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 9899-1999 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: Terminierung des Prozesses.
  - Ignorieren.
  - Bearbeitung durch einen Signalbehandler.
- Es mag harsch erscheinen, dass die Voreinstellung 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.
- Ein Fehlerfall wird mit *SIG_ERR* quittiert.

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.

Die Falle eines optimierenden Übersetzers

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 9899-1999 nennt nur *abort*(), *_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.

- 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!");
   }
}
```

Wecksignale mit *alarm*

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.

Lese-Operationen mit Zeitlimit

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.

- Grundsätzlich kann ein Prozess einem anderen Prozess (einschliesslich sich selbst) ein Signal senden.
- Voraussetzung ist dabei unter UNIX, dass der andere Prozess dem gleichen Benutzer gehört oder der das Signal versendende Prozess mit Superuser-Privilegien arbeitet.
- Der ISO-Standard für C sieht zum Signalversand nur eine Funktion *raise*() vor, die es erlaubt, ein Signal an den eigenen Prozess zu versenden.
- Im POSIX-Standard kommt der Systemaufruf *kill*() hinzu, der es erlaubt, ein Signal an einen anderen Prozess zu verschicken, sofern die dafür notwendigen Privilegien vorliegen.

killparent.c

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

void sigterm_handler(int signo) {
   const char msg[] = "Goodbye, cruel world!\n";
   write(1, msg, sizeof msg - 1);
   _Exit(1);
}

int main() {
   if (signal(SIGTERM, sigterm_handler) == SIG_ERR) {
      perror("signal"); exit(1);
   }

   pid_t child = fork();
   if (child == 0) {
      kill(getppid(), SIGTERM);
      exit(0);
   }
   int wstat;
   wait(&wstat);
   exit(0);
}
```

killparent.c

```
kill(getppid(), SIGTERM);
```

- Der Systemaufruf *kill* benötigt zwei Parameter, wobei der erste die Prozess-ID des Signalempfängers und der zweite Parameter das zu versendende Signal nennt.
- Das Versenden von *SIGTERM* gilt per Konvention als „freundliche“ Bitte, den Prozess zu terminieren.
- Der Empfänger erhält so die Gelegenheit, Aufräumarbeiten vorzunehmen, bevor er abschließt.
- Alternativ zu *SIGTERM* gibt es auch *SIGKILL*, das sich nicht behandeln lässt, d.h. das den Empfänger unter keinen Umständen mehr zum Zuge kommen lässt.

killparent.c

```
void sigterm_handler(int signo) {
   const char msg[] = "Goodbye, cruel world!\n";
   write(1, msg, sizeof msg - 1);
   _Exit(1);
}
```

- Hier ist vorgesehen, dass der Signalbehandler im Falle von *SIGTERM* noch eine Meldung ausgibt, bevor der Prozess terminiert wird.
- Da die Verwendung von Funktionen der *stdio* wie etwa *puts* innerhalb von Signalbehandlern tabu ist, wird hier der Systemaufruf *write* verwendet.
- Ebenfalls tabu ist *exit*, da dabei Funktionen der *stdio* zur Leerung aller Puffer aufgerufen werden.
- Alternativ kann die Funktion *_Exit* aufgerufen werden, die mit dem ISO-Standard 9899-1999 eingeführt wurde. Diese umgeht sämtliche Aufräumarbeiten und terminiert unmittelbar den aufrufenden Prozess.

- Der Systemaufruf *kill*() erfüllt aber auch noch einen weiteren Zweck. Bei einer Signalnummer von 0 wird nur die Zulässigkeit des Signalversendens überprüft.
- Dies kann dazu ausgenutzt werden, um die Existenz eines Prozesses zu überprüfen.
- Mit folgenden Fehler-Codes ist dabei zu rechnen:
  - *ESRCH*: Die genannte Prozess-ID ist zur Zeit nicht vergeben.
  - *EPERM*: Die genannte Prozess-ID existiert, aber es fehlen die Privilegien, dem Prozess ein Signal zu senden.

Schonendes Anklopfen

waitfor.c

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

int main(int argc, char** argv) {
   char* cmdname = *argv++; --argc;
   if (argc != 1) {
      fprintf(stderr, "Usage: %s pid\n", cmdname);
      exit(1);
   }

   /* convert first argument to pid */
   char* endptr = argv[0];
   pid_t pid = strtol(argv[0], &endptr, 10);
   if (endptr == argv[0]) {
      fprintf(stderr, "%s: integer expected as argument\n",
         cmdname);
      exit(1);
   }

   while (kill(pid, 0) == 0) sleep(1);

   if (errno == ESRCH) exit(0);
   perror(cmdname); exit(1);
}
```

- Gelegentlich kommt es vor, dass Prozesse nur auf das Eintreffen eines Signals warten möchten und sonst nichts zu tun haben.
- Theoretisch könnte ein Prozess dann in eine Dauerschleife mit leerem Inhalt treten (auch *busy loop* bezeichnet).
- Dies wäre jedoch nicht sehr fair auf einem System mit mehreren Prozessen, da dadurch Rechenzeit vergeudet würde.
- Abhilfe schafft hier der Systemaufruf *pause*(), der einen Prozess schlafen legt, bis ein Signal eintrifft.

Auf eintreffende Signale warten

pingpong.c

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

static volatile sig_atomic_t sigcount = 0;

void sighandler(int sig) {
   ++sigcount;
   if (signal(sig, sighandler) == SIG_ERR) _Exit(1);
}

int main() {
   /* this signal setting is inherited to our child */
   if (signal(SIGUSR1, sighandler) == SIG_ERR) {
      perror("signal SIGUSR1"); exit(1);
   }

   pid_t parent = getpid();
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      sigcount = 1; /* give the ball to the child... */
      playwith(parent);
   } else {
      playwith(child);
   }
}
```

Auf eintreffende Signale warten

pingpong.c

```
static void playwith(pid_t partner) {
   for(int i = 0; i < 10; ++i) {
      if (!sigcount) pause();
      printf("[%d] send signal to %d\n",
         (int) getpid(), (int) partner);
      if (kill(partner, SIGUSR1) < 0) {
         printf("[%d] %d is no longer alive\n",
            (int) getpid(), (int) partner);
         return;
      }
      --sigcount;
   }
   printf("[%d] finishes playing\n", (int) getpid());
}
```

- Mit *pause* wartet der aufrufende Prozess bis zum Eintreffen eines Signals. Wenn dieser Systemaufruf beendet wird, ist das Resultat immer negativ und *errno* ist auf *EINTR* gesetzt.

Auf eintreffende Signale warten

pingpong.c

```
static volatile sig_atomic_t sigcount = 0;
void sighandler(int sig) {
   ++sigcount;
   if (signal(sig, sighandler) == SIG_ERR) _Exit(1);
}

   /* ... */
   if (signal(SIGUSR1, sighandler) == SIG_ERR) {
      perror("signal SIGUSR1"); exit(1);
   }
   /* ... */
```

- *SIGUSR1* gehört zusammen mit *SIGUSR2* zu den Signalen ohne Sonderbedeutung, die problemlos für Zwecke der Prozesskommunikation verwendet werden können.
- Wenn *sighandler* noch vor *fork* als Signalbehandler installiert wird, dann erbt auch der neu erzeugte Prozess diese Einstellung.
- *sighandler* installiert sich selbst erneut, da der ISO-Standard 9899-1999 offen lässt, ob der Signalbehandler nach dem Eintreffen des Signals installiert bleibt oder nicht.

Die vorangegangenen Beispiele werfen die Frage auf, wie UNIX bei der Zustellung von Signalen vorgeht, wenn

- der Prozess zur Zeit nicht aktiv ist,
- gerade ein Systemaufruf für den Prozess abgearbeitet wird oder
- gerade ein Signalbehandler bereits aktiv ist.

Vom ISO-Standard 9899-1999 für C wird in dieser Beziehung nichts festgelegt.

Der POSIX-Standard geht jedoch genauer darauf ein:

- Wenn ein Prozess ein Signal erhält, wird dieses Signal zunächst in den zugehörigen Verwaltungsstrukturen des Betriebssystems vermerkt. Signale, die für einen Prozess vermerkt sind, jedoch noch nicht zugestellt worden sind, werden als *anhängige* Signale bezeichnet.
- Wenn mehrere Signale mit der gleichen Nummer anhängig sind, ist nicht festgelegt, ob eine Mehrfachzustellung erfolgt. Es können also Signale wegfallen.
- Nur aktiv laufende Prozesse können Signale empfangen. Prozesse werden normalerweise durch die Existenz eines anhängigen Signals aktiv — aber dieses kann auch längere Zeit in Anspruch nehmen, wenn dem zwischenzeitlich mangelnde Ressourcen entgegenstehen.
- Für jeden Prozess gibt es eine Menge blockierter Signale, die im Augenblick nicht zugestellt werden sollen. Dies hat nichts mit dem Ignorieren von Signalen zu tun, da blockierte Signale anhängig bleiben, bis die Blockierung aufgehoben wird.

- Der POSIX-Standard legt nicht fest, was mit der Signalbehandlung geschieht, wenn ein Signalbehandler aufgerufen wird.
- Möglich ist das Zurückfallen auf *SIG_DFL* (Voreinstellung mit Prozeßterminierung) oder die temporäre automatische Blockierung des Signals bis zur Beendigung des Signalbehandlers.
- Alle modernen UNIX-Systeme wählen die zweite Variante.
- Dies lässt sich aber gemäß dem POSIX-Standard auch erzwingen, indem die umfangreichere Schnittstelle *sigaction*() anstelle von *signal*() verwendet wird. Allerdings ist *sigaction*() nicht mehr Bestandteil des ISO-Standards für C.

- UNIX unterscheidet zwischen unterbrechbaren und unterbrechungsfreien Systemaufrufen. Zur ersteren Kategorie gehören weitgehend alle Systemaufrufe, die zu einer längeren Blockierung eines Prozesses führen können.
- Ist ein nicht blockiertes Signal anhängig, kann ein unterbrechbarer Systemaufruf aufgrund des Signals mit einer Fehlerindikation beendet werden. *errno* wird dann auf *EINTR* gesetzt.
- Dabei ist zu beachten, dass der unterbrochene Systemaufruf nach Beendigung der Signalbehandlung *nicht* fortgesetzt wird, sondern manuell erneut gestartet werden muss.
- Dies kann leider zu unerwarteten Überraschungseffekten führen, weil insbesondere auch die *stdio*-Bibliothek keinerlei Vorkehrungen trifft, Systemaufrufe automatisch erneut aufzusetzen, falls es zu einer Unterbrechung kam.
- Dies ist eine wesentliche Schwäche sowohl des POSIX-Standards als auch der *stdio*-Bibliothek und ein Grund mehr dafür, auf die Verwendung der *stdio* in kritischen Anwendungen völlig zu verzichten.

- Für die genauere Regulierung der Signalbehandlung bietet POSIX (jedoch nicht ISO-C) den Systemaufruf *sigaction* an. Während bei *signal* zur Spezifikation der Signalbehandlung nur ein Funktionszeiger genügte, kommen bei der **struct** *sigaction*, die *sigaction*() verwendet, die in der folgenden Tabelle genannten Felder zum Einsatz:

Datentyp	Feldname	Beschreibung
void(∗)(**int**)	*sa_handler*	Funktionszeiger (wie bisher)
void(∗)(**int**, *siginfo_t*∗, **void**∗)	*sa_sigaction*	alternativer Zeiger auf einen Signalbehandler, der mehr Informationen zum Signal erhält
sigset_t	*sa_mask*	Menge von Signalen, die während der Signalbehandlung dieses Signals zu blockieren sind
int	*sa_flags*	Menge von Boolean-wertigen Optionen

strikeback.c

```
volatile int signo = 0;
volatile pid_t pid = 0;

void sighandler(int sig, siginfo_t* siginfo, void* context) {
   signo = sig;
   pid = siginfo->si_pid;
   if (pid) { /* strike back */
      kill(pid, sig);
   }
}

int main() {
   int signals[] = {SIGHUP, SIGINT, SIGTERM, SIGUSR1, SIGUSR2};
   struct sigaction sigact = {0};
   sigact.sa_sigaction = sighandler;
   sigact.sa_flags = SA_SIGINFO;
   for (int index = 0; index < sizeof(signals)/sizeof(int); ++index) {
      signo = signals[index];
      if (sigaction(signo, &sigact, 0) < 0) {
         perror("sigaction"); exit(1);
      }
   }
   for(;;) {
      pause();
      if (signo) {
         printf("got signal %d from %d\n", signo, (int) pid);
         fflush(stdout);
      }
   }
}
```

- Bei der *sigaction*-Schnittstelle ist es möglich, die Zustellung einiger Signale aufzuhalten während einer Signalbehandlung.
- Dies betrifft implizit das gerade empfangene Signal und auch mögliche weitere Signale. Letzteres wird über das Feld *sa_mask* spezifiziert.
- Blockierte Signale sind dann zunächst anhängig und warten dann darauf, dass der Block aufgehoben wird.
- Wenn mehrfach das gleiche blockierte Signal eintrifft, dann ist nicht definiert, ob dies auch mehrfach zugestellt wird, sobald der Block aufgehoben wird.
- Es kann somit zum Verlust an Signalen kommen.

sigfire.c

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

static const int NOF_SIGNALS = 1000;
static volatile sig_atomic_t received_signals = 0;
static volatile sig_atomic_t terminated = 0;

static void count_signals(int sig) {
   ++received_signals;
}

void termination_handler(int sig) {
   terminated = 1;
}
```

- Dieses Beispiel soll den potentiellen Verlust von Signalen demonstrieren, indem gezählt wird, wieviel von insgesamt 1000 verschickten Signalen ankommen.

Verlust von Signalen

sigfire.c

```
int main() {
   sighold(SIGUSR1); sighold(SIGTERM);
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      struct sigaction action = {0};
      action.sa_handler = count_signals;
      if (sigaction(SIGUSR1, &action, 0) != 0) {
         perror("sigaction"); exit(1);
      }
      action.sa_handler = termination_handler;
      if (sigaction(SIGTERM, &action, 0) != 0) {
         perror("sigaction"); exit(1);
      }
      sigrelse(SIGUSR1); sigrelse(SIGTERM);
      while (!terminated) pause();
      printf("[%d] received %d signals\n",
         (int) getpid(), received_signals);
      exit(0);
   }
   sigrelse(SIGUSR1); sigrelse(SIGTERM);

   for (int i = 0; i < NOF_SIGNALS; ++i) {
      kill(child, SIGUSR1);
   }
   printf("[%d] sent %d signals\n", (int) getpid(), NOF_SIGNALS);
   kill(child, SIGTERM); wait(0);
}
```

sigfire.c

```
sighold(SIGUSR1); sighold(SIGTERM);
/* ... */
sigrelse(SIGUSR1); sigrelse(SIGTERM);
```

- Mit der Funktion *sighold* kann ein Signal auch außerhalb eines Signalbehandlers explizit geblockt werden.
- Mit *sigrelse* kann dies wieder rückgängig gemacht werden.
- Auf diese Weise können kritische Bereiche geschützt werden.

- Mit Hilfe der Funktionen *wait*() oder *waitpid*() wird die Terminierung erzeugter Prozesse *synchron* abgewickelt.
- Gelegentlich ist es auch sinnvoll, sich die Terminierung über Signale *asynchron* mitteilen zu lassen. Dies geht mit dem Signal *SIGCHLD*, das an den Erzeuger versendet wird, sobald eine der von ihm erzeugten Prozesse terminiert.
- Per Voreinstellung wird dieses Signal ignoriert.

sigchld.c

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
#include "processlist.h"

static processlist alive, dead;

void child_term_handler(int sig) {
   pid_t pid; int wstat;
   while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
      if (pl_move(&alive, &dead, pid)) {
         pl_modify(&dead, pid, wstat);
      }
   }
}
```

- In diesem Beispiel werden zahlreiche Prozesse erzeugt, deren Exit-Status zeitnah in einer Datenstruktur verwaltet wird.

Signale als Indikatoren für terminierte Prozesse 106

sigchld.c

```
int main() {
   struct sigaction action = {0};
   action.sa_handler = child_term_handler;
   if (sigaction(SIGCHLD, &action, 0) != 0) {
      perror("sigaction");
   }
   pl_alloc(&alive, 4); pl_alloc(&dead, 4);
   sighold(SIGCHLD);
   for (int i = 0; i < 10; ++i) {
      fflush(0);
      pid_t child = fork();
      if (child < 0) {
         perror("fork"); exit(1);
      }
      if (child == 0) {
         srand(getpid()); sleep(rand() % 5); exit((char) rand());
      }
      pl_add(&alive, child, 0);
   }
   sigrelse(SIGCHLD);
   while (pl_length(&alive) > 0 || pl_length(&dead) > 0) {
      if (pl_length(&dead) == 0) pause();
      while (pl_length(&dead) > 0) {
         sighold(SIGCHLD);
         int wstat; pid_t pid = pl_pick(&dead, &wstat);
         sigrelse(SIGCHLD);
         printf("[%d] %d\n", (int) pid, WEXITSTATUS(wstat));
      }
   }
}
```

sigchld.c

```
#ifndef PROCESSLIST_H
#define PROCESSLIST_H

typedef struct process {
   pid_t pid; int wstat;
   struct process* next;
} process;

typedef struct processlist {
   unsigned int size, length;
   process** bucket; /* hash table */
   unsigned int it_index;
   process* it_entry;
} processlist;

// All functions with the exception of pl_length, pl_next,
// and pl_pick return 1 on success, 0 in case of failures.

/* allocate a hash table for processes with the given bucket size */
int pl_alloc(processlist* pl, unsigned int size);

/* add tuple (pid,wstat) to the process list, pid must be unique */
int pl_add(processlist* pl, pid_t pid, int wstat);

/* modify wstat for a given pid */
int pl_modify(processlist* pl, pid_t pid, int wstat);
```

sigchld.c

```
/* delete tuple by pid */
int pl_remove(processlist* pl, pid_t pid);

/* move entry for pid to another list */
int pl_move(processlist* from, processlist* to, pid_t pid);

/* return number of elements */
unsigned int pl_length(processlist* pl);

/* lookup wstat by pid */
int pl_lookup(processlist* pl, pid_t pid, int* wstat);

/* start iterator */
int pl_start(processlist *pl);

/* fetch next pid from iterator; returns 0 on end */
pid_t pl_next(processlist *pl);

/* pick and remove one element out of the list */
pid_t pl_pick(processlist *pl, int* wstat);

/* free allocated memory */
int pl_free(processlist* pl);

#endif
```

```
doolin$ tinysh
% cat >OUT
Some input...
^Cdoolin$
```

- Die zuvor vorgestellte Shell *tinysh* kümmerte sich nicht um die Signalbehandlung.
- Entsprechend führt ein *SIGINT* auf dem kontrollierenden Terminal nicht nur zum Abbruch des aufgerufenen Kommandos, sondern auch unerfreulicherweise zum abrupten Ende von *tinysh*.

Wie muss also die Signalbehandlung einer Shell aussehen?

- Wenn ein Kommando *im Vordergrund* läuft, muss die Shell die Signale *SIGINT* und *SIGQUIT* ignorieren.
- Wenn ein Kommando **im Hintergrund** läuft, müssen für diesen Prozess *SIGINT* und *SIGQUIT* ignoriert werden.
- Wenn die Shell ein Kommando einliest, sollten *SIGINT* und *SIGQUIT* die Neu-Eingabe des Kommandos ermöglichen.
- Bezüglich *SIGHUP* muss nichts unternommen werden.

Signalbehandlung in einer Shell

tiny2sh.c

```
static volatile sig_atomic_t interrupted = 0;

void interrupt_handler(int sig) {
   interrupted = 1;
}

int main() {
   struct sigaction action = {0};
   action.sa_handler = interrupt_handler;
   if (sigaction(SIGINT, &action, 0) != 0 ||
         sigaction(SIGQUIT, &action, 0) != 0) {
      perror("sigaction");
   }

   stralloc line = {0};
   while (getline(&line)) {
      strlist tokens = {0};
      stralloc_0(&line); /* required by tokenizer() */
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      command cmd = {0};
      if (!scan_command(&tokens, &cmd)) continue;

      sighold(SIGINT); sighold(SIGQUIT);
      // ... fork & (exec | wait) ...
      sigrelse(SIGINT); sigrelse(SIGQUIT);
   }
}
```

tiny2sh.c

```
sighold(SIGINT); sighold(SIGQUIT);
pid_t child = fork();
if (child == -1) {
   perror("fork"); continue;
}
if (child == 0) {
   sigrelse(SIGINT); sigrelse(SIGQUIT);
   if (cmd.background) {
      sigignore(SIGINT); sigignore(SIGQUIT);
   }
   exec_command(&cmd);
   perror(cmd.cmdname);
   exit(255);
}

if (cmd.background) {
   printf("%d\n", (int)child);
} else {
   int wstat;
   pid_t pid = waitpid(child, &wstat, 0);
   if (!WIFEXITED(wstat) || WEXITSTATUS(wstat)) {
      print_child_status(pid, wstat);
   }
}
sigrelse(SIGINT); sigrelse(SIGQUIT);
```

tiny2sh.c

```
int getline(stralloc* line) {
   int first = 1;
   interrupted = 0;
   for(;;) {
      if (interrupted) {
         interrupted = 0;
         printf("\n");
         first = 1;
      }
      if (first) {
         status_report();
         printf("%% ");
         first = 0;
      }
      errno = 0;
      if (readline(stdin, line)) return 1;
      if (errno != EINTR) return 0;
   }
}
```

tiny2sh.c

```
void print_child_status(pid_t pid, int wstat) {
   printf("[%d] ", (int) pid);
   if (WIFEXITED(wstat)) {
      printf("exit %d", WEXITSTATUS(wstat));
   } else if (WIFSIGNALED(wstat)) {
      printf("terminated with signal %d", WTERMSIG(wstat));
      if (WCOREDUMP(wstat)) printf(" (core dump)");
   } else {
      printf("???");
   }
   printf("\n");
}

void status_report(void) {
   pid_t pid; int wstat;
   while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
      print_child_status(pid, wstat);
   }
}
```

tinysh2.c

```
pid_t pid; int wstat;
while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
   print_child_status(pid, wstat);
}
```

- Die Funktion *waitpid* wartet auf einen gegebenen Kindprozess.
- Wenn (*pid_t*)−1 angegeben wird, dann werden alle Kinder akzeptiert.
- Mit der Option *WNOHANG* blockiert *waitpid* nicht und liefert 0 zurück, falls momentan noch kein Exit-Code für einer der Kind-Prozesse zur Verfügung steht.

tiny2sh.c

```
#ifndef COMMAND_H
#define COMMAND_H

#include <fcntl.h>
#include "strlist.h"

typedef struct fd_assignment {
   char* path;
   int oflags;
   mode_t mode;
} fd_assignment;

typedef struct command {
   char* cmdname;
   strlist argv;
   int background;
   /* for file descriptors 0 and 1 */
   fd_assignment assignments[2];
} command;

/* convert list of tokens into a command record */
int scan_command(strlist* tokens, command* cmd);

/*
 * open input and output files, if required, and
 * exec to the given command
 */
void exec_command(command* cmd);

#endif
```