G 5. Übung

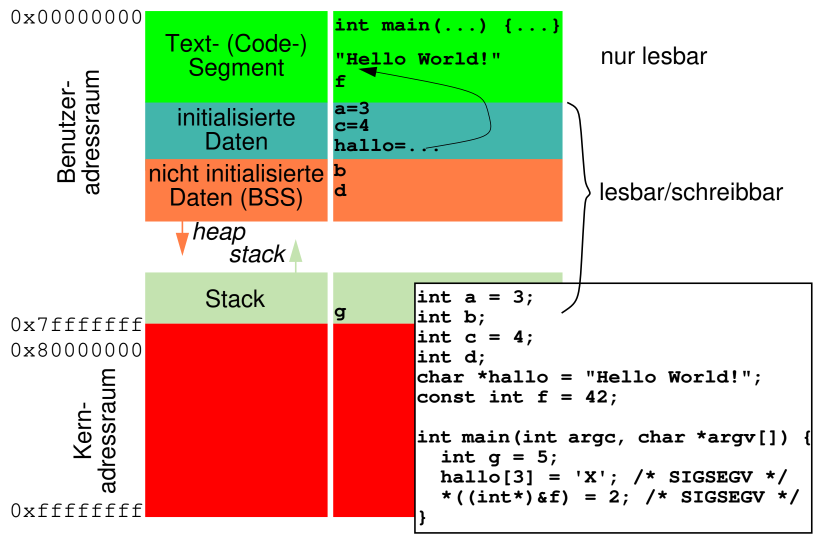
G-1 Überblick

- Besprechung 3. Aufgabe
- Infos zur Aufgabe 5: fork, exec
- Rechenzeiterfassung

G-2 Hinweise zur 5. Aufgabe

- Prozesse
- fork, exec
- exit
- wait

G-3 Aufbau der Daten eines Prozesses

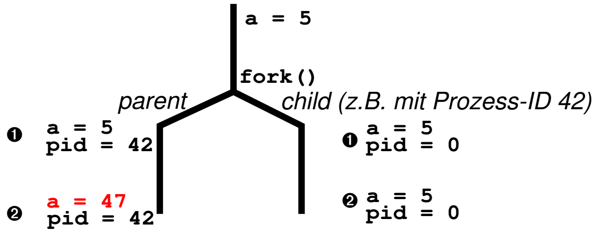


G-4 fork

- Vererbung von
  - Datensegment (neue Kopie, gleiche Daten)
  - Stacksegment (neue Kopie, gleiche Daten)
  - Textsegment (gemeinsam genutzt, da nur lesbar)
  - Filedeskriptoren
  - Arbeitsverzeichnis
  - Benutzer- und Gruppen-ID (uid, gid)
  - Umgebungsvariablen
  - Signalbehandlung
  - ...
- Neu:
  - Prozess-ID

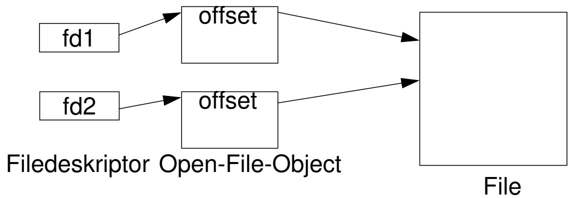
G-4 fork

```
int a;
a = 5;
pid_t pid = fork();
❶
a += pid; ❷
if (pid == 0) {
    ...
} else {
    ...
}
```

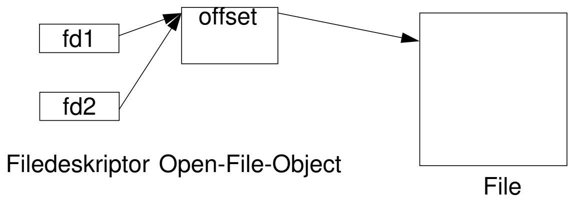


G-5 fork und Filedeskriptoren

- erneutes Öffnen eines Files



- bei fork werden FD vererbt, aber Files werden nicht neu geöffnet!



G-6 exec

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

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


G-6 exec 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 execv(const char *path, char *const argv[]);
```

- mit Umgebungsvariablen in `envp`

```
int execle(const char *path,char *const arg0, ... , const char
*argn, char * /*NULL*/, char *const envp[]);

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

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

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

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

G-7 exit

- beendet aktuellen Prozess
- 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)

G-8 wait

- warten auf Statusinformationen von Kind-Prozessen (Rückgabe: PID)
  - **wait(int *status)**
  - **waitpid(pid_t pid, int *status, int options)**
- Beispiel:

```
int main(int argc, char *argv[]) {
  int pid;
  if ((pid=fork()) > 0) {
    /* parent */
    int status;
    wait(&status); /* ... Fehlerabfrage */
    printf("Kindstatus: %d", status);
  } else if (pid == 0) {
    /* child */
    execl("/bin/cp", "cp", "/etc/passwd", "/tmp/passwd", 0);
      /* diese Stelle wird nur im Fehlerfall erreicht */
  } else {
      /* pid == -1 --> Fehler bei fork */
  }
}
```


G-9 Rechenzeiterfassung

- Betriebssystem erfasst die Rechenzeit der Prozesse
  - ➤ user time: Rechenzeit im Benutzermodus
  - ➤ system time: Rechenzeit im Systemkern (priv. Modus)
- für jeden Prozess wird außerdem die Rechenzeit aller Kind-Prozesse aufsummiert
  - ➤ nach Terminieren eines Kind-Prozesses bleiben die Daten in Datenstruktur des ZOMBIE-Prozesses gespeichert
  - ➤ bei **wait()/waitpid()** werden die Daten in den Vaterprozess übernommen
- Rechenzeiten werden in clock ticks angegeben (meist 10 ms)
  - ➤ clock ticks/Sekunde kann durch das Makro **CLK_TCK** abgefragt werden

G-9 Rechenzeiterfassung (2)

- Prototyp

```
#include <sys/times.h>

clock_t times(struct tms *buf);
```

- tms Datenstruktur

```
struct  tms  {
        clock_t tms_utime;  /* user time */
        clock_t tms_stime;  /* system time */
        clock_t tms_cutime; /* user time of children */
        clock_t tms_cstime; /* system time of children */
}
```