMBER
```

Die Spalten sechs und sieben zeigen Anfang und Ende der gelockten Region. Im Beispiel gilt die Sperre vom Anfang 0 bis zum Ende (EOF) der Datei. Die letzten Spalten zeigen auf Kernel-interne Datenstrukturen für spezielle Debugging-Funktionen.

4.4.2 Das Verzeichnis /proc/modules

In */proc/modules* finden Sie eine Liste von allen Modulen, die vom System geladen wurden. Auch hierbei hängt die Ausgabe von den Einstellungen des Systems ab. Diese Ausgabe kann wie folgt aussehen:

```
$ cat /proc/modules
isofs 40100 1 - Live 0xe0c36000
udf 88356 0 - Live 0xe0c94000
crc_itu_t 10112 1 udf, Live 0xe0bff000
btusb 19736 0 - Live 0xe0c23000
nls_iso8859_1 12032 1 - Live 0xe0c1f000
nls_cp437 13696 1 - Live 0xe0c09000
vfat 18816 1 - Live 0xe0c03000
fat 57376 1 vfat, Live 0xe0c0f000
ipv6 263972 15 - Live 0xe0c52000
gpu_nvidia_geforce 804020 4 - Live 0xe0000000
```

In der ersten Spalte befindet sich jeweils der Name des geladenen Moduls, gefolgt von der Speichergröße in Bytes. In der dritten Spalte wird angezeigt, wie oft das Modul gerade benutzt wird (*usage count*). In der letzten Spalte finden Sie die Module, die benötigt werden, damit andere Module funktionieren. Zum Beispiel benötigt das Modul *cdrom* das Modul *ide-cd*, um ordentlich ausgeführt zu werden. `autoclean` gibt an, ob sich das Modul nach einer gewissen Zeit selbst deaktiviert, `unused`, dass es nicht benutzt wird. Die Speicheradresse, an der das Modul geladen wird, befindet sich am Ende des Eintrags. Auf diese Weise sehen Sie auch, wie eine erfolgreich aktivierte Grafikbeschleunigung aussehen muss: Einer der Kernel-Bootparameter muss den Namen des Grafikprozessors (hier `gpu=nvidia_geforce`) enthalten, und `modprobe` muss in */proc/modules* auch einen Eintrag erzeugen, der den Treiber (in diesem Fall `gpu_nvidia_geforce`) enthält. Allerdings haben Geforce-Treiber eine Eigenheit, die man kennen sollte: `804020` ist nämlich hier die Revision des verwendeten Nvidia-Chips, `4` ist die Version (also ist es eine Geforce 4). Hier werden also zwei Parameter (nämlich der Speicherverbrauch und die Anzahl der geladenen Module) für andere Zwecke missbraucht. `Live` bedeutet natürlich auch hier, dass der Treiber erfolgreich geladen wurde (also quasi lebt), und `0xe0000000` ist die Startadresse des Framebuffer-Bereichs.

Auch hier verhalten sich die Dinge auf dem Raspberry Pi etwas anders, denn dort müssen Sie die Grafikbeschleunigung in der Datei *config.txt* im Verzeichnis */boot* aktivieren. Wie dies geht, entnehmen Sie am besten dem Raspberry-Pi-Forum, denn die Datei *config.txt* wird dauernd überarbeitet und durch neue Versionen ersetzt.

4.5 Verschiedene Dateisysteme unter Linux verwalten

Weitere Informationen, die Sie im */proc*-Verzeichnis finden, sind die Informationen über verschiedene Dateisysteme, vor allem darüber, welche eingehängt (gemountet) wurden. Das Verzeichnis */proc/filesystems* listet alle Dateisysteme auf, die dem Kernel bekannt sind und mit denen er auch arbeiten kann. Einige davon sind intern und können nicht extra gemountet werden, auch wenn sie hier aufgelistet werden. Dazu zählt z. B. *pipefs*, aber auch *proc*, wobei diese Auflistung nicht vollständig ist (es gibt noch viele Dateisysteme, die nachgeladen werden können und gerade nicht aktiv sind). Andere Dateisysteme wiederum sind nur statisch gelinkt und werden erst bei Bedarf aktiviert.

Für den Umgang mit Dateisystemen unter Linux sollten Sie jedoch wissen, dass Linux keine Laufwerkbuchstaben verwendet, anders als z. B. Windows (dies kann für einen Umsteiger verwirrend sein). Stattdessen werden ganze Laufwerke (eben durch das `mount`-Kommando) in einen normalen Verzeichnispfad eingehängt. Natürlich muss es zu diesem Zweck eine oberste Ebene geben, und dies ist die sogenannte *Root-Partition /*. Alle weiteren Laufwerke werden in Verzeichnisse unterhalb / eingehängt, auch externe Festplatten und SD-Karten. Natürlich müssen Sie diese Verzeichnisse zu diesem Zweck vorher anlegen, aber wenn ein bestimmtes Verzeichnis (z. B. */media*) einmal angelegt wurde, können Sie etwa eine SD-Karte daran binden. Allerdings erscheint die SD-Karte selbst als neuer Eintrag in */dev*, nachdem sie eingesteckt wurde, und es ist nicht immer intuitiv erfassbar, welchen Namen sie bekommt. Auf dem Pi wird diese Erkennung z. B. im Hintergrund ausgeführt, und wenn Sie eine SD-Karte in ein externes Lesegerät einstecken, erscheint diese sofort in */media*. Auf PCs muss dies aber nicht der Fall sein, und Sie müssen dann unter Umständen Ihre SD-Karte von Hand mounten. Wechseln Sie dazu in das Verzeichnis */dev*, und geben Sie `ls` ein. Nun stecken Sie die SD-Karte ein und geben wieder `ls` ein. Was hat sich verändert? Es kann beispielsweise sein, dass eine SD-Karte, die Sie zuvor unter Windows mit FAT32 formatiert haben, nun als */dev/hdc1* erscheint. Um nun Ihre SD-Karte an das Verzeichnis */media* zu binden, benutzen Sie den folgenden `mount`-Befehl:

```
sudo mount /dev/hdc1 -t vfat /media
```

oder auch (bei sehr großen Karten):

```
sudo mount /dev/hdc1 -t exfat /media
```

Wie gesagt, auf dem Pi ist dieser Schritt meist nicht nötig.

4.5.1 Das Verzeichnis /proc/mounts

Was alles gerade eingehängt ist, finden Sie in */proc/mounts* verzeichnet, z. B. auf die folgende Weise:

```
rootfs / rootfs rw 0 0
/dev/root / reiserfs rw 0 0
proc /proc proc rw 0 0
sysfs /sys sysfs rw 0 0
devpts /dev/pts devpts rw 0 0
tmpfs /dev/shm tmpfs rw 0 0
/dev/hda2 /C vfat rw,nodiratime,fmask=0022,dmask=0022,codepage=cp437 0 0
usbdevfs /proc/bus/usb usbdevfs rw 0 0
/dev/hdb /F iso9660 ro,nosuid,nodev 0 0
```

Die Ausgabe von */proc/mount* entspricht fast exakt der Ausgabe von */etc/mtab*, nur dass */etc/mtab* von `mount` verwaltet wird und somit nur Dateisysteme auflistet, die mit `mount` eingehängt worden sind, nicht jedoch z. B. mit dem Systemcall `mount()`. Im Endeffekt ist dies eine Sicherheitslücke, denn auf diese Weise können Sie eine Partition quasi am System vorbei in ein Verzeichnis einhängen. Es gibt aber bis jetzt (Stand 2023) keine Angriffe, die den Systemcall `mount()` erfolgreich ausnutzen.

In der ersten Spalte einer jeden Zeile finden Sie das Gerät und in der zweiten Spalte den dazugehörenden Mountpoint. Der Dateisystemtyp wird in der dritten Spalte aufgelistet. Ob darauf nur lesend (`ro`) oder auch schreibend (`rw`) zugegriffen werden kann, wird – nebst anderen möglichen Optionen – in der vierten Spalte angezeigt. Die letzten beiden Spalten sind Dummy-Werte, damit das */proc/mount*-Format exakt dem von */etc/mtab* entspricht.

Die vom Kernel beim Bootvorgang einzuhängenden Dateisysteme werden allerdings in */etc/fstab* abgelegt. Auf diese Weise könnten Sie z. B. auch Ihre SD-Karte (falls sie vorher eingesteckt wurde) automatisch in */media* laden. Sie müssten dazu nur in */etc/fstab* die folgende Zeile eintragen:

```
/dev/hdc1 /media vfat(bzw. exfat) rw 0 0
```

Inzwischen funktioniert dies sogar mit Windows-Festplatten. Um beispielsweise Ihre Windows-Daten-Partition (die fast immer die zweite Partition auf der Festplatte am ersten SATA-Anschluss ist) auch unter Linux zu nutzen, verwenden Sie den folgenden Eintrag (Standardformatierung):

```
/dev/sda2 /windows ntfs rw 0 0
```

4.6 Weiterführendes

Zum */proc*-Verzeichnis könnten noch viele Seiten geschrieben werden, aber irgendwo muss Schluss sein. Sie wissen jetzt, wenn Sie bestimmte Informationen für Ihr Programm oder auch zur Administration benötigen, wo und wie Sie an diese Informationen herankommen. Weitere Informationen zum */proc*-Verzeichnis finden Sie auf der Manual Page (`man 5 proc`).

Dass Linux frei im Quellcode ist (Open Source), weiß – denken wir – schon jeder. Daher hierzu einige Anwendungen, deren Quellcode zu studieren sinnvoll erscheint, da sie kräftig Gebrauch vom */proc*-Verzeichnis machen.

Anwendung	Beschreibung
`mount`	Informationen über gemountete Datenträger
`ps`	Informationen über Prozesse
`top`	Auslastung der CPU
`xload`	durchschnittliche Auslastung des Systems
`xosview`	durchschnittliche Auslastung des Systems, der CPU, des Speicherbedarfs, Interrupts ...

Tabelle 4.3 Anwendungen, die vom /proc-Verzeichnis profitieren

Kapitel 10
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 auch parallel ablaufen. Sie erhalten in diesem Kapitel 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 Linux-Threads und sind somit nicht ohne weiteres auf anderen UNIXen lauffähig. Die BSD-Threads z. B. arbeiten zum Teil ähnlich. Es kann aber sein, dass das eine Programm läuft und ein anderes nicht und daher auf die Linux-Threads verlinkt werden muss. Voraussetzung sind also Linux-Threads. Speziell unter (Free)BSD müssen Sie die Linux-Threads aus den Ports installieren und das Programm folgendermaßen übersetzen:

```
$ gcc -o thserver thserver.c  \
  -I/usr/local/include/pthread/linuxthreads \
  -L/usr/local/lib -llthread -llgcc_r
```

Unter Linux reicht es aber aus, Ihre Programme wie folgt zu kompilieren:

```
gcc [programmname.c] -o [programmname] -lpthread
```

10.1 Unterschiede zwischen Threads und Prozessen

Prozesse wurden ja bereits ausführlich erklärt. Sie wissen also bereits, wie Sie eigene Prozesse mittels `fork()` kreieren können, und mit Interprozesskommunikationen (IPC) haben Sie erfahren, wie man einzelne Prozesse synchronisiert. Den Aufwand, den Sie bei den Interprozesskommunikationen betrieben haben, entfällt bei den Threads fast komplett.

Ein weiterer Nachteil bei der Erstellung von Prozessen gegenüber Threads ist der enorme Aufwand, der für die Duplizierung des Namensraumes nötig ist – den hat man mit den Threads nicht, da diese in einem gemeinsamen Adressraum ablaufen. Somit stehen den ein-

zelnen Threads dasselbe Codesegment, Datensegment, der Heap und alle anderen Zustandsdaten, die ein gewöhnlicher Prozess besitzt, zur Verfügung – was somit auch die Arbeit beim Austausch von Daten und bei der Kommunikation miteinander erheblich erleichtert. Weil aber kein Speicherschutzmechanismus unter 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 Threads auch nur innerhalb eines Prozesses ausgeführt werden. Der gravierende Unterschied zwischen Threads und Prozessen besteht darin, dass Threads unabhängige Befehlsfolgen innerhalb eines Prozesses sind. Man könnte auch sagen, Threads sind in einem Prozess gefangen oder gekapselt.

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

10.2 Thread-Bibliotheken

Zwar werden in diesem Buch nur die Linux-Threads behandelt, dennoch sollen weitere Bibliotheken nicht unerwähnt bleiben. Mitte der 90er-Jahre begann die Entwicklung von zahlreichen Thread-Bibliotheken, wobei sich letztendlich die Bibliothek von Xavier Leroy im Jahre 1997 unter dem Namen Pthread-Lib durchgesetzt hat. Diese Bibliothek wurde von Ulrich Drepper in *glibc2* an die Standardbibliothek angebunden und ist somit auf jedem Linux-System vorhanden. Nachdem Linux dann endlich für Mehrprozessorsysteme interessant wurde, wurden die Klagen über die Linux-Thread-Bibliothek lauter. Da es immer wieder Probleme mit Signalen, der hierarchischen Beziehung zwischen Threads, der immer noch nicht ganz implementierten POSIX-Konformität und noch andere Sorgen gab, war man auf der Suche nach neuen Thread-Bibliotheken.

- Einen interessanten Ansatz bietet die von Ralf S. Engelschall begonnene Bibliothek *GNU Pth*, die bereits 1999 als offizielles GNU-Projekt gestartet wurde. Das Gute an dieser Bibliothek ist, dass sie sich als eine möglichst portable Bibliothek für nicht präemptives Multitasking eignet.
- Programmierer von Intel und IBM verwendeten *GNU Pth* anschließend als Einstiegspunkt, um daraus die *Next Generation Posix Threads* (NGPT) zu entwickeln, die vor allem die Skalierung auf Mehrprozessorsysteme verbessern sollte.

- Als Dritte im Bunde kann man die Bibliothek von Red Hat mit der *Native Posix Thread Library* hervorheben, die gar eine Veränderung des Kernels nötig machte. Dabei wurden u. a. neue Systemaufrufe, ein erweiterter `clone()`-Aufruf und vor allem ein funktionierendes Signalhandling eingebaut.

Es sind zwar noch weitere Thread-Bibliotheken in Entwicklung, aber diese hier scheinen uns am interessantesten. Da einige dieser Thread-Bibliotheken noch recht neu sind und es immer noch an Dokumentation dazu fehlt, ergibt es relativ wenig Sinn, sie Ihnen hier zu demonstrieren. Daher greifen wir hier auf die altbewährte Linux-Thread-Bibliothek zurück.

10.3 Kernel- und User-Threads

Es gibt zwei Implementierungen von Threads, zum einen die Kernel-Threads, zum anderen die User-Threads. Die *User-Threads* sind in einer Bibliothek implementiert, die im Speicherbereich des Benutzers abläuft. Damit ist es möglich, Threads auch auf Betriebssystemen zu verwenden, die von Haus aus keine Threads unterstützen. Der Nachteil an den User-Threads ist aber, dass die einzelnen Threads eines Prozesses nicht auf unterschiedlichen Prozessoren laufen – dies gilt auch für unterschiedliche Kerne, für die zumindest gilt, dass eine Standardanwendung nicht bestimmen kann, welcher Thread welchen Kern benutzt. *Kernel-Threads* hingegen sind bereits im Betriebssystem integrierte Thread-Unterstützungen. Dabei wird natürlich auch das Scheduling des Betriebssystems verwendet. Somit ist es möglich, die einzelnen Threads eines Prozesses auf verschiedenen Prozessoren oder bestimmten Kernen laufen zu lassen. Die Linux-Threads, die in diesem Kapitel verwendet werden, unterstützen sowohl Kernel- als auch User-Threads.

Ein wenig überraschend ist es aber doch, dass die User-Threads, sofern man nicht an die Multiprozessorprogrammierung appelliert, einen gewissen Vorteil gegenüber der Kernel-Implementierung besitzen. Threads, die vom Kernel verwendet werden, setzen doch einige Grenzen, etwa was die Anzahl der gleichzeitigen Benutzer angeht. Aber noch ein wenig überraschender ist, dass User-Threads effizienter als Kernel-Threads ablaufen, da sie keinen extra Befehl (Software-Interrupt) zum Umschalten auf einen anderen Thread vom Kernel benötigen.

10.4 Scheduling und Zustände von Threads

Auch bei der Thread-Programmierung ist (wie bei den Prozessen) ein Scheduler entweder in der Thread-Bibliothek oder im Betriebssystem vorhanden, der bestimmt, wann welcher Thread Prozessorzeit erhält. Auch hier kann die Zuteilung wie schon bei den Prozessen prioritäts- und zeitgesteuert erfolgen. Bei zeitgesteuerten Threads bedeutet dies, dass jedem Thread eine bestimmte Zeit (des Prozessors oder der Prozessoren) zur Verfügung steht, ehe er automatisch unterbrochen wird und anschließend ein anderer Thread an der Reihe ist.

Sind die Threads prioritätsgesteuert, so erhält der Thread mit der höchsten Priorität vom Scheduler den Zuschlag. Außerdem wird ein laufender Thread abgebrochen, wenn ein Thread mit einer höheren Priorität ausgeführt wird. Bitte beachten Sie außerdem, wenn Sie das rein prioritätsgesteuerte Scheduling für die Thread-Programmierung verwenden, dass ein Thread mit der höchsten Priorität den Prozessor für eine uneingeschränkte Zeit verwenden und somit alle anderen Threads von der Arbeit abhalten kann. Wenn Sie einen Prozessor mit mehreren Kernen haben, dann fällt diese »Blockierung« natürlich nicht so dramatisch aus, aber manchmal, z. B. bei Spielen, entscheidet diese Tatsache dann doch darüber, ob das Spiel noch Spaß macht.

Bei einer User-Level-Thread-Implementierung steht Ihnen nur ein prioritätsgesteuertes Scheduling zur Verfügung, da ein zeitgesteuertes Scheduling die Verwendung von Systemcalls erfordert. Beide Scheduling-Arten hingegen stehen Ihnen zur Verfügung, wenn die Thread-Bibliothek im Kernel des Betriebssystems implementiert wurde. Bei modernen Linux-Versionen ist dies fast immer der Fall.

Anhand von Abbildung 10.1 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.

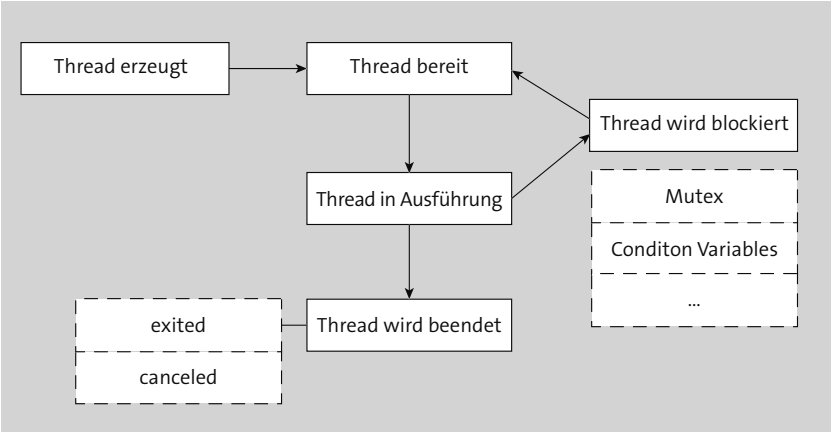

Abbildung 10.1 Zustände von Threads

- 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[-Kern] ein Thread).
- Wartet: Der Thread wird im Augenblick blockiert und wartet auf einen bestimmten Zustand (z. B. Bedingungsvariable, Mutex-Freigabe).
- Beendet: Ein Thread hat sich beendet oder wurde abgebrochen.

10.5 Die grundlegenden Funktionen der Thread-Programmierung

Hinweis: Rückgabewerte der pthread-Funktionen

Einen Hinweis gleich zu Beginn der Thread-Programmierung: Alle Funktionen aus der `pthread`-Bibliothek geben bei Erfolg `0`, ansonsten (also bei einem Fehler) `-1` zurück.

10.5.1 Mit pthread_create einen neuen Thread erzeugen

Sie können einen neuen Thread mit der Funktion `pthread_create` erzeugen. Diese Funktion erwartet die folgenden Parameter:

```
#include <pthread.h>

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

Wenn Sie sich die Funktion genau anschauen, dürfte Ihnen die Ähnlichkeit mit der Funktion `clone()` auffallen (siehe Manual Page), auf der `pthread_create()` unter Linux ja auch aufbaut. Jeder Thread bekommt eine eigene Identifikationsnummer (ID), die aber Bestandteil einer Variablen vom Typ `struct pthread_t` ist, die im ersten Parameter `thread` als Zeiger übergeben wird. Anhand dieser ID werden alle Threads voneinander unterschieden. Mit dem zweiten Parameter, `attribute`, können Sie bei dem neu zu startenden Thread Attribute setzen, wie z. B. die Priorität, die Stackgröße und noch einiges mehr. Auf die einzelnen Attribute werden wir später noch detaillierter eingehen, auch mittels Beispielen. Geben Sie für die Attribute `NULL` an, werden Standardattribute für den Thread genommen. Mit dem dritten Parameter bestimmen Sie die Funktion für den Thread selbst. Hierbei geben Sie natürlich nur die Anfangsadresse einer Routine an, mit der der Thread starten soll (Typ `void*`). Das bedeutet, dass, wenn sich die hier angegebene Funktion beendet, dies auch automatisch das Ende des Threads ist und Threads deshalb fast immer in einer längeren Schleife laufen. 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 der Anweisungen hinter `pthread_create()` fort. Der neu erzeugte Thread führt also sofort asynchron seine Arbeit aus. Jetzt werden 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 zunächst mit seiner Ausführung beginnt, ist nicht festgelegt (dasselbe Verhalten wie bei `fork`).

10.5.2 Mit pthread_exit einen Thread beenden

Sie können einen Thread auf unterschiedliche Weise beenden. Meistens werden Threads jedoch mit der Funktion `pthread_exit()` beendet, die folgende Syntax besitzt:

```
#include <pthread.h>

void pthread_exit( void * wert );
```

Die Funktion `pthread_exit()` beendet nur den Thread, mit dem Sie sie aufrufen. Mit dem Argument `wert` geben Sie den Exit-Status des Threads an. Diesen Status können Sie mit `pthread_join()` ermitteln (die Erklärung folgt in den nächsten Abschnitten). 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 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 gerne verwendet, um z. B. temporäre Dateien zu löschen, Mutexe freizugeben oder eben sonstige Reinigungsarbeiten durchzuführen. Ein solcher vorher eingerichteter Exit-Handler wird automatisch bei Beendigung eines Threads mit z. B. `pthread_exit()` oder `return` ausgeführt. Das Prinzip ist ähnlich, wie Sie es von der Standardbibliotheksfunktion `atexit()` bereits kennen sollten, wenn Sie die ersten Kapitel aufmerksam gelesen haben: Auch bei `pthread_exit()` werden bei mehreren Exit-Handlern die einzelnen Funktionen in umgekehrter Reihenfolge der Einrichtung ausgeführt. Die beiden folgenden Funktionen dienen zum Einrichten von Exit-Handlern:

```
#include <pthread.h>

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

Eine Funktion, die als Exit-Handler dienen soll, richten Sie also mit der Funktion `pthread_cleanup_push()` ein. Als ersten Parameter übergeben Sie dabei die Funktion, die ausgeführt 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.

Etwas, was leider sehr gewöhnungsbedürftig ist, ist, dass die beiden Funktionen `pthread_cleanup_push()` und `pthread_cleanup_pop()` als Makros implementiert sind – was nicht so schlimm wäre, wenn `pthread_cleanup_push()` eine sich öffnende geschweifte Klammer enthielte und `pthread_cleanup_pop()` eine sich schließende. Dies bedeutet, dass Sie beide Funktionen im selben Anweisungsblock ausführen müssen. Sie müssen also immer ein `_push` und ein `_pop` verwenden, auch wenn Sie wissen, dass eine `_pop`-Marke nie erreicht wird.

10.5.3 Mit pthread_join auf das Ende eines Threads warten

Bevor Sie sich dem ersten Listing widmen können, benötigen Sie noch Kenntnis der Funktion `pthread_join()`, die folgende Syntax besitzt:

```
#include <pthread.h>

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

`pthread_join()` hält den hier aufgerufenen Thread (meistens ist dies der Haupt-Thread), der zuvor einen Thread mit `pthread_create()` erzeugt hat, so lange an, bis der Thread `thread` vom Typ `pthread_t` beendet wurde. Der Exit-Status (also z. B. der Rückgabewert) des Threads wird an die Adresse von `thread_return` geschrieben. Sind Sie nicht am Rückgabewert interessiert, können Sie auch `NULL` verwenden. `pthread_join()` ist also das, was Sie bei den Prozessen als `waitpid()` kennen. Allerdings werden Thread-IDs ganz anders berechnet als `PID`s.

Ein Thread, der sich beendet, wird übrigens so lange nicht freigegeben (bzw. als beendeter Thread betrachtet), bis ein anderer Thread `pthread_join()` aufruft. Es können also auch mit Threads Zombieprozesse erzeugt werden. Daher sollten Sie für jeden zuvor erzeugten Thread einmal `pthread_join()` aufrufen, es sei denn, Sie haben einen Thread vom Hauptprozess abgehängt (dazu mehr in den nächsten Abschnitten).

10.5.4 Mit pthread_self die ID von Threads ermitteln

Die Identifikationsnummer eines noch auszuführenden Threads können Sie mit der Funktion `pthread_self()` erfragen, die folgende Syntax hat:

```
#include <pthread.h>

pthread_t pthread_self(void);
```

Als Rückgabewert erhalten Sie die Thread-ID als Variable vom Datentyp `pthread_t`. Auf den meisten Linux-Systemen ist eine Thread-ID identisch mit einer Variablen vom Typ `uint32_t`, aber dies muss nicht unbedingt immer der Fall sein. Auf einigen 64-Bit-Linuxen kann auch der Datentyp `unsigned long long` eingesetzt werden. Damit könnte man theoretisch so viele Thread-IDs erzeugen, wie es Teilchen im Universum gibt. Dass man hier so lange Zahlen verwendet, hat eher etwas mit der optimalen Wortbreite moderner Prozessoren zu tun als mit der Unterstützung möglichst vieler Möglichkeiten.

Zur Erstellung von Threads folgt ein einfaches Beispiel, das alle bisher vorgestellten Funktionen in klarer Weise demonstriert.

```
/* threads1.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 u. mit Werten belegen */
        f = (struct data *)malloc(sizeof(struct data));
        if(f == NULL) {
            printf("Konnte keinen Speicher reservieren ...!\n");
            exit(EXIT_FAILURE);
        }
        /* Zufallszahl zwischen 1 und 10 (Spezial) */
        f->wert =  1+(int) (10.0*rand()/(RAND_MAX+1.0));
        snprintf (f->msg, BUF, "Ich bin Thread Nr. %d", i+1);
        /* Jetzt Thread erzeugen */
        if(pthread_create(&th[i], NULL, &mythread, f) != 0) {
            fprintf (stderr, "Konnte Thread nicht erzeugen\n");
            exit (EXIT_FAILURE);
        }
```

```
    }
    /* Auf das Ende der Threads warten */
    for( i=0; i < MAX_THREADS; i++)
        pthread_join(th[i], &ret[i]);
    /* Rückgabewert der Threads ausgeben */
    for( i=0; i < MAX_THREADS; i++)
        printf("<-Thread %ld ist fertig\n", ret[i]);
    /* Haupt-Thread ist jetzt auch fertig */
    printf("<- Main-Thread beendet (ID: %ld)\n",
        pthread_self());
    return EXIT_SUCCESS;
}
```

Listing 10.1 Das Erzeugen und das sichere Beenden von Threads

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads1.c -o threads1 -lpthread
$ ./threads1

-> 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: Casten von void*

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 wen die Warnmeldung stört, der kann entweder trotzdem `void*` casten oder aber mit dem Parameter `-wstd` statt `-wall` die Warnungen auf »Standard« stellen und dadurch die Meldung abschalten, dass Sie `void*` casten sollten. Eine andere Möglichkeit ist, möglichst den neusten C-Standard zu verwenden, entweder mit `-std=c18` oder wenn verfügbar schon mit `-std=c23`.

Das letzte Beispiel demonstriert auf einfache Weise, wie Sie Daten an einen neu erzeugten Thread übergeben können (hier mit der Struktur `data`). Ebenfalls wird die Verwendung eines Exit-Handlers demonstriert, der hier aber 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 geeignet. In dem letzten Beispiel wurden außerdem drei Threads vom Typ `mythread` erzeugt, die im Prinzip alle dasselbe machen, nämlich eine einfache Ausgabe der Daten, die vorher an die Threads übergeben wurden. Hierbei müssen wir nochmals explizit darauf hinweisen, dass die Reihenfolge, in der die Threads starten, nicht vorgegeben ist, auch wenn dies hier einen anderen Anschein macht. Hierzu wären extra Synchronisationsmechanismen erforderlich, die in dem letzten Beispiel fehlen. Jeder Thread wird am Ende 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 mit der Arbeit 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 nicht mehr gültige Strukturvariablen zugreifen würden. Was dann geschieht, kann nicht wirklich vorausgesagt werden, meistens wird der Zugriff auf nicht mehr gültige Speicherbereiche aber die Meldung »segmentation fault« nach sich ziehen.

Rückgabewerte von Threads auswerten

Zwar sind wir schon auf den Rückgabewert von Threads eingegangen, aber es wurden nur threadspezifische Daten zurückgegeben (hier die Thread-ID). Aber genauso wie schon bei der Wertübergabe an Threads können Sie auch ganze Strukturen zurückgeben, was in der Praxis auch häufig der Fall ist. Es folgt hierzu ein ähnliches Beispiel wie schon im Listing *threads1.c*, nur dass jetzt die Daten der Struktur `data` auch 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;
}
```

Listing 10.2 Zeigt, wie Sie Threads Daten übergeben und diese Daten ausgeben können.

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads2.c -o threads2 -lpthread
$ ./threads2

-> 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)
```

10.5.5 Mit pthread_equal die ID von zwei Threads vergleichen

Um einen Thread mit einem anderen Thread zu vergleichen, kann die Funktion `pthread_equal()` verwendet werden. Sie wird häufig eingesetzt, um sicherzugehen, dass nicht ein Thread gleich dem 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. Vielleicht wundern Sie sich darüber, wie es sein kann, dass zwei Threads ein und derselbe Thread sind, deshalb folgt hier auch ein weiteres Beispiel, das die Sache erklärt.

Das folgende Beispiel erzeugt drei Threads – jeder Thread soll wiederum eine andere Aktion ausführen. Im Beispiel ist dies zwar nur die Ausgabe eines Textes, aber in der Praxis könnten Sie auch eigene Funktionen aufrufen. Für die ersten drei Threads wird jeweils eine bestimmte Aktion festgelegt. Alle anderen Threads führen nur noch den `else`-Zweig aus. Dies ist z. B. sinnvoll, wenn Sie in Ihrer Anwendung bestimmte Vorbereitungen treffen wollen (im Beispiel eben drei Vorbereitungen), wie z. B. Dateien anlegen, Müll beseitigen, eine Serververbindung herstellen und noch vieles mehr. Sind diese Vorbereitungen getroffen, wird später immer mit der gleichen Funktion fortgefahren. Damit der Vergleich von Threads mit `pthread_equal()` auch funktioniert, werden die Thread-IDs, die beim Anlegen mit `pthread_create()` erzeugt werden, in globalen Variablen abgelegt (und sind daher auch für alle Threads sichtbar). Im Endeffekt ist diese Lösung natürlich suboptimal, denn globale Variablen bergen immer ein gewisses Risiko (eine gute Übung wäre an dieser Stelle sicherlich, *threads3.c* mit Semaphoren zu implementieren).

```
/* threads3.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;
}
```

Listing 10.3 Demonstriert die Funktion pthread_equal().

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads3.c -o threads3 -lpthread
$ ./threads3
->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 erkennen an dem letzten Beispiel, dass die ersten drei Threads jeweils »abc«, »efg« und »jkl« ausgeben. Alle danach folgenden Threads führen dann stets »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.

10.5.6 Mit pthread_detach einen Thread unabhängig vom Hauptprogramm machen

Das Gegenteil von `pthread_join()` stellt die Funktion `pthread_detach()` dar. Mit dieser Funktion legen Sie fest, dass nicht mehr auf die Beendigung eines Threads gewartet werden soll. Die Funktion `pthread_detach()` hat die folgende Syntax:

```
#include <pthread.h>

int pthread_detach( pthread_t thread );
```

Sie lösen durch `pthread_detach()` praktisch den Thread mit der ID `thread` von der Hauptanwendung. Sie können diesen Vorgang gerne mit Dämonprozessen vergleichen. Dass dieser Thread-Dämon selbstständig ist, ist nichts Magisches, denn im Grunde markieren Sie den Thread damit nur mit einem bestimmten Flag, so dass bei seinem Beenden der Exit-Status und die Thread-ID gleich mit freigegeben werden. Ohne `pthread_detach()` würde dies erst nach einem `pthread_join()`-Aufruf geschehen. Natürlich bedeutet die Verwendung von `pthread_detach()`, dass auch kein `pthread_join()` mehr auf das Ende des Threads reagiert, aber zum Glück auch, dass (da ja der Thread automatisch entfernt wird) keine Zombieprozesse entstehen können.

Hinweis zu pthread_detach()

Ein Thread, der mit `pthread_detach()` oder 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. Allerdings beendet sich der losgelöste Thread automatisch, wenn das Hauptprogramm beendet wird.

Ein typischer Codeausschnitt, der zeigt, wie Sie einen Thread von den anderen loslösen können, sieht wie folgt 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);
}
```

10.6 Die Attribute von Threads und das Scheduling

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

```
#include <pthread.h>

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

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

mit pthread_attr_setdetachedstate() wird es gesetzt. Neben dem eben erwähnten Attribut PTHREAD_CREATE_JOINABLE, das auch immer die Standardeinstellung eines erzeugten Threads ist, können Sie natürlich auch PTHREAD_CREATE_DETACHED verwenden. Das Setzen von PTHREAD_CREATE_DETACHED entspricht exakt dem Verhalten der Funktion pthread_detach() (siehe Abschnitt 10.5.6) und kann auch stattdessen verwendet werden. Benötigen Sie das Attribut attr nicht mehr, können Sie es mit pthread_attr_destroy() löschen. Somit ergeben die letzten beiden Funktionen auch erst Sinn, wenn Sie bereits mit pthread_detach() einen Thread ausgehängt haben und diesen eventuell wieder zurückholen wollen (Sie setzen also wieder das Flag PTHREAD_CREATE_JOINABLE).

Bedeutend wichtiger im Zusammenhang mit den Attributen von Threads ist aber sicherlich das Setzen der Prozessorzuteilung (*Scheduling*). Laut POSIX gibt es drei verschiedene solcher Prozessorzuteilungen (*Scheduling Policies*):

- SCHED_OTHER: Dies ist die normale Priorität wie bei einem gewöhnlichen Prozess. Der Thread wird schlafen gelegt, entweder wenn seine Zeit um ist und er deshalb warten muss, bis er wieder am Zug ist, oder wenn ein anderer Thread oder Prozess gestartet wurde, der mit einer höheren Priorität ausgestattet ist.
- Echtzeit (SCHED_FIFO): Dies ist die Zuteilung für 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:
  1. Er tritt in eine Warteschleife ein und wartet auf ein externes Ereignis (z. B. einen Interrupt).
  2. Er verlässt freiwillig die CPU und lässt sich auslagern (z. B. mit sched_yield()).
  3. Er 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 sie verstrichen, so kommt der nächste Prozess an die Reihe. Unter Linux werden diese Prozesse genauso behandelt wie die Echtzeitprozesse, mit dem Unterschied, dass diese an das Ende der *run-queue* gesetzt werden, wenn sie den Prozessor verlassen.

An dieser Stelle haben wir zum ersten Mal Echtzeitoperationen ins Spiel geworfen, und deshalb folgt nun auch ein kleiner kurzer Exkurs dazu, damit Sie die Echtzeitstrategie nicht mit »jetzt, gleich, sofort« übersetzen. Die Abarbeitung von Daten in »Echtzeit« kann natürlich niemals direkt sofort erfolgen, sondern auch hier müssen Sie sich damit begnügen, dass die Echtzeitaufträge innerhalb einer vorgegebenen Zeitspanne abgearbeitet werden müssen. Wenn Sie z. B. eine Audiodatei im Hintergrund abspielen, dann müssen Sie nicht gewährleisten, dass alle anderen Prozesse warten, bis die Datei abgespielt wurde (dies ist dann ja auch nicht beabsichtigt), sondern nur, dass der Audiopuffer nicht leerläuft. Allerdings müssen solche Echtzeitoperationen auch unterbrechbar sein, um auf unvorhersehbare Ereignisse reagieren zu können (etwa wenn die Datei beschädigt ist und der Audiopuffer dann aufgrund eines Fehlers leerläuft). Daher unterscheidet man zwischen weichen und harten Echtzeitan-

forderungen. Die Anforderungen hängen natürlich stets vom Anwendungsfall ab. So kann man beispielsweise bei einem Computerspiel zunächst weiche Echtzeitanforderungen ausprobieren und später genau dort harte Anforderung setzen, wo sich dies nicht vermeiden lässt (z. B. muss bei der 3D-Engine oft innerhalb einer vorgegebenen Zeit reagiert werden, sonst »ruckelt« die Ausgabe). Die Hauptbereiche von Echtzeitanwendungen sind also die folgenden:

- Multimedia, Audio, Video, Gaming (besonders 3D-Spiele)
- Steuerung von Maschinen, Robotern und autonomen Modellfahrzeugen (KI)

Damit eine solche Zuteilungsstrategie auch funktioniert, muss das System sie natürlich auch unterstützen. Dies ist gegeben, wenn bei Ihnen 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.

Verändern bzw. abfragen können Sie die Zustellungsstrategie 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 Sie (`set`) oder ermitteln Sie (`get`) die Zustellungsstrategie eines Threads mit der ID `thread` vom Typ `pthread_t`. Die Strategie selbst legen Sie mit dem Parameter `policy` fest. Hierbei kommen die bereits beschriebenen Konstanten `SCHED_OTHER`, `SCHED_FIFO` und `SCHED_RR` in Frage. 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;
};
```

In dem folgenden Beispiel zeigen wir Ihnen, wie einfach es ist, die Zuteilungsstrategie und die Priorität von Threads zu verändern. Sie finden hierbei auch zwei Funktionen, nämlich eine, mit der Sie die Strategie und die Priorität abfragen, und eine, mit der Sie diese Werte neu setzen. Allerdings benötigen Sie für das Setzen von Thread-Prioritäten Superuser-Rechte, was in dem folgenden Beispiel ebenfalls abgefragt wird.

```
/* threads4.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;
}
```

Listing 10.4 Die Verwendung und das Setzen von Thread-Prioritäten

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads4.c -o threads4 -lpthread
$ ./threads4
->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:********
# ./threads4
->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 ...
```

Zuteilungsstrategien und Prioritäten können Sie übrigens mit folgenden Funktionen auch über sogenannte *Attributobjekte* (`pthread_attr_t`) setzen und abfragen:

```
#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 oder 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`, was bedeutet, dass der Kind-Thread die Attribute (mitsamt Zuteilungsstrategie und der Priorität) des Eltern-Threads übernimmt, und `PTHREAD_EXPLICIT_SCHED`, was bedeutet, eben nichts zu übernehmen, sondern nur 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.

10.7 Threads synchronisieren

In vielen Fällen (eigentlich bei Threads fast immer) werden mehrere parallellaufende Prozesse benötigt, die gemeinsame Daten verwenden und/oder austauschen. Einfachstes Beispiel: Ein Thread schreibt gerade etwas in eine Datei, während ein anderer Thread daraus etwas liest. Dasselbe Problem haben Sie natürlich beim Zugriff auf globale Variablen. Wenn also mehrere Threads auf dieselben Ressourcen zugreifen und Sie keine Vorkehrungen getroffen haben, die Ressourcen zu verwalten, ist nicht vorherzusagen, welcher Thread welche Variable wann bearbeitet. Wenn Sie also z. B. mathematische Berechnungen auf mehrere Threads verteilen wollen, müssen Sie die einzelnen Threads logischerweise synchronisieren.

Hierfür sei das folgende einfache Beispiel gegeben: Zwei Threads greifen auf eine globale Variable zu, in diesem Fall auf einen zuvor geöffneten `FILE`-Zeiger. Ein Thread wird nun erzeugt, um etwas in diese Datei zu schreiben, und ein weiterer Thread soll diese Daten wieder auslesen. Ein simples Beispiel, wie es scheint, nur dass es hierbei schon zu Synchronisationsproblemen (sogenannten *Race Conditions*) kommt. Aber schauen Sie selbst, was in dem nächsten Beispiel geschieht:

```
/* threads5.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;
}
```

Listing 10.5 Zeigt Ihnen, dass bei einer gleichzeitigen Benutzung einer Datei durch zwei Threads Synchronisationsprobleme auftreten.

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads5.c -o threads5 -lpthread
$ ./threads5
->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
```

Anhand der obigen Eingaben erkennen Sie , dass der Thread `thread_lesen()` bereits mit seiner Ausgabe begonnen hat und sich schon wieder beendet hat, noch bevor Sie etwas auf der Tastatur eintippen konnten. Richtig ausgeführt sollte hier folgende Ausgabe bei der Programmausführung entstehen:

```
$ ./threads5
->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 schon etwas einfallen, z. B. ein Polling mit einer Schleife um den Lese-Thread herum, die immer wieder abfragt, ob `fgets()` etwas eingelesen hat. Aber vielleicht würde ein echter C-Guru an dieser Stelle dann doch was völlig anderes programmieren, z. B. mit Semaphoren. Natürlich ist das letzte Beispiel auch nicht für Echtzeitanwendungen geeignet, denn wenn beispielsweise die Weichenschaltung einer U-Bahn in einer pollenden Schleife jedes Mal warten muss, bevor die Weiche gestellt werden kann, dann könnte dies sogar zu erheblichen Verzögerungen führen (es sind sogar oft solche einfachen Dinge, die für die meisten Verzögerungen im Nahverkehr verantwortlich sind). Sie sehen also, dass Sie noch lange nicht am Ende sind und dass sogar Profis oft Fehler machen, bei denen sie anschließend denken: »Wie konnte gerade mir so was Dummes passieren?« Aber deshalb kommen ja auch noch mehr Beispiele, die Ihnen aufzeigen, welche Synchronisationsmöglichkeiten Ihnen die Thread-Bibliothek noch so anbietet.

10.7.1 Mutexe

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

Um Threads zu synchronisieren, haben Sie zwei Möglichkeiten: Zum einen mit sogenannten *Locks*, die Sie in diesem Kapitel in Kürze mit den Mutexen verwenden werden, und zum anderen mit einem *Monitor*. Mit dem Monitor werden sogenannte *Condition-Variablen* verwendet, die (wie der Name ja schon andeutet) in Abhängigkeit von Bedingungen gesetzt werden. Zunächst verwenden wir aber Mutexe. Ein *Mutex* ist im Endeffekt eine bestimmte Art von Semaphor: Wenn es gesetzt ist (= `1` = wahr), dann kann ein Thread einen kritischen Bereich passieren. Wenn es gelöscht ist (ein anderer Thread hat dann quasi die Marke entfernt), dann muss der Thread, der versucht, den kritischen Bereich zu passieren, auf die Marke warten.

Hinweis

Der Begriff Mutex steht für *mutual exclusion* (= gegenseitiger Ausschluss). Ein Mutex ist somit ohne Besitzer und quasi eine globale Variable.

Die Funktionsweise von Mutexen ähnelt also den Semaphoren bei den Prozessen. Genauer noch: Ein Mutex ist nichts weiter als ein Semaphor, auf dem nur eine atomare Operation ausgeführt wird (sperren oder freigeben). Trotzdem lassen sich Mutexe aber wesentlich einfacher erstellen. Das Prinzip ist simpel: 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 sie wieder frei. Mutexe sind also vom Typ binärer Operator.

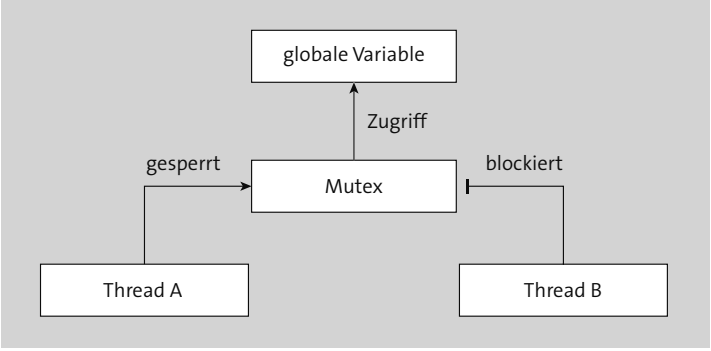

Abbildung 10.2 Nur ein Thread kann einen Mutex sperren.

Anhand der obigen Erklärung dürfte auch klar sein, dass Sie selbst dafür verantwortlich sind, keinen Deadlock zu erzeugen. In den folgenden Fällen könnten auch bei Threads Deadlocks auftreten:

- Threads fordern Ressourcen an, obwohl sie bereits die Ressourcen besitzen.
- Ein Thread gibt seine Ressource nicht mehr frei, weil er abgestürzt ist.
- Eine Ressource ist frei, aber im Besitz eines exklusiven Threads.

Im Fall eines Deadlocks kann keiner der beteiligten Threads seine Arbeit mehr fortsetzen, und somit ist meist keine normale Beendigung mehr möglich. Datenverluste können ebenso 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);
```

Hinweis: Abstraktionsebene bei Mutex-Variablen

Wie Sie in der obigen Auflistung erkennen, können Sie Mutexe nur sperren und entsperren. Welche Werte bei den dahintersteckenden Semaphoren benutzt werden, verschwindet hier hinter einer Abstraktionsschicht. Normalerweise entfernt aber die Lock-Funktion eine Marke, und die Unlock-Funktion setzt sie wieder.

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

Die Funktion `pthread_mutex_trylock()` arbeitet 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 eine korrigierte Version von *threads5.c*, die das vorige Synchronisationsproblem mithilfe eines Mutex behebt. Zuerst wird der globale Mutex mit der Konstanten `PTHREAD_MUTEX_INITIALIZER` statisch initialisiert, und anschließend werden im Beispiel die Sperren dort gesetzt und wieder freigegeben, wo dies sinnvoll ist.

```
/* threads6.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;
}
```

Listing 10.6 Die Synchronisationsmöglichkeiten von Threads durch Mutexe

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads6.c -o threads6 -lpthread
$ ./threads6
->Haupt-Thread (ID: -1209412512) gestartet ...
Bitte Eingabe machen: Hallo Welt mit Mutexe
Ausgabe Thread -1217807440: Hallo Welt mit Mutexe
<-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()` wie folgt umändern:

```
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 andere Arbeiten ausführen.

Das veränderte Programm gibt bei der Ausführung mit `pthread_mutex_trylock()` Folgendes aus:

```
$ gcc threads6.c -o threads6 -lpthread
$ ./threads6
->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 Mutexe, für die zur Laufzeit mit `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 das Mutex verwenden. Wird hier `NULL` angegeben, werden die Standardattribute verwendet (auf die Attribute von Mutexen werden wir in den nächsten Abschnitten noch eingehen). Freigeben können Sie einen solchen dynamisch angelegten Mutex wieder mit `pthread_mutex_destroy()`. Hierzu folgt dasselbe Beispiel (also *thread6.c*), nur mit einem dynamisch angelegten Mutex:

```
/* threads7.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;
}
```

Listing 10.7 Verwendet dynamische Mutex-Variablen.

Die Schilderung des Programms bei der Ausführung kann hier entfallen, 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 für `kind` verwenden:

- `PTHREAD_MUTEX_FAST_NP` (Standardeinstellung): `pthread_mutex_lock()` blockiert den aufrufenden Thread für unbegrenzte Zeit, bis er wieder freigegeben wird.
- `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 er 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 mit dem Suffix `_NP` (*non-portable*) verbunden sind, sind sie nicht mit dem POSIX-Standard vereinbar und somit nicht geeignet für portable Programme.

10.7.2 Condition-Variablen (Bedingungsvariablen)

Bedingungsvariablen werden dazu verwendet, auf das Eintreten einer bestimmten Bedingung zu warten bzw. die Erfüllung oder den Eintritt einer Bedingung anzuzeigen. Bedingungsvariablen werden außerdem mit Mutexen verknüpft. Dabei wird beim Warten auf eine Bedingung eine Sperre auf diesem Mutex freigegeben (natürlich muss zuvor eine Sperre auf den Mutex erfolgt sein).

Umgekehrt sollte vor dem Eintreten einer Bedingung eine Sperre auf dem verknüpften Mutex erfolgen, so dass 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 Bedingungsvariablen wird der Datentyp `pthread_cond_t` verwendet. Damit eine solche Bedingungsvariable überhaupt als statisch definiert ist, muss sie mit der Konstanten `PTHREAD_COND_INITIALIZER` initialisiert werden. Es folgen die Funktionen, mit denen 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 die Funktion `pthread_cond_wait()` verwenden, müssen Sie im aufrufenden Thread das Mutex `mutex` sperren. Mit einem anschließenden `pthread_cond_wait()` wird der Mutex dann freigegeben, und der Thread wird anschließend mit der Bedingungsvariablen `cond` bis zum Eintreten einer bestimmten Bedingung blockiert. Bei einem erfolgreichen Aufruf von `pthread_cond_wait()` wird auch für den Mutex automatisch die Sperre wieder eingerichtet – oder einfach: Es herrscht wieder der Zustand wie vor dem `pthread_cond_wait()`-Aufruf.

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

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

Natürlich gibt es auch noch eine Funktion, mit der Sie im Gegensatz zu `phtread_cond_wait()` nur eine gewisse Zeit auf die Bedingungsvariable `cond` warten können: die Funktion `pthread_cond_timewait()`. Als Zeitangabe können Sie den Parameter `abstime` verwenden, mit dem 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
};
```

Das folgende, recht einfache Beispiel demonstriert die Funktionalität von Bedingungsvariablen und deren Verwendung:

```
/* threads8.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 warten, 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);
}
```

Listing 10.8 Die Verwendung von Condition-Variablen

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads8.c -o threads8 -lpthread
$ ./threads8
->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 in dem letzten Beispiel, das, sobald der Haupt-Thread eine Bedingungsvariable sendet, eine Kettenreaktion entsteht: Die Threads werden entsprechend der Reihenfolge, in der sie vorher in der Queue angelegt wurden, abgearbeitet. Das heißt also, dass hier wieder eine Warteschlange angelegt wird. Dazu folgt ein weiteres simples Beispiel: Der Thread Nummer 2 wartet auf die Condition-Variable von Thread Nummer 1. Thread Nummer 1 weist einem globalen Zahlenarray `werte` zehn Zahlen zu, die Thread 2 anschließend für weitere Berech-

nungen benutzt. Dies ist natürlich wieder ein primitives Beispiel und soll nur die Funktion von Condition-Variablen demonstrieren.

```
/* threads9.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 Codeausschnitt 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;
}
```

Listing 10.9 Die Synchronisation zweier Threads mittels Condition-Variablen

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads9.c -o threads9 -lpthread
$ ./threads9
->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: Fehlerprüfungen

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 steht Ihnen – wie bei den Mutexen auch – die Möglichkeit zur Verfügung, Bedingungsvariablen dynamisch anzulegen, 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 (darauf werden wir im nächsten Abschnitt eingehen). 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()`.

Es folgt dasselbe Beispiel wie schon im Listing *threads9.c*, nur eben als Variante mit dynamischen Datenstrukturen.

```
/* threads10.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 Codeausschnitt 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;
}
```

Listing 10.10 Verwendet im Gegensatz zu Listing 10.9 dynamische Datenstrukturen.

Ein weiteres 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 Server-Client-Prinzip also. Der Haupt-Thread simuliert dann die Situation, dass er zwei Datenpakete an einen Client-Thread verschickt. Der Client-Thread simuliert anschließend die Situation, dass er die Datenpakete bearbeiten muss. Im nächsten Beispiel werden übrigens wieder statische Bedingungsvariablen verwendet.

```
/* threads11.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);
    }

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

       /* Schütze geteilte (shared) Daten und Flags */
       rc = pthread_mutex_lock (&dataMutex);
       checkResults ("pthread_mutex_lock()\n", rc);
       printf ("->Server: Sperre die Daten und gib eine "
           "Meldung an Consumer\n");
       ++geteilteDaten;     /* Füge "shared" Daten hinzu  */
       DatenVorhanden = 1;  /* ein vorhandenes Datenpaket */
       /* Client wieder aufwecken */
       rc = pthread_cond_signal (&DatenVorhandenCondition);
       if (rc) {
```

```
            pthread_mutex_unlock (&dataMutex);
            printf ("Server: Fehler beim Aufwecken von "
                "Client, rc=%d\n", rc);
            exit (EXIT_FAILURE);
        }
        printf("->Server: Gibt die gesperrten Daten"
            " wieder frei\n");
        rc = pthread_mutex_unlock (&dataMutex);
        checkResults ("pthread_mutex_lock()\n", rc);
    }// Ende while(anzahlDaten--)
    for (i = 0; i < NUMTHREADS; ++i) {
        rc = pthread_join (thread[i], NULL);
        checkResults ("pthread_join()\n", rc);
    }
    printf ("->Main-Thread ist fertig\n");
    return EXIT_SUCCESS;
}
```

Listing 10.11 In diesem Listing schicken sich Threads gegenseitig Nachrichten.

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads11.c -o threads11 -lpthread
$ ./threads11
->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 ergeben 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.

10.7.3 Threads und Semaphore

Threads können auch mit Semaphoren synchronisiert werden. Wie Sie bereits in den Kapiteln zur Interprozesskommunikation (Kapitel 7 und 8) 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 ihren Threads verwendet oder aber, wenn hier 0 gesetzt wird, 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 suspendierte Thread mit seiner Ausführung fortfahren. Des Weiteren dekrementiert sem_wait 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, der mit sem_wait() suspendiert wurde, aufwecken, müssen Sie sem_post() aus einem anderen Thread heraus aufrufen. sem_post() ist genau wie sem_trywait() 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 letzten Listing, nur dass anstatt Bedingungsvariablen und Mutexen eben Semaphore verwendet werden. Mithilfe von Semaphoren lässt sich eine Synchronisation zwischen Threads erheblich einfacher realisieren.

```
/* threads12.c */
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>
#include <semaphore.h>

static int werte[10];
sem_t sem;

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

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

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

    /* Semaphor um 1 inkrementieren */
    sem_post(&sem);
```

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

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

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

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

    printf ("\t->Thread %ld gestartet ...\n",
        pthread_self ());
    for (i = 0; i < 10; i++)
        summe += werte[i];

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

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

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

    /* Mit Absicht umgekehrt */
    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);
    /* Semphore löschen */
    sem_destroy(&sem);
    printf("->Semaphor gelöscht\n");
    printf("->Main-Thread %ld beendet\n", pthread_self());
    return EXIT_SUCCESS;
}
```

Listing 10.12 Zeigt, wie Sie Threads mit Semaphoren synchronisieren können.

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads12.c -o threads12 -lpthread
$ ./threads12
->Main-Thread -1209416608 gestartet
->Semaphor initialisiert (Wert: 0)

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

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

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

10.7.4 Weitere Synchronisationstechniken im Überblick

Neben den hier vorgestellten Synchronisationsmechanismen bietet Ihnen die `phtread`-Bibliothek 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 (geteilten, also *shared*) Datenbereich lesen, aber nur ein Thread zum selben Zeitpunkt etwas in diesen Bereich schreiben darf (*one writer, many readers*). Alle Funktionen dazu beginnen mit dem Präfix `pthread_rwlock_`.
- *Barrier*: Als Barrier bezeichnet man einen Punkt, der als zunächst unüberwindbare Barriere verwendet wird, die erst dann überwunden werden kann, wenn eine bestimmte Anzahl von Threads an diese Barriere kommt, eben das Prinzip der hohen Mauer bei den

Pfadfindern, die man nur im Team (also mit einer gewissen Anzahl von Personen) überwinden kann. Solange eine gewisse Anzahl von Threads nicht vorhanden ist, müssen 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 gute Synchronisationsmöglichkeit. Alle Funktionen zu den Barriers beginnen mit dem Präfix `pthread_barrier_`.

- *Spinlocks*: Spinlocks sind nur für Multiprozessorsysteme oder CPUs mit mehreren Kernen interessant. Das Prinzip ist dasselbe wie bei den Mutexen, nur dass – anders als bei den Mutexen – ein Thread, der auf einen Spinlock wartet, nicht die CPU freigibt, sondern eine sogenannte *Busy Loop* (also eine Warteschleife, in der sonst nichts passiert) ausführt, bis der Spinlock wieder freigegeben wird. Dadurch bleibt dem Prozessor ein *Kontextwechsel* erspart. Beim Kontextwechsel wird ein Thread blockiert, und alle Informationen, die für das Weiterlaufen benötigt werden, müssen vorher gespeichert werden. Sie können also mit Spinlocks eine Menge Zeit einsparen, diese aber auch verschenken, z. B. wenn Sie durch Spinlocks einige Ihrer Prozessorkerne »lahmlegen«, weil diese plötzlich nicht mehr an die Reihe kommen. Natürlich hängt es auch vom Prozessor ab, ob ein Kontextwechsel schnell oder langsam vor sich geht. Alle Funktionen, die Spinlocks betreffen, beginnen mit dem Präfix `pthread_spin_`.

10.8 Threads abbrechen (canceln)

Wird ein Thread abgebrochen oder beendet, wurde in den bisherigen Programmbeispielen auch immer der komplette Thread beendet. Doch auch hierbei ist es möglich, auf eine Abbruchaufforderung zu reagieren. Sie haben drei Möglichkeiten:

- `PTHREAD_CANCEL_DISABLE`: Damit legen Sie fest, dass ein Thread nicht abbrechbar ist. Dennoch bleiben Abbruchaufforderungen von anderen Threads nicht unbeachtet – sie bleiben bestehen, und es kann gegebenenfalls darauf reagiert werden, wenn man den Thread wieder in einen abbrechbaren Zustand mittels `PTHREAD_CANCEL_ENABLE` 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 wird. Man spricht von einem *verzögerten Abbruchpunkt*. Einen solchen Abbruchpunkt stellen u. a. 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`: Der Thread wird gleich nach dem Eintreffen einer Abbruchaufforderung beendet. Hierbei handelt es sich um einen asynchronen Abbruch.

Die Funktionen, mit denen Sie einem anderen Thread ein Abbruchsignal senden können und/oder die Abbruchmöglichkeiten selbst festlegen sind die folgenden:

```
#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 sofort abbricht oder erst beim nächsten Abbruchpunkt, hängt davon ab, ob `PTHREAD_CANCEL_DEFERRED` (Standard) oder `PTHREAD_CANCEL_ASYNCHRONOUS` verwendet wird. Bevor sich der Thread beendet, werden noch, falls vorher definiert, 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`). Mit dem zweiten Parameter `oldstate` können Sie den zuvor eingestellten Wert für den Thread in dem hier übergebenen Zeiger sichern, oder, falls nicht benötigt, stattdessen `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 im Zeiger der `oldtype` sichern oder `NULL` verwenden.

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

Es folgt ein einfaches Beispiel zu der Funktion `pthread_cancel()`.

```
/* threads13.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) {
    /* Pseudosynchronisation, 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 zu Ende\n", pthread_self());
        pthread_exit ((void *) 0);
    }
}

static void cancel_test2 (void) {
    sleep(1);  // Pseudosynchronisation
    if (zufallszahl <= 25) {
        pthread_cancel (t2);
        printf ("(%d) : Thread %ld beendet %ld\n",
            zufallszahl, pthread_self(), t2);
        printf ("%ld zu Ende\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 ptread_create ...\n");
        exit (EXIT_FAILURE);
    }
    if((pthread_create(&t2, NULL, cancel_test1, NULL))!=0) {
        fprintf (stderr, "Fehler bei ptread_create...\n");
        exit (EXIT_FAILURE);
    }
    if((pthread_create(&t3, NULL, cancel_test2, NULL))!=0) {
        fprintf (stderr, "Fehler bei ptread_create ...\n");
        exit (EXIT_FAILURE);
    }
    pthread_join (t1, NULL);
    pthread_join (t2, NULL);
    pthread_join (t3, NULL);
    return EXIT_SUCCESS;
}
```

Listing 10.13 Der Abbruch von Threads mit pthread_cancel()

In dem letzten Beispiel 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, je nach Beendigung, einer der beiden Threads mit `pthread_cancel()` zweimal 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 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 oder 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 muss man aber von asynchronen Abbrüchen abraten, da diese an jeder Stelle auftreten können. Wird z. B. vorher `pthread_mutex_lock()` aufgerufen und tritt hier der Abbruch ein, nachdem das Mutex gesperrt wurde, hat man schnell einen Deadlock erzeugt. Einen asynchronen Abbruch sollten Sie in der Praxis also 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 enthält (wie reservierter Speicherplatz, 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 realisieren Sie solche Codebereiche, indem Sie den kritischen Abschnitt als unabbrechbar einrichten und gleich danach den alten Zustand wiederherstellen:

```
int oldstate;

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

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

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

Ein einfaches Beispiel hierzu ist das folgende Listing:

```
/* threads14.c */
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>

static void cancel_test (void) {
    int oldstate;

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

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

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

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

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

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

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

Listing 10.14 Zeigt, wie Sie kritische Codeabschnitte einrichten, die nicht unterbrochen werden können.

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads14.c -o threads14 -lpthread
$ ./threads14
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 ausgeben »Thread –1209418832 nach dem kritischen Codeabschnitt« – am besten testen Sie dies, indem Sie das Verändern des Cancel-Status auskommentieren oder eben anstatt PTHREAD_CANCEL_DISABLE die Konstante PTHREAD_CANCEL_ENABLE verwenden.

10.9 Erzeugen von threadspezifischen Daten (TSD)

Vor dem Aufruf einer Funktion werden die Parameter auf dem Stack abgelegt und auch von dort von der Funktion abgeholt. Auch die lokalen Variablen werden zwischengespeichert, sind aber nur innerhalb der Funktion gültig. Da sich alle Threads das Codesegment und das

Datensegment teilen, ist dies auch kein Problem – deshalb können Sie dann Daten für andere Threads auch einfach über globale Strukturvariablen austauschen. Wenn Sie aber andererseits vorhaben, eine Bibliothek für den Multithread-Gebrauch zu schreiben, ist dies nicht mehr möglich, da man hierbei die Argumentzahl nicht mehr so verändern kann, dass auch ältere Programme ohne Threads diese Bibliothek verwenden können. Außerdem weiß man nicht immer, wie viele Threads die Bibliotheksfunktionen nutzen werden, und so kann man keine Aussage darüber machen, wie groß die globalen Daten sein sollen.

Das Ganze vielleicht noch etwas vereinfacht erklärt: Es ist einfach nicht möglich, dass globale und statische Variablen unterschiedliche Werte in den verschiedenen Threads haben (diese teilen sich ja das Datensegment). Aus diesem Grund wurden spezielle Schlüssel eingeführt: die threadspezifischen Daten (kurz TSD). Bei den TSD handelt es sich um eine Art Zeiger, der immer auf die Daten verweist, die dem Thread gehören, der einen entsprechenden Schlüssel benutzt. Beachten Sie allerdings, dass hierbei immer mehrere Threads den gleichen Schlüssel benutzen – nicht jeder Thread bekommt also einen Extraschlüssel.

Jeder Thread bekommt also 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.

Hierzu dienen nun die folgenden Funktionen, mit denen Sie threadspezifische Daten erzeugen bzw. abfragen 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 (als zweites Argument) entweder `NULL` oder der Funktion, um den Speicher der Daten wieder freizugeben (also 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 für den TSD-Schlüssel `key` über den Zeiger `pointer` fest.

Mit der Funktion `pthread_getspecific()` lesen Sie die Daten aus dem TSD-Schlüssel aus.

In dem nun folgenden Beispiel werden `MAX_THREADS` Threads erzeugt, von denen jeder Thread eine Thread-eigene Datei mittels TSD erzeugt. Im Beispiel wird hierbei 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 Log-Dateien verwenden können.

```
/* threads15.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 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 threadspezifischen Dateinamen erzeugen */
    sprintf(th_fpname,"thread%d.thread",(int) pthread_self());
    /* eine Datei öffnen  */
    th_fp = fopen (th_fpname, "w");
    if( th_fp == NULL )
        pthread_exit(NULL);
    /* TSD 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;
}
```

Listing 10.15 Die Verwendung von TSDs

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads15.c -o thread15 -lpthread
$ ./threads15
$ ls *.thread
thread-1209418832.thread  thread-1217811536.thread  thread-1226204240.thread
$ cat thread-1209418832.thread
Thread wurde gestartet ...

Thread ist fertig ...
```

10.10 Mit pthread_once() einen Codeabschnitt auf einmal ausführen

In den letzten Beispielen haben Sie häufig mehrere Threads gestartet. Manchmal tritt aber eine Situation ein, wo man gewisse Vorbereitungen treffen muss – z. B. eine bestimmte Datei anlegen, auf die mehrere Threads zugreifen müssen. Ist es nicht möglich, solche Vorbereitungen direkt beim Start des Haupt-Threads zu treffen, benötigt man einen Mechanismus,

der eine Funktion exakt nur einmal ausführt, egal, wie oft und von welchem Thread aus diese aufgerufen wurde. Es folgt ein einfaches Beispiel für dieses Szenario:

```
/* threads16.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;
}
```

Listing 10.16 Demonstriert die Funktion thread_once().

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc -o threads16.c -o threads16 -lpthread
$ ./threads16
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 threadspezifischen Synchronisationen nachhelfen, aber die Thread-Bibliothek bietet Ihnen mit `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()` müssen Sie eine statische Variable mit der Konstanten `PTHREAD_ONCE_INIT` initialisieren, bevor Sie diese Variable an `pthread_once()` (erster Parameter) übergeben. 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, so dass diese Funktion kein zweites Mal mehr ausgeführt werden kann. Es folgt nun das Beispiel *threads17.c*, nochmals mit `phtread_once()`:

```
/* threads17.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;
}
```

Listing 10.17 Eine Alternative zu Listing 10.16, die nun zuverlässig läuft

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads17.c -o threads17 -lpthread
$ ./threads17
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.

10.11 Thread-safe-Funktionen (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 ihres Codes erlauben. Mit der Einführung der *glibc 2.0* wurden die Linux-Threads in den Bibliotheken implementiert und müssen nicht extra besorgt werden. Somit ist etwa `strcmp()` – auch wenn schon über 15 Jahre alt – thread-safe.

`readdir()` hingegen ist z. B. nicht thread-safe. Das Problem mit `readdir()` ist, wenn mehrere Threads denselben `DIR`-Zeiger verwenden, dass sie immer den aktuellen Rückgabewert vom zuvor erhaltenen Thread überschreiben. Als Alternative für Funktionen, die nicht als thread-safe gelten (auch wenn es nicht sehr viele sind), wurden thread-sichere Schnittstellen, die in

der Regel mit der Endung _r (rekursive Funktionen) gekennzeichnet wurden, eingeführt. Die thread-sichere Alternative zu `readdir()` lautet somit `readdir_r()`.

Die Endung _r zeigt Ihnen außerdem nicht nur an, dass die Funktion thread-sicher ist, sondern auch, dass diese Funktion keinen internen statischen Puffer verwendet (der beim Aufruf derselben Funktion in mehreren Threads jedes Mal überschrieben wird). In Tabelle 10.1 folgt eine kurze Liste zu einigen gängigen rekursiven *F*unktionen, deren genauere Syntax und Verwendung Sie bitte der entsprechenden Manpage entnehmen.

nicht thread-sicher	thread-sicher	Bedeutung
`getlogin`	`getlogin_r`	Login-Name ermitteln
`ttyname`	`ttyname_r`	Terminalpfadname ermitteln
`readdir`	`readdir_r`	Verzeichniseinträge lesen
`strtok`	`strtok_r`	String anhand Tokens zerlegen
`asctime`	`asctime_r`	Zeitfunktion
`ctime`	`ctime_r`	Zeitfunktion
`gmtime`	`gmtime_r`	Zeitfunktion
`localtime`	`localtime_r`	Zeitfunktion
`rand`	`rand_r`	(Pseudo-)Zufallszahlen generieren
`getpwuid`	`getpwuid_r`	Eintrag in */etc/passwd* erfragen (via `UID`)
`getpwnam`	`getpwnam_r`	Eintrag in */etc/passwd* erfragen (via Login-Name)
`getgrgid`	`getgrgid_r`	Eintrag in */etc/group* erfragen (via `GID`)
`getgrnam`	`getgrnam_r`	Eintrag in */etc/group* erfragen (via Gruppenname)

Tabelle 10.1 Nicht thread-sichere Funktionen und die Alternativen

10.12 Threads und Signale

Signale lassen sich auch mit Threads realisieren, nur müssen Sie hierbei Folgendes beachten:

- Signale, die von der Hardware gesendet werden, bekommt immer der Thread, der das Hardwaresignal 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 bestehende Thread-Signalmaske abfragen oder ändern. Im Prinzip entspricht diese Funktion `sigprocmask()`, nur eben auf Threads und nicht auf Prozesse bezogen. Abgesehen von den Signalen `SIGKILL` und `SIGSTOP` können Sie auch hier alle bereits bekannten Signale verwenden. Schlägt die Funktion `pthread_sigmask()` fehl, wird die Signalmaske des Threads nicht verändert.

Wir empfehlen Ihnen, für die Funktion `pthread_sigmask()` das Kapitel über die Signale nochmals durchzulesen, da das Prinzip hier ähnlich wie zwischen Prozessen funktioniert. Als erstes Argument für `how` wird bei `pthread_sigmask()` eine Angabe erwartet, die angibt, wie Sie die Signale verändern wollen. Mögliche Konstanten hierfür sind `SIG_BLOCK`, `SIG_UNBLOCK` und `SIG_SETMASK`. Als zweites Argument ist ein Zeiger auf einen Satz von Signalen nötig, der die aktuelle Signalmaske ergänzt, sie entfernt oder die Signalmaske ganz übernimmt. Hier 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`. Die Signale `SIGKILL`, `SIGTERM` und `SIGSTOP` weisen folgende Besonderheiten auf: Sie gelten weiterhin prozessweit – senden Sie z. B. mit `phtread_kill()` das Signal `SIGKILL` an einen Thread, wird der komplette Prozess beendet, nicht nur der Thread. Ebenso sieht dies mit dem Signal `SIGSTOP` aus – es hält den ganzen Prozess (mit allen laufenden Threads) an, 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.

Es folgt ein einfaches Beispiel, das Ihnen die Verwendung von Signalen in Verbindung mit Threads demonstriert:

```
/* threads18.c */
#include <stdio.h>
#include <pthread.h>
#include <signal.h>

pthread_t tid2;

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

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

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

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

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

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

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

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

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

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

    pthread_exit((void *)99);
}

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

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

    /* Setup Signalhandler 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);
}
```

Listing 10.18 Zeigt, wie Sie Signale zusammen mit Threads verwenden.

Das Programm gibt bei der Ausführung Folgendes aus:

```
$ gcc threads18.c -o threads18 -lpthread
$ ./threads18
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
```

Das letzte Beispiel erzeugt drei Threads (inklusive dem Haupt-Thread). Im Haupt-Thread wird die Signalmaske so eingerichtet, dass alle Signale im Haupt-Thread blockiert werden (SIGBLOCK). Als Nächstes werden Signalhandler für SIGINT und SIGUSR1 eingerichtet. Thread 1 wird nun von den anderen Threads abgehängt (*detached*), und Thread 2 wird nicht von den anderen Threads abgehängt. Dann werden im Haupt-Thread (Thread 0) die Signale SIGINT und SIGUSR1 an Thread 1 gesendet, und der Haupt-Thread beendet sich danach.

Thread 1 (da abgehängt) hebt die Blockierung der Signale auf (SIG_UNBLOCK) und wartet nun auf Signale. In diesem Beispiel wurde 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 und wartet danach (mittels pthread_join()), bis sich Thread 2 beendet. Danach gibt Thread 1 den Exit-Status von Thread 2 aus, bevor sich Thread 1 ebenfalls beendet.

Thread 2 hingegen hebt nur die Blockierung für SIGUSR1 auf – alle anderen Signale werden durch die Weitervererbung des Haupt-Threads weiterhin blockiert. Anschließend wartet Thread 2 auf Signale. Trifft SIGUSR1 ein, wird der Signalhandler ausgeführt und der Thread mit einem Rückgabewert beendet (auf den Thread 1 ja mit pthread_join() wartet).

10.13 Zusammenfassung und Ausblick

Threads sind sicherlich ein ziemliches Schlagwort in den letzten Jahren geworden, und die Programmierer sind in zwei Lager gespalten: in die Befürworter und die Gegner von Threads. Zugegeben, Threads haben auch ihre Macken, und nicht immer ist das »Multitasking auf Teufel komm raus« die Lösung für alle Probleme. Gerade in der Netzwerkprogrammierung sorgen Threads bezüglich der Skalierbarkeit immer wieder für Diskussionsstoff, vor allem, weil durch Threads auch Sicherheitslücken entstehen können. Das Problem besteht fast immer darin, dass es bei Threads häufig ein hartes Limit für gleichzeitig laufende Threads gibt oder auch Speicherlecks auftreten können, die gerne für Angriffe ausgenutzt werden. Ferner wird auch das menschliche Gehirn nicht selten von Multitasking überfordert, besonders im Arbeitsalltag treten deswegen immer wieder auch grobe Fehler auf, die sogar zu Gefahrensituationen führen können. Dies trifft sogar manchmal auf Software zu, die durch Threads eben viel komplexer und auch fehleranfälliger wird. Wie dem auch sei: Trotz aller Bedenken lassen sich die Hersteller von namhafter Software wie den Serveranwendungen BIND und Apache nicht aufhalten, Thread-Unterstützung in ihre Software einzubauen und auch damit zu werben. Auch durch die modernen Mehrkernprozessoren bekommen Threads neuen Aufwind. Da dadurch sogar Profisoftware komplexer (und wiederum fehleranfälliger) wird, bekommen wiederum die Thread-Gegner Aufwind.

Ob Sie nun selbst Threads in Ihre Software einbauen wollen oder nicht, müssen Sie letztendlich selbst entscheiden. Größtenteils hängt es von der Art Software ab, die Sie entwickeln wollen. Wenn Sie z. B. Spiele programmieren, werden Sie ohne Threads nicht weit kommen,

denn Sie werden höchstwahrscheinlich zumindest einen Song im Hintergrund abspielen wollen, um dem Spiel einen bestimmten Charme zu geben. Aber auch, wenn die Steuerung nicht unabhängig vom Rest des Spiels läuft oder sogar »ruckelt«, dann wird Ihnen Ihr Spiel sogar selbst keinen Spaß machen. Wenn Sie allerdings nur einen Hexeditor programmieren wollen, der bestimmte Funktionen enthält, die nur Sie benötigen (z. B. einen 6510-Disassembler für D64-Diskettenabbilder), dann kommen Sie wahrscheinlich auch ohne Threads aus. Hinzu kommt, dass fast 80 Prozent der Algorithmen, die es so gibt, nicht parallelisierbar sind und hier auch die beste Thread-Umgebung nicht viel nutzt. Wir raten Ihnen also deshalb: Setzen Sie Threads sparsam ein und nur dort, wo sie sich wirklich lohnen oder nicht vermeidbar sind (z. B. bei Spielen oder Datenbanken für mehrere Benutzer). Mit diesem Kapitel haben Sie nun auf jeden Fall ein fundiertes Polster an Wissen, so dass Sie Software mit Thread-Unterstützung schreiben können, wenn sie es müssen. Aber wie auch bei fast jedem Thema eines Buches gibt es immer noch einiges mehr, was man zu den Threads noch hätte schreiben können.