Signale werden für vielfältige Zwecke eingesetzt. Sie können verwendet werden,

- um den normalen Ablauf eines Prozesses für einen wichtigen Hinweis zu unterbrechen,
- um die Ausführung eines Prozesses zu suspendieren,
- um die Terminierung eines Prozesses zu erbitten oder zu erzwingen und
- um schwerwiegende Fehler bei der Ausführung zu behandeln wie z.B. den Verweis durch einen invaliden Zeiger.

- Signale sind unter UNIX die einzige Möglichkeit, den normalen Programmablauf eines Prozesses zu unterbrechen.
- Signale werden durch kleine natürliche Zahlen repräsentiert, die in jeder UNIX-Umgebung fest vordefiniert sind.
- Darüber hinaus stehen kaum weitere Informationen zur Verfügung. Signale ersetzen daher keine Interprozeßkommunikation.
- Signale können von verschiedenen Parteien ausgelöst werden: Von anderen Prozessen, die die dafür notwendige Berechtigung haben (entweder der gleiche Benutzer oder der Super-User), durch den Prozess selbst entweder indirekt (durch einen schwerwiegenden Fehler) oder explizit oder auch durch das Betriebssystem.

- Der ISO-Standard 9899-2011 für die Programmiersprache C definiert eine einfache und damit recht portable Schnittstelle für die Behandlung von Signalen. Hier gibt es neben der Signalnummer selbst keine weiteren Informationen.
- Der IEEE Standard 1003.1 (POSIX) bietet eine Obermenge der Schnittstelle des ISO-Standards an, bei der wenige zusätzliche Informationen (wie z.B. die Angabe des invaliden Zeigers) dabei sein können und der insbesondere eine sehr viel feinere Kontrolle der Signalbehandlung erlaubt.

Die Terminalschnittstelle unter UNIX wurde ursprünglich für ASCII-Terminals mit serieller Schnittstelle entwickelt, die nur folgende Eingabemöglichkeiten anboten:

- Einzelne ASCII-Zeichen, jeweils ein Byte (zusammen mit etwas Extra-Kodierung wie Prüf- und Stop-Bits).
- Ein BREAK, das als spezielles Signal repräsentiert wird, das länger als die Kodierung für ein ASCII-Zeichen währt.
- Ein HANGUP, bei dem ein Signal wegfällt, das zuvor die Existenz der Leitung bestätigt hat. Dies benötigt einen weiteren Draht in der seriellen Leitung.

Diese Eingaben werden auf der Seite des Betriebssystems vom Terminal-Treiber bearbeitet, der in Abhängigkeit von den getroffenen Einstellungen

- die eingegebenen Zeichen puffert und das Editieren der Eingabe ermöglicht (beispielsweise mittels BACKSPACE, CTRL-u und CTRL-w) und
- bei besonderen Eingaben Signale an alle Prozesse schickt, die mit diesem Terminal verbunden sind.

Ziel war es, dass im Normalfall ein BREAK zu dem Abbruch oder zumindest der Unterbrechung der gerade laufenden Anwendung führt. Und ein HANGUP sollte zu dem Abbruch der gesamten Sitzung führen, da bei einem Wegfall der Leitung keine Möglichkeit eines regulären Abmeldens besteht.

Heute sind serielle Terminals rar geworden, aber das Konzept wurde dennoch beibehalten:

- Zwischen einem virtuellen Terminal (beispielsweise einem xterm) und den Prozessen, die zur zugehörigen Sitzung gehören, ist ein sogenanntes Pseudo-Terminal im Betriebssystem geschaltet, das der Sitzung die Verwendung eines klassischen Terminals vorspielt.
- Da es BREAK in diesem Umfeld nicht mehr gibt, wird es durch ein beliebiges Zeichen ersetzt wie beispielsweise CTRL-c.
- Wenn das virtuelle Terminal wegfällt (z.B. durch eine gewaltsame Beendigung der xterm-Anwendung), dann gibt es weiterhin ein HANGUP für die Sitzung.

- Auf fast alle Signale können Prozesse, die sie erhalten, auf dreierlei Weise reagieren:
  - Voreinstellung: Normalerweise die Terminierung des Prozesses. (*SIG_DFL*)
  - Ignorieren. (*SIG_IGN*)
  - Bearbeitung durch einen Signalbehandler.
- Es mag harsch erscheinen, dass die Voreinstellung fast durchweg zur Terminierung eines Prozesses führt. Aber genau dies führt bei normalen Anwendungen genau zu den gewünschten Effekten wie Abbruch des laufenden Programms bei BREAK (die Shell ignoriert das Signal) und Abbau der Sitzung bei HANGUP.
- Wenn ein Prozess diese Signale ignoriert, sollte es genau wissen, was es tut, da der Nutzer auf diese Weise eine wichtige Kontrollmöglichkeit seiner Sitzung verliert.

sigint.c

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

volatile sig_atomic_t signal_caught = 0;

void signal_handler(int signal) {
   signal_caught = signal;
}

int main() {
   if (signal(SIGINT, signal_handler) == SIG_ERR) {
      perror("unable to setup signal handler for SIGINT");
      exit(1);
   }
   printf("Try to send a SIGINT signal!\n");
   int counter = 0;
   while (!signal_caught) {
      for (int i = 0; i < counter; ++i);
      ++counter;
   }
   printf("Got signal %d after %d steps!\n", signal_caught, counter);
}
```

- Dieses Beispiel demonstriert die Behandlung des Signals *SIGINT*, das dem BREAK entspricht.

sigint.c

```
volatile sig_atomic_t signal_caught = 0;

void signal_handler(int signal) {
   signal_caught = signal;
}
```

- Die Deklaration für *signal_caught* wird noch genauer diskutiert. Zunächst kann davon ausgegangen werden, dass es sich dabei um eine globale ganzzahlige Variable handelt, die zu Beginn mit 0 initialisiert wird.
- Die Funktion *signal_handler* ist ein Signalbehandler. Als einziges Argument erhält sie die Nummer des eingetroffenen Signals, das es zu behandeln gilt. Einen Rückgabewert gibt es nicht.

sigint.c

```
if (signal(SIGINT, signal_handler) == SIG_ERR) {
   perror("unable to setup signal handler for SIGINT");
   exit(1);
}
```

- Mit der Funktion *signal* kann für eine Signalnummer (hier *SIGINT*) ein Signalbehandler (hier *signal_handler*) spezifiziert werden.
- Wenn die Operation erfolgreich war, wird der zuletzt eingesetzte Signalbehandler zurückgeliefert.
- Im Fehlerfall liefert *signal* den Wert *SIG_ERR*. (Damit ist normalerweise nicht zu rechnen, es sei denn, es werden nicht zulässige Einstellungen vorgenommen, wie etwa das Ignorieren von *SIG_KILL*.)

sigint.c

```
printf("Try to send a SIGINT signal!\n");
int counter = 0;
while (!signal_caught) {
   for (int i = 0; i < counter; ++i);
   ++counter;
}
printf("Got signal %d after %d steps!\n", signal_caught, counter);
```

- Das Hauptprogramm arbeitet eine Endlosschleife ab, die nur beendet werden kann, wenn auf „magische“ Weise die Variable *signal_caught* einen Wert ungleich 0 erhält.

sigint.c

```
while (!signal_caught) {
   for (int i = 0; i < counter; ++i);
   ++counter;
}
```

- Wenn ein optimierender Übersetzer die Schleife analysiert, könnten folgende Punkte auffallen:
  - Die Schleife ruft keine externen Funktionen auf.
  - Innerhalb der Schleife wird *signal_caught* nirgends verändert.
- Daraus könnte vom Übersetzer der Schluss gezogen werden, dass die Schleifenbedingung nur zu Beginn einmal überprüft werden muss. Findet der Eintritt in die Schleife statt, könnte der weitere Test der Bedingung ersatzlos wegfallen.
- Analysen wie diese sind für heutige optimierende Übersetzer Pflicht, um guten Maschinen-Code erzeugen zu können.
- Es wäre fatal, wenn darauf nur wegen der Existenz von asynchron aufgerufenen Signalbehandlern verzichtet werden würde.

sigint.c

```
volatile sig_atomic_t signal_caught = 0;
```

- Um beides zu haben, die fortgeschrittenen Optimierungstechniken und die Möglichkeit, Variablen innerhalb von Signalbehandlern setzen zu können, wurde in C die Speicherklasse **volatile** eingeführt.
- Damit lassen sich Variablen kennzeichnen, deren Wert sich jederzeit ändern kann — selbst dann, wenn dies aus dem vorliegenden Programmtext nicht ersichtlich ist.
- Entsprechend gilt dann auch in C, dass alle anderen Variablen, die nicht als **volatile** klassifiziert sind, sich nicht durch „magische“ Effekte verändern dürfen.

Damit die Effekte eines Signalbehandlers wohldefiniert sind, schränken sich die Möglichkeiten stark ein. So ist es nur zulässig,

- lokale Variablen zu verwenden,
- mit **volatile** deklarierte Variablen zu benutzen und
- Funktionen aufzurufen, die sich an die gleichen Spielregeln halten.

- Die Verwendung von Ein- und Ausgabe innerhalb eines Signalbehandlers ist nicht zulässig.
- Der ISO-Standard 9899-2011 nennt nur *abort*(), *_Exit*(), *quick_exit*() und *signal*() als zulässige Bibliotheksfunktionen.
- Beim POSIX-Standard werden noch zahlreiche weitere Systemaufrufe genannt.
- Auf den Manualseiten von Solaris wird dies dokumentiert durch die Angabe „Async-Signal-Safe“ bei „MT-Level“.
- Ansonsten ist nach expliziten Hinweisen zu suchen, ob eine Funktion mehrfach parallel ausgeführt werden darf, d.h. ob sie *reentrant* ist.
- Beispiele von Funktionen der Standard-Bibliothek, die nicht *reentrant* sind: *ctime* und *strtok*. Hier gibt es die alternativen Fassungen *ctime_r* und *strtok_r*, die *reentrant* sind.

- Variablenzugriffe sind nicht notwendigerweise atomar.
- Das hat zur Konsequenz, dass eine unterbrochende Variablenzuweisung möglicherweise nur teilweise durchgeführt worden ist. Auf einer 32-Bit-Maschine mit einem 32 Bit breiten Datenbus wäre es etwa denkbar, dass eine 64-Bit-Größe (etwa **long long** oder **double**) nur zur Hälfte kopiert ist, wenn eine Unterbrechung eintrifft.
- Dies bedeutet, dass im Falle einer Unterbrechung eine Variable nicht nur einen alten oder neuen Wert haben kann, sondern auch einen undefinierten.
- Um solche Probleme auszuschließen, bietet der ISO-Standard 9899-1999 den ganzzahligen Datentyp *sig_atomic_t* an, der in <*signal.h*> definiert ist.
- Bei Zugriffen auf Variablen dieses Typs wird im Falle einer Unterbrechung nur der alte oder der neue Wert beobachtet, jedoch nie ein undefinierter.
- *sig_atomic_t* wird typischerweise in Kombination mit **volatile** verwendet.

Wecksignale mit *alarm*

sigalrm.c

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

static volatile sig_atomic_t time_exceeded = 0;

static void alarm_handler(int signal) {
   time_exceeded = 1;
}

int main() {
   if (signal(SIGALRM, alarm_handler) == SIG_ERR) {
      perror("unable to setup signal handler for SIGALRM");
      exit(1);
   }
   alarm(2);
   puts("Na, koennen Sie innerhalb von zwei Sekunden etwas eingeben?");
   int ch = getchar();
   if (time_exceeded) {
      puts("Das war wohl nichts.");
   } else {
      puts("Gut!");
   }
}
```

sigalrm.c

```
if (signal(SIGALRM, alarm_handler) == SIG_ERR) {
   perror("unable to setup signal handler for SIGALRM");
   exit(1);
}
alarm(2);
```

- Für jeden Prozess verwaltet UNIX einen Wecker, der entweder ruht oder zu einem spezifizierten Zeitpunkt sich mit dem Signal *SIGALRM* meldet.
- Der Wecker wird mit *alarm* gestellt. Dabei wird die zu verstreichende Zeit in Sekunden angegeben.
- Mit einer Angabe von 0 lässt sich der Wecker ausschalten.

tread.h

```
#ifndef TREAD_H
#define TREAD_H

#include <unistd.h>

int timed_read(int fd, void* buf, size_t nbytes, unsigned seconds);

#endif
```

- Mit Hilfe des Weckers lässt sich der Systemaufruf *read* zu *timed_read* erweitern, das ein Zeitlimit berücksichtigt.
- Falls das Zeitlimit erreicht wird, ist kein Fehler, sondern es wird ganz schlicht 0 zurückzugeben.
- Wie bereits beim vorherigen Beispiel wird hier ausgenutzt, dass nicht nur normale Programmabläufe, sondern auch einige Systemaufrufe wie etwa *read* unterbrechbar sind.

tread.c

```
#include <signal.h>
#include <unistd.h>
#include "tread.h"

static volatile sig_atomic_t time_exceeded = 0;

static void alarm_handler(int signal) {
   time_exceeded = 1;
}
```

- Der Signalbehandler für *SIGALRM* arbeitet wie gehabt. Allerdings wird im Unterschied zu zuvor die Variable und der Behandler **static** deklariert, damit diese Deklarationen privat bleiben und nicht in Konflikt zu anderen Deklarationen stehen.

Lese-Operationen mit Zeitlimit

tread.c

```
int timed_read(int fd, void* buf, size_t nbytes, unsigned seconds) {
   if (seconds == 0) return 0;
   /*
    * setup signal handler and alarm clock but
    * remember the previous settings
    */
   void (*previous_handler)(int) = signal(SIGALRM, alarm_handler);
   if (previous_handler == SIG_ERR) return -1;
   time_exceeded = 0;
   int remaining_seconds = alarm(seconds);
   if (remaining_seconds > 0) {
      if (remaining_seconds <= seconds) {
         remaining_seconds = 1;
      } else {
         remaining_seconds -= seconds;
      }
   }

   int bytes_read = read(fd, buf, nbytes);

   /* restore previous settings */
   if (!time_exceeded) alarm(0);
   signal(SIGALRM, previous_handler);
   if (remaining_seconds) alarm(remaining_seconds);

   if (time_exceeded) return 0;
   return bytes_read;
}
```

tread.c

```
void (*previous_handler)(int) = signal(SIGALRM, alarm_handler);
```

- Aus der Sicht einer Bibliotheksfunktion muss damit gerechnet werden, dass auch noch andere Parteien einen Wecker benötigen und deswegen *alarm* aufrufen.
- Deswegen ist es sinnvoll, die eigene Nutzung so zu gestalten, dass die Weckfunktion für die anderen nicht sabotiert wird.
- Dies ist prinzipiell möglich, weil *signal* den gerade eingesetzten Signalbehandler im Erfolgsfalle zurückliefert. Dieser wird hier der Variablen *previous_handler* zugewiesen.

tread.c

```
time_exceeded = 0;
int remaining_seconds = alarm(seconds);
if (remaining_seconds > 0) {
   if (remaining_seconds <= seconds) {
      remaining_seconds = 1;
   } else {
      remaining_seconds -= seconds;
   }
}
```

- Die gleiche Rücksichtnahme erfolgt bei dem Aufruf von *alarm*.
- Im Erfolgsfalle liefert *alarm* den Wert 0, falls zuvor der Wecker ruhte oder einen positiven Wert, der die zuvor noch verbliebenen Sekunden bis zum Signal spezifiziert.
- Die Variable *remaining_seconds* wird auf den Wert gesetzt, den wir abschließend verwenden, um den Wecker neu zu stellen, nachdem er in dieser Funktion nicht mehr benötigt wird.

- *read* hat in diesem Szenario verschiedene Möglichkeiten, zurückzukommen. Erstens kann *read* ganz normal etwas einlesen (positiver Rückgabewert), es kann ein Eingabeende vorliegen (Rückgabewert gleich 0) oder es kann ein Fehler eintreten (negativer Rückgabewert).
- Im Falle einer Unterbrechung durch ein Signal bricht der Systemaufruf mit einem Fehler ab, d.h. es wird -1 zurückgeliefert. Die Variable *errno* hat dann den Wert *EINTR*.
- Wenn *read* unterbrochen wird und mit -1 endet, wurde nichts weggelesen. Ein unterbrochener *write*-Systemaufruf, der -1 liefert, hat nichts geschrieben. Wenn *read* bzw. *write* bereits gelesen bzw. geschrieben haben, wenn sie unterbrochen werden, dann liefern sie nicht -1, sondern die Zahl der bereits gelesenen bzw. geschriebenen Bytes zurück.
- In diesem Beispiel wird jedoch nicht *errno* überprüft, sondern die Variable *time_exceeded* untersucht.

tread.c

```
int bytes_read = read(fd, buf, nbytes);

/* restore previous settings */
if (!time_exceeded) alarm(0);
signal(SIGALRM, previous_handler);
if (remaining_seconds) alarm(remaining_seconds);

if (bytes_read < 0 && time_exceeded) return 0;
return bytes_read;
```

- Bevor *alarm* erneut aufgesetzt wird, muss zuvor der alte Signalbehandler restauriert werden.
- Wenn dies in umgekehrter Reihenfolge geschehen würde, dann gibt es ein kleines Zeitfenster, in dem das Signal *SIGALRM* eintreffen könnte, noch bevor es zum Aufruf von *signal* kam.
- In diesem Falle würde der andere Signalbehandler nicht wie geplant aufgerufen werden.
- Daher wird hier zuerst der alte Signalbehandler eingesetzt, bevor *alarm* aufgerufen wird. Auf diese Weise wird das Fenster geschlossen.

In Erweiterung zu *alarm* stehen die Funktionen der POSIX-Realtime-Erweiterung optional zur Verfügung. Hierzu gehören insbesondere *timer_create* und *timer_set*:

- Die Timer sind dann nicht nur sekundengenau, sondern können mit der maximal auf der Plattform erreichbaren Genauigkeit spezifiziert werden.
- Beliebig viele Timer können parallel laufen.
- Periodisch sich wiederholende Signale können konfiguriert werden.