Übungen zu Systemnahe Programmierung in C (SPiC)

Moritz Strübe, Rainer Müller
(Lehrstuhl Informatik 4)

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Sommersemester 2014

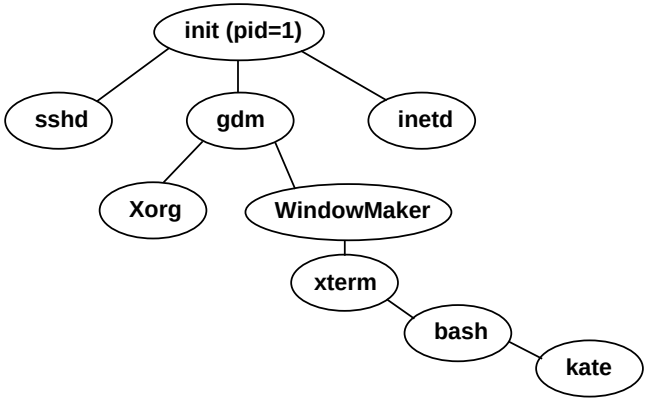
Inhalt

Prozesse

- Prozesse sind eine Ausführumgebung für Programme
  - haben eine Prozess-ID (PID, ganzzahlig positiv)
  - führen ein Programm aus
- Mit einem Prozess sind Ressourcen verknüpft, z.B.
  - Speicher
  - Adressraum
  - offene Dateien

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**

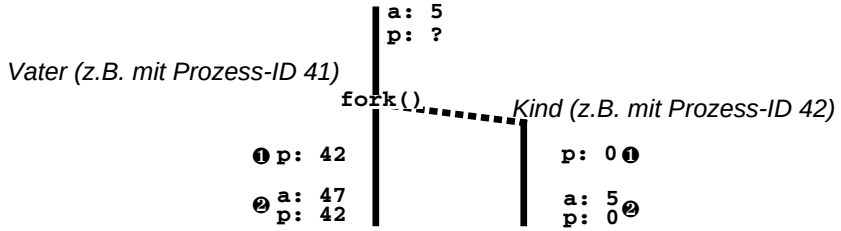
Kindprozess erzeugen – fork(2)

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

- Erzeugt einen neuen Kindprozess
- Exakte Kopie des Vaters...
  - Datensegment (neue Kopie, gleiche Daten)
  - Stacksegment (neue Kopie, gleiche Daten)
  - Textsegment (gemeinsam genutzt, da nur lesbar)
  - Filedeskriptoren (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(2)`) entscheiden, ob es sich um den Vater- oder den Kindprozess handelt

Kindprozess erzeugen – fork(2)

```
int a=5;
pid_t p = fork(); // (1)
a += p;           // (2)
switch(p) {
    case -1: // Fehler - kein Kind
        ...
    case 0: //  Kind
        ...
   default: // Vater
        ...
}
```



Programm ausführen – exec(3)

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

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

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

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

- Lädt Programm zur Ausführung in den aktuellen Prozess
  - aktuell ausgeführtes Programm wird ersetzt (Text-, Daten- und Stacksegment)
  - erhalten bleiben: Filedeskriptoren (= geöffnete Dateien), Arbeitsverzeichnis, ...
- Aufrufparameter für `exec(3)`
  - Pfad bzw. Dateiname des neuen Programmes
  - Argumente für die `main`-Funktion

Beispiele zu exec(3)

- Mit absolutem Pfad und einer statischen Liste

```
execl("/bin/cp", "/bin/cp", "x.txt", "y.txt", NULL);
```

- Mit Suche in `PATH` und einer statischen Liste

```
execlp("cp", "cp", "x.txt", "y.txt", NULL);
```

- Mit Suche in `PATH` und einer veränderbar großen Liste

```
char *args[4];
args[0] = "cp";
args[1] = "x.txt";
args[2] = "y.txt";
args[3] = NULL;
execvp(args[0], args);
```

- Anmerkungen
  - Alle Varianten von `exec(3)` erwarten als letzten Eintrag in der Argumentenliste einen `NULL`-Zeiger
  - Alle Varianten von `exec(3)` kehren nur im Fehlerfall zurück

Prozess beenden – exit(3)

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

- beendet aktuellen Prozess mit angegebenem Exitstatus
- gibt alle Ressourcen frei, die der Prozess belegt hat, z.B.
  - Speicher
  - Filedeskriptoren (schließt alle offenen Dateien)
  - 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
  - Zombie-Prozesse belegen Ressourcen und sollten zeitnah 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

Auf Kindprozess warten – wait(2)

- Warten auf die Beendigung von Kind-Prozessen (Rückgabe: PID)

```
1 pid_t wait(int *statusbits);
```

- Beispiel

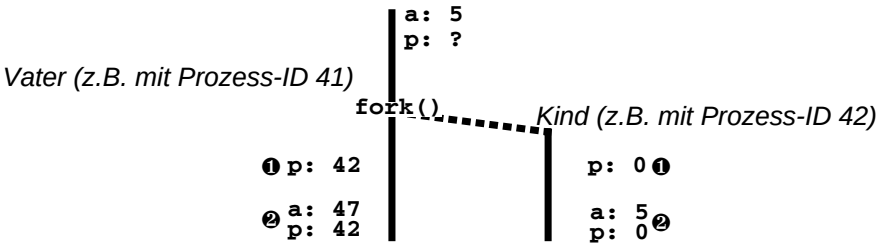
```
1  int main(int argc, char *argv[]) {
2      pid_t pid;
3      pid = fork();
4      if (pid > 0) {                                    /* Vater */
5          int stbits;
6          wait(&stbits); /* Fehlerbehandlung nicht vergessen! */
7         printf("Kindstatus: %x", stbits); /* nackte Status-Bits */
8      } else if (pid == 0) {                             /* Kind */
9          execl("/bin/cp","/bin/cp","x.txt","y.txt", NULL);
10         /* diese Stelle wird nur im Fehlerfall erreicht */
11         perror("exec /bin/cp"); exit(EXIT_FAILURE);
12     } else {
13         /* pid == -1 --> Fehler bei fork */
14     }
15 }
```

Auf Kindprozess warten – wait(2)

- `wait(2)` blockiert, bis ein Kind-Prozess terminiert wird
- PID dieses Kind-Prozesses wird als Rückgabewert geliefert
- als Parameter kann ein Zeiger auf einen `int`-Wert mitgegeben werden, in dem unter anderem der Exitstatus 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(3)` terminiert: `WIFEXITED(stbits)`
    - Exitstatus: `WEXITSTATUS(stbits)`
  - ⇒ Prozess durch Signal abgebrochen: `WIFSIGNALED(stbits)`
    - Nummer des Signals: `WTERMSIG(stbits)`
- weitere siehe `man 2 wait`

Funktionsweise einer minimalen Shell (Lish)

- Auf Eingaben vom Benutzer warten
- Neuen Prozess erzeugen
- Kind: Startet Programm
- Vater: Wartet auf die Beendigung des Kindes
- Ausgabe der Kindzustands



Einlesen von der Standard-Eingabe mit fgets(3)

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

- `fgets(3)` liest eine Zeile vom übergebenen Eingabe-Kanal und schreibt diese in einen vorher angelegten Speicherbereich
- Es werden maximal `size-1` Zeichen gelesen und mit `'\0'` abgeschlossen
- Das `'\n'` am Ende der Zeile wird auch gespeichert
- Rückgabewert ist der Zeiger auf den übergebenen Speicherbereich; oder `NULL` am Ende der Eingabe oder im Fehlerfall
  - Unterscheidung zwischen End-Of-File und Fehler muss mittels `feof(3)` oder `ferror(3)` erfolgen
- Beispiel

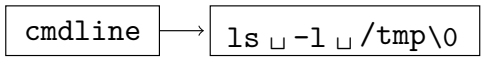
```
1 char buf[23];
2 while (fgets(buf, 23, stdin) != NULL) { /* Fehlerüberprüfung! */
3     /* buf enthält die eingelesene Zeile */
4 }
```

Stringmanipulation mit strtok(3)

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

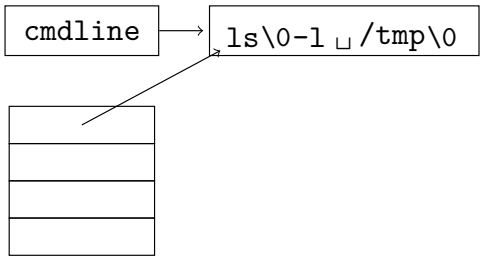
- `strtok(3)` teilt einen String in Tokens auf, die durch bestimmte Trennzeichen getrennt sind
- Wird sukzessive aufgerufen und liefert jeweils einen Zeiger auf das nächste Token (mehrere aufeinanderfolgende Trennzeichen werden hierbei übersprungen)
  - `str` ist im ersten Aufruf ein Zeiger auf den zu teilenden String, in allen Folgeaufrufen *NULL*
  - `delim` ist ein String, der alle Trennzeichen enthält, z.B. `" \t\n"`
- Bei jedem Aufruf wird das einem Token folgende Trennzeichen durch `'\0'` ersetzt
- Ist das Ende des Strings erreicht, gibt `strtok(3)` `NULL` zurück

Stringmanipulation mit strtok(3)



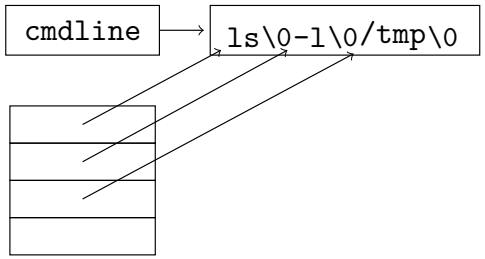

```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```

Stringmanipulation mit strtok(3)



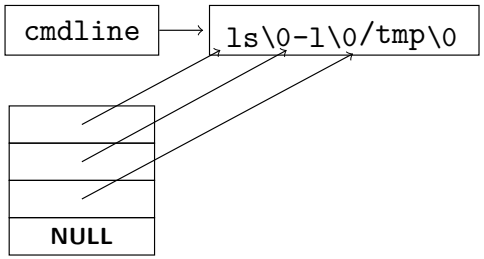

```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```

Stringmanipulation mit strtok(3)




```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```

Stringmanipulation mit strtok(3)




```
1 a[0] = strtok(cmdline, " ");
2 a[1] = strtok(NULL, " ");
3 a[2] = strtok(NULL, " ");
4 a[3] = strtok(NULL, " ");
```

Testprogramme

- Unter /proj/i4spic/pub/aufgabe7
  - spic-wait

```
1 /proj/i4spic/pub/aufgabe7/spic-wait
2 My PID: 20746
3 Try
4  kill 20746
5        to terminate without core dump (SIGTERM)
6  kill -QUIT 20746
7        to terminate with core dump (SIGQUIT)
```

  - spic-exit

```
1 /proj/i4spic/pub/aufgabe7/spic-exit 12
2 Exiting with status 12
```