Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2014

Grundlagen der Prozessverwaltung

Jeder Prozess unter Linux hat eine eindeutige Nummer, die Prozess-Id (PID).
Nebenbei weiß jeder Prozess noch die Prozess-Id des Elternprozesses, die PPID.

Informationen zu den Prozessen des Systems oder eines Users können mit dem Programm `ps` angezeigt werden (oder Programmen wie `pstree`), unter C stehen folgende Funktionen zur Verfügung:

```
#include <unistd.h>
pid_t getpid();
pid_t getppid ();
```

Beispiel

Das folgende C-Programm zeigt an, unter welcher PID es läuft, und wie die PID des Elternprozesses, also die PPID lautet:

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

int main() {
    printf("Programm PID %d, PPID %d\n",
           getpid(),
           getppid());
    return(0);
}
```


Fork eines Prozesses

Beim Fork eines Prozesses wird eine exakte Kopie des existierenden Prozesses als Kindprozess angelegt. Hierfür wird die Funktion `fork()` verwendet.
Der Rückgabewert von `fork()` vom Typ `pid_t` (=int) bedeutet:

- < 0 Fork ging schief,
- 0 Rückgabewert im *Kindprozess*,
- > 0 Rückgabewert im *Elternprozess*, der Wert enthält die PID des per fork erzeugten Kindprozesses.

Beispiel

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

int main() {
    int child = fork();
    if (child>0) {
        printf("Eltern PID: %d; PPID: %d\n",
               getpid(), getppid());
        printf("Kind erzeugt mit PID %d\n", child);
        sleep(10);
        printf("Elternprozess wird beendet\n");
```

Beispiel (Forts.)

```
    } else {
        sleep(2);

        int i;
        for (i=0; i<20; i++) {
            printf("Kind PID: %d; PPID: %d\n",
                   getpid(), getppid());
            sleep(1);
        }
    }
}
```


fork und Daten

Nach dem `fork` teilen sich Eltern- und Kindprozess zwar den Programmbereich (wobei die beiden Prozesse eigene Befehlszähler haben), aber *nicht* den Datenbereich.

```
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

int main() {
    int i = 1;
    int j = 5;
    int result = fork();
    switch (result) {
        case -1:
            printf("Fork failed\n");
            return(EXIT_FAILURE);
            break;
```


fork und Daten (Forts.)

```
        case 0:
            printf("Address of i in child: %x\n", &i);
            while (j--) {
                printf("child i: %d\n", i++);
                sleep(1);
            }
            break;
        default:
            printf("Address of i in parent: %x\n", &i);
            srand(time(0));
            i = random();
            while (j--) {
                printf("parent i: %d\n", i++);
                sleep(1);
            }
            break;
    }
    return(EXIT_SUCCESS);
}
```

Ausgabe

Der vorstehende C-Code erzeugt folgende Ausgabe:

```
Address of i in parent: bf815df4
parent i: 1251837046
Address of i in child: bf815df4
child i: 1
parent i: 1251837047
child i: 2
parent i: 1251837048
child i: 3
parent i: 1251837049
child i: 4
parent i: 1251837050
child i: 5
```

Obwohl beide Prozesse die Variable *i* kennen (die sich bzgl. des Datensegments der Prozesse sogar an der selben Adresse befindet), arbeitet der Kindprozess tatsächlich mit einer Kopie.

Ein `fork` erzeugt also keinen gemeinsam benutzten Speicherbereich!

Ende des Elternprozesses

Was passiert nun, wenn sich der Elternprozess beendet?

```
int child = fork();
if (child>0) {
    printf("Eltern PID: %d; PPID: %d\n",
           getpid(), getppid());
    printf("Kind erzeugt mit PID %d\n", child);
    sleep(10);
    printf("Elternprozess wird beendet\n");
    return(0);
} else {
    sleep(2);

    int i;
    for (i=0; i<20; i++) {
        printf("Kind PID: %d; PPID: %d\n",
               getpid(), getppid());
        sleep(1);
    }
}
```


Analyse

- Der Elternprozess beendet sich nach einer Wartezeit von 10 Sekunden.
- Der Kindprozess gibt im Sekundentakt seine PID und die PPID, also die PID des Elternprozesses aus.
- Solange der Elternprozess läuft, hat der Kindprozess die PID des Elternprozesses als PPID.
- Nachdem sich der Elternprozess beendet hat (ab jetzt sind wieder Eingaben in der Bash möglich), bekommt der Kindprozess eine neue PPID: 1.
- Neuer Elternprozess des per `fork` erzeugten Prozesses ist also der `init`-Prozess.
- Trotz Ende des Elternprozesses läuft der Kindprozess also weiter (was u. a. in den letzten Vorlesungen genutzt wurde, um Dämonen im Hintergrund laufen zu lassen).

Beenden des Kindprozesses

Nun sorgen wir dafür, dass sich der Kindprozess bei laufendem Elternprozess beendet.

```
int child = fork();
if (child>0) {
    printf("Eltern PID: %d; PPID: %d\n", getpid(), getppid());
    printf("Kind erzeugt mit PID %d\n", child);
    sleep(20);
    printf("Elternprozess wird beendet\n");
    return(0);
} else {
    sleep(2);
    printf("Kind PID: %d; PPID: %d\n", getpid(), getppid());
    sleep(2);
    printf("Kindprozess wird beendet\n");
    return(0);
}
```


Analyse

- Nun beendet sich der Kindprozess nach etwa 4 Sekunden.
- Der Elternprozess läuft hier aber noch weiter, bis er sich nach dem Ende der Wartezeit von 20 Sekunden beendet.
- Interessant ist hier ein Blick in die Prozessliste nach Ende des Kindprozesses:

```
1824 ?        Sl     0:04 gnome-terminal
1999 pts/1    Ss     0:00  \_ bash
2143 pts/1    S+     0:00      \_ ./fork
2144 pts/1    Z+     0:00          \_ [fork] <defunct>
```

- Der Kindprozess ist zwar beendet, bleibt aber als so genannter *Zombie* in der Prozessliste stehen.
- Der Elternprozess ist dafür verantwortlich, den Kindprozess nach dessen Ende aus der Prozessliste auszutragen!

Warten auf das Ende des Kindprozesses

Da der Elternprozess für den Kindprozess mitverantwortlich ist, sollte er das Ende des Kindes abfangen.
Hierfür kennt C zwei Funktionen:

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);

pid_t waitpid(pid_t pid, int *status, int options);
```

Die Funktion wait(int*)

Mit `wait(int *status)` wird gewartet, bis sich der nächste Kindprozess des aktuellen Prozesses beendet. Der aktuelle Prozess wird also durch den Aufruf blockiert!

Rückgabewert ist die PID des beendeten Kindprozesses. Weiterhin werden Informationen in die Variable *status* geschrieben. Folgenden Makros dienen der Analyse:

WIFEXITED(status) ist `true`, falls der Prozess normal beendet wurde (mit exit oder return innerhalb von main), und

WEXITSTATUS(status) gibt den Return-Status des Kindprozesses an.

WIFSIGNALED(status) ist `true`, falls der Prozess durch ein Signal (z. B. per kill) beendet wurde, und

WTERMSIG(status) gibt die Nummer des entsprechenden Signals aus.

Beispiel

```
int child = fork();
if (child>0) {
    printf("Eltern PID: %d; PPID: %d\n", getpid(), getppid());
    printf("Kind erzeugt mit PID %d\n", child);
    int status;
    int pid=wait(&status);
    printf("Kindprozess %d wurde beendet ...\n",pid);
    if (WIFEXITED(status)) {
        printf("mit Status %d\n",WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("mit Signal %d\n",WTERMSIG(status));
    }
    return(0);
} else {
    sleep(2);
    printf("Kind PID: %d; PPID: %d\n", getpid(), getppid());
    sleep(2);
    return(1);
}
```


Die Funktion waitpid(int,*int,int)

`wait(int *status)` ist ein Spezialfall der Funktion
`waitpid(int pid, int *status, int options)`
Mit dem Parameter *pid* kann auf das Ende eines ganz bestimmten Kindprozesses gewartet werden, während der Wert -1 einem beliebigen Kindprozess entspricht. Mit den Optionen kann angegeben werden, ob der Aufruf blockiert. Mit der Konstanten `WNOHANG` als Option kehrt die Funktion direkt zurück (und liefert den Rückgabewert 0, wenn kein Kindprozess beendet wurde).

`wait(&status)` entspricht also `waitpid(-1,&status,0)`. Im folgenden Beispiel wird gezeigt, wie `WNOHANG` benutzt werden kann.

Beispiel

```
int child = fork();
if (child>0) {
    int status;
    int pid;
    while ( (pid=waitpid(child,&status,WNOHANG)) == 0) {
        printf("Warte auf Ende des Kindprozesses ...\n");
        sleep(1);
    }
    printf("Kindprozess %d wurde beendet ...\n",pid);
    if (WIFEXITED(status)) {
        printf("mit Status %d\n",WEXITSTATUS(status));
    } else if (WIFSIGNALED(status)) {
        printf("mit Signal %d\n",WTERMSIG(status));
    }
    return(0);
} else {
    sleep(5);
    return(2);
}
```


Ändern der User-ID

Auf Dateisystemebene wurde schon gezeigt, wie mit dem Superuser-Bit die effektive User-ID (EUID) eines Programmes auf den Dateibesitzer gesetzt werden kann.

Auch innerhalb von Programmen kann der ausführende User beeinflusst werden, aus Sicherheitsgründen aber im Wesentlichen nur für root.

Die Funktion `setuid(int uid)`

- ausgeführt als root (EUID 0) setzt reale und effektive User-ID neu, hierbei sind beliebige UID zulässig,
- ausgeführt mit EUID $\neq 0$ setzt die *effektive* User-ID, aber nur dann, wenn der neue Wert entweder der realen oder effektiven UID des Prozesses entspricht.

Beispiel

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

int main() {
    printf("UID: %d, EUID: %d\n", getuid(), geteuid());
    int newuid;
    printf("Neue UID: ");
    scanf("%d",&newuid);
    if (setuid(newuid)==0) {
        printf("UID: %d, EUID: %d\n",
               getuid(), geteuid());
    } else {
        printf("Fehler %d: %s\n", errno, strerror(errno));
    }
    return(0);
}
```

Umgebungsvariablen

Auch von C aus kann auf Umgebungsvariablen zugegriffen werden:

`char* getenv(char* var)` gibt den Wert der Umgebungsvariablen *var* zurück.

`int putenv(char* string)` erwartet einen String der Form `var=value` und setzt die Variable entsprechend. Ein evtl. existierende Variable wird überschrieben. Rückgabewert ist 0 bei Erfolg, sonst $\neq 0$.

`int unsetenv(char* var)` entfernt die entsprechende Variable, Rückgabewert wie zuvor.

Beispiel

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    printf("Home: %s\n", getenv("HOME"));
    putenv("FOO=bar");
    unsetenv("ABCDEFG");
    return(0);
}
```


Prozess-Infos unter /proc

Unterhalb von `/proc` gibt es ein Verzeichnis für jeden Prozess mit der PID als Name.
Dieses enthält Informationen über den Prozess, u. a.:

- `cmdline` Kommandozeile
- `env` Liste der Umgebungsvariablen in der Form `var=value`, mit `\0` getrennt.
- `fd` Verzeichnis mit den Filedescriptoren für stdin (0), stdout (1), stderr (2).
- `root` Link auf das Root-Verzeichnis des Prozesses.
- `exec` Link auf des Executable.

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 <stdlib.h>
#include <string.h>
#include <errno.h>
#define BUF_LEN 255

int main() {
    int pipe_fd[2];

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

    write(pipe_fd[0], "Hallo", strlen("Hallo"));

    char zeile[BUF_LEN+1];
    int len = read(pipe_fd[0], zeile, BUF_LEN);
    string[len] = '\0';
    printf("Gelesen: %s\n", zeile);
    return(EXIT_SUCCESS);
}
```


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.