Hardwarenahe Programmierung

Eine Einführung

Jürgen Plate, 22. September 2012

Inhaltsverzeichnis

1

E/A-Programmierung unter Linux

In diesem Skript geht es um das Ansprechen von Hardware per Programm. Es gibt grundsätzlich zwei Möglichkeiten, E/A-Schnittstellen anzusprechen: Treiber oder direkter Portzugriff, der unter dem Begriff *User-Mode-Programm* läuft. Kernel-Treiber sind, wie der Name schon sagt, Bestandteil des Kernels. Sobald sie eingebunden sind, können sie von allen Programmen gleichermaßen und ohne besondere Privilegien genutzt werden. Sie werden aus anderen Programmen angesprochen über

- Einträge im */dev*-Verzeichnis,
- das */proc*-Verzeichnis oder
- *ioctl()*-Aufrufe.

Damit folgen die Treiber auch dem UNIX/Linux-Grundsatz „Alles ist Datei". Dieser Vorteil wird durch eine komplexere Programmierung erkauft. So ist z. B. der Zugriff auf Bibliotheksfunktionen eingeschränkt.
Die Alternative besteht in User-Mode-Programmen, die direkt auf E/A-Ports zugreifen und damit auch immer nur mit Root-Privilegien starten müssen (nach erfolgreicher Missetat kann man auf normale User-Privilegien umschalten). Verwendet man dagegen ein User-Mode-Programm für Hardwarezugriffe, gilt:

- die C Library kann benutzt werden,
- Debugging ist einfacher und
- Paging ist möglich.

Die Programmierung ist somit viel einfacher. Aber es gibt auch Nachteile gegenüber einem Treiber:

- Interrupts lassen sich nicht verwenden,
- Performance ist nicht so gut wie bei Kernel-Treibern,
- Verzögerungen entstehen durch den Scheduler und
- Root-Rechte sind für den Zugriff erforderlich.

Bei Embedded Systems, wo nur wenige Prozesse laufen und ein User-Login meist gar nicht möglich ist, überwiegt oft der Vorteil der einfachen Programmierung. Deshalb wird nach einem kurzen Ausflug zum Compiler in diesem Skript nur die User-Mode-Programmierung behandelt.

1.1 Prozesse und Signale

Linux ist bekanntermaßen ein Multitasking-System und kann somit mehrere Aufgaben gleichzeitig erledigen. Wird unter Linux ein Programm ausgeführt, bekommt es eine eindeutige Prozess-Identifikation (PID) zugewiesen, die im Bereich zwischen 1 und 32767 liegt. Anhand dieser PID kann das Betriebssystem in Ausführung befindliche Programme identifizieren und auf diese zugreifen. Beim Beenden eines Programms wird auch seine PID freigegeben und kann später wieder verwendet werden.
Das Erzeugen neuer Prozesse (sogenannte Kindprozesse) aus einem Elternprozess heraus kommt – ebenso wie das Bearbeiten und Senden von Signalen oder die Prozesskommunikation – in „normalen“ Programmen nicht so häufig zur Anwendung. Bei Programmen für ein Steuerungssystem kann es jedoch höchst sinnvoll sein, wenn ein Prozess die von ihm benötigten anderen Prozesse startet und mit diesen kommuniziert. Deshalb soll in diesem Abschnitt auf Linux-Prozesse, Signale und einige ausgewählte Möglichkeiten der Prozesskommunikation eingegangen werden, die später nützlich sein könnten. Die im Folgenden beschriebenen Systemfunktionen sind in der Regel nur bei *User-Mode-Programmen* sinnvoll einsetzbar, bei Treibern muss man andere Wege beschreiten.
Grundsätzlich gilt für ein Multitasking-Betriebssystem, dass jeder Prozess den anderen Prozessen die Möglichkeit geben muss, auch zum Zuge zu kommen. So sind Programme, die bei Mikrocontroller-Anwendungen durchaus üblich sind, bei Multitasking-Systemen kontraproduktiv. Ein typisches Beispiel hierfür ist das „busy waiting“, bei dem in einer Schleife ständig ein E/A-Port abgefragt wird, bis die gewünschten Daten anliegen. Unter Linux sollte hier eine kurze Pause (per `nanosleep()`) eingebaut werden, weil sonst die CPU-Last in unendliche Höhen ansteigt. Ein zweites Beispiel: Sie wollen beim Auftreten irgendwelcher Ereignisse bestimmte Töne ausgeben. Statt nun im eigenen Programm eine Soundausgabe zu programmieren, rufen Sie ein passendes Programm auf. Wenn dies als Kindprozess erfolgt, kann das Elternprogramm ungestört weitermachen – das Abspielen der Sounds erfolgt nebenläufig. Oder man legt einen Prozess komplett schlafen und weckt ihn durch ein Signal wieder auf.

1.1.1 Prozesse

Linux stellt spezielle Funktionen zur Verfügung, mit deren Hilfe man die PID eines Prozesses und die PID seines Elternprozesses abfragen kann. Beide Funktionen sind in der Header-Datei *`unistd.h`* definiert:

```
pid_t getpid(void);
pid_t getppid(void);
```

Die erste Funktion, `getpid()`, liefert die PID des Prozesses zurück, der `getpid()` aufgerufen hat. Die zweite Funktion, `getppid()`, liefert die Eltern-PID des Prozesses. Der Rückgabewert ist jeweils vom Typ `pid_t`, der in einer der in `stdlib.h` eingeschlossenen Header-Dateien als `int` definiert ist. Das folgende Beispiel ermittelt die ID des aktuellen Prozesses und seines Elternprozesses.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(void)
  {
  pid_t  pid;
  pid = getpid();
  printf ("Meine PID = %d\n", pid) ;

  pid = getppid();
  printf ("Meine Eltern-PID = %d\n", pid) ;
  return 0;
  }
```

Mit fork() andere Prozesse starten

Linux verfügt über eine Standardmethode zum Starten anderer Prozesse, die auf der Funktion `fork()` basiert. Ebenso wie `getpid()` liefert `fork()` eine Prozess-ID zurück und ist in der Header-Datei `unistd.h` definiert. Ihr Prototyp lautet `pid_t fork(void)`. Tritt kein Fehler auf, erzeugt `fork()` einen neuen Prozess, der mit dem aufrufenden Prozess identisch ist. Sowohl der alte als auch der neue Prozess werden danach – ab der Anweisung hinter dem `fork()`-Aufruf – parallel ausgeführt. Obwohl beide Prozesse das gleiche Programm ausführen, verfügen sie über eigene Kopien aller Daten und Variablen. Eine dieser Variablen ist der Rückgabewert von `fork()`.

- Im Kindprozess ist der Wert 0.
- Im Elternprozess ist es der Wert der Prozess-ID des Kindprozesses.
- Wenn `fork()` scheitert, wird `-1` zurückgegeben.

Da der Elternprozess eine vollständige Kopie seiner Daten für den Sohn erzeugt, besteht im Anschluss keine Möglichkeit, dass Vater und Sohn über gemeinsame Variablen kommunizieren. Jeder hat von jeder Variablen ja sein eigenes Exemplar. Beispiel: Mit Hilfe von fork() einen neuen Prozess erzeugen.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
  {
  pid_t  pid;
  int x = 22;

  pid = fork();
  if (pid < 0)
    {
    printf("Fehler: fork()-Rsultat %d.\n", pid);
    exit(1);
    }
  if (pid == 0)
    {
    printf("Kind: PID = %u. Eltern-PID = %u\n",
            getpid(), getppid());
    printf("Kind: xalt = %d\n", x);
    x = 11;
    printf("Kind: xneu = %d\n", x);
    sleep(2);
    puts ("Kind: Beendet.");
    exit(42);
    }
  else
    {
    printf("Eltern: PID = %u. Kind-PID = %u\n",
            getpid(), pid);
    puts("Eltern:  60 Sekunden Pause.");
    sleep(60);
    puts("Eltern: wieder wach.");
    printf("Eltern: x = %d\n", x);
    }

  return 0;
  }
```

Die Ausgabe sieht dann etwa folgendermaßen aus:

```
Eltern: PID = 1535. Kind-PID = 1536
Eltern:  60 Sekunden Pause.
Kind: PID = 1536. Eltern-PID = 1535
Kind: xalt = 22
Kind: xneu = 11
Kind: Beendet.
Eltern: wieder wach.
Eltern: x = 22
```

Anhand des Rückgabewertes von `fork()` wird festgestellt, ob ein Fehler aufgetreten ist. Sind keine Fehler aufgetreten, werden zwei Prozesse ausgeführt. Im Kindprozess ist der Wert von `pid` 0, im Elternprozess enthält die Variable eine Prozess-ID im Bereich zwischen 1 und 32767. Die `if`-Anweisung wird von beiden Prozessen ausgewertet. Der Kindprozess führt danach den Block nach dem `if` aus, der Elternprozess den Block nach dem `else`.

Beenden eines Prozesses (exit)

Der Aufruf der C-Bibliotheksroutine `exit(int status)` beendet einen Prozess und sorgt vor dem eigentlichen Beenden dafür, dass Dateien geschlossen werden. Der Parameter `status` dient dazu, dem Vaterprozess beispielsweise Informationen über die ordnungsgemäße Abwicklung des Sohnes zukommen zu lassen. Der Vater kann den Status mit der Systemfunktion `wait()` abfragen. Wenn

ein Anwenderprogramm keine der Exit-Funktionen explizit aufruft, erfolgt dies implizit nach dem Verlassen der `main()`-Routine.
Das obige Programmbeispiel enthält allerdings noch einen Fehler, der in bestimmten Situationen Probleme verursachen kann. Ein Blick in die Prozesstabelle während des Programmlaufs zeigt, dass der Kind-Prozess als erloschen (`defunct`) gemeldet wird. Prozesse verwenden normalerweise zum Beenden die `return`-Anweisung oder rufen die Funktion `exit()` auf. Das Betriebssystem lässt den Prozess so lange in seiner Prozesstabelle eingetragen, bis entweder der Elternprozess des Prozesses den zurückgelieferten Wert liest oder der Elternprozess selbst beendet wird. Im Beispiel oben geschieht dies nicht.
Es gibt mehrere Wege, die Entstehung von solchen Prozessen zu verhindern. Am häufigsten wird die Systemfunktion `pid_t wait(int *status)` verwendet (Header-Datei `sys/wait.h`). Wird die Funktion aufgerufen, hält sie die Ausführung des Elternprozesses so lange an, bis ein Kindprozess beendet wird. Beim Aufruf von „wait“ gibt es drei mögliche Ergebnisse:

- `wait()` liefert -1: der Prozess hat keine Kinder (mehr).
- Der Prozess hat zwar Kinder, aber alle leben noch – dann schläft der Vater, bis der folgende Fall eintritt.
- Der Prozess hat mindestens ein Zombie-Kind: eines davon wird ausgewählt, seine Verwaltungsdaten werden freigegeben und seine PID als Rückgabewert der Funktion abgeliefert. Zuvor werden in den Parameter `status` noch die folgenden Informationen eingetragen:
  - Bits 8 bis 15: der „exit“-Status des Sohnes
  - Bits 0 bis 7: die Nummer des Signals, das den Tod des Sohnes verursacht hat

Wenn Sie an dem Rückgabewert des Kindprozesses nicht interessiert sind, übergeben Sie `wait()` den Wert `NULL`. Das folgende Beispiel zeigt die Anwendung von `wait()`:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
  {
  pid_t  pid;
  int    status;

  pid = fork();
  if (pid < 0)
    {
    printf("Fehler: fork()-Rsultat %d.\n", pid);
    exit(1);
    }
  if (pid == 0)
    {
    printf("Kind: PID = %u. Eltern-PID = %u\n",
            getpid(), getppid());
    sleep(1);
    puts ("Kind: Beendet.");
    exit(42);
    }
  else
    {
     printf("Eltern: PID = %u. Kind-PID = %u\n",
             getpid(), pid);
     puts("Eltern:  10 Sekunden Pause.");
     sleep(10);
     puts("Eltern: wieder wach.")
     pid = wait(&status);
     printf("Eltern: Kind mit PID %u ", pid);
     if (WIFEXITED(status) != 0)
       printf("wurde mit Status %d beendet\n",
               WEXITSTATUS(status));
     else
       printf("wurde mit Fehler beendet.\n");
     }
  return 0;
  }
```

Die `wait()`-Funktion ist offensichtlich recht nützlich, wenn man weiß, dass der Kindprozess bereits beendet wurde. Sollte dies nicht der Fall sein, hält die `wait()`-Funktion den Elternprozess so lange an, bis der Kindprozess beendet wird. Wird dieses Verhalten nicht gewünscht, kann man die Funktion `pid_t waitpid(pid_t pid, int *status, int options)` verwenden, die in der Header-Datei `sys/wait.h` definiert ist. Mit ihr können Sie auf einen bestimmten Prozess (spezifiziert durch seine Prozess-ID) oder einen beliebigen Kindprozess (falls für `pid` der Wert `-1` übergeben wird) warten. Der Exit-Status des Kindprozesses wird im zweiten Argument zurückgeliefert. Dem letzten Parameter, *options*, kann man eine der Konstanten `WNOHANG`, `WUNTRACED` oder `0` (`waitpid()` verhält sich dann wie `wait()`) übergeben. Die erste dieser Konstanten ist die interessanteste, da sie dafür sorgt, dass `waitpid()` sofort mit einem Wert von `0`, einer ungültigen Prozess-ID, zurückkehrt, wenn kein Kindprozess beendet wurde. Der Elternprozess kann dann mit der Ausführung fortfahren und `waitpid()` zu einem späteren Zeitpunkt wieder aufrufen. Das folgende Programm zeigt die Anwendung der Funktion:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(void)
  {
  pid_t  pid;
  int    status;

  pid = fork();
  if (pid < 0)
    {
    printf("Fehler: fork()-Rsultat %d.\n", pid);
    exit(1);
    }
  if (pid == 0)
    {
    printf("Kind: PID = %u. Eltern-PID = %u\n",
           getpid(), getppid());
    sleep(10);
    puts ("Kind: Beendet.");
    exit(-1);
    }
  else
    {
    printf("Eltern: PID = %u. Kind-PID = %u\n",
           getpid(), pid);
    while ((pid = waitpid (-1, &status,  WNOHANG)) == 0)
      {
      printf("Eltern: Kein Kind beendet.");
      puts(" 1 Sekunde Pause.");
      sleep(1);
      }
    printf("Eltern: Kind mit PID %u ", pid);
    if (WIFEXITED(status) != 0)
      printf("wurde mit Status %d beendet\n", WEXITSTATUS(status));
    else
      printf("wurde mit Fehler beendet.\n");
    }
  return 0;
  }
```

Einen Prozess durch einen anderen ersetzen

Die `fork()`-Funktion ist nur ein Teil der Lösung; der zweite Teil besteht darin, einen laufenden Prozess durch einen anderen zu ersetzen. Unter Linux gibt es gleich eine ganze Reihe von Systemfunktionen, die so genannte `exec`-Familie, mit denen man einen Prozess unter Beibehaltung der PID auf ein anderes Programm umschalten kann. In der `exec`-Manpage finden Sie ausführliche Informationen zu den verschiedenen Mitgliedern der `exec`-Familie. Wir werden uns jetzt auf die Funktion `int execl( const char *path, const char *arg, ...)` konzentrieren, die in der Header-Datei `unistd.h` definiert ist. Diese Funktion kehrt nur dann zurück, wenn ein Fehler auftritt. Andernfalls wird der aufrufende Prozess vollständig durch den neuen Prozess ersetzt. Den Programmnamen des Prozesses, der den aufrufenden Prozess ersetzen soll, übergibt man im Argument zu `path`, etwaige Kommandozeilen-Parameter werden danach übergeben. Im Unterschied zu

Funktionen wie `printf()` ist `execl()` darauf angewiesen, dass man als letztes Argument einen `NULL`-Zeiger übergibt, der das Ende der Argumentenliste anzeigt.
Der zweite an `execl()` übergebene Parameter ist nicht der erste Kommandozeilen-Parameter, der an das aufzurufende Programm (spezifiziert in `path`) übergeben wird. Vielmehr ist er der Name, unter dem der neue Prozess in der vom `ps`-Befehl erzeugten Prozessliste aufgeführt wird. Der erste Parameter, der an das (in `path` spezifizierte) Programm übergeben wird, ist also tatsächlich der dritte Parameter von `execl()`. Wenn Sie beispielsweise das Programm `/bin/ls` mit dem Parameter `-lisa` aufrufen wollen und möchten, dass das Programm in der Prozessliste unter dem Namen „`verz`“ aufgerufen wird, würden Sie `execl()` wie folgt aufrufen:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>

int main(void)
  {
  pid_t  pid ;

  pid = getpid();
  printf ("Meine PID = %u\n", pid);
  execl("/bin/ls", "verz", "-lisa", NULL);
  puts("Ein Fehler ist aufgetreten.");
  return 0;
  }
```

Beachten Sie, dass der ursprüngliche Prozess die gleiche Prozess-ID trägt wie später der neue Prozess, der ihn ersetzte.
An dieser Stelle soll nur noch ein weiterer Vertreter der Familie vorgestellt werden: `int execve (char *filename,char *argv[], char *envp[])`. Der Parameter `filename` bezeichnet dann entweder ein ausführbares Programm oder ein Skript, das von einem Interpreter ausgeführt wird. `argv` ist ein Feld von Zeichenketten, das die Aufrufparameter enthält, mit denen das Programm versorgt werden soll. Dabei muss `argv[0]` der Name des Programmes selbst sein. `envp` ist ebenfalls ein Feld von Zeichenketten und enthält die Umgebungsvariablen (mit Inhalt) in der Form „NAME=inhalt“, die dem Programm übergeben werden sollen. Beide Felder müssen mit einem NULL-Zeiger abgeschlossen sein.
Bei Erfolg kehrt die Funktion `execve()` wie `execl()` nicht zurück. Stattdessen wird das aufrufende Programm durch das aufgerufene Programm ersetzt (überschrieben) und dieses gestartet. Das gestartete Programm erhält die gleiche Prozessnummer wie der aufrufende Prozess und erbt in der Regel alle „offenen“ Dateideskriptoren Im Fehlerfall liefert die Funktion den Wert -1 zurück. Beispiel:

```
char *parameter[] = \{ \qq{ls}, \qq{lisa}, NULL \};
char *umgebung[]  = \{ \qq{PATH=/bin:/usr/bin}, \qq{HOME=/root}, NULL \};

execve(\qq{/bin/ls},parameter,umgebung);
printf(\qq{Ooops! ls konnte nicht gestartet werden\backslash n});
```

Die Tabelle der Dateideskriptoren gehört ebenfalls zu den Daten des Prozesses. Hat der Elternprozess „offene“ Dateideskriptoren, hat sie auch der Kindprozess, und beide zeigen auf dieselben Einträge in der Dateitabelle, da diese **nicht** zu den Prozessdaten gehört und damit nicht kopiert wird. Beide Prozesse können somit gemeinsam auf offene Dateien zugreifen und benutzen dabei denselben Dateioffset. Weil das Schreiben aber asynchron erfolgt, ist die Nutzung einer gemeinsamen Datei zur Prozesskommunikation keine besonders gute Idee. Besser werden dazu Pipes (siehe unten) verwendet.

Alles zusammen

Die C-Funktion `system()` kann Kommandos an UNIX übergeben, vereint also `fork()` und `exec..()`. Ihr Eingabeparameter ist eine Stringkonstante (z. B. `system(„ls -l\);`) oder eine Stringvariable (z. B. char kommando[20]; ...; system(kommando);). Dieser Parameter ist das Kommando, das dann von Linux ausgeführt wird. `system()` erzeugt einen eigenen Prozess. Dieser führt das Kommando aus, was aber keinen Effekt für den aufrufenden Prozess hat.

Priorität verändern

Linux ist ein Multitasking-Betriebssystem, aber kein Realzeit-System. Daher kann keine exakte Voraussage darüber getroffen werden, zu welchem Zeitpunkt ein bestimmter Prozess der CPU zugeteilt wird. Je nach Gesamtlast kann das mal kürzer und mal länger dauern. Bei Messprogrammen kann es dazu führen, dass die gewünschten Werte manchmal zu spät eingelesen werden. In so einem Fall kann die Priorität des Messprozesses heraufgesetzt werden. Dazu stellt die Bibliothek zwei Funktionen zum Lesen und Setzen der Scheduler-Parameter und eine Strukturdefinition bereit (Headerfile datsched.h). Von der Struktur wird nur ein Wert, die Priorität, benötigt:

```
int sched_setparam(pid_t pid, const struct sched_param *p);
int sched_getparam(pid_t pid, struct sched_param *p);

  struct sched_param {
    ...
    int sched_priority;
    ...
    };
```

Der erste Parameter, `pid`, enthält die Nummer des Prozesses, dessen Priorität verändert werden soll. Für den aktuellen Prozess wird 0 eingesetzt. Ein Programmfragment zur Änderung der Priorität könnte folgendermaßen aussehen.

```
#define PRIO 10
  ...
struct sched_param s;
  ...
/* Prio lesen */
if (sched_getparam(0, &s) < 0)
  {
  perror("Fehler beim Lesen der Prioritaet\n");
  exit(1);
  }
/* neuen Prio-Wert setzen */
s.sched_priority = PRIO;
if (sched_setparam(0, &s) < 0)
  {
  perror("Fehler beim Aendern der Prioritaet\n");
  exit(1);
  }
  ...
```

1.1.2 Signale

Ein weiteres wichtiges Element der Unix-ähnlichen Betriebssysteme stellen – neben der Möglichkeit, neue Prozesse zu starten oder einen Prozess durch einen anderen Prozess zu ersetzen – die Signale dar, die vielfach auch als Software-Interrupts bezeichnet werden. Signale sind Meldungen, die vom Betriebssystem an einen laufenden Prozess geschickt werden. Manche Signale werden durch Fehler im Programm selbst ausgelöst, andere sind Anforderungen, die der Anwender beispielsweise über die Tastatur auslöst und die vom Betriebssystem an den laufenden Prozess weitergeleitet werden.
Alle Signale, die an ein Programm gesendet werden, verfügen über ein vordefiniertes Verhalten, das durch das Betriebssystem festgelegt wird. Einige Signale, insbesondere die aufgrund irgendwelcher aufgetretener Fehlerbedingungen an das Programm geschickten Signale, führen dazu, dass das Programm beendet und eine „Core Dump"-Datei erzeugt wird.
In der Tabelle 1.1 finden Sie eine Liste der am häufigsten unter Linux ausgelösten Signale. Eine vollständige Liste der für Linux definierten Signale finden Sie in der Header-Datei `/usr/include/bits/signum.h`.
Abgesehen von `SIGSTOP` und `SIGKILL` kann man das Standardverhalten jedes Signals durch Installation einer Signal-Bearbeitungsroutine anpassen. Eine Signal-Bearbeitungsroutine ist eine Funktion, die vom Programmierer implementiert wurde und jedes Mal aufgerufen wird, wenn der Prozess ein entsprechendes Signal empfängt. Abgesehen von `SIGSTOP` und `SIGKILL` können Sie für jedes Signal eine eigene Signal-Bearbeitungsroutine einrichten. Eine Funktion, die als Signal-Bearbeitungsroutine fungieren soll, muss einen einzigen Parameter vom Typ `int` und einen `void`-Rückgabetyp definieren. Wenn ein Prozess ein Signal empfängt, wird die Signal-Bearbeitungsroutine mit der Kennnummer des Signals als Argument aufgerufen.

Tabelle 1.1: Die wichtigsten Signale

Name	Wert	Funktion
SIGHUP	1	Logoff
SIGINT	2	Benutzer-Interrupt (ausgelöst durch [Strg]+[C])
SIGQUIT	3	Benutzeraufforderung zum Beenden (ausgelöst durch [Strg]+[\])
SIGFPE	8	Fließkommafehler, beispielsweise Null-Division
SIGKILL	9	Prozess killen
SIGUSR1	10	Benutzerdefiniertes Signal
SIGSEGV	11	Prozess hat versucht, auf Speicher zuzugreifen, der ihm nicht zugewiesen war
SIGUSR2	12	Weiteres benutzerdefiniertes Signal
SIGALRM	14	Timer (Zeitgeber), der mit der Funktion `alarm()` gesetzt wurde, ist abgelaufen
SIGTERM	15	Aufforderung zum Beenden
SIGCHLD	17	Kindprozess wird aufgefordert, sich zu beenden
SIGCONT	18	Nach einem `SIGSTOP`- oder `SIGTSTP`-Signal fortfahren
SIGSTOP	19	Den Prozess anhalten
SIGTSTP	20	Prozess suspendiert, ausgelöst durch [Strg)+[Z]

Um Signale abfangen und mit einer geeigneten Signal-Bearbeitungsroutine bearbeiten zu können, muss der Programmierer dem Betriebssystem mitteilen, dass es bei jedem Auftreten des betreffenden Signals für das Programm die zugehörige Signal- Bearbeitungsroutine aufrufen soll. Zwei Funktionen gibt es, mit denen man unter Unix eine Signal-Bearbeitungsroutine verändern oder untersuchen kann: `signal()` und `sigaction()`, die beide in der Header-Datei `signal.h` definiert sind. Die zweite Funktion, `sigaction()`, ist die aktuellere und wird auch häufiger eingesetzt. Sie ist wie folgt definiert: `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)`.
Im Erfolgsfall liefert die Funktion `0` zurück, im Fehlerfall `-1`. Der erste Parameter von `sigaction()` ist die Nummer des Signals, dessen Verhalten Sie verändern oder untersuchen wollen. Man übergibt dem Parameter aber nicht die tatsächliche Signal-Nummer, sondern die zugehörige symbolische Konstante – also beispielsweise `SIGINT` statt der Zahl `2`. Der zweite und der dritte Parameter sind Zeiger auf eine `sigaction`-Struktur. Diese Struktur ist in `signal.h` definiert:

```
struct sigaction
  {
  void (*sa_handler)(int);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void);
  }
```

Indem Sie dem zweiten Parameter der `sigaction()`-Funktion einen Zeiger auf eine korrekt eingerichtete `sigaction`-Struktur übergeben, können Sie das Verhalten für das zugehörige Signal verändern. Indem Sie einen Zeiger auf eine solche Struktur als dritten Parameter übergeben, fordern Sie die `sigaction()`-Funktion auf, die Daten, die das aktuelle Verhalten zu dem Signal bestimmen, in die übergebene `sigaction`-Struktur zu kopieren. Beiden Parametern kann man auch `NULL`-Zeiger übergeben.
Es ist also möglich, das aktuelle Verhalten zu ändern sowie das aktuelle Verhalten zu untersuchen, ohne es zu ändern; außerdem das aktuelle Verhalten zu untersuchen und vor dem Ändern abzuspeichern, so dass es später wiederhergestellt werden kann.

- Das Verhalten ändern: `sigaction(SIGINT, &neueaktion, NULL);`
- Das Verhalten untersuchen: `sigaction(SIGINT, NULL, &alteaktion);`
- Kopie des aktuellen Verhaltens anlegen und neues Verhalten einrichten: `sigaction(SIGINT, &neueaktion, &alteaktion);`

Bei dem ersten Element der `sigaction`-Struktur, `sa_handler`, handelt es sich um einen Zeiger auf eine Funktion, die ein `int`-Argument übernimmt. Dieses Element dient als Zeiger auf die Funktion, die als Signal-Bearbeitungsroutine für das zu bearbeitende Signal fungieren soll. Sie können diesem Strukturelement auch die symbolischen Konstanten `SIG_DFL` oder `SIG_IGN` zuweisen. `SIG_DFL` stellt das Standardverhalten für das Signal wieder her, `SIG_IGN` bewirkt, dass das Signal ignoriert wird. Für das `sa_flags`-Element gibt es eine ganze Reihe möglicher Einstellungen, die uns aber nicht weiter interessieren sollen; wir werden das Element in der Regel auf `0` setzen. Über das `sa_mask`-Element kann man angeben, welche anderen Signale während der Ausführung der Signal-Bearbeitungsroutine blockiert werden sollen. Meist wird dieses Strukturelement mit Hilfe der Funktion `sigemptyset()` gesetzt, die in `signal.h` definiert ist als `int sigemptyset(sigset_t *set)`.Das letzte Element der Struktur, `sa_restorer`, wird heute nicht mehr verwendet. Das folgende Listing zeigt ein einfaches Beispiel zur Behandlung von Signalen.

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

static int BEENDEN = 0;

void  sig_bearbeiter(int sig)
  {
  printf("Signal %d empfangen. Programm wird beendet.\n", sig);
  BEENDEN = 1;
  }

int main(void)
  {
  struct sigaction sig_struct;

  sig_struct.sa_handler = sig_bearbeiter;
  sigemptyset(&sig_struct.sa_mask);
  sig_struct.sa_flags = 0;

  if (sigaction(SIGINT,&sig_struct,NULL) != 0)
    {
    puts ("Fehler beim Aufruf von sigaction!") ;
    exit (1);
    }

  puts("Programm gestartet, beenden mit [Strg]+[C].");
  while (BEENDEN == 0)
    {
    puts("Programm läuft.");
    sleep(1);
    }

  puts("Erstmal aufraeumen.");
  sleeep(5);
  puts("Fertig!");
  return 0;
  }
```

Wurde die Signal-Bearbeitungsroutine korrekt eingerichtet, gibt das Programm in eine Meldung aus und tritt in die Schleife des Hauptprogramms ein. Solange die Variable `BEENDEN` gleich `0` ist, gibt die `while`-Schleife die Meldung „`Programm läuft.`\ aus und legt sich jeweils für eine Sekunde schlafen.
Wenn die Signal-Bearbeitungsroutine `sig_bearbeiter()` aufgerufen wird, gibt sie die Meldung „`Signal 2 empfangen. Programm wird beendet.`\ auf den Bildschirm aus und setzt danach den Wert der statischen Variablen `BEENDEN` auf `1`. Allein das führt zum Beeenden und nicht das Betätigen von `[Ctrl]+[C]`. Das Programm könnte auch einfach weiterlaufen und die Benutzerunterbrechung ignorieren. Hier die Ausgabe eines Beispiel-Laufs:

```
Beenden mit [Strg]+[C].
Programm läuft.
Programm läuft.
Programm läuft.
Signal 2 empfangen. Programm wird beendet.
Erstmal aufraeumen.
Fertig!
```

Setzen eines Timers (alarm)

Mit der Systemfunktion `unsigned int alarm(unsigned int seconds);` kann ein „Wecker“ aufgezogen werden, der nach „seconds“ Sekunden das Signal „SIGALRM“ an den aufrufenden Prozess sendet. Wird keine benutzerspezifische Signalreaktion vereinbart, bricht der Prozess nach Empfang des Signals ab. Ein eventuell bereits vorher „aktiver“ Wecker wird zurückgesetzt. Wenn der Rückgabewert der Funktion `alarm()` 0 ist, dann war zuvor kein „Wecker“ aktiv. Wenn der Wert ungleich 0 ist, gibt er an, nach wie viel Sekunden ein zuvor eingestellter Wecker abgelaufen wäre.

Warten auf ein Signal (pause)

Die Systemfunktion `int pause(void);` bewirkt, dass der aufrufende Prozess in den Schlafzustand versetzt wird und dort so lange verharrt, bis irgend ein Signal eintrifft. Damit ist allerdings noch nicht festgelegt, welche Reaktion im Anschluss erfolgen soll. Ohne entsprechende Maßnahmen kehrt „pause“ bei den meisten Signalen nicht zurück, sondern bricht das Programm ab. Davon abweichendes Verhalten kann mit der Systemfunktion „signal“ erreicht werden.

Vereinbarung einer Signalreaktion (signal)

Die Vereinbarung einer Reaktion auf ein bestimmtes Signal erfolgt mit: `void (*signal(int signum, void (*handler)(int)))(int);` Weil dieser Funktionsprototyp etwas verwirrend ist, hier ein Beispiel:

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
#include <ctype.h>

void tick (int dummy) /* nur Wecker neu aufziehen */
  { alarm(1); }

void beenden(int signal_nummer)
  { // Signal-Bearbeitungsroutine
  char c;
  if (signal_nummer == SIGINT)
    {
    printf("Prozess wirklich beenden ?");
    c = getchar();
    if (c == 'j' || c == 'J') exit(1);
    else                      return;
    }
  else
    {
    printf("unerwartetes Signal %d\n");
    exit(1);
    }
  }

int main(void)
  {
  signal(SIGINT,beenden);
  signal(SIGALRM,tick);
  alarm(1);  // Wecker aufziehen
  for (;;)
    {
    pause();  // auf Signal warten
    putchar('.');
    };
  }
```

Das Hauptprogramm plant für das Signal „SIGINT“ (z. B. Drücken von Ctrl-C) die Bearbeitungsroutine „beenden“ und für das Signal „SIGALRM“ (Timer-Signal) die Routine „tick“ ein. Im Anschluss daran wird der Timer mit „alarm(1)“ auf 1 Sekunde gesetzt.
Nun folgt eine Endlosschleife, in der mit „pause()“ auf ein beliebiges Signal gewartet wird. Falls der Benutzer innerhalb der nächsten Sekunde nichts tut, wird beim Eintreffen von „SIGALRM“ die Funktion „tick“ aufgerufen, in der nur der Timer neu gesetzt wird. Die Folge ist eine regelmäßige Ausgabe der Zeichenfolge „tick“ auf dem Terminal.
Wird allerdings Ctrl-C betätigt (was das Signal „SIGINT“ auslöst), dann wird „beenden“ aufgerufen und der Benutzer gefragt, ob er das Programm tatsächlich abbrechen möchte. Wenn er dann nicht mit „j“ oder „y“ antwortet, wird das Programm einfach fortgesetzt.

Die Signal-Bearbeitungsroutinen müssen void-Funktionen mit einem int-Parameter sein. Dieser repräsentiert die Nummer des Signals, das den Aufruf verursacht hat. Dadurch ist es möglich, **eine** Routine für verschiedene Signale einzuplanen und in der Routine die auslösende Ursache zu ermitteln. Innerhalb einer Signal-Bearbeitungsroutine wird ein erneutes Eintreffen des gleichen Signals ignoriert. In unserem Beispiel heißt dies, dass das wiederholte Drücken von Ctrl-C (während des Dialoges) keine Wirkung hat.
Statt des Namens einer Bearbeitungsfunktion kann an der Position des zweiten Parameters von „signal“ auch eine von zwei vordefinierten Konstanten angegeben werden:

- `SIG_IGN` bedeutet, dass das Signal ignoriert werden soll.
- `SIG_DFL` bedeutet, dass die Standardbearbeitung für das Signal eingestellt werden soll.

Schreibt man beispielsweise `signal(SIGINT,SIG_IGN);`, wird das Drücken von Ctrl-C grundsätzlich ignoriert. Möchte man wissen, welche Signalbearbeitung vor dem Aufruf von „signal“ eingestellt ist, muss man den Rückgabewert von „signal“ auswerten. Dieser repräsentiert die „alte“ Signalreaktion. Damit kann beispielsweise eine Signalbearbeitung vorübergehend modifiziert werden, um sie im Anschluss wieder auf den vorigen Mechanismus zurückzusetzen. Mit `void (*old)(int) = signal(SIGINT,beenden);` holt man die „alte“ Signalreaktion, und mit `signal(SIGINT,old);` wird sie wieder eingesetzt.

Senden eines Signals (kill)

Die Systemfunktion `int kill(int pid, int signal);` wird verwendet, um einem Prozess ein Signal zuzusenden. Wenn der Parameter „pid“ größer als 0 ist, wird das Signal dem Prozess mit der entsprechenden Nummer zugestellt. Wenn „pid“ gleich 0 ist, wird es allen Prozessen übermittelt, die zur gleichen Gruppe wie der Aufrufer gehören. Im Falle „pid“ gleich -1 werden alle existierenden Prozesse (außer „init“) adressiert, und bei „pid“ kleiner -1 wird es an eine andere Gruppe geschickt, deren Nummer gleich dem Absolutwert von „pid“ ist. Wenn der aufrufende Prozess keine Superuser-Rechte besitzt, kann ein Signal nur an einen Prozess desselben Benutzers geschickt werden.
Zum gezielten Abbrechen eines Prozesses sollte möglichst das Signal „SIGINT“ benutzt werden. Der Empfänger kann dasselbe „abfangen“ und hat damit die Chance, vor dem eigentlichen Beenden Aufräumarbeiten durchzuführen (z. B. Daten abspeichern). Allerdings kann er sich auch dafür entscheiden, das Signal zu ignorieren, wodurch „SIGINT“ keine sichere Maßnahme zum „Killen“ eines Prozesses darstellt.
Als Anwendung und zur Demonstration soll ein Reaktionstest dienen, der mit der normalen Tastatur auskommt:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <math.h>
#include <unistd.h>

clock_t start, ende, differenz, record = 1000000;

/* Signal-Handler-Routinen */
void  strgbackslash_faenger(int sig)
  {
  signal(SIGQUIT, SIG_IGN);
  printf("Die schnellste STRG-C Tastenfolge dauerte %7.3f Sekunden\n",
          record/(double)CLOCKS_PER_SEC);
  exit(0);
  }

void  strgc_faenger(int sig)
  {
  /* Fuer die Dauer dieser Funktionsausfuehrung muessen weitere  */
  /* SIGINT-Signale ignoriert werden.                            */
  signal(SIGINT, SIG_IGN);

  /* Gebrauchte Zeit berechnen und ausgeben */
  ende = clock();
  differenz = ende - start;
  printf("Benoetigte Zeit:  %10.3f Sekunden\n",
          differenz/(double)CLOCKS_PER_SEC);
```

```
  if (differenz < record)
    {
    record = differenz;
    printf("Neuer Rekord: %10.3f Sekunden\n",
            record/(double)CLOCKS_PER_SEC);
    }
  sleep(rand()%2+1);
  printf("\nDruecke so schnell wie moeglich STRG-C\n");
  start = clock();

  /* Signal-Handler wieder fuer SIGINT installieren */
  signal(SIGINT, SIG_IGN);
  if (signal(SIGINT, strgc_faenger) == SIG_ERR)
    {
    printf("Fehler: SIGINT-Handler nicht installiert!\n");
    exit(1);
    }
  }

int main(void)
  {
  /* Startwert für Pseude-Zufallszahlen erzeugen und   */
  /* Signal Handler installieren                       */
  srand(time(NULL));
  if (signal(SIGQUIT, strgbackslash_faenger) == SIG_ERR)
    {
    printf("Fehler: SIGQUIT-Handler nicht installiert!\n");
    exit(1);
    }
  signal(SIGINT, SIG_IGN);
  sleep(rand()%2+1);
  printf("Bitte merk Dir: Beenden mit STRG-\\\n");
  printf("\nDruecke so schnell wie moeglich STRG-C\n");
  if (signal(SIGINT, strgc_faenger) == SIG_ERR)
    {
    printf("Fehler: SIGINT-Handler nicht installiert!\n");
    exit(1);
    }
  start = clock();
  while (1); /* busy waiting - normalerweise "boese" */
  return(0);
  }
```

Prozess-Synchronisation durch Signale

Bei der Synchronisation/Steuerung von Prozessen durch Signale kann man die Reihenfolge festlegen, in der bestimmte Prozesse bearbeitet werden. Die Funktionen `signal()`, `pause()` und `kill()` werden hierfür verwendet. Durch die Funktion `signal()` wird ein Signal-Handler, der beim Eintreffen des Signals ausgeführt wird, an das Signal gebunden. Bei größeren Programmen, die mehrere Prozesse haben, wird es allerdings schwierig, den Überblick zu behalten. Bei dieser Methode wird die Reihenfolge festgelegt in der Prozesse bzw. Teile von Prozessen ausgeführt werden. Die „parallele“ Bearbeitung von Prozessen wird dadurch eingeschränkt. Ein weiteres Problem bei der Arbeit mit Signalen ist die Tatsache, dass Signale nicht vom System gespeichert werden. Erhält ein Prozess ein Signal, bevor dieser selbst die Funktion `pause()` aufgerufen hat, geht dieses Signal verloren und der Prozess wartet, wenn er später die Funktion `pause()` aufruft, vergeblich auf ein Signal. Beispiel für die Synchronisation durch Signale:

```
#include<stdio.h>
#include<signal.h>
#include <sys/types.h>
#include <unistd.h>

void sighand(void) /* Signal Handler wird beim Eintreffen */
  {               /* des Signales SIGUSR1 ausgefuehrt */
  signal(SIGUSR1,&sighand);
  /* Hier wird die Bindung des Signals */
  /* an den Signal Handler sighand() erneuert.  */

  puts("Signalhandler aktiv!\n");
  }

int main(void)
```

```
{
 /* PIDs der Soehne */
int vater_pid, prozess1_pid, prozess2_pid;

signal (SIGUSR1,&sighand);         /* Bindung des Signals SIGUSR1 */
                                   /* an den Signalhandler  */
if ((prozess1_pid = fork()) == 0)  /* Sohnprozess 1 erzeugen */
  {                                /* und starten */
  vater_pid = getppid();           /* Sohnprozess erfragt die */
                                   /* PID des Vaters */
  printf("Sohn 1 laeuft\n");
  sleep(3);
  kill(vater_pid,SIGUSR1);        /* Dem Vaterprozess wird das */
                                   /* Signal SIGUSR1 gesendet */
  printf("Sohn 1 terminiert\n");
  exit(0);
  }

if ((prozess2_pid = fork()) == 0) /* Sohnprozess 2 wird */
  {                                /* erzeugt und gestartet */
  printf("Sohn 2 gestartet - wartet\n");
  pause();                         /* Sohnprozess 2 wartet */
                                   /* auf ein Signal */

  printf("Sohn 2 terminiert\n");
  exit(0);
  }
printf("Vater wartet auf Signal von Sohn 1\n");
pause();
printf("Vater: Signal von Sohn 1, kille Sohn 2\n");
kill(prozess2_pid,SIGUSR1);
putchar('\n');
return(0);
}
```

Die Ausgabe des Programms:

```
Vater wartet auf Signal von Sohn 1
Sohn 1 laeuft
Sohn 2 gestartet - wartet
Sohn 1 terminiert
Signalhandler aktiv!
Vater: Signal von Sohn 1, kille Sohn 2
Signalhandler aktiv!
Sohn 2 terminiert
```

1.1.3 Prozesskommunikation mit Pipes

Eine Pipe ist ein Datenkanal, der wie eine Datei behandelt wird. Ein Prozess schreibt die Daten in diese Pipe, und ein anderer Prozess kann diese Daten in der Reihenfolge auslesen, in der sie vom anderen Prozess geschrieben wurden. Eine Pipe in Unix/Linux ist unidirektional, so dass die Daten nur in eine Richtung übermittelt werden. Eine Pipe ist aus Sicht des Prozesses eine Datei, auf die er sequentiell schreibt oder von der er sequentiell liest. Ein Prozess, der aus einer leeren Pipe lesen will, muss warten, bis von einem anderen Prozess in die Pipe geschrieben wurde. Ein Prozess, der in eine Pipe schreiben will, muss warten, wenn der Pipe-Buffer voll ist.

Unbenannte Pipe

Die (unbenannte) Pipe ist eingeschränkt. Ihre Lebensdauer ist abhängig von der Lebensdauer der Prozesse, die mit ihr arbeiten. Sind all diese Prozesse beendet, wird die Pipe gelöscht. Die Kommunikation über eine unbenannte Pipe ist nur für Prozesse möglich, die im gleichen Prozessbaum liegen. Mit dem `pipe()`-Aufruf besitzt ein Prozess zunächst eine Pipe zu sich selbst, aus der er mit Filehandle 0 Daten lesen kann. Mit dem Filehandle 1 kann er Daten in diese Pipe schreiben. Sinnvoll wird das erst, wenn der Vaterprozess durch einen `fork()`-Aufruf einen Sohnprozess erzeugt, der mit dem Vaterprozess Daten austauscht. Dieser Sohnprozess erbt die Pipe seines Vaters. Die Richtung des Datenstromes wird dadurch beeinflusst, welcher Prozess die Lese- bzw. Schreibseite der Pipe schließt.
Sollen zwei Söhne durch eine unbenannte Pipe miteinander kommunizieren, müssen folgende Schritte ausgeführt werden:

1. Vaterprozess richtet durch den Aufruf `pipe()` eine Pipe ein.

2. Der Vaterprozess erzeugt mit `fork()` einen „Schreib-Sohn“.
3. Der Vaterprozess schließt die Schreibseite der Pipe.
4. Der „Schreib-Sohn“ schließt die Leseseite der Pipe.
5. Der Vaterprozess erzeugt nun mittels `fork()` einen „Lese-Sohn“.
6. Der Vaterprozess schließt nun auch die Leseseite der Pipe.
7. Dieser „Lese-Sohn“ schließt die Schreibseite der Pipe.

Die so erstellte Pipe bildet nun eine Verbindung zwischen dem ersten (Schreibprozess) und dem zweiten Sohn (Leseprozess). Der Vaterprozess hat nach dem Erstellen keinen Einfluss auf die Pipe, da er die Lese- und Schreibseite geschlossen hat.
Das folgende Beispiel demonstriert, wie eine Shell prinzipiell vorgeht, wenn sie eine „Prozess-Pipeline“ ausführt. Angenommen, das Kommando `ls | sort` wird eingegeben. Dann läuft, vereinfacht dargestellt, der folgende Mechanismus ab:

```
int Pipe[2];
int status;
char *parls[]   = { "/bin/ls", NULL };
char *parsort[] = { "/usr/bin/sort", NULL };

int main(void)
  {
  ...
  pipe(Pipe);           // Pipe erzeugen
  if (fork() == 0)      // erster Sohn: "ls"
    {
    dup2(Pipe[1],1);    // Pipeausgabe->Standardausgabe
    close(Pipe[0]);     // Pipeeingabe nicht benötigt
    execve("/bin/ls",parls,NULL);
    }
  else
    {
    if (fork() == 0)    // zweiter Sohn: "sort"
      {
      dup2(Pipe[0],0);  // Pipeeingabe->Standardeingabe
      close(Pipe[1]);   // Pipeausgabe nicht benötigt
      execve("/usr/bin/sort",parsort,NULL);
      }
    else                // Vater (Shell)
      {
      close(Pipe[0]);
      close(Pipe[1]);
      wait(&status);
      wait(&status);
      }
    }
  ...
  }
```

Benannte Pipe

Eine benannte Pipe (named pipe) besitzt einen Geräteeintrag vom Typ FIFO (First In First Out) und hat einen Namen, mit dem sie von jedem Prozess durch `open()` angesprochen werden kann. Eine benannte Pipe wird vom System nicht automatisch gelöscht, wenn alle Prozesse beendet sind. Durch den Aufruf `unlink()` muss der Anwender die benannte Pipe innerhalb eines Prozesses selber löschen. Für benannte Pipes gibt es folgende Schnittstellenfunktionen:

- `close` schließt ein Schreib- oder Leseende einer Pipe.

  Prototyp: `int close(int fd);`

  Parameter: `fd`: Lese- bzw. Schreibdeskriptor einer Pipe

- `mkfifo` erzeugt eine benannte Pipe.

  Prototyp: `int mkfifo (char *name, int mode);`

Parameter: `*name`: Name bzw. Pfad der Pipe, `mode`: Bitmaske für Zugriffsrechte auf die Pipe. Die Position und Bedeutung dieser Bits sind so wie beim numerischen `chmod`-Kommando (z. B. 0755 [führende Null wg. Oktalangabe]).

Rückgabewert: 0 bei erfolgreicher Ausführung, sonst -1.

- `open` öffnet eine Pipe bzw. Datei.

  Prototyp: open (char *name, int flag, int mode);

  Parameter: `*name`: Name bzw. Pfad der Pipe; `flag`: Bitmuster für Zugriff auf die Pipe (`O_RDONLY` Lesezugriff, `O_WRONLY` Schreibzugriff, `O_NONBLOCK` Prozessverhalten). Wird `O_NONBLOCK` nicht angegeben (Normalfall), blockiert der Leseprozess, bis ein anderer Prozess die Pipe zum Schreiben öffnet und umgekehrt.

  Rückgabewert: -1 bei Fehler oder Dateideskriptor für die Pipe.

- `pipe` erzeugt eine unbenannte Pipe.

  Prototyp: `int ipe (int fd[2]);`

  Parameter: `fd[2]`: zwei Dateideskriptoren, die zurückgegeben werden, wobei `fd[0]` der Dateideskriptor für die Leseseite und `fd[1]` Dateideskriptor für die Schreibseite der Pipe ist.

- `read` liest Daten aus einer Pipe. Ist die Pipe leer, blockiert die Funktion.

  Prototyp: `int read (int fd, char *outbuf, unsigned bytes);`

  Parameter: `fd`: Diskriptor der Pipe; `*outbuf`: Zeiger auf den Speicherbereich, in dem die Daten gespeichert werden und `bytes`: Maximale Anzahl der Bytes, die gelesen werden.

  Rückgabewert: Anzahl der tatsächlich gelesenen Bytes, -1 bei einem Fehler und 0, wenn die Schreibseite der Pipe geschlossen wurde.

- `write` schreibt Daten in eine Pipe. Ist der Pipe-Buffer voll, blockiert diese Funktion.

  Prototyp: `int write (int fd, char *outbuf, unsigned bytes);`

  Parameter: `fd`: Deskriptor der Pipe; `*outbuf`: Zeiger auf den Speicherbereich, in dem die zu schreibenden Daten stehen und `bytes`: Anzahl der Bytes, die geschrieben werden.

Das folgende Beispiel zeigt die Anwendung einer Named Pipe für zwei getrennte Prozesse. In Linux können benannte Pipes auch für die Kommunikation zwischen Prozessen eingesetzt werden, die nicht miteinander „verwandt“ sind:

```
/* Empfaenger */
#include<stdio.h>
#include<signal.h>
#include <unistd.h>
#include<fcntl.h>

int main(void)
  {
  int ein;               /* Hilfsvariable fuer Programmstart */
  int hilf;
  char outbuffer[2];  /* Buffer zum Auslesen der Pipe */
  int fd;              /* Dateideskriptor fuer Pipe */
  int gelesen;           /* speichert die Anzahl der gelesenen Bytes */

  printf("Empfaengerprozess wurde gestartet\n\n");
  do
    {
    fd = open("TESTPIPE",O_RDONLY);  /* Oeffnen der Pipe zum Lesen */
    if (fd == -1) printf("Prozess zum Schreiben in die Pipe starten!\n");
    sleep(2);
    }
  while (fd == -1);
  do
    {
    gelesen = read(fd,outbuffer,2); /* 2 Bytes werden ausgelesen */
    if (gelesen != 0) printf("Lese %c aus der Pipe\n",outbuffer[0]);
    sleep(2);
    }
  while (gelesen > 0);
  unlink("TESTPIPE"); /* benannte Pipe wird geloescht */
```

```
  return(0);
  }

/* Sender */
#include<stdlib.h>
#include <unistd.h>
#include<stdio.h>
#include<fcntl.h>

int main(void)
  {
  int hilf;
  char inbuffer[2];  /* Buffer zum Schreiben in die Pipe */
  int fd;            /* Dateideskriptor fuer Pipe */

  system("mkfifo TESTPIPE -m 666");   /* benannte Pipe wird erzeugt */
  printf("Sendeprozess wurde gestartet\n\n");

  mkfifo("TESTPIPE",0666);           /* benannte Pipe wird erzeugt */
  fd = open("TESTPIPE",O_WRONLY);    /* Oeffnen der Pipe zum */
  for(hilf=0; hilf<10; hilf++)       /* Zaehler zum Schreiben */
    {
    inbuffer[0] = (int)'0' + hilf;
    inbuffer[1] = '\0';
    write(fd,inbuffer,2);            /* 2 Bytes werden geschrieben */
    printf("Schreibe %c in die Pipe\n",inbuffer[0]);
    sleep(1);
    }
  return(0);
  }
```

1.1.4 Programme schlafen legen

Für Pausen definierter Länge kennt die C-Bibliothek drei Funktionen, `sleep()`, `usleep` und `nanosleep`, wobei die beiden ersten in `unistd.h` und die letzte in `time.h` definiert sind. Da Linux kein Echtzeitbetriebssystem ist, werden die angegebenen Zeiten nicht exakt eingehalten, sie gelten gewissermaßen als minimale Wartezeit. Die Sleep-Funktionen haben eine wichtige Aufgabe: Sie erlauben anderen Prozessen die Nutzung der CPU. Wenn Sie beispielsweise in einer Programmschleife einen E/A-Port abfragen, nimmt dieses Programm eine recht hohe CPU-Last auf. Durch „Schlafpausen“ innerhalb der Schleife wird die Belastung des Systems auf ein vernünftiges Maß reduziert. Wir werden die Funktionen auch benötigen, um beispielsweise Impulse definierter Länge auf einer Ausgangsleitung zu realisieren.
Derzeit ist die kürzeste Pausenzeit bei Linux 1/HZ Sekunden (z. B. 10 ms bei Linux/i386 und 1 ms bei Linux/Alpha). Deshalb wird jede Zeitangabe auf ein Vielfaches dieses Wertes aufgerundet.

```
unsigned int sleep(unsigned int seconds);
```

Diese Funktion lässt den Prozess für die angegebene Anzahl von Sekunden schlafen. Die Funktion gibt 0 zurück, wenn die Zeit abgelaufen ist, oder die Anzahl der Restsekunden, falls ein Interrupt aufgetreten ist.

```
void usleep(unsigned long usec);
```

Diese Funktion ermöglicht eine Pause von `usec` Mikrosekunden (mit Aufrunden, wie oben erwähnt). Es wird empfohlen, nicht mehr `usleep()` zu verwenden, sondern stattdessen `nanosleep()`. In manchen Beispielen werden Sie trotzdem `usleep()` antreffen (meist in der „Millisekundenvariante“ `usleep(x * 1000L)`).

```
int nanosleep(const struct timespec *req, struct timespec *rem);
```

Die Funktion „schläft“ mindestens die angegebene Zeit. Wird sie durch einen Interrupt (Signal) früher beendet, gibt sie -1 zurück und schreibt die verbleibende Zeit in die Structure `rem` zurück (außer, es wurde für `rem` NULL angegeben). Dieser Wert kann dann für die Fortsetzung der Warteaktion verwendet werden.
In `timespec` wird die Wartezeit in Nanosekunden festgehalten (auch wenn das System nur längere Intervalle erlaubt). Sie hat folgenden Aufbau:

```
struct timespec
  {
    time_t  tv_sec;         /* seconds */
    long    tv_nsec;        /* nanoseconds */
  };
```

Der Wert im Nanosekundenfeld muss im Intervall 0 bis 999999999 liegen. Im Gegensatz zu `sleep()` und `usleep()` beeinflusst `nanosleep()` keine Signale, entspricht dem POSIX-Standard und hat die feinste Auflösung. Das folgende Programmfragment wartet 100 ms:

```
#include <time.h>
 ...
struct timespec zeit;
 ...
zeit.tv_sec = 0;              /* seconds */
zeit.tv_nsec = 100*1000000;  /* nanoseconds */
nanosleep(zeit,NULL);
 ...
```

Werden innerhalb eines Treibers Zeitverzögerungen benötigt, so wird der den Treiber aufrufende Prozess für die entsprechende Zeit in den Zustand „wartend“ versetzt. Dazu kann die Funktion `schedule_timeout` verwendet werden. Diese Funktion legt die Treiberinstanz schlafen. Nach Ablauf der angegebenen Zeit wird eine Funktion aufgerufen, die die Treiberinstanz wieder aufweckt.

1.2 User-Mode-Programmierung

Insbesondere Programmierer aus der einstigen DOS-Welt werden bei der ersten Berührung mit Linux zunächst die Funktionen *inportb()* und *outportb()* für die einfache Ansteuerung externer Hardware vermissen. Analoge Befehle stehen aber auch unter Linux zur Verfügung. Bei Anwendungen im User-Space können – wie schon erwähnt – beliebig komplexe C-Bibliotheken hinzugelinkt werden, und die Fehlersuche kann mit herkömmlichen Debuggern erfolgen. Außerdem ist es problemlos möglich, ein blockierendes Programm im User-Space zur Terminierung zu zwingen. Fehlerhafte Implementierungen können (auf Grund von Speicherschutzmechanismen) kaum Schaden am laufenden System anrichten und beeinflussen deshalb nur selten die Stabilität von Linux.
Programme mit direkter Hardware-Schreib-/Leseberechtigung dürfen aus Sicherheitsgründen nur von *root* ausgeführt werden, da sonst jeder beliebige Benutzer den Rechner bei falscher E/A-Programmierung lahmlegen könnte. Das dennoch potentiell auftretende Sicherheitsrisiko bei Eigenentwicklungen spielt bei *Embedded Systems* oder Heim-Linux-PCs eine geringere Rolle. Direkte E/A-Adressierung durch Programme im User-Space eignet sich für einfache und zeitunkritische Mess- und Steueraufgaben, etwa zur Programmierung von Relaissteckkarten, AD-Wandlern oder alphanumerischen LC-Displays (z. B. über den Parallelport).

1.2.1 Programme mit Root-Rechten ausstatten

Grundsätzlich ist es nur dem Betriebssystem, also dem Kernel, und dem Benutzer *root* gestattet, auf die Hardware zuzugreifen. Doch wie ist es dann möglich, dass jeder normale Benutzer z. B. sein Passwort ändern kann, obwohl die Datei `/etc/shadow` nur von *root* geöffnet werden darf? Wieso kann man drucken, obwohl das Gerät `/dev/lp0` keine Schreiberlaubnis für gewöhnliche User besitzt?
Unter Linux gibt es neben den Zugriffsrechten Lesen (R), Schreiben (W) und Ausführen (X) auch das Recht „set user ID“ (SUID). Ist dieses Bit bei einer ausführbaren Datei gesetzt, so erhält der ausführende Prozess für die Dauer der Ausführung die Benutzer-ID des Dateieigentümers (normalerweise laufen alle Programme mit den Rechten des aufrufenden Users). Gehört also ein Programm dem Benutzer *root*, werden dem ausführenden Prozess dessen Privilegien verliehen. So klappt das beispielsweise mit dem Programm `passwd`. Mit einem eigenen Steuerprogramm sieht das ungefähr folgendermaßen aus:

```
> gcc -O2 -c relaisbox.c -o relaisbox
> su
Kennwort:
# chown root:root relaisbox
# chmod 4711 relaisbox
# exit
```

Wenn Sie das Programm `relaisbox` jetzt als normaler Benutzer starten, läuft es dennoch mit *root*-Privilegien und kann entsprechendes Chaos anrichten. Die Berechtigung 4711 bedeutet, dass *root* die Datei lesen und schreiben darf, alle anderen Benutzer (*group* und *others*) sie nur ausführen dürfen und die SUID-Berechtigung gesetzt ist.

1.2.2 UID und GID

Jeder Prozess besitzt eine *reale* User Id (UID) und eine *reale* Group Id (GID), die vom Elternprozess geerbt werden. Für die Festlegung von Zugriffsrechten ebenso wichtig sind die *effektive* UID und die *effektive* GID. Normalerweise sind *reale* und *effektive* UID/GID identisch. Ausnahmen ergeben sich bei den Programmen, bei denen die SUID-Berechtigung (oder analog die SGID-Berechtigung) gesetzt ist. Ist SUID gesetzt, wird beim Ausführen des Programms die effektive UID des Prozesses gleich der des Datei-Eigentümers und damit i. a. ungleich der des Users, der das Programm startet. Die dabei entstehende *effektive* UID wird gesichert. Zur Bestimmung von UID und GIG gibt es die folgenden Bibliotheksfunktionen:

```
pid_t getuid();   /* Gibt reale User-Id zurueck */
pid_t getgid();   /* Gibt reale Gruppen-Id zurueck */
pid_t geteuid();  /* Gibt effektive User-Id zurueck */
pid_t getegid();  /* Gibt effektive Gruppen-Id zurueck */
```

Das Resultat ist jeweils die gefragte Größe. Zum Setzen von UID und GID stehen zwei Bibliotheksfunktionen zur Verfügung:

```
pid_t setuid(pid_t uid);   /* Setzen der User-Id */
pid_t setgid(pid_t gid);   /* Setzen der Gruppen-Id */
```

Das einzige Argument ist die gewünschte UID bzw. GID. Der Funktionswert 0 zeigt Erfolg, -1 Misserfolg an. Das Ergebnis des `setuid()`-Aufrufs hängt von der *effektiven* UID des aufrufenden Prozesses ab. Ist sie 0 (Super-User, root), ändern sich die *reale* und die *effektive* UID auf den angebenen Wert. Ist sie ungleich 0, so setzt das System die *effektive* UID gleich der *realen* UID des Prozesses, wenn uid mit dieser übereinstimmt oder wenn uid gleich der gesicherten *effektiven* UID ist. Andernfalls endet die Funktion mit einem Fehler. Das folgende Programm wendet die genannten Systemaufrufe an:

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(void)
  {
  int uid, euid, pid, res;

  pid = getpid();
  uid = getuid();       /* reale UID ermitteln */
  euid = geteuid();     /* effektive UID ermitteln */
  printf("Der Prozess %d hat UID %d und EUID %d\n", pid, uid, euid);

  res = setuid(uid);
  printf("Result: setuid(%d): %d\n", uid, res);
  printf("Nun ist UID %d und EUID %d\n", getuid(), geteuid());

  res = setuid(euid);
  printf("Result: setuid(%d): %d\n", euid, res);
  printf("Nun ist UID %d und EUID %d\n", getuid(), geteuid());

  return 0;
  }
```

Das Programm wird kompiliert und mit SUID-Berechtigung ausgestattet. Es zeigt, wie man zwischen *effektiver* und *realer* User-Id hin- und herschalten kann. Nur der Weg zur User-Id 0 zurück geht nicht, wie folgendes Beispiel zeigt. Die obere Ausgabe des folgenden Listings zeigt das Umschalten zwischen zwei Usern (UID 100 und UID 101), wobei das Programm dem User mit der Id 101 gehört. Dann wird das Programm SUID root gesetzt. Das untere Protokoll zeigt, dass zu root kein Weg zurückführt.

```
> ./utest
Der Prozess 20512 hat UID 100 und EUID 101
Result: setuid(100): 0
Nun ist UID 100 und EUID 100
Result: setuid(101): 0
Nun ist UID 100 und EUID 101

>su
Passwort:
# chown root utest
# chmod u+s utest
# ./utest
Der Prozess 20596 hat UID 100 und EUID 0
```

```
Result: setuid(100): 0
Nun ist UID 100 und EUID 100
Result: setuid(0): -1
Nun ist UID 100 und EUID 100
```

Auf diese Weise können Sie Programme schreiben, die zuerst mit Root-Berechtigung starten und alle privilegierten Aufgaben ausführen (etwa Portzugriffe) und dann eine unprivilegierte Benutzeridentität annehmen. Mit der können sie dann kaum noch Schaden anrichten.

1.2.3 Zugriff auf E/A-Ports im User-Space

Ein C-Programm muss verschiedene Funktionen aufrufen, um auf Ports verschiedener Größe zuzugreifen. Um die Portabilität zu erhöhen, täuscht der Kernel bei Computer-Architekturen, die in den Speicher abgebildete E/A-Register (memory mapped i/o) besitzen, die Port-E/A vor, indem er Port-Adressen auf Speicheradressen abbildet. Die hier geschilderten Funktionen erlauben den Zugriff auf die Hardware nicht nur für „normale“ Programme, sondern werden natürlich auch innerhalb von Treibermodulen eingesetzt.

Port-Zugriffe

Die architekturabhängige Header-Datei *asm/io.h* definiert die unten aufgeführten Inline-Funktionen für die E/A-Portzugriffe. Mit *unsigned* ohne weitere Typangabe wird eine architekturabhängige Definition verwendet, bei der es auf den genauen Typ nicht ankommt. Die Funktionen sind fast immer portabel, weil der Compiler die Werte automatisch während der Zuweisung mit dem Cast-Operator umwandelt. Die Funktionen

```
unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
```

lesen oder schreiben Byte-Ports (8 Bit). Das Argument *port* ist auf manchen Plattformen als *unsigned long* und auf anderen als *unsigned short* definiert. Der Rückgabewert von `inb()` unterscheidet sich auch auf den einzelnen Hardware-Architekturen.

```
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
```

Diese Funktionen greifen auf 16-Bit-Ports (Wort) zu.

```
unsigned inl(unsigned port);
void outl(unsigned longword, unsigned port);
```

Diese Funktionen greifen auf 32-Bit-Ports zu. *longword* ist je nach Plattform entweder als *unsigned long* oder als *unsigned int* definiert.
Die oben genannten Funktionen sind hauptsächlich für die Verwendung in Gerätetreibern vorgesehen, können aber auch vom User-Space aus aufgerufen werden – zumindest auf PCs. Neben den oben erwähnten Funktionen existieren noch weitere; eine vollständige Übersicht liefert die Tabelle 1.2.
Die GNU-C-Bibliothek definiert diese Funktionen in *sys/io.h*. Damit die Funktionen im User-Space-Code verwendet werden können, müssen aber folgende Bedingungen erfüllt sein:

- Das Programm muss mit der Optimierungs-Option -O (oder -O2) kompiliert werden, um die Expansion der Inline-Funktionen (Makros) zu erzwingen.
- Mittels `ioperm()` und `iopl()` muss die Erlaubnis beantragt werden, E/A-Operationen auf Ports benutzen zu dürfen. `ioperm()` holt sich diese Erlaubnis für spezielle Ports, `iopl()` für den gesamten E/A-Adressraum. Beide Funktionen sind Intel-spezifisch.
- Das Programm muss unter root-Berechtigung laufen, da sonst `ioperm()` oder `iopl()` beim Aufruf die Arbeit verweigern (siehe vorhergehenden Abschnitt).

```
int ioperm (unsigned long from, unsigned long num, int turn\_on);
```

setzt die Bits für die Steuerung des Zugriffsrechts auf E/A-Ports für *num* Bytes beginnend mit der Portadresse *from* auf den Wert *turn_on* (0 oder 1). Der Gebrauch benötigt wie gesagt Superuser-Rechte. Es können nur die ersten 1024 E/A-Ports von `ioperm()` freigegeben werden. Bei weiteren Ports muss die Funktion `iopl()` verwendet werden. Die Zugriffsrechte werden nicht von `fork()`, jedoch von `exec()` vererbt. Bei Erfolg wird 0, im Fehlerfall -1 zurückgegeben und die Variable `errno` entsprechend gesetzt. Wenn die Systemaufrufe `ioperm()` und `iopl()` auf der Host-Plattform nicht zur Verfügung stehen, kann man vom User-Space aus trotzdem noch auf die E/A-Ports zugreifen, indem man die Gerätedatei `/dev/port` verwendet (sehr plattformabhängig!).

Tabelle 1.2: Makros für den Zugriff auf E/A-Ports

Makro	Beschreibung
inb(Adresse) inw(Adresse) inl(Adresse) insb(Adresse,*Puffer,Count) insw(Adresse,*Puffer,Count) insl(Adresse,*Puffer,Count)	Der Inhalt des E/A-Bereichs an der angegegebenen Adresse wird ausgelesen. Dabei wird je nach Endung des Makros ein Byte (b), Word (w) oder Long-Word (l) zurückgegeben, ggf. auch komplette Strings (s).
outb(Wert,Adresse) outw(Wert,Adresse) outl(Wert,Adresse) outsb(Adresse,*Puffer,Count) outsw(Adresse,*Puffer,Count) outsl(Adresse,*Puffer,Count)	Der Ausgabe-Wert wird an die E/A-Adresse geschrieben. Dabei wird Wert je nach Endung des Makros als Byte (b), Word (w) oder Long-Word (1) interpretiert, ggf. auch komplette Strings (s).
inb_p(Adresse) inw_p(Adresse) inl_p (Adresse) outb_p(Wert,Adresse) outw_p(Wert,Adresse) outl_p(Wert,Adresse)	Im Unterschied zu den Makros ohne `_p` wird bei diesen Funktionen die Ausführung etwas verzögert, um langsamer Hardware die Möglichkeit der „Erholung“ zu geben.

Ein erstes Beispiel

```
#include <asm/io.h>

int main(int argc, char* argv[])
  {
  int base = atoi(argv[1]);
  int value = atoi(argv[2]);
  ioperm(base,1,1);
  outb(value,base);
  ioperm(base,1,0);
  return 0;
  }
```

Bei der Ausführung dieses Beispiels auf der Kommandozeile müssen zwei Argumente übergeben werden (z. B. `iop 888 85`). Der erste Wert ist die Basis-Adresse eines freien Parallelports. Der Wert 888 (= 0x378) sollte hier für einen normalen PC brauchbar sein, solange nicht der Druckeranschluss per Jumper oder BIOS in einen anderen E/A-Adressraum verlegt wurde. Das zweite Argument wird als Bitmuster verwendet, welches an die Datenleitungen des Parallelports angelegt wird. Der verwendete E/A-Bereich wird durch den Aufruf von `ioperm()` freigeschaltet. Ab der Basisadresse wird der Zugriff auf genau einen E/A-Port erlaubt, da der dritte Parameter von `ioperm()` 1 ist. In der folgenden Programmzeile wird das angegebenen Bitmuster (hier „01010101“) auf den Datenleitungen des Druckeranschlusses ausgegeben. Schließlich wird der direkte Zugriff auf die Ports mit Hilfe einer 0 als letztem Parameter von `ioperm()` wieder verboten. Das Programm wird mit folgender Befehlszeile übersetzt:

```
gcc -O2 -o iop iop.c
```

Das Programm kann nun z.B. mit `./iop 888 255` gestartet werden – unter der Voraussetzung, dass man als `root` eingeloggt ist. Sie können beim Binary auch das Setuid-Bit für root setzen (chmod 4711 iop), wenn Sie gerne gefährlicher leben und mit Ihrer Hardware spielen möchten, ohne explizite Zugriffsrechte zu erwerben. Sie sollten sich aber vor Augen halten, dass Sie immer ein wenig mit dem Feuer spielen. Wenn Sie mal aus Versehen eine falsche Adresse erwischen, kann das nicht nur zum Blockieren des Systems führen, sondern auch zum globalen Datenverlust (sofern Sie nämlich die Portadresse des Festplattencontrollers erwischen). Die beste Lösung ist es, einen separaten PC für die Experimente zu verwenden.

Hinweis

Manche alte ISA-Bus-Steckkarten können oft die Daten nicht schnell genug verarbeiten. Zwei unmittelbar aufeinanderfolgende `outb()`-Zugriffe auf dasselbe Gerät können dann zu unerwünschter (bzw. unbewusster) Fehlprogrammierung führen. Gegebenenfalls muss eine Wartezeit im Programm eingebaut werden (z. B. mit `nanoleep()`) oder Sie verwenden die Funktionen, die auf „`_p`“ enden.

Datentypen für den Hardwarezugriff

Die realen Längen (in Bit) der unterschiedlichen Datentypen in C sind nicht festgelegt. Der Zugriff auf die Register der Hardware erfolgt jedoch in durch die Hardware festgelegten Längen von ein, zwei, vier oder acht Byte. Derartige Datentypen werden sowohl für die Programme im User-Space als auch für die Treibermodule in Headerdateien festgelegt. Für die User-Space-Schnittstelle gibt es die Datentypen (zwei Unterstreichungszeichen zu Beginn):

`__u8`	unsigned 8 Bit	`__s8`	signed 8 Bit
`__u16`	unsigned 16 Bit	`__s16`	signed 16 Bit
`__u32`	unsigned 32 Bit	`__s32`	signed 32 Bit
`__u64`	unsigned 64 Bit	`__s64`	signed 64 Bit

Im Treiber stehen die gleichen Datentypen jedoch ohne die beiden Underlines zur Verfügung:

`u8`	unsigned 8 Bit	`s8`	signed 8 Bit
`u16`	unsigned 16 Bit	`s16`	signed 16 Bit
`u32`	unsigned 32 Bit	`s32`	signed 32 Bit
`u64`	unsigned 64 Bit	`s64`	signed 64 Bit

2

Compiler, Linker, Libraries

2.1 Programme installieren

Wenn Sie mit gängigen Distributionen arbeiten, installieren Sie zumeist nur fertig kompilierte Programme (sogenannte Binärpakete). Für unsere Anwendungen müssen Sie in der Regel den Quellcode herunterladen oder selbst verfassen und dann das Programm selbst kompilieren. Deshalb gebe ich Ihnen dazu einige einführende Tipps.
Praktisch alle Linux-Programme verwenden dieselben Standardfunktionen, beispielsweise zum Zugriff auf Dateien, zur Ausgabe am Bildschirm, zur Unterstützung von X etc. Es wäre sinnlos, wenn jedes noch so kleine Programm all diese Funktionen unmittelbar im Code enthalten würde – riesige Programmdateien wären die Folge. Stattdessen bauen die meisten Linux-Programme auf sogenannten *shared libraries* auf: Bei der Ausführung eines Programms werden automatisch auch die erforderlichen Libraries (Bibliotheken) geladen. Der Vorteil: Wenn mehrere Programme Funktionen derselben Library nutzen, muss diese nur einmal geladen werden.
Bibliotheken spielen eine zentrale Rolle dabei, ob und welche Programme auf Ihrem Rechner ausgeführt werden können. Fehlt auch nur eine einzige Bibliothek (bzw. steht sie in einer zu alten Version zur Verfügung), kommt es beim Programmstart sofort zu einer Fehlermeldung.
Kompilierte Programme können nur dann ausgeführt werden, wenn die dazu passenden Bibliotheken installiert sind und auch gefunden werden. Mit dem Kommando `ldd` kann man feststellen, welche Bibliotheken von einem Programm benötigt werden. `ldd` wird als Parameter der vollständige Dateiname des Programms übergeben. Als Reaktion listet `ldd` alle Libraries auf, die das Programm benötigt. Außerdem wird angegeben, wo sich eine passende Library befindet und welche Libraries fehlen bzw. nur in einer veralteten Version zur Verfügung stehen. Wenn `ldd` das Ergebnis *not a dynamic executable* liefert, handelt es sich um ein Programm, das alle erforderlichen Bibliotheken bereits enthält (ein statisch gelinktes Programm).
Beim Start eines Programms ist der sogenannte *runtime linker* `ld.so` dafür zuständig, alle Bibliotheken zu finden und zu laden. `ld.so` berücksichtigt dabei alle in der Umgebungsvariablen *`LD_LIBRARY_PATH`* enthaltenen Verzeichnisse. Außerdem wertet der Linker die Datei `/etc/ld.so.cache` aus. Dabei handelt es sich um eine Binärdatei mit allen relevanten Bibliotheksdaten (Versionsnummern, Zugriffspfade etc.).
Bevor Sie eigene Programme kompilieren können, müssen einige Voraussetzungen erfüllt sein:

- Die *GNU Compiler Collection* (Pakete `gcc` und `gcc-c++`) muss installiert sein. Diese Pakete enthalten kompiler für C und C++.
- Hilfswerkzeuge wie `make`, `automake`, `autoconf` etc. müssen installiert sein. Diese Programme sind für die Konfiguration und Durchführung des Kompilationsprozesses erforderlich.
- Die Entwickler-Versionen diverser Bibliotheken müssen installiert sein (Pakete `glibc-devel` und `libxxx-devel`).

Im Internet finden Sie den Quellcode zumeist in komprimierten TAR-Archiven (Kennung **.tar.gz* oder **.tgz* oder **.tar.bz2*). Nach dem Download entpacken Sie den Code mittels `tar xzf name.tar.gz` oder `tar xjf name.tar.bz2` in ein lokales Verzeichnis. Ausgehend von diesem Verzeichnis finden Sie üblicherweise für jedes Quellcodepaket mehrere Dateien. *SOURCES/name.tar.xxx* enthält den eigentlichen Code als TAR-Archiv. *SOURCES/name-xxx.patch* oder *SOURCES/name.dif* enthält distributionsspezifische Veränderungen am ursprünglichen Code. Falls Sie die Codedateien entsprechend ändern (patchen) möchten, führen Sie das Kommando `patch < name.dif/patch` aus. *SPECS/name.spec* enthält eine Paketbeschreibung, die auch für die Erstellung von RPM-Paketen dient.
Zum Kompilieren und Installieren von Programmen sind drei Kommandos erforderlich: `.configure`, `make` und `make install`. Die drei Kommandos werden im Folgenden näher beschrieben. Vorausgesetzt wird, dass Sie sich im Quellcodeverzeichnis befinden.
`configure` ist ein Skript, das testet, ob alle erforderlichen Programme und Bibliotheken verfügbar sind. Nach dieser Systemanalyse adaptiert `configure` die Datei *Makefile*, die alle Kommandos enthält, um die diversen Codedateien zu kompilieren und zu linken. Bei manchen (zumeist eher kleineren Programmen) kann es sein, dass es das Script *configure* nicht gibt. In diesem Fall führen Sie sofort `make` aus.
`make` löst die Verarbeitung der kompile- und Link-Kommandos aus. Sie sehen nun (manchmal schier endlose) Nachrichten und Warnungen der verschiedenen kompiler-Läufe über das Konsolenfenster huschen. Solange kein Fehler auftritt, können Sie diese Meldungen getrost ignorieren. Als Ergebnis sollte sich im Quellcodeverzeichnis nun die ausführbare Datei *name* befinden. In vielen Fällen können Sie das Programm nun sofort starten (Kommando `./name`) und testen (sofern nicht eine spezielle Konfiguration erforderlich ist oder das Programm nur durch Init-V-Scripts korrekt gestartet werden kann).
Der letzte Schritt besteht darin, das Programm allen Benutzern zugänglich zu machen. Dazu müssen die Programm- und eventuell auch Bibliotheksdateien in öffentlich zugängliche Verzeichnisse kopiert werden. Das erfordert *root*-Rechte. Vor der Ausführung von `make install` sollten Sie sicherstellen, dass das betreffende Programm nicht schon installiert ist! Wenn dies der Fall ist, sollte es vorher deinstalliert werden.
Während des Übersetzens können vielfältige Probleme auftreten. Am wahrscheinlichsten ist, dass irgendwelche Compiler-Hilfswerkzeuge oder zum Kompilieren notwendige Bibliotheken (die Entwickler-Versionen dieser Bibliotheken) fehlen. Diese Probleme werden in der Regel bereits durch `configure` festgestellt und lassen sich meist relativ leicht beheben, indem das fehlende Paket einfach installiert wird.
Mit dem Kommando `gcc -Wall -o name name.c` kompilieren Sie das Programm „name.c“. Das Ergebnis ist das Binärprogramm „name“. Die Option `-Wall` schaltet alle Warnungen des Compilers ein, und die Option `-o name` benennt die fertige Binärdatei. Manchmal müssen wir mit „`-O`“ noch die Optimierung setzen, weil sonst einige Makros nicht richtig expandiert werden. Danach sind gegebenenfalls noch die passenden Eigentümer- und Zugriffsrechte für das Binärprogramm zu setzen.

2.2 Compiler und Linker

Wie von Anfang an bekannt, kann unser C-Compiler nicht nur den Quellcode übersetzen, sondern auch einzelne Binärobjekte zu einen ausführbaren Programm zusammenbinden (linken). Mittels `gcc -c foo.c` erzeugt man ein Object-File `foo.o` und mittels `gcc -o foo foo.o` kann daraus ein ausführbares Programm erzeugt werden. Das Ganze funktioniert auch mit mehreren Quell- und Object-Dateien, z. B.:

```
gcc -c foo.c            Erzeugen der Object-Dateien (*.o)
gcc -c bar.c
gcc -c test.c
gcc -o go *.o           Erzeugen des Executables (go)
```

Die Option `-c` weist den `gcc` an, nur zu kompilieren und die Option `-o` sorgt für das Linken. Der gcc hat noch ein paar wichtige Optionen zu bieten:

`-c`	nur übersetzen, nicht linken
`-Idir`	Include-Dateien in dir suchen
`-Wall`	alle Warnungen aktivieren
`-g`	Debugging-Symbole erzeugen
`-o file`	Binärcode in file schreiben

`-Olevel`	Optimierungen einschalten, z. B. `-O2`
`-foption`	generelle Compiler-Optionen, z. B. `-ffast-math`
`-llib`	Bibliothek linken, z. B. bindet `-lm` die `libm.o` ein
`-Ldir`	Bibliotheken in `dir` suchen

Daneben gibt es noch zahlreiche andere Optionen. `gcc` bildet also ein bequemes Frontend zum Linker und zu anderen Utilities, die während der Kompilierung aufgerufen werden.
Es ist auch gar nicht schwierig, eigene Bibliotheken zu erzeugen. Wenn Sie eine Reihe von Funktionen geschrieben haben, können Sie aus dieser Gruppe von Quelldateien Objektdateien erzeugen und aus diesen Objektdateien eine Bibliothek erstellen. Um eine Bibliothek zu erzeugen, müssen Sie nur die Quelldateien (die **keine** Funktion `main()` enthalten dürfen) kompilieren:

```
gcc -c foo.c bar.c
```

Damit erhalten Sie `foo.o` und `bar.o`. Danach generieren Sie aus diesen Objektdateien die Bibliothek. Eine Bibliothek ist einfach nur eine Archivdatei, die mit dem Befehl `ar` erzeugt wird. Ich nenne meine Bibliothek `libfoobar.a`:

```
ar -q libfoobar.a foo.o bar.o
```

Wenn Sie eine Bibliothek aktualisieren möchten, können Sie die Objectdateien mit dem Parameter `-r` ersetzen. Die Option `-t` listet den Inhalt der Bibliothek auf. Als letzten Schritt erstellen Sie einen Index zu dieser Bibliothek, damit der Linker die Routinen darin finden kann. Dazu rufen Sie `ranlib` auf:

```
ranlib libfoobar.a
```

Dieser Befehl fügt zusätzliche Informationen in die Bibliothek selbst ein; es wird keine eigenständige Indexdatei erzeugt. Sie könnten die letzten beiden Schritte mit `ar` und `ranlib` auch kombinieren, indem Sie bei `ar` den Schalter `-s` angeben:

```
ar -rs libfoobar.a foo.o bar.o
```

Mit `libfoobar.a` haben Sie nun eine statische Library, die Ihre Routinen enthält. Bevor Sie Programme mit dieser Bibliothek binden können, müssen Sie noch eine Header-Datei schreiben, die den Inhalt der Bibliothek beschreibt (siehe unten im Abschnitt über Make).
Wie kompilieren wir Programme, die auf die soeben fertiggestellte Bibliothek samt Header-Datei zugreifen? Zunächst müssen wir beide an einer Stelle abspeichern, wo der Compiler sie finden kann. Viele Programmierer legen eigene Bibliotheken im Unterverzeichnis `lib` ihres Home-Verzeichnisses ab und eigene Include-Dateien unter`include`. Dann ruft man den Compiler folgendermaßen auf:

```
gcc -Iinclude -Llib -o doit doit.c -lfoobar
```

Mit der Option `-I` weisen Sie den `gcc` an, das Verzeichnis `include` in den Include-Pfad einzufügen, in dem `gcc` nach Include-Dateien sucht. Die Option `-L` funktioniert ähnlich für die Bibliotheken.
Mit der Anweisung `-lfoobar` der Kommandozeile wird der Linker angewiesen, die Bibliothek `libfoobar.a` einzubinden (sofern sie im Library-Pfad zu finden ist). Das `lib` am Anfang des Dateinamens wird für Bibliotheken automatisch angenommen.
Sie sollten den Schalter `ll` auf der Befehlszeile jedesmal benutzen, wenn Sie andere als die Standardbibliotheken einbinden wollen. Wenn Sie beispielsweise mathematische Routinen aus `math.h` benutzen möchten, sollten Sie am Ende der `gcc`-Befehlszeile `-lm` anhängen, womit `libm` eingebunden wird. Bedenken Sie aber, daß die Reihenfolge der `-l`-Optionen von Bedeutung sein kann. Wenn beispielswesie Ihre Bibliothek `libfoobar` Routinen aus `libm` aufruft, dann muss in der Befehlszeile `-lm` hinter `-lfoobar` stehen:

```
gcc -Iinclude -Llib -o doit doit.c -lfoobar -lm
```

Damit zwingen Sie den Linker, `libm` nach `libfoobar` zu binden; dabei können die noch nicht aufgelösten Verweise in `libfoobar` bearbeitet werden.
Per Voreinstellung werden Bibliotheken in verschiedenen Verzeichnissen gesucht; das wichtigste davon ist `/usr/lib`. Dort finden Sie auch Dateien mit der Endung `.so`, ggf. mit angehängter Versionsnummer. Dabei verbergen sich hinter den `.a`-Dateien die statischen Bibliotheken, wogegen die `.so`-Dateien dynamisch ladbare Bibliotheken sind, die sowohl den zur Laufzeit dazu gelinkten Code als auch den Stub-Code enthalten, den der Laufzeit-Linker (`ld.so`) benötigt, um die dynamische Bibliothek zu finden. Die Ziffer hinter *`.so.`* gibt die Hauptversionsnummer an. In den typischen Library-Verzeichnissen (zumeist */lib*, */usr/lib*, */usr/local/lib*, */usr/X11R6/lib* und */opt/lib*) befinden sich oft Links von der Library-Hauptversion auf die tatsächlich installierte Library.
Der Linker versucht per Voreinstellung Shared Libraries einzubinden. Es gibt allerdings auch Situationen, in denen die statischen Bibliotheken benutzt werden sollen. Sie können mit dem Schalter `-static` beim `gcc` das Einbinden von statischen Bibliotheken festlegen.

2.3 Make

Unter den C-Quellen (Dateien mit den Endungen „.C2“ bzw. „.h“) ergibt sich bei einem Projekt ein System von Abhängigkeiten. Wenn bestimmte Quell-Dateien modifiziert wurden, müssen einige, aber in der Regel nicht alle, Module neu erstellt werden. Bei mehreren tausend Dateien (wie z. B. beim Linux-Kernel) wäre es aber unsinnig bei jeder kleinen Korrektur alle Dateien neu zu übersetzen. Ebenso mühsam ist, manuell diejenigen Dateien herauszufinden, die sich geändert haben. Man braucht also ein Werkzeug, das diese Abhängigkeiten erkennt und jeweils nur die nötigen Module neu übersetzt.
Das Make-Utility leistet das Gewünschte. Dem Make-Utility muss ein sogenanntes Makefile (Datei-Name: „Makefile“, diese Schreibweise ist wichtig) bereit gestellt werden, das die Abhängigkeiten eines Projekts beschreibt: Module, Versionen (Debug-Version, Release-Version) sowie andere Informationen. Make erkennt anhand der Modifikations-Zeit der Dateien was sich geändert hat. Das Makefile beschreibt die Abhängigkeiten mittels „Targets“ (engl. Ziele). Das Makefile bestehen aus einer Menge von Regeln zur Steuerung der Übersetzung. Sein Aufbau ist:

```
Ziel: Quelle Quelle ...
      Shellkommando
      Shellkommando
      ...
```

Das Ziel gibt meist an, welche Datei erzeugt wird. Falls eine der Quellen neuer ist als das Ziel, wird das Ziel aktualisiert. Aktualität und Vorhandensein der Quellen werden vorher rekursiv sichergestellt. Die Shellkommandos (durch Tabulatorzeichen eingeleitet!) erzeugen das Ziel aus den Quellen. Dazu ein Beispiel (Bild 2.1):

```
go: bar.o foo.o test.o
    gcc bar.o foo.o test.o -lm -lglib -o go
bar.o: bar.c
    gcc -Wall -O2 -c bar.c
foo.o: foo.c
    gcc -Wall -O2 -c foo.c
test.o: test.c
    gcc -Wall -O2 -c test.c
clean:
    rm -rf *.o
```

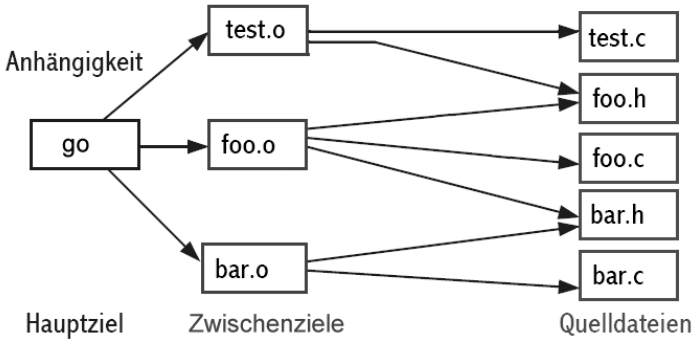

Bild 2.1: Abhängigkeiten des Beispiels

Ein Eintrag eines Makefiles sieht formal folgendermaßen aus:

```
Target_Name: <Dateien oder Sub-Targets von den dieses abhängt>
   <Tab-Zeichen>   <Regel zum erstellen dieses Targets>
```

Eine gemeine Falle für Anfänger ist, dass die zweite Zeile mit einem Tabulatorzeichen anfangen muss, und nicht mit Leerzeichen.
Beim Aufruf von make kann angegeben werden, welches Target man erstellen möchte. Make erzeugt dann alle hierfür nötigen Subtargets. Wird make ohne Parameter gestartet, wird das erste (oberste) Target aus dem Makefile erstellt. Im Beispiel oben werden dann `foo.o, bar.o, test.o` und `go` erzeugt.
Es gibt auch Targets, die keine Abhängigkeiten haben. Im obigen Makefile haben wir ein zusätzliches Target „clean“, das dazu da ist, alle o-Files wieder sauber zu löschen. Somit kann man mit dem Kommando `make clean` wieder aufräumen.
Viele targets können nicht (wie oben) durch einen einzigen Befehl erzeugt werden, sondern benötigen mehrere Kommandos. In diesem Fall folgen auf die Zeile mit den Abhängigkeiten einfach mehrere

Zeilen, die alle mit <tab> beginnen. Auch die Abhängigkeiten für ein target dürfen auf mehrere Zeilen verteilt sein.
Im Beispiel oben ist „go“ gleichzeitig ein Target-Name und ein Datei-Name. make sieht darin keinen Unterschied. Auch die beiden Dateien `foo.c` und `bar.c` sind für make nichts anderes als Targets. Diese hängen von keinen anderen Targets ab, sind also immer aktuell, und es gibt auch keine Regel, um sie zu erzeugen. Würde nun beispielsweise die Datei `bar.c` nicht existieren, würde make feststellen, daß es das target `bar.c`, welches für `bar.o` benötigt wird, nicht erzeugen kann. Die Fehlermeldung lautet dementsprechend:

```
make: *** No rule to make target ‘bar.c’. Stop.
```

`make` kennt noch viele weitere Möglichkeiten, von denen hier nur einige besprochen werden. Es ist u. a. möglich, in Makefiles Variablen (eigentlich sind es Makros) zu definieren und zu benutzen. Normalerweise verwendet man Großbuchstaben. Gebräuchlich sind beispielsweise folgende Variablen:

```
CC        der Compiler
CFLAGS    Compiler-Optionen
LDFLAGS   Linker-Optionen
```

Auf den Inhalt dieser Variablen greift man dann mit `$(CC)`, `$(CFLAGS)` bzw. `$(LDFLAGS)` zu. Ein einfaches Makefile eines Programmes namens `go`, welches aus einer Reihe von Objekt-Dateien zusammengelinkt werden soll, könnte also so aussehen:

```
VERSION = 3.02
CC      = /usr/bin/gcc
CFLAGS  = -Wall -g -DVERSION=\"$(VERSION)\"
LDFLAGS = -lm -lglib

OBJ = datei1.o datei2.o datei3.o datei4.o datei5.o

all: $(OBJ)
        $(CC) $(CFLAGS) -o go $(OBJ) $(LDFLAGS)

%.o: %.c
        $(CC) $(CFLAGS) -c $<
```

Das %-Zeichen ist hier der Platzhalter für Regelmengen. Das Defaulttarget ist hier `all`, das erzeugte ausführbare Programm `go`. Dieses hängt von allen Objekt-Dateien ab. Beim Linken werden zwei Libraries dazugelinkt. An diesem Beispiel sieht man, dass eine Shell die Befehle ausführt: Ohne die Backslashes in `\"$(VERSION)\"` würde diese nämlich die Anführungzeichen entfernen. Die Versionsnummer soll dem C-Präprozessor aber als konstante Zeichenkette übergeben werden. Interessant ist die letzte Zeile, wo der Compiler angewiesen wird, eine Quelle namens $< zu übersetzen. Bei $< handelt es sich um eines der sogenannten automatischen Makros, deren es unter anderem folgende gibt:

```
$@     Ziel
$<     erste Quelle
$^     alle Quellen
$?     Quellen, die neuer sind als das Ziel
```

Dazu ein Beispiel:

```
go:   $(OBJECTS)
      $(CC) $^ -lm -lglib -o $@
%.o: %.c
      $(CC) $(CFLAGS) -c $<
```

2.4 Module

Module sind die einzige höhere Abstraktion, die C bietet, man sollte also reichlich Gebrauch davon machen. Wie zerlegt man nun einem Programm in Module? Betrachten Sie dazu das folgende einfache Beispiel:

```
#include <stdio.h>

/* Funktions-Prototypen */
int foo(int a, int b);
void bar(int a, int *result);

int main(void)
  {
```

```
  int X = 2,Y = 3, res;
  res = foo(X, Y);
  printf("%d\n", res);
  bar(X, &res);
  printf("%d\n", res);
  return(0);
  }

int foo(int a, int b)
  {
  return a + b;
  }

void bar(int a, int *result)
  {
  *result = a;
  }
```

Es sollen nun die Funktionen `foo()` und `bar()` in ein Modul ausgelagert werden. Es ergibt sich erstens ein Programm-Testmodul `test.c`:

```
/* Testmodul */
#include <stdio.h>

/* Funktions-Prototypen */
int foo(int a, int b);
void bar(int a, int *result);

int main(void)
  {
  int X = 2,Y = 3, res;
  res = foo(X, Y);
  printf("%d\n", res);
  bar(X, &res);
  printf("%d\n", res);
  return(0);
  }
```

Das Zweite ist ein Funktionsmodul `modul.c`:

```
int foo(int a, int b)
  {
  printf("I am foo\n");
  return a + b;
  }

void bar(int a, int *result)
  {
  printf("I am bar\n");
  *result = a;
  }
```

Das ist schon ganz nett und mittels `make` kann man `modul.c` und `test.c` übersetzen (ergibt `modul.o` und `test.o`) und zusammenlinken. Das Modul `test.c` muss auf jeden Fall den Prototyp von `foo()` und `bar()` kennen. Man könnte nun versucht sein, diese manuell vor das `main()` zu schreiben. Dies ist aber nicht ratsam, denn man müsste alle Prototyp-Definitionen überall eintragen bzw. ändern, wenn irgendwo `modul` verwendet respektive geändert würde.
Besser ist es, die Prototypen in eine separate Datei zu schreiben, die man `modul.h` nennt (das „h“ steht für „Header-Datei“). Für das Beispiel oben sieht die Header-Datei `mdul.h` so aus:

```
/* Funktions-Prototypen */
int foo(int a, int b);
void bar(int a, int *result);
```

Bei jeder Verwendung von `module` muss man jetzt nur noch dafür sorgen, dass der Inhalt des Header-Files am Anfang hinein kopiert wird, was sich mittels `#include "modul.h"` erledigen lässt. Beachten Sie die Gänsefüßchen! Für das `#include`-Makro gilt: Dateien zwischen `< >` werden in den vordefinierten Header-Verzeichnissen gesucht, Dateien zwischen `" "` werden im aktuellen Verzeichnis (relativ zur C-Quelle) gesucht. Unser Testmodul sieht nun so aus:

```
/* Testmodul */
#include <stdio.h>
#include "modul.h"

int main(void)
```

```
{
int X = 2,Y = 3, res;
res = foo(X, Y);
printf("%d\n", res);
bar(X, &res);
printf("%d\n", res);
return(0);
}
```

Es ist übrigens sinnvoll, auch in der Datei `modul.c` zu Beginn das Header-File `modul.h` per `#include` einzubinden (schon, damit der Compiler warnt, falls die Prototypen und die Funktionen sich auseinander entwickeln).
Bei größeren Systemen kann es durchaus vorkommen, dass einzelne Module die Funktionen anderer Module verwenden und daher deren Header-Dateien includieren. So kann es recht schnell zum mehrfachen Einbinden der Header-Dateien (mit entsprechen seltsamen Verhaltensweisen von Programm und Compiler) kommen. Mit den Präprozessor-Konstrukten `#define` und `#ifndef` kann man gewährleisten, dass eine Header-Datei jeweils nur beim ersten Mal effektiv includiert wird. Damit sieht unser Beispiel-Headerfile folgendermaßen aus:

```
#ifndef _MODUL_H_
#define _MODUL_H_
/* Funktions-Prototypen */
int foo(int a, int b);
void bar(int a, int *result);
#endif
/* _MODUL_H_ */
```

Nur wenn `_MODUL_H_` noch nicht definiert worden ist, wird der Quellcode zwischen `#ifndef` und `#endif` eingefügt. Zugleich wird auch `_MODUL_H_` definiert. Hiermit wird sichergestellt, dass die `#ifndef`-Bedingung bei zukünftigen Includes nicht mehr erfüllt ist.
Wir folgen der C/C++-Konvention, indem wir für die jeweiligen Symbole den Datei-Namen der Header-Datei verwenden und jeweils ein Unterline-Zeichen vorne und hinten anfügen. Ebenso wird der Punkt im Dateinamen durch das Underline-Zeichen ersetzt. Wenn Sie sich für jede Header-Datei an diese Konvention halten, kann nichts mehr schief gehen. Tun Sie es auch, wenn Sie der festen Überzeugung sind, dass keine Mehrfach-Inkludierung vorkommt. Beim `#endif` geben Sie als Kommentar die Bezeichnung zum zugehörigen Symbol der `#ifndef`-Direktive an. So wissen wir immer, zu welchem `#ifndef` ein `#endif` gehört.
Das Makefile zum Beispiel muss natürlich nun auch das Header-File berücksichtigen:

```
go: modul.o test.o
    gcc modul.o test.o -lm -lglib -o go
modul.o: modul.c modul.h
    gcc -Wall -O2 -c modul.c
test.o: test.c
    gcc -Wall -O2 -c test.c
clean:
    rm -rf *.o
```

Anhang

A.1 Literatur zu Embedded Linux

- Robert Schwebel: *Embedded Linux – Handbuch für Entwickler*, mitp
- John Lombardo: *Embedded Linux*, New Riders
- Craig Hollabaugh: *Embedded Linux – Hardware, Software, and Interfacing*, Addison Wesley
- Karim Yaghmour: *Building Embedded Linux Systems*, O'Reilly
- Bob Smith, John Hardin, und Graham Phillips: *Linux Appliance Design*, No Starch Press
- Thomas Brinker, Heiko Degenhardt, Gerald Kupris: *Embedded Linux – Praktische Umsetzung mit uClinux*, VDE Verlag
- John Catsoulis: *Designing Embedded Hardware*, O'Reilly
- Jürgen Quade, Eva-Katharina Kunst: *Linux-Treiber entwickeln*, dpunkt
- Wolfgang Mauerer: *Linux Kernelarchitektur*, Hanser

A.2 Literatur zum Programmieren unter Linux

- Martin Gräfe: *C und Linux*, Hanser
- R. Krienke: *UNIX-Shell-Programmierung*, Hanser
- B. Kernighan, R. Pike: *UNIX-Werkzeugkasten*, Hanser
- R. Stones, N. Matthew: *Linux Programmierung*, mitp
- E. de Castro Lopo, P. Aitken, B. L. Jones: *C-Programmierung für Linux*, Markt & Technik
- Jürgen Wolf: *Linux-UNIX-Programmierung*, Galileo Computing
- Stefan Fischer, Walter Müller: *Netzwerkprogrammierung unter LINUX und UNIX*, Hanser
- W. Richard Stevens: *Programmierung von UNIX-Netzen*, Hanser

A.3 Links

Embedded Linux Survey
http://www.bluemug.com/products/els/
Linux Automation: Portalseite für Automatisierungstechnik mit Linux
http://www.linux-automation.de
Embedded Linux Consortium
http://www.embedded-linux.org
Embedded Linux
http://www.linuxembedded.com
http://www.embedlinux.net

uCLinux Linux
http://www.uclinux.org
Realtime Linux
http://www.realtimelinux.org
Linux Devices
http://LinuxDevices.com

Stichwortverzeichnis

T

U

W

Z