

23 Überblick über die 4. Übung

- Dateisystem: Systemaufrufe
- Aufgabe 2: qsort
- Infos zur Aufgabe 4: fork, exec

24 Dateisystem Systemcalls

- open / close
- read / write
- lseek
- chmod
- umask
- utime
- truncate

24.1 open

■ Funktions-Prototyp:

```
#include <fcntl.h>
int open(const char *path, int oflag, ... /* [mode_t mode] */ );
```

■ Argumente:

- ◆ Maximallänge von path: **PATH_MAX**
- ◆ **oflag**: Lese/Schreib-Flags, Allgemeine Flags, Synchronisierungs I/O Flags
 - Lese/Schreib-Flags: **O_RDONLY**, **O_WRONLY**, **O_RDWR**
 - Allgemeine Flags: **O_APPEND**, **O_CREAT**, **O_EXCL**, **O_LARGEFILE**, **O_NDELAY**, **O_NOCTTY**, **O_NONBLOCK**, **O_TRUNC**
 - Synchronisierung: **O_DSYNC**, **O_RSYNC**, **O_SYNC**
- ◆ **mode**: Zugriffsrechte der erzeugten Datei (nur bei **O_CREAT**) - siehe **chmod**

■ Rückgabewert

- ◆ Filedeskriptor oder -1 im Fehlerfall (**errno** wird gesetzt)

24.1 open - Flags

- **O_EXCL**: zusammen mit **O_CREAT** - nur *neue* Datei anlegen
- **O_TRUNC**: Datei wird beim Öffnen auf 0 Bytes gekürzt
- **O_APPEND**: vor jedem Schreiben wird der Dateizeiger auf das Dateiende gesetzt
- **O_NDELAY**, **O_NONBLOCK**: Operationen arbeiten nicht-blockierend (bei Pipes, FIFOs und Devices)
 - ◆ open kehrt sofort zurück
 - ◆ read liefert -1 zurück, wenn keine Daten verfügbar sind
 - ◆ wenn genügend Platz ist, schreibt write alle Bytes, sonst schreibt write nichts und kehrt mit -1 zurück
- **O_NOCTTY**: beim Öffnen von Terminal-Devices wird das Device nicht zum Kontroll-Terminal des Prozesses

24.1 open Flags (2)

■ Synchronisierung

- ◆ **O_DSYNC**: Schreibaufruf kehrt erst zurück, wenn Daten in Datei geschrieben wurden (Blockbuffer Cache!!)
- ◆ **O_SYNC**: ähnlich **O_DSYNC**, zusätzlich wird gewartet, bis Datei-Attribute wie Zugriffszeit, Modifizierungszeit, auf Disk geschrieben sind
- ◆ **O_RSYNC** | **O_DSYNC**: Daten die gelesen wurden, stimmen mit Daten auf Disk überein, d.h. vor dem Lesen wird der Buffercache geflushet
- ◆ **O_RSYNC** | **O_SYNC**: wie **O_RSYNC** | **O_DSYNC**, zusätzlich Datei-Attribute

24.2 close

■ Funktions-Prototyp:

```
#include <unistd.h>
int close(int fildes);
```

■ Argumente:

- ◆ **fildes**: Filedeskriptor der zu schließenden Datei

■ Rückgabewert:

- ◆ 0 bei Erfolg, -1 im Fehlerfall

24.3 read

■ Funktions-Prototyp:

```
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
```

■ Argumente

- ◆ **fildes**: Filedeskriptor, z.B. Rückgabe vom open-Aufruf
- ◆ **buf**: Zeiger auf Puffer
- ◆ **nbyte**: Größe des Puffers

■ Rückgabewert

- ◆ Anzahl der gelesenen Bytes oder -1 im Fehlerfall

```
char buf[1024];
int fd;
fd = open("/etc/passwd", O_RDONLY);
if (fd == -1) ...
read(fd, buf, 1024);
```

24.4 write

■ Funktions-Prototyp

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

■ Argumente

- ◆ äquivalent zu **read**

■ Rückgabewert

- ◆ Anzahl der geschriebenen Bytes oder -1 im Fehlerfall

24.5 lseek

■ Funktions-Prototyp

```
#include <unistd.h>
off_t lseek(int fildes, off_t offset, int whence);
```

■ Argumente

- ◆ **fildes**: Filedeskriptor
- ◆ **offset**: neuer Wert des Dateizeigers
- ◆ **whence**: Bedeutung von offset
 - **SEEK_SET**: absolut vom Dateianfang
 - **SEEK_CUR**: Inkrement vom aktuellen Stand des Dateizeigers
 - **SEEK_END**: Inkrement vom Ende der Datei

■ Rückgabewert

- ◆ Offset in Bytes vom Beginn der Datei oder -1 im Fehlerfall

24.6 chmod

■ Funktions-Prototyp:

```
#include <sys/stat.h>
int chmod(const char *path, mode_t mode);
```

■ Argumente:

- ◆ **path**: Dateiname
- ◆ **mode**: gewünschter Dateimodus, z.B.
 - **S_IRUSR**: lesbar durch Besitzer
 - **S_IWUSR**: schreibbar durch Benutzer
 - **S_IRGRP**: lesbar durch Gruppe

■ Rückgabewert: 0 wenn OK, -1 wenn Fehler

■ Beispiel:

```
chmod("/etc/passwd", S_IRUSR | S_IRGRP);
```

24.7 umask

■ Funktions-Prototyp:

```
#include <sys/stat.h>
mode_t umask(mode_t cmask);
```

■ Argumente

- ◆ **cmask**: gibt Permission-Bits an, die beim Erzeugen einer Datei ausgeschaltet werden sollen

■ Rückgabewert

- ◆ voriger Wert der Maske

24.8 utime

■ Funktions-Prototyp:

```
#include <utime.h>
int utime(const char *path, const struct utimbuf *times);
```

■ Argumente

- ◆ **path**: Dateiname
- ◆ **times**: Zugriffs- und Modifizierungszeit (in Sekunden)

■ Rückgabewert: 0 wenn OK, -1 wenn Fehler

■ Beispiel: setze atime und mtime um eine Stunde zurück

```
struct utimbuf times;
struct stat buf;
stat("/etc/passwd", &buf); /* Fehlerabfrage */
times.actime = buf.st_atime - 60 * 60;
times.modtime = buf.st_mtime - 60 * 60;
utime("/etc/passwd", &times); /* Fehlerabfrage */
```

24.9 truncate

■ Funktions-Prototyp:

```
#include <unistd.h>
int truncate(const char *path, off_t length);
```

■ Argumente:

- ◆ **path**: Dateiname
- ◆ **length**: gewünschte Länge der Datei

■ Rückgabewert: 0 wenn OK, -1 wenn Fehler

24.10 POSIX I/O vs. Standard-C-I/O

■ POSIX Funktionen open/close/read/write/... arbeiten mit Filedescriptoren

■ Standard-C Funktionen fopen/fclose/fgets/... arbeiten mit Filepointern

■ Konvertierung von Filepointer nach Filedescriptor

```
#include <stdio.h>
int fileno(FILE *stream);
```

■ Konvertierung von Filedescriptor nach Filepointer

```
#include <stdio.h>
FILE *fdopen(int fd, const char* type);
```

- ◆ type kann sein "r", "w", "a", "r+", "w+", "a+"
(fd muß entsprechend geöffnet sein!)

■ Filedescriptoren in <unistd.h>:

STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO

25 Aufgabe2: Sortieren mittels qsort

■ Prototyp aus stdlib.h:

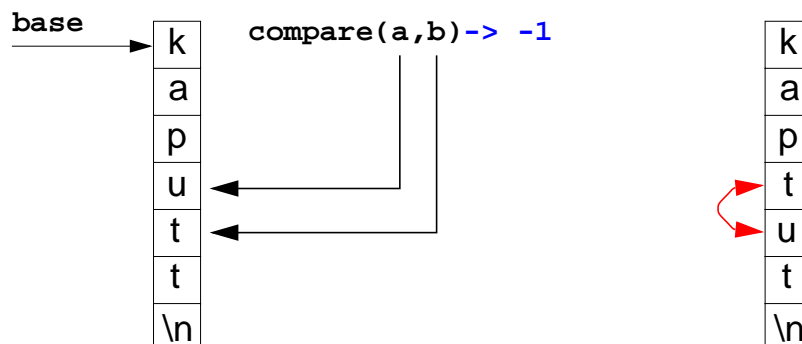
```
void qsort(void *base,
           size_t nel,
           size_t width,
           int (*compare) (const void *, const void *));
```

■ Bedeutung der Parameter:

- ◆ **base** : Zeiger auf das erste Element des Feldes, dessen Elemente sortiert werden sollen
- ◆ **nel** : Anzahl der Elemente im zu sortierenden Feld
- ◆ **width**: Größe eines Elements
- ◆ **compare**: Vergleichsfunktion

25 Sortieren mittels qsort (2)

- ◆ **qsort** vergleicht je zwei Elemente mit Hilfe der Vergleichsfunktion compare
- ◆ sind die Elemente zu vertauschen, dann werden die entsprechenden Felder komplett ausgetauscht, z.B.:



25.1 Vergleichsfunktion

- Die Vergleichsfunktion erhält Zeiger auf Feldelemente, d.h. die übergebenen Zeiger haben denselben Typ wie das Feld
- Die Funktion vergleicht die beiden Elemente und liefert:
 - <0 , falls Element 1 kleiner bewertet wird als Element 2
 - 0 , falls Element 1 und Element 2 gleich gewertet werden
 - >0 , falls Element 1 größer bewertet wird als Element 2

■ Beispiel:

- ◆ 'z', 'a' -> 1
- ◆ 1, 5 -> -1
- ◆ 5,5 -> 0

25.1 wsort - Datenstrukturen (1. Möglichkeit)

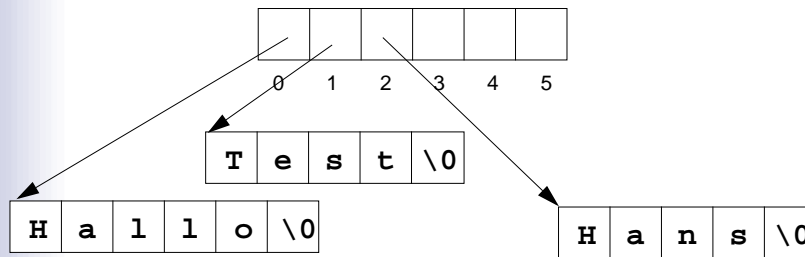
- Array von Zeichenketten

H	a	l	l	o	\0	...	\0	T	e	s	t	\0	...	\0	H	a	n	s	...
0						...		99	100				...		199	200			

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

25.2 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

25.2 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

25.2 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);
}
```

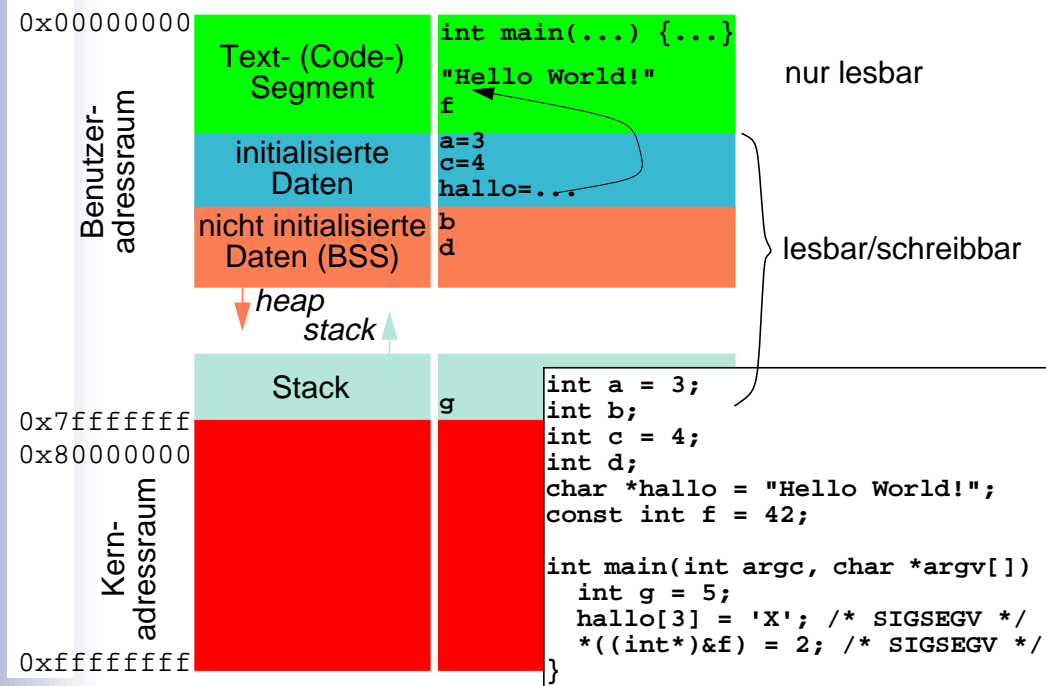
◆ beim qsort-Aufruf:

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

26 Hinweise zur 4. Aufgabe

- Prozesse
- fork, exec
- exit
- wait

26.1 Aufbau der Daten eines Prozesses



26.1 fork

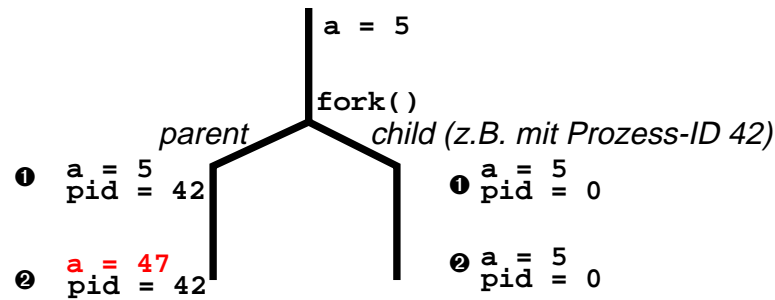
- Vererbung von
 - ◆ Datensegment (neue Kopie, gleiche Daten)
 - ◆ Stacksegment (neue Kopie, gleiche Daten)
 - ◆ Textsegment (gemeinsam genutzt, da nur lesbar)
 - ◆ Filedeskriptoren
 - ◆ Arbeitsverzeichnis
 - ◆ Benutzer- und Gruppen-ID (uid, gid)
 - ◆ Umgebungsvariablen
 - ◆ Signalbehandlung
 - ◆ ...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

26.2 fork

```

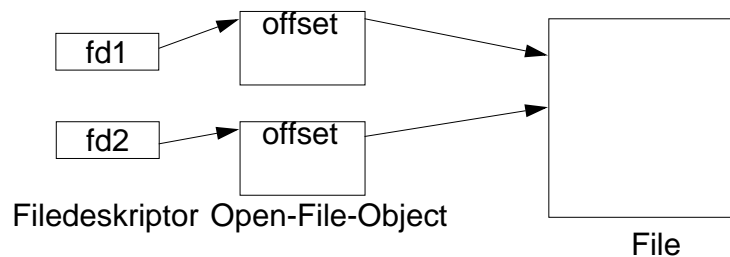
int a;
a = 5;
pid_t pid = fork();
❶
a += pid; ❷
if (pid == 0) {
    ...
} else {
    ...
}

```

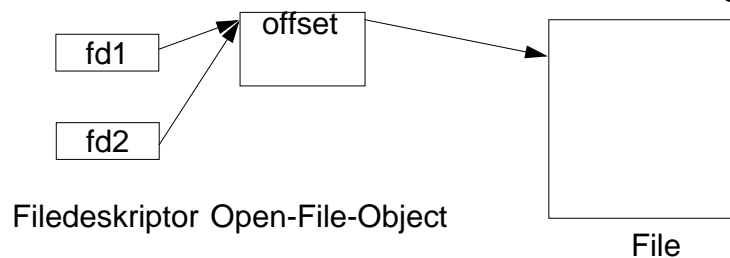


26.3 fork und Filedeskriptoren

- erneutes Öffnen eines Files



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



26.4 exec

- Lädt Programm zur Ausführung in den aktuellen Prozeß
- ersetzt Text-, Daten- und Stacksegment
- behält: Filedesktoren, 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);
```

26.4 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 execl(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[]);
```

26.5 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)

26.6 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 */
    }
}
```