8.3 Sonstiges/Signalbehandlung

8.3.1 Was sind Signale?

Signale bilden in der Sprache C die Funktionalität ab, die in Maschinensprache (bzw. Assembler) von Interrupts bekannt ist: Ein Prozess kann zu beliebiger Zeit an einer beliebigen Stelle von außen oder durch sich selbst unterbrochen werden. In der Sprache ANSI-C sind verschiedene Signale und ihre Behandlung festgelegt. Unter MS-Windows sind leider nicht alle diese Signale vorhanden, so dass die folgenden Beispiele eher unter Linux oder anderen Unix-artigen Systemen ausgeführt werden sollten. Trotzdem gehören Signale zum C-Sprachstandard.

Jedes Signal besitzt eine Nummer (meistens zwischen 1 und 31), so dass man eine Reihe von Signalen unterscheiden kann. In `<signal.h>` sind außerdem Namen für die Signale festgelegt. Einige Beispiele zeigt Tabelle 1. Die in der vierten Spalte gekennzeichneten Signale sind im C-

Nr.	Name	Bedeutung	ANSI-C
1	SIGHUP	Hangup — Ausgeloggen der Standard-Eingabe	
2	SIGINT	Interrupt — Strg-C gedrückt	Ja
3	SIGQUIT	Abbruch durch Tastatur — Strg-AltGr-\ gedrückt	
4	SIGILL	Ungültiger CPU-Befehl	Ja
5	SIGTRAP	...	
6	SIGABRT	Anormaler Programmabbruch	
7	SIGBUS	...	
8	SIGFPE	Gleitkomma-Division durch null	Ja
9	SIGKILL	Ende, kann nicht abgefangen werden	
10	SIGUSR1	Frei für Benutzer	
11	SIGSEGV	Schutzverletzung	Ja
12	SIGUSR2	Frei für Benutzer	
13	SIGPIPE	FIFO wurde geschlossen	
14	SIGALRM	Timer abgelaufen	
15	SIGTERM	Programmende (oder Neuladen der Konfiguration)	
	...	...	
18	SIGCONT	Programm soll weiterlaufen nach SIGSTOP oder SIGTSTP	
19	SIGSTOP	Prozessstop, kann nicht abgefangen werden	
20	SIGTSTP	Unterbrechung – Strg-Z gedrückt	
	...	...	

Tabelle 1: Nummern und Namen einiger Signale

Sprachstandard genannt.

8.3.2 Erzeugen von Signalen

Signale können auf verschiedene Arten erzeugt werden:

a) Manche Signale können vom Benutzer erzeugt werden, der z. B. an der Tastatur eine bestimmte Tastenkombination eingibt. Das Signal wird an der Standard-Eingabe (genauer gesagt, am Tastatur-Treiber) vorbei zu Prozess geschickt. Zu diesen Signalen gehören SIGHUP (Beenden der Standard-Eingabe), SIGINT (Strg-C), SIGQUIT (Strg-AltGr-\) und SIGTSTP (Strg-Z).

b) Manche Signale werden durch den Prozess selbst verursacht, z. B. durch Zugriff auf eine ungültige Speicheradresse (SIGSEGV), durch Division durch 0 (SIGFPE) oder durch ungültige CPU-Befehle (SIGILL).

c) Manche Signale werden uns von anderen Prozessen, z. B. dem Elternprozess oder dem Kernel geschickt. Jeder Prozess kann nämlich jedem anderen Prozess ein beliebiges Signal schicken. Allerdings wird das Signal nur dann zugestellt, wenn die Berechtigungen es erlauben.

d) Schließlich kann sich ein Prozess selbst ein beliebiges Signal schicken. Dazu gibt es in der C-Standard-Bibliothek die Funktionen `raise()` und `abort()`.

8.3.3 Signale erzeugen auf der Tastatur

Das Programm `sleep` bewirkt ein Abwarten des Prozesses. Mit Strg + C kann er unterbrochen werden:

Terminal

```
schueler@debian964:~$ sleep 100
 Strg + C
^C
schueler@debian964:~$ echo $?
130
```

Die Ausgabe `^C` bedeutet Strg + C . Der Rückgabewert ergibt sich aus der Summe von 128 und der Signalnummer 2.

8.3.4 Signale erzeugen durch das Programm selbst

Das folgende Programm (`src/division1.c`) enthält eine Division durch 0.

```c
#include <stdio.h>
int main(void)
{
    int x=0, y=0, z;
    z=x/y;
    return 0;
}
```

Terminal

```
schueler@debian964:~$ gcc -o division1 division1.c
schueler@debian964:~$ division1
Gleitkomma-Ausnahme
schueler@debian964:~$ echo $?
136
```

Wieder ergibt sich der Rückgabewert aus der Summe von 128 und der Signalnummer 8.

8.3.5 Signale erzeugen mit `raise()`

Mit der Funktion `raise()` kann sich ein Prozess selbst ein Signal schicken lassen (Programm `src/raise1.c`):

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

int main(void)
{
    printf("La\n");
    printf("Le\n");
    raise(SIGXFSZ); /* Signal Nr. 23 */
    printf("Lu\n");
```

```
10      return 0;
11  }
```

Zeile 8 Hier bewirkt der Prozess, dass er unverzüglich das Signal `SIGXFSZ` geschickt bekommt. Standardmäßig bewirkt dieses Signal einen Programmabbruch.

Zeile 9 Diese Zeile wird nicht mehr erreicht, da der Prozess schon abgebrochen wurde.

Das Programm hat folgende Ausgabe:

Terminal

```
schueler@debian964:~$ raise1
La
Le
Die maximale Dateigröße ist überschritten
schueler@debian964:~$ echo $?
153
```

Die Meldung wurde durch das Betriebssystem generiert. Der Rückgabewert ergibt sich aus der Summe von 128 und der Signalnummer 25.

8.3.6 Programmabbruch-Signal erzeugen mit `abort()`

Die Funktion `abort()` ist gewissermaßen eine Spezialisierung der Funktion `raise()`: Mit ihr kann eine Funktion sich selbst das Signal `SIGABRT` schicken lassen. Mit `abort()` kann ein Programm neben `exit()` einen zweiten Ausgang bekommen.

Wenn man ein Programm ganz normal mit `exit()` beendet (oder `main()` mit `return` verlässt), dann werden zuerst alle mit `atexit()` registrierten Callback-Funktionen aufgerufen, anschließend der so genannte *cleanup*-Code, der offene Dateien schließt und (falls vorhanden) temporäre Datei löscht. Erst dann wird zum Kernel zurückgekehrt.

Die Funktion `abort()` dagegen umgeht diesen Weg und bewirkt eine *sofortige* Rückkehr zum Kernel. Deshalb ist festgelegt, dass es von der Funktion `abort()` keine Rückkehr gibt[1]. Im folgenden Programm wird `abort()` ausgeführt (`src/abort1.c`):

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void)
5  {
6      printf("La\n");
7      printf("Le\n");
8      abort();
9      printf("Lu\n");
10     return 0;
11 }
```

Zeile 2 `abort()` ist in `<stdlib.h>` deklariert.

Zeile 8 Hier bewirkt der Prozess, dass er das Signal `SIGABRT` geschickt bekommt.

Zeile 9 Diese Zeile wird nicht mehr erreicht, da der Prozess schon abgebrochen wurde.

Das Programm hat folgende Ausgabe:

[1] Allerdings darf man das Signal abfangen, um vorher noch eigene Aufräumarbeiten auszuführen.

Terminal
```
schueler@debian964:~$ raise1
La
Le
Abgebrochen
schueler@debian964:~$ echo $?
134
```

8.3.7 Senden eines Signals an einen fremden Prozess I

Auf der Kommandozeilenebene gibt es unter Linux die Programme `kill` und `killall`, mit denen ein Prozess einem anderen ein Signal schicken kann, unter der Voraussetzung, dass er die Berechtigung dazu besitzt:

Terminal
```
schueler@debian964:~$ xeyes &              # im Hintergrund
[1] 5305                  # Prozess-ID 5305
schueler@debian964:~$ ps                   # Prozesse anzeigen
  PID TTY          TIME CMD
 5140 pts/1    00:00:00 bash
 5305 pts/1    00:00:00 xeyes  # Erste Spalte: Prozess-ID
 5312 pts/1    00:00:00 ps
schueler@debian964:~$ kill 5305
[1]+  Beendet                 xeyes
schueler@debian964:~$ xeyes &              # im Hintergrund
[1] 5326
schueler@debian964:~$ xeyes &              # auch im Hintergrund
[2] 5327
schueler@debian964:~$ killall xeyes
[1]-  Beendet                 xeyes
[2]+  Beendet                 xeyes
```

Standardmäßig wird dabei das Signal 15 (`SIGTERM`) verschickt (erkennbar an der Meldung „Beendet“). Will man ein anderes Signal schicken, kann man das mit einer Option machen. Man kann entweder die Signalnummer oder den Signalnamen angeben:

Terminal
```
schueler@debian964:~$ kill -18 5305     # Signal Nr. 18 (CONT)
schueler@debian964:~$ kill -CONT 5305   # Signal Nr. 18 (CONT)
```

8.3.8 Senden eines Signals an einen fremden Prozess II

Auf der Programmebene gibt es unter Linux und verwandten Systemen den Systemaufruf `kill()`, der ähnlich wie das Programm `kill` an einen anderen Prozess ein Signal schicken kann. Ein Beispiel zeigt das Programm `src/kill1.c`:

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <signal.h>
4
5  int main(void)
6  {
7     int signr, erg;
8     pid_t pid;
9     printf("Prozess-ID: "); scanf("%i", &pid);
10    printf("Signal-Nr.: "); scanf("%i", &signr);
```

```
11      erg=kill(pid, signr);
12      printf("Returncode: %i\n", erg);
13      return 0;
14  }
```

Und so kann man das Programm benutzen (Benutzereingaben in Fettdruck):

Terminal

```
schueler@debian964:~$ kill1
Prozess-ID: 1  # Versuche,System z.Absturz..
Signal-Nr.: 9  # und zwar sofort ...
Returncode: -1          # ging nicht
schueler@debian964:~$ sleep 200 &        # eigenen Prozess erzeugen
[1] 14469               # das ist seine Nummer
schueler@debian964:~$ kill1
Prozess-ID: 14469 # diesem Proz.Signal schicken
Signal-Nr.: 24    # Signal Nr. 24
Returncode: 0             # das hat geklappt
[1]+  Rechenzeitbegrenzung überschritten      sleep 200
```

Und wieder wurde durch das Betriebssystem eine Meldung generiert.

8.3.9 Abfangen von Signalen mit der Funktion `signal()`

Die Funktion `signal()` bekommt eine Nummer und einen Funktionszeiger als Parameter und gibt einen anderen Funktionszeiger zurück. Hier ein Beispiel (`src/signal1.c`):

```
1   #include <signal.h>
2   #include <stdio.h>
3
4   volatile int aufgerufen=0;
5
6   void tunichtviel(int signalnummer)
7   {
8      ++aufgerufen;
9   }
10
11  int main(void)
12  {
13                                  /* Signal einrichten */
14      void (*rc)(int);
15      rc=signal(SIGINT, tunichtviel);
16      if(rc==SIG_ERR)
17      {
18          printf("Signalhandler nicht eingerichtet.\n");
19          return 1;
20      }
21                                  /* Hauptprogramm */
22      while(aufgerufen<10)
23      {
24          printf("\rBis jetzt %d-mal Strg-C", aufgerufen);
25          fflush(stdout);
26      }
27      printf("\nInsgesamt %d-mal Strg-C\n", aufgerufen);
28      return 0;
29  }
```

Zeile 15 Hier wird der Signal-Handler für das Signal Nr. 2 (`SIGINT`) eingerichtet. Der Signal-Handler ist eine Callback-Funktion, die immer dann aufgerufen wird, wenn das Signal kommt. Der zweite Parameter `tutnichtviel` ist die Adresse dieser Funktion.

Ab diesem Zeitpunkt kann man übrigens das Programm mit Strg - C nicht mehr abbrechen[2].

Zeile 6–9 Hier ist die Funktion, die immer aufgerufen wird, wenn das Signal SIGINT ankommt. Bei jedem Drücken wird diese Funktion aufgerufen. Sie setzt die Variable `aufgerufen` um eins hoch danach beendet sie sich.

Zeile 4 Eine globale Variable! Das ist hier nötig, weil der Signal-Handler keine Parameter (außer der Signalnummer) und keinen Rückgabewert hat. Im Hauptprogramm wird in Zeile 22 der Wert dieser Variable abgefragt und beeinflusst damit das Programm.

Zeile 4 Die Variable `aufgerufen` ist hier mit dem Attribut `volatile` ausgestattet. Das heißt, diese Variable kann quasi von außen ihren Wert ändern. Das hindert den Compiler daran, die `while`-Schleife wegzuoptimieren. Ein solches Attribut sollte man im Vornherein für alle Variablen vergeben, die durch den Signalhandler gesetzt werden.

Zeile 22 Erst wenn die Variable `aufgerufen` auf 10 hochgezählt wurde, beendet sich das Programm.

8.3.10 Die Standard-Signalhandler

Statt eines Signalhandlers kann man der `signal()`-Funktion als zweiten Parameter auch eine der folgenden Konstanten mitgeben:

`SIG_IGN` – Signal ignorieren und nichts tun

`SIG_DFL` – Standard-Verhalten, meistens Programmende

Man nennt sie *Standard-Signalhandler*.

8.3.11 Rückgabewerte von `signal()`

Die `signal()`-Funktion kann zwei verschiedene Informationen zurückgeben:

a) `SIG_ERR` – ein Fehlercode, wenn die Einrichtung des Signalhandlers nicht geklappt hat.

b) die Adresse des bisherigen Signalhandlers, der soeben mit dem `signal()`-Aufruf abgelöst wurde, wenn dies geklappt hat.[3]

In den Programmbeispielen hier wird nur die erste Information (Fehler oder nicht) abgefragt.

8.3.12 Signalhandler für mehrere Signale

Im folgenden Programm sieht man, wozu der einzige Parameter `signum` der Signalhandler-Funktion gut ist: Man kann einen Signalhandler für mehrere Signale verwenden. Anhand des Parameters findet man heraus, welches Signal gerade aufgetreten ist. Ein Beispiel zeigt `src/signal2.c`:

```
1 #include <signal.h>
2 #include <stdio.h>
3 volatile int anzahl_aufrufe=0;
4 volatile int nr_letztes_signal=0;
5 volatile int neues_signal_angekommen=0;
6
```

[2] Mit Strg - AltGr - \ geht es noch immer.

[3] Mit einem neuen `signal()`-Aufruf könnte man ihn wieder aktivieren.

```
7  void tunichtviel(int signalnummer)
8  {
9     nr_letztes_signal=signalnummer;
10    ++anzahl_aufrufe;
11    neues_signal_angekommen=1;
12 }
13
14 int main(void)
15 {
16    int lauf;
17    typedef void (*sighandler_t)(int);
18    sighandler_t rc;
19    for(lauf=1; lauf<32; ++lauf)
20    {
21       rc=signal(lauf, tunichtviel);
22       if(rc==SIG_ERR)
23          printf("Signalhandler %d: Fehler.\n", lauf);
24       else if (rc==SIG_DFL)
25          printf("Signalhandler %d: war vorher SIG_DFL (Beenden)\n");
26       else if (rc==SIG_IGN)
27          printf("Signalhandler %d: war vorher SIG_IGN (Ignorieren)\n");
28       else
29          printf("Signalhandler %d: war vorher Funktion an Adresse %p\n", rc);
30    }
31                                  /* Hauptprogramm */
32    while(anzahl_aufrufe < 10)
33    {
34       printf("Bis jetzt %d Signale\n", anzahl_aufrufe);
35       printf("Letztes Signal war %d\n", nr_letztes_signal);
36       while(!neues_signal_angekommen) /* auf naechstes Signal warten */
37       {}
38       neues_signal_angekommen=0;
39    }
40    printf("Bis jetzt %d Signale\n", anzahl_aufrufe);
41    printf("Allerletztes Signal war %d\n", nr_letztes_signal);
42    return 0;
43 }
```

Zeilen 3–5 Auch hier sind die globalen Variablen wieder mit dem Attribut `volatile` ausgestattet.

Zeilen 7–11 Dieser Signalhandler setzt immerhin drei globale Variablen. Je mehr, desto unübersichtlicher wird es. Deshalb unbedingt „sprechende“ Variablennamen verwenden!

Zeile 17 `rc` soll den Rückgabewer von `signal()` aufnehmen. Deshalb ist es eigentlich ein Funktionszeiger (siehe in den Kommentarklammern am Ende der Zeile). Aber wegen der `typedef`-Typendefinition in Zeile 16 kann man auch einfach `sighandler_t` schreiben.

Zeile 18 Das sind die bei Linux üblichen Signale.

Zeile 21 Signal Nr. `lauf` bekommt die Adresse der Funktion `tutnichtviel()` zugewiesen. Ohne Funktionsklammern, weil es hier nur die Adresse ist (und kein Funktionsaufruf).

Zeile 22 Falls ein Fehler auftritt: Bescheid geben.

Zeile 32 Nach zehn erhaltenen Signalen soll das Programm sich beenden.

Zeile 36–37 Solange kein neues Signal ankommt, soll die äußere Schleife nicht ständig durchlaufen werden, damit der Bildschirm nicht vollgeschrieben wird. Daher hier die leere Warteschleife.