

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.

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.

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
```

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);`

Hands-on, 5': TLPI Beispiele bauen

[**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 libacl1-dev
```

```
$ make
```

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

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
```

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);
```


File *open()* Flags

O_RDONLY

Öffnen zum Lesen

O_WRONLY

Öffnen zum Schreiben

O_RDWR

Öffnen zum Lesen und Schreiben

O_CLOEXEC

Das *close-on-exec* Flag setzen

O_CREAT

File erstellen, falls es nicht bereits existiert

O_DIRECTORY

Fehler, falls *pathname* kein Directory ist

O_EXCL

Mit *O_CREAT*: File exklusiv erstellen

O_LARGEFILE

Auf 32-bit Systemen um grosse Files zu öffnen

O_NOCTTY

Pathname kann nicht kontroll. Terminal sein

O_NOFOLLOW

Symbolische Links nicht dereferenzieren

O_TRUNC

Existierendes File auf Länge 0 kürzen

O_APPEND

Write wird am Ende des Files angehängt

O_ASYNC

Signal generieren, wenn I/O möglich wird

O_DIRECT

File I/O umgeht Buffer Cache

O_DSYNC

Datenintegrität für synchronisierten I/O

O_NOATIME

Bei *read* last access time nicht updaten

O_NONBLOCK

Im "nonblocking" Modus öffnen

O_SYNC

Macht *write* synchron

File *open()* Modes

S_ISUID	Set-user-ID
S_ISGID	Set-group-ID
S_ISVTX	Sticky
S_IRUSR	User-read
S_IWUSR	User-write
S_IXUSR	User-execute
S_IRGRP	Group-read
S_IWGRP	Group-write
S_IXGRP	Group-execute

S_IROTH	Other-read
S_IWOTH	Other-write
S_IXOTH	Other-execute

```
// z.B. rw-rw-rw- =>
mode_t mode =
    S_IRUSR | S_IWUSR |
    S_IRGRP | S_IWGRP |
    S_IROTH | S_IWOTH;
```

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 O_CREAT 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		

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';
```

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

EBADF

File Descriptor ungültig oder nicht zum Lesen geöffnet

EFAULT

Der Buffer *buf* ist ausserhalb des dem Caller zugänglichen Speichers

EINTR

Der Call wurde von einem Signal unterbrochen, vor dem Lesen

EINVAL

File Descriptor zeigt auf Objekt das nicht gelesen werden kann

EIO

I/O Fehler bei low-level I/O, oder weil Call aus Hintergrundprozess

EISDIR

File Descriptor zeigt auf Directory

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); }
```

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

EBADF File Descriptor ungültig oder nicht zum Schreiben geöffnet

EDQUOT User Quota an Blocks auf der von *fd* referenzierten Disk ist erschöpft

EFAULT Der Buffer *buf* ist ausserhalb des dem Caller zugänglichen Speichers

EINTR Der Call wurde von einem Signal unterbrochen, vor dem Schreiben

EINVAL File Descriptor zeigt auf Objekt das nicht beschreibbar ist

EIO I/O Fehler bei low-level I/O, oder weil Call aus Hintergrundprozess

ENOSPC Das von *fd* referenzierte Device hat keinen Speicherplatz mehr

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.

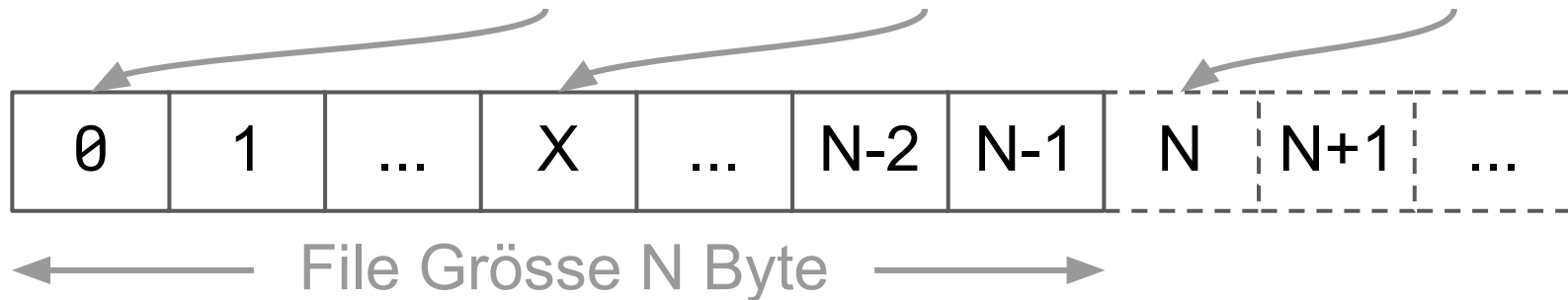
File Offset

`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
```



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.

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.

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);
```

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
```

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
```

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.

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

FDs/Proc.

0:	Descr. Flags	
1:		
0:		
1:		
2:		

Open Files/OS

0:	File Offset	Status Flags	
23:	5	w	
41:	64	r	
42:	0	r	

i-nodes/Filesys.

66:	File Type	File Locks	...
98:			
99:			
8192:	dir		

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

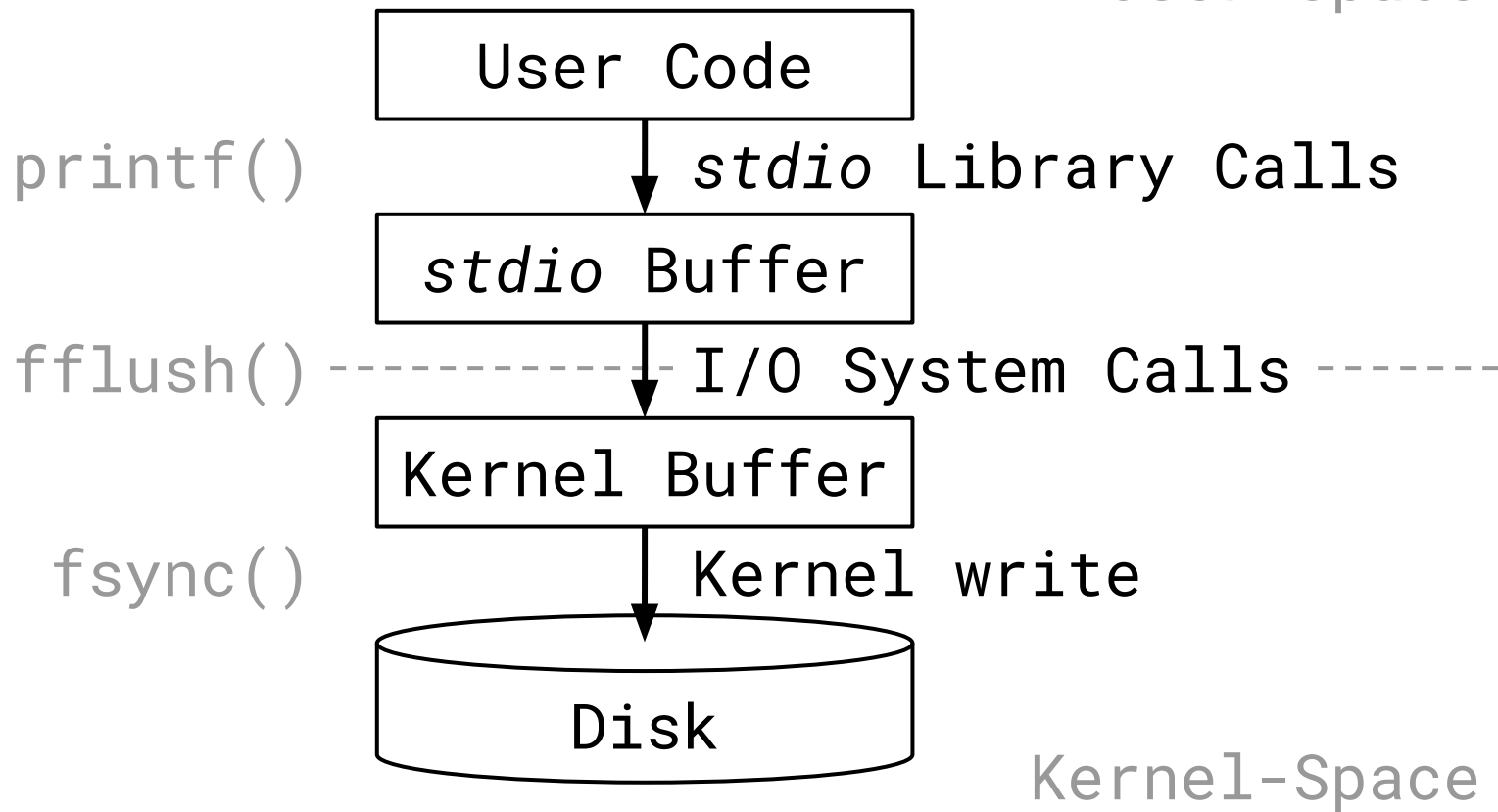
Bei regulären Files sind *read/write()* Calls gebuffert, der Kernel flushed seinen Buffer später auf die Disk.

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.

File I/O Buffering

User-Space



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.

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.

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, 16, 256, 4096:

```
$ time write_bytes my_file num-bytes buf-size
```

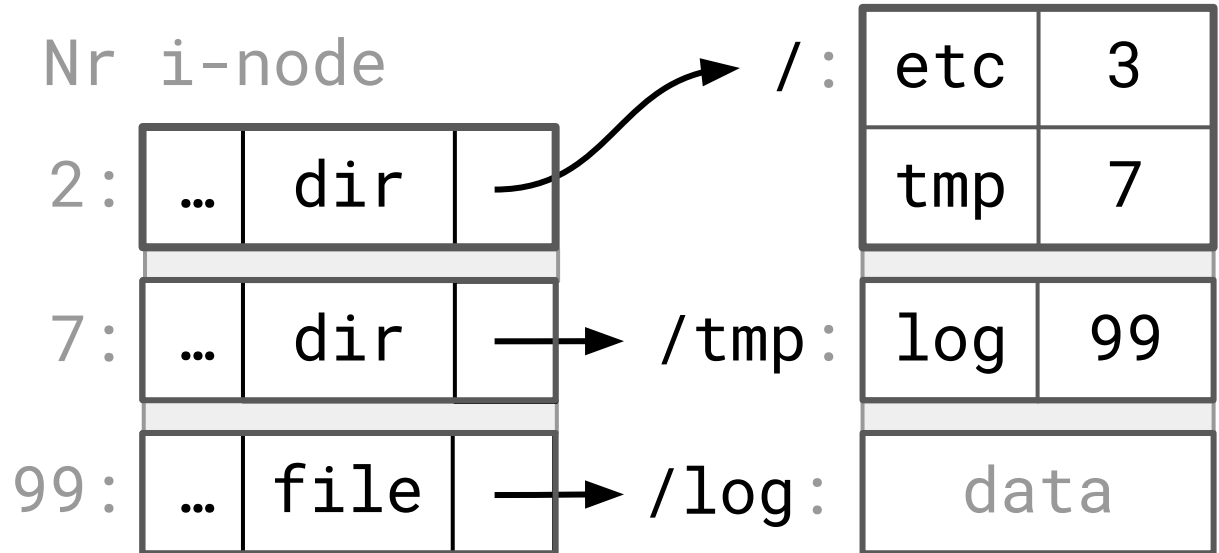
Welchen Einfluss hat die Buffergrösse? Und *O_SYNC*?

Wann/wozu ist Synchronisieren überhaupt nötig?

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.



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.

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.

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?

Feedback oder Fragen?

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

Oder per Email an thomas.amberg@fhnw.ch

Danke für Ihre Zeit.