Übungen zu Systemnahe Programmierung in C

Abschnitt 12.1: Prozesse

13.07.2020

Tim Rheinfels
Benedict Herzog
Bernhard Heinloth

Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

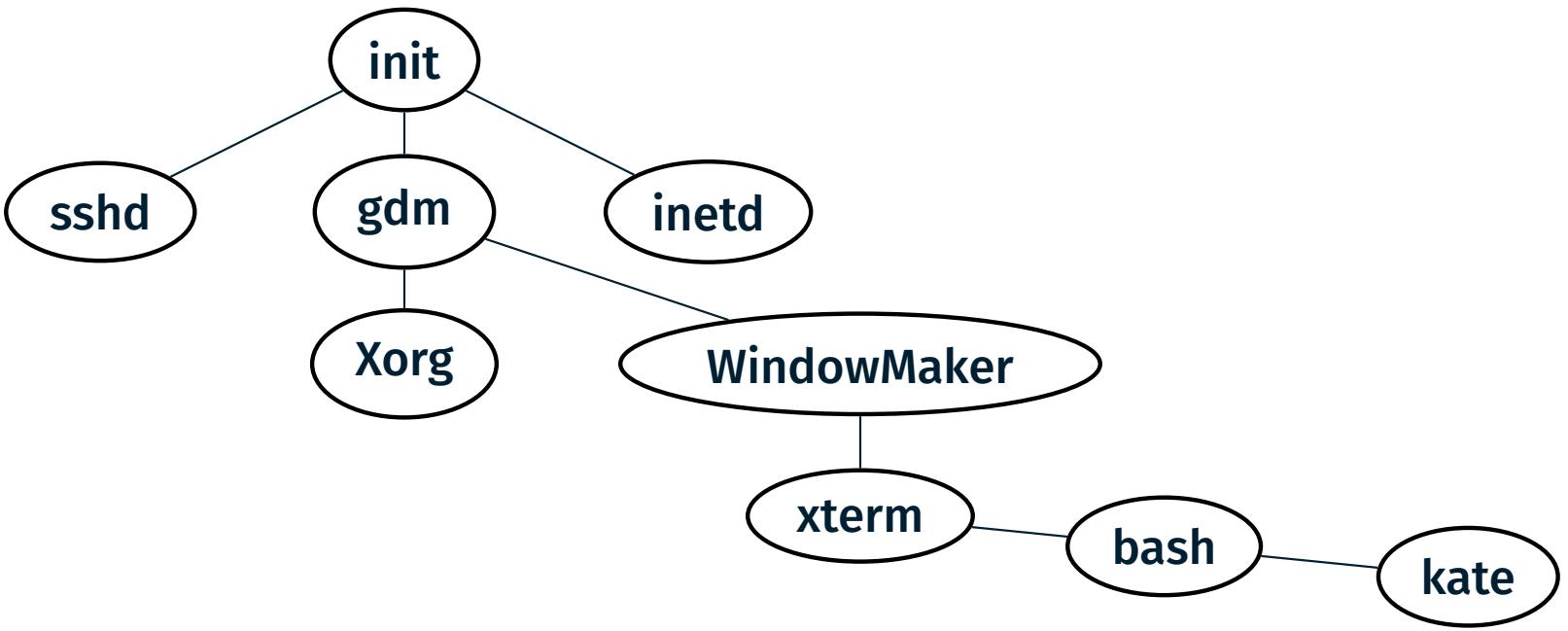
TECHNISCHE FAKULTÄT

Prozesse

- Prozesse sind eine Ausführumgebung für Programme
  - Haben eine Prozess-ID (PID, ganzzahlig positiv)
  - Führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft
  - Speicher
  - Adressraum
  - Geöffnete Dateien
  - …
- Visualisierung von Prozessen: `ps(1)`, `pstree(1)`, `htop(1)`

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



- **kate** ist ein Kind von **bash**, **bash** wiederum ein Kind von **xterm**

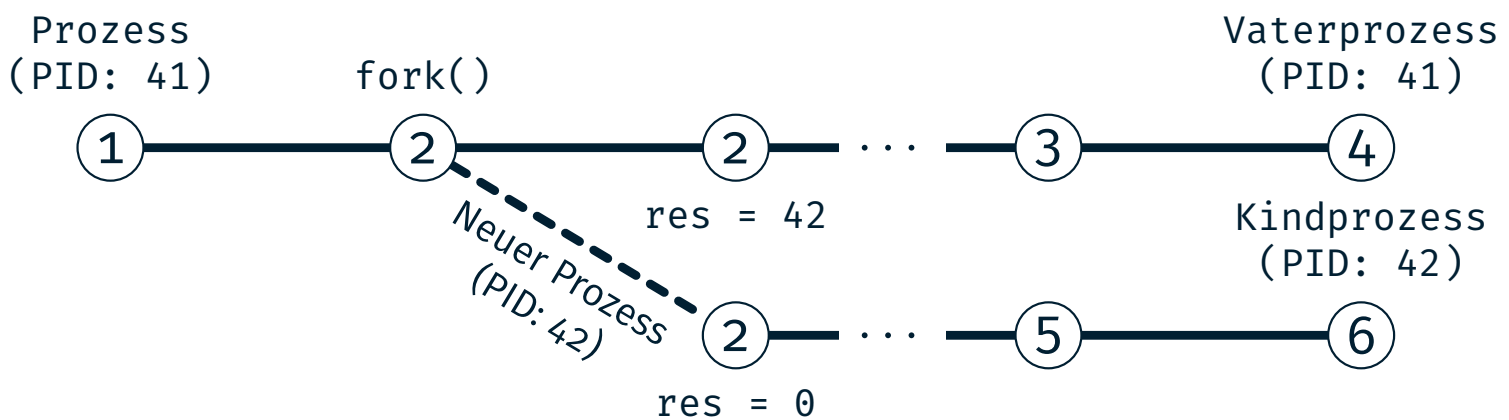
Kindprozess erzeugen (1)

```
01 pid_t fork(void);
```

- Erzeugt einen neuen Kindprozess
- Exakte Kopie des Vaters:
  - Daten- und Stacksegment (Kopie)
  - Textsegment (gemeinsam genutzt)
  - Dateideskriptoren (geöffnete Dateien)
  - **Ausnahme:** Prozess-ID
- Vater-/Kindprozess kehren beide aus dem `fork(2)` zurück
- Unterscheidbar am Rückgabewert von `fork(2)`
  - Vater: PID des Kindes
  - Kind: 0
  - Fehler: `-1`

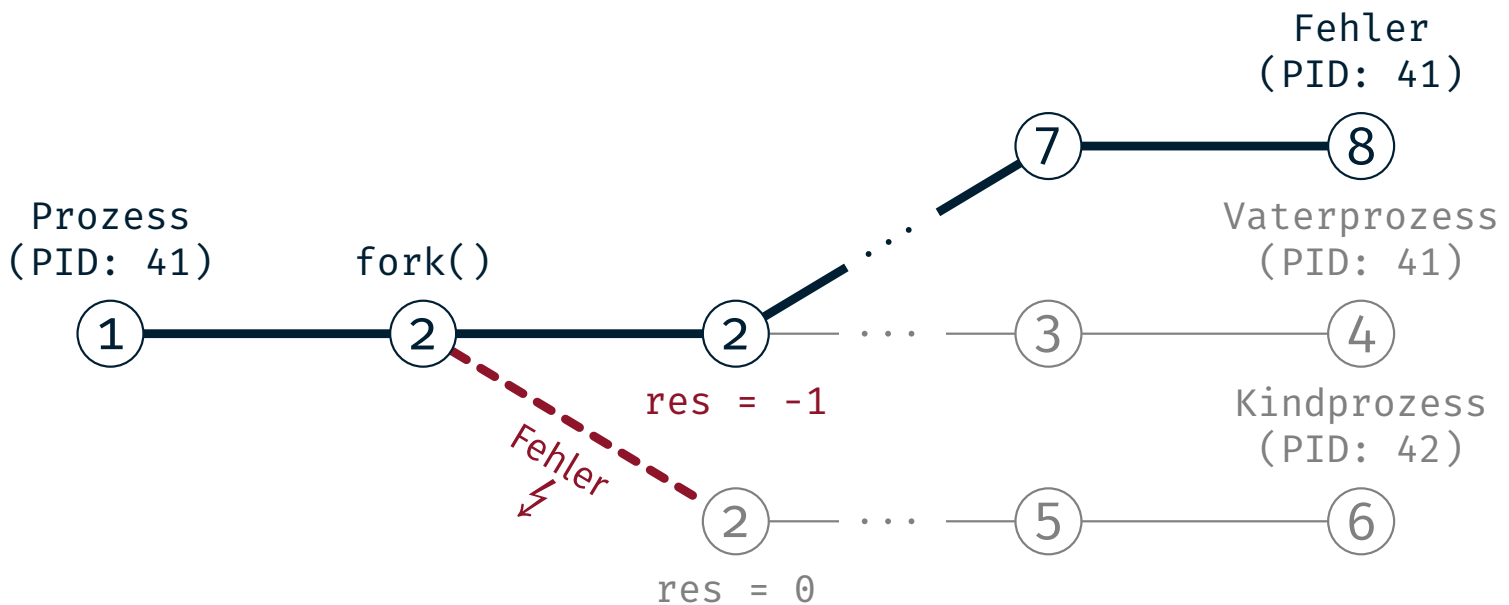
Kindprozess erzeugen (2)

```c
printf("Prozess (PID: %d)", getpid());
pid_t res = fork();
if(res > 0) {
    printf("Vaterprozess (PID: %d)", getpid());
} else if(res == 0) {
    printf("Kindprozess (PID: %d)", getpid());
} else {
    printf("Fehler (PID: %d)", getpid());
    // [...] Fehlerbehandlung
}
```



Kindprozess erzeugen (2)

```
01 printf("Prozess (PID: %d)", getpid());
02 pid_t res = fork();  // Schlägt fehl ↯
03 if(res > 0) {
04     printf("Vaterprozess (PID: %d)", getpid());
05 } else if(res == 0) {
06     printf("Kindprozess (PID: %d)", getpid());
07 } else {
08     printf("Fehler (PID: %d)", getpid());
09     // [...] Fehlerbehandlung
10 }
```



Auf Kindprozess warten (1)

```
01 pid_t wait(int *status);
```

- `wait(2)` blockiert bis ein beliebiger Kind-Prozess terminiert
- Rückgabewert
  - `> 0` Prozess-ID des Kindprozesses
  - `-1` Fehler
- Status enthält Grund des Terminierens:
  - `WIFEXITED(status)` `exit(3)` oder return aus `main()`
  - `WIFSIGNALED(status)` Prozess durch Signal abgebrochen
  - `WEXITSTATUS(status)` Exitstatus
  - `WTERMSIG(status)` Signalnummer

↝ 13.1

- Weitere Makros: siehe Dokumentation `wait(2)`

Auf Kindprozess warten (2)

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

- `waitpid(2)` blockiert bis bestimmter Kind-Prozess terminiert

  `pid > 0` Kindprozess mit Prozess-ID pid

  `pid = -1` Beliebige Kindprozesse

  …

- Optionen:

  `WNOHANG` sofort zurückkehren, wenn kein Kind beendet wurde (nicht blockieren)

  …

- Rückgabewert

  `> 0` Prozess-ID des Kindprozesses

  `0` kein Prozess beendet (bei Verwendung von `WNOHANG`)

  `-1` Fehler – Details siehe `waitpid(2)`

Prozess beenden

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

- Beendet aktuellen Prozess mit angegebenem Exitstatus
- Gibt alle Ressourcen frei, die der Prozess belegt hat
  - Speicher
  - Dateideskriptoren
  - Prozessverwaltungsdaten
  - …
- Prozess geht in den *Zombie*-Zustand über
  - Ermöglicht Vater auf Terminieren des Kindes zu reagieren
  - Zombie-Prozesse belegen Ressourcen

  ⇒ Vaterprozess muss seine Zombies aufräumen ↝ wait(2) / waitpid(2)
- Ist der Vater schon vor dem Kind terminiert:

⇒ Weiterreichen an `init`-Prozess und von diesem weggeräumt

Programm ausführen (1)

```
01 int execl(const char *path, const char *arg0, ..., NULL);
02 int execv(const char *path, char *const argv[]);
```

- Ersetzt das aktuell ausgeführte Programm im Prozess
  - **Wird ersetzt:** Text-, Daten- und Stacksegment
  - **Bleibt erhalten:** Dateideskriptoren, Arbeitsverzeichnis, …
- Aufrufparameter für `exec(3)`
  - Pfad des neuen Programmes
  - Argumente für die `main()`-Funktion
- Statische Zahl von Argumenten: `execl(3)`
- Dynamische Zahl von Argumenten: `execv(3)`
- Letztes Argument: `NULL`-Zeiger
- `exec(3)` kehrt nur im Fehlerfall zurück

PATH Umgebungsvariable

- Finden von ausführbaren Programmen mit PATH

```
01 $> cp dat dat-copy
02 $> ls
03 dat dat-copy                    # keine Datei 'cp'
04
05 $> echo $PATH                   # PATH enthält
06 /usr/local/bin:/usr/bin:/bin    #    - /usr/local/bin/
07                                 #    - /usr/bin/
08                                 #    - /bin/
09 $> which cp
10 /bin/cp                         # 'cp' liegt also in /bin/
11
12 $> ls /bin/                     # /bin/ enthält noch viele
13 [...]                           # weitere bekannte Programme
14 rm
15 cp
16 ls
17 [...]
```

Programm ausführen (2)

```
01 int execlp(const char *file, const char *arg0, ..., NULL);
02 int execvp(const char *file, char *const argv[]);
```

- Wie `execl(3)`/`execv(3)` mit Suche in PATH

Beispiele:

```
01 // absoluter Pfad und statische Liste von Argumenten
02 execl("/bin/cp", "/bin/cp", "x.txt", "y.txt", NULL);
03
04 // Suche in PATH und statischer Liste von Argumenten
05 execlp("cp", "cp", "x.txt", "y.txt", NULL);
06
07 // Suche in PATH und dynamischer Liste von Argumenten
08 char *args[] = { "cp", "dat", ..., "copy/", NULL };
09 execvp(args[0], args);
```

Beispiel: fork(), exec() und wait()

```
01 static void die(const char *reason) {
02     perror(reason); exit(EXIT_FAILURE);
03 }
04
05 // [...] Prozess läuft
06 pid_t res = fork();
07 if(res > 0) { // Vaterprozess
08     int status;
09     pid_t term_pid = wait(&status);
10     if(term_pid == -1) { // Fehler in wait()
11         die("wait");
12     } else {
13         printf("Child %d terminated\n", term_pid);
14     }
15 } else if(res == 0) { // Kindprozess
16     execlp("cp", "cp", "dat", "dat-copy", NULL);
17     // Fehler in execlp()
18     die("execlp");
19 } else {  // Fehler -- Kein Kindprozess erzeugt
20     die("fork");
21 }
```