Lösungen zum Praktikum: "Prozesse, Threads und Dämonen"

Aufgabe 1: top

Resultat selbsterklärend, nur zur Bedienung von top.

Aufgabe 2: Prozesse erzeugen mit fork(): was läuft ab ?

Verstehen was, abläuft, wenn fork() aufgerufen wird. Sie stellen fest:

- dass in beiden Prozessen genau nach fork() weitergefahren wird, weil der Text nach dem switch Befehl ".... und wer bin ich ?" zweimal ausgedruckt wird
- dass die Variable i auch an den Kindprozess übergeben wird
- dass beide Prozesse auf den gleichen Ausgabekanal schreiben und eigentlich nicht klar ist, wer welchen Text ".... und wer bin ich ?" schreibt
- wie PIDs organisiert sind

Hier die Programmausgabe, die Sie sehen sollten:

```
i vor fork: 5

Hallo... ich bin der Elternprozess 398, mein Kind ist 399, stamme von 285 ab
Mein i: 4

Hoi... ich bin das Kind 399, mein Elternprozess ist 398
Mein i: 6

. . . . . und wer bin ich ?

. . . . . und wer bin ich ?
```

Aufgabe 3: Prozess erzeugen und ausführen mit execlp()

Verstehen was, abläuft, wenn ein Kindprogramm mit exec() überlagert werde. Sie stellen fest:

- das Programm (in Unix auch Text genannt) wird durch das neue Programm ersetzt, demzufolge wird der Text nach dem switch Befehl ".... und wer bin ich ?" nur einmal, nämlich vom Elternprozess, ausgedruckt.
- dass die Variable i natürlich ans Kind übergeben werden muss (etwas aufwendig)

Hier die Programmausgabe, die Sie sehen sollten:

```
i vor fork: 5

Hallo... ich bin der Elternprozess 441, mein Kind ist 442, stamme von 283 ab
Mein i: 4

Hoi... ich bin das Kind 442, mein Elternprozess ist 441
Mein i: 6

. . . . . und wer bin ich ?
```

Aufgabe 4: Prozesshierarchie: was fork() alles kann

Beachten Sie, dass ein Prozess jeweils genau beim nächsten Befehl nach fork() weiterfährt. Damit ist es möglich, die Prozesshierarchie aufzuzeichnen.

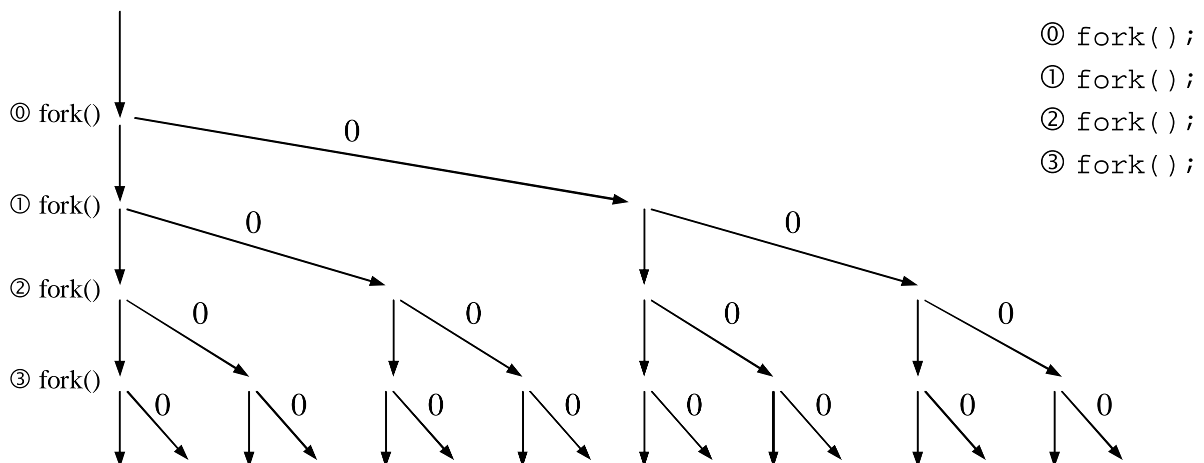
Mit Hilfe der Programmausgabe können Sie Ihr Resultat überprüfen. Der sleep() Befehl wird benötigt, damit kein Prozess stirbt, bevor alle anderen Prozesse erzeugt worden sind (falls dies der Fall wäre, würde PPID = 1 werden: siehe dazu auch Aufgabe 6).

Hier die Programmausgabe, die Sie sehen sollten (die bei PIDs und PPIDs sein bei Ihnen anders, weil Linux PIDs dynamisch, in aufsteigender Reihenfolge, vergibt):

```
PID: 522            PPID: 285
PID: 523            PPID: 522
PID: 524            PPID: 522
PID: 527            PPID: 523
PID: 525            PPID: 522
PID: 528            PPID: 523
PID: 530            PPID: 524
PID: 532            PPID: 527
PID: 526            PPID: 522
PID: 529            PPID: 523
PID: 531            PPID: 524
PID: 533            PPID: 527
PID: 534            PPID: 525
PID: 535            PPID: 528
PID: 536            PPID: 530
PID: 537            PPID: 532
```

Wenn sie im Programm den exit(0) Befehl durch "**`while (1) {}`**" ersetzen, bleiben alle ihre Prozess aktiv hängen.

Eine Handzeichnung kann wie folgt erstellt werden (ohne auf die Prozessidentifikationsnummern zu achten):



Bei jedem `fork()` wird nach rechts das neu erzeugte Kind aufgetragen, "nach unten" läuft der ursprüngliche Prozess weiter. Weiter ist zu beachten, dass alle Prozesse als nächstes die Instruktion unmittelbar nach dem `fork()` ausführen. Untenstehend finden Sie die Ausgabe des Prozessbaumes von **`ps f`**, beachten Sie, dass nun die Prozesshierarchie etwas anders aussieht, der Elternprozesse wird nicht weitergezeichnet.

Mit **ps f** oder **pstree** können Sie die Prozesshierarchie betrachten. Sie sollten dabei etwa folgendes sehen (nicht vollständig, ich habe einige Zeilen gelöscht):

```
  PID TTY STAT TIME COMMAND
  283  p1 S    0:00 -csh
  538  p1 R    0:00  \_ ps -f
  284  p2 S    0:00 -csh
  285  p0 S    0:00 -csh
  522  p0 R    0:02  \_ E
  523  p0 R    0:02      \_ E
  527  p0 R    0:01      |   \_ E
  532  p0 R    0:01      |   |   \_ E
  537  p0 R    0:01      |   |   |   \_ E
  533  p0 R    0:01      |   |   \_ E
  528  p0 R    0:01      |   \_ E
  535  p0 R    0:01      |   |   \_ E
  529  p0 R    0:01      |   \_ E
  524  p0 R    0:01      \_ E
  530  p0 R    0:01      |   \_ E
  536  p0 R    0:01      |   |   \_ E
  531  p0 R    0:01      |   \_ E
  525  p0 R    0:01      \_ E
  534  p0 R    0:01      |   \_ E
  526  p0 R    0:01      \_ E
  290   1 S    0:00 xclock -bg #c0c0c0 -padding 0 -geometry -1500-1500
  299   1 S    0:00 xload -nolabel -scale 1 -bg grey60 -update 5 -geometry
```

Aufgabe 5: Zeitliche Synchronisation: wer macht wann was ?

Hier sollen erfahren, dass nicht vorausgesagt werden kann, in welcher Reihenfolge auch ganz einfache Prozesse abgearbeitet werden. Selbstverständlich müssen, die Programm rechenintensiv sein, also mehr Rechenzeit als den time slice benötigen. Wir haben das im Programm mit einer einfachen for - Schleife realisiert.

Hier z.B. zwei Durchläufe des Programms (sieht bei Ihnen ev. anders aus):

0 Papa	0 Papa
0 Anna Katharina	0 Anna Katharina
1 Papa	1 Papa
2 Papa	2 Papa
1 Anna Katharina	1 Anna Katharina
3 Papa	3 Papa
2 Anna Katharina	2 Anna Katharina
4 Papa	4 Papa
3 Anna Katharina	3 Anna Katharina
5 Papa	5 Papa
6 Papa	6 Papa
4 Anna Katharina	4 Anna Katharina
7 Papa	7 Papa
5 Anna Katharina	5 Anna Katharina
8 Papa	8 Papa
6 Anna Katharina	9 Papa
7 Anna Katharina	6 Anna Katharina
9 Papa	7 Anna Katharina
10 Papa	10 Papa
8 Anna Katharina	8 Anna Katharina
9 Anna Katharina	11 Papa
11 Papa	9 Anna Katharina
12 Papa	12 Papa
13 Papa	10 Anna Katharina
10 Anna Katharina	13 Papa
14 Papa	14 Papa
11 Anna Katharina	11 Anna Katharina
15 Papa	15 Papa
12 Anna Katharina	16 Papa
16 Papa	12 Anna Katharina
17 Papa	17 Papa
13 Anna Katharina	13 Anna Katharina
18 Papa	18 Papa
14 Anna Katharina	14 Anna Katharina
19 Papa	19 Papa
I go it ..	I go it ..
15 Anna Katharina	15 Anna Katharina
16 Anna Katharina	16 Anna Katharina
17 Anna Katharina	17 Anna Katharina
18 Anna Katharina	18 Anna Katharina
19 Anna Katharina	19 Anna Katharina
I go it ..	I go it ..

Aufgabe 6: Was geschieht mit verwaisten Kindern ?

Wenn Sie das Programm genau anschauen, stellen Sie fest, dass der Kindprozess 5 Sekunden lebt, der Elternprozess jedoch nur 2 Sekunden. Frage: was geschieht mit dem verwaisten Kinder. Es gibt zwei Möglichkeiten: das Kind stirbt mit den Eltern, das Kind lebt weiter, muss aber in diesem Fall adoptiert werden (in Unix hängen alle Prozess in einer Hierarchie). Wenn Sie das Programm starten, wird klar wer hier Verantwortung übernimmt: das Kind lebt weiter, und wird von Prozess 1 adoptiert, der auch für das Kind sorgt (z.B. auf seine Terminierung wartet, siehe auch Aufgabe 7). Jetzt ist auch klar, wieso es bei Aufgabe 4 ein sleep() braucht.

Hier die Programmausgabe:

```
Hoi... ich bin das Kind 585

Mein Elternprozess ist 584
Mein Elternprozess ist 584
Mein Elternprozess ist 584
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
Mein Elternprozess ist 1
... so das wars
```

Aufgabe 7: Zombies auch in Unix . . . ?

Diese Aufgabe zeigt Ihnen,

- dass Prozesse die teminieren zu Zombies werden,
- dass Prozesse vom Zombiedasein mit den Funktionen wait() (resp. waitpid()) erlöst werden.

Wie Sie wissen, wird ein Prozess zum Zombie, wenn er terminiert. D.h. er existiert immer noch und belegt Resourcen, aber es lassen sich auch Informationen zum Ablauf, etc. auslesen. Um die Resourcen freizugeben, sollten die Prozesse erlöst werden, das geschieht mit wait(). (Verwaiste, von Prozess 1 adoptierte Kinder, werden automatisch erlöst).

Hier drei Schnappschüsse Ihres Bildschirmes (allerdings mit ps und nicht mit top gemacht):

```
  PID TTY STAT TIME COMMAND
  250   1 S    0:00 -tcsh
  269   1 S    0:00 sh /usr/X11R6/bin/startx
  270   1 S    0:00 xinit /home/tha/.xinitrc --
  274   1 S    0:00 fvwm2
  282   1 S    0:00 /usr/X11R6/lib/X11/fvwm2//FvwmButtons 7 4 .fvwm2rc 0 8
  283  p1 S    0:00 -csh
  284  p2 S    0:00 -csh
  285  p0 S    0:00 -csh
  290   1 S    0:00 xclock -bg #c0c0c0 -padding 0 -geometry -1500-1500
  298   1 S    0:00 /usr/X11R6/lib/X11/fvwm2//FvwmPager 9 4 .fvwm2rc 0 8 0 0
  299   1 S    0:00 xload -nolabel -scale 1 -bg grey60 -update 5 -geometry -1500
  672  p0 S    0:00 e___________.e
  673  p0 S    0:00 e___________.e
  674  p0 S    0:00 e___________.e
  675  p0 S    0:00 e___________.e
  676  p1 R    0:00 ps
  677  p1 R    0:00 -csh
```

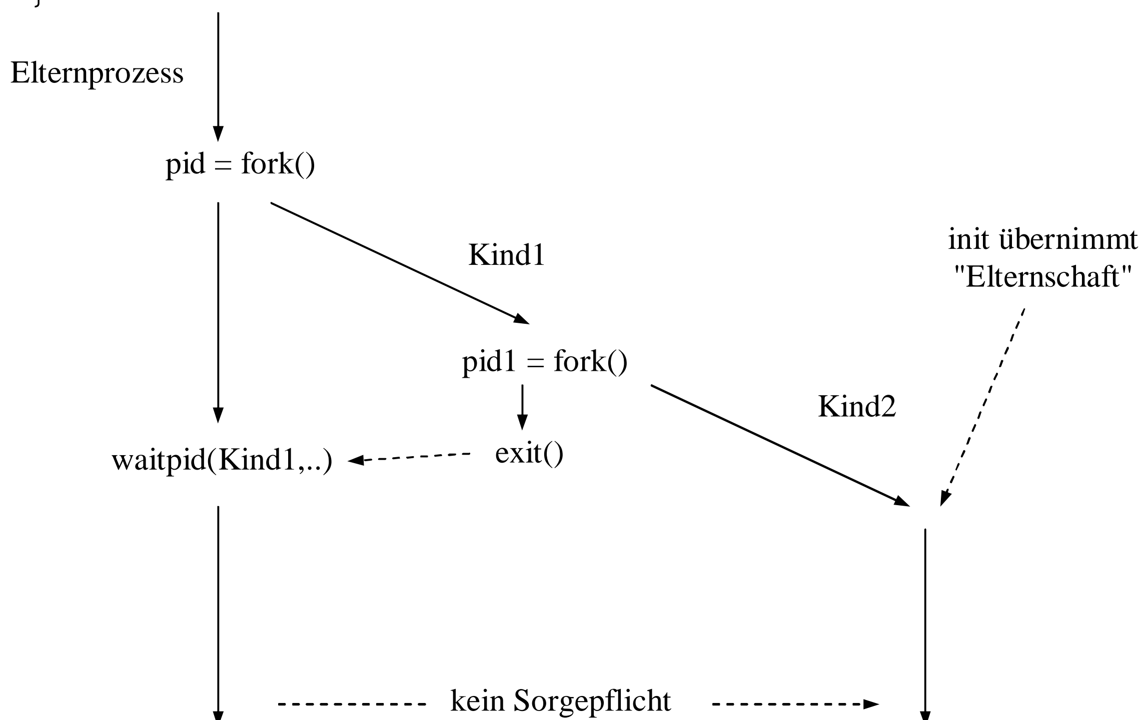
```
  PID TTY STAT TIME COMMAND
  250   1 S    0:00 -tcsh
  269   1 S    0:00 sh /usr/X11R6/bin/startx
  270   1 S    0:00 xinit /home/tha/.xinitrc --
  274   1 S    0:00 fvwm2
  282   1 S    0:00 /usr/X11R6/lib/X11/fvwm2//FvwmButtons 7 4 .fvwm2rc 0 8
  283  p1 S    0:00 -csh
  284  p2 S    0:00 -csh
  285  p0 S    0:00 -csh
  290   1 S    0:00 xclock -bg #c0c0c0 -padding 0 -geometry -1500-1500
  298   1 S    0:00 /usr/X11R6/lib/X11/fvwm2//FvwmPager 9 4 .fvwm2rc 0 8 0 0
  299   1 S    0:00 xload -nolabel -scale 1 -bg grey60 -update 5 -geometry -1500
  672  p0 S    0:00 e___________.e
  673  p0 Z    0:00 (e___________.e <defunct>)
  674  p0 S    0:00 e___________.e
  675  p0 S    0:00 e___________.e
  678  p1 R    0:00 ps
  679  p1 R    0:00 -csh
```

```
  PID TTY STAT TIME COMMAND
  250   1 S    0:00 -tcsh
  269   1 S    0:00 sh /usr/X11R6/bin/startx
  270   1 S    0:00 xinit /home/tha/.xinitrc --
  274   1 S    0:00 fvwm2
  282   1 S    0:00 /usr/X11R6/lib/X11/fvwm2//FvwmButtons 7 4 .fvwm2rc 0 8
  283  p1 S    0:00 -csh
  284  p2 S    0:00 -csh
  285  p0 S    0:00 -csh
  290   1 S    0:00 xclock -bg #c0c0c0 -padding 0 -geometry -1500-1500
  298   1 S    0:00 /usr/X11R6/lib/X11/fvwm2//FvwmPager 9 4 .fvwm2rc 0 8 0 0
  299   1 S    0:00 xload -nolabel -scale 1 -bg grey60 -update 5 -geometry -1500
  672  p0 S    0:00 e__________.e
  673  p0 Z    0:00 (e__________.e <defunct>)
  674  p0 Z    0:00 (e__________.e <defunct>)
  675  p0 S    0:00 e__________.e
  682  p1 R    0:00 ps
  683  p1 R    0:00 -csh
```

Verhindern von Zombies

Zombies entstehen, wenn ein Kindprozess terminiert und der Elternprozess nicht auf das Kind wartet. Wenn ein Elternprozess die Sorge für die Kinder abgeben möchte und keine Zombies entstehen sollen, kann dies folgendermassen erreicht werden (keine Fehler abgefangen):

```
pid = fork()
switch (pid) {
     case 0:    pid1 = fork();
                if (pid1 != 0)    // falls nicht Kind: exit
                     exit(0);
                else
                     ...          // überlebendes Kind, init
                                  // übernimmt Elternpflicht
     default:   waitpid(pid,..); // hier wartet Elternprozsse
                                  // auf exit von Kind1
}
```



Aufgabe 8: Prozessräume und was sie nach dem fork()'en enthalten

Hier lernen Sie, das Daten von Eltern an Kinder vererbt werden, dass die Kinder und Eltern nachher z.T. eigenständig weiterleben.

Teilaufgabe 1

Sie erkennen,

- dass vom Elternprozess erzeugte Daten (her GArray) dem Kind zwar übergeben werden, nach fork() aber als zwei unabhängige Kopien weiterexistieren (Linux verwendet COW copy on write bei der Prozesserzeugung)
- dass damit das Kind eine Kopie des Elternprozesses ist
- dass ein neues, vollständig unabhängiges Prozessimage erzeugt wird

Hier die Programmaugabe:

```
Array vor fork()

- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

Elternarray

p p p p p p p p
p p p p p p p p
p p p p p p p p
p p p p p p p p
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

Kinderarray

- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
c c c c c c c c
c c c c c c c c
c c c c c c c c
c c c c c c c c
```

Teilaufgabe 2

Im Gegensatz zum Programm aus Teilaufgabe 1, werden hier Daten vom Kind- und Elternprozesss in ein File geschrieben. Siehe File AnyOutPut.txt.

Sie erkennen,

- dass die Ausgabe von Eltern- und Kindprozess im gleichen File steht,
- dass wiederum keine Aussgae zur Ausführungsreihenfolge gemacht werden kann,
- dass Filepointer demzufolge vererbt werden, resp. Files nicht geschlossen werden (dies gilt für alle Typen von Filepointern, auch Sockets, Pipes, etc.)

Teilaufgabe 3

Hier sehen Sie, dass auch noch andere Datenstrukturen an das Kind vererbt werden, z.B. zu der Ouputbuffer und der dazugehörige Positionszeiger. Vor fork() wird ein Text in den Buffer geschrieben (er wird nicht ausgegeben, weil kein EndOfLine angefügt wird). Im Kindprozess wird dann weiter Text in den Buffer geschrieben und mit fflush() geleert (d.h. ausgegeben). Die Ausgabe im Kindprozess und Elternprozess ist gleich, beide Textfragmente sind zusammengehängt:

```
Hallo, I am on the way to fork now, ......look: now I am the parent
clear ?

Hallo, I am on the way to fork now, ......look: now I am the child
clear ?
```

Aufgabe 9: Threads, was ist anders ?

Im Gegensatz zu Prozessen, laufen Threads im gleichen Prozessraum. D.h. Threads müssten Lese- und Schreibzugriff auf alle Daten im Elternthread. Das Programm ist wesentlichen eine Realisierung des Programmes aus Aufgabe 8 / 1, allerdings diesmal mit Threads, zudem verwenden wir einen Thread als "Kindthread" und einen Thread als "Elternthread" Wie erwartet, haben beide Zugriff auf GArray (der natürlich global deklariert ist).

Hier die Programmausgabe, die Sie sehen:

```
Array vor Threads
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

... p
p p p p p p p p
p p p p p p p p
p p p p p p p p
p p p p p p p p
- - - - - - - -
- - - - - - - -
- - - - - - - -
- - - - - - - -

... c
p p p p p p p p
p p p p p p p p
p p p p p p p p
p p p p p p p p
c c c c c c c c
c c c c c c c c
c c c c c c c c
c c c c c c c c


... nach Threads
p p p p p p p p
p p p p p p p p
p p p p p p p p
p p p p p p p p
c c c c c c c c
c c c c c c c c
c c c c c c c c
c c c c c c c c
```

Aufgabe 10: . . . und wie schnell sind sie denn ?

Hier lernen Sie, wie man den "Zeitaufwand" von Prozessen messen kann. Dies ist oft wichtig, um festzustellen, wie effizient eine Implementation ist

Hier ein Beispiel: ProcA10_1.e make (zuerst "touch *.cc"):

```
g++ -c  ProcA10_1.cc
g++  ProcA10_1.o -lpthread  -o ProcA10_1.e
g++ -c  ProcA10_2.cc
g++  ProcA10_2.o -lpthread  -o ProcA10_2.e


Befehl:                       make
Uhrzeit:                             1.07
User CPU-time:                0
System CPU-time:                     0
Children user CPU-time:              0.93
Children system CPU-time:     0.14
```

Aufgabe 11: Mr. Daemon, what's the time please ?

Ein "normaler Prozess" wie PlapperMaul stirbt wenn sein Kontrollterminal geschlossen wird.

Ein Daemon hat kein Kontrollterminal und lebt (was er ja auch soll) sogar beim ausloggen weiter.

`ps jx`: im Feld TTY steht bei MrTimeDaemon ein "?": der Prozess hat kein Kontrollterminal