Systemnahe Software II
SS 2008

Andreas F. Borchert
Universität Ulm

26. Mai 2008

Signalbehandlung in einer Shell

```
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*.

Signalbehandlung in einer Shell

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);
   }
}
```

Signalbehandlung in einer Shell

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);
```

Signalbehandlung in einer Shell

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;
   }
}
```

Signalbehandlung in einer Shell

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);
   }
}
```

Signalbehandlung in einer Shell

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.

Signalbehandlung in einer Shell

tiny2sh.c

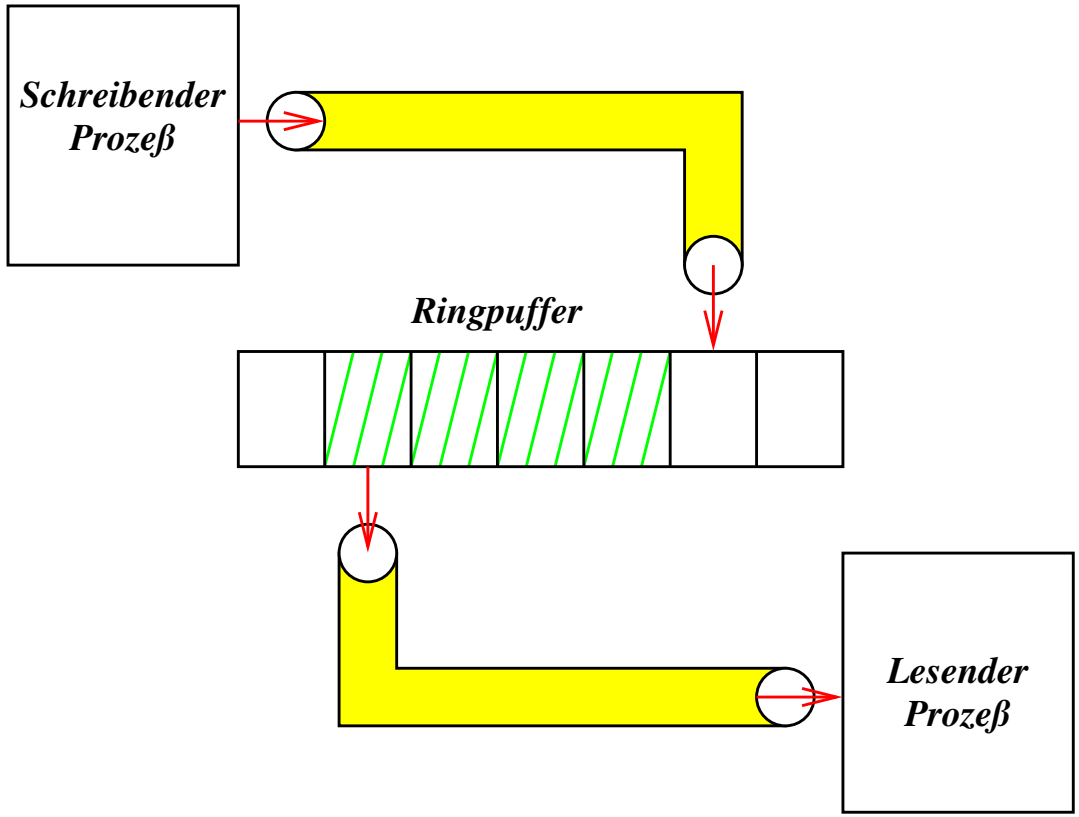
```
#ifndef COMMAND_H
#define COMMAND_H

#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
```

Pipelines

```
theseus$ ypcat passwd | cut -d: -f5 | cut -d:' ' -f1 |
> sort | uniq -c | sort -rn | head
     93 Michael
     92 Christian
     84 Florian
     83 Stefan
     81 Andreas
     69 Martin
     67 Thomas
     63 Matthias
     61 Daniel
     57 Markus
theseus$
```

- Welches sind die 10 häufigsten Vornamen unserer Benutzer?
- Dank Pipelines und dem Unix-Werkzeugkasten lässt sich diese Frage schnell beantworten.
- Die Notation und die zugehörige Art der Interprozesskommunikation wurde von einem der Mitautoren der ersten Unix-Shell Douglas McIlroy in den 70er-Jahren entwickelt und hat sehr zur Popularität von Unix beigetragen.

Aufbau einer Pipeline



- Pipelines sind unidirektionale Kommunikationskanäle. Die beiden Enden einer Pipeline werden über verschiedene Dateiverbindungen angesprochen.
- Sie werden innerhalb des Unix-Betriebssystems mit Hilfe eines festdimensionierten Ringpuffers implementiert.

Aufbau einer Pipeline

- Der festdimensionierte Ringpuffer hat typischerweise eine Größe von 4 Kilobyte.
- Wenn der Puffer vollständig gefüllt ist, wird ein Prozess, der ihn weiter zu füllen versucht, blockiert, bis wieder genügend Platz zur Verfügung steht.
- Wenn der Puffer leer ist, wird ein lesender Prozeß blockiert, bis der Puffer sich zumindest partiell füllt.
- Dies ist vergleichbar mit der Datenstruktur einer FIFO-Queue (*first in, first out*) mit explizit begrenzter Kapazität.
- Der POSIX-Standard unterstützt sowohl benannte Pipelines als auch solche, die mit Hilfe des Systemaufrufs *pipe*() erzeugt werden. Die benannten Pipelines sind aber kaum noch in Gebrauch, da die bidirektionalen UNIX-Domain-Sockets (mehr dazu später) normalerweise bevorzugt werden.

Konstruktion einer Pipeline

pipehello.c

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
enum {PIPE_READ = 0, PIPE_WRITE = 1};
int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      close(pipefds[PIPE_WRITE]);
      char buf[32];
      ssize_t nbytes;
      while ((nbytes = read(pipefds[PIPE_READ],
            buf, sizeof buf)) > 0) {
         if (write(1, buf, nbytes) < nbytes) exit(1);
      }
      exit(0);
   }
   close(pipefds[PIPE_READ]);
   const char message[] = "Hello!\n";
   write(pipefds[PIPE_WRITE], message, sizeof message - 1);
   close(pipefds[PIPE_WRITE]);
   wait(0);
}
```

Konstruktion einer Pipeline

pipehello.c

```
enum {PIPE_READ = 0, PIPE_WRITE = 1};
int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   /* ... */
}
```

- Mit dem Systemaufruf *pipe* wird eine Pipeline erzeugt.
- Zurückgegeben wird dabei ein Array mit zwei Dateiverbindungen, die auf das lesende (Index 0) und das schreibende (Index 1) Ende verweisen.
- Eine Interprozesskommunikation auf Basis von *pipe* lässt sich nur über *fork* aufbauen, indem das entsprechende andere Ende der Pipeline an einen neu erzeugten Prozess vererbt wird.
- Solche Pipelines können also nur zwischen verwandten Prozessen existieren, bei denen ein gemeinsamer Urahn diese mit *pipe* angelegt hat.

Schreibende Seite einer Pipeline

pipehello.c

```
pid_t child = fork();
if (child < 0) {
   perror("fork"); exit(1);
}
if (child == 0) {
   /* ... */
}
close(pipefds[PIPE_READ]);
const char message[] = "Hello!\n";
write(pipefds[PIPE_WRITE], message, sizeof message - 1);
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Der in eine Pipeline schreibende Prozess sollte das nicht genutzte Ende der Pipeline (hier das lesende) schließen. (Mehr dazu später.)
- Danach kann auf das schreibende Ende ganz normal mit *write* (oder auch darauf aufbauend der *stdio*) geschrieben werden.
- Sobald dies abgeschlossen ist, sollte das schreibende Ende geschlossen werden, damit ein Eingabe-Ende auf der anderen Seite der Pipeline erkannt werden kann.

Lesende Seite einer Pipeline

pipehello.c

```
if (child == 0) {
   close(pipefds[PIPE_WRITE]);
   char buf[32];
   ssize_t nbytes;
   while ((nbytes = read(pipefds[PIPE_READ],
         buf, sizeof buf)) > 0) {
      if (write(1, buf, nbytes) < nbytes) exit(1);
   }
   exit(0);
}
```

- Der von einer Pipeline lesende Prozess sollte das nicht genutzte Ende der Pipeline (hier das schreibende) schließen. (Mehr dazu später.)
- Danach kann auf das lesende Ende ganz normal mit *read* (oder auch darauf aufbauend der *stdio*) geschrieben werden.
- Die Schleife kopiert einfach alle Eingaben aus der Pipeline zur Dateiverbindung 1 (Standard-Ausgabe).
- Sobald alle schreibenden Enden geschlossen und der Ringpuffer geleert sind, wird ein Eingabe-Ende erkannt.

Gefahr von Hängern auf der lesenden Seite

- Nach *pipe* und *fork* haben zwei Prozesse jeweils beide Enden der Pipeline.
- Ein Eingabe-Ende auf der lesenden Seite wird genau dann (und nur dann!) erkannt, wenn **alle** schreibenden Enden geschlossen sind.
- Wenn also die lesende Seite es versäumt, die schreibende Seite zu schließen, wird sie kein Eingabe-Ende erkennen, wenn der andere Prozess seine schreibende Seite schließt.
- Stattdessen käme es zu einem endlosen Hänger.

Gefahr von Hängern auf der schreibenden Seite

- Genau dann (und nur dann!) wenn es kein Ende der Pipeline zum Lesen mehr gibt, führt das Schreiben auf das Ende zum Schreiben zur Zustellung des *SIGPIPE*-Signals bzw. dem Fehler *EPIPE*.
- Wenn die schreibende Seite es versäumt, ihr Ende zum Lesen zu schließen und der lesende Prozess aus irgendwelchen Gründen terminiert, ohne die Pipeline auslesen zu können, dann füllt sich zunächst der Ringpuffer und danach wird die schreibende Seite endlos blockiert.
- Entsprechend gäbe es wieder einen endlosen Hänger.
- Deswegen ist es von kritischer Bedeutung, dass die nicht benötigten Enden nach *fork* bei beiden Prozessen sofort geschlossen werden, um diese Probleme zu vermeiden.

Behandlung von *SIGPIPE*

sigpipe.c

```
int main() {
   int pipefds[2];
   if (pipe(pipefds) < 0) {
      perror("pipe"); exit(1);
   }
   pid_t child = fork();
   if (child < 0) {
      perror("fork"); exit(1);
   }
   if (child == 0) {
      close(pipefds[PIPE_WRITE]);
      char buf[32];
      ssize_t nbytes = read(pipefds[PIPE_READ],
            buf, sizeof buf);
      if (nbytes > 0) {
         if (write(1, buf, nbytes) < nbytes) exit(1);
      }
      exit(0);
   }
   close(pipefds[PIPE_READ]);
   struct sigaction action = {0}; action.sa_handler = sigpipe_handler;
   if (sigaction(SIGPIPE, &action, 0) < 0) {
      perror("sigaction"); exit(1);
   }
   while (!sigpipe_received) {
      const char message[] = "Hello!\n";
      write(pipefds[PIPE_WRITE], message, sizeof message - 1);
   }
   close(pipefds[PIPE_WRITE]); wait(0);
}
```

Behandlung von *SIGPIPE*

sigpipe.c

```
volatile sig_atomic_t sigpipe_received = 0;

void sigpipe_handler(int sig) {
   sigpipe_received = 1;
}
```

- Der Signalbehandler für *SIGPIPE* setzt hier nur eine globale Variable, so dass entsprechend getestet werden kann.
- Alternativ könnte als Signalbehandler auch *SIG_IGN* eingetragen werden. Das würde keine Funktion benötigt werden und es müsste dann explizit jede *write*-Operation überprüft werden. Wenn niemand mehr das andere Ende lesen kann, würde *errno* auf *EPIPE* gesetzt werden.

Behandlung von *SIGPIPE*

sigpipe.c

```
if (child == 0) {
   close(pipefds[PIPE_WRITE]);
   char buf[32];
   ssize_t nbytes = read(pipefds[PIPE_READ],
         buf, sizeof buf);
   if (nbytes > 0) {
      if (write(1, buf, nbytes) < nbytes) exit(1);
   }
   exit(0);
}
```

- Anders als zuvor ruft der neu erzeugte Prozess *read* nur ein einziges Mal auf und endet dann.
- Sobald sich dieser Prozess mit *exit* verabschiedet, bleibt kein lesendes Ende der Pipeline mehr offen, so dass damit dann die schreibende Seite das Signal *SIGPIPE* erhält, sobald sie in die Pipeline weiterhin schreibt.

Behandlung von *SIGPIPE*

sigpipe.c

```
close(pipefds[PIPE_READ]);
struct sigaction action = {0};
action.sa_handler = sigpipe_handler;
if (sigaction(SIGPIPE, &action, 0) < 0) {
   perror("sigaction"); exit(1);
}
while (!sigpipe_received) {
   const char message[] = "Hello!\n";
   write(pipefds[PIPE_WRITE], message, sizeof message - 1);
}
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Beim übergeordneten Prozess wird zunächst der Signalbehandler für *SIGPIPE* eingesetzt.
- Danach wird solange in die Pipeline geschrieben, bis das Signal endlich eintrifft.

Behandlung von *SIGPIPE*

sigpipe2.c

```
close(pipefds[PIPE_READ]);
sigignore(SIGPIPE);
ssize_t nbytes;
do {
   const char message[] = "Hello!\n";
   nbytes = write(pipefds[PIPE_WRITE],
      message, sizeof message - 1);
} while (nbytes > 0);
if (errno != EPIPE) perror("write");
close(pipefds[PIPE_WRITE]);
wait(0);
```

- Alternativ könnte *SIGPIPE* ignoriert werden.
- Dann ist die Überprüfung der *write*-Operationen zwingend notwendig.