Schmid Kevin
1710418023

# Prozesshierachien

## 1 Prozessmanagement

### 1.1 Speicherbereiche von Prozessen

#### a) Klare Definition der Arbeitsbereiche von Eltern- und Kindprozess im Code

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char **argv){
    printf("--beginning of program\n");

    int counter = 0;
    pid_t pid = fork();

    if (pid > 0){
        // parent process
        counter = counter + 10;
        printf("Parent: %d \n", counter);
    }
    else if (pid == 0){
        counter = counter + 1;
        printf("Child: %d \n", counter);
        // child process
    } else{
        printf("fork() failed!\n");
        return 1;
    }

    printf("--end of program--\n");
    return 0;
}
```

```
main(int argc, char ** argv)   Ln 6, Col 28   Spaces: 4   UTF-8   LF   C
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ gcc schleife.c
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ ./a.out
--beginning of program
Parent: 10
--end of program--
Child: 1
--end of program--
```

Obwohl beide Prozesse auf die Variable `counter` zugreifen, werden sie nicht voneinander beeinflusst.

### b) Ausgabe der PID der Prozesse

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(int argc, char **argv){
    printf("--beginning of program\n");

    int counter = 0;
    pid_t pid = fork();

    if (pid > 0){
        // parent process
        counter = counter + 10;
        printf("Parent PPID:%d PID:%d counter:%d \n", getppid(), getpid(), counter);
    }
    else if (pid == 0){
        counter = counter + 1;
        printf("Child: PPID:%d PID:%d counter:%d \n", getppid(), getpid(), counter);
        // child process
    } else{
        printf("fork() failed!\n");
        return 1;
    }

    printf("--end of program--\n");
    return 0;
}
```

(Global Scope) Ln 30, Col 1 Spaces: 4 UTF-8 LF C

```
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ gcc schleife.c
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ ./a.out
--beginning of program
Parent PPID:5086 PID:7148 counter:10
--end of program--
Child: PPID:7148 PID:7149 counter:1
--end of program--
```

### c) Woran erkennt man die Prozesswechsel? Wer ist dafür zuständig?

Durch die Prozess-ID. Zuständig ist der Process Scheduler (Linux default ist der Completely Fair Scheduler)

### d) Wären die Ausgaben bzw. das Scheduling exakt reproduzierbar? Warum?

Nicht wirklich. Der CFS verwendet einen sich ausbalanzierenden Red-Black-Tree um den Prozess mit der höchsten wait_runtime zu finden. Dieser Wert basiert auf Nanosekunden. Dies exakt zu reproduzieren ist nicht möglich. Mit einem alternativen Process Scheduler könnte man dies eventuell schaffen.

## 1.2 Terminieren eines Prozesses

### 1.2.1 Prozessadoption durch init

**1. Führen Sie das Programm adoption.c aus**

```
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ gcc adoption.c
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ ./a.out
--- Im Elternprozess (9159) ---
--- Im Kindprozess (9160) ---
Elternprozess : 9159
 --- Elternprozess endet vorzeitig ---
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ --- Im Kindprozess (9160) ---
Elternprozess : 2254
_
```

Man sieht wie man nach Beendigung des Elternprozesses wieder in der Shell landet und der Kindprozess danach noch einen Output schreibt.

**2. Monitoren Sie die Prozesstabelle mit den bekannten Programmen und interpretieren Sie das Ergebnis. Dokumentieren Sie den Zusammenhang Prozessstatus in der Prozesshierachie.**

```
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ ./a.out
--- Im Elternprozess (10043) ---
--- Im Kindprozess (10044) ---
Elternprozess : 10043
 --- Elternprozess endet vorzeitig ---
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ --- Im Kindprozess (10044) ---
Elternprozess : 2254
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ _
```

```
kevin@Kevin-XPS15:~$ pstree | grep a.out -C 5
kevin@Kevin-XPS15:~$ pstree -p | grep a.out -C 5
           |                |            |-{snap}(3736)
           |                |            |-{snap}(3737)
           |                |            |-{snap}(3738)
           |                |            |-{snap}(3741)
           |                |            `-{snap}(3743)
           |                `-tilix(5079)-+-bash(5086)---a.out(10043)---a.out(10044)
           |                              |-bash(9185)-+-grep(10046)
           |                              |            `-pstree(10045)
           |                              |-{tilix}(5080)
           |                              |-{tilix}(5081)
           |                              `-{tilix}(5082)
kevin@Kevin-XPS15:~$ pstree -p | grep a.out -C 5
           |                |-{spotify}(8274)
           |                |-{spotify}(8275)
           |                |-{spotify}(8278)
           |                `-{spotify}(8316)
           |-systemd(2254)-+-(sd-pam)(2258)
           |                |-a.out(10044)
           |                |-at-spi-bus-laun(2470)-+-dbus-daemon(2475)
           |                |                       |-{at-spi-bus-laun}(2471)
           |                |                       |-{at-spi-bus-laun}(2472)
           |                |                       `-{at-spi-bus-laun}(2474)
           |                |-at-spi2-registr(2477)-+-{at-spi2-registr}(2484)
```

Hier sieht man wie der Kindprozess im ersten pstree am Elternprozess hängt, beim zweiten bereits von systemd adoptiert wurde.

### 1.2.2 Zombieprozess

Ergänzen Sie die Codestruktur von `zombie.c`

**1. Der Kindprozess wird erzeugt, gleich mit der Systemfunktion sleep(5) schlafen gelegt und dann mit exit(1) beendet.**

```
14          case  0 :   // Code von Kindprozess
15            sleep(5);
16            exit(1);
```

**2. Der Elternprozess soll mit sleep(30) länger schlafen und dann mit exit(2) beendet werden.**

```
17          default :   // Code von Elternprozess
18            sleep(30);
19            exit(2);
```

**3. Öffnen Sie während der Ausführung des Programms eine neuen Konsole und geben sie ps x zum Monitoren aller Prozesse ein:**

```
kevin@Kevin-XPS15:~$ ps x -ejf | sed -n -e '1p' -e '/a.out/p'
UID        PID  PPID  PGID   SID  C STIME TTY      STAT   TIME CMD
kevin    11876  5086 11876  5086  0 11:23 pts/0    S+     0:00 ./a.out
kevin    11877 11876 11876  5086  0 11:23 pts/0    S+     0:00 ./a.out
kevin    11879  9185 11878  9185  0 11:23 pts/1    S+     0:00 sed -n -e 1p -e /a.out/p
kevin@Kevin-XPS15:~$ ps x -ejf | sed -n -e '1p' -e '/a.out/p'
UID        PID  PPID  PGID   SID  C STIME TTY      STAT   TIME CMD
kevin    11876  5086 11876  5086  0 11:23 pts/0    S+     0:00 ./a.out
kevin    11877 11876 11876  5086  0 11:23 pts/0    Z+     0:00 [a.out] <defunct>
kevin    11885  9185 11884  9185  0 11:24 pts/1    S+     0:00 sed -n -e 1p -e /a.out/p
```

a. Wo ist der Eltern- bzw. der Kindprozess zu sehen?

Anhand der PID und PPID ist dies gut erkennbar.

b. Welcher Prozessstatus fällt dabei auf?

Nach 5 Sekunden fällt der `defunct` bzw +Z Status auf beim Zombie Prozess.

## 1.3 Prozessverkettung mit exec

**1. Alle Prozesse auf dem System anzeigen lassen ps –f**

```
kevin@Kevin-XPS15:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
kevin     9185  5079  0 10:52 pts/1    00:00:00 /bin/bash
kevin    12606  9185  0 11:42 pts/1    00:00:00 ps -f
```

Schmid Kevin
1710418023

**2. Einen Prozess bash starten**

```
kevin@Kevin-XPS15:~$ bash
```

**3. Prozesse auf dem System neu anzeigen lassen ps –f**

```
kevin@Kevin-XPS15:~$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
kevin       9185  5079  0 10:52 pts/1    00:00:00 /bin/bash
kevin      12730  9185  0 11:45 pts/1    00:00:00 bash
kevin      12765 12730  0 11:45 pts/1    00:00:00 ps -f
```

**4. Kommando exec ps –f ausführen – was passiert?**

```
kevin@Kevin-XPS15:~$ exec ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
kevin       9185  5079  0 10:52 pts/1    00:00:00 /bin/bash
kevin      12730  9185  0 11:45 pts/1    00:00:00 ps -f
```

Der zuvor gestartete bash Prozess wurde überlagert. Ein erneutes ausführen des exec Befehls hat die Bash geschlossen.

**5. Prozesse auf dem System neu anzeigen lassen ps –f**

```
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
kevin       5086  5079  0 09:50 pts/0    00:00:00 /bin/bash
kevin      13156  5086  0 11:49 pts/0    00:00:00 ps -f
```

Schmid Kevin
1710418023

**Aufbauend auf ein fork-Programm von oben soll der Elternprozess mit Hilfe von exec() durch die Systemfunktion date() ersetzt werden:**

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

int main (void) {
    pid_t pid;
    switch (pid = fork()) {
        case -1:
            printf("Fehler bei fork()\n");
            exit(EXIT_FAILURE);
        case 0:
            printf("...Kindprozess...\n");
            printf("[Kind] Die PID des Kindprozesses ist %d\n",getpid());
            printf("[Kind] Die PID des Elternprozesses ist %d\n", getppid());
            execl("/bin/date", "-u", NULL);
        default:
            sleep(1);
            printf("...Elternprozess...\n");
            printf("[Eltern] Die PID des Elternprozesses ist %d\n", getpid());
            printf("[Eltern] Die PID des Kindprozesses ist %d\n",pid);
            printf("[%d ]Programmende\n", getpid());
    }
    return EXIT_SUCCESS;
}

```

(Global Scope) Ln 26, Col 1 Spaces: 4 UTF-8

```
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ ./a.out
...Kindprozess...
[Kind] Die PID des Kindprozesses ist 14489
[Kind] Die PID des Elternprozesses ist 14488
So Apr 28 12:06:19 CEST 2019
...Elternprozess...
[Eltern] Die PID des Elternprozesses ist 14488
[Eltern] Die PID des Kindprozesses ist 14489
[14488 ]Programmende
```

Schmid Kevin
1710418023

# 2 Prozesshierachie

## 2.1 Prozesshierarchie implementieren

### 1. Führen Sie das Programm aus und monitoren Sie die Prozesshierachie mit den PIDs pstree -p und versuchen Sie die generierten Prozesse zu sehen

```
kevin@Kevin-XPS15:~$ pstree -p | grep a.out -C 5
           |               |            |-{snap}(3736)
           |               |            |-{snap}(3737)
           |               |            |-{snap}(3738)
           |               |            |-{snap}(3741)
           |               |            `-{snap}(3743)
           |               `-tilix(5079)-+-bash(5086)---a.out(16499)-+-a.out(16500)-+-a.out(16503)---a.out(16506)
           |                            |                           |               `-a.out(16505)
           |                            |                           |-a.out(16501)---a.out(16504)
           |                            |                           `-a.out(16502)
           |                            |-bash(16269)-+-grep(16508)
           |                            |             `-pstree(16507)
           |                            |-{tilix}(5080)
           |                            |-{tilix}(5081)
           |                            `-{tilix}(5082)
```

### 2. Wie viele Kindprozesse wurden hier genau erzeugt?

7 Kindprozesse

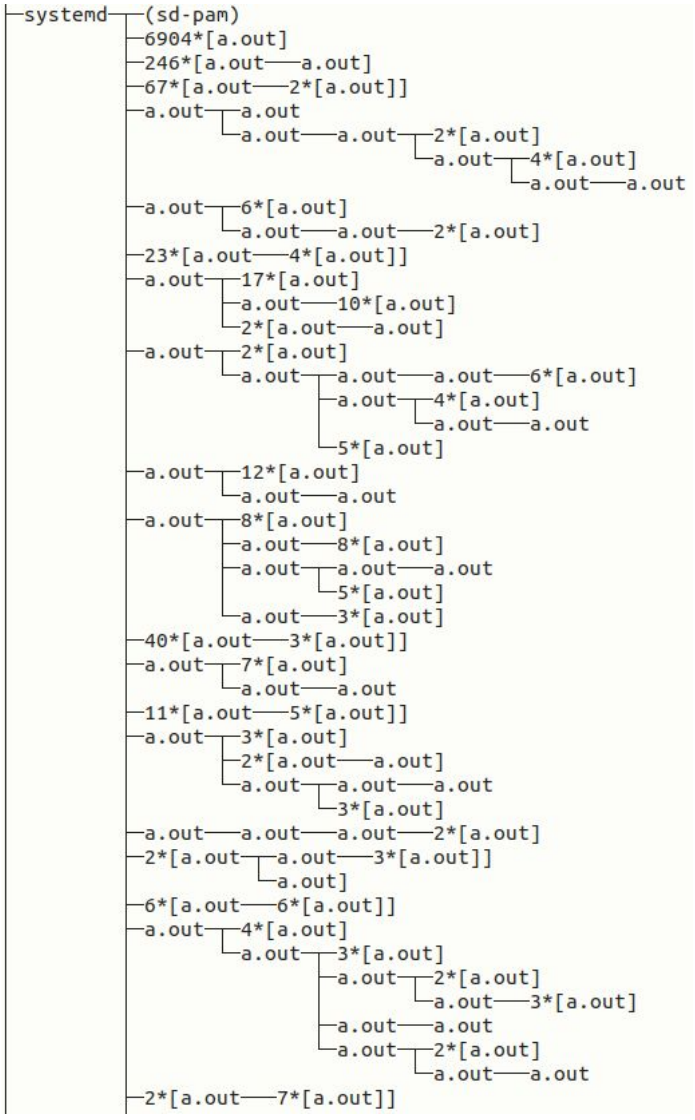
## 2.2 Prozesshierarchie und Systemgrenzen

### 1. Führen Sie das Programm hierachie_schleife.c aus

```
sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab g
eht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´s
ab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab ge
ht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sa
b geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geh
t´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab
 geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht
´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab
geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´
sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab g
eht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´s
ab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab ge
ht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sa
b geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geh
t´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab
 geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht
´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab
geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´
sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab g
eht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´sab geht´s
```

Schmid Kevin
1710418023

**2. Versuchen Sie parallel in einem anderen Terminal pstree zu starten und die Prozesshierachie zu sehen. Warum kann das manchmal schwierig sein bzw. gar nicht funktionieren?**



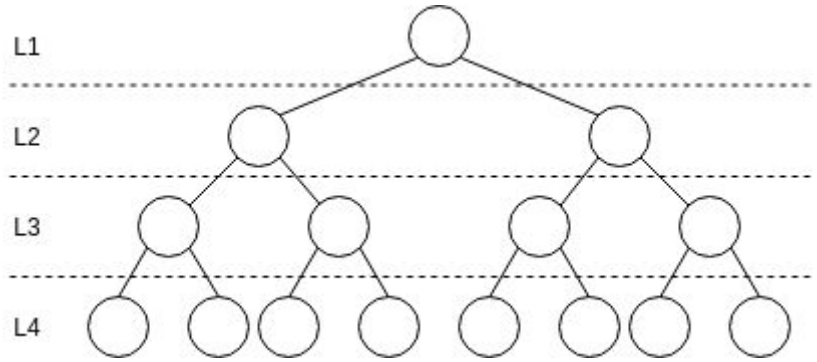

Zu viele gestartet Prozesse können das System lahmlegen.

**3. Was passiert hier prinzipiell und was wird ausgegeben?**

Es werden 30 Prozesse in einer Schleife erzeugt, die alle wiederum 30 Prozesse in endloser Schleife erzeugen. Sobald der fork Befehl fehlschlägt wird der Text ausgegeben.

**4. Wie viele Kindprozesse werden erzeugt? Skizzieren Sie die ersten 3-4 Ebenen der Prozesshierachie.**

$2^{30}$-1 Kindprozesse




**5. Ändern sie den Zähler in der Schleife und versuchen Sie eine rechenbare Grenze zu ermitteln. Von was kann der maximale Wert der Schleife bzw. Prozesserzeugung abhängen, bevor das System aufgibt?**

```
kevin@Kevin-XPS15:~$ cat /proc/sys/kernel/pid_max
32768
```

$2^{15}$ ist somit das vom System vorgegebene mögliche Maximum an gleichzeitig abarbeitbaren Prozessen. Ein kurzer Test bestätigt, dass mit $2^{14}$ das System ohne Probleme funktioniert.

**6. Überleitung zum nächsten Thema: welches Problem könnte prinzpiell auftreten, wenn mehrere fork-Aufrufe auf gemeinsame Variablen zugreifen wollen?**

Falls die Variablen großen Speicher einnehmen könnte so schnell der verfügbare Speicher verbraucht werden.

## 2.3 Design einer Prozesshierarchie

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

void main(void){
  if(fork()) {
    if(fork()) {
      printf("Parent A of all: %d\n", getpid());
    } else {
      if(fork()) {
        printf("Child B with id: %d and it's Parent id: %d\n", getpid(), getppid());
      } else {
        printf("Child D with id: %d and it's Parent id: %d\n", getpid(), getppid());
      }
    }
  } else {
    if(fork()) {
      printf("Child C with id: %d and it's Parent id: %d\n", getpid(), getppid());
    } else {
      printf("Child E with id: %d and it's Parent id: %d\n", getpid(), getppid());
    }
  }
  exit(EXIT_SUCCESS);
}
```

Schmid Kevin
1710418023

```
kevin@Kevin-XPS15:~/FH/S4/sysprog/ue3$ ./a.out
Parent A of all: 19625
Child C with id: 19626 and it's Parent id: 19625
Child B with id: 19627 and it's Parent id: 19625
Child E with id: 19628 and it's Parent id: 19626
Child D with id: 19629 and it's Parent id: 19627
```

# 3 Kontrollfragen

## 1. Prozessverwaltung

**a. Wie viele Prozesszustände braucht ein Prozessmodell mindestens? Warum?**

running und waiting um den Zustand der aktiven Verarbeitung und dem Warten auf Verarbeitung darstellen zu können.

**b. Mit welcher Technik kommunizieren Prozesse untereinander prinzipiell? Welche „Adressen“ werden bei dieser Kommunikation verwendet?**

IPC: Interprocess Communication. Der bestimmte Adressraum shared memory wird dafür genutzt.

**c. Welche Aufgaben muss ein OS prinzipiell durchführen, um einen neuen Prozess erzeugen zu können?**

Speicherbereich zur Verfügung stellen, eine ID generieren, diese mit einem Elternprozess in eine Prozesstabelle eintragen.

**d. Mit welcher Technik (und mit welchem Kommando) kann das Scheduling von Prozessen vom Benutzer beeinflusst werden?**

Mit dem Befehl `renice` kann man die Priorität eines Prozesses ändern.

**e. Linux-Cross-Reference: Wie ist die PID in Linux prinzipiell definiert?**

Die PID ist die eindeutige Identifikation eines Prozesses im Zahlenraum 1 bis `/proc/sys/kernel/pid_max`

## 2. Prozesserzeugung

**a. Welche drei Möglichkeiten gibt es, um einen Prozess zu erzeugen?**

Es gibt nur eine Möglichkeit einen Prozess zu erzeugen: `fork`. Als eine Art von Erzeugung kann auch `exec` angesehen werden, der einen Prozess ersetzt bzw `system`, was im Grunde nur ein Wrapper für `fork`, `exec` ist.

**b. Wie heißt der „erste Prozess“ unter Linux und welche PID hat dieser meistens?**

`init` mit der PID 1

Schmid Kevin
1710418023

**c. Durch das Erzeugen immer neuer Kind-Prozesse mit fork() kann ein beliebig tiefer Baum von Prozessen erzeugt werden – richtig oder falsch? Begründen Sie Ihre Antwort.**

Nein. Theoretisch würde nichts dagegen sprechen, doch irgendwann erreicht man die Grenzen der Hardware.

**d. Bei der Prozesserzeugung gibt weitere interessante Systemfunktionen. Beschreiben Sie die prinzipielle Funktionsweise und Syntax folgender System Calls: kill, clone, den Unterschied zwischen clone und fork. Dokumentation mit Codeausschnitte und/oder kleinem Programm.**

```
int kill(pid_t pid, int sig);
```

Der Systemaufruf kill() kann verwendet werden, um ein beliebiges Signal an eine Prozessgruppe oder einen Prozess zu senden.

```
int clone( int (*fn)(void *), void *child_stack, int flags,
           void *arg, ...)
```

clone() erzeugt einen neuen Prozess, ähnlich wie fork(). Im Gegensatz zu fork() ermöglicht clone() dem Kindprozess, Teile seines Ausführungskontexts mit dem aufrufenden Prozess zu teilen, wie z.B. den virtuellen Adressraum, die Tabelle der Dateideskriptoren und die Tabelle der Signalhandler.

```c
static int child_func(void* arg) {
  char* buf = (char*)arg;
  printf("Child sees buf = \"%s\"\n", buf);
  strcpy(buf, "hello from child");
  return 0;
}

int main(int argc, char** argv) {
  // Allocate stack for child task.
  const int STACK_SIZE = 65536;
  char* stack = malloc(STACK_SIZE);
  if (!stack) {
    perror("malloc");
    exit(1);
  }

  // When called with the command-line argument "vm", set the CLONE_VM flag on.
  unsigned long flags = 0;
  if (argc > 1 && !strcmp(argv[1], "vm")) {
    flags |= CLONE_VM;
  }

  char buf[100];
  strcpy(buf, "hello from parent");
  if (clone(child_func, stack + STACK_SIZE, flags | SIGCHLD, buf) == -1) {
    perror("clone");
    exit(1);
  }

  int status;
  if (wait(&status) == -1) {
    perror("wait");
    exit(1);
  }

  printf("Child exited with status %d. buf = \"%s\"\n", status, buf);
  return 0;
}
```

Quelle:
https://github.com/eliben/code-for-blog/blob/master/2018/clone/clone-vm-sample.c

**e. Richtig oder falsch:**

**i. Ein Elternprozesse kann mehrere Kindprozesse haben, aber ein Kindprozess nur einen Elternprozess**

richtig

**ii. Der neue Kindprozess ist im ersten Schritt ohne zusätzlichen Code eine identische Kopie des Elternprozesses. Beide besitzen dieselben Daten, denselben Befehlszähler, dieselben offenen Dateien, dieselbe User-ID und dasselbe Arbeitsverzeichnis.**

richtig

**iii. Auch Pages vom Heap-Speicher werden sofort vom Eltern- zum Kindprozess übergeben**

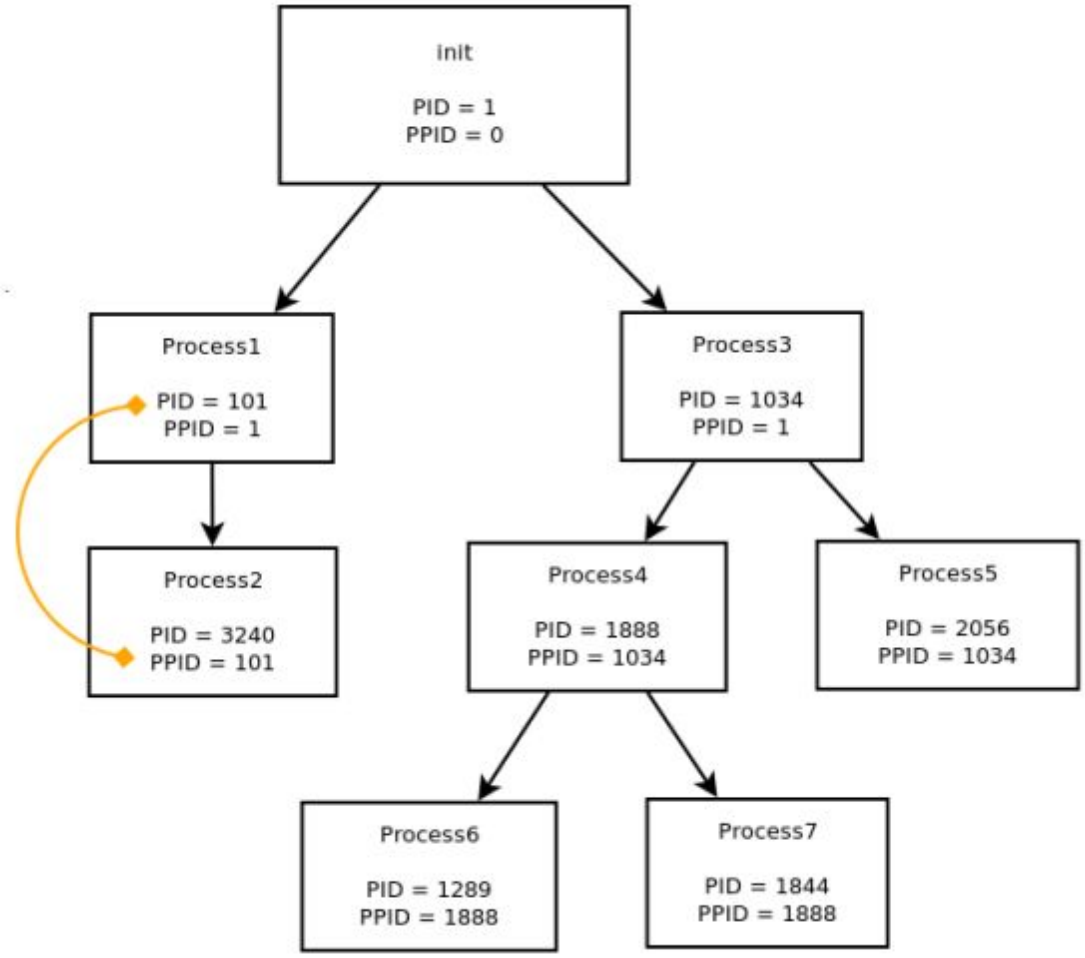
falsch

Schmid Kevin
1710418023

**iv. Mit sleep() kann die Reihenfolge der Ausgabe nicht synchronisiert werden, aber verzögert, was eine bestimmte Reihenfolge bedingt**

richtig

**f. Interpretieren und beschreiben Sie die Bedeutung folgender Abbildung mit Schwerpunkt Prozesshierarchie, init, PID und PPID**




In diesem Bild ist eine Prozesshierarchie abgebildet. An oberste Stelle der init Prozess der mit der PID 1 das System initialisiert und von dem alle weiteren Prozesse ableiten. Process1 und Process3 sind direkte Kinder des init Prozesses (erkennbar an der PPID). Diese besitzen wiederum Kinder die von Ihnen abgeleitet sind. Bei der PID / PPID Verbindung von Process1 und Process2 ist dies neben der PID / PPID ansich noch zusätzlich visuell mit der orangen Relation gekennzeichnet.

Schmid Kevin
1710418023

## 3. Prozessterminierung

**a. Listen Sie die Schritte auf, mit denen ein Prozess korrekt beendet werden kann**

Alle Resourcen des Prozesses müssen freigegeben werden. Danach wird ein SIGCHILD Signal an den Elternprozess gesendet. Der Elternprozess der durch wait() auf das Signal reagiert trägt den Kindprozess aus der Prozesstabelle aus.

**b. Wie ist ein Waisenprozess, wie ein Zombieprozess definiert?**

Waisenprozesse sind Prozesse deren Elternprozess terminiert wurde. Der init Prozess adoptiert den Waisenprozess.
Zombieprozesse sind terminierte Prozesse bei denen der Elternprozess nicht informiert wurde. Somit bleibt der terminierte Prozess weiterhin in der Prozesstabelle erhalten

**c. Was wird passieren, wenn das Signal SIGCHLD vom Kindprozess an den Elternprozess verloren geht?**

Der Kindprozess ist dadurch ein Zombieprozess.

**d. Was passiert, wenn wait() nicht aufgerufen wird? Wie kann man diesen Zustand sehen?**

So entsteht ebenfalls ein Zombieprozess.

**e. Angenommen viele unterschiedliche Prozesse beenden sich: welchen Prozess gibt es, der immer für alle Prozesse bis zum Herunterfahren des Systems als Elternprozess zur Verfügung steht?**

Der init Prozess