Übungen zu Systemnahe Programmierung in C (SPiC) – Wintersemester 2022

Übung 8

Phillip Raffeck
Maximilian Ott

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

Lehrstuhl für Verteilte Systeme und Betriebssysteme

FRIEDRICH-ALEXANDER UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Vorstellung Aufgabe 4

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

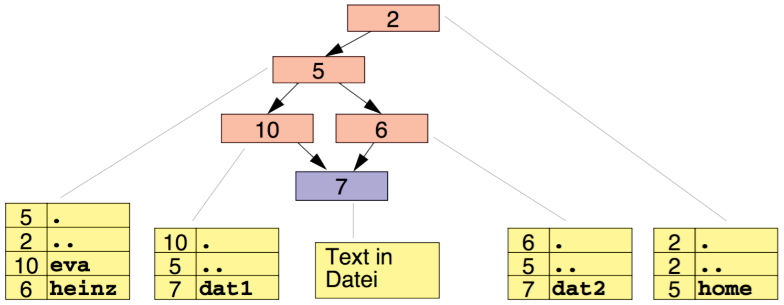
- `int fclose(FILE *fp);`
  - Schreibt ggf. gepufferte Ausgabedaten des Dateikanals
  - Schließt anschließend die Datei

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)

POSIX Verzeichnisschnittstelle

Dateisystem: Dateien, Verzeichnisse und inodes



inode: Enthält Dateiattribute & Verweise auf Datenblöcke

Datei: Block mit beliebigen Daten

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
- `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`)

Fehlerbehandlung bei `readdir(3)`

- Fehlerprüfung durch Setzen und Prüfen der `errno`:

```
01 #include <errno.h>
02 // [...]
03     DIR *dir = opendir("/home/eva/"); // Fehlerbehandlung!!
04
05     struct dirent *ent;
06     while(1) {
07         errno = 0;
08         ent = readdir(dir);
09         if(ent == NULL) {
10             break;
11         }
12         // keine weiteren break-Statements in der Schleife
13         // [...]
14     }
15
16     // EOF oder Fehler?
17     if(errno != 0) { // Fehler
18         // [...]
19     }
20     closedir(dir);
```


Datei-Attribute ermitteln: `stat(2)`

- `readdir(3)` liefert **nur Name und inode-Nummer** eines Verzeichniseintrags
- Weitere Attribute stehen im **inode**

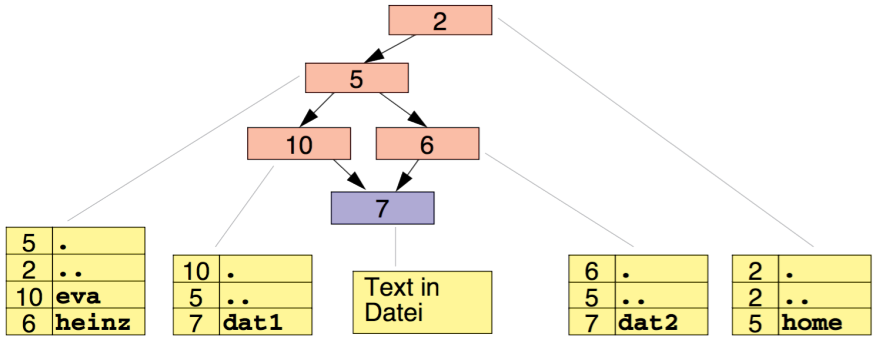
- `int stat(const char *path, struct stat *buf);`
  - Abfragen der Attribute eines Eintrags (folgt symlinks)
- `int lstat(const char *path, struct stat *buf);`
  - Abfragen der Attribute eines Eintrags (folgt symlinks nicht)

Das `struct stat`

- Inhalte des inode sind u.a.:
  - Geräte- und inode-Nummer
  - Eigentümer und Gruppenzugehörigkeit
  - Dateityp und -rechte
  - Dateigröße
  - Zeitstempel (letzte(r) Veränderung, Zugriff, ...)
  - ...

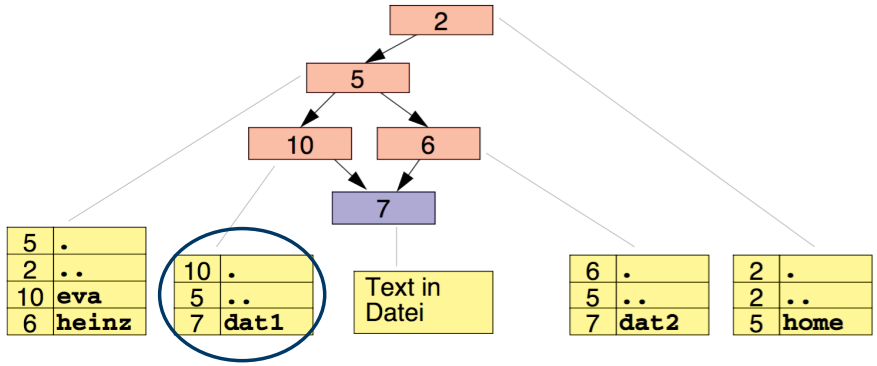
- Der Dateityp ist im Feld `st_mode` codiert
  - Reguläre Datei, Ordner, symbolischer Verweis (*symbolic link*), ...
  - Zur einfacheren Auswertung
    `S_ISREG(m)` - is it a regular file?
    `S_ISDIR(m)` - directory?
    `S_ISCHR(m)` - character device?
    `S_ISLNK(m)` - symbolic link?

Beispiel



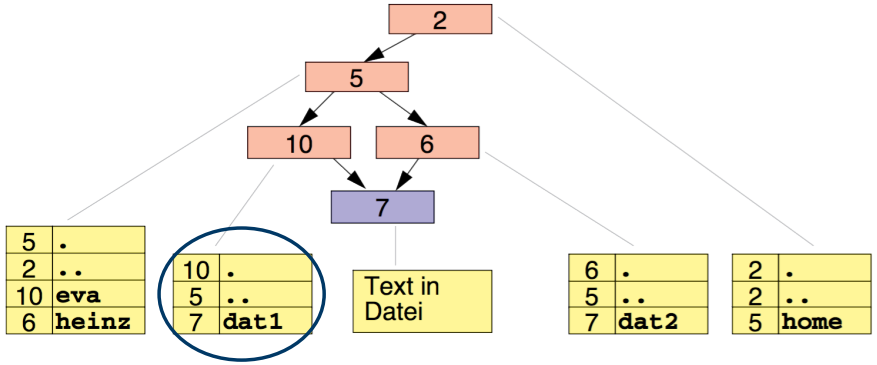

```
$> find /
/home
/home/eva
/home/eva/dat1
/home/heinz
/home/heinz/dat2
```


Beispiel



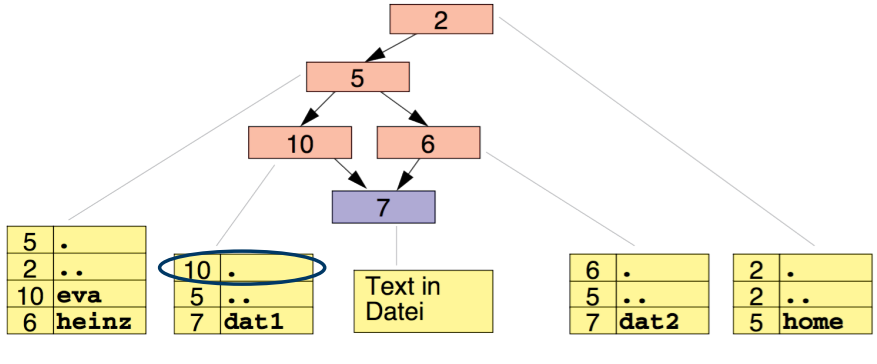

```
DIR *dir = opendir("/home/eva/");
if(dir == NULL) {
    perror("opendir");
    exit(EXIT_FAILURE);
}
```


Beispiel



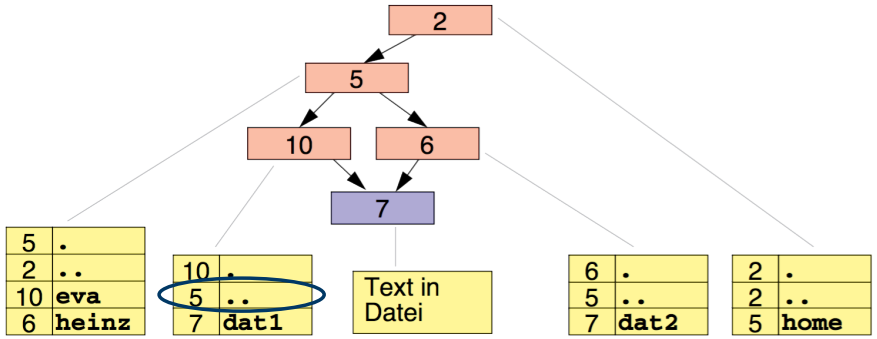

```
struct dirent *ent;
errno = 0;
while((ent = readdir(dir)) != NULL) {
    //...
    errno = 0;
}
if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```


Beispiel



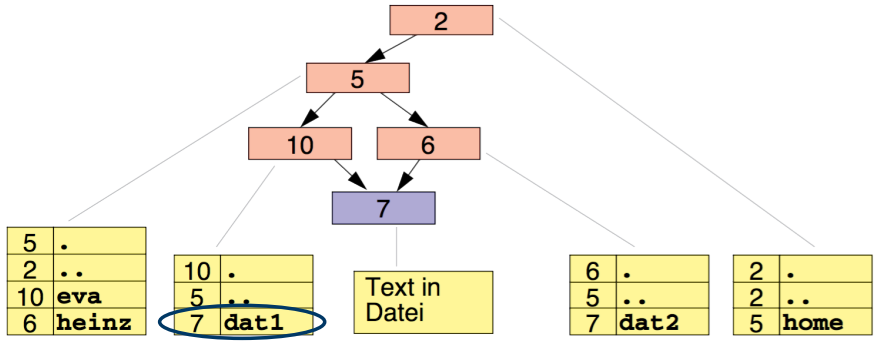

```
struct dirent *ent;
errno = 0;
while((ent = readdir(dir)) != NULL) {
    //...
    errno = 0;
}
if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```

Beispiel



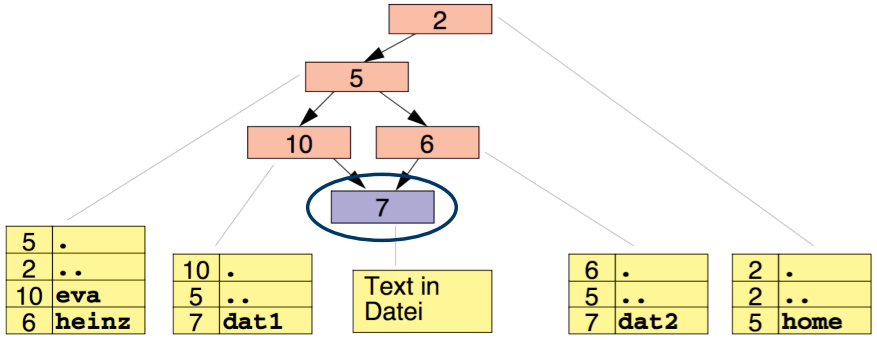

```c
struct dirent *ent;
errno = 0;
while((ent = readdir(dir)) != NULL) {
    //...
    errno = 0;
}
if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```


Beispiel




```c
struct dirent *ent;
errno = 0;
while((ent = readdir(dir)) != NULL) {
    //...
    errno = 0;
}
if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```

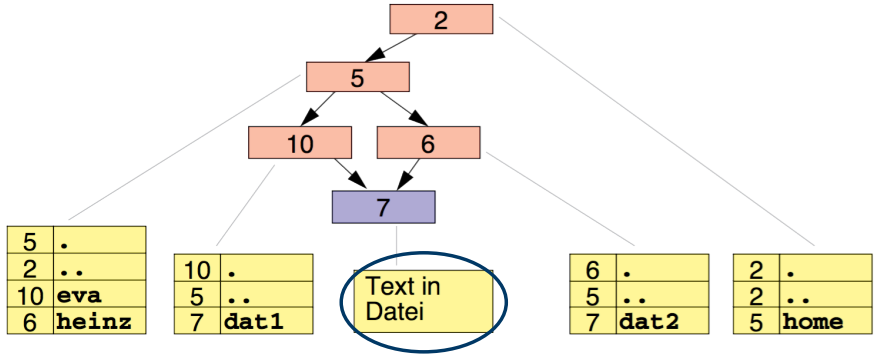

Beispiel




```c
char path[len];
strcpy(path, "/home/eva/");
strcat(path, ent->d_name); // d_name = "dat1"

struct stat buf;
if(lstat(path, &buf) == -1) {
    perror("lstat"); exit(EXIT_FAILURE);
}
```


Beispiel




```c
FILE *file = fopen(path, "r");
if(file == NULL) {
    perror("fopen");
    exit(EXIT_FAILURE);
}
```

Diskussion Fehlerbehandlung `stdout` (1)

Minimale Implementierung von `cat`:

```
01 FILE *f = fopen(path, "r");
02 if(f == NULL) die("fopen");
03
04 char buf[1024];
05 while(fgets(buf, 1024, file) != NULL) {
06     printf("%s", buf);
07 }
08
09 if(ferror(file) != 0) die("fgets");
10 if(fclose(file) != 0) die("fclose");
```

```
01 $> tail -n 1 num.dat
02 499999
03 $> ./cat num.dat && echo "Success" || echo "Failed"
04 1
05 2
06 [...]
07 499999
08 Success
```


Diskussion Fehlerbehandlung `stdout` (1)

Minimale Implementierung von `cat`:

```
01 FILE *f = fopen(path, "r");
02 if(f == NULL) die("fopen");
03
04 char buf[1024];
05 while(fgets(buf, 1024, file) != NULL) {
06     printf("%s", buf);
07 }
08
09 if(ferror(file) != 0) die("fgets");
10 if(fclose(file) != 0) die("fclose");
```

```
01 $> ./cat num.dat > dir/file && echo "Success" || echo "Failed"
02 Success
03 $> tail -n 1 dir/file
04 35984
```

- Warum wird nicht die ganze Datei geschrieben?
- Warum wird kein Fehler ausgegeben?

Diskussion Fehlerbehandlung `stdout` (2)

```
01 $> ls -lh num.dat
02 -rw-rw-r-- user group 3,3M Jan 01 00:00 num.dat
03
04 $> ls -lh dir/file
05 -rw-rw-r-- user group 200K Jan 01 00:00 tmp/file
06
07 $> df dir/
08 Filesystem      Size  Used Avail Use% Mounted on
09 tmpfs           200K  200K     0 100% /home/user/dir
```

- `stdout` kann in eine Datei umgeleitet werden
- Das Schreiben in eine Datei kann fehlschlagen
  - Kein Speicherplatz mehr
  - Fehlende Schreibberechtigung
  - Festplatte kaputt
- Fehlerbehandlung für *wichtige* Ausgaben
  - Was ist wichtig?
  - Fehlerbehandlung für `printf(3)` schwierig
- Für den Übungsbetrieb: Keine Fehlerbehandlung für `printf(3)`

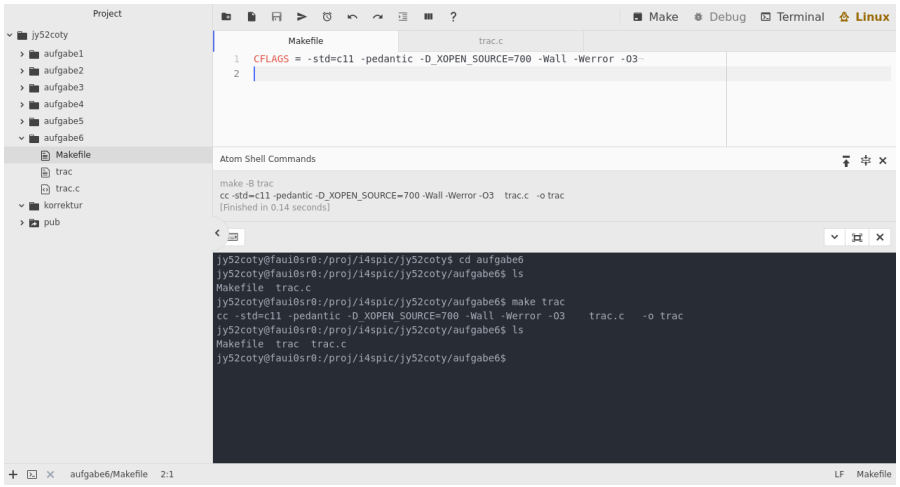
`make`

- `make`: Build-Management-Tool
- Baut automatisiert ein Programm aus den Quelldateien
- Baut nur die Teile des Programms neu, die geändert wurden

```
01 CFLAGS = -pedantic -Wall -Werror -O3 -std=c11 -D_XOPEN_SOURCE=700
02
03 trac.o: trac.c
04     gcc $(CFLAGS) -c -o trac.o trac.c
05
06 trac: trac.o
07     gcc $(CFLAGS) -o trac trac.o
```

- Objektdatei `trac.o` wird aus Quelldatei `trac.c` gebaut (Compiler)
- Binary `trac` wird aus Objektdatei `trac.o` gebaut (Linker)

make



- SPiC-IDE erkennt Makefiles (Make Button)
  - ⇒ alternativ: `make <binary>`
- `make` hat eingebaute Regeln (ausreichend für SPiC)
  - ⇒ nur Angabe der Compilerflags (`CFLAGS`) nötig

Aufgabe: trac

Aufgabe: trac

- TRAC: TRanslate Arbitrary Characters

```
01 $ ./trac <FIND> <REPLACE>
02
03 $ ./trac aie bei     # a -> b;  i -> e;  e -> i
04 Dies ist ein Test    # Eingabe
05 Deis est ien Tist    # Ausgabe
```

- Ähnlich wie das Kommando `tr(1)` in Unix-artigen Betriebssystemen
- Programmablauf
  - Kommandozeilenparameter prüfen
    - Gegebenenfalls Fehlermeldung ausgeben und Programm beenden
  - Zeichen von `stdin` bis EOF oder Fehler einlesen
    - Zeichen entsprechend Abbildung ersetzen
    - Zeichen ausgeben
  - Fehler erkennen und passende Fehlermeldung ausgeben
  - Programm (mit entsprechendem exit-Status) beenden

Nützliche Bibliotheksfunktionen (1)

- `int getchar(void);`
  - Einzelnes Zeichen von `stdin` einlesen

  → getchar(3)
- `int putchar(int c);`
  - Einzelnes Zeichen auf `stdout` ausgeben

  → putchar(3)
- `size_t strlen(const char *s);`
  - Länge einer Zeichenfolge bestimmen

  → strlen(3)
- `int fprintf(FILE *stream, const char *format, ↪ ...);`
  - Formatierte Ausgaben auf einem Dateikanal (z.B. `stderr`)

  → fprintf(3)

Nützliche Bibliotheksfunktionen (2)

- `int ferror(FILE *stream);`
  - Fehlerzustand eines Dateikanals abfragen
  - → ferror(3)
- `void perror(const char *s);`
  - Gibt die übergebene Zeichenfolge gefolgt von einer Beschreibung des letzten Fehlers auf `stderr` aus
  - Nutzt die globale `errno` Variable (⇒ nur für Bibliotheksfehler)
  - → perror(3)
- `void exit(int status);`
  - Beenden des Programms mit übergebenem exit-Status
  - → exit(3)

Mögliche Fehler

```
01 $ ./trac <FIND> <REPLACE>
```

- Bedienungsfehler
  - Falsche Anzahl an Argumenten
  - Unterschiedliche Länge von `FIND` und `REPLACE`
  - Selbes Zeichen kommt mehrfach in `FIND` vor
  - ⇒ Erklärung der Verwendung auf `stderr` ausgeben und Programm beenden
- Ein-/Ausgabefehler
  - Zugriff auf eine Datei ist nicht möglich (umgeleiteter Dateikanal)
  - ⇒ Fehler auf `stderr` ausgeben und Programm beenden

Fehlerbehandlung putchar(3)

```
01 if(putchar('A') == EOF) {
02   ...
03 }
```

- „fputc(), putc() and putchar() return the character written as an unsigned char cast to an int **or EOF on error.**"

Fehlerbehandlung getchar(3)

```
01 int c;
02 while ((c=getchar()) != EOF) {
03   ...
04 }
05
06 /* EOF oder Fehler? */
07 if(ferror(stdin)) {
08   /* Fehler */
09   ...
10 }
```

- „fgetc(), getc() and getchar() return the character read as an unsigned char cast to an int **or EOF on end of file or error.**"
- Wie kann man den Fehlerfall von EOF unterscheiden?

  ⇒ `ferror(3)`

Hands-on: sgrep

Screencast: https://www.video.uni-erlangen.de/clip/id/19103

Hands-on: sgrep (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 mehrerere Dateien nach einer Zeichenkette
- 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: sgrep (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`