Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2014

Interprozesskommunikation 1: Pipes

exec-Funktionen

popen

Filedeskriptoren

Linux benutzt numerische Filedeskriptoren, normalerweise

I/O-Kanal	Deskriptor
stdin	0
stdout	1
stderr	2

Ein Filedeskriptor kann verbunden sein mit

- einem Terminal
- einer Datei
- einer Pipe

Beispiel für eine Pipe

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main() {
    int pipe_fd[2];
    if (pipe(pipe_fd) == -1) {
        printf("Fehler %d: %s\n", errno, strerror(errno));
        return(1);
    }

    char zeile[255];
    write(pipe_fd[1],"Hallo",strlen("Hallo"));
    int size = read(pipe_fd[0],zeile,254);
    zeile[size]='\0';
    printf("Gelesen: %s\n", zeile);
    return(0);
}
```

Verwenden einer Pipe

Eine Pipe wird mit der C-Funktion

```
#include <unistd.h>

int pipe(int pipefd[2]);
```

geöffnet, wobei das erste Element von `pipefd` (Index 0) die lesende und das zweite Element (Index 1) die schreibende Seite der Pipe kennzeichnet.
Rückgabewert ist 0 bei Erfolg, sonst -1.

Eselsbrücke: Index 0 entspricht Filedeskriptor 0 (stdin), also lesend, Index 1 entspricht Filedeskriptor 1 (stdout), also schreibend.

Interprozesskommunikation

Das vorliegende Beispiel benutzte eine Pipe innerhalb eines Prozesses.

Spannender wird es, wenn eine Pipe zur Interprozesskommunikation,

Inter process communication – IPC

verwendet wird.

Merkregel: Bei einem Fork werden die existierenden Filedeskriptoren dupliziert, d. h. eine vorher angelegte Pipe steht in *beiden* Prozessen zur Verfügung.

Beispiel

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main() {
    int pipe_fd[2];
    if (pipe(pipe_fd) == -1) {
        printf("Fehler %d: %s\n", errno, strerror(errno));
        return(1);
    }

    int child = fork();
```

Beispiel (Forts.)

```
    if (child>0) {
        close(pipe_fd[0]);
        char zeile[255];
        printf("Zeile eingeben: ");
        gets(zeile);
        write(pipe_fd[1],zeile,strlen(zeile));
        wait();
        return(0);
    } else {
        close(pipe_fd[1]);
        char zeile[255];
        int size = read(pipe_fd[0],zeile,255);
        zeile[size]='\0';
        printf("Kindprozess hat gelesen: %s\n", zeile);
        return(0);
    }
}
```

Sind Daten da?

Im vorigen Beispiel blockiert die `read`-Funktion den Kindprozess so lange, bis Daten zur Verfügung stehen (oder EOF vorliegt, z. B. weil der Elternprozess die schreibende Seite schließt).

Um zu prüfen, ob Daten anstehen, kann die Funktion `select` verwendet werden.
Diese Funktion prüft für eine Menge von Filedeskriptoren, ob Daten zum Lesen anliegen (ein `read` also direkt ausgeführt wird), Daten direkt geschrieben werden können oder eine Ausnahme vorliegt).

Die Funktion select

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

mit

- **nfds** Höchste Nummer eines zu beobachtenden FD *plus 1*.
- **readfds, writefd, exceptfds** Menge der FD, die auf Lesen, Schreiben bzw. Ausnahme getestet werden.
- **timeout** Maximale Wartezeit auf Änderung in einem der beobachteten FD.

Rückgabewert ist die Zahl der in den FD-Sets enthaltenen FD.

Struktur timeval und Makros zu FD-Sets

Die Struktur `timeval` (aus `sys/time.h`) definiert einen Zeitraum mit Sekunden- und Mikrosekundenanteil:

```
struct timeval {
    long    tv_sec;         /* seconds */
    long    tv_usec;        /* microseconds */
};
```

Folgende Makros fügen einem FD-Set einen Filedeskriptor hinzu oder entfernen diesen, löschen alle FD aus einem FD-Set oder testen darauf, ob ein FD in einem FD-Set enthalten ist:

```
void FD_SET(int fd, fd_set *set);
void FD_CLR(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
int FD_ISSET(int fd, fd_set *set);
```

Beispiel

```
#include <stdio.h>
#include <unistd.h>
#include <sys/select.h>
#include <string.h>
#define BUF_LEN 255

int main() {
    int pipe_fd[2];
    pipe(pipe_fd);
    int child = fork();
    if (child>0) {
        close(pipe_fd[0]);
        sleep(5);
        write(pipe_fd[1],"Hallo",strlen("Hallo"));
        sleep(5);
        write(pipe_fd[1],"Otto",strlen("Otto"));
        sleep(3);
        write(pipe_fd[1],"Ende",strlen("Ende"));
        wait();
        return(0);
```

Beispiel (Forts.)

```
    } else {
        close(pipe_fd[1]);
        char str[BUF_LEN+1];
        int size;
        fd_set rd_fds;
        struct timeval timeout;
        do {
            do {
                FD_ZERO(&rd_fds);
                FD_SET(pipe_fd[0],&rd_fds);
                timeout.tv_sec  = 1;
                timeout.tv_usec = 0;
                printf("Warte auf Daten ...\n");
                select(pipe_fd[0]+1, &rd_fds, NULL, NULL, &timeout);
            } while (!FD_ISSET(pipe_fd[0], &rd_fds));
            size = read(pipe_fd[0],str,BUF_LEN);
            str[size] = '\0';
            printf("Gelesen: +++%s+++\n", str);
        } while (strcmp(str,"Ende"));
        return(0);
    }
}
```

Duplizieren von Filedeskriptoren

Filedeskriptoren können auch dupliziert werden:

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

In der ersten Variante wird ein Filedeskriptor auf den freien FD mit der niedrigsten Nummer dupliziert.
In der zweiten Variante wird der als erstes Argument angegebene FD auf den als zweites Argument angegebenen FD dupliziert. Letzterer wird vorher geschlossen.
Rückgabewert ist bei Erfolg die Nummer des duplizierten FD, sonst -1.

Beispiel

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

int main() {
    int pipe_fd[2];
    pipe(pipe_fd);

    int child = fork();
```

Beispiel (Forts.)

```
    if (child>0) {
        close(pipe_fd[0]);
        char zeile[255];
        printf("Zeile eingeben: ");
        gets(zeile);
        close(STDOUT_FILENO);
        dup(pipe_fd[1]);
        printf("%s\n",zeile);
        wait();
        return(0);
    } else {
        close(pipe_fd[1]);
        dup2(pipe_fd[0],STDIN_FILENO);
        char zeile[255];
        gets(zeile);
        printf("Kindprozess hat gelesen: %s\n", zeile);
        return(0);
    }
}
```

Analyse

Im vorigen wird das Duplizieren von Filedeskriptoren benutzt, um `stdout` in die Pipe bzw. die Pipe nach `stdin` umzuleiten.

Im Elternprozess wird zunächst der Filedeskriptor mit der Nummer `STDOUT_FILENO` geschlossen. Da dies dann die niedrigste freie Nummer für einen FD ist, wird somit die schreibende Seite der Pipe mit `dup` nach `stdout` dupliziert, alle Ausgaben nach `stdout` landen ab nun also in der Pipe.

Im Kindprozess wird alternativ `dup2` verwendet, um die lesende Seite der Pipe nach `stdin` zu duplizieren. Beim Lesen von `stdin` stammen die Daten von nun an also aus der Pipe.

Die Funktion execve

Mit folgender Funktion wird der aktuelle Prozess komplett (also Programmcode und Daten) durch einen neuen ersetzt, wobei jedoch die alte PID und offene Filedeskriptoren erhalten bleiben!

```
#include <unistd.h>

int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

Hierbei sind `argv` und `envp` String-Arrays (zweiterer mit Strings der Form „`var=value`“) *mit NULL als letztem Element*.

Achtung, `argv[0]` wird analog zu den Parametern von `main(...)` als Programmname interpretiert.
`execve` kehrt niemals zurück, es sei denn bei fehlerhaften Aufruf, dann mit Rückgabewert -1.

Beispiel

```
#include <unistd.h>
#include <errno.h>

int main() {
    char* args[] = { "bash", "-c", "set", NULL };
    char* env[] = { "FOO=bar", NULL };
    if (execve("/bin/bash", args, env)==-1) {
        printf("Fehler %d: %s\n", errno, strerror(errno);
    }
}
```

Varianten

```
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
           char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

Bei den Varianten mit „`l`“ werden die Argumente nicht als Array `args` sondern als Liste übergeben (NULL nicht vergessen), z. B.

```
execle("/bin/bash", "bash", "-c", "set", NULL, env)
```

während die Varianten mit „`p`“ bei Dateinamen ohne „`/`“ im Namen im aktuellen `PATH` nach einem entsprechenden Programm suchen, z. B.

```
execlp("ls", "ls", "-l", "/", NULL)
```

Beispiel für Kommunikation mit externem Programm

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#define BUF_LEN 255

int main() {
    int p_fd[2];
    if (pipe(p_fd)<0) return(1);

    int child = fork();
    if (child < 0) return(2);
```

Beispiel (Forts.)

```
    if (child>0) {
        close(p_fd[0]);
        FILE* p = fdopen(p_fd[1],"w");
        printf("Befehl eingeben: ");
        char zeile[BUF_LEN+1];
        gets(zeile);
        fprintf(p, "%s\n", zeile);
        fclose(p);
        wait();
        return(0);
    } else {
        close(p_fd[1]);
        dup2(p_fd[0], STDIN_FILENO);
        if (execlp("bash", "bash", NULL)==-1) {
            printf("Fehler %d: %s\n", errno, strerror(errno));
        }
    }
}
```

Die Funktion popen

Im letzten Beispiel wurden `fork`, Pipes und die `exec`-Funktionen miteinander verknüpft:

- Per `fork` wurde ein Kindprozess erzeugt,
- dieser benutzte eine vorher angelegt Pipe, wobei `stdin` auf die lesende Seite der Pipe umgebogen wurde.
- Dann wurde der Kindprozess durch eine `bash` ersetzt, die über die Pipe aus dem Elternprozess Befehle bekam.

Letzlich geht diese Aktion allerdings einfacher, nämlich mit

```
#include <stdio.h>

FILE* popen(const char* cmd, const char* type)
```

wobei `type` entweder „`r`“ oder „`w`“ ist.

Die Funktion popen (Forts.)

Die Pipe an das ausgeführte Programm wird mit
`pclose(FILE *stream)`
geschlossen. Hierbei wird (mit `waitpid`) auf das Ende des aufgerufenen Programms gewartet, es wird also erwartet, dass dieses bei `EOF` auf `stdin` bzw. beim Schließen von `stdout` endet.

Die Kommunikation mit dem externen Programm ist unidirektional, also können *entweder* Eingaben an das Programm geschickt *oder* dessen Ausgabe gelesen werden. Es gibt in der C-Bibliothek kein `popen2` mit bidirektionaler Kommunikation, dieses muss entweder selbst implementiert oder von dritter Seite bereit gestellt werden.

Beispiel

Im folgenden Beispiel soll aus einem Programm heraus eine E-Mail verschickt werden (wie es z. B. Backup-Programme bei Fehlern bei der Datensicherung tun). Hierfür wird das Programm `sendmail` aufgerufen, das unter Linux Mails entgegennimmt und transportiert.

```
#include <stdio.h>

int main() {
    FILE* p = popen("/usr/lib/sendmail -t -oem -oi","w");
    fprintf(p,"From: klaus\n");
    fprintf(p,"To: karl\n");
    fprintf(p,"Subject: Test\n");
    fprintf(p,"\n");
    fprintf(p,"Hallo\n");
    pclose(p);
    return(0);
}
```