FH | JOANNEUM
University of Applied Sciences

Systemnahe Programmierung
SS 2024

Unit 6

Prozesse

Was ist ein Prozess?

Ein Prozess besteht aus einem eigenen virtuellen Adressraum (text, bss, heap, stack) und einer im Kernel verwalteten Datenstruktur.

Der Betriebssystemkernel verwaltet alle relevanten Daten für einen Prozess in einem *Prozesskontrollblock* :

- Prozessnummer von Prozess (pid) und Elternprozess (ppid)
- Gruppen- und Benutzer-ID (gid, uid)
- Effektive Benutzer- und Gruppen-ID (egid, euid)
- Prozesspriorität (pri)
- Angeforderte Priorität (nice)
- Physikalische Adresse im Speicher
- Zustand des Prozesses
- Terminal von dem der Prozess gestartet wurde (controlling terminal, ctty)
- Statistiken wie verbrauchte Rechenzeit des Prozesses (time) usw.
- Programm mit dem der Prozess gestartet wurde (cmd)
- Aktuelles Arbeitsverzeichnis

Prozesse

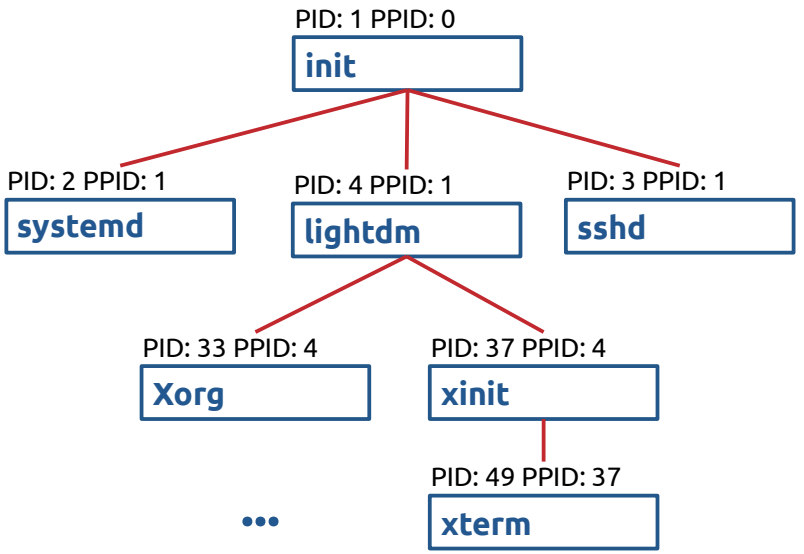
Prozessstatus

Ein Prozess kann verschiedene Stati annehmen:

- Running ... der Prozess wird ausgeführt
- Runnable ... der Prozess ist zur Ausführung bereit
- Sleeping ... wartet auf Ressourcen oder sleep()
- Stopped ... der Prozess wurde angehalten, wartet auf Fortsetzung (Crtl+Z)
- Zombie ... der Prozess ist beendet, aber noch nicht aus der Prozesstabelle entfernt (Parent hat noch nicht `wait()` aufgerufen).

Prozesse

Prozesshierachie



Prozesse bilden ein Hierarchie, neue Prozesse werden durch bereits existierende Prozesse erzeugt. Es besteht eine Eltern-Kind Beziehung (parent - child). Die Beziehung wird durch die `ppid` (parent-process-id) festgehalten.

Prozesse

PID und PPID

Die PID und PPID eines Prozesses können mit diesen Systemcalls ausgelesen werden:

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

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

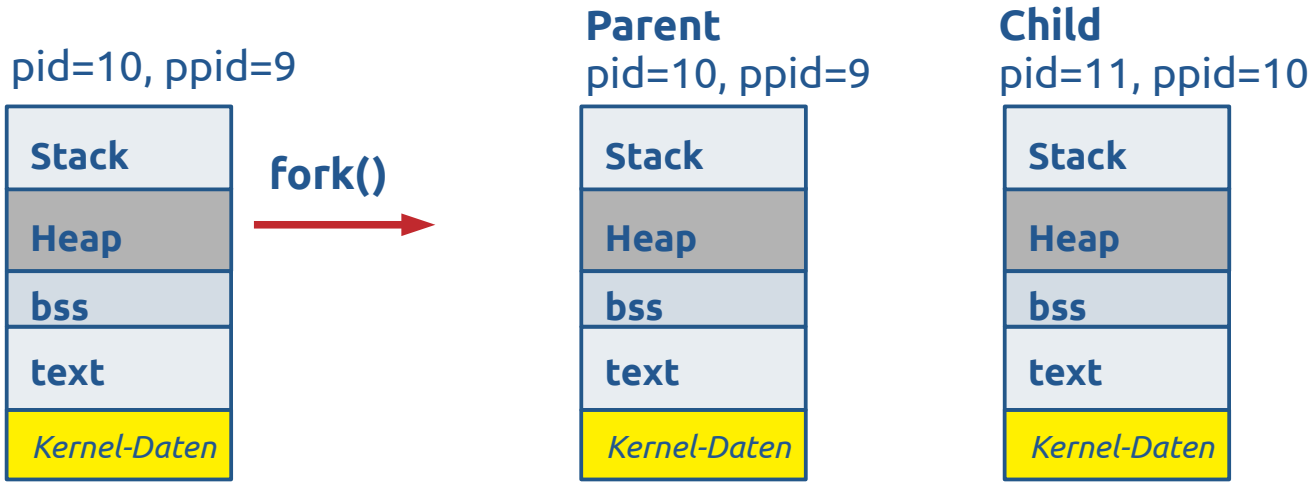
„These functions are always successful.“

Prozesse

Einen neuen Prozess erzeugen

Ein neuer Prozess wird erzeugt, indem das Betriebssystem eine (fast) exakte Kopie des erzeugenden Prozesses anlegt und startet.

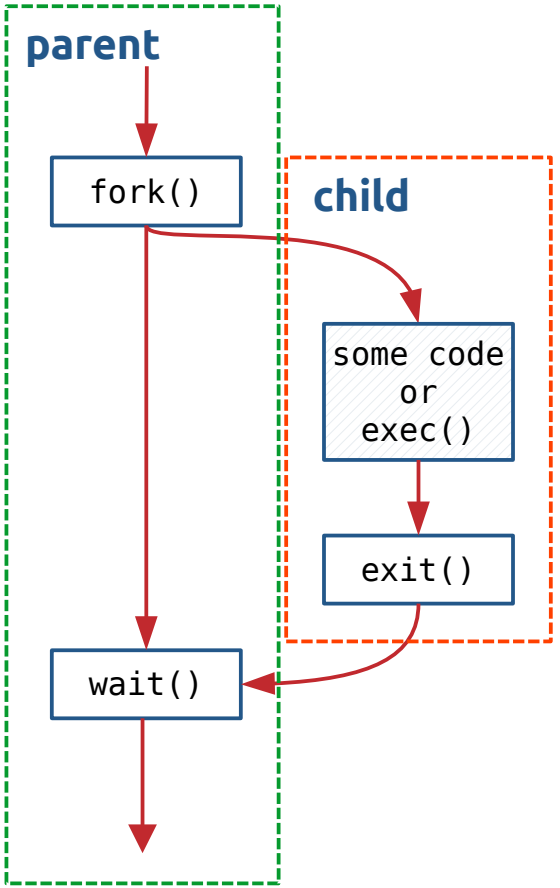
Ruft ein Prozess die Systemfunktion `fork()` auf, so wird ein Klon (child) erzeugt, die PPID wird auf die PID des erzeugenden Prozesses (parent) gesetzt.
Der Parent-Prozess spaltet sich quasi auf ...



Prozesse

Prozesse erzeugen - Ablauf

- `fork()` erzeugt einen neuen Prozess
- `exec()` ersetzt das Programm eines Prozesses durch ein Neues (optional)
- `exit()` beendet den Prozess
- `wait()` wartet auf die Beendigung eines Kindprozesses



Prozesse

Prozesse erzeugen

Systemcall:

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

pid_t fork(void);
```

`fork()` erzeugt eine Kopie des Elternprozesses.
Heap, Stack, Textsegment werden kopiert.
Die Dateideskriptoren der geöffneten Dateien werden übernommen.
Nicht vererbt werden: PID, PPID, Statistiken, offene Signale, Dateisperren

Der Rückgabewert von `fork()` ist die PID des erzeugten Prozesses und wird dazu verwendet zu unterscheiden, um welchen Prozess es sich nach dem Aufruf handelt:

- **<0** → Ein Fehler ist aufgetreten
- **0** → im Kindprozess (child)
- **>0** → im Elternprozess (parent). Der Rückgabewert ist die PID des childs.

Prozesse

Prozesse erzeugen

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

int main(int argc, char *argv[]) {
  pid_t childPid;
  childPid = fork();
  /* -> parent und child machen nach fork() hier weiter! Einziger
        Unterschied ist der Wert der Variable childPid! */
  if (childPid < 0) { → Fehler bei fork()
    printf("failed to fork..\n");
  }
  else if (childPid == 0) { → Dieser Zweig wird im Kindprozess ausgeführt
    /* child code ...*/
  }
  else { → Dieser Zweig wird im Elternprozess ausgeführt
    /* parent code, childPid ist childs pid ...*/
  }
  return 0;
}
```

Prozesse

Übung 1 - VPL

Schreiben sie ein Programm, das einen neuen Prozess mit `fork()` erzeugt.
Geben sie die PID, PPID und den pid Wert aus `fork()` im jeweiligen Prozess mit `printf()` aus.
Im Elternprozess: pid (Returnvalue) aus `fork()`, eigene PID und PPID,
Im Kindprozess: pid (Returnvalue) aus `fork()`, eigene PID und PPID

Verwenden sie `getpid()` und `getppid()`

Ausgabeformat:

```
Elternprozess pid: 44982 PPID: 2407 PID: 44981
Kindprozess pid: 0 PPID: 44981 PID: 44982
```

Prozesse

Prozesse beenden

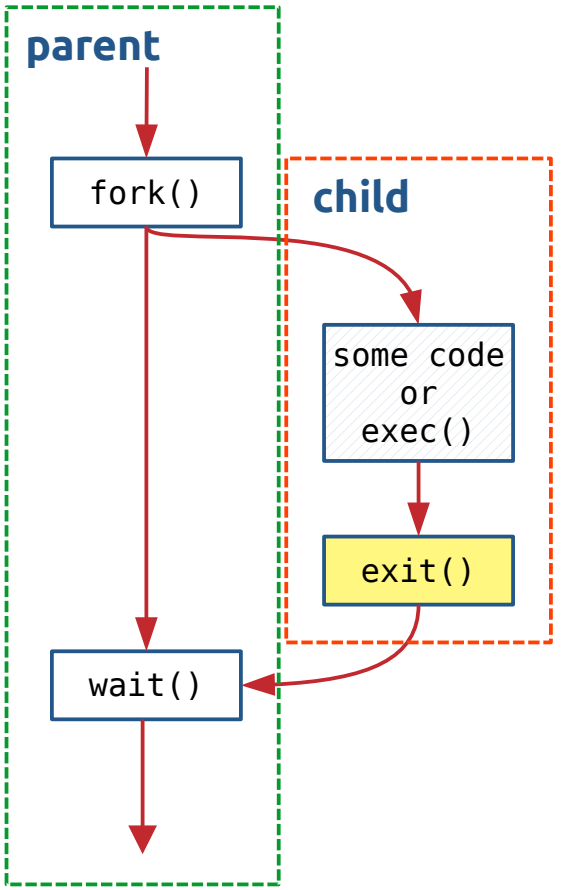
Prozess beenden:

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

Status:

```
EXIT_SUCCESS
EXIT_FAILURE
```

oder jeder beliebige `int` Wert, aber nur 1 Byte davon wird übergeben (0-255).



`exit()`

- leert die `stdio` Buffer (wie `fflush()`)
- schließt alle offenen Dateideskriptoren
- löscht temporäre Dateien
- ruft Exithandler auf
- Der Prozess geht in den Zombie-Zustand über, dies ermöglicht es dem Elternprozess auf die Beendigung des Kindprozesses zu reagieren

Prozesse

Zombies und Orphans

Zombie

Der Kindprozess ist beendet und der Elternprozess hat noch kein `wait()` durchgeführt.

- Der Kindprozess wird auf den Zustand Zombie gesetzt.
- Der Eintrag in der Prozesstabelle bleibt erhalten bis der Elternprozess `wait()` ausführt.

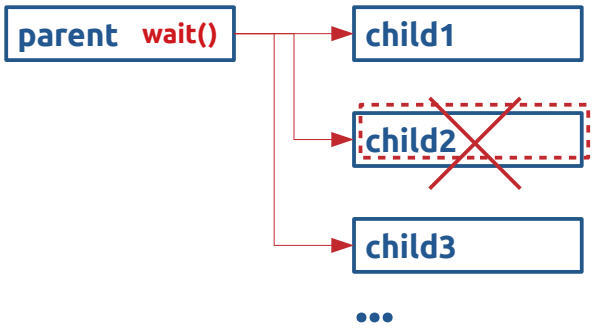
Orphan

Der Elternprozess existiert nicht mehr (beendet) und der Kindprozess läuft noch.

- Der Kindprozess wird eine „Waise“ und dem Init-Prozess zugeordnet (Reparenting, PPID=1)
- Wenn der Kindprozess terminiert, dann entfernt der Init-Prozess den Eintrag aus der Prozesstabelle.

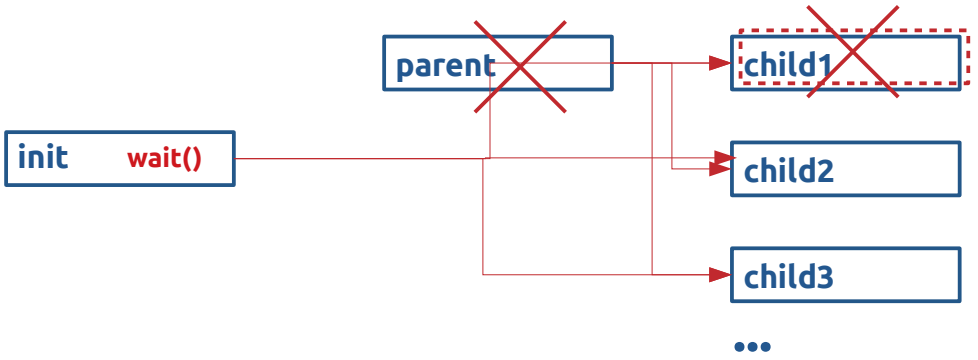
Prozesse

Zombies



Prozesse

Orphans



Prozesse

Übung 1a

Erweitern sie das Programm aus der letzten Übung um damit einen Zombie-Prozess zu erzeugen.

Lassen sie den Parent-Prozess lange genug am „Leben“ indem sie die Funktion `sleep()` verwenden, damit sie mit dem `ps` Befehl den Status des Kind-Prozesses überprüfen können.

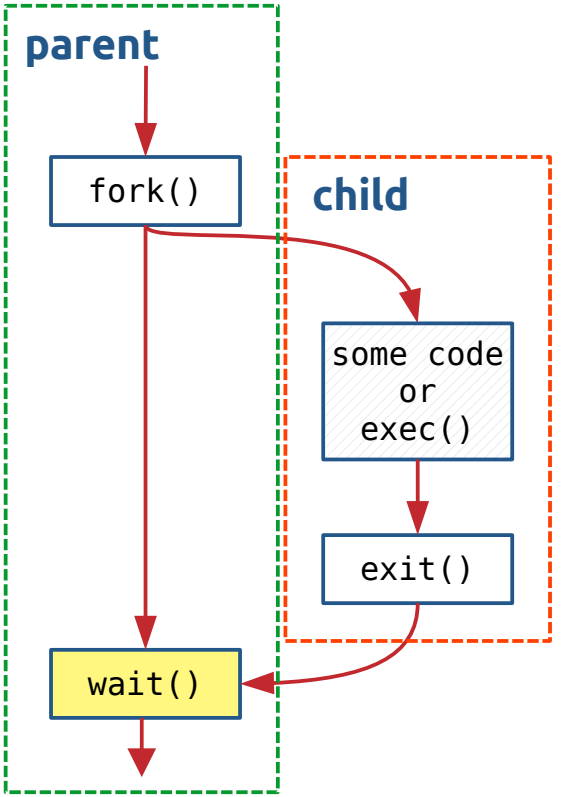
Wie ist der Prozess im `ps` Listing gekennzeichnet ?

Prozesse

Warten auf Kindprozesse

Der Elternprozess sollte immer auf das Beenden der Kindprozesse warten. Damit ist sichergestellt, dass Ressourcen nicht länger als notwendig reserviert werden.

Die Systemaufrufe `waitpid()` und `wait()` warten auf Kindprozesse.



- Verbleibende Ressourcen des Zombies werden aufgeräumt
- Die PID wird als frei markiert
- Der Prozesskontrollblock wird freigegeben
- Falls aktuell kein Kindprozess im Zombie-Zustand ist, wartet `wait()` bis zum Terminieren des nächsten Kindprozesses und räumt diesen dann ab.

Prozesse

Warten auf Kindprozesse

Die Funktion `wait()` wartet, bis sich ein Kindprozess beendet, und gibt dessen PID als Rückgabewert zurück. Bei Fehler wird -1 zurückgegeben.

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
```

In der Statusvariable wird ein Status übergeben, der mit den Makros aus `<sys/wait.h>` ausgewertet werden kann.

Er besteht aus dem `exit()` Status des Programms (nur 1 Byte wird ausgewertet) und einem Wert, der die Ursache des Beendens beschreibt.

Prozesse

Makros für wait() Status:

Makro	Beschreibung
WIFEXITED(status) WEXITSTATUS(status)	Ist TRUE, wenn der Child-Prozess normal beendet wurde. Mit WEXITSTATUS() kann der Exitstatus des Child-Prozesses abgefragt werden (0-255).
WIFSIGNALED(status) WTERMSIG(status) WCOREDUMP(status)	WIFSIGNALED ist TRUE, wenn der Child-Prozess durch ein Signal terminiert wurde, das dieser nicht abgefangen hat. Mit WTERMSIG kann die Nummer des Signals ermittelt werden, das den Prozessabbruch bewirkt hat. WCOREDUMP ist TRUE, wenn durch das Signal, das den Prozess terminiert hat, ein Coredump angelegt wurde.
WIFSTOPPED(status) WSTOPSIG(status)	WIFSTOPPED liefert TRUE, wenn ein Child-Prozess angehalten wurde. WSTOPSIG liefert die Nummer des Signals, das den Prozess gestoppt hat.

Prozesse

Warten auf einen speziellen Kindprozess

Die Funktion `waitpid()` wartet auf einen bestimmten Kindprozess

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Der Parameter `pid` kann folgende Werte annehmen:

- < -1 auf einen Kindprozess warten, dessen Prozess-Gruppen ID gleich dem Absolutwert von pid ist.
- -1 auf einen beliebigen Kindprozess warten.
- 0 auf einen beliebigen Kindprozess warten dessen Prozess-Gruppen ID gleich der des aufrufenden Prozesses ist.
- >0 auf den Kindprozess warten, dessen PID == pid ist.

Für die Auswertung von `status` können die gleichen Makros verwendet werden, wie bei `wait()`.

Prozesse

Warten auf einen speziellen Kindprozess

Für `waitpid()` können im Parameter `options` folgende Optionen angegeben werden:

- WNOHAN
  `waitpid()` kehrt sofort zurück, falls kein passender Zombie-Prozess vorhanden ist. Ist geeignet um beendete Prozesse periodisch abzufragen (Polling)
- WUNTRACED
  `waitpid()` kehrt auch zurück, wenn der Kindprozess gestoppt wurde.
- WCONTINUED
  `waitpid()` kehrt auch zurück, wenn ein Kindprozess fortgesetzt wurde.

Mehrere Optionen können mit bitweisem ODER verknüpft werden

Prozesse

Beispiel - wait()

```
int status;
pid_t childPid, pid;

childPid = fork();
if(childPid == -1){
  perror("fork");
  exit(EXIT_FAILURE);
}
else if ( childPid == 0 ) {  // child
  /* Do something here ... */
}
else { // parent
  pid = wait(&status);
  if(WIFEXITED(status)){
    int retval = WEXITSTATUS(status);
    printf("parent: child %d finished, Exitstatus: %d\n", pid, retval);
  }
}
...
```

Prozesse

Übung 2 - VPL

Erweitern sie das Programm aus der letzten Übung (Zombie), indem der Elternprozess auf die Beendigung des Kindprozesses wartet.
Beenden sie den Kindprozess mit dem Status 47.

Geben sie im Elternprozess die pid aus `wait()` und den Exitstatus des Kindprozesses aus.

Ausgabeformat:

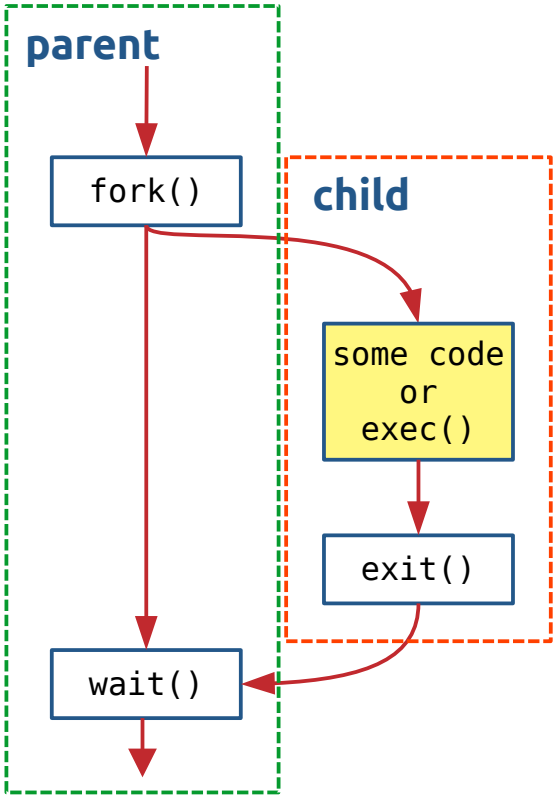
```
Waited for: <pid>, Exitstatus: 47
```

Prozesse

Ein anderes Programm ausführen

Mit `exec()` kann ein anderes Executable im aktuellen Prozess ausgeführt werden.

- Das aktuell ausgeführte Programm wird ersetzt. (Text, Daten, Stack und Heap Segmente)
- Erhalten bleiben die PID, Dateideskriptoren, Arbeitsverzeichnis ...
- `exec()` kehrt nur im Fehlerfall zurück



Prozesse

exec()

Der `exec()` Sytemcall existiert in mehreren Varianten:

```
#include <unistd.h>

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 execvpe(const char *file, char *const argv[], char *const envp[]);
```

Ersetzt das aktuelle Executable durch das im Parameter `path/file` spezifizierte Programm und startet dieses.

Prozesse

exec() - PATH

1) Parameter `'file'`: Wenn der Parameter `file` kein `'/'` enthält, dann wird die Umgebungsvariable `PATH` ausgewertet.
`PATH` enthält eine Liste von durch `':'` getrennten Verzeichnissen, in denen nach ausführbaren Programmen gesucht wird.
Beispiel:

```
echo $PATH ->
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

Betrifft: `execlp, execvp, execvpe`
Beispiel:

```
execlp("ls", "ls","-ltr",NULL);  -> findet /bin/ls
```

2) Parameter `'path'`: Absoluter Pfad, keine Auswertung der `PATH` Umgebungsvariable

Betrifft: `execl, execle, execv`
Beispiel:

```
execl("/bin/ls", "/bin/ls", "-ltr", NULL);
```

Prozesse

exec()

Dem Programm müssen die Commandline-Argumente übergeben werden.
Im Falle von `execl*()` werden die Argumente als einzelne Strings übergeben (variable Anzahl von Argumenten).
Die Funktionen vom Typ `execv*()` nehmen die Argumente in Form eines Arrays von Strings entgegen.

In beiden Fällen gilt:

- Das 1. Argument ist der Programmname (`argv[0]`)
- Die Argumentliste muss mit einem NULL-Zeiger enden

Die Varianten `execle und execvpe` erlauben es, das Environment für den Prozess zu spezifizieren (als Liste von Strings der Form VARIABLE=WERT).
Auch hier muss der letzte Wert in der Liste ein NULL-Zeiger sein.

Im Fehlerfall wird ein Wert von -1 zurückgegeben und `errno` gesetzt.

Prozesse

Beispiel

```
int main(int argc, char* argv){
  int status;
  pid_t childPid, pid;
  char *childArgs[] = {"ls", "-l",NULL};
  char *cmd = "ls";

  childPid = fork();
  if(childPid == -1) {
    perror("fork");
    exit(EXIT_FAILURE);
  }
  else if (childPid == 0) {  // child
    if (execvp(cmd, childArgs) < 0 ) {
      perror(cmd);
      exit(EXIT_FAILURE);
    }
  }
  else { // parent
    pid = wait(&status);
    if(WIFEXITED(status)){
      int retval = WEXITSTATUS(status);
      printf("parent: child %d finished, Exitstatus: %d\n", pid, retval);
    }
  }
}
```

Prozesse

Übung 3

Schreiben sie ein Programm `worker`, das eine Zufallszahl zwischen 0 und 100 erzeugt und diese als Exitstatus zurückgibt.
Verwenden sie zur Erzeugung der Zufallszahlen die Funktionen `srand()` und `rand()`.

Das Programm `worker` erwartet eine Integer-Zahl als Argument und initialisiert den Zufallszahlgenerator mit der Summe aus dem Argument und seiner PID.

Der `worker` soll mit `printf()` das Argument und die erzeugte Zufallszahl auf `stdout` ausgeben.

Prozesse

Übung 3

Schreiben sie ein Hauptprogramm `parent`, das 10 Child-Prozesse erzeugt (Schleife) und in jedem Child das Programm `worker` ausführt.
Übergeben sie `worker` als Argument eine Zahl (Schleifenindex) zwischen 0 und 9.

Beachten sie, dass der Pfad zum Programm `worker` vollständig ist.

Das Hauptprogramm wartet auf die Child-Prozesse und gibt die PID und Zufallszahl jedes childs (= Exit-Status) in folgender Form auf `stdout` aus:

```
<childpid>: <zufallszahl>
```

Geben sie abschließend die größte gefundene Zahl aus.

Achten sie darauf, dass die Ausgaben der Kindprozesse von der des Hauptprogramms (`parent`) unterschieden werden können (z.B.: unterschiedliches Präfix).

Prozesse

Übung 3

Die Child-Prozesse in einem Loop erzeugen.