PROZESSE

Hinweise: Sämtliche Ausführungen gelten für unixoide Betriebssysteme. Lesen Sie bitte auch die zugehörige Praktikumsdokumentation durch.

1. GRUNDLAGEN

Für Anwender von Rechnern ist die parallele Ausführung von Programmen selbstverständlich. Für Prozessoren ist diese parallele Ausführung aber ganz und gar nicht selbstverständlich: Sie arbeiten sequenziell die Befehle ab. Damit für den Benutzer der Eindruck von Parallelität entstehen kann, führt der Prozessor jedes Programm nur einen kurzen Augenblick aus und wechselt dann zum nächsten Programm. Das macht der Prozessor aber nicht selbstständig. Darum kümmert sich das Betriebssystem (siehe hierzu die Vorlesung).

Um die Abwicklung und Steuerung der einzelnen Programme zu vereinfachen, wurde das Prozessmodell eingeführt. Vereinfacht ausgedrückt ist ein Prozess ein Programm während seiner Ausführung. Jeder Prozess hat eine ganze Reihe von Attributen. Für dieses Übungsblatt werden insbesondere die Prozessnummer (PID) und die Prozessnummer des Vaterprozesses (PPID) gebraucht.

Die Prozessnummer ist eine systemweit eindeutige Nummer, die vom Betriebssystem fortlaufend vergeben wird. Der Vaterprozess ist immer der übergeordnete Prozess, der einen anderen untergeordneten Prozess gestartet hat. Auf Grundlage dieses Mechanismus entsteht eine Prozesshierarchie. An der Spitze der Hierarchie steht der Initialisierungsprozess (`init`), der alle anderen Prozesse startet.

2. PROZESSE VERWALTEN

Um herauszufinden, welche Prozesse gerade auf dem System ausgeführt werden, existiert der Befehl `ps` (***p**rocess **s**tatus*). Mit dem Befehl `man ps` können Sie die Dokumentation zu den möglichen Parametern für den Befehl aufrufen.

Aufgabe 2.1: Veranlassen Sie mit Hilfe des Befehls `ps` eine Ausgabe der PIDs und PPIDs aller laufenden Prozesse. Filtern Sie das Ergebnis mit Hilfe des Befehls `grep` nach einem bestimmten Prozess, den Sie zuvor gestartet haben (z. B. `gedit`).

Leider sind die Argumente von `ps` nicht sehr intuitiv. Weiterhin gibt es in Linux den Befehl `top`. Dieser erzeugt eine ganz ähnliche Ausgabe wie `ps`, welche aber regelmäßig aktualisiert wird. Auch der Befehl `top` verfügt über eine man-Page, die mit `man top` aufgerufen werden kann.

Prozesse werden mit dem Befehl `kill` beendet. `kill 3412` beendet z. B. den Prozess mit der PID 3412. Selbstverständlich existiert auch zu `kill` eine man-Page: `man kill`

Aufgabe 2.2: Starten Sie zwei Terminalinstanzen. Von der ersten Instanz starten Sie ein beliebiges Programm (z.B. `gedit`). In der zweiten Terminalinstanz ermitteln Sie (wie in Aufgabe 2.1) die zugehörige PID. Anschließend beenden Sie den Prozess mit dem Befehl `kill` und der ermittelten PID.

3. PROZESSE ERKENNEN

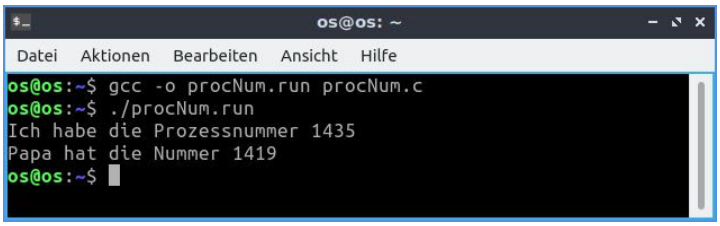
Im Folgenden wird mit der Programmiersprache C gearbeitet. Der Compiler `gcc` sollte in allen Linux-Distributionen enthalten sein und ein Texteditor sollte für die Bearbeitung der Aufgaben ausreichen. Selbstverständlich steht es Ihnen aber frei eine Entwicklungsumgebung einzusetzen (wie z. B. das bereits installierte *Eclipse*).

Die Funktionen

```
pid_t getpid(void);
pid_t getppid(void);
```

liefern die PID und PPID eines Prozesses. Damit die beiden Funktionen eingesetzt werden können, sollten Sie die Header-Dateien `unistd.h` und `sys/types.h` einbinden.

Aufgabe 3.1: Schreiben Sie ein Programm, das folgende Ausgabe erzeugt:



Aufgabe 3.2: Welcher Prozess steckt in Aufgabe 3.1. hinter „Papa“?

Meilenstein 1: Das Programm von Aufgabe 3.1 funktioniert so wie beschrieben.

4. FORK

Der Systemaufruf `fork` ist für UNIX-Systeme essentiell. Genau genommen ist `fork` der einzig existierende Systemaufruf zur Erzeugung von Prozessen. Wird `fork` aufgerufen, erzeugt das System eine exakte Kopie des aufrufenden Prozesses. Ein Befehl, der nach einem `fork`-Aufruf ausgeführt wird, wird also sowohl vom Vaterprozess als auch vom Kindprozess ausgeführt. Ein Befehl der vor einem `fork`-Aufruf ausgeführt wird, wird nur vom Vaterprozess ausgeführt (der Kindprozess existiert ja noch gar nicht).

In C ist der Systemaufruf mit der Funktion

```
pid_t fork(void);
```

implementiert. Die Funktion wird ebenfalls über die Header-Datei `unistd.h` eingebunden.

Wie auch schon bei `getpid` und `getppid` liefert die Funktion eine Prozessnummer zurück. Beim Aufruf von `fork` ist dies aber nicht die eigene PID. `fork` liefert auf Seiten des Vaterprozesses die PID des Kindprozesses. Auf der Seite des Kindprozesses liefert `fork` den Wert 0. Wenn ein Fehler auftritt, wird -1 zurückgegeben.

Es ist also relativ einfach zu unterscheiden, ob das Programm ein Vaterprozess oder ein Kindprozess ist. Somit kann auch ein Kindprozess ein anderes Verhalten zeigen als der Vaterprozess, auch wenn beide exakt identische Kopien zum Zeitpunkt des `fork`-Aufrufes sind. Betrachten Sie hierzu das folgende Beispiel:

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

int main( void ) {
   int procId;
   procId = fork();
   switch(procId){
      case -1:    // es ist ein Fehler aufgetreten
         break;
      case 0:     // Kindprozess
         printf("Ich bin der Kindprozess\n");
         break;
      default:    // Vaterprozess
         printf("Ich bin der Elternprozess\n");
         break;
   }
   return EXIT_SUCCESS;
}
```

Aufgabe 4.1: Wie viele Prozesse werden von folgendem *Listing* gestartet? Erklären Sie die Anzahl anhand einer Zeichnung (Baumstruktur). Wie viele Prozess würden gestartet werden, wenn noch ein fünfter `fork`-Befehl dazu käme?

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

int main( void ) {
   fork();
   fork();
   fork();
   fork();
   return EXIT_SUCCESS;
}
```

Aufgabe 4.2: Erweitern Sie das *Listing* um eine Ausgabe der PID vor der Beendigung des Prozesses. Führen Sie das Programm mehrmals aus. Was fällt Ihnen auf und wie erklären Sie sich diesen Effekt? Beachten Sie auch die Reihenfolge der ausgegebenen PIDs.

Die Effekte, die Sie in Aufgabe 4.2 beobachten konnten, sind natürlich nicht immer erwünscht. Damit solche Effekte vermieden werden können, existieren die Befehle

```
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

Die Befehle sind in der Header-Datei `sys/wait.h` deklariert. `wait` wartet auf die Beendigung eines beliebigen Kindprozesses (nicht aber auf die Beendigung mehrerer Kindprozesse), bevor die Ausführung des anschließenden Codes fortgeführt wird.

`waitpid` wartet auf die Beendigung eines konkreten Prozesses, dessen PID als erster Parameter übergeben wird.

Aufgabe 4.3: Analysieren Sie das folgende *Listing*. In welcher Reihenfolge werden die Prozesse gestartet und in welcher Reihenfolge werden die Prozesse beendet? Begründen Sie ihre Annahme grafisch (Baumstruktur). Was hat sich jetzt im Vergleich zur Aufgabe 4.1 verändert?

```
/* forkVielWait.c */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main( void ) {
   int status;

   fork();
   wait(&status);
   fork();
   wait(&status);
   fork();
   wait(&status);
   fork();
   wait(&status);

   return EXIT_SUCCESS;
}
```

Meilenstein 2: Die Baumstruktur liegt vor.

5. Waisen und Zombies

Mit dem Befehl

```
void exit(int status);
```

können Prozesse beendet werden. Mit der Funktion

```
unsigned int sleep(unsigned int seconds);
```

kann der Prozess dazu veranlasst werden, die übergebene Anzahl von Sekunden zu warten.

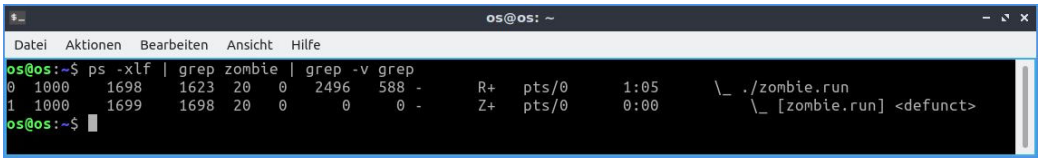
Eine Waise ist ein Prozess, dessen Vaterprozess beendet wurde, bevor er selbst terminiert. In vielen Linux-Distributionen (nicht in allen!) adoptiert in diesem Fall der `init`-Prozess (PID 1) den verwaisten Prozess.

Aufgabe 5.1: Schreiben Sie ein Programm, welches eine Waise erzeugt. Lassen Sie hierzu den Kindprozess länger schlafen als den Vaterprozess. Ihr Programm sollte am Ende folgende Ausgabe erzeugen:

Bei Zombies verhält es sich genau umgekehrt: Hier beendet sich ein der Kindprozess ohne dass dies vom Vaterprozess behandelt wird. Der Vaterprozess enthält also keinen `wait`-Befehl, welcher auf das Beenden des Kindprozesses reagieren würde. Wichtig: Zum Zombie wird nicht der Vaterprozess, sondern der bereits beendete Kindprozess.

Meilenstein 3: Das Programm von Aufgabe 5.1 funktioniert so wie beschrieben.

In der folgenden Grafik ist der Prozess mit der PID 1699 ein Zombie. Erkennen lässt sich dies an der Kennzeichnung `<defunct>` :



Aufgabe 5.2: Programmieren Sie einen Zombie, bei dem Sie genau das Verhalten der obigen Abbildung nachvollziehen können.

6. EXEC-FUNKTIONEN

In Aufgabe 4 wurde erwähnt, dass in unixoiden Systemen alle Prozesse mit einem `fork`-Aufruf generiert werden. Aber wie werden dann neue Programme geladen und ausgeführt? `fork` erzeugt ja jedes Mal wieder eine Kopie des Vaterprozesses. Die Antwort liefert die Familie der `exec`-Funktionen. Jede der `exec`-Funktionen geht grundsätzlich gleich vor: Sie überschreibt den aktuellen Programmcode des Prozesses durch einen neuen Programmcode. Die Familie der `exec`-Funktionen umfasst folgende Mitglieder:

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

Die Namen der `exec`-Funktionen starten alle mit `exec`. Danach kommen Buchstaben, die beschreiben, welche Parameter erwartet werden:

e	Es werden Umgebungsparameter (***e**nvironment*) erwartet
l	Es wird eine **L**iste von Kommandozeilenargumenten erwartet
v	Es wird ein **V**ektor mit Kommandozeilenparametern erwartet.
p	Die angegebene Datei wird in allen Verzeichnissen der Umgebungsvariable **P**ATH gesucht.

Folgender Quellcode startet das Programm `ps` mit dem Parameter `xlf`. Hierfür wird der Befehl `execlp` verwendet. Der erste Parameter („`ps`“) ist der Dateiname des Programms, welches gestartet werden soll. Ab dem zweiten Parameter wird eine Liste von Parametern analog zum `argv`-Array übergeben. Der erste Eintrag in der Liste ist der Name des Programms. Danach folgen die Parameter für das Programm. Abgeschlossen wird die Liste immer von einem `NULL`.

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

int main( void ) {
   int status;
   int procId;
   procId = fork();
   switch(procId){
      case -1:    //es ist ein Fehler aufgetreten
         break;
      case 0:     //Kindprozess
         execlp("ps","ps","xlf",NULL);
         break;
      default:    //Vaterprozess
         wait(&status);
         printf("CTRL+C zum Beenden druecken\n");
         while(1);
         break;
   }

   return EXIT_SUCCESS;
}
```

Aufgabe 6.1: Versuchen Sie nachzuvollziehen, welche PID der Prozess erhält, der von `execlp` gestartet wird. Hierzu können Sie das obige Beispiel so modifizieren, dass der Kindprozess zunächst seine PID ausgibt und erst dann `execlp` aufruft. Warum erhält der Prozess die identifizierte PID?