Übungen zu Systemnahe Programmierung in C (SPiC) – Sommersemester 2018

Übung 8

Benedict Herzog
Sebastian Maier

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

Lehrstuhl für Verteilte Systeme und Betriebssysteme

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Dateien & Dateikanäle

Dateikanäle

- Ein- und Ausgaben erfolgen über gepufferte Dateikanäle
- `FILE *fopen(const char *path, const char *mode);`
  - öffnet eine Datei zum Lesen oder Schreiben (je nach `mode`)
  - liefert einen Zeiger auf den erzeugten Dateikanal

`r`	Lesen
`r+`	Lesen & Schreiben
`w`	Schreiben; Datei wird ggf. erstellt oder Inhalt ersetzt
`w+`	Lesen & Schreiben; Datei wird ggf. erstellt oder Inhalt ersetzt
`a`	Schreiben am Ende der Datei; Datei wird ggf. erstellt
`a+`	Schreiben am Ende der Datei; Lesen am Anfang; Datei wird ggf. erstellt

- `int fclose(FILE *fp);`
  - schreibt ggf. gepufferte Ausgabedaten des Dateikanals
  - schließt anschließend die Datei

Ein-/Ausgaben

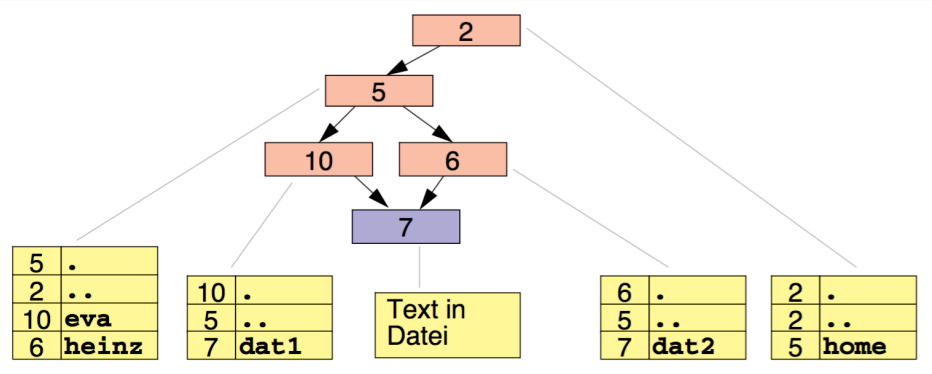
- standardmäßig geöffnete Dateikanäle

`stdin`	Eingaben
`stdout`	Ausgaben
`stderr`	Fehlermeldungen

- `int fgetc(FILE *stream);`
  - liest ein einzelnes Zeichen aus der Datei
- `char *fgets(char *s, int size, FILE *stream);`
  - liest max. `size` Zeichen in einen Buffer ein
  - stoppt bei Zeilenumbruch und EOF
- `int fputc(int c, FILE *stream);`
  - schreibt ein einzelnes Zeichen in die Datei
- `int fputs(const char *s, FILE *stream);`
  - schreibt einen null-terminierten String (ohne das Null-Zeichen)

Dateisystem: Dateien, Verzeichnisse und inodes

POSIX Verzeichnisschnittstelle



inode enthält Dateiattribute & Verweise auf Datenblöcke

Verzeichnis spezielle Datei mit Paaren aus Namen & inode-Nummer

opendir, closedir, readdir

- `DIR *opendir(const char *name);`
  - öffnet ein Verzeichnis
  - liefert einen Zeiger auf den Verzeichniskanal
- `struct dirent *readdir(DIR *dirp);`
  - liest einen Eintrag aus dem Verzeichniskanal und gibt einen Zeiger auf die Datenstruktur `struct dirent` zurück
  - gibt `NULL` zurück, wenn EOF erreicht wurde **oder** im Fehlerfall
  - ⇝ bei EOF bleibt errno unverändert (auch wenn `errno != 0`), im Fehlerfall wird `errno` entsprechend gesetzt
- `int closedir(DIR *dirp);`
  - schließt den Verzeichniskanal

struct dirent

```
01 struct dirent {
02   ino_t          d_ino;       // inode number
03   off_t          d_off;       // not an offset; see NOTES
04   unsigned short d_reclen;    // length of this record
05   unsigned char  d_type;      // type of file; not supported
06                               // by all filesystem types
07   char           d_name[256]; // filename
08 };
```

- entnommen aus Manpage `readdir(3)`
- nur `d_name` und `d_ino` Teil des POSIX-Standards
- relevant für uns: Dateiname (`d_name`)

Einschub: Komma-Operator

- Funktionsweise:
  1. Auswertung des ersten Ausdrucks (Verwerfen dieses Ergebnisses)
  2. Auswertung des zweiten Ausdrucks (Rückgabe dieses Ergebnisses)

```
01 int c = (add(3,2), sub(3,2));
```

- Geeignet für Initialisierungen vor Überprüfung der Schleifenbedingung

⇒ cli/sei

```
01 while(cli(), event != 0){
02   sleep_enable();
03   sei();
04   sleep_cpu();
05   ...
06 }
```

- Elegant, aber keine Notwendigkeit!

Fehlerbehandlung bei readdir

- Fehlerprüfung durch Setzen und Prüfen von `errno`:

```
01 #include <errno.h>
02 ...
03     struct dirent *ent;
04     while(errno=0, (ent=readdir()) != NULL) {
05         ... /* keine weiteren break-Statements in der Schleife */
06     }
07     /* EOF oder Fehler? */
08     if(errno != 0) {
09         /* Fehler */
10         ...
11     }
```

- `errno=0` unmittelbar vor Aufruf der problematischen Funktion
  ⇒ `errno` wird nur im Fehlerfall gesetzt und bleibt sonst unverändert
- Abfrage der errno unmittelbar nach Rückgabe des pot. Fehlerwerts
  ⇒ `errno` könnte sonst durch andere Funktion verändert werden
- Zuweisungsausdruck hat nach Zuweisung Wert des **li. Operanden**
- Keine Hilfsfunktion hier (`ferror()` bei `getchar()`)

Fehlerbehandlung bei readdir

- Fehlerprüfung durch Setzen und Prüfen von `errno`:

```
01 #include <errno.h>
02 ...
03     struct dirent *ent;
04     while(1) {
05         errno = 0;
06         ent = readdir(...);
07         if(ent == NULL) break;
08         ... /* keine weiteren break-Statements in der Schleife */
09     }
10     /* EOF oder Fehler? */
11     if(errno != 0) {
12         /* Fehler */
13         ...
14     }
```

- `errno=0` unmittelbar vor Aufruf der problematischen Funktion
  ⇒ `errno` wird nur im Fehlerfall gesetzt und bleibt sonst unverändert
- Abfrage der errno unmittelbar nach Rückgabe des pot. Fehlerwerts
  ⇒ `errno` könnte sonst durch andere Funktion verändert werden

Datei-Attribute ermitteln: stat

- `readdir(3)` liefert **nur Name und inode-Nummer** eines Verzeichniseintrags
- Weitere Attribute stehen im **inode**
- `stat(2)` Funktions-Prototyp:

```
01 #include <sys/types.h>
02 #include <sys/stat.h>
03 int stat(const char *path, struct stat *buf);
```

- Argumente:
  - `path`: Dateiname
  - `buf`: Zeiger auf Puffer, in den inode-Informationen eingetragen werden
- Rückgabewert: `0` wenn OK, `-1` wenn Fehler
- Beispiel:

```
01 struct stat buf;
02 stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
03 printf("inode-Nummer: %ld\n", buf.st_ino);
```

Das struct stat

- Ausgewählte Elemente
  - `dev_t st_dev` Gerätenummer (des Dateisystems) = Partitions-Id
  - `ino_t st_ino` inode-Nummer (Tupel `st_dev`, `st_ino` eindeutig im System)
  - `mode_t st_mode` Dateimodus (u.a. Zugriffsrechte) und Dateityp
  - `nlink_t st_nlink` Anzahl der (Hard-)Links auf den Inode
  - `uid_t st_uid` UID des Besitzers
  - `gid_t st_gid` GID der Dateigruppe
  - `dev_t st_rdev` DeviceID, nur für Character oder Blockdevices
  - `off_t st_size` Dateigröße in Bytes
  - `time_t st_atime` Zeit des letzten Zugriffs (in Sekunden seit 1.1.1970)
  - `time_t st_mtime` Zeit der letzten Veränderung (in Sekunden ...)
  - `time_t st_ctime` Zeit der letzten Änderung der Inode-Information (...)
  - `unsigned long st_blksize` Blockgröße des Dateisystems
  - `unsigned long st_blocks` Anzahl der von der Datei belegten Blöcke

stat / lstat: st_mode

- `st_mode` enthält Informationen über den Typ des Eintrags:
  - `S_IFMT 0170000` bitmask for the file type bitfields
  - `S_IFSOCK 0140000` socket
  - `S_IFLNK 0120000` symbolic link
  - `S_IFREG 0100000` regular file
  - `S_IFBLK 0060000` block device
  - `S_IFDIR 0040000` directory
  - `S_IFCHR 0020000` character device
  - `S_IFIFO 0010000` FIFO

```
01 mode_t m = stat_buf.st_mode;
02 if( (m & S_IFMT) == S_IFREG) ...
```

- Zur einfacheren Auswertung werden Makros zur Verfügung gestellt:
  - `S_ISREG(m)` - is it a regular file?
  - `S_ISDIR(m)` - directory?
  - `S_ISCHR(m)` - character device?
  - `S_ISLNK(m)` - symbolic link?

Aufgabe: printdir

- Iteration über alle via Parameter übergebene Verzeichnisse
- Ausgabe aller darin enthaltenen Einträge mit Größe und Name
- Anzeige der Anzahl von regulären Dateien und deren Gesamtgröße (pro Verzeichnis)
- Relevante Funktionen:
  - `opendir(3)` bekommt einen Pfad
  - `readdir(3)` liefert nur einen Dateinamen
  - `stat(2)` weiß nicht auf welchen Pfad sich dieser Dateiname bezieht
    - ⇒ `stat(2)` braucht einen vollständigen Pfad mit Datei
    - ⇒ `strncpy(3)`, `strncat(3)`, `snprintf(3)`
    - ⇒ Beim Kopieren von Zeichenketten muss man aufpassen, dass immer genug Speicher zur Verfügung steht.
- Fehlerbehandlung:
  - Jede falsche Benutzereingabe abfangen
    - ⇒ den DAU annehmen ☺
  - Aussagekräftige Fehlermeldungen

Aufgabe: printdir

Hands-on: simple grep

Hands-on: simple grep (1)

```
# Usage: ./sgrep <text> <files...>
$ ./sgrep "SPiC" klausur.tex aufgabe.tex
Klausur im Fach SPiC
SPiC Aufgabe
SPiC ist cool
```

- Einfache Variante des Kommandozeilentools `grep(1)`
- Durchsucht den Inhalt mehrerer Dateien nach einer Zeichenkette
  und gibt alle Zeilen aus, die diese Zeichenkette enthalten
- Ablauf
  - Dateien zeilenweise einlesen
  - Zeile nach Zeichenkette durchsuchen
  - Zeile ggf. auf `stdout` ausgeben
- Sinnvolle Fehlerbehandlung beachten
  - Fehlende Dateien melden und überspringen
  - Fehlermeldungen auf `stderr` ausgeben

Hands-on: simple grep (2)

- Hilfreiche Funktionen:
  - `fopen(3)` ⇒ Öffnen einer Datei
  - `fgets(3)` ⇒ Einlesen einer Zeile
  - `fputs(3)` ⇒ Ausgeben einer Zeile
  - `fclose(3)` ⇒ Schließen einer Datei
  - `strstr(3)` ⇒ Suche eines Teilstrings

```
char *strstr(const char *haystack, const char *needle);
```

```
# Usage: ./sgrep [-i] <text> <files...>
$ ./sgrep -i "spic" klausur.tex aufgabe.tex
klausur.tex:13: Klausur im Fach SPiC
aufgabe.tex:32: SPiC Aufgabe
aufgabe.tex:56: SPiC ist cool
```

- Erweiterung
  - `strstr(3)` selbst implementieren
  - Ausgabe von Dateinamen/Zeilennummer vor jeder Zeile
  - Ignorieren der Groß-/Kleinschreibung mit Option `-i`