Kapitel 2

Signale

2.1 Einführung

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

- um den normalen Ablauf eines Prozesses für einen wichtigen Hinweis zu unterbrechen,
- 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 ersetzen keine Interprozeßkommunikation, da sie fast keine Informationen mit sich führen. In Abhängigkeit von der jeweiligen Systemumgebung gibt es mehr oder weniger fest definierte Signale, die über natürliche Zahlen identifiziert werden.

Der ISO-Standard 9899-2011 für die Programmiersprache C definiert eine einfache und damit recht portable Schnittstelle für die Behandlung von Signalen und nennt auch die in **#include** <signal.h> festgelegten Makronamen für einige Signale: *SIGABRT*, *SIGFPE*, *SIGILL*, *SIGINT*, *SIGSEGV* und *SIGTERM*. 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.

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.

Typisch für letzteres ist die Terminalschnittstelle unter UNIX. Diese 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 durch ein spezielles Signal repräsentiert wird, das zeitlich 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. Und 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 dem Abbruch des laufenden Programms bei BREAK (die Shell ignoriert das Signal) und dem 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. Sinnvoll ist es natürlich, eine Anwendung mit einem Signalbehandler zu ergänzen, wenn dadurch Datenverluste bei einer Terminierung vermieden werden können.

2.2 Signalbehandler

Der folgende Programmtext demonstriert die Behandlung des Signals *SIGINT*. Als Signalbehandler operiert die Funktion *signal_handler*(). Signalbehandler erhalten als Argument eine ganze Zahl, worüber sie die Nummer des Signals erhalten, das sie gerade bearbeiten. Einen Rückgabewert gibt es nicht.

Programm 2.1: Behandlung eines *SIGINT*-Signals (*sigint.c*)

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  volatile sig_atomic_t signal_caught = 0;
6
7  void signal_handler(int signal) {
8      signal_caught = signal;
```

```
 9  }
10
11  int main() {
12      if (signal(SIGINT, signal_handler) == SIG_ERR) {
13          perror("unable to setup signal handler for SIGINT");
14          exit(1);
15      }
16      printf("Try to send a SIGINT signal!\n");
17      int counter = 0;
18      while (!signal_caught) {
19          for (int i = 0; i < counter; ++i);
20          ++counter;
21      }
22      printf("Got signal %d after %d steps!\n", signal_caught, counter);
23  }
```

Der Signalbehandler in diesem Beispiel setzt nur eine globale Variable auf den Wert des erhaltenen Signals (Zeile 8). Die Verwendung des Typs **volatile** *sig_atomic_t* wird später diskutiert. *main*() richtet auf Zeile 12 die Funktion *signal_handler*() als Signalbehandler ein. Dies geschieht mit der Funktion *signal*(), die eine Signalnummer (hier: *SIGINT*) und eine Reaktion (entweder *SIG_DFL* für die Prozeßterminierung, *SIG_IGN* für das Ignorieren oder einen Funktionszeiger) als Parameter erwartet. Im Erfolgsfalle liefert *signal*() die frühere Einstellung zurück, ansonsten *SIG_ERR*.

Weiter geht es auf Zeile 18 mit einer Schleife, die erst dann abbricht, wenn sich der Wert von *signal_caught* ändert. Dies kann in diesem Beispiel nur durch den Signalbehandler passieren. Wenn dies geschieht, wird auf Zeile 22 die Nummer des eingetroffenen Signals ausgegeben und das Programm beendet. So könnte ein Aufruf dieses Programms aussehen, wenn das Versenden des Signals *SIGINT* durch den Terminaltreiber durch die Eingabe von CTRL-c recht schnell geschieht:

```
doolin$ sigint
Try to send a SIGINT signal!
^CGot signal 2 after 2074 steps!
doolin$
```

Die 2 ist dabei die Nummer des Signals *SIGINT*:

```
doolin$ grep SIGINT /usr/include/sys/iso/signal_iso.h
#define SIGINT  2       /* interrupt (rubout) */
doolin$
```

Leider sind mit der Bearbeitung von Signalen große Probleme verbunden. Wenn ein optimierender Übersetzer den obigen Programmtext analysiert, könnten folgende Punkte auffallen:

- Die Schleife in den Zeilen 18 bis 21 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 also fatal, wenn darauf nur wegen der Existenz von asynchron aufgerufenen Signalbehandlern verzichtet werden würde.

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.

Daraus folgt, dass korrekte Signalbehandler in ihren Möglichkeiten stark eingeschränkt sind. 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.

Letzteres schließt insbesondere die Verwendung von Ein- und Ausgabe innerhalb eines Signalbehandlers aus. Der ISO-Standard 9899-1999 nennt nur *abort*(), *_Exit*()[1] und *signal*() als zulässige Bibliotheksfunktionen. Bei ISO 9899-2011 kommt noch *quick_exit*() hinzu, dass vor dem Aufruf von *_Exit*() noch Aufräummöglichkeiten für solche Notfälle vorsieht, die sich wiederum an die gleichen Spielregeln halten müssen.[2] 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“.[3]

Wozu der Datentyp *sig_atomic_t*? Dieser Datentyp ist ein ganzzahliger Typ, bei dem Lese- und Schreiboperationen garantiert atomar ablaufen. Es handelt sich also um einen Datentyp, den der verwendete Prozessor mit einer einzigen ununterbrechbaren Instruktion laden oder speichern kann. Bei allen anderen Datentypen ist es theoretisch nicht ausgeschlossen, dass mitten in einer Zuweisung ein asynchrones Signal eintreffen kann und der zugehörige Signalbehandler dann eine partiell modifizierte Variable vorfindet.

2.3 Wecksignale mit *alarm*

Zu den Signalen, die der POSIX-Standard definiert, gehört auch *SIGALRM*, das sich als Wecksignal verwenden lässt. Der Wecker wird mit *alarm*() unter Angabe einer relativen Weckzeit in Sekunden gestellt. Am Ende dieser Frist kommt es zum Eintreffen des Signals *SIGALRM*.

Programm 2.2: *read*-Operation mit Zeitlimit (*tread.c*)

```
1  /*
2   * Timed read operation. timed_read() works like read() but
3   * returns 0 if no input was received within the given number
4   * of seconds.
5   */
6
7  #include <signal.h>
8  #include <unistd.h>
9  #include "tread.h"
10
11 static volatile sig_atomic_t time_exceeded = 0;
```

[1] `_Exit()` unterlässt im Vergleich zu `exit()` sämtliche Aufräumarbeiten.

[2] Diese Funktion ist überwiegend noch nicht implementiert. Sie wird vom aktuellen POSIX-Standard auch noch nicht benannt.

[3] „MT“ steht hier für Multi-Threading, da dabei ähnliche Probleme auftreten, wenngleich in einem noch größeren Umfang.

```
12
13 static void alarm_handler(int signal) {
14     time_exceeded = 1;
15 }
16
17 int timed_read(int fd, void* buf, size_t nbytes, unsigned seconds) {
18     if (seconds == 0) return 0;
19     /*
20      * setup signal handler and alarm clock but
21      * remember the previous settings
22      */
23     void (*previous_handler)(int) = signal(SIGALRM, alarm_handler);
24     if (previous_handler == SIG_ERR) return -1;
25     time_exceeded = 0;
26     int remaining_seconds = alarm(seconds);
27     if (remaining_seconds > 0) {
28         if (remaining_seconds <= seconds) {
29             remaining_seconds = 1;
30         } else {
31             remaining_seconds -= seconds;
32         }
33     }
34
35     int bytes_read = read(fd, buf, nbytes);
36
37     /* restore previous settings */
38     if (!time_exceeded) alarm(0);
39     signal(SIGALRM, previous_handler);
40     if (remaining_seconds) alarm(remaining_seconds);
41
42     if (time_exceeded) return 0;
43     return bytes_read;
44 }
```

Der vorstehende Programmtext zeigt, wie *alarm*() verwendet werden kann, um eine Operation zeitlich zu befristen. Die Funktion *timed_read*() arbeitet genauso wie *read*(), wobei jedoch die Wartezeit auf die gegebene Zahl von Sekunden begrenzt wird. Wenn die Zeit verstreicht, ohne dass eine Eingabe erfolgte, wird 0 zurückgeliefert.

Was geschieht, wenn mehrere Wecksignale nebeneinander eingerichtet werden? Da *alarm*() nur die Verwaltung einer einzigen Weckzeit unterstützt, ist eine kooperative Vorgehensweise notwendig. Dies wird dadurch erleichtert, indem *alarm*() bei der Einrichtung einer neuen Weckzeit entweder 0 zurückgibt (vorher war keine Weckzeit aktiv) oder eine positive Zahl von Sekunden als Restzeit zum vorher eingestellten Wecksignal. Entsprechend wird auf Zeile 26 in der Variablen *remaining_seconds* die alternative Weckzeit notiert und später auf Zeile 40 findet eine Wiedereinsetzung statt, nachdem zuvor auf Zeile 31 die verbleibende Restzeit neu berechnet worden ist. Analog wird in Zeile 23 der frühere Signalbehandler für *SIGALRM* in der Variablen *previous_handler* gesichert, so dass er in Zeile 39 wieder restauriert werden kann.

Bei der Restaurierung ist es wichtig, dass kein Fenster offenbleibt, das zum Verlust einer Signalbehandlung führt oder den alten Signalbehandler auf das noch nicht eingetretene Signal für das von *tread*() eingerichtete Zeitlimit reagieren lässt. Wenn der alte Signalbehandler *SIG_DFL* war, also die Voreinstellung, dann würde das sogar unerwartet zur Terminierung unseres Prozesses führen. Deswegen wird in Zeile 38 der Wecker zuerst

deaktiviert, wenn er bislang sein Signal noch nicht von sich gegeben hat. Danach kann in Zeile 39 der alte Signalbehandler eingesetzt werden, ohne dass wir Gefahr laufen, dass er sofort verwendet wird. In Zeile 40 wird dann der Wecker neu aufgesetzt, falls er vor dem Aufruf von *tread*() eingeschaltet war.

Es bleibt hier noch anzumerken, dass es noch bessere Wege gibt, Leseoperationen mit Zeitbeschränkungen zu versehen. Es empfiehlt sich hier entweder die Verwendung der Systemaufrufe *poll*() oder *select*() oder der Einsatz asynchroner Lesetechniken. Dennoch ist die Verwendung von *alarm*() sinnvoll, da es genügend Operationen gibt, bei denen es keine Variante mit Zeitbeschränkung gibt. Alternativ zu *alarm*() gibt es auch noch *getitimer*() und *setitimer*() in einer weit implementierten POSIX-Erweiterung, die sowohl periodische *SIGALRM*-Signale als auch Zeitperioden von unter einer Sekunde unterstützen.

2.4 Das Versenden von Signalen

Der ISO-Standard für C sieht nur eine Funktion *raise*() vor, die es erlaubt, ein Signal an den eigenen Prozess zu versenden. Im POSIX-Standard kommt die Funktion *kill*() hinzu, die es erlaubt, ein Signal an einen anderen Prozess zu verschicken, sofern die dafür notwendigen Privilegien vorliegen. Das folgende Programm zeigt ein Beispiel, bei dem ein neu erzeugter Prozess ein Signal an den Erzeuger sendet.

Programm 2.3: Versenden eines Signals an den übergeordneten Prozess (*killparent.c*)

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/wait.h>
6
7  void sigterm_handler(int signo) {
8      const char msg[] = "Goodbye, cruel world!\n";
9      write(1, msg, sizeof msg - 1);
10     _Exit(1);
11 }
12
13 int main() {
14     if (signal(SIGTERM, sigterm_handler) == SIG_ERR) {
15         perror("signal"); exit(1);
16     }
17
18     pid_t child = fork();
19     if (child == 0) {
20         kill(getppid(), SIGTERM);
21         exit(0);
22     }
23     int wstat;
24     wait(&wstat);
25     exit(0);
26 }
```

Trotz ihres geringen Informationsgehalts dienen Signale gelegentlich zur Kommunikation. So gibt es eine weitverbreitete Konvention, dass bei langfristig laufenden Diensten *SIGHUP* das erneute Einlesen der Konfiguration veranlasst und *SIGTERM* eine geordnete Terminierung einleiten soll. Gelegentlich sind für Dienste auch Reaktionen für *SIGUSR1*

und *SIGUSR2* definiert. So wird der Apache-Webserver beispielsweise bei *SIGUSR1* veranlasst, bei nächster Gelegenheit auf sanfte Weise neu zu starten. Anders als bei *SIGHUP* kommt es dann nicht zur Unterbrechung aktueller Netzwerkverbindungen.

Der Systemaufruf *kill*() erfüllt aber auch noch einen weiteren Zweck. Bei einer Signalnummer von 0 wird nur die Zulässigkeit des Signalversendens überprüft. Das folgende Beispiel demonstriert, wie dies dazu verwendet werden kann, um die Existenz eines Prozesses zu überprüfen:

Programm 2.4: Verwendung von *kill* zur Überprüfung der Existenz eines Prozesses (*waitfor.c*)

```
1  #include <errno.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6
7  int main(int argc, char** argv) {
8      char* cmdname = *argv++; --argc;
9      if (argc != 1) {
10         fprintf(stderr, "Usage: %s pid\n", cmdname);
11         exit(1);
12     }
13
14     /* convert first argument to pid */
15     char* endptr = argv[0];
16     pid_t pid = strtol(argv[0], &endptr, 10);
17     if (endptr == argv[0]) {
18         fprintf(stderr, "%s: integer expected as argument\n",
19             cmdname);
20         exit(1);
21     }
22
23     while (kill(pid, 0) == 0) sleep(1);
24
25     if (errno == ESRCH) exit(0);
26     perror(cmdname); exit(1);
27 }
```

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), aber dies wäre 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.

Der folgende Programmtext demonstriert das Warten auf ein Signal anhand eines virtuellen Ballspiels zweier Prozesse. Das Programm beginnt in Zeile 37 mit der Erzeugung eines weiteren Prozesses, um einen Spielpartner zu haben. Beide Prozesse rufen anschließend *playwith*() auf, um das Spiel durchzuführen. Der neu erzeugte Prozess hat zuerst den virtuellen Ball und darf mit dem Spiel beginnen.

Programm 2.5: Virtuelles Ballspiel zweier Prozesse (*pingpong.c*)

```
1  #include <signal.h>
2  #include <stdio.h>
```

```
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  static volatile sig_atomic_t sigcount = 0;
7
8  void sighandler(int sig) {
9      ++sigcount;
10     if (signal(sig, sighandler) == SIG_ERR) _Exit(1);
11 }
12
13 static void playwith(pid_t partner, int start) {
14     if (signal(SIGUSR1, sighandler) == SIG_ERR) {
15         perror("signal␣SIGUSR1"); exit(1);
16     }
17     /* give our partner some time for preparation */
18     if (start) sleep(1);
19     /* start the ping pong game */
20     if (start) sigcount = 1;
21     for(int i = 0; i < 10; ++i) {
22         if (!sigcount) pause();
23         printf("[%d]␣send␣signal␣to␣%d\n",
24             (int) getpid(), (int) partner);
25         if (kill(partner, SIGUSR1) < 0) {
26             printf("[%d]␣%d␣is␣no␣longer␣alive\n",
27                 (int) getpid(), (int) partner);
28             return;
29         }
30         --sigcount;
31     }
32     printf("[%d]␣finishes␣playing\n", (int) getpid());
33 }
34
35 int main() {
36     pid_t parent = getpid();
37     pid_t child = fork();
38
39     if (child < 0) {
40         perror("fork"); exit(1);
41     }
42     if (child == 0) {
43         playwith(parent, 1);
44     } else {
45         playwith(child, 0);
46     }
47 }
```

Der Besitzer des virtuellen Balles sendet in Zeile 25 den Ball mit Hilfe des Signals *SIGUSR1* an den Partner und legt sich anschließend in Zeile 22 mittels *pause*() schlafen, um auf das Wiedereintreffen des Balls zu warten, das wiederum durch *SIGUSR1* signalisiert wird.

Die Variable *sigcount* in Zeile 6 repräsentiert die Zahl der Bälle, die sich im Augenblick im Besitz des Prozesses sind. Diese Zahl wird von dem Signalbehandler *sighandler*() in Zeile 9 hochgezählt, wenn ein *SIGUSR1*-Signal eintrifft und in Zeile 30 wieder herunter-

gezählt, nachdem das *SIGUSR1*-Signal an den Partner verschickt wurde.

Zu Beginn setzen beide Spieler in Zeile 14 *sighandler*() als Signalbehandler ein. Derjenige, der mit dem Spiel beginnt, wartet dann in Zeile 18 noch eine Sekunde, um zu vermeiden, dass ein Signal geschickt wird, bevor der Spielpartner die Gelegenheit hatte, seinen Signalbehandler aufzusetzen.

Zu beachten ist dabei, dass der Signalbehandler in Zeile 10 sich selbst wieder einsetzt, da der POSIX-Standard nicht festlegt, ob diese Einstellung nach Eintreffen eines Signals erhalten bleibt. *signal*() gehört mit zu den vom POSIX-Standard genannten Funktionen, die auch innerhalb eines Signalbehandlers aufgerufen werden dürfen.

2.5 Die Zustellung von Signalen

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:[4]

- 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 (*pending signal*).
- 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.

[4] Siehe „Signal Concepts“, im Web unter
`http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_04`

- 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.

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

Tabelle 2.1: Felder der **struct** *sigaction*

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 Tabelle 2.1 genannten Felder zum Einsatz.

Ein wesentlicher Unterschied zwischen *sigaction*() und *signal*() besteht bereits darin, dass per Voreinstellung das Signal, das eine Signalbehandlung auslöst, während der Bearbeitung automatisch blockiert wird. Ferner findet (solange nichts gegenteiliges in den Optionen angegeben wurde) keine implizite Veränderung des Signalbehandlers auf *SIG_DFL* nach der Signalbehandlung statt. Bei *signal*() ist dies ebenfalls möglich, jedoch nicht garantiert.

Das folgende Beispiel demonstriert den möglichen Verlust von Signalen trotz umfangreicher Vorsichtsmaßnahmen. Das Experiment besteht hier im Versenden von 1000 *SIGUSR1*-Signalen, die beim Empfänger nachgezählt werden. In den Zeilen 26 bis 30 setzt der neu erzeugte Prozess die Funktion *count_signals*() als Signalbehandler für *SIGUSR1* auf. Dabei wird hier *sigaction*() anstelle von *signal*() verwendet. Dies garantiert uns, dass während der Bearbeitung des Signals *SIGUSR1* weitere eintreffende *SIGUSR1*-Signale aufgeschoben und nicht sofort in verschachtelter Form bearbeitet werden.

Programm 2.6: Verlust von Signalen (*sigfire.c*)

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  static const int NOF_SIGNALS = 1000;
7  static volatile sig_atomic_t received_signals = 0;
8  static volatile sig_atomic_t terminated = 0;
```

```
9
10 static void count_signals(int sig) {
11     ++received_signals;
12 }
13
14 void termination_handler(int sig) {
15     terminated = 1;
16 }
17
18 int main() {
19     sighold(SIGUSR1); sighold(SIGTERM);
20
21     pid_t child = fork();
22     if (child < 0) {
23         perror("fork"); exit(1);
24     }
25     if (child == 0) {
26         struct sigaction action = {0};
27         action.sa_handler = count_signals;
28         if (sigaction(SIGUSR1, &action, 0) != 0) {
29             perror("sigaction"); exit(1);
30         }
31         action.sa_handler = termination_handler;
32         if (sigaction(SIGTERM, &action, 0) != 0) {
33             perror("sigaction"); exit(1);
34         }
35         sigrelse(SIGUSR1); sigrelse(SIGTERM);
36         while (!terminated) pause();
37         printf("[%d] received %d signals\n",
38             (int) getpid(), received_signals);
39         exit(0);
40     }
41
42     sigrelse(SIGUSR1); sigrelse(SIGTERM);
43     for (int i = 0; i < NOF_SIGNALS; ++i) {
44         kill(child, SIGUSR1);
45     }
46     printf("[%d] sent %d signals\n",
47         (int) getpid(), NOF_SIGNALS);
48     kill(child, SIGTERM); wait(0);
49 }
```

Auf diese Weise ist die Atomizität des Hochzählens der Variable *received_signals* garantiert. Zu bedenken ist dabei, dass der Datentyp *sig_atomic_t* selbst nur die Atomizität einer einzelnen Lese- oder Schreiboperation gewährleistet. Bei einem Inkrement liegt jedoch eine Lese- *und* eine Schreib-Operation vor. Zwischen diesen Operationen wäre eine Unterbrechung wegen einer Signalbehandlung denkbar.

In diesem Beispiel versendet der übergeordnete Prozess 1000 Signale und der neu erzeugte Prozess zählt die eingetroffenen Signale. Wann darf der übergeordnete Prozess davon ausgehen, dass der neu erzeugte Prozess seinen Signalbehandler fertig aufgesetzt hat, so dass er mit dem Zählen beginnen kann? Wenn der übergeordnete Prozess zu früh Signale versendet, während noch die voreingestellte Reaktion für *SIGUSR1* eingerichtet ist, würde dies nur zur vorzeitigen Terminierung des erzeugten Prozesses führen. Diese

Problematik lässt sich vermeiden, indem die Signale, für die der erzeugte Prozess Behandler aufsetzt, vor der Prozeßerzeugung blockiert werden. Das geht am einfachsten mit der Funktion *sighold*() (auf Zeile 19), die zum POSIX-Standard gehört. Zu jedem Prozess unterhält das Betriebssystem eine Menge blockierter Signale. Mit *sighold*() wird das angegebene Signal zu der Menge hinzugefügt. Entscheidend ist hier, dass die Menge der blockierten Signale an den neu erzeugten Prozess vererbt wird. So können nach dem *fork*() in aller Ruhe auf den Zeilen 26 bis 34 die Signalbehandler für *SIGUSR1* und *SIGTERM* aufgesetzt werden, bevor auf Zeile 35 diese Signale wieder mit Hilfe von *sigrelse*() (eine unglückliche Abkürzung von *signal release*) wieder aus der Menge der blockierten Signale entfernt werden. Auch der übergeordnete Prozess nimmt die beiden Signale wieder heraus auf der Zeile 42.

Wie wird das Experiment beendet? Da, wie das Experiment zeigen soll, Signale verloren gehen können, sollte der erzeugte Prozess nicht darauf warten, bis *NOF_SIGNALS* eingetroffen sind. Stattdessen wird *SIGTERM* vom übergeordneten Prozess an den erzeugten Prozess verwendet, um das Ende des Experiments zu signalisieren.

Hier sind einige Läufe des Experiments, die demonstrieren, wie sehr die Werte voneinander abweichen können:

```
doolin$ sigfire
[22073] sent 1000 signals
[22074] received 264 signals
doolin$ sigfire
[22075] sent 1000 signals
[22076] received 227 signals
doolin$ sigfire
[22077] sent 1000 signals
[22078] received 481 signals
doolin$ sigfire
[22079] sent 1000 signals
[22080] received 136 signals
doolin$
```

Wenn anstelle von nur *SIGUSR1* zwei Signale *SIGUSR1* und *SIGUSR2* abwechselnd verwendet werden, können höhere Werte erzielt werden, wobei der Erfolg keinesfalls garantiert ist:

```
doolin$ sigfire2
[22142] sent 1000 signals
[22143] received 495 signals
doolin$ sigfire2
[22144] sent 1000 signals
[22145] received 462 signals
doolin$ sigfire2
[22146] sent 1000 signals
[22147] received 468 signals
doolin$ sigfire2
[22151] sent 1000 signals
[22152] received 688 signals
doolin$
```

Dieses Experiment untermauert die Regel, dass einzelne Signale zuverlässig zugestellt werden, während es bei dem Mehrfachen Eintreffen des gleichen Signals zu Verlusten kommen kann. In der Praxis zeigen sich jedoch Signalverluste nur bei härteren Rahmenbedingungen, sei es durch ein explizites Dauerfeuer (wie in diesem Experiment) oder durch eine hohe Belastung der Maschine.

2.6 Signale als Indikatoren für terminierte Prozesse

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.

Programm 2.7: Demonstration von *SIGCHLD* (*sigchld.c*)

```
1  #include <signal.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/wait.h>
6  #include "processlist.h"
7
8  static processlist alive, dead;
9
10 void child_term_handler(int sig) {
11     pid_t pid; int wstat;
12     while ((pid = waitpid((pid_t)-1, &wstat, WNOHANG)) > 0) {
13         if (pl_move(&alive, &dead, pid)) {
14             pl_modify(&dead, pid, wstat);
15         }
16     }
17 }
18
19 int main() {
20     struct sigaction action = {0};
21     action.sa_handler = child_term_handler;
22     if (sigaction(SIGCHLD, &action, 0) != 0) {
23         perror("sigaction");
24     }
25     pl_alloc(&alive, 4); pl_alloc(&dead, 4);
26     sighold(SIGCHLD);
27     for (int i = 0; i < 10; ++i) {
28         fflush(0);
29         pid_t child = fork();
30         if (child < 0) {
31             perror("fork"); exit(1);
32         }
33         if (child == 0) {
34             srand(getpid()); sleep(rand() % 5);
35             exit((char) rand());
36         }
37         pl_add(&alive, child, 0);
38     }
39     sigrelse(SIGCHLD);
40     while (pl_length(&alive) > 0) {
41         if (pl_length(&dead) == 0) pause();
42         while (pl_length(&dead) > 0) {
43             int wstat;
44             sighold(SIGCHLD);
```

```
45              pid_t pid = pl_pick(&dead, &wstat);
46              sigrelse(SIGCHLD);
47              printf("[%d] %d\n", (int) pid, WEXITSTATUS(wstat));
48          }
49      }
50  }
```

Der vorstehende Programmtext demonstriert die Verwendung von *SIGCHLD*. In den Zeilen 27 bis 38 werden zehn Prozesse erzeugt, die nach einem zufällig bestimmten Zeitintervall mit einem zufälligen *exit*-Wert enden. Die Prozesse werden in der Zeile 37 in die Liste der noch lebenden Prozesse *alive* eingetragen. Der folgende Programmtext zeigt die Schnittstelle der Prozeßverwaltung:

Programm 2.8: Schnittstelle der Bibliothek für Listen von Prozessen (*processlist.h*)

```
1   #ifndef PROCESSLIST_H
2   #define PROCESSLIST_H
3
4   typedef struct process {
5       pid_t pid; int wstat;
6       struct process* next;
7   } process;
8   typedef struct processlist {
9       unsigned int size, length;
10      process** bucket; /* hash table */
11      unsigned int it_index;
12      process* it_entry;
13  } processlist;
14
15  // All functions with the exception of pl_length, pl_next,
16  // and pl_pick return 1 on success, 0 in case of failures.
17
18  /* allocate a hash table for processes with the given bucket size */
19  int pl_alloc(processlist* pl, unsigned int size);
20
21  /* add tuple (pid,wstat) to the process list, pid must be unique */
22  int pl_add(processlist* pl, pid_t pid, int wstat);
23
24  /* modify wstat for a given pid */
25  int pl_modify(processlist* pl, pid_t pid, int wstat);
26
27  /* delete tuple by pid */
28  int pl_remove(processlist* pl, pid_t pid);
29
30  /* move entry for pid to another list */
31  int pl_move(processlist* from, processlist* to, pid_t pid);
32
33  /* return number of elements */
34  unsigned int pl_length(processlist* pl);
35
36  /* lookup wstat by pid */
37  int pl_lookup(processlist* pl, pid_t pid, int* wstat);
38
39  /* start iterator */
```

```
40 int pl_start(processlist *pl);
41
42 /* fetch next pid from iterator; returns 0 on end */
43 pid_t pl_next(processlist *pl);
44
45 /* pick and remove one element out of the list */
46 pid_t pl_pick(processlist *pl, int* wstat);
47
48 /* free allocated memory */
49 int pl_free(processlist* pl);
50 #endif
```

Der Signalbehandler *child_term_handler* auf den Zeilen 10 bis 17 verzeichnet alle Eingänge des *SIGCHLD*-Signals, indem mit Hilfe von *waitpid*() der Status abgeholt wird und der terminierte Prozess von der Liste *alive* nach *dead* verschoben wird. Das Problem ist, dass bei einer zeitgleichen Terminierung mehrerer Prozesse es wiederum zu Verlusten des *SIGCHLD*-Signals kommen kann. Somit können wir uns nicht darauf verlassen, dass für jeden terminierten Prozess der Signalbehandler genau einmal aufgerufen wird. Deswegen empfiehlt es sich, den Status aller bereits terminierter Prozesse abzurufen. Dies wird mit *waitpid*() unter der Verwendung der Option *WNOHANG* erreicht. Diese verhindert ein Blockieren des Systemaufrufs *waitpid*() und somit kann *waitpid*() gefahrlos solange aufgerufen werden, bis *waitpid*() 0 oder einen negativen Wert zurückliefert.

Um die Prozesse und deren Status zu erfassen, sind umfangreichere Datenstrukturen erforderlich, die nicht mehr ohne weiteres mit dem Attribut **volatile** versehen werden können. Außerdem sollte es nicht dazu kommen, dass das Entfernen eines Eintrages aus der Liste der toten Prozesse in Zeile 45 von einer Signalbehandlung unterbrochen wird, die in Zeile 13 in genau die gleiche Liste, einen Eintrag hinzuzufügen versucht. Nur ein zuverlässiger gegenseitiger Ausschluß bewahrt uns hier vor Überraschungen.

Dies wird erreicht, indem gleich zu Beginn in Zeile 26 das Signal *SIGCHLD* blockiert wird. Das ist schon aus dem Grund wichtig, dass die Liste der noch lebenden Prozesse gefüllt werden kann, bevor diese Prozesse die Gelegenheit haben, sich allzu frühzeitig zu verabschieden. Nachdem alle zehn Prozesse erzeugt worden sind, wird in Zeile 39 die Blockierung wieder aufgehoben. Später, wenn die Liste *dead* modifiziert werden soll, wird die kritische Operation mit Hilfe von *sighold*() und *sigrelse*() wiederum geschützt.

Wenn auf gemeinsame Datenstrukturen von mehreren Seiten in asynchroner Form zugegriffen werden kann, dann sind die zugreifenden Programmbereiche sogenannte *kritische Regionen*. Nur durch den gegenseitigen Ausschluß wird verhindert, dass die betroffene Datenstruktur durch einen ungünstigen Unterbrechungszeitpunkt inkonsistent wird. Da *sigaction*() anstelle von *signal*() verwendet wird, ist bereits sichergestellt, dass *child_term_handler*() nicht mehrfach in verschachtelter Form aufgerufen wird. Somit müssen nur alle verbliebenen Programmbereiche, die auf die gleiche Datenstruktur zugreifen, in *sighold*() und *sigrelse*() geklammert werden.

2.7 Signalbehandlung in einer Shell

Die im vorherigen Kapitel vorgestellte einfache Shell kümmerte sich nicht um die Signalbehandlung. Dies kann zu überraschenden Effekten führen, wenn der Versuch unternommen wird, aufgerufene Prozesse beispielsweise mit *SIGINT* zu unterbrechen:

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

Hier wurde zunächst *cat* aufgerufen und nach der ersten eingegebenen Zeile CTRL-c eingegeben, welches bei den aktuellen Einstellungen zu einem SIGINT an alle Prozesse der aktuellen Sitzung führte. Zur aktuellen Sitzung gehört jedoch nicht nur das gerade laufende *cat*-Kommando, sondern natürlich auch *tinysh*. Da *tinysh* keinerlei Vorkehrungen traf, wurde es genauso wie *cat* einfach terminiert, weil dies die voreingestellte Reaktion ist. Wäre *tinysh* die Login-Shell gewesen, wäre damit die gesamte Sitzung beendet. Hier in diesem Beispiel wurde *SIGINT* offensichtlich von der normalen Shell ignoriert.

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.

Der folgende Programmtext zeigt das Hauptprogramm einer einfachen Shell mit Signalbehandlung. Das zeilenweise Einlesen wurde in die Funktion *getline*() ausgelagert, die Unterbrechungen berücksichtigt. Bei jeder sich bietenden Gelegenheit werden nichtblockierend mit *waitpid*() die Statusinformationen der inzwischen beendeten Prozesse abgeholt. Das Parsieren einer Kommandozeile wurde ausgelagert, wobei als weiteres unterstütztes Token noch "&" hinzugekommen ist, das für die Ausführung im Hintergrund steht.

Programm 2.9: Einfache Shell mit Signalbehandlung (*tinysh2.c*)

```
1  /*
2
3  =head1 NAME
4
5  tinysh2 −− a tiny shell with a minimal set of features
6
7  =head1 SYNOPSIS
8
9  B<tinysh2>
10
11 =head1 DESCRIPTION
12
13 B<tinysh2> reads command lines from its standard input
14 and executes them. tinysh2> '' is given as prompt.
15
16 Each command line consists of space−separated tokens. Special tokens
17 begin with <'', indicating the input file, >'', indicating the output
18 file, >>'', specifying an output file that is to be extended, and
19 &'', indicating a background execution. The first non−special token
20 specifies the command name which must be either an absolute path of an
```

```
21 executable file or locatable within the list of directories provided by
22 the environment variable B<PATH>.
23
24 =head1 DIAGNOSTICS
25
26 B<tinysh2> prints the exit code of a terminated program if it is non−zero.
27 An exit code of 255 is used by subprocesses if they are unable to start
28 the command for some reason.
29
30 =head1 BUGS
31
32 No pipelines, no builtins, no shell variables.
33
34 =head1 AUTHOR
35
36 Andreas Borchert
37
38 =cut
39
40 */
41
42 #include <errno.h>
43 #include <signal.h>
44 #include <stdio.h>
45 #include <stdlib.h>
46 #include <unistd.h>
47 #include <sys/wait.h>
48 #include "command.h"
49 #include "sareadline.h"
50 #include "strlist.h"
51 #include "tokenizer.h"
52
53 static sig_atomic_t interrupted = 0;
54
55 void interrupt_handler(int sig) {
56     interrupted = 1;
57 }
58
59 void print_child_status(pid_t pid, int wstat) {
60     printf("[%d] ", (int) pid);
61     if (WIFEXITED(wstat)) {
62         printf("exit %d", WEXITSTATUS(wstat));
63     } else if (WIFSIGNALED(wstat)) {
64         printf("terminated with signal %d", WTERMSIG(wstat));
65         if (WCOREDUMP(wstat)) printf(" (core dump)");
66     } else {
67         printf("???");
68     }
69     printf("\n");
70 }
71
72 void status_report(void) {
73     pid_t pid; int wstat;
```

```
74      while ((pid = waitpid((pid_t)−1, &wstat, WNOHANG)) > 0) {
75          print_child_status(pid, wstat);
76      }
77  }
78
79  int getline(stralloc* line) {
80      int first = 1;
81      interrupted = 0;
82      for(;;) {
83          if (interrupted) {
84              interrupted = 0;
85              printf("\n");
86              first = 1;
87          }
88          if (first) {
89              status_report();
90              printf("%% ");
91              first = 0;
92          }
93          errno = 0;
94          if (readline(stdin, line)) return 1;
95          if (errno != EINTR) return 0;
96      }
97  }
98
99  int main() {
100     struct sigaction action = {0};
101     action.sa_handler = interrupt_handler;
102     if (sigaction(SIGINT, &action, 0) != 0 ||
103             sigaction(SIGQUIT, &action, 0) != 0) {
104         perror("sigaction");
105     }
106
107     stralloc line = {0};
108     while (getline(&line)) {
109         strlist tokens = {0};
110         stralloc_0(&line); /* required by tokenizer() */
111         if (!tokenizer(&line, &tokens)) break;
112         if (tokens.len == 0) continue;
113         command cmd = {0};
114         if (!scan_command(&tokens, &cmd)) continue;
115
116         sighold(SIGINT); sighold(SIGQUIT);
117         pid_t child = fork();
118         if (child == −1) {
119             perror("fork"); continue;
120         }
121         if (child == 0) {
122             sigrelse(SIGINT); sigrelse(SIGQUIT);
123             if (cmd.background) {
124                 sigignore(SIGINT); sigignore(SIGQUIT);
125             }
126             exec_command(&cmd);
```

```
127             perror(cmd.cmdname);
128             exit(255);
129         }
130
131         if (cmd.background) {
132             printf("%d\n", (int)child);
133         } else {
134             int wstat;
135             pid_t pid = waitpid(child, &wstat, 0);
136             if (!WIFEXITED(wstat) || WEXITSTATUS(wstat)) {
137                 print_child_status(pid, wstat);
138             }
139         }
140         sigrelse(SIGINT); sigrelse(SIGQUIT);
141     }
142 }
```

Programm 2.10: Schnittstelle für die Behandlung von Kommandos (*command.h*)

```
1  #ifndef COMMAND_H
2  #define COMMAND_H
3
4  #include <fcntl.h>
5  #include "strlist.h"
6
7  typedef struct fd_assignment {
8      char* path;
9      int oflags;
10     mode_t mode;
11 } fd_assignment;
12
13 typedef struct command {
14     char* cmdname;
15     strlist argv;
16     int background;
17     /* for file descriptors 0 and 1 */
18     fd_assignment assignments[2];
19 } command;
20
21 /* convert list of tokens into a command record */
22 int scan_command(strlist* tokens, command* cmd);
23
24 /*
25  * open input and output files, if required, and
26  * exec to the given command
27  */
28 void exec_command(command* cmd);
29
30 #endif
```

Programm 2.11: Funktionen für die Behandlung von Kommandos (*command.c*)

```
1  #include "command.h"
```

```
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5
6  int scan_command(strlist* tokens, command* cmd) {
7      char* path; int oflags;
8      strlist_clear(&cmd->argv);
9      for (int fd = 0; fd < 2; ++fd) {
10         cmd->assignments[fd].path = 0;
11     }
12     cmd->background = 0;
13     for (int i = 0; i < tokens->len; ++i) {
14         switch (tokens->list[i][0]) {
15             case '<':
16                 cmd->assignments[0].path = &tokens->list[i][1];
17                 cmd->assignments[0].oflags = O_RDONLY;
18                 cmd->assignments[0].mode = 0;
19                 break;
20             case '>':
21                 path = &tokens->list[i][1];
22                 oflags = O_WRONLY | O_CREAT;
23                 if (*path == '>') {
24                     ++path; oflags |= O_APPEND;
25                 } else {
26                     oflags |= O_TRUNC;
27                 }
28                 cmd->assignments[1].path = path;
29                 cmd->assignments[1].oflags = oflags;
30                 cmd->assignments[1].mode = 0666;
31                 break;
32             case '&':
33                 cmd->background = 1;
34                 break;
35             default:
36                 strlist_push(&cmd->argv, tokens->list[i]);
37                 if (cmd->cmdname == 0) {
38                     cmd->cmdname = tokens->list[i];
39                 }
40         }
41     }
42     strlist_push0(&cmd->argv);
43     return cmd->cmdname != 0;
44 }
45
46 /* assign an opened file with the given flags and mode to fd */
47 void fassign(int fd, char* path, int oflags, mode_t mode) {
48     int newfd = open(path, oflags, mode);
49     if (newfd < 0) {
50         perror(path); exit(255);
51     }
52     if (dup2(newfd, fd) < 0) {
53         perror("dup2"); exit(255);
54     }
```

```
55      close(newfd);
56  }
57
58  void exec_command(command* cmd) {
59      for (int fd = 0; fd < 2; ++fd) {
60          if (cmd->assignments[fd].path != 0) {
61              fassign(fd, cmd->assignments[fd].path,
62                  cmd->assignments[fd].oflags,
63                  cmd->assignments[fd].mode);
64          }
65      }
66      execvp(cmd->cmdname, cmd->argv.list);
67  }
```

2.8 Überblick der Signale aus dem POSIX-Standard

Die Tabelle 2.2 liefert einen Überblick aller vom POSIX-Standard genannten Signale. Einzelne Implementierungen können noch weitere Signale unterstützen. Die Signale lassen sich dabei in mehrere Gruppen aufteilen:

- Programmierfehler: *SIGBUS*, *SIGFPE*, *SIGILL*, *SIGSEGV* und *SIGSYS*.
- Ressourcenverbrauch: *SIGVTALRM*, *SIGXCPU* und *SIGXFSZ*.
- Prozeßkontrolle: *SIGCONT*, *SIGKILL*, *SIGSTOP*, *SIGTERM* und *SIGTRAP*.
- Sitzungskontrolle: *SIGHUP*, *SIGINT*, *SIGQUIT* und *SIGTSTP*.
- Ereignis-Indikatoren: *SIGALRM*, *SIGCHLD*, *SIGPIPE*, *SIGPOLL*, *SIGPROF*, *SIGTTIN*, *SIGTTOU*, *SIGURG*, *SIGUSR1*, *SIGUSR2* und *SIGVTALRM*.

Signal	Voreinstellung	Beschreibung
SIGABRT	A	Process abort signal.
SIGALRM	T	Alarm clock.
SIGBUS	A	Access to an undefined portion of a memory object.
SIGCHLD	I	Child process terminated, stopped, or continued.
SIGCONT	C	Continue executing, if stopped.
SIGFPE	A	Erroneous arithmetic operation.
SIGHUP	T	Hangup.
SIGILL	A	Illegal instruction.
SIGINT	T	Terminal interrupt signal.
SIGKILL	T	Kill (cannot be caught or ignored).
SIGPIPE	T	Write on a pipe with no one to read it.
SIGQUIT	A	Terminal quit signal.
SIGSEGV	A	Invalid memory reference.
SIGSTOP	S	Stop executing (cannot be caught or ignored).
SIGTERM	T	Termination signal.
SIGTSTP	S	Terminal stop signal.
SIGTTIN	S	Background process attempting read.
SIGTTOU	S	Background process attempting write.
SIGUSR1	T	User-defined signal 1.
SIGUSR2	T	User-defined signal 2.
SIGPOLL	T	Pollable event.
SIGPROF	T	Profiling timer expired.
SIGSYS	A	Bad system call.
SIGTRAP	A	Trace/breakpoint trap.
SIGURG	I	High bandwidth data is available at a socket.
SIGVTALRM	T	Virtual timer expired.
SIGXCPU	A	CPU time limit exceeded.
SIGXFSZ	A	File size limit exceeded.

Voreinstellung	Beschreibung
T	Abbruch des Prozesses. Bei dem bei *wait*() zurückgelieferten Status ist *WIFSIGNALED* wahr und über *WTERMSIG* lässt sich das Signal ermitteln.
A	Analog zu T. Hinzu kommt möglicherweise noch die Erzeugung eines Speicherauszugs (in der Datei *core*). Letzteres lässt sich mit *WCOREDUMP* untersuchen.
I	Das Signal wird ignoriert.
S	Der Prozess wird gestoppt.
C	Der Prozess wird fortgesetzt.

Tabelle 2.2: Im POSIX-Standard genannte Signale (Quelle: www.opengroup.org)