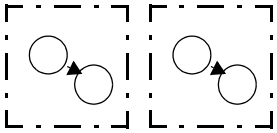
E 1 Parallele Prozesse unter Unix



Lernziele

- Prozessverdoppelung und Prozessverkettung unter Unix praktisch durchführen
- Wichtige Eigenschaften dieser Systemfunktionen kennen lernen
- Systemaufrufe zur Prozessverwaltung in eigenen Programmen einsetzen (inkl. Fehlerbehandlung)
- Mit Unix-Handbuchseiten arbeiten lernen

Übungsumgebung

- Diese Übung kann unter Unix oder Linux ausgeführt werden. Etwaige Unterschiede sind bezeichnet mit:
  SVR4: gilt für Unix System V Rel. 4 (z.B. Sun Solaris)
  Linux: gilt für Linux 2.4 und neuer
- GCC (GNU Compiler Collection) installiert
- Vorlagedateien (C-Quellcode): **`forktest.c`**, **`demoexec.c`**

Allgemeine Hinweise

- Notieren Sie sich Ihre Antworten zu den Fragen auf dem Aufgabenblatt für späteres Nachschlagen.
- Die Lösung dieser Übung besteht in praktischer Selbsterarbeitung, die Musterlösung dient nur zur nachträglichen Kontrolle.
- Eine besser aufbereitete Version der Unix-Handbuchseiten ist auf dem Web unter:
  `http://www.opengroup.org/onlinepubs/009695399/toc.htm`
  zu finden. Auf dieser Webseite ist zweimal "System Interfaces" zu wählen, um eine Übersichtsliste aller Systemaufrufe zu erhalten. Über diese können detaillierte Beschreibungen abgefragt werden.

Aufgaben

E 1.1 Systemaufruf **`wait()`**.
Hinweis: Anwort zu den Fragen gibt **`man 2 wait`** (Linux) bzw. **`man -s2 wait`** (SVR4)

a) Was ist das Resultat des Systemaufrufs **`wait()`**?

...on success returns the process id of the terminated child
on error -1
...

b) Welche Bedeutung hat der Funktionsparameter in **`wait()`**?

&status -> store status information (int) to which it points ....

.....

c) Wie muß ich **`waitpid()`** aufrufen, damit ich das gleiche Verhalten habe wie bei **`wait()`**?

waitpid(-1, &status,0);

...

.....

d) Wie prüfe ich beim bzw. nach dem **`wait()`**-Aufruf, ob der Kindprozess normal terminierte?

WIFEXITED(status) -> returns true if the child terminated normaly

e) Wie erhalte ich im Elternprozess den Wert **`x`**, den der Kindprozess mit **`exit(x)`** als Rückgabewert bereitstellt?

E 1.2 Prozessverdoppelung mit **`fork()`**

Testen Sie die Prozessverdoppelung durch **`fork()`** mit dem Beispielsprogramm **`forktest.c`**.
Der Quellcode kann mit **`gcc forktest.c -o forktest`** in die ausführbare Datei **`forktest`** übersetzt werden.

Für die Beobachtung der Prozesse ein anderes Fenster öffnen und vorhandene Prozesse mit **`ps -la`** (auch **`ps -lea`** verwenden) ermitteln. Prozesstabelle vor, während des Parallelablaufs und nach Beendigung des ersten Prozesses anschauen (*Linux*: unter Gnome steht das komfortable Programm "Systemmonitor" zur Verfügung). Anschließend Programm starten mit **`forktest`** (falls dies nicht geht mit **`./forktest`**) und Ablauf beobachten (inklusive **`ps ...`**).

a) Gibt es Zombies? (Eintrag **`<defunct>`** bei SVR4 bzw. "**`z`**" bei Linux in Prozesstabelle)

Nein, nur mit zusätzlichem while(1);

b) Entfernen Sie aus dem Programm **`forktest.c`** den **`wait()`**-Aufruf. Wie steht es nun mit den Zombies?

c) Leben von Ihnen gestartete Prozesse weiter, auch wenn Sie sich vom System abmelden? Frage für Vordergrund- und Hintergrundprozesse getrennt beantworten.
Hinweis: einen Hintergrundprozess starten Sie, indem Sie auf der Kommandozeile dem Namen der ausführbaren Datei ein **`&`** hinten anhängen (**`z.B. ./forktest&`**).

Vordergrund: Nein
Hintergrund: Ja

d) Wie Teilaufgabe c), aber mit Kommandozeilenbefehl **`nohup ./forktest`**. Lebt der Prozess nun über das Abmelden hinaus weiter? Und wie sieht es mit der Konsolenausgabe des Prozesses aus?

JA

E 1.3 Prozesse ersetzen bzw. verketten unter Unix (Chaining).

a) Ermitteln Sie mit Hilfe der Manual-Page von **`execl`** die *Unterschiede* von:
`execl, execv, execle, execlp, execvp`

Beschreiben Sie die Unterschiede nachfolgend:

.....

.....

.....

.....

.....

.....

b) Was erbt der neue Prozess vom alten Prozess? (Manual-Page von **`execve`** gibt Antwort; **`execve`** ist der System Call, der bei allen Varianten von a) verwendet wird.)

.....

.....

E 1.4 Prozessverkettung

a) Testen Sie das Chaining von Programmen (**`execl`**) mit dem Beispiel **`demoexec.c`** (s. Anhang).

b) Unter welchen Umständen könnte die Ausgabe **`"This should not happen!\n"`** erscheinen?

.....

.....

c) Wieso wird der Text **`"Child just died"`** zweimal ausgegeben, wenn das Programm **`date`** nicht gestartet werden kann?

.....

.....

d) Bekommt der mit **`execl`** gestartete Prozess eine neue PID?

.....

.....

e) Kann der neuen Prozess weiterlaufen, auch wenn der **`execl`** aufrufende Prozess terminiert?
Falls ja: welches ist sein Elternprozess?
Hinweis: Evtl. Programm abändern, so dass diese Fragen experimentell beantwortet werden können.

.....

.....

f) Das Programm soll **`0`** im Normalfall bzw. **`-1`** im Fehlerfall zurückgeben. Erweitern Sie das Programm derart, dass mögliche Fehler behandelt werden. NB: Sie können den Rückgabewert eines Programms auf der Konsole zur Anzeige bringen, indem Sie nach der Programmbeendigung den Befehl **`echo $?`** eingeben.
Mit **`echo $?`** erhalten Sie stets den Rückgabewert des zuletzt auf der gleichen Konsole ausgeführten Programms. Beispiel (**`$`** für Eingabeaufforderung bei Zeilenbeginn):

```
$ demoexec
$ echo $?
0
$
```

E 1.5 Chaining/Prozessverdoppelung

Ändern Sie das Beispiel aus Aufgabe E1.4 derart ab, dass der Kindprozess das Programmbeispiel aus Aufgabe E1.2 startet (oder etwas nach eigenen Ideen).
Praktische Durchführung, Printout (bzw. Datei) des Quellcodes behalten.

E 1.6 Drei Parallelprozesse (optional, bei Interesse und Zeit)

Schreiben Sie ein C-Programm, das drei Prozesse erzeugt, die je einen Text ausgeben und sich dann beenden. Der Elternprozess soll auf das Ende der drei Kindprozesse warten, dann einen Text ausgeben und sich ebenfalls beenden. Praktische Durchführung, Printout (bzw. Datei) des Quellcodes behalten.

Beilagen:

Quelltext von forktest.c:

```
/* Fork-Demo                    */
/* V3 22.8.97 sc                */
/* V4 03.12.04 gle              */

#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>


void warten() { /* "Rechenzeit verbrauchen" */
  int i; float x;

  for (i=0;i<800000;i++) {
    x+=92.77/29388;
  }
```

```
}
int main()
{
  pid_t k, pid;
  int p, status, q;

 k = fork();
  if (k != 0) { /* falls <> 0 --> THEN Eltern*/
    /* Eltern mit k= Prozessnummer des Kindes*/
    for (p=0; p<20; p++) {
      printf("\t\t\tElternteil %d, Kind=%d\n",p,k);
      warten();
    }
    pid=wait(&status);
    printf("\t\t\tIst Kindprozess beendet? PID war, %d\n",pid);
    while(1); /*auf  Abbruch durch <CTRL/C> warten*/
  }
  else { /* Kind mit k=0 */
    printf("Kindprozessnummer= %d \n", getpid());
    for (p=0; p<10; p++) {
      printf ("Kind %d,%d\n",p,k);
      warten();
    }
    exit(1);
  }
}
```

Quelltext von demoexec.c:

```
/* Nach: An Introductory 4.3BSD IPC Tutorial, S. Sechrest */
/* 03.12.04 / gle */

#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>

int main () {
  pid_t pid;
  int status;

  if (fork() == 0) {
    printf ("Child executing \"/bin/date\"\n");
    execl ("/bin/date", "date", 0);
    printf ("This should not happen!\n");
  }
  printf ("Parent waiting for child's dead\n");
  pid=wait (&status);
  printf ("Parent: Child just died.\n");
  exit(0);
```

```
}
```