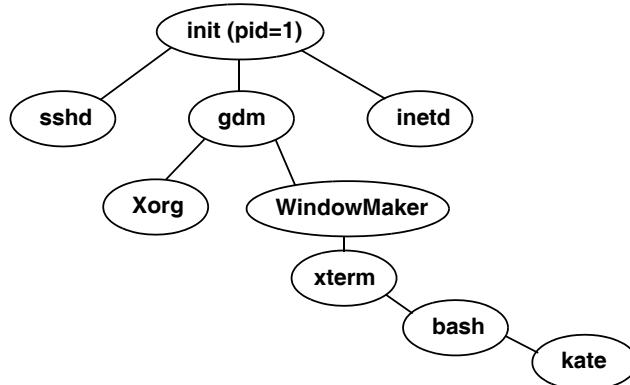


- Prozesse
- POSIX-Prozess-Systemfunktionen
- POSIX-Signale

- Prozesse sind eine Ausführungsumgebung für Programme
  - haben eine Prozess-ID (PID, ganzzahlig positiv)
  - führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft (s. Vorlesung ab J)

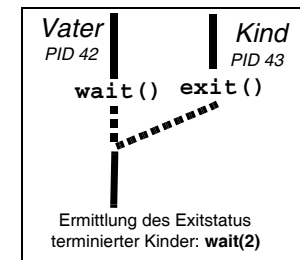
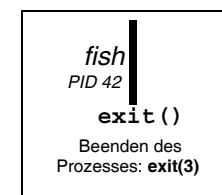
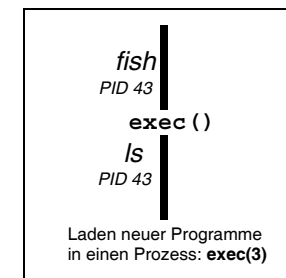
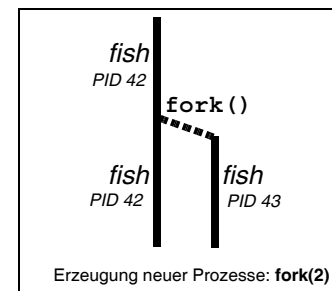
## U6-1 UNIX-Prozesshierarchie

- Zwischen Prozessen bestehen Vater-Kind-Beziehungen
  - ◆ der erste Prozess wird direkt vom Systemkern gestartet (z.B. *init*)
  - ◆ es entsteht ein Baum von Prozessen bzw. eine Prozesshierarchie



- ◆ Beispiel: **kate** ist ein Kind von **bash**, **bash** wiederum ein Kind von **xterm**

## U6-2 POSIX Prozess-Systemfunktionen

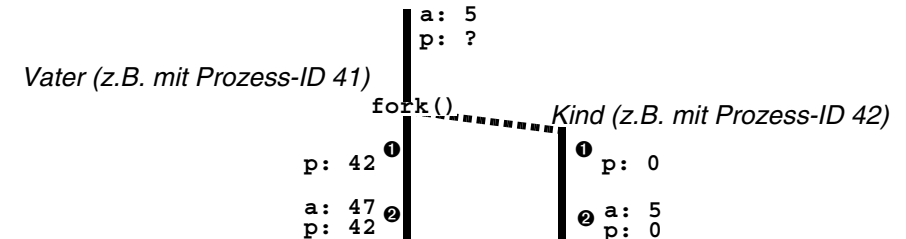


## 1 fork(2): Erzeugung eines neuen Prozesses

- Erzeugt einen neuen Kindprozess
- Exakte Kopie des Vaters...
  - ◆ Datenssegment (neue Kopie, gleiche Daten)
  - ◆ Stacksegment (neue Kopie, gleiche Daten)
  - ◆ Textsegment (gemeinsam genutzt, da nur lesbar)
  - ◆ Filedesktoren (geöffnete Dateien)
  - ◆ ...
- ...mit Ausnahme der Prozess-ID
- Kind startet Ausführung hinter dem fork() mit dem geerbten Zustand
  - das ausgeführte Programm muss anhand der PID (Rückgabewert von **fork()**) entscheiden, ob es sich um den Vater- oder den Kindprozess handelt

## 1 fork(2): Beispiel

```
int a=5; pid_t p = fork(); ❶
a += p; ❷
switch(p) {
    case -1: /* fork-Fehler, es wurde kein Kind erzeugt */
        ...
    case 0: /* Hier befinden wir uns im Kind */
        ...
    default: /* Hier befinden wir uns im Vater */
        ...
}
```



## 2 exec(3)

- Lädt Programm zur Ausführung in den aktuellen Prozess
- **ersetzt** aktuell ausgeführtes Programm: Text-, Daten- und Stacksegment
- behält: Filedesktoren (= geöffnete Dateien), Arbeitsverzeichnis, ...
- Aufrufparameter:
  - ◆ Dateiname des neuen Programmes (z.B. `"/bin/cp"`)
  - ◆ Argumente, die der `main`-Funktion des neuen Programms übergeben werden (z.B. `"/bin/cp"`, `"/etc/passwd"`, `"/tmp/passwd"`)
  - ◆ evtl. Umgebungsvariablen

### ■ Beispiel

```
execl("/bin/cp", "/bin/cp", "/etc/passwd", "/tmp/passwd", NULL);
```

- **exec** kehrt nur im Fehlerfall zurück

## 2 exec(3) Varianten

- mit Angabe des vollen Pfads der Programm-Datei in `path`

```
int execl(const char *path, const char *arg0, ...,
          const char *argn, char * /*NULL*/);
```

```
int execlp(const char *path, char *const argv[]);
```

- zum Suchen von `file` wird die Umgebungsvariable `PATH` verwendet

```
int execlp(const char *file, const char *arg0, ..., const char
          *argn, char * /*NULL*/);
```

```
int execlp(const char *file, char *const argv[]);
```

### 3 exit(3)

- beendet aktuellen Prozess mit einem Status-Byte
  - Konvention: Status 0 bedeutet Erfolg, alles andere eine Fehlernummer
  - Bedeutung der Exitstatus üblicherweise in Manpage dokumentiert
  - Exitstatus `EXIT_FAILURE` und `EXIT_SUCCESS` vordefiniert
- gibt alle Ressourcen frei, die der Prozess belegt hat, z.B.
  - ◆ Speicher
  - ◆ Filedeskriptoren (schließt alle offenen Files)
  - ◆ Kerndaten, die für die Prozessverwaltung verwendet wurden
- Prozess geht in den *Zombie*-Zustand über
  - ◆ ermöglicht es dem Vater auf den Tod des Kindes zu reagieren (**wait(2)**)
  - ◆ Zombie-Prozesse belegen Systemressourcen und sollten schnellstmöglich beseitigt werden!
  - ◆ ist der Vater schon vor dem Kind terminiert, so wird der Zombie an den Prozess mit PID 1 (z.B. *init*) weitergereicht, welcher diesen sofort beseitigt

### 4 wait(2)

- `wait` blockiert den Vater, bis ein Kind terminiert oder gestoppt wird
  - ◆ `pid` dieses Kind-Prozesses wird als Ergebnis geliefert
  - ◆ als Parameter kann ein Zeiger auf einen `int`-Wert mitgegeben werden, in dem der Exitstatus (**16 Bit**) des Kind-Prozesses abgelegt wird
  - ◆ in den Status-Bits wird eingetragen "was dem Kind-Prozess zugestoßen ist", Details können über Makros abgefragt werden:
    - Prozess mit `exit()` terminiert: `WIFEXITED(status)`
      - exit-Parameter (unteres Byte): `WEXITSTATUS(status)`
    - Prozess durch Signal abgebrochen: `WIFSIGNALED(status)`
      - Nummer des Signals: `WTERMSIG(status)`
  - weitere siehe man 2 wait

### 4 wait(2)

- Warten auf Statusinformationen von Kind-Prozessen (Rückgabe: PID)
 

```
pid_t wait(int *status);
```
- Beispiel:

```
int main(int argc, char *argv[]) {
    pid_t pid;
    pid = fork();
    if (pid > 0) {
        /* Vater */
        int status;
        wait(&status); /* Fehlerbehandlung nicht vergessen! */
        printf("Kindstatus: %x", status); /* nackte Status-Bits ausg. */
    } else if (pid == 0) {
        /* Kind */
        execl("/bin/cp", "/bin/cp", "x.txt", "y.txt", NULL);
        /* diese Stelle wird nur im Fehlerfall erreicht */
        perror("exec /bin/cp"); exit(EXIT_FAILURE);
    } else {
        /* pid == -1 --> Fehler bei fork */
    }
}
```

## U6-3 POSIX-Signale

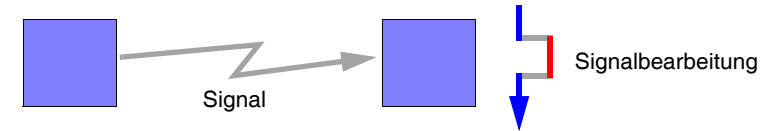
- Zwei Arten von Signalen
  - ◆ synchrone Signale: durch Prozessaktivität ausgelöst (Analogie: Traps)
  - ◆ asynchrone Signale: "von außen" ausgelöst (Analogie: Interrupts)
- Zwecke von Signalen
  - ◆ Ereignissignalisierung des Betriebssystemkerns an einen Prozess
  - ◆ Ereignissignalisierung zwischen Prozessen

# 1 Reaktion auf Signale

- **abort**
  - ◆ erzeugt Core-Dump (Segmente + Registerkontext) und beendet Prozess
- **exit**
  - ◆ beendet Prozess, ohne einen Core-Dump zu erzeugen
- **ignore**
  - ◆ ignoriert Signal
- **stop**
  - ◆ stoppt Prozess
- **continue**
  - ◆ setzt einen gestoppten Prozess fort
- **signal handler**
  - ◆ Aufruf einer Signalbehandlungsfunktion, danach Fortsetzung des Prozesses

# 2 POSIX-Signalbehandlung

- Signal bewirkt Aufruf einer Funktion



- ◆ nach der Behandlung läuft Prozess an unterbrochener Stelle weiter
- **Systemschnittstelle**
  - ◆ sigaction
  - ◆ sigprocmask
  - ◆ sigsuspend
  - ◆ kill

## 3 Signalhandler installieren: sigaction

- Prototyp

```
#include <signal.h>

int sigaction(int sig, /* Signal */
              const struct sigaction *act, /* Handler */
              struct sigaction *oact /* Alter Handler */ );
```

- Handler bleibt solange installiert, bis ein neuer Handler mit **sigaction** installiert wird

- **sigaction**-Struktur

```
struct sigaction {
    void (*sa_handler)(int); /* Behandlungsfunktion */
    sigset_t sa_mask; /* Signalmaske während der Behandlung */
    int sa_flags; /* Diverse Einstellungen */
};
```

## 3 Signalhandler installieren: sigaction Handler (sa\_handler)

- Signalbehandlung kann über **sa\_handler** eingestellt werden:

- **SIG\_IGN** Signal ignorieren
- **SIG\_DFL** Default-Signalbehandlung einstellen
- **Funktionsadresse** Funktion wird in der Signalbehandlung aufgerufen und ausgeführt
  - ⇒ **SIG\_IGN** und **SIG\_DFL** werden über **exec(3)** vererbt, nicht aber eine Behandlungsfunktion (nicht möglich, warum?)

- Mit **sa\_flags** lässt sich das Verhalten beim Signalempfang beeinflussen
  - bei uns gilt: **sa\_flags=SA\_RESTART**

### 3 Signalhandler installieren: sigaction Maske (sa\_mask)

- verzögerte Signale
  - ◆ während der Ausführung der Signalhandler-Prozedur wird das auslösende Signal blockiert
  - ◆ bei Verlassen der Signalbehandlungsroutine wird das Signal deblockiert
  - ◆ es wird maximal eine Zustellung je Signal zwischengespeichert
- mit `sa_mask` in der `struct sigaction` kann man zusätzliche Signale während der Behandlung des Signals blockieren
- Auslesen und Modifikation von Signal-Masken des Typs `sigset_t` mit:
  - ◆ `sigaddset()`: Signal zur Maske hinzufügen
  - ◆ `sigdelset()`: Signal aus Maske entfernen
  - ◆ `sigemptyset()`: Alle Signale aus Maske entfernen
  - ◆ `sigfillset()`: Alle Signale in Maske aufnehmen
  - ◆ `sigismember()`: Abfrage, ob Signal in Maske enthalten ist

### 3 Signalhandler installieren: Beispiel

- Beispiel:

```
#include <signal.h>
void my_handler(int sig) { ... }
...
struct sigaction action;
sigemptyset(&action.sa_mask);
action.sa_flags = SA_RESTART;
action.sa_handler = my_handler;
sigaction(SIGUSR1, &action, NULL);
```

### 4 Signal zustellen

- Systemaufruf `kill(2)`

```
int kill(pid_t pid, int signo);
```

- Kommando `kill(1)` aus der Shell (z. B. `kill -USR1 <pid>`)

### 5 Ausgewählte POSIX-Signale

Das Defaultverhalten bei den meisten Signalen ist die Terminierung des Prozesses, bei einigen Signalen mit Anlegen eines Core-Dumps.

- SIGALRM: Timer abgelaufen (`alarm(2)`, `setitimer(2)`)
- SIGCHLD (ignore): Statusänderung eines Kindprozesses
- SIGINT: Interrupt; (Shell: CTRL-C)
- SIGQUIT (core): Quit; (Shell: CTRL-\)
- SIGKILL (nicht behandelbar): beendet den Prozess
- SIGTERM: Terminierung; Standardsignal für `kill(1)`
- SIGSEGV (core): Speicherschutzverletzung
- SIGUSR1, SIGUSR2: Benutzerdefinierte Signale

### 6 Ändern der prozessweiten Signal-Maske

```
int sigprocmask(int how, /* Verknüpfung der Masken */
               const sigset_t *set, /* neue Maske */
               sigset_t *oset /* Speicher für alte Maske */ );
```

- how:

- ◆ `SIG_BLOCK`: blockiert Signale ( `Maske |= *set` )
- ◆ `SIG_SETMASK`: blockiert Signale der Maske ( `Maske = *set` )
- ◆ `SIG_UNBLOCK`: deblockiert Signale der Maske ( `Maske &= ~( *set )` )

- Beispiel

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigprocmask(SIG_BLOCK, &set, NULL); /* Blockiert SIGUSR1 */
```

- Anwendung: Sperren der Signalbehandlung in kritischen Abschnitten  
Vgl.: Sperren der Interruptbehandlung in kritischen Abschnitten (`cli()`, `sei()`)
- Die prozessweite Signal-Maske wird über `exec(3)` vererbt.

## 7 Warten auf Signale

- Problem: Prozess will in einem kritischen Abschnitt auf ein Signal warten
  - Signal deblockieren
  - Auf Signal warten
  - Signal blokieren
  - Kritischen Abschnitt bearbeiten
- Operationen müssen atomar am Stück ausgeführt werden!
  - ➡ gleiche Problematik wie bei den Stromsparmodi des AVR-Prozessors
- Sigsuspend

```
#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

- (1) `sigsuspend(mask)` setzt `mask` als Signal-Maske
- (2) Der Prozess blockiert bis zum Eintreffen eines Signals
- (3) Gegebenenfalls wird der Signalhandler ausgeführt
- (4) `sigsuspend` restauriert die ursprüngliche Signal-Maske und kehrt zurück

## 8 POSIX-Signale vs. Interrupts

|                         | Signale                                                                    | Interrupts                                    |
|-------------------------|----------------------------------------------------------------------------|-----------------------------------------------|
| Behandlung installieren | <code>sigaction()</code>                                                   | ISR ( ) -Makro der C-Bibliothek               |
| Behandlungsfunktion     | Signalhandler                                                              | Interrupthandler                              |
| Auslösung               | durch Prozesse mit <code>kill()</code><br>oder<br>durch das Betriebssystem | durch die Hardware                            |
| Synchronisation         | <code>sigprocmask()</code>                                                 | <code>cli()</code> , <code>sei()</code>       |
| Warten auf IRQ/Signal   | <code>pause()</code>                                                       | <code>sleep_cpu()</code>                      |
|                         | <code>sigsuspend()</code>                                                  | <code>sei()</code> + <code>sleep_cpu()</code> |

- Signale und Interrupts sind sehr ähnliche Konzepte auf unterschiedlichen Ebenen
- Viele Probleme treten in beiden Fällen auf und sind konzeptionell identisch zu lösen