

System-Programmierung

3: File In-/Output

CC BY-SA, Thomas Amberg, FHNW
(Soweit nicht anders vermerkt)
Slides: tmb.gr/syspr-3



Überblick

Diese Lektion behandelt *File Input und Output*.

Wie man Files öffnet, liest und schreibt.

Datenstrukturen im Filesystem.

2

File I/O

Alle System Calls für I/O beziehen sich auf einen File Deskriptor, ein (kleiner) positiver Integer Wert.

File Deskriptoren können sich auf Pipes, FIFOs, Sockets, Terminals, Devices oder Dateien beziehen.

Jeder Prozess hat sein eigenes Set an Deskriptoren, per Konvention mindestens *stdin*, *stdout* und *stderr*, von der Shell geöffnet und an den Prozess vererbt.

3

Standard File Deskriptoren

File Deskriptor Nummer

```
0 // standard input
1 // standard output
2 // standard error
```

POSIX Konstante

```
#include <unistd.h>
STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
```

4

File I/O System Calls

open() öffnet das File *pathname*, ergibt Deskriptor *fd*:

```
fd = open(pathname, flags, mode); // -1: error
```

read() liest $r \leq n$ bytes aus File *fd* in den Buffer *buf*:

```
r = read(fd, buf, n); // r = 0: EOF, -1: error
```

write() schreibt $w \leq n$ bytes aus Buffer *buf* ins File *fd*:

```
w = write(fd, buf, n); // w = -1: error
```

close() schliesst das File *fd*: `result = close(fd)`;

5

Hands-on, 5': TLPI Beispiele bauen

[TLPI] Beispielcode Setup auf dem Raspberry Pi ~:

```
$ wget http://man7.org/tlpi/code/download/\
tlpi-210511-book.tar.gz
$ tar xfmv tlpi-210511-book.tar.gz
$ cd tlpi-book
$ sudo apt-get install libcapi-dev
$ sudo apt-get install libacl1-dev
$ make
```

Der Code ist open-source, mit GNU GPLv3 Lizenz.

6

File *open()* System Call

Deklaration:

```
int open(const char *pathname, int flags, ...  
/* mode_t mode */); // nur mit O_CREAT
```

Access, creation & status *flags* werden mit | verodert:

```
O_RDONLY, O_WRONLY, O_RDWR // access mode  
O_CREAT, ... // creation, O_APPEND, ... // status
```

Falls *O_CREAT* in *flags*, setzt *mode* Zugriffsrechte:

```
S_IRUSR, S_IWUSR, ... // mit | kombinierbar
```

7

File *open()* Beispiele

Existierende Datei zum Lesen öffnen:

```
char *f = "a.txt"; int fd = open(f, O_RDONLY);
```

Existierende oder neue Datei öffnen, zum Lesen und Schreiben, R+W für Owner, sonst keine Permissions:

```
fd = open(f, O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
```

Datei öffnen, um etwas am Dateiende anzuhängen:

```
fd = open(f, O_WRONLY|O_APPEND);
```

8

File *open()* Flags

O_RDONLY	Öffnen zum Lesen	O_NOFOLLOW	Symbolische Links nicht dereferenzieren
O_WRONLY	Öffnen zum Schreiben	O_TRUNC	Existierendes File auf Länge 0 kürzen
O_RDWR	Öffnen zum Lesen und Schreiben	O_APPEND	Write wird am Ende des Files angehängt
O_CLOEXEC	Das close-on-exec Flag setzen	O_ASYNC	Signal generieren, wenn I/O möglich wird
O_CREAT	File erstellen, falls es nicht bereits existiert	O_DIRECT	File I/O umgeht Buffer Cache
O_DIRECTORY	Fehler, falls <i>pathname</i> kein Directory ist	O_DSYNC	Datenintegrität für synchronisierten I/O
O_EXCL	Mit <i>O_CREAT</i> : File exklusiv erstellen	O_NOATIME	Bei read last access time nicht updaten
O_LARGEFILE	Auf 32-bit Systemen um grosse Files zu öffnen	O_NONBLOCK	Im "nonblocking" Modus öffnen
O_NOCTTY	<i>Pathname</i> kann nicht kontroll. Terminal sein	O_SYNC	Macht write synchron

9

File *open()* Modes

S_ISUID	Set-user-ID	S_IROTH	Other-read
S_ISGID	Set-group-ID	S_IWOTH	Other-write
S_ISVTX	Sticky	S_IXOTH	Other-execute
S_IRUSR	User-read		
S_IWUSR	User-write	// z.B. rw-rw-rw- =>	
S_IXUSR	User-execute	mode_t mode =	
S_IRGRP	Group-read	S_IRUSR	S_IWUSR
S_IWGRP	Group-write	S_IRGRP	S_IWGRP
S_IXGRP	Group-execute	S_IROTH	S_IWOTH

10

File *open()* Errors

Bei Fehlern liefert *open()* den Wert -1 und setzt *errno*:

```
fd = open(pathname, flags, mode);  
if (fd == -1) { printf("%d\n", errno); }
```

Eine Auswahl an möglichen Fehler-Werten:

EACCES	Ungenügende Permissions	ENOENT	Ein Teil des File-Pfads existiert nicht, oder <i>O_CREAT</i> fehlt
EISDIR	Das File ist ein Directory, Schreiben ist nicht möglich	EROFS	Das File ist auf einem read-only File-System, Schreiben geht nicht
EMFILE	Zu viele offene Files im Prozess, RLIMIT_NOFILE ist erreicht	ETXTBSY	File ist ein laufendes Executable, muss zuerst terminiert werden
ENFILE	Zu viele offene Files im System		

11

File *read()* System Call

Deklaration:

```
ssize_t read(int fd, void *buf, size_t n);  
Resultat:  $r \leq n$  Bytes gelesen aus File fd in Buffer buf, falls  $r = 0$ , wurde End of File (EOF) erreicht.
```

z.B. Bytes aus ASCII File lesen, als String ausgeben:

```
char buf[32 + 1]; int fd = open("a.txt", ...);  
ssize_t r = read(fd, buf, 32);  
buf[r] = '\0'; printf("read: %s\n", buf);
```

12

File *read()* Errors

Bei Fehlern liefert *read()* den Wert *-1* und setzt *errno*:

```
r = read(fd, buf, n);  
if (r == -1) { printf("%d\n", errno); }
```

Eine Auswahl an möglichen Fehler-Werten:

EAGAIN	Lese-Operation würde ein als non-blocking geöffnetes File blockieren	EINVAL	File Descriptor zeigt auf Objekt das nicht gelesen werden kann
EBADF	File Descriptor ungültig oder nicht zum Lesen geöffnet	EIO	I/O Fehler bei low-level I/O, oder weil Call aus Hintergrundprozess
EFAULT	Der Buffer <i>buf</i> ist ausserhalb des dem Caller zugänglichen Speichers	EISDIR	File Descriptor zeigt auf Directory
EINTR	Der Call wurde von einem Signal unterbrochen, vor dem Lesen		

13

File *write()* System Call

Deklaration:

```
ssize_t write(int fd, const void *buf, size_t n);  
Resultat:  $w \leq n$  Bytes geschrieben aus Buffer buf in fd, falls  $w < n$  gab es zu wenig Diskplatz, oder ein Signal.
```

z.B. Bytes eines Strings in ein ASCII File schreiben:

```
char buf[] = "hello";  
int fd = open("a.txt", O_WRONLY|O_APPEND);  
ssize_t w = write(fd, buf, 5); // ohne '\0'
```

14

File *write()* Errors

Bei Fehlern liefert *write()* den Wert *-1* und setzt *errno*:

```
w = write(fd, buf, n);  
if (w == -1) { printf("%d\n", errno); }
```

Eine Auswahl an möglichen Fehler-Werten:

EAGAIN	Schreib-Op. würde ein als non-blocking geöffnetes File blockieren	EINTR	Der Call wurde von einem Signal unterbrochen, vor dem Schreiben
EBADF	File Descriptor ungültig oder nicht zum Schreiben geöffnet	EINVAL	File Descriptor zeigt auf Objekt das nicht beschreibbar ist
EDQUOT	User Quota an Blocks auf der von <i>fd</i> referenzierten Disk ist erschöpft	EIO	I/O Fehler bei low-level I/O, oder weil Call aus Hintergrundprozess
EFAULT	Der Buffer <i>buf</i> ist ausserhalb des dem Caller zugänglichen Speichers	ENOSPC	Das von <i>fd</i> referenzierte Device hat keinen Speicherplatz mehr

15

Hands-on, 20': File I/O

copy.c^{TLPI}

Schreiben Sie ein Programm *my_copy.c*, das eine beliebig grosse, existierende Datei *source* liest und in eine neue Datei *dest* kopiert, wie das *cp* Kommando:
\$./my_copy source dest

Verwenden Sie System Calls: *open()*, *read()*, *write()*.

Fertig? Bauen Sie Fehlerbehandlung ein.

16

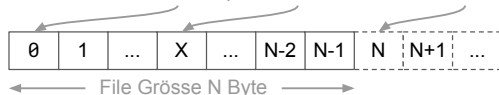
File Offset

[lseek.c](#) | [seek_io.c](#)^{TLPI}

Für jedes offene File hält der Kernel einen *File Offset*, die Stelle wo das nächste *read()* oder *write()* beginnt.

lseek() erlaubt, den *offset* zu setzen, gezählt ab *from*:

```
off_t lseek(int fd, off_t offset, int from);  
// from = SEEK_SET, SEEK_CUR oder SEEK_END
```



17

Naives Append

Bytes sollen ans Ende eines Files geschrieben werden:

```
if (lseek(fd, 0, SEEK_END) != -1) { // → EOF  
    write(fd, buf, len); // append to end  
}
```

Suchen und Schreiben geschehen hier *nicht atomar*. Durch die ungünstige zeitliche Verzahnung von Teiloperationen können *Race Conditions* entstehen.

18

Race Condition

Hier eine Race Condition bei *write* ins selbe File *fd*:

```
A1: if (lseek(fd, 0, SEEK_END) != -1) {
B1:   if (lseek(fd, 0, SEEK_END) != -1) {
A2:     write(fd, buf, len); // append
A3:   }
B2:     write(fd, buf, len); // (!)
B3:   } // bei B2 ist SEEK_CUR != EOF
```

(!) Prozess B überschreibt Daten von Prozess A.

19

O_APPEND

[atomic_append.c](#)^{TLPI}

O_APPEND macht das Anfügen mit *write()* atomar:
`open(..., ...|O_APPEND); ...; write(...); // atomic`

Das Programm *atomic_append* zeigt den Unterschied von Anhängen mit *lseek()* und *O_APPEND*:

```
$ ./atomic_... f1 100000 x & ./atomic_... f1 100000 x
$ ./atomic_... f2 100000 & ./atomic_... f2 100000
$ ls -al f*
```

Das Flag *x* bedeutet hier "*lseek()* statt *O_APPEND*" 20

File exklusiv erstellen

Wenn O_EXCL zusammen mit O_CREAT verwendet wird, gibt es einen Fehler, falls das File schon existiert.

Prüfen und Erstellen geschieht dann in einem Schritt; bei Erfolg wurde das File garantiert "von uns" erstellt.

```
int fd = open(pathname, O_CREAT|O_EXCL|O_RDWR,
S_IRUSR|S_IWUSR);
```

21

File *ftruncate()* System Call

[truncate.c](#)

ftruncate() kürzt die Länge des Files auf *length* Bytes:
`int ftruncate(int fd, off_t length); // 0 or -1`

Feature Test Makro für *glibc*, aus der [Doku](#):

```
_XOPEN_SOURCE >= 500
|| _POSIX_C_SOURCE >= 200112L // seit 2.3.5
|| _BSD_SOURCE // glibc Version <= 2.19
```

Compiler Flag bei *gcc*, falls z.B. *-std=c99*:

```
-D_XOPEN_SOURCE=500
```

22

Einschub: Feature Test Makros

Mit [Feature Test Macros](#) kann die *glibc* Library prüfen, welche Definitionen der aufrufende Code erwartet:

```
// features.h, z.B. via unistd.h
... if defined _XOPEN_SOURCE && ...
```

Das zugehörige *define* muss vor dem 1. *include* stehen:

```
// my_code.c
#define _XOPEN_SOURCE 500 // od. als gcc Flag
#include <unistd.h> ... // d.h. POSIX.1, POSIX.2, X/Open (XPG4) Definitions,
// und SUSv2 (UNIX 98 & XPG5) Extensions
```

23

Kernel-Datenstrukturen für Files

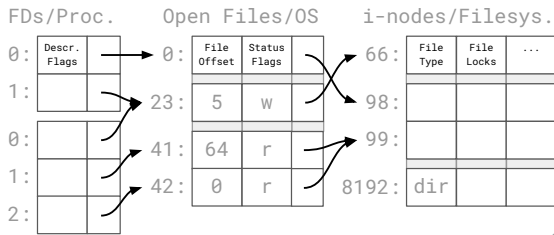
Der Kernel hat eine File Deskriptor Tabelle / Prozess, mit dem *close-on-exec* Flag und einem File Pointer.

Dazu führt er eine systemweite Tabelle offener Files, mit Status Flags, Access Mode und *i-node* Pointer.

Für das Filesystem gibt es eine Tabelle mit *i-nodes* die den File Typ (regulär, Socket, FIFO), Permissions und einen Zeiger auf eine Liste von Locks enthält.

24

Der Kernel führt eine Tabelle mit *fd* pro Prozess, mit offenen Files im System & mit *i-nodes* im Filesystem:



File Status Lesen

fcntl() liest oder ändert Zugriff, Status offener Files:

```
int flags = fcntl(fd, F_GETFL); // Flags lesen
if (flags & O_SYNC) { ... } // Flags prüfen
int mode = flags & O_ACCMODE; // Zugriff lesen
if (mode == O_RDONLY) { ... } // Zugriff prüfen
```

```
flags |= O_APPEND; // Flags modifizieren
fcntl(fd, F_SETFL, flags); // Flags schreiben
```

Nützlich, wenn man ein File schon offen bekommt. 26

File I/O Buffering

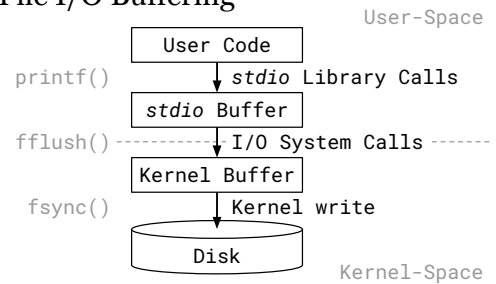
Bei regulären Files sind *read/write()* Calls gebuffert, der Kernel flushed seinen Buffer später auf die Disk.

Wenn nach *write()*, aber vor dem flushen ein *read()* kommt, retourniert der Kernel Bytes aus dem Buffer.

Damit sind *read()* und *write()* schnell genug, auch wenn der Zugriff auf die Disk relativ langsam ist.

27

File I/O Buffering



Flushen von *stdio* Buffers [stdio_buf.c](#)

Die *fflush()* Funktion entleert den Buffer mit *write()*:

```
int fflush(FILE *stream); // 0 od. EOF, errno
```

Falls *stream = NULL* ist, werden alle Buffer in *stdio* "gespült", die zu Output Streams gehören.

Beim Flushen von Input Streams wird der gebufferte Input verworfen; Buffer bleibt leer bis wieder *read()*.

Bei *close()* auf Streams wird *fflush()* aufgerufen. 29

Flushen von Kernel Buffers

Der *fsync()* Call schreibt den File Buffer auf die Disk, bzw. erstellt den "file integrity completion" Zustand:

```
int fsync(int fd); // 0 oder -1, errno
```

Denselben Effekt erreicht man mit dem *O_SYNC* Flag, welches nachfolgende *write()* Calls "synchron" macht:

```
int fd = open(f, O_SYNC|...); // write does fsync
```

Der Call *sync()* flushed alle File Buffer im System.

30

Hands-on, 10': Buffering `write_bytes.c`^{TLPI}

Kompilieren Sie das Programm `write_bytes` zuerst mit und dann ohne die Compiler Option `-DUSE_O_SYNC`.

Messen Sie die Laufzeit (*real*, *sys*) der Binaries, je mit `num-bytes = 100000` und `buf-size = 1 / 256 / 4096`:
\$ `time write_bytes my_file num-bytes buf-size`

Welchen Einfluss hat die Buffergrösse? Und `O_SYNC`?
Wann/wozu ist Synchronisieren überhaupt nötig?

31

Directories

Directories sind im Filesystem wie Files gespeichert, aber mit einem anderen File Typ im i-node Eintrag.

Directory "Files" sind als Tabelle von File-Namen und i-node Nr. organisiert.

Nr	i-node	
2:	...	dir
7:	...	dir
99:	...	file

Arrows point from the 'dir' entries to the root directory table and from the 'file' entry to the log file table.

/:	etc	3
	tmp	7
/tmp:	log	99
/log:	data	

Directory Operationen

`dir.c`

Directory mit Pfad `pathname` erstellen, mit `mkdir()`:
int `mkdir`(const char *pathname, mode_t mode);

File von Pfad `old` zu `new` umbenennen mit `rename()`:
int `rename`(const char *old, const char *new);

File oder Directory löschen mit `remove()`:
int `remove`(const char *pathname);

File in Directory öffnen: siehe `open()` weiter oben.

33

File oder Directory Löschen

Die `remove()` Funktion löscht ein File / Directory:
int `remove`(const char *pathname);

`remove()` ruft entweder `unlink()` oder `rmdir()` auf:
int `rmdir`(const char *pathname); // für Dir's
int `unlink`(const char *pathname); // für Files

Falls kein anderer Prozess mehr das File offen hat, wird es gelöscht und der Speicherplatz freigegeben.

34

Selbststudium: Experten & Pioniere

Um state-of-the-art C von einem Experten zu lernen, schauen Sie [How I program C](#), mit Eskil Steenberg.
Notieren Sie sich drei Tipps, die neu sind für Sie.

Um den Ursprung und Einfluss von C zu verstehen: [C Programming Language](#), mit Brian Kernighan und [Why C is so Influential](#), mit David Brailsford.

Ist C eine high- oder low-level Sprache?

35

Feedback oder Fragen?

Gerne im Slack <https://fhnw-syspr.slack.com/>

Oder per Email an thomas.amberg@fhnw.ch

Danke für Ihre Zeit.

36