Inhaltsverzeichnis

1 UNIX-Prozesskonzept

UNIX unterstützt Multitasking, in einigen Varianten auch Multithreading (z.B. Solaris). UNIX vergibt für jeden Prozess bei der Erzeugung eine eindeutige Prozessidentifikationsnummer (PID). Die Prozesse sind in einer strengen Eltern-Kind-Hierarchie angeordnet, alle Prozesse stammen direkt oder indirekt vom ersten Prozess mit der PID 1 (auch *init*-Prozess) ab.

Prozesserzeugung und Progammaufruf sind separate Operationen. Bei der Prozesserzeugung (*fork*) wird kein auszuführendes Programm angegeben. Stattdessen fertigt der Erzeuger einen „Klon“, also ein Duplikat von sich selbst an.

1.0.1 fork – Prozess erzeugen

```
#include <unistd.h>

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

Fork erzeugt einen neuen Prozess als *Subprozess* (auch *Kindprozess*). Der neue Prozess gleicht in fast jeder Hinsicht dem Erzeuger

- gleiches Programm
- identische Variablenwerte
- gleicher Programmzähler
- gleiche Dateien geöffnet

– gleiche sonstige Attribute (Eigentümer, Zugriffsrechte usw.)

Aus Sicht des Betriebssystems ist *fork()* eine Kopieroperation: Sowohl der Programmspeicher als auch der Prozesstabelleneintrag des Erzeugers werden kopiert.

Die wesentlichen Unterschiede zwischen Erzeuger und Subprozess:

- Der Subprozess hat eine neue PID
- der *fork()*-Resultatswert ist unterschiedlich:

$$fork - Resultat = \begin{cases} Subprozess: & 0 \\ Elternprozess: & Subprozess - PID \end{cases}$$

Bei Misserfolg (Hauptspeichermangel, Prozesstabelle voll) liefert *fork -1* als Resultat und die globale Variable *errno* enthält eine Codierung der Fehlerursache.

Es macht meist keinen Sinn, wenn Eltern- und Kindprozess sich identisch verhalten. Ein Subprozess wird i.d.R. erzeugt, um eine andere Aufgabe nebenläufig auszuführen. Dies muss durch eine explizite Fallunterscheidung anhand des unterschiedlichen *fork*-Resultats programmiert werden.

```
#include <ostream.h>
#include <unistd.h>
#include <stdio.h>

main(){
  pid_t kind_pid;  // ueblicherweise: pid_t = "unsigned long"
  pid_t meine_pid;

  printf("vor fork, PID=%d\n", meine_pid=getpid());
  switch (kind_pid=fork()) {
    case -1: perror("fork-Fehler");
             exit(1);
    case  0: printf("Kind: PID=%d\n", meine_pid=getpid());
             printf("Kind: Eltern-PID=%d\n", getppid());
             break;
    default: ("Elternprozess: nach fork, Kind-PID=%d\n", kind_pid);
  }

  printf("ich mache Feierabend: meine PID=\d\n", meine_pid);
  exit(0);
}
```

Man beachte, dass die Ausgabe hinter der *switch*-Anweisung nicht mehr vom *fork*-Resultat abhängig ist und damit von beiden Prozessen ausgeführt wird. Bei der Ausführung hat die Variable *meine_pid* in beiden Prozessen unterschiedliche Werte, jeder Prozess hat seine eigene Kopie der Variablen.

Die Funktion *getpid* liefert die PID des Aufrufers, *getppid* („get parent process identification“) die seines Elternprozesses.

Die *perror*-Funktion gibt eine zum aktuellen *errno*-Wert zugeodnete Fehlermeldung aus (zusätzlich zu der Argument-Zeichenkette).

Die Ausgabe des Programms könnte so aussehen:

```
vor fork, PID=453
Kind: PID=454
Kind: Elternprozess-PID=453
ich mache Feierabend: meine PID=454
Elternprozess: nach fork, Kind-PID=454
ich mache Feierabend: meine PID=453
```

Natürlich sind andere Abfolgen möglich, die Bearbeitung ist schliesslich nebenläufig. Auch folgende Ausgabe ist denkbar:

```
vor fork, PID=453
Elternprozess: nach fork, Kind-PID=454
ich mache Feierabend: meine PID=453
Kind: PID=454
Kind: Elternprozess-PID=1
ich mache Feierabend: meine PID=454
```

Der Elternprozess bekommt den Prozessor zuerst, macht seine Ausgaben und terminiert. Der Kindprozess ist zum Waisenkind geworden. Waisenkinder werden vom Prozess 1 adoptiert. Deswegen gibt Prozess 454 als Elternprozess den *init*-Prozess 1 an.

vfork

Eine zweite Prozesserzeugungsoperation, *vfork*, ist für den häufig vorkommenden Fall gedacht, dass der Subprozess ein anderes Programm ausführen soll. Der Aufwand, den *fork* durch Kopieren des Programmspeichers verursacht, ist in diesem Fall nicht gerechtfertigt. Der Subprozess wechselt unmittelbar nach dem Erzeugen der Programmkopie in ein anderes Programm.

Der *vfork*-Aufruf verzichtet stattdessen auf das Kopieren. Eltern- und Kindprozess benutzen den gleichen Programmspeicher. Damit keine Wettbewerbsbedingungen auftreten, wird der Elternprozess solange blockiert, bis der Subprozess entweder ein anderes Programm ausführt oder terminiert.

1.0.2 exec – Programmaufruf

Mit den *exec*...-Aufrufen wird ein neues Programm ausgeführt. Im ersten Argument wird der Programmpfad angegeben, danach die dem Programm zu übergebenden Argumente.

Neben den Argumenten erhält das Programm auch eine „Umgebung“ (Environment), die aus beliebig vielen Definitionen der Form *NAME=Wert* besteht. Die so definierten Variablen heissen **Umgebungsvariablen**. Auf diese Umgebung kann das Programm mit der *getenv*-Funktion zugreifen, *getenv*(*NAME*) liefert den Wert der Umgebungsvariablen *NAME*.

Beim Programmaufruf wird kein neuer Prozess erzeugt, der Programmspeicher des Prozesses wird vielmehr durch das neue Programm ersetzt. Es gibt also **keine Rückkehr** in das aufrufende Programm nach dem Muster eines Funktionsaufrufs.

Mehrere *exec*-Varianten sind verfügbar:

```
#include <unistd.h>

extern char **environ;

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 system(const char *command); (ANSI)
```

Werden dem neuen Programm *n* Argumente übergeben, sind diese bei *execl* einzeln ab Argumentposition 2 aufzuzählen. Die Liste ist mit `NULL` abzuschliessen.

Man beachte beim Programmaufruf folgende Konvention: Der erste an das Programm zu übergebende Parameter ist der Programmpfad.

Beispiel: Aufruf des C++-Compilers

- aus der Kommandozeile:

  ```
  g++ -g -o uebung-1 uebung-1.cc
  ```

- aus dem C-Programm:

  ```
  execl("g++", "g++", "-g", "-o",
        "uebung-1", "uebung-1.cc", NULL)
  ```

Wahrscheinlich wird *execl* scheitern und *-1* zurückliefern, weil der Pfad der Programmdatei nicht vollständig spezifiziert wurde und somit nur im aktuellen Verzeichnis nach g++ gesucht wird. Bei *execl* ist der komplette Pfad anzugeben!

Bei *execlp* dagegen wird nach der Programmdatei in allen Verzeichnissen gesucht, die in der Umgebungsvariablen *PATH* aufgezählt sind.

Bei *execle* wird gegenüber *execl* als weiteres Argument die Umgebung des Programms explizit übergeben. Bei Benutzung der anderen *exec*-Funktionen wird das Betriebssystem die Umgebung des aktuellen Programms ohne Änderung in das neue Programm kopieren.

Bei *execv* und *execvp* wird im zweiten Argument die Adresse eines Argumentvektors übergeben, in dem die zu übergebenden Zeichenketten (als Zeichen-Pointer) enthalten sein müssen.

Beispiel:

```
char arg0[]="g++";
char arg1[]="-g";
char arg2[]="-o";
char arg3[]="uebung-1";
char arg4[]="uebung-1.cc";

char *argv[]={arg0, arg1, arg2, arg3, arg4, NULL};

execvp("g++", argv);
```

Sind die Argumente bei Programmierung des *exec*-Aufrufs bekannt, ist diese Form natürlich zu umständlich gegenüber den *execl*-Funktionen. Wenn aber der Argumentvektor dynamisch bestimmt wird, wie etwa innerhalb einer Shell (hier werden die Argumente aus der Kommandozeile ermittelt), kann *execl* nicht verwendet werden.

1.0.3 exit – Prozessterminierung

```
#include <stdlib.h>
void exit(int status);
```

Ein Prozess beendet sich mit dem *exit*-Aufruf. Das Argument wird dem Elternprozess zugänglich gemacht, der es mit einer *wait*- oder *waitpid*-Operation erfragen kann.

Konvention: Das *exit*-Argument ist 0 bei normaler Terminierung, bei unerwarteten Fehlern dagegen 1.

1.0.4 Signale

Der Signalmechanismus ist ein auf Anwendungsebene verfügbarer Mechanismus zum Signalisieren und Behandeln von Ereignissen, die für das Anwendungsprogramm von Interesse sind. Beispielsweise werden Hardware-Ereignisse teilweise in Form von Signalen an die Anwendungsebene weitergeleitet, wo der Programmierer dann eine spezielle Signalbehandlung vorsehen kann.

Im einfachsten Fall dienen Signale dazu, ein ausser Kontrolle geratenes Programm mit einer bestimmten Tastatureingabe (z.B. CTRL-C) abzubrechen.

Beispiele für Signal-auslösende Ereignisse:

- Tastatur-generierte Signale
- Systemaufrufe z.B. (kill)
- Hardwarefehler (z.B. Division durch Null, Speicherzugriffsfehler)
- Subprozessterminierung
- Ablauf des „Weckers“

- Beim Eintreffen dringender Nachrichten vom Netzwerk
- Bei Beendigung asynchroner Ein-/Ausgabe-Operationen

Ein Signal wird zunächst *generiert* und dann den Empfängerprozessen *ausgeliefert*. Bei Auslieferung eines Signals wird der reguläre Kontrollfluß des Empfänger-Prozesses für die *Signalbehandlung* unterbrochen. Nach der Signalbehandlung wird die reguläre Verarbeitung ggf. fortgesetzt.

Die Signalbehandlung erfolgt nach Maßgabe des Empfängerprozesses durch

- *Ignorieren* des Signals
- Ausführung einer vorher dem Signal zugeordneten *Signalbehandlungsfunktion*
- Ausführung einer *Default-Behandlung*, die für viele Signale mit dem *Abbruch* des Empfängerprozesses verbunden ist

Signalbehandlungen können durch andere Signale unterbrochen werden.

1.1 Signaltypen

SIGABRT	abnormale Terminierung (*abort*-Funktion)
SIGALRM	Alarm-Timer abgelaufen (*alarm*-Funktion)
SIGBUS	System-abhängiger Hardware-Fehler
SIGCHLD	Zustandsänderung eines Subprozesses
SIGCLD	Zustandsänderung eines Subprozesses **Vorsicht:** Signal aus Kompatibilitätsgründen von alten UNIX-Systemen übernommen, aber völlig andere Semantik als SIGCHLD
SIGCONT	Job-Control-Signal, Fortsetzen der unterbrochenen Verarbeitung
SIGEMT	System-abhängiger Hardware-Fehler
SIGFPE	Arithmetik-Fehler
SIGHUP	Signal für den Sitzungsleiter bei Unterbrechung der Verbindung zum Kontrollterminal. Signal für Vordergrundprozesse bei Terminierung des Sitzungsleiters
SIGILL	illegalle Maschinenoperation
SIGINFO	Terminalstatus-Abfrage
SIGINT	Interrupt-Taste (DELETE oder CTRL-C)
SIGIO	Asynchrones E/A-Ereignis
SIGIOT	System-abhängiger Hardware-Fehler
SIGKILL	nicht verhinderbarer Prozeß-Abbruch
SIGPIPE	Schreiben in eine Pipeline, Leser terminiert
SIGPOLL	E/A-Ereignis beim Pollen
SIGPROF	Profiling Timer abgelaufen
SIGPWR	USV-Signal
SIGQUIT	Terminal-Quit-Taste
SIGSEGV	unzulässiger Hauptspeicherzugriff
SIGSTOP	Job-Control-Signal, Unterbrechen der Verarbeitung
SIGSYS	ungültiger Systemaufruf
SIGTERM	Software-Terminierung (kill-Kommando Defaultwert)
SIGTRAP	System-abhängiger Hardware-Fehler
SIGTSTP	Job-Control-Signal, Unterbrechen der Verarbeitung (CTRL-Z)
SIGTTIN	Job-Control-Signal, Hintergrundprozeß will vom Kontrollterminal lesen
SIGTTOU	Job-Control-Signal, Hintergrundprozeß will auf Kontrollterminal schreiben
SIGURG	Out-of-band Nachricht über Netz
SIGUSR1	Benutzer-definierbar
SIGUSR2	Benutzer-definierbar
SIGVTALRM	Timer abgelaufen
SIGWINCH	Änderung der Terminal-Fenstergröße
SIGXCPU	CPU-Limit überschritten
SIGXFSZ	Datei-Limit überschritten

- *kill(pid, signo)* – Sendet Signal *signo* an einen Prozeß oder eine Gruppe von Prozessen *pid* (Signal-Broadcast)

  Testet, ob ein Prozeß existiert (*signo* = 0)
- *raise(signo)* – erzeugt ein Signal für den aufrufenden Prozeß
- *alarm(n)* – erzeugt nach n Sekunden ein Alarm-Signal für den aufrufenden Prozeß

- *abort* – erzeugt SIGABRT
- *setitimer* – erzeugt Timer-Signale für einen von drei Timern

1.2 `signal` – ANSI-C Signalbehandlung

```
#include <signal.h>

void (*signal(int signum, void (*handler)(int)))(int);
```

Der Prototyp der Funktion wird leichter verständlich, wenn man ihn mit einer Typdefinition für eine Signalbehandlungsfunktion strukturiert:

```
typedef void (*signal_handler)(int)

signal_handler signal(int signalnum, signal_handler handler)
```

Eine Signalbehandlungsfunktion hat ein *int*-Argument und kein Resultat. *signal(n, handler)* definiert für ein Signal *n* eine (neue) Signalbehandlungsfunktion (*handler*) und liefert die aktuelle Signalbehandlungsfunktion als Resultatswert.

Nach dem *signal*-Aufruf wird beim Ausliefern des Signals *n* die Funktion *handler* aufgerufen und die Nummer des auslösenden Signals übergeben. Damit kann die gleiche Signalbehandlungsfunktion für mehrere Signale mit unterschiedlicher verwendet werden: Über das Argument ist eine Signal-spezifische Behandlung innerhalb der Signalbehandlungsfunktion möglich.

1.3 `sigaction` – Spezifikation der Signalbehandlung

Syntax:

```
#include <signal.h>

int sigaction(int signo, const struct sigaction *action,
                         const struct sigaction *oldaction);

struct sigaction {
  void (*sa_handler)(int); /* Adresse Signalbehandlungsfunktion */
  sigset_t sa_mask;        /* zusaetzlich zu block. Signale     */
  int      sa_flags;       /* verschiedene Optionen             */
};
```

Der Aufruf definiert die zukünftige Behandlung des Signals *signo* gemäß *action* und liefert die Spezifikation der bisherigen Signalbehandlung von *signo* in *oldaction*.

Ist *action* ein NULL-Pointer, bleibt die Signalbehandlung unverändert.

Die mit `sa_flags` spezifizierbaren Optionen:

SA_NOCLDSTOP	(nur SIGCHLD) kein SIGCHLD Signal bei Unterbrechung eines Subprozesses
SA_RESTART	Wiederholung von unterbrochenen System-Calls
SA_ONSTACK	Umschaltung auf benutzerdefinierten Stack während der Signalbehandlung
SA_NOCLDWAIT	(nur SIGCHLD) keine Zombies, *wait*-Semantik wird geändert
SA_NODEFER	keine automatische Blockade des gerade behandelten Signals während der Signalbehandlung
SA_RESETHAND	zu Beginn der Signalbehandlung wird `sa_handler` auf `SIG_DFL` zurückgesetzt
SA_SIGINFO	zusätzliche Parameter an den `sa_handler` übergeben

1.3.1 waitpid, wait – Warten auf Subprozessterminierung

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

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

Mit beiden Operationen erhält der Elternprozess Statusinformation über einen Subprozess. Ein häufige Anwendung ist das Warten auf Subprozessterminierung.

```
int kind_status;
pid_t kind_pid;

switch (kind_pid=fork()) {
    case -1: perror("fork-Fehler");
             exit(1);
    case  0: .
             .
             .
    default: // Warten auf Subprozess-Ende
             waitpid(kind_pid, &kind_status, 0);

             // normale Terminierung mit exit ?
             if (WIFEXITED(kind_status))
               // normale Terminierung, exit-Argument ausgeben
               cout << "Subprozess-Status:" << WEXITSTATUS(kind_status) <<
               endl ;
             else if (WIFSIGNALED(kind_status))
               // Terminierung durch Signal, welches ?
               cout << "Subprozess durch Signal abgebrochen, Signalnummer:"
                    << WTERMSIG(kind_status) <<  endl ;
```

```
    }

    exit(0);
```