

Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2024

Übung 10

Maxim Ritter von Onciul
Arne Vogel

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



Lehrstuhl für Informatik 4
Systemsoftware



Friedrich-Alexander-Universität
Technische Fakultät

Vorstellung Aufgabe 6

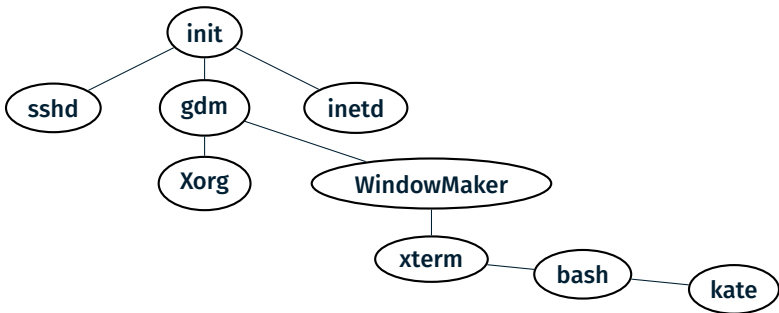
Prozesse



- Prozesse sind eine Ausführungsumgebung 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)`, `ps tree(1)`, `htop(1)`



- 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**



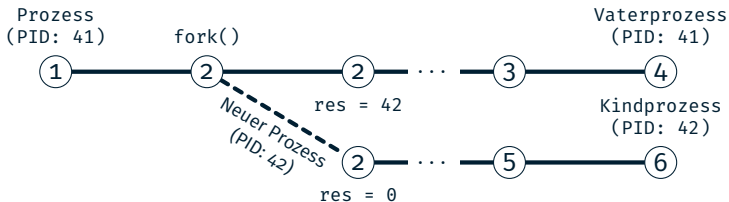
```
01 pid_t fork(void);
```

- 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)



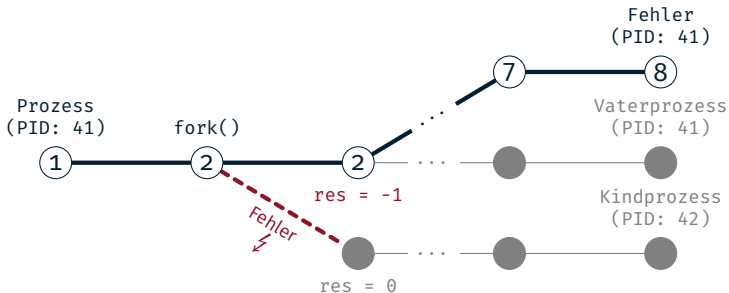
```
01 printf("Prozess (PID: %d)", getpid());
02 pid_t res = fork();
03 if(res > 0) {
04     printf("Vaterprozess (PID: %d)", getpid());
05 } else if(res == 0) {
06     printf("Kindprozess (PID: %d)", getpid());
07 } else {
08     printf("Fehler (PID: %d)", getpid());
09     // [...] Fehlerbehandlung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



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 }
```





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



```
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
- Ist der Vater schon vor dem Kind terminiert:
⇒ Weiterreichen an `init`-Prozess und von diesem weggeräumt



```
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



■ 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 [...]
```



```
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);
```

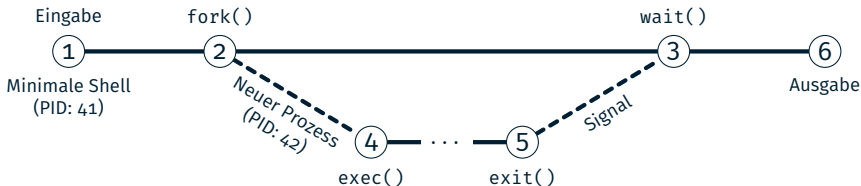


```
01 static void die(const char *reason) {
02     perror(reason); exit(EXIT_FAILURE);
03 }
04
05 // [...] Prozess läuft
06 pid_t res = fork();
07 if(res > 0) { // Vaterprozess
08     int status;
09     pid_t term_pid = wait(&status);
10     if(term_pid == -1) { // Fehler in wait()
11         die("wait");
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(3)
18     die("execlp");
19 } else { // Fehler -- Kein Kindprozess erzeugt
20     die("fork");
21 }
```

Minimale Shell



1. Auf Eingaben vom Benutzer warten
2. Neuen Prozess erzeugen
3. Vater: Wartet auf die Beendigung des Kindes
4. Kind: Startet Programm
5. Kind: Programm terminiert
6. Vater: Ausgabe der Kindzustands





```
01 char *fgets(char *s, int size, FILE *stream);
```

- fgets(3) liest eine Zeile vom übergebenen Kanal
- '\n' wird mitgespeichert
- Maximal size-1 Zeichen + finales '\0'
- Im Fehlerfall oder EOF wird NULL zurückgegeben

⇒ Unterscheidung ferror(3) oder feof(3)

```
01 char buf[23];
02 while (fgets(buf, 23, stdin) != NULL) {
03     // buf enthält Zeile
04 }
05
06 if(ferror(stdin)) { // Fehler
07     [...]
08 }
```

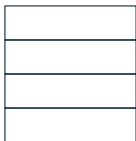


```
01 char *strtok(char *str, const char *delim);
```

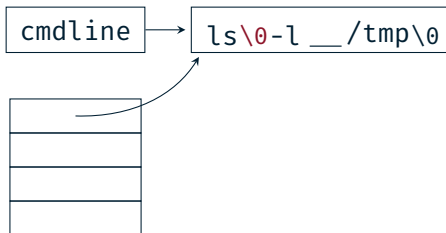
- strtok(3) teilt einen String in Tokens auf
- Tokens werden durch Trennzeichen getrennt
- Liefert bei jedem Aufruf Zeiger auf nächsten Token
- delim: String, der alle Trennzeichen enthält (z.B. " \t\n")
- str:
 - **erster Aufruf** Zeiger auf zu teilenden String
 - **alle Folgeaufrufe** NULL
- Aufeinanderfolgende Trennzeichen werden übersprungen
- Trennzeichen nach Token werden durch '\0' ersetzt
- Am Ende des Strings: strtok(3) gibt NULL zurück



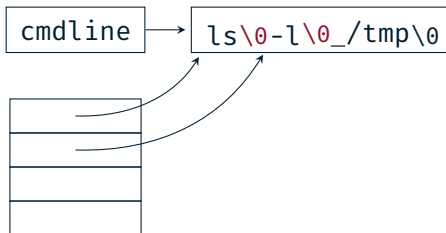
cmdline → ls -l __/tmp\0



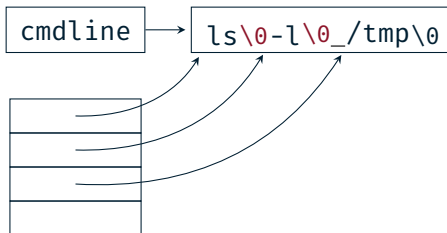
```
01 char cmdline[] = "ls -l /tmp";  
02 char *a[4];  
03 a[0] = strtok(cmdline, " ");  
04 a[1] = strtok(NULL, " ");  
05 a[2] = strtok(NULL, " ");  
06 a[3] = strtok(NULL, " ");
```



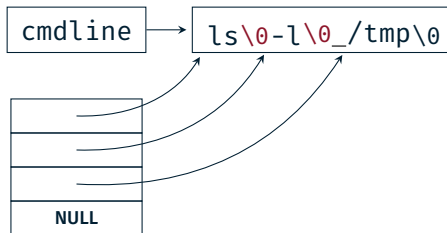
```
01 char cmdline[] = "ls -l /tmp";  
02 char *a[4];  
03 a[0] = strtok(cmdline, " ");  
04 a[1] = strtok(NULL, " ");  
05 a[2] = strtok(NULL, " ");  
06 a[3] = strtok(NULL, " ");
```



```
01 char cmdline[] = "ls -l /tmp";
02 char *a[4];
03 a[0] = strtok(cmdline, " ");
04 a[1] = strtok(NULL, " ");
05 a[2] = strtok(NULL, " ");
06 a[3] = strtok(NULL, " ");
```



```
01 char cmdline[] = "ls -l /tmp";  
02 char *a[4];  
03 a[0] = strtok(cmdline, " ");  
04 a[1] = strtok(NULL, " ");  
05 a[2] = strtok(NULL, " ");  
06 a[3] = strtok(NULL, " ");
```



```
01 char cmdline[] = "ls -l /tmp";  
02 char *a[4];  
03 a[0] = strtok(cmdline, " ");  
04 a[1] = strtok(NULL, " ");  
05 a[2] = strtok(NULL, " ");  
06 a[3] = strtok(NULL, " ");
```


Aufgabe: mish

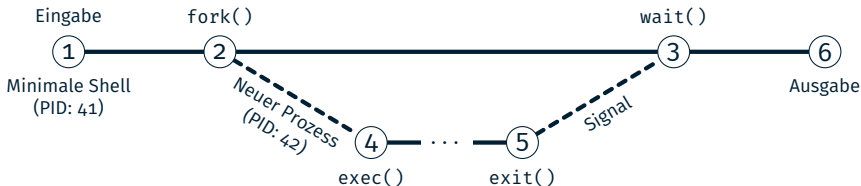


- Einfache Shell (**mini shell**) zum Ausführen von Kommandos
- Typischer Ablauf:
 - Ausgabe des Prompts
 - Warten auf Eingaben
 - Zerlegen der Eingaben
 - Kommandoname
 - Argumente
 - Neuen Prozess erstellen
 - **Vater**: Warten auf Terminierung des Kindes
 - **Kind**: Ausführen des Kommandos
 - Ausgabe des Exitstatus



■ Wiederholung: Basisablauf einer minimalen Shell

1. Auf Eingaben vom Benutzer warten
2. Neuen Prozess erzeugen
3. Vater: Wartet auf die Beendigung des Kindes
4. Kind: Startet Programm
5. Kind: Programm terminiert
6. Vater: Ausgabe der Kindzustands





Beispiele:

```
01 # Reguläre Beendigung durch Exit (Exitstatus = 0)
02 mish> ls -l
03 ...
04 Exit status [2110] = 0
05
06 # Ungültige/Leere Eingaben
07 mish>
08 mish> foo
09 foo: No such file or directory
10 Exit status [7342] = 1
11
12 # Beendigung durch Signal (hier SIGINT = 2)
13 mish> sleep 10
14 Signal [1302] = 2
```



- Prompt druckt kein '\n'
 - Standardbibliothek puffert stdout zeilenweise
- ⇒ Nach Ausgabe den Zeilenpuffers mittels fflush(3) leeren



- Testprogramme: /proj/i4spic/<idm>/pub/aufgabe8/
- spic-wait (ohne Parameter)

```
01 mish> /proj/i4spic/[...]/spic-wait
02 [...]
03 - send 'SIGPIPE' to this process
04 Command: kill -PIPE 3372
05 Expected Output: Signal [3372] = 13
06
07 [...]
08
09 Signal [3372] = 13
10 mish>
```

```
01
02
03
04
05
06
07
08 $> kill -PIPE 3372
09
10
```

- spic-wait (mit Parameter)

```
01 mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-wait 15
02 Sending signal 15 (Terminated) to myself (PID: 4239)
03 Signal [4239] = 15
04 mish>
```



- Testprogramme: /proj/i4spic/<idm>/pub/aufgabe8/
- spic-wait (ohne Parameter)

```
01 mish> /proj/i4spic/[...]/spic-wait
02 [...]
03 - send 'SIGPIPE' to this process
04   Command: kill -PIPE 3372
05   Expected Output: Signal [3372] = 13
06
07 [...]
08
09 Signal [3372] = 13
10 mish>
```

```
01
02
03
04
05
06
07
08 $> kill -PIPE 3372
09
10
```

- spic-wait (mit Parameter)

```
01 mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-wait 15
02 Sending signal 15 (Terminated) to myself (PID: 4239)
03 Signal [4239] = 15
04 mish>
```



- Testprogramme: /proj/i4spic/<idm>/pub/aufgabe8/
- spic-wait (ohne Parameter)

```
01 mish> /proj/i4spic/[...]/spic-wait
02 [...]
03 - send 'SIGPIPE' to this process
04   Command: kill -PIPE 3372
05   Expected Output: Signal [3372] = 13
06
07 [...]
08
09 Signal [3372] = 13
10 mish>
```

```
01
02
03
04
05
06
07
08 $> kill -PIPE 3372
09
10
```

- spic-wait (mit Parameter)

```
01 mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-wait 15
02 Sending signal 15 (Terminated) to myself (PID: 4239)
03 Signal [4239] = 15
04 mish>
```




■ spic-exit

```
01 mish> /proj/i4spic/<idm>/pub/aufgabe8/spic-exit 12
02 Exiting with status 12
03 Exit status [6272] = 12
04 mish>
```



```
01 // DESCRIPTION:
02 //   printStatus() examines the termination of a process and
03 //   prints the source of the exit (signal or exit) and the
04 //   exit code or signal number, respectively.
05 //
06 // PARAMETER:
07 //   pid:   PID of the exited child process
08 //   status: Status bits as retrieved from waitpid(2)
09 //
10 static void printStatus(pid_t pid, int status) {
11     // TODO IMPLEMENT
12 }
```

- `/proj/i4spic/<idm>/pub/aufgabe8/mish_vorlage.c`
- Die Vorlage enthält jedoch **nicht**:
 - Alle Funktionen, Funktionalitätsbeschreibungen, Variablen etc.
- Vorlage ersetzt nicht eigenständiges Nachdenken zur Struktur
- Während der Entwicklung kann es sinnvoll sein, das `Werror` Flag im Makefile wegzulassen

Hands-on: run

Screencast: <https://www.video.uni-erlangen.de/clip/id/19832>



```
01 ./run <programm> <param0> [params...]
```

- run erhält einen Programmnamen und eine Liste mit Parametern
 - Erstellt für jeden Parameter einen neuen Prozess
 - Führt das angegebene Programm aus und übergibt den zugehörigen Parameter
 - Wartet auf dessen Beendigung und behandelt nächsten Parameter
- Aufrufbeispiel: `./run echo Auto Haus Katze`
- Generierte Programmaufrufe:
 - `echo Auto`
 - `echo Haus`
 - `echo Katze`
- (System-) Aufrufe: `fork(2)`, `exec(3)`, `wait(2)`
- Fehlerbehandlung beachten