- 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. dass der Empfänger unter keinen Umständen mehr zum Zuge kommt.

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);
   }
}
```

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 9899-2011 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 9899-2011 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 normalerweise *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.

Quote from The Rise of "Worse is Better":

> *Two famous people, one from MIT and another from Berkeley (but working on Unix) once met to discuss operating system issues. The person from MIT was knowledgeable about ITS (the MIT AI Lab operating system) and had been reading the Unix sources. He was interested in how Unix solved the PC loser-ing problem. The PC loser-ing problem occurs when a user program invokes a system routine to perform a lengthy operation that might have significant state, such as IO buffers. If an interrupt occurs during the operation, the state of the user program must be saved. Because the invocation of the system routine is usually a single instruction, the PC of the user program does not adequately capture the state of the process. The system routine must either back out or press forward. The right thing is to back out and restore the user program PC to the instruction that invoked the system routine so that resumption of the user program after the interrupt, for example, re-enters the system routine.*

It is called "PC loser-ing" because the PC is being coerced into "loser mode," where "loser" is the affectionate name for "user" at MIT.

The MIT guy did not see any code that handled this case and asked the New Jersey guy how the problem was handled. The New Jersey guy said that the Unix folks were aware of the problem, but the solution was for the system routine to always finish, but sometimes an error code would be returned that signaled that the system routine had failed to complete its action. A correct user program, then, had to check the error code to determine whether to simply try the system routine again. The MIT guy did not like this solution because it was not the right thing.

Die beiden Ansätze im Vergleich:

- *PC loser-ing*: Bei der Unterbrechung eines Systemaufrufs wird dieser wie bei einer abgebrochenen Transaktion zum Zeitpunkt des Aufrufs zurückgerollt, d.h. der Zustand vor dem Aufruf wird wiederhergestellt. Der Kontext des Benutzerprozesses wird auf die Instruktion gesetzt, die zum Systemaufruf führte.
- Ansatz von UNIX: Der Systemaufruf wird bei einer Unterbrechung immer beendet. Teilweise mit der Rückgabe eines Fehlercodes (*EINTR*) oder durch eine teilweise Umsetzung (etwa bei *read* oder *write*). Dies zwingt den Programmierer, ggf. bei *EINTR* den Systemaufruf manuell neu zu starten.

Wie behandelt UNIX dies heute:

- Nach wie vor ist der historische UNIX-Ansatz die Voreinstellung.
- Die *sigaction*-Schnittstelle ermöglicht optional (Flag *SA_RESTART*) den automatisierten Neustart eines Systemaufrufs, wo sonst *EINTR* zurückgeliefert wird.

Einerseits wird *EINTR* durchaus benötigt, da ein Abbruch durchaus ein erwünschtes Verhalten sein kann. Andererseits treffen mit *EINTR* unterbrochene Systemaufrufe möglicherweise auch Bibliotheken, die nicht dafür vorbereitet sind. Nicht immer lässt sich das leicht lösen. Die *stdio* bietet beispielsweise keine Vorkehrungen dazu.

- 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

Folgende Optionen werden im Rahmen des aktuellen POSIX-Standards unterstützt:

SA_NOCLDSTOP	*SIGCHLD* ist nicht zu generieren, wenn Kindprozesse stoppen oder Kindprozesse fortgesetzt werden
SA_ONSTACK	wenn ein alternativer Stack mit *sigaltstack* für Signalbehandlungen spezifiziert wurde, ist dieser zu verwenden
SA_RESETHAND	nach der Signalbehandlung wird die Einstellung für das Signal auf *SIG_DFL* zurückgesetzt
SA_RESTART	unterbrochene Systemaufrufe geben nicht *EINTR* zurück und werden stattdessen neu gestartet
SA_SIGINFO	dem Signalbehandler werden weitere Infos zur Verfügung gestellt
SA_NOCLDWAIT	bei *SIGCHLD* bedeutet dies, dass der Exit-Status der Kindprozesse sofort entsorgt wird
SA_NODEFER	während des Aufrufs des Signalbehandlers wird das eingetroffene Signal blockiert

restart.c

```
volatile sig_atomic_t gotit = 0;
void sighandler(int sig) {
   char msg[] = "I am still waiting for input...\n";
   if (!gotit) {
      write(1, msg, sizeof msg - 1);
      alarm(1);
   }
}

int main() {
   struct sigaction sigact = {
      .sa_handler = sighandler,
      .sa_flags = SA_RESTART,
   };
   if (sigaction(SIGALRM, &sigact, 0) < 0) {
      exit(1);
   }
   char msg[] = "Please type in something.\n";
   write(1, msg, sizeof msg - 1);
   alarm(1);
   char inbuf[32];
   ssize_t nbytes = read(0, inbuf, sizeof inbuf);
   gotit = 1;
   if (nbytes > 0) {
      char thanks[] = "Thanks!\n";
      write(1, thanks, sizeof thanks - 1);
   } else {
      char ooh[] = "Ooh :-(\n";
      write(1, ooh, sizeof ooh - 1);
   }
}
```

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.

Verlust von Signalen

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 = {
         .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.

sigchld.c

```
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));
      }
   }
}
```

```
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

tinysh2.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}; strlist tokens = {0}; command cmd = {0};
   while (getline(&line)) {
      stralloc_0(&line); /* required by tokenizer() */
      tokens.len = 0;
      if (!tokenizer(&line, &tokens)) break;
      if (tokens.len == 0) continue;
      if (!scan_command(&tokens, &cmd)) continue;

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

tinysh2.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);
```

tinysh2.c

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

tinysh2.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 if (WIFSTOPPED(wstat)) {
      printf("stopped with signal %d", WSTOPSIG(wstat));
   } else if (WIFCONTINUED(wstat)) {
      printf("continued");
   } 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 <afblib/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
```