

# System-Programmierung

## 3: File In-/Output

CC BY-SA, Thomas Amberg, FHNW  
(soweit nicht anders vermerkt)  
Slides: [tmb.gr/syspr-3](http://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, 1, 2 // standard input, output, error

POSIX Konstante

```
#include <unistd.h>
```

```
STDIN_FILENO, STDOUT_FILENO, STDERR_FILENO
```

*stdio* Stream

```
#include <stdio.h>
```

```
stdin, stdout, stderr
```

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, 15': TLPI Beispiele builden

[TLPI] Beispielcode Setup auf dem Raspberry Pi ~:

```
$ wget http://man7.org/tlpi/code/download/tlpi-180725-book.tar.gz
```

```
$ tar xfmv tlpi-180725-book.tar.gz
```

```
$ cd tlpi-book
```

```
$ sudo apt-get install libcap-dev
```

```
$ sudo apt-get install libc11-dev
```

```
$ make
```

Der Code ist open-source, mit [GNU GPLv3](https://www.gnu.org/licenses/gpl-3.0.html) 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

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

9

## File *open()* Modes

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

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:

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

Lesen mit *read()* von *stdin*, geöffnet von der Shell:

```
char buf[32]; r = read(STDIN_FILENO, buf, 32);  
if (r != -1) { printf("read: %s\n", buf); }  
// Garbage? => buf[r] = '\0';
```

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.

Schreiben mit *write()* auf *stdout*, von Shell geöffnet:

```
w = write(STDOUT_FILENO, {'h', 'i', '!'}, 3);
if (w != -1) { printf("wrote %d bytes", w); }
```

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, 15': File I/O

copy.c<sup>TLPI</sup>

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 bekannte File I/O System Calls.

16

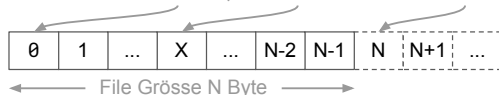
## File Offset

[lseek.c](#) | [seek\\_io.c](#)<sup>TLPI</sup>

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

## Hands-on, 15': File Append

logger.c!

Schreiben Sie ein Programm *my\_logger.c*, welches */proc/uptime* liest und den aktuellen Wert hinten an eine neue bzw. existierende Datei *my.log* anhängt.

Nutzen Sie *lseek()* um das Ende der Datei zu finden.

Nutzen Sie *sleep()* um in einer Schleife jede Sekunde einen neuen Wert an den Log anzuhängen.

18

## O\_APPEND

[atomic\\_append.c](#)<sup>TLPI</sup>

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" 19

## Atomizität von System Calls

Der Kernel garantiert, dass kritische Calls atomar sind, ohne Unterbruch durch andere Prozesse oder Threads.

Das verhindert *Race Conditions*, also Fehler durch die ungünstige zeitliche Verzahnung von Teiloperationen.

Das Problem wird deutlich bei diesem naiven *append*:

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

 20

## 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.

21

## 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);
```

22

## 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
```

23

## 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
#include <unistd.h> ...
```

```
// od. als gcc Flag
-D_XOPEN_SOURCE=500
// d.h. POSIX.1, POSIX.2, X/Open (XPG4) Definitions,
// und SUSv2 (UNIX 98 & XPG5) Extensions
```

24

## Files vs. Deskriptoren

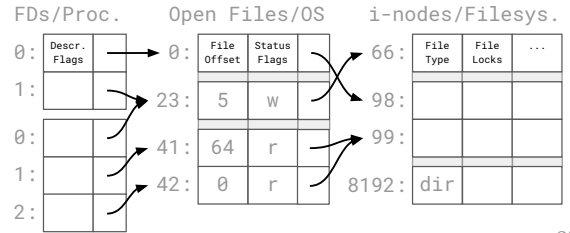
Der Kernel hat eine File Deskriptor Tabelle / Prozess, mit dem *close-on-exec* Flag und einem File Pointer.

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.

25

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



26

## 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.

27

## File Deskriptor Duplizieren

Das Shell Kommando `2>&1` biegt *stderr* auf *stdout* um, File Deskriptor 2 wird Duplikat von File Deskriptor 1.

Beide haben nun denselben Offset im File, so dass z.B. bei Append Daten korrekt aneinandergehängt werden.

Denselben Effekt erreicht man mit dem *dup2()* Aufruf:

```
int fd = dup2(1, 2); // = 2; oder -1, errno
// schliesst 2; dupliziert flags, ptr von 1
```

28

## Hands-on, 10': Dup (auf Papier) `dup.!c`

Was steht im File *f*, nach jedem Aufruf von *write()*?

```
int fd1 = open(f, O_RDWR | O_CREAT | O_TRUNC,
S_IRUSR | S_IWUSR);
int fd2 = dup(fd1), fd3 = open(f, O_RDWR);
write(fd1, "Ente,", 5);
write(fd2, "Hund,", 5);
lseek(fd2, 0, SEEK_SET);
write(fd1, "Haus,", 5);
write(fd3, "Pferd", 5);
```

29

## Temporäre Files

`tmp_file.c`

Der Aufruf *mkstemp()* erzeugt ein temporäres File:

```
char template[] = "/tmp/my-XXXXXX"; // X muss
int fd = mkstemp(template); // oder -1, errno
printf("created filename is %s\n", template);
unlink(template); // Name wird "gelöscht"
close(fd); // File wird geschlossen
```

Alternativen *tmpnam()*, *tempnam()*, und *mktemp()* werden nicht empfohlen, höchstens noch *tmpfile()*.

30

## 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.

31

## User-Space *stdio* Buffering [stdio\\_buf.c](#)

Die C Library I/O Funktionen *fprintf()*, *fscanf()*, ... nutzen Buffering, um System Calls zu reduzieren.

Die Buffergrösse kann im Voraus eingestellt werden:

```
FILE *stream = stdout; // or any other FILE *
res = setvbuf(stream, buf, _IOFBF, BUF_SIZE);
if (res != 0) { ... } // non-zero (!) => error
fprintf(stream, format, ...); // uses BUF_SIZE
Buffer mode kann _IO{Line|Fully|Non}BF sein.
```

32

## 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.

33

## 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
```

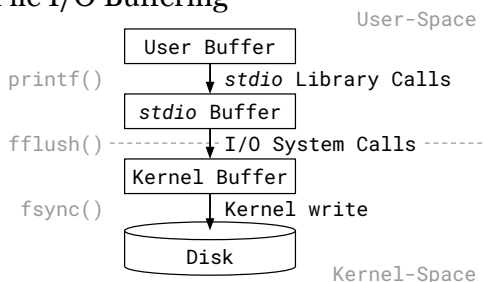
Denselben Effekt erreicht man mit dem *O\_SYNC* Flag, welches nachfolgende *write()* Calls "synchron" macht:

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

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

34

## File I/O Buffering



35

## Hands-on, 5': Buffering [write\\_bytes.c](#)<sup>TLPI</sup>

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, 16, 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?

36

## 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

Diagramm zur Darstellung der Directory-Struktur:

- Die i-node Tabelle zeigt Einträge für Dateien (file) und Verzeichnisse (dir).
- Ein Verzeichnis (z.B. mit i-node 2) enthält eine Liste von Dateien (etc, tmp, log, data).
- Die i-node Nummer (Nr.) ist mit dem i-node selbst verknüpft.

## 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.

38

## 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.

39

## Selbststudium, 3h: 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?

40

## Feedback oder Fragen?

Gerne im Slack <https://fhnw-syspr.slack.com/>  
Oder per Email an [thomas.amberg@fhnw.ch](mailto:thomas.amberg@fhnw.ch)

Danke für Ihre Zeit.

41

