

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 9899-2011 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 genügt, kommen bei der **struct** *sigaction*, die *sigaction()* verwendet, die in der folgenden Tabelle genannten Felder zum Einsatz:

Datentyp	Feldname	Beschreibung
void(*) (int)	<i>sa_handler</i>	Funktionszeiger (wie bisher)
void(*) (int , <i>siginfo_t*</i> , void*)	<i>sa_sigaction</i>	alternativer Zeiger auf einen Signalbehandler, der mehr Informationen zum Signal erhält
<i>sigset_t</i>	<i>sa_mask</i>	Menge von Signalen, die während der Signalbehandlung dieses Signals zu blockieren sind
int	<i>sa_flags</i>	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 = {
        .sa_sigaction = sighandler,
        .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.


```
int main() {
    sighold(SIGUSR1); sighold(SIGTERM);
    pid_t child = fork();
    if (child < 0) {
        perror("fork"); exit(1);
    }
    if (child == 0) {
        struct sigaction 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.

```
int main() {
    struct sigaction action = {
        .sa_handler = child_term_handler,
    };
    if (sigaction(SIGCHLD, &action, 0) != 0) {
        perror("sigaction"); exit(1);
    }
    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));
        }
    }
}
```

processlist.h

```
#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);
```

processlist.h

```
/* 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.

tiny2sh.c

```
static volatile sig_atomic_t interrupted = 0;

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

int main() {
    struct sigaction 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.

command.h

```
#ifndef COMMAND_H
#define COMMAND_H

#include <fcntl.h>
#include <afbib/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
```