Kapitel 3

Signale

Ein Signal dient zur Benachrichtigung eines Processes mittels Software. Es wird erzeugt in dem Augenblick in dem das Ereignis eintritt, über das mittels Signal berichtet werden soll (z.B. division by zero). Das Signal wird übermittelt in dem Augenblick, in dem der Prozess auf das Signal reagiert. Ein Signal, das noch nicht übermittelt wurde heisst *pending*. Ein Process fängt ein Signal ab, wenn er bei Übermittlung einen sog. Signalhandler startet. Ein Process ignoriert ein Signal, wenn es nach Übermittlung weg geworfen wird. Die Signalhandler werden mit dem Aufruf sigaction() installiert, Parameter ist eine Funktion oder die Konstanten SIG_DFL oder SIG_IGN. SIG_IGN bedeutet ignorieren und SIG_DFL bedeutet Ausführen der default Aktion. Es gibt ferner noch eine signal mask, hier können Signale blockiert werden, d.h. sie sind pending bis der Process die Abarbeitung dieser Signale wieder aufnimmt. Hierfür gibt es den Befehl sigprocmask.

3.1 Senden von Signalen

Signale haben symbolische Namen (intern sind es Zahlen) die alle mit SIG beginnen. Nach POSIX gibt es folgende:

Name	Bedeutung
SIGABRT	von routine abort()
SIGALRM	timer
SIGFPE	division durch null
SIGHUP	hangup on controlling terminal
SIGILL	invalid hardware instruction
SIGINT	
SIGKILL	kann nicht abgefangen werden
SIGPIPE	write on pipe with no reader
SIGQUIT	interactive termination
SIGSEGV	segmentation violation
SIGTERM	
SIGUSR1/2	

Die default action ist in allen Fällen die abnormale Beendigung des Processes. Es gibt ferner nach POSIX noch einen Satz von Signalen zur Job Control

Name	Bedeutung
SIGCHLD	child terminates/stopped
SIGCONT	continue if stopped
SIGSTOP	anhalten
SIGTSTP	interactive stop signal
SIGTTIN	backgroud process liest von controlling terminal
SIGTTOU	background process schreibt auf controlling terminal

Default ist das Stoppen ausser bei SIGCHLD (wird ignoriert) und bei SIGCONT (es geht weiter, auch wenn geblockt oder ignoriert). Signale kann man nur an eigene Processe schicken. In der shell gibt es dafür den kill Befehl:

```
kill(1) kill(1)

NAME

kill - Sends a signal to a running process

SYNOPSIS
kill -l [exit_status]
kill [-signal_name | -signal_number] process_ID ...
kill -s signal_name process_ID ...
```

Die Option -l ist zum listen aller verfügbaren Signale. Auch auf Programmierebene heisst der entsprechende Aufruf kill:

```
kill(2)

NAME
kill - Sends a signal to a process or to a group of
processes

SYNOPSIS
#include <signal.h>

int kill( pid_t pid, int signal );
```

Ist pid > 0 wird an den jeweiligen Process das entsprechende Signal gesandt. Ist pid < 0 wird an an die entsprechende Process Gruppe geschickt. Ist pid = 0 wird an die gleiche Gruppe wie der aufrufende Process das Signal geschickt. Der Rückgabewert ist -1 im Fehlerfall und errno wird entsprechend gesetzt. Man kann auch an den eigenen Process mit raise() ein Signal senden:

```
raise(3)

NAME
raise - Sends a signal to the executing process or
thread

LIBRARY
Standard C Library (libc.so, libc.a)

SYNOPSIS
#include <signal.h>

int raise( int signal );
```

Viele Signale können auch über Tastatur an den aktuell laufenden Process gesandt werden. Dazu sollte man mit dem Aufruf stty -a die Einstellung des eigenen Terminals anschauen:

```
$ stty -a
#2 disc;speed 9600 baud; 24 rows; 80 columns
erase = ^H; werase = ^W; kill = ^U; intr = ^C; quit = ^\; susp = ^Z
dsusp = ^Y; eof = ^D; eol = <undef>; eol2 = <undef>; stop = ^S
start = ^Q; lnext = ^V; discard = ^O; reprint = ^R; status = <undef>
time = 0; min = 1
-parenb -parodd cs8 -cstopb hupcl cread -clocal -crtscts
-ignbrk brkint -ignpar -parmrk -inpck -istrip -inlcr -igncr icrnl
-iuclc
ixon ixany -ixoff imaxbel
isig icanon -xcase echo echoe -echok -echonl -noflsh -mdmbuf -nohang
-tostop echoctl -echoprt echoke -altwerase iexten -nokerninfo
opost -olcuc onlcr -ocrnl -onocr -onlret -ofill -ofdel tabs -onoeot
$
```

Mit ctl-c wird SIGINT ausgelöst, mit ctl-\ wird ein SIGQUIT ausgelöst mit ctl-z ein SIGSTOP und mit ctl-y ein SIGCONT. Das SIGALARM kann man durch den Aufruf der Funktion alarm() auslösen:

```
ALARM(2)


NAME
alarm - set an alarm clock for delivery of a signal

SYNOPSIS
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Die Aufrufe wirken nicht als Stack, d.h. es wird der einzige Alarm auf den entsprechenden Wert gesetzt. Mit 0 wird er gelöscht. Der Rückgabewert ist die vorher noch verbliebene Zeit des letzten alarm Aufrufes. Dazu ein kleines Beispiel:

```
#include <unistd.h>

main()
{
alarm(10);
while(1);
}
```

Und startet man das Programm nach dem kompilieren:

```
$ cc   t1.c  -o t1
$ time t1
time: command terminated abnormally.

real   10.1
user   2.9
sys    0.0
$
```

So sieht man, dassdas Programm nach 10 Sekunden abnormal beendet wird (default auf SIGALRM).

3.2 signal mask

Hiermit kann man Signale blocken, d.h. sie werden nicht entgegen genommen, sind also pending bis man die Maske ändert. Dabei arbeitet man mit Signalmengen, ist ein Signal in der Menge bedeutet dies, dass es geblockt wird. Es bleibt pending. Man hat verschiedene Befehle zur Verfügung:

```
SIGSETOPS(3)

NAME
sigemptyset, sigfillset, sigaddset, sigdelset,
sigismember -
  POSIX signal set operations.

SYNOPSIS
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Mit sigemptyset wird eine leere Signalmenge erzeugt, oder mit sigfillset wird eine Signalmenge mit allen Signalen erzeugt. Nachfolgend wird eine Signalmenge mit sigaddset vergrössert oder mit mit sigdelset verkleinert. Der Befehl sigismember dient den Test ob ein Signal enthalten ist. Mit dem Aufruf von sigprocmask

```
SIGACTION(2)

NAME
sigaction, sigprocmask, sigpending, sigsuspend -
       POSIX signal handling functions.

SYNOPSIS
#include <signal.h>
…
int sigprocmask(int how, const sigset_t *set, sigset_t
*oldset);
```

kann man nun die Maske setzen. Der zweite Parameter ist die Signalmenge, der dritte wird die alte Signalmenge vor der Änderung, d.h. die Maske wie sie vorher aktiv war. Der erste Parameter sagt mit SIG_BLOCK, dass die neue Menge mit geblockt werden soll. SIG_UNBLOCK bedeutet, dass die neue Menge nicht mehr geblockt werden soll. SIG_SETMASK setzt eine neue Maske. Ist der zweite Parameter NULL wird nur der aktuelle Stand abgefragt. Ist der dritte NULL wird der alte Wert nicht gespeichert. Hierzu ein Beispiel:

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

void main(int argc, char * argv[])
{
double y;
sigset_t intmask;
int i, repeat_factor;

if (argc != 2) {
fprintf(stderr,"Usage: %s repeat_factor\n", argv[0]);
exit(1); }

repeat_factor = atoi(argv[1]);
sigemptyset(&intmask);
sigaddset(&intmask, SIGINT);

for( ; ; ) {
  sigprocmask(SIG_BLOCK, &intmask, NULL);
  fprintf(stderr, "SIGINT signal blocked\n");
  for (i = 0; i < repeat_factor; i++) y =
sin((double)i);
  fprintf(stderr, "Blocked calculation is finished\n");
  sigprocmask(SIG_UNBLOCK, &intmask, NULL);
  fprintf(stderr, "SIGINT signal unblocked\n");
  for (i = 0; i < repeat_factor; i++) y =
sin((double)i);
  fprintf(stderr, "Unblocked calculation is
finished\n");
  }
}
```

Eine Rechnung (sinnlos) wird ausgeführt einmal mit geblockten Signal, dann wird ctl-c abgefangen bis zum Ende oder aber ohne geblockten Signal, dann wird sofort abgebrochen. Abgebrochen wird auch wenn das Signal in der ersten Phase ankam, aber erst in der zweiten entgegen genommen wird.

3.3 Abfangen, Ignorieren

Mit dem Aufruf sigaction kann ein eigner Signalhandler installiert werden.

```
sigaction(2)

NAME
sigaction - detailed signal management

SYNOPSIS
#include <signal.h>
int sigaction(int sig, const struct sigaction *act,
        struct sigaction *oact);
```

Der Aufruf hat drei Parameter, der erste ist die Signalnummer, natürlich soll man den symbolischen Namen verwenden, dann der neue handler, und als dritter Parameter der alte handler. Wie schon bei der Maske kann man mittels NULL Parameter nur Abfragen oder aber den alten ignorieren.

Die Struktur sigaction hat drei (eigentlich vier, ist aber momentan nicht wichtig) Komponenten:

```
void (*sa_handler)(int);
sigset_t sa_mask;
int sa_flags;
```

Ein Signal handler ist eine Funktion vom typ void und einem int Parameter, dies wird beim Aufruf die Nummer des Signals. sa_mask ist eine Maske von Signalen die während der Arbeit des handler zusätzlich geblockt werden sollen. Prinzipiell ist es aber nicht möglich den handlern noch Parameter mit zu übergeben. Als handler gibt es noch die Konstanten SIG_DFL und SIG_IGN. Der dritte Parameter dient zur Steuerung des Aufrufs des handlers. Dies wird hier nicht betrachtet. Ein Beispiel:

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

char handmsg[] = "I found ^c\n";
void catch_ctrl_c(int signo)
{
write(STDERR_FILENO, handmsg, strlen(handmsg));
}
...
struct sigaction act;
...
act.sa_handler = catch_ctrl_c;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
if (sigaction(SIGINT, &act, NULL) < 0)
/* handle error here */
...
```

Es wird ein handler für SIGINT eingebaut. Beim Programmieren von handlern ist es wichtig auf async-signal-safe (d.h. kann von signal handlern aufgerufen werden) zu achten. Von der Funktion write weiss man das, von fprintf ist es nicht klar. Man kann z.B. unter Solaris über man 5 attributes feststellen, welche Funktion async-signal-safe sind. Unter POSIX wird vorgeschrieben, dass einige Funktion async-signal-safe sind. So z.B. write().

3.4 Warten

Mittels Signalen (timern) kann man vermeiden, dass ein Programm CPU Zeit für das Warten verschwendet. Die bessere Methode ist in den sleep Status zu gehen bis ein Signal eintrifft. Dazu gibt es die Funktionen pause() und sigsuspend().

```
pause(2)

NAME
pause - suspend process until signal

SYNOPSIS
#include <unistd.h>
int pause(void);
```

Der Aufruf von pause() wartet bis ein nicht ignoriertes Signal eintrifft und abgearbeitet ist. Danach endet der Aufruf von pause(). Der Rückgabewert ist immer -1. Will

man mit pause() auf ein spezielles Signal warten braucht man eine globale Variable, die vom entsprechenden signal handler bearbeitet wird. Das schaut dann so aus:

```
#include <unistd.h>
int signal_received = 0;
/* external static variable */

..
while(signal_received == 0)
    pause();
```

Das Problem ist, wenn das Signal zwischen dem Test und dem Aufruf von pause() eintritt? Man bekommt eine Endlosschleife. Dies ist das prinzipielle Problem von pause(). Man kann noch mit Blocken des Signals arbeiten und dem anschliessenden Aufruf, aber auch das sind zwei Funktionsaufrufe, und das Signal kann dazwischen kommen. Man braucht eine Lösung die atomic ist d.h. es kann nichts dazwischen kommen. Dies ist sigsuspend().

```
sigsuspend(2)

NAME
sigsuspend - install a signal mask and suspend caller
until signal

SYNOPSIS
#include <signal.h>
int sigsuspend(const sigset_t *set);
```

Es wird eine Maske gesetzt und gewartet bis ein Signal eintrifft und bearbeitet wird. Mit der Maske wird das zuvor geblockte Signal auf unblocked gesetzt und dann gewartet. Wenn sigsuspend beendet wird, wird die ursprüngliche Maske wieder hergestellt. Unser Beispiel funktioniert dann so:

```
#include <signal.h>
int signal_received = 0;
/* external static variable */
...
sigset_t sigset;
sigset_t sigoldmask;
int signum;

sigprocmask(SIG_SETMASK, NULL, &sigoldmask);
sigprocmask(SIG_SETMASK, NULL, &sigset);
sigaddset(&sigset, signum);
sigprocmask(SIG_BLOCK, &sigset, NULL);
sigdelset(&sigset, signum);
while(signal_received == 0)
      sigsuspend(&sigset);
sigprocmask(SIG_SETMASK, &sigoldmask, NULL);
```

3.5 system calls und Signale

Es kann zwei Probleme geben. Was passiert wenn ein Signal eintrifft. Soll der unterbrochene system call wiederholt werden? Das zweite Problem ist mit non reentrant system calls in den signal handlern.

3.5.1 slow system calls

Einige system calls blocken, d.h. der process wartet bis der system call fertig ist (z.B. I/O). Dies sind sog. langsame system calls, und diese können durch Signale unterbrochen werden. Diese liefern dann einen Fehler (-1) und errno wird auf EINTR gesetzt. Der Programmierer muss dann den Befehl wiederholen:

```
#include <sys/types.h>
#include <sys/uio.h>
#include <unistd.h>
#include <errno.h>
int fd;
int retval;
int size;
char *buf;

while (retval = read(fd, buf, size),
               retval == -1 && errno == EINTR) ;

if (retval == -1)
/* handle errors here */
```

Dieses Verhalten kann man auch automatisch erreichen indem man das sa_flag beim Aufruf von sigaction verwendet. Dies muss auf SA_RESTART setzen. Dann wird der unterbrochene system call automatisch wieder gestartet. Das Problem mit slow system calls ist, dass sie unbestimmte Zeit blocken, z.B. weil der remote Rechner, der das file liefert momentan nicht erreichbar ist. Man sollte daher sicher programmieren:

```
#include <unistd.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
int fd;
int size;
char *buf;

alarm(10);
if (read(fd, buf, size) < 0) {
if (errno == EINTR)
/* handle timeout here */
else
/* handle other errors here */
}
alarm(0);
```

Dies funktioniert natürlich nur wenn sigaction für SIGALRM kein RESTART enthält.

3.6 longjmp

Es gibt Situation, in denen auf Signale durch Neustart oder Wiederholen eines Programmteils reagiert werden soll. Programmiertechnisch ist dies ein Sprung an eine gewisse Stelle des Programms. Man kann dies Erreichen indem man mittels globaler Variable im Programm testet ob diese Situation vorliegt, dies wird unangenehm bei hoher Schachtelungstiefe. Es gibt aber auch eine direkte Möglichkeit. Ein Paar von Funktionen sigsetjmp() und siglongjmp(). sigsetjmp() entspricht dem setzen eines labels und siglongjmp() entspricht dem goto.

```
sigsetjmp(3)
NAME
sigsetjmp, siglongjmp - Saves and restores the current
execution context

SYNOPSIS
#include <setjmp.h>
int sigsetjmp( sigjmp_buf env, int savemask);
void siglongjmp( sigjmp_buf env, int value);
```

Mit sigsetjmp() wird ein Zustand in der Variable env gesichert. Ist savemask <> 0 wird auch die aktuelle Signalmaske mit gespeichert. Der Rückgabewert ist 0. Um nun zu einem label zu springen muss siglongjmp() mit dem entsprechenden env Wert aufgerufen werden. Und das Programm macht an der Stelle nach dem Aufruf von sigsetjmp() weiter, aber mit dem Rückgabewert value und nicht 0 wie beim Setzen des labels. Ein Beispiel wie dieses Paar verwendet werden kann:

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

static volatile sig_atomic_t jumpok = 0;
static sigjmp_buf jmpbuf;
..
void int_handler(int errno)
{
if (jumpok == 0) return;
siglongjmp(jmpbuf, 1);
}
...
void main(void)
{
struct sigaction act;
...
act.sa_handler = int_handler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
if (sigaction(SIGINT, &act, NULL) < 0) {
perror("Error setting up SIGINT handler");
exit(1);
}
...
if (sigsetjmp(jmpbuf, 1))
fprintf(stderr, "Returned to main loop due to ^c\n");
jumpok = 1;
... /* start of main loop */
}
```

Auf das Signal SIGINT wird durch Neustart reagiert. Man muss Aufpassen, dass erst siglongjmp() aufgerufen wird wenn das label mit sigsetjmp() gesetzt wurde. Dazu die globale Variable jumpok.

3.7 realtime signals

Damit ist eine Erweiterung der sigaction Syntax gemeint. Damit ist es möglich dem Signal handler weitere Parameter zu übermitteln. Diese erweiterte Syntax wird verwendet wenn bei sa_flags SA_SIGINFO gesetzt ist. Dann wird die Komponente sa_sigaction der erweiterten Syntax verwendet:

```
void (*sa_handler)(int);
sigset_t sa_mask;
int sa_flags;
void(*) (int, siginfo_t *, void *) sa_sigaction;
```

Die neu hinzugekommene Komponente ist ein handler, der als Parameter wieder die Signal Nummer bekommt, der zweite Parameter ist ein Zeiger auf eine Struktur, die mindestens folgende Komponenten enthält: si_signo, die Signalnummer, si_code, die Ursache des Signals: SI_USER, SI_QUEUE, SI_TIMER, SI_ASYNCHIO, SI_MESGQ, dritte Komponente ist si_value, was ein Parameter an den signal handler darstellt. Das gehandelte Signal kann auf verschiedene Weisen ausgelöst werden. Im Falle vom abort(), kill() oder raise() ist si_code = SI_USER und kein si_value kann gesetzt werden. Ein Problem mit den bisherigen Signal handling ist wenn mehrere gleiche Signale pending sind, das Verhalten ist nicht definiert, dieses ist aber kritisch für realtime Systeme. Dazu gibt es die Möglichkeit des queings von Signalen:

```
sigqueue(3)
NAME
sigqueue - Queue a signal and data to a running process.

SYNOPSIS
#include <signal.h>
int sigqueue ( pid_t pid, int signo, const union sigval
value);
```

Das Signal wird mit si_code = SI_QUEUE erzeugt. Es kann ein Parameter für si_value übergeben werden. Die maximale Anzahl der in der Queue gespeicherten Signale ist SIGQUEUE_MAX (=32). Nachfolgend ein Programm, welches ein queueed Signal an einen Process schickt:

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

void main(int argc, char *argv[])
{
union sigval qval;
int val;
int pid;
int signo;

if (argc != 4) {
  fprintf(stderr, "Usage: %s pid signal value\n",
argv[0]);
  exit(1);
}

pid = atoi(argv[1]);
signo = atoi(argv[2]);
val = atoi(argv[3]);
fprintf(stderr, "Sending signal %d with value %d to
process %d\n",
             signo, val, pid);
qval.sival_int = val;
sigqueue(pid, signo, qval);
}
```

3.8 asynchronous I/O

Um das blocking bei slow system calls zu umgehen (speziell read/write) gibt es die Möglichkeit des asynchronous I/O, hier wird ein Signal ausgelöst wenn die Daten da sind, bzw geschrieben sind. Dies ist nicht Bestandteil von POSIX.1 aber z.B. Spec1170:

```
#include <unistd.h>
#include <stropts.h>
#include <fcntl.h>

if ((fd = open(pathname, O_RDONLY | O_NONBLOCK)) == -1)
/* handle errors here */

if (ioctl(fd, I_SETSIG, S_RDNORM) == -1)
/* could not set file descriptor for asynchronous read
*/
```

Nach dem des Streams Öffnen mit O_NONBLOCK wird der filedescriptor (=int zahl fd) auf asynchronous I/O mittels Signalen eingestellt. I_SETSIG bedeutet dass das Signal SIGPOLL erzeugt wird und RD_NORM bedeutet dass es erzeugt wird wenn etwas am Stream ankommt. (man streamio).

```
streamio (7) SUN
....
I_SETSIG Tells the Stream head that the user process wants a
SIGPOLL
          signal to be issued by the kernel for a particular
event
          that can occur on a Stream.
          This command provides support for
          asynchronous processing in STREAMS.
..
          The arg parameter contains a bitmask specifying the
particular
          events that SIGPOLL is to be sent for. The value is
the
          bitwise-OR of any combination of the following
constants:
....
           S_RDNORM Specifies that an ordinary message has
arrived
                    on the read queue of the Stream head. This
bit
                    is set even for zero length messages.
```

Es gibt ausserdem eine POSIX.1b asynchronous I/O extension, siehe hierzu aio_read(), aio_write(), aio_return(), aio_error().

```
aio(3) SUN
....
#include <aio.h>
ssize_t aio_return(struct aiocb * aiocbp);
int aio_error(const struct aiocb * aiocbp);
int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
```

Wobei die Struktur ähnlich den read/write Parametern ist

```
struct aiocb {
int aio_fildes; /* file descriptor */
volatile void *aio_buf; /* buffer location */
size_t aio_nbytes; /* length of transfer */
off_t aio_offset; /* file offset */
int aio_reqprio; /* request priority offset */
struct sigevent aio_sigevent; /* signal number and
offset */
int aio_lio_opcode; /* listio operation */
};
```

Es wird der request (read oder write) abgesetzt, das Programm geht weiter und mittels aio_error() wird nachgeschaut ob der Wert noch EINPROGRESS ist. Wenn aio_error() einen anderen Wert liefert, kann mit aio_return() geschaut werden, wie read/write geendet hat, dann ist der Rückgabewert wie bei read/write. Die Signale verwenden die Möglichkeit der Datenübergabe.

```
#include <aio.h>
#include <errno.h>
#include <signal.h>
struct aiocb my_aiocb;
struct sigaction my_sigaction;
void my_aio_handler(int, siginfo_t *, void *);
......
my_sigaction.sa_flags = SA_SIGINFO;
my_sigaction.sa_sigaction = my_aio_handler;
sigsetempty(&my_sigaction.sa_mask);
(void) sigaction(SIGRTMIN, &my_sigaction, NULL);
my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
/* es soll ein signal erzeugt werden */

my_aiocb.aio_sigevent.sigev_signo = SIGRTMIN;
/* dieses signal wird erzeugt */

my_aiocb.aio_sigevent.sigev_value.sival_ptr = &myaiocb;
/* daten zum signal */

(void) aio_read(&my_aiocb);

void my_aio_handler(int signo, siginfo_t *siginfo, void
*context)
{
int my_errno;
struct aiocb *my_aiocbp;

my_aiocbp = siginfo.si_value.sival_ptr;
if ((my_errno = aio_error(my_aiocb)) != EINPROGRESS) {
    int my_status = aio_return(my_aiocb);
    if (my_status >= 0){ /* start another operation */
                    ... }
    else { /* handle I/O error */ ... }
    }
}
```