

Übungen zu Systemnahe Programmierung in C (SPiC)

Sebastian Maier
(Lehrstuhl Informatik 4)

Übung 7



Wintersemester 2016/2017



Inhalt

POSIX Verzeichnisschnittstelle

Dateisystem: Dateien, Verzeichnisse und inodes
opendir, closedir, readdir
Fehlerbehandlung bei readdir
Verwendung von stat

Aufgabe: printdir

Debuggen

GDB & CGDB
Valgrind

Hands-on: simple grep



Inhalt

POSIX Verzeichnisschnittstelle

Dateisystem: Dateien, Verzeichnisse und inodes
opendir, closedir, readdir
Fehlerbehandlung bei readdir
Verwendung von stat

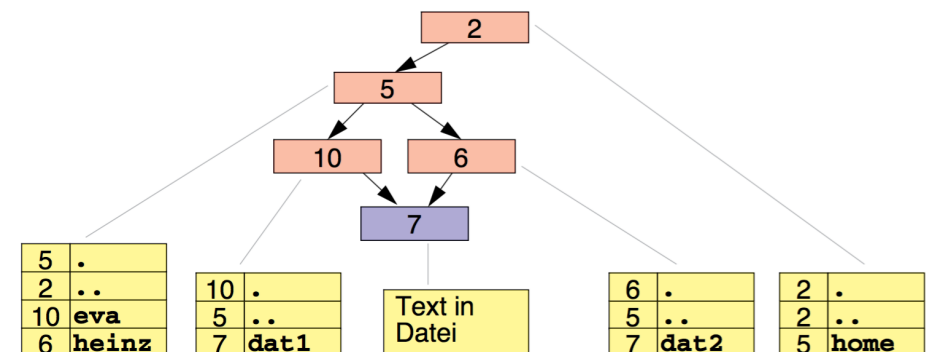
Aufgabe: printdir

Debuggen

Hands-on: simple grep



Dateisystem: Dateien, Verzeichnisse und inodes



inode enthält Dateiattribute & Verweise auf Datenblöcke

Verzeichnis spezielle Datei mit Paaren aus Namen & inode-Nummer



opendir, closedir, readdir

Funktions-Prototypen (Details siehe Vorlesung)

```
1 #include <sys/types.h>
2 #include <dirent.h>
3
4 DIR *opendir(const char *dirname);
5 int closedir(DIR *dirp);
6 struct dirent *readdir(DIR *dirp);
```

Rückgabewert von readdir

- Zeiger auf Datenstruktur vom Typ struct dirent
- NULL, wenn EOF erreicht wurde **oder** im Fehlerfall

~ bei EOF bleibt errno unverändert (auch wenn errno != 0),
im Fehlerfall wird errno entsprechend gesetzt



struct dirent

```
1 struct dirent {
2     ino_t      d_ino;        // inode number
3     off_t      d_off;        // not an offset; see NOTES
4     unsigned short d_reclen; // length of this record
5     unsigned char d_type;    // type of file; not supported
6                             // by all filesystem types
7     char       d_name[256]; // filename
8 };
```

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)

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

Geeignet für Initialisierungen vor Überprüfung der Schleifenbedingung

⇒ cli/sei

```
1 while(cli(), event != 0){
2     sleep_enable();
3     sei();
4     sleep_cpu();
5     ...
6 }
```

Elegant, aber keine Notwendigkeit!



Fehlerbehandlung bei readdir

Fehlerprüfung durch Setzen und Prüfen von errno:

```
1 #include <errno.h>
2 ...
3 struct dirent *ent;
4 while(1) {
5     errno = 0;
6     ent = readdir(...);
7     if(ent == NULL) break;
8     ... /* keine weiteren break-Statements in der Schleife */
9 }
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 evtl. unverändert
- Abfrage der errno unmittelbar nach Rückgabe des pot. Fehlerwerts
⇒ errno könnte sonst durch andere Funktion verändert werden



Fehlerbehandlung bei readdir

- Fehlerprüfung durch Setzen und Prüfen von errno:

```
1 #include <errno.h>
2 ...
3 struct dirent *ent;
4 while(errno=0, (ent=readdir()) != NULL) {
5     ... /* keine weiteren break-Statements in der Schleife */
6 }
7 /* EOF oder Fehler? */
8 if(errno != 0) {
9     /* Fehler */
10    ...
11 }
```

- errno=0 unmittelbar vor Aufruf der problematischen Funktion
⇒ errno wird nur im Fehlerfall gesetzt und bleibt sonst evtl. 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())



Datei-Attribute ermitteln: stat

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

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 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:

```
1 struct stat buf;
2 stat("/etc/passwd", &buf); /* Fehlerabfrage ... */
3 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 Inodenummer (Tupel st_dev, st_ino eindeutig im System)
- mode_t st_mode Dateimode, u.a. Zugriffs-Bits 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

```
1 mode_t m = stat_buf.st_mode;
2 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?



Inhalt

POSIX Verzeichnisschnittstelle

Aufgabe: printdir

Debuggen

Hands-on: simple grep



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(3)` weiß nicht auf welchen Pfad sich dieser Dateiname bezieht
 - ⇒ `stat(3)` 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



Inhalt

POSIX Verzeichnisschnittstelle

Aufgabe: printdir

Debuggen
GDB & CGDB
Valgrind

Hands-on: simple grep



Debuggen

```
8 #define For(i, s, n) for(int i = (s); i < (n); ++ i)
9 #define Clr(a) memset(a, 0, sizeof a)
10 #define Ot(c, d) cout << "Case #" << c << ": " << (d) << endl
11
12 template<class T>
13 string str(T a){stringstream ss; ss << a; string ret; ss >> ret; return ret;}
14
15 int lr[41][2];
16 bool odd[41];
17
18 int main(){
19     int T;
20     cin >> T;
21     for(int c = 1; c <= T; ++ c){
22
23         ReadInt(N);
24         For(i, 1, N){
25             cin >> lr[i][0] >> lr[i][1];
26         }
27         Clr(odd);
28
29         int UPPER = (N - 1) * (1 << (N - 1));
30         int cur = 1; odd[1] = true;
31         int result = 0;
32
33         //Users/mirikle/Algorithm/e/5.C
34         cin >> lr[1][0] >> lr[1][1];
35         Clr(odd);
36
37         int UPPER = (N - 1) * (1 << (N - 1));
38     }
39     (gdb) r
```



Debuggen mit GDB & CGDB

- Übersetzen mit Debug-Symbolen (-g) & ohne Optimierungen (-O0)

```
1 gcc -g -pedantic -Wall -Werror -O0 -std=c99 -D_XOPEN_SOURCE=500
```

- Starten des Debuggers

```
1 gdb ./cworld
2 # alternativ ...
3 cgdb --args ./cworld arg0 arg1 ...
```

- Kommandos

- b(reak): Breakpoint setzen
- r(un): Programm bei main() starten
- n(ext): nächste Anweisung (nicht in Unterprogramme springen)
- s(tep): nächste Anweisung (in Unterprogramme springen)
- p(rint) <var>: Wert der Variablen var ausgeben

⇒ **Debuggen ist (fast immer) effizienter als Trial-and-Error!**



Valgrind

- Informationen über:

- Speicherlecks (malloc/free)
- Zugriffe auf nicht gültigen Speicher

- Ideal zum Lokalisieren von Segmentation Faults (SIGSEGV)

- Aufrufe:

- valgrind ./cworld
- valgrind --leak-check=full --show-reachable=yes --track-origins=yes ./cworld



Inhalt

POSIX Verzeichnisschnittstelle

Aufgabe: printdir

Debuggen

Hands-on: simple grep



Hands-on: simple grep (1)

```
1 # Usage: ./sgrep <text> <files...>
2 $ ./sgrep "SPiC" klausur.tex aufgabe3.tex
3 Klausur im Fach SPiC
4 SPiC Aufgabe 3
5 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



■ 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

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

```
1 # Usage: ./sgrep [-i] <text> <files...>
2 $ ./sgrep -i "spic" klausur.tex aufgabe3.tex
3 klausur.tex:13: Klausur im Fach SPiC
4 aufgabe3.tex:32: SPiC Aufgabe 3
5 aufgabe3.tex:56: SPiC ist cool
```

■ Erweiterung

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

