

# F 4. Übung

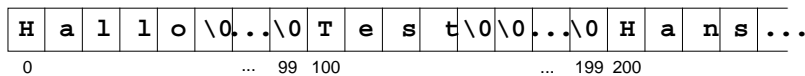
## F.1 Überblick

- Aufgabe 2: qsort - Fortsetzung
- Infos zur Aufgabe 4: fork, exec

## F.2 Aufgabe 2: Sortieren mittels qsort (Fortsetzung)

### 1 wsort - Datenstrukturen (1. Möglichkeit)

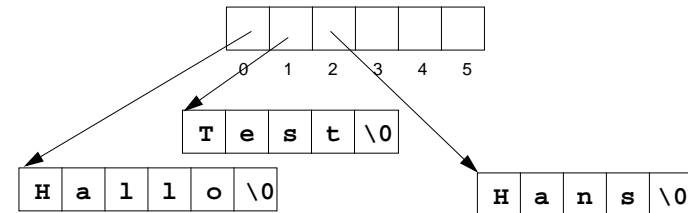
- Array von Zeichenketten



- Vorteile:
  - ◆ einfach
- Nachteile:
  - ◆ hoher Kopieraufwand
  - ◆ Maximale Länge der Worte muß bekannt sein
  - ◆ Verschwendung von Speicherplatz

## 2 wsort - Datenstrukturen (2. Möglichkeit)

- Array von Zeigern auf Zeichenketten



- Vorteile:
  - ◆ schnelles Sortieren, da nur Zeiger vertauscht werden müssen
  - ◆ Zeichenketten können beliebig lang sein
  - ◆ sparsame Speichernutzung

### 3 Speicherverwaltung

- Berechnung des Array-Speicherbedarfs
  - ◆ bei Lösung 1: Anzahl der Wörter \* 100 \* sizeof(char)
  - ◆ bei Lösung 2: Anzahl der Wörter \* sizeof(char\*)
- realloc:
  - ◆ Anzahl der zu lesenden Worte ist unbekannt
  - ◆ Array muß vergrößert werden: realloc
  - ◆ Bei Vergrößerung sollte man aus Effizienzgründen nicht nur Platz für ein neues Wort (Lösungsvariante 1) bzw. einen neuen Zeiger (Lösungsvariante 2) besorgen, sondern für mehrere.
  - ◆ Achtung: realloc kopiert möglicherweise das Array (teuer)
- Speicher sollte wieder freigegeben werden
  - ◆ bei Lösung 1: Array freigeben
  - ◆ bei Lösung 2: zuerst Wörter freigeben, dann Zeiger-Array freigeben

## 4 Vergleichsfunktion

- Problem: qsort erwartet folgenden Funktionstyp:

```
int (*compar) (const void *, const void *)
```

- Lösung: "casten"

- ◆ innerhalb der Funktion, z.B. (Feld vom Typ char \*\*):

```
int compare(const void *a, const void *b) {
    return strcmp(*(char **)a, *(char **)b);
}
```

- ◆ beim qsort-Aufruf:

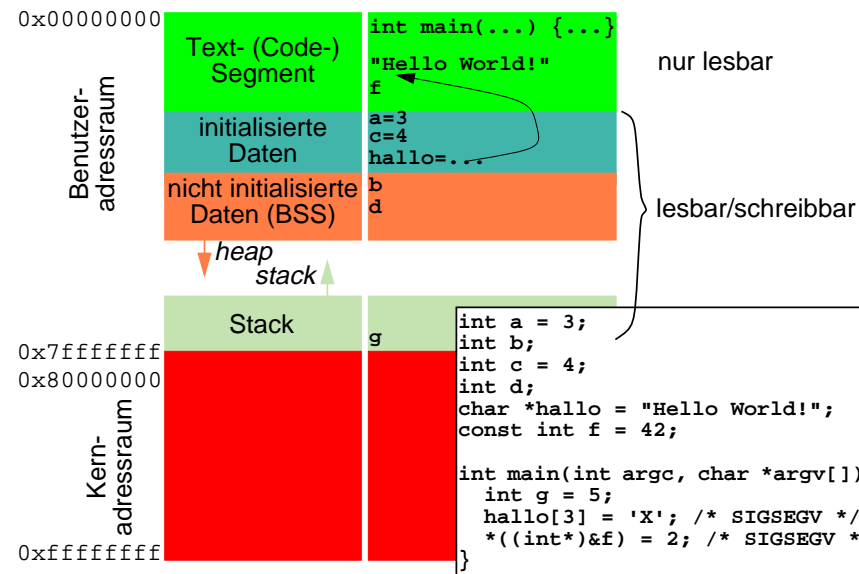
```
int compare(char **a, char **b);
...
qsort( field, nel, sizeof(char *),
      (int (*)(const void *, const void *))compare);
```

## F.3 Hinweise zur 4. Aufgabe

F.3 Hinweise zur 4. Aufgabe

- Prozesse
- fork, exec
- exit
- wait

## F.4 Aufbau der Daten eines Prozesses



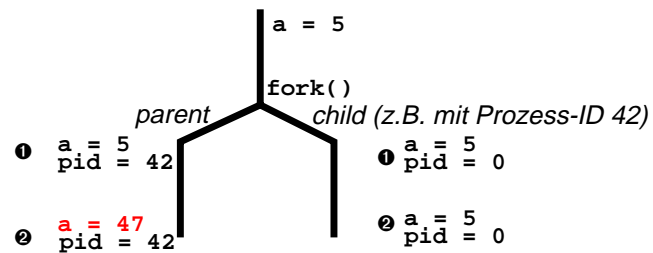
## F.5 fork

F.5 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

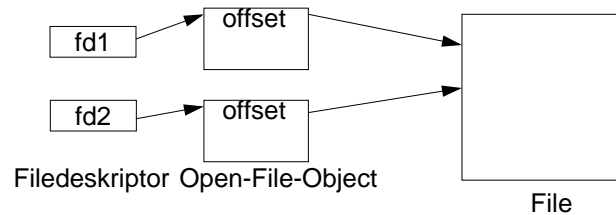
## F.5 fork

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

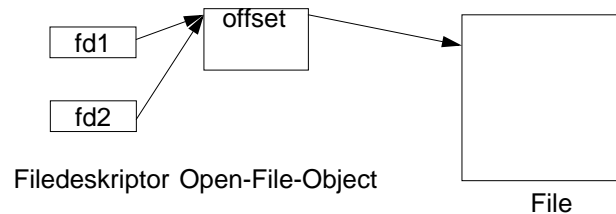


## F.6 fork und Filedeskriptoren

- erneutes Öffnen eines Files



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



## F.7 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);
```

## F.7 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 execlp(const char *path, char *const argv[]);
```

- mit Umgebungsvariablen in `envp`

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

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

## F.8 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)

## F.9 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 */
    }
}
```