Betriebssysteme: UNIX-Operationen zur Prozesskontrolle

WS 2016/17

8. November 2016

Prozesse und Programme

Programm

- Verschiedene Repräsentationen
  - Quelltext: Dateien auf Datenträgern
  - Maschinenprogramm: Datei auf Datenträger
  - Prozess: Hauptspeicher

Prozess: Programm in Ausführung

- Programmspeicher (auch „Adressraum“, „Prozess-Image“), in Segmente unterteilt: Code, statische Daten, Heap, Stack
- Deskriptor (Metadaten des Prozesses), Element der Prozesstabelle
  - Identifikationsnummer
  - Deskriptor der offenen Dateien
  - Deskriptor für Programmspeicher-Segmente
  - User-ID
  - Priorität
  - Prozessorzustand
  - ...

Prozesserzeugung

„Normale“ Prozesserzeugung: Untrennbar gekoppelt an Programmaufruf

- Programm wird aufgerufen (interaktiv oder durch anderes Programm)
  - Prozessdeskriptor initialisieren
  - Programmspeicher reservieren
  - Programm laden (Programmspeicher initialisieren)
  - Warten auf Prozessorzuteilung

UNIX: Trennung von Programmaufruf und Prozesserzeugung

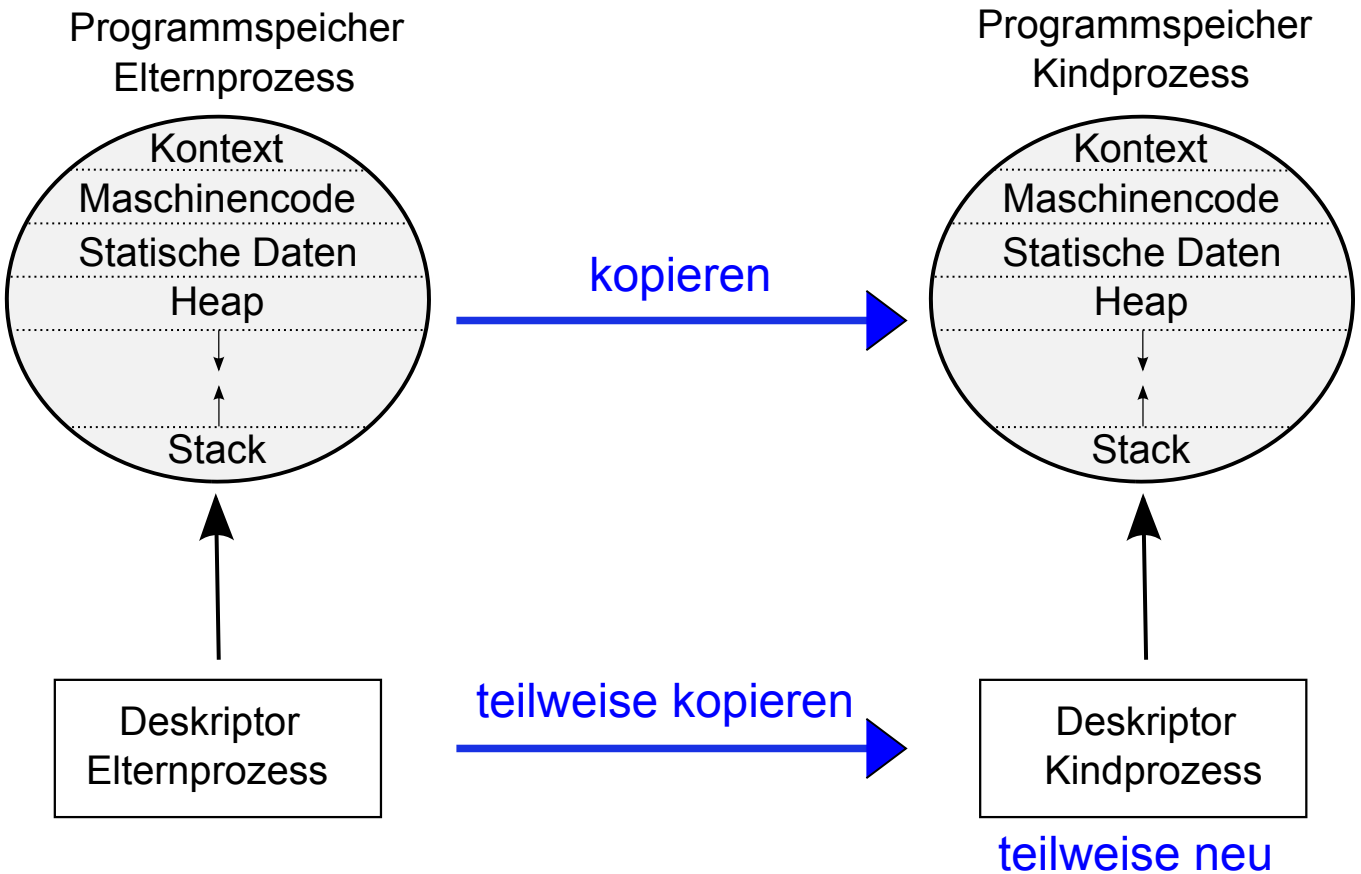
- *exec*: Systemkern „tauscht den Programmspeicher des Prozesses aus“
- *fork*: Systemkern kopiert vorhandenen Prozess

Windows-Beispiel: CreateProcess (Programm in neuem Prozess starten)

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ...

    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    ) {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

UNIX „fork“: Systemkern kopiert aufrufenden Prozess

fork - Erzeugung eines neuen Prozesses durch Clone-Operation



fork: Kopieren des Programmspeichers

Separater Programmspeicher pro Prozess

- Kind hat eigenen, separaten Programmspeicher
- Speicher direkt nach dem fork absolut identisch
- Kind kann seinen Speicher danach unabhängig vom Elternprozess ändern

Effiziente Implementierung

- Prinzip: Zwei logisch verschiedene Speicherbereiche mit gleichem Inhalt werden durch einen gemeinsam genutzten physikalischen Speicherbereich implementiert.
- Beispiel: Das Maschinecode-Segment wird gemeinsam benutzt
- „Copy-On-Write“: Zunächst alles gemeinsam nutzen, erst bei Änderungen kopieren
- Details dazu später im Themenbereich „Speicherverwaltung“

fork: Initialisierung des Prozessdeskriptors

Vom Elternprozess vererbte Attribute

- Deskriptoren für offene Dateien und Kommunikationsendpunkte (sockets, pipes)
- Benutzer
- aktuelles Verzeichnis
- Priorität („nice“-Wert)
- Prozessorzustand (leicht modifiziert)
- ...

Nicht vererbte Attribute

- Prozess-Identifikationsnummer (PID) wird neu vergeben
- PID des Elternprozesses
- Erzeugungszeit
- verbrauchte CPU-Zeit
- ...

fork: Prozessorzustandsdeskriptor

Nutzungsprinzip

- Systemkern kann einem laufenden Prozess den Prozessor temporär „wegnehmen“
- Die für die spätere Fortsetzung relevanten Registerinhalte werden „gerettet“
- Retten: Kopieren in den Prozessdeskriptor als „Prozessorzustands-Deskriptor“ (PZD)

Bestandteile des PZD

- Programmzähler („program counter“, PC)
- Befehlsregister, Statusregister
- Mehrzweckregister
- ...

Der PZD des Kindprozesses

- Der PZD des Elternprozesses wird (fast) kopiert
- Speziell: PC wird vererbt, Kindprozess ist an der gleichen Stelle (Rückkehr aus *fork*)
- Nicht vererbt: Das Returnwert-Register enthält beim Kind Null, beim Elternprozess die Kind-PID

fork: Programmbeispiel

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

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

  printf("vor fork, PID=%d\n", meine_pid=getpid());
  kind_pid=fork();

  // Ab hier sind zwei Prozesse im Spiel!

  switch (kind_pid) {
    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("Ende: PID=\d\n", meine_pid);
  exit(0);
}
```

Welche Ausgabe erzeugt dieses Programm, falls Eltern-PID=10 und Kind-PID=11?

Fehlschlag von fork

Mögliche Gründe für fork-Fehler

- Limit für die Anzahl der Prozesse eines Benutzer erreicht
- Ungenügend Speicherplatz für den Programmspeicher
- Ungenügend Speicherplatz für den neuen Deskriptor

Fehlerbehandlung im Programm

- *fork*-Rückgabewert bei Fehlschlag: -1
- globale Variable *errno* (`errno.h`) enthält Fehlercode, z.B. `ENOMEM`
- Funktion *perror* gibt Fehlertext zum Fehlercode aus, z.B.
  fork-Fehler: cannot allocate memory

```
switch (kind_pid = fork()) {
  case -1: perror("fork-Fehler");
           exit(1);
  ...
```

Mit `exit(1)` terminiert der aktuelle Prozess. Der Parameter 1 wird dem Elternprozess zur Verfügung gestellt und signalisiert einen fatalen Fehler.

exec

Programmaufruf

Funktionsweise

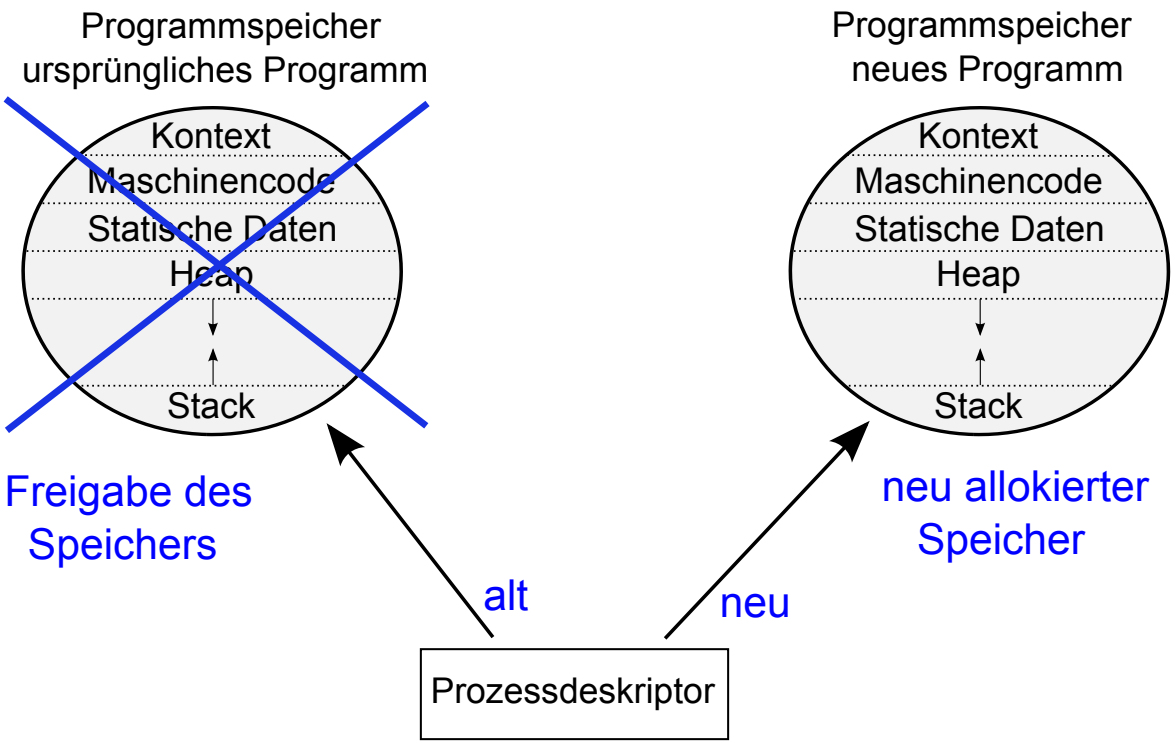
- Einem Prozess können **nacheinander** unterschiedliche Programme zugeordnet sein
- *exec* ersetzt das aufrufende Programm durch ein anderes Programm
- neues Programm wird ab dem ersten Befehl ausgeführt
- kein Rücksprung in das ursprüngliche Programm
- Dateien bleiben offen

Parameter

- Dateipfad des aufzurufenden Programms
- Liste der an das Programm zu übergebenden Parameter (beliebige Zeichenketten)
- Liste der Umgebungsvariablen („Environment“)

exec: Programmaufruf (kein neuer Prozess!)

exec - "Programmspeicher austauschen"



Nur ein Prozess ist beteiligt!

Die exec-Familie

exec-Varianten

```
#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)
```

Die Buchstaben hinter *exec*:

- l - Listenformat für Parameter
- v - Vektorformat für Parameter
- p - Pfadsuche gemäß PATH
- e - Umgebungsvariablen (Environment) werden explizit angegeben

execl vs. *execv*

Argumentliste (l) versus Argumentvektor (v)

- Bei *execl* wird jedes an das Programm zu übergebende Argument einzeln übergeben, ein NULL-String terminiert die variabel lange Liste

```
execl("/usr/bin/ls", "ls", "-l", "/tmp", NULL)
```

- Bei *execv* wird ein Array übergeben, die `char*`-Elemente sind die an das Programm zu übergebenden Zeichenketten

```
char* argumentvektor[] = {"ls", "-l", "/tmp", NULL};
execv("/usr/bin/ls", argumentvektor)
```

exec mit oder ohne „p“

PATH: Liste von Verzeichnissen mit Programmen

- Bei Standardprogrammen entscheidet der Administrator über den Speicherort. Hart-kodierte Pfade in Programmen sind daher ungeeignet.
- Umgebungsvariable PATH enthält die Liste der Programm-Verzeichnisse, z.B.

  `PATH=/opt/sbin:/usr/local/bin:/usr/bin:/bin`
- Keine automatische Suche im aktuellen Verzeichnis
  (Was macht folgendes Shell-Kommando? `PATH=$PATH:.`)
- Die PATH-Variable wird von *execvp* und *execlp* benutzt
- Die anderen *exec*-Varianten erwarten einen vollständigen Pfad
- Auch ein relativer Pfad ist ein vollständiger Pfad, z.B. `"./testprog"`)

exec-Funktionen mit „e“ am Ende

Umgebungsvariablen

- Umgebungsvariablen sind neben Parametern eine weitere Möglichkeit, das Programmverhalten zu konfigurieren
- Der Zugriff erfolgt in C
  - mit getenv: `getenv("HOME")`
  - oder über den 3. *main*-Parameter:

```
int main(int agumentanzahl,
         const char *argumentvektor[],
         const char *umgebungsvariablen[])
```

- Mit *execle* und *execvpe* kann man dem neuen Programm eine beliebige Liste von Umgebungsvariablen übergeben
- Bei den anderen exec-Varianten erhält das neue Programm eine Kopie der aktuellen Umgebungsvariablen

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar" fehlt
- ENOMEM – Hauptspeichermangel
- ENOENT – Datei nicht gefunden (no directory Entry)

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar" fehlt
- ENOMEM – Hauptspeichermangel
- ENOENT – Datei nicht gefunden (no directory file, ...)

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar“ fehlt
- ENOMEM – Hauptspeichermangel
- ENOENT – Datei nicht gefunden (no directory file)

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar“ fehlt
- ENOMEM – Hauptspeichermangel
- ENOENT – Datei nicht gefunden (no directory file (e.g.

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar“ fehlt
- ENOMEM – Hauptspeichrmangel
- ENOENT – Datei nicht gefunden (no directory file (e.g.

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar“ fehlt
- ENOMEM – Hauptspeichrmangel
- ENOENT – Datei nicht gefunden (no directory file (e.g.

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar“ fehlt
- ENOMEM – Hauptspeichrmangel
- ENOENT – Datei nicht gefunden (NO directory ENTry)

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar“ fehlt
- ENOMEM – Hauptspeichrmangel
- ENOENT – Datei nicht gefunden (NO directory ENTry)

Mögliche Fehler beim Programmaufruf?

- E2BIG – Argumentliste zu groß
- EACCESS – Zugriffsrecht „ausführbar“ fehlt
- ENOMEM – Hauptspeichrmangel
- ENOENT – Datei nicht gefunden (NO directory ENTry)

fork und *exec* kombinieren

In fast allen Anwendungsfällen werden *fork* und *exec* kombiniert

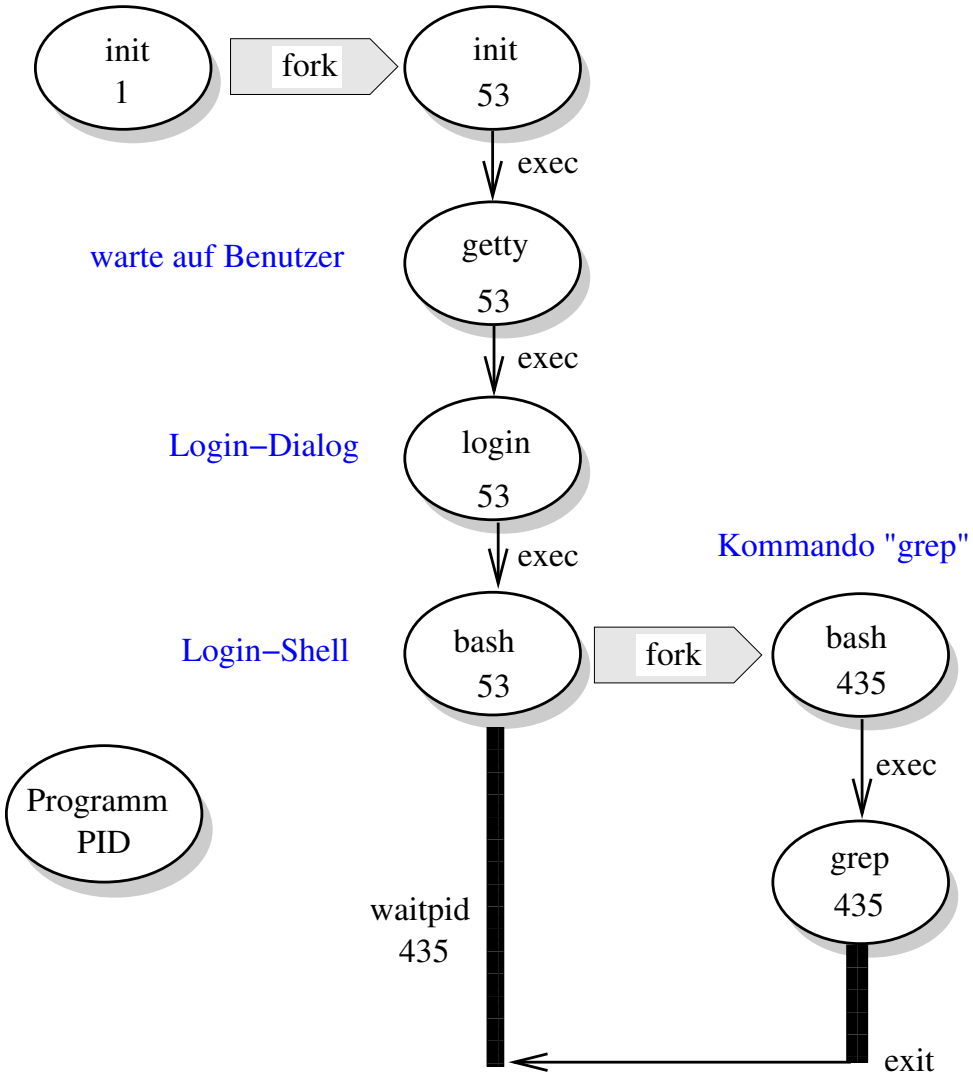
- Elternprozess erzeugt Kindprozess
- Kindprozess führt neues Programm aus
- Elternprozess setzt die Ausführung des aktuellen Programms fort

Kombinationsschema

```
switch (pid = fork()){
case -1: ...
case  0:
         // Kind ruft anderes Programm auf
         execvp( ... );
         // exec gescheitert
         perror("exec error");
         exit(1);
}

// Elternprozess macht weiter
...
```

Was passiert, wenn man das exit(1) vergisst?

fork und *exec* kombinieren



Prozessterminierung

Prozess terminiert sich selbst

- Prozess kann sich selbst terminieren: *exit(Exitcode)*
  exit-Code: 0 - Erfolg, 1 - fataler Fehler
- Systemkern gibt Ressourcen frei:
  - Programmspeicher
  - Deskriptoren für Dateizugriffe und Kommunikationsendpunkte
  - Prozessorzustands-Deskriptor
  - ...
- Prozessdeskriptor bleibt erhalten („Zombie“),
  enthält:
  - exit-Code
  - Ressourcenverbrauchswerte (z.B. verbrauchte CPU-Zeit, gesamte Laufzeit)

Prozess wird durch Signal terminiert

- Ein besonderes Ereignis tritt ein (z.B. Benutzer betätigt STRG-C-Taste)
- Systemkern erzeugt ein Signal für einen Empfängerprozess
- Empfängerprozess wird durch Signal terminiert (andere Signalbehandlung möglich)
- Prozessdeskriptor enthält Details zum Ereignis
- ansonsten wie bei *exit*

exit und *_exit*

Exit-Handler

- Jeder Prozess kann mit *atexit(f)* eine Funktion *f* als **exit-Handler** registrieren
- Bei Terminierung mit *exit* wird dann *f* aufgerufen
- Verwendung: „Aufräumaktionen“ aller Art, z.B. temporäre Datei löschen
- _exit terminiert den Prozess **ohne** Aufruf von exit-Handlern

Wann _exit statt exit?

```
switch (pid = fork()){
case -1: ...
case  0:
         execvp( ... );
         // exec gescheitert
         perror("exec error");
         _exit(1);
...
```

Was passiert hier bei Aufruf von exit, wenn ein exit-Handler registriert ist, der temporäre Dateien löscht?

waitpid - Status Kindprozess abrufen

Statusinformationen

- Kind durch ein Signal terminiert: Signalnummer
- Kind durch *exit* terminiert: exit-Argument

Warte-Szenarien

- Bei waitpid-Aufruf ist Kind schon terminiert: Kein Warten
- Bei waitpid-Aufruf ist Kind noch aktiv: Warten, bis Kind terminiert
- waitpid-Aufruf mit Option `WNOHANG`: Kein Warten

Mehrere Kindprozesse

- waitpid(15,...) warten auf das Kind mit PID 15
- waitpid(-1,...) wartet auf irgendein Kind, Returnwert: Kind-PID
- weitere Möglichkeiten, siehe Manual

waitpid: Makros zur Statusanalyse

Beispiel-Code: Programm starten und Ende abwarten

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

...

int  kindstatus;

switch (pid = fork()){
case -1: ...
case  0:
         execvp( ... );
         // exec gescheitert
         perror("exec error");
         _exit(1);
}

waitpid(pid, &status, 0);

if(WIFEXITED(status))
   printf("Kind terminierte mit exit(%d)\n", WEXITSTATUS(status));
else if(WIFSIGNALED(status))
   printf("Kind terminierte durch Signal %d\n", WTERMSIG(status));
```