Praktikum Betriebssysteme – Übung 4

Die folgenden Aufgaben behandeln die Prozesserzeugung auf UNIXoiden Betriebssystemen.

Aufgabe 1 – Verständnisfragen

a) Prozesse werden mit Hilfe der Bibliotheksfunktion `exit(result)` beendet, der ein Parameter (`result`) übergeben wird. Wo wird dieses Resultat gespeichert und wie kann nach dem Ende des Prozesses noch darauf zugegriffen werden?

b) Nennen Sie einige Komponenten des Prozesskontrollblocks und beschreiben Sie stichwortartig, warum die Komponenten vorhanden sind.

Aufgabe 2 – Prozesse unter GNU/Linux

a) Sehen Sie sich die Prozess-Liste (alle aktiven bzw. wartenden Programme) mit dem Befehl `ps` an. Was bewirken die Optionen `a` und `x`? Was bedeuten die Werte in der Spalte `STAT`? (→ `man ps`)

b) Testen Sie nun einmal das Programm `pstree`, das Ihnen alle laufenden Prozesse in Form eines Baums anzeigt. Sie sehen somit also die Eltern-Kind-Beziehung der Prozesse. Finden Sie ihren eigenen `pstree`-Aufruf in der Ausgabe von `pstree`? Welcher Prozess ist der Eltern-Prozess von `pstree`?

c) Mit den Programmen `kill` bzw. `pkill` können Sie nach Prozessen suchen und diesen *Signale* senden (Signals sehen wir uns in einigen Wochen noch genauer an). Versuchen Sie den Prozess ihrer Shell (`bash`) zu finden und diesen zu beenden (Siganl Nummer 9 (SIGTERM) bzw. 15 (SIGKILL)).

Hinweis: Es existieren natürlich auch verschiedene interaktive Tools, wie `htop`, `btop` oder `glances` im Terminal oder diverse grafische Prozessmanager um die Systemauslastung und Prozessinformationen anzuzeigen.

Aufgabe 3 – fork() Grundlagen

Beschäftigen Sie sich in dieser Aufgabe mit den in der Vorlesung besprochenen Konzepten für die Erzeugung und Kontrolle von Prozessen (`fork()`, `wait()`, `getpid()`, usw.). Beachten Sie: Mit `fork()` wird lediglich eine Kopie des aufrufenden Prozesses erzeugt.

a) Übersetzen Sie das Beispielprogramm (→ gitlab) und führen Sie es aus. Das Programm ruft `fork()` auf und gibt anschließend im Eltern- und im Kindprozess jeweils die eigene Prozess-ID und die des Elternprozesses aus.

Hinweis für die weiteren Aufgaben: Lassen Sie in einem anderen Terminal das Kommando „`watch -n0.1 ps a`“ (ohne die „“) laufen, um kontinuierlich Ihre eigenen Prozesse und deren Status anzuzeigen (`watch -n0.1` führt das danach folgende Kommando, `ps a`, alle 0.1 Sekunden neu aus). Mehr in der Manpage (`man ps`).

b) Erweitern Sie das Programm. Nutzen Sie `sleep()` um den Kindprozess für einige Sekunden schlafen zu lassen, bevor er erneut seine eigene Prozess-ID und die des Elternprozesses ausgibt. Was passiert, falls der Elternprozess bereits beendet wurde (er also kein `wait()` aufruft), während der Kindprozess noch läuft? Welche Prozess-ID gibt wird in dem Fall für den Elternprozesses ausgegeben?

c) Lassen Sie nun den Elternprozess schlafen und den Kindprozess enden. Was passiert, wenn das Kind beendet wird, der Eltern-Prozess aber noch nicht `wait()` aufgerufen hat? Was passiert, wenn nie `wait()` aufgerufen wird?

Hinweis: Eltern- und Kindprozess sind unabhängig voneinander. In beiden Prozessen läuft zwar der gleiche Programmcode, jedoch besitzen die Prozesse unterschiedliche Adressräume. In der folgenden Aufgabe sollte das deutlich werden.

d) Initialisieren Sie vor dem `fork()`-Aufruf eine globale und eine lokale Variable und geben Sie deren Speicheradresse nach dem Aufruf von `fork()` aus. Nutzen die beiden Prozesse die selbe Variable im Speicher? Was passiert, falls beide Prozesse die Variable verändern?

e) Starten Sie im Hauptprogramm (Elternprozess) mehrere Kindprozesse und lassen Sie jeden der Kindprozesse in einer Schleife mehrfach etwas aufwändiges auf der CPU tun (z.B. eine Fibonaccizahl berechnen). Geben Sie in jedem Schleifendurchlauf die Prozess-ID und den Index aus (Beispiel: „Prozess 382766 Durchlauf 3“). Wenn Sie das Programm mehrfach laufen lassen, werden Sie feststellen, dass die Reihenfolge der Ausgabe nicht deterministisch ist.

Es gibt viele weitere Dinge, die Sie ausprobieren können.

- *Ein Elternprozess erzeugt immer neue Kindprozesse*
- *Ein Elternprozess erzeugt einen Kindprozess, der Kindprozess wiederum erzeugt einen weiteren Kindprozess, usw.*
- *Wie viele Prozesse können Sie erzeugen bevor die Prozess-IDs ausgehen? Wird das System durch zu viele Prozesse langsamer? Stürzt es irgendwann ab?*

Aufgabe 4 – exec

In Aufgabe 2 haben wir neue Prozesse erzeugt, jedoch war der neue Prozess lediglich eine identische Kopie des Elternprozesses. In der Regel möchte man stattdessen ein anderes Programm im Kindprozess ausführen. Dafür stehen die verschiedenen `exec..()` Funktionen zur Verfügung, die den aktuellen Prozess durch ein anders Programm ersetzen. Die unterschiedlichen Arten von `exec..()` bieten unterschiedliche Möglichkeiten Argumente zu übergeben. In der Manpage erfahren Sie, worin sich die verschiedenen Varianten unterscheiden (→ `man 3 exec`).

Für die Übung bietet sich die Nutzung von `execv()` (`int execv(const char *path, char *const argv[]);`) an. Diese erhält als erstes Argument den vollständigen Namen des auszuführenden Programms und als zweites Argument einen Parameter ähnlich wie `*argv[]`, den Sie aus der `main()`-Funktion kennen. Es handelt sich also um einen Zeiger auf ein Array, in dem jeweils „Strings“ (char*) abgespeichert sind. Das letzte Argument muss NULL sein, um das Ende des Array anzuzeigen.

a) Schreiben Sie ein Programm, das `exec..()` nutzt um sich selbst mit dem Programm `ls` (`/usr/bin/ls`) zu überlagern und so den Inhalt des aktuellen Verzeichnis auszugeben.

b) Schreiben Sie zwei kleine C-Programme, die jeweils Parameter beim Aufruf erhalten. Was die beiden Programme tun sollen ist im folgenden beschrieben.

Hinweise zu Parametern:

- Allgemeiner Aufruf von Programmen: `./programm parameter1 parameter2 ...`
- Referenzen auf diese Parameter werden in `argv[]` gespeichert
- `argc` zeigt Ihnen die Anzahl der übergebenen Parameter an
- Der erste Parameter `argv[0]` ist der Programmname des laufenden Programms selbst

Programm 1

Dem Programm werden bei Aufruf beliebig viele Parameter übergeben. Der erste dieser Parameter soll der Name von Programm 2 sein. Alle weiteren Parameter können beliebige Werte besitzen.

Beispiel: `./programm1 programm2 foo bar baz`

Nach dem Start gibt Programm 1 den eigenen Namen aus und ruft dann Programm 2 (mittels des übergebenen Parameter) auf. Dem Programm 2 werden dabei alle übrigen Parameter (im Beispiel also `foo`, `bar` und `baz`) übergeben.

Programm 2

Das Programm soll seinen eigenen Namen ausgeben. Anschließend sollen alle weiteren Parameter (die beim Aufruf durch Programm 1 übergeben wurden) ausgegeben werden.

Hinweis: Auch `./programm1 ps ax` sollte somit funktionieren und die gleiche Ausgabe liefern, die `px ax` auch selbst erzeugen würde.

Aufgabe 5 – posix_spawn()

Die Entscheidung neue Prozesse mittels fork-exec (Erzeugen einer Kopie des aktuellen Prozesses und anschließend Überlagerung des Kindprozesses mit einem anderen Programm) zu erzeugen stammt aus den 70er Jahren und ist heute – obwohl sie in gewissen Situationen Vorteile bietet – zurecht umstritten. Seit den 90er Jahren bietet der POSIX-Standard mit `posix_spawn()` eine Alternative, die fork und exec kombiniert.

Versuchen Sie Aufgabe 4 mittels `posix_spawn()` zu lösen.

Hinweis: Unter Linux verwendet `posix_spawn()` genau wie `fork()` und `pthread_create()` den äußerst flexiblen Systemaufruf `clone()` um neue Prozesse und Threads („Tasks“) zu erzeugen (→ `man 2 clone`).