

Netzwerk - Programmierung

Prozesse

Jan Krueger

jkrueger@cebitec.uni-bielefeld.de

Alexander Sczyrba

asczyrba@cebitec.uni-bielefeld.de

Übersicht

- Prozesse
- *fork()*
- Parents und Children
- *system()* und *exec()*

Prozesse

Moderne Betriebssysteme (UNIX-artige, Windows ab NT/2000) sind Multitasking-fähig:

- Mehrere Programme können simultan ablaufen
- Jedes in einem separaten Prozess
- Das OS bestimmt den Wechsel zwischen den einzelnen Prozessen

Multitasking in Python

Python unterstützt zwei Arten von Multitasking:

1. Fork

- Basiert auf traditionellem UNIX Multiprozess-Modell
- Erlaubt dem aktuellen Prozess, sich selbst zu klonen
- Ergebnis: Zwei in fast jeder Hinsicht identische Prozesse

2. Multithreading

- Modernes Konzept eines Threads
- Aufgaben werden in einem einzigen Prozess gehalten
- Einzelnes Programm kann aus nebeneinander ausgeführten Threads bestehen, von denen jeder unabhängig von den anderen läuft → Nebenläufigkeit

`fork()`

- Jedem Prozess im System wird eine eindeutige pos. Zahl zugeordnet: Prozess-ID oder PID
 - Funktion *`fork()`* ist in allen *NIX Versionen von Python verfügbar
 - Erzeugt ein exaktes Duplikat des aktuellen Prozesses, nach dem `fork`-Aufruf existieren parent und child gleichzeitig
- child-Prozess übernimmt:
 - Aktuelle Werte aller Variablen
 - Dateihandles (inkl. Daten in I/O-Puffern)
 - Weitere Datenstrukturen
- Voraussetzung: Parent und Child wissen, wer von ihnen wer ist

Aufgabe

- Ermittle, wie viele Prozesse unter Deiner *userid* laufen. Verwende dazu den Befehl `ps` mit den entsprechenden Optionen (siehe `man ps`).

Welche Informationen liefert `ps`?

- Der Befehl `p[s]tree*` gibt Informationen über Prozesse, ihre Kinder und Eltern in Baumstruktur aus.

Welcher Prozess ist der Parent aller Prozesse?

* *Solaris/BSD – ptree, Linux/OSX – pstree* → Unter OSX nicht Teil des Systems

`fork()`

```
import os  
newpid = os.fork()
```

- Verzweigt in einen neuen Prozess
- Rückgabewert
 - Im Parent-Prozess: PID des Child-Prozesses
 - Im Child-Prozess: *0*
 - Im Falle eines Fehlers: negativer Wert
- Vgl. `> pydoc os.fork`

Hilfreiche Funktionen

- `os.getppid()`
 - Gibt PID des Parent-Prozesses zurück.
- `os.getpid()`
 - Enthält die PID des aktuellen Prozesses.
- `os.wait()`
 - Wartet auf die Terminierung eines Child Prozesses und gibt die PID samt Exit-Status des terminierten Prozesses zurück.
- `os.waitpid()`
 - Wartet auf die Terminierung eines **bestimmten** Child Prozesses und gibt die PID samt Status des terminierten Prozesses zurück.

Beispiel `fork()`

#Python3 Code - Ersetzt `input()` durch `raw_input()` für Python2.X

```
import os
```

```
def child():  
    print('I am a Child with PID: ' + str(os.getpid()) )  
    os._exit(0)
```

```
def parent():  
    while 1:  
        newpid = os.fork()  
        if newpid == 0:  
            child()  
        else:  
            pids = (os.getpid(), newpid)  
            print("Parent: %d with Child: %d" % pids)  
            if input() == 'q': break
```

Zombies

- Terminiert der Parent vor dem Child, wird dieser zum Waisenkind und vom init-Prozess adoptiert
- Ein Prozess, der terminiert ist, dessen Parent aber nicht darauf gewartet hat, wird Zombie genannt

Aufgabe

- Ändere das *fork()*-Beispiel um, so dass
 1. das Child vom init-Prozess übernommen wird.
 2. das Child zu einem Zombie wird.
- Benutze `top`, `ps`, `p[s]tree` um das Verhalten der Skripte zu analysieren

`system()` und `exec()`

- `os.system()`
 - Führt ein anderes Programm als Unterprozess aus
 - Wartet auf dessen Beendigung
 - Rückgabewert: exit-Status
- `os.exec[l|v]p()`
 - Ersetzt den aktuellen Prozess durch angegeben Befehl
 - Bei Erfolg kehrt der Aufruf nie zurück
 - Neuer Prozess hat die gleiche PID wie alter Prozess
 - Verwendet dieselben STDIN-, STDOUT-, STDERR-Dateihandles
 - Offene Handles werden **nicht geflusht!**

Aufgabe

Schreibe ein Programm, das alle 5 Sekunden die aktuelle Uhrzeit ausgibt. Verwende dazu:

- `os.exec[1|v]p()`
- `time.sleep()`
- `> date`