

# Exercises in System Level Programming (SLP) – Summer Term 2024

## Exercise 10

Maximilian Ott

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



## Files & File Streams

## Presentation Assignment 5

### File Streams



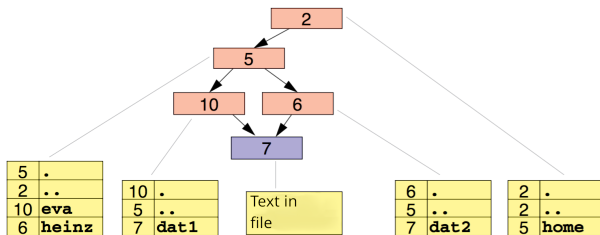
- In- and output implemented via buffered file streams
- `FILE *fopen(const char *path, const char *mode);`
  - Opens a file for reading or writing (depending on mode)
  - Returns a pointer to the created file stream
  - r Read
  - r+ Read & write
  - w Write; file is possibly created or contents are replaced
  - w+ Read & write; file is truncated to zero length or created
  - a Writing at the end of the file; file is possibly created
  - a+ Writing at the end of the file; reading at the start; file is possibly created
- `int fclose(FILE *fp);`
  - Flushes all buffered data to the file stream
  - Afterwards file is closed



- File streams opened by default
  - `stdin` Inputs
  - `stdout` Outputs
  - `stderr` Error messages
- `int fgetc(FILE *stream);`
  - Reads one character from stream
- `char *fgets(char *s, int size, FILE *stream);`
  - Reads at most size characters into a buffer
  - Stops on line break and EOF
- `int fputc(int c, FILE *stream);`
  - Writes one character to stream
- `int fputs(const char *s, FILE *stream);`
  - Writes a null terminated string (without the trailing null character)

## POSIX File-System Interface

2



- inode:** Contains file attributes & references to data blocks
- File:** Block with arbitrary data
- Directory:** Special file with pairs of names & inode number

- `DIR *opendir(const char *name);`
  - Opens a directory
  - Returns a pointer to the directory stream
- `struct dirent *readdir(DIR *dirp);`
  - Reads one entry from the directory stream and returns a pointer to the data structure `struct dirent`
- `int closedir(DIR *dirp);`
  - Closes the directory stream

3

4



```

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 };

```

- Taken from the man page readdir(3)
- Only d\_name and d\_ino are mandated by POSIX
- Relevant for us: file name (d\_name)

- Error handling by setting and checking of errno:

```

01 #include <errno.h>
02 // [...]
03 DIR *dir = opendir("/home/eva/"); // Error handling!!
04
05 struct dirent *ent;
06 while(1) {
07     errno = 0;
08     ent = readdir(dir);
09     if(ent == NULL) {
10         break;
11     }
12     // no further break instructions inside the loop
13     // [...]
14 }
15
16 // EOF or error?
17 if(errno != 0) { // error
18     // [...]
19 }
20 closedir(dir);

```

5

6



- readdir(3) returns **only name and inode number** of a directory entry
- Further information can be found in the **inode**
- `int stat(const char *path, struct stat *buf);`
  - Returns information about an entry (follows symlinks)
- `int lstat(const char *path, struct stat *buf);`
  - Returns information about an entry (does **not** follow symlinks)

- Contents of the inode are i.a.:
  - Device ID and inode number
  - Owner and group ID
  - File type and permissions
  - File size
  - Time stamp (last changed, accessed, ...)
  - ...
- The type of the file is encoded in the field `st_mode`
  - Regular file, directory, symbolic link, ...
  - For easy decoding
    - S\_ISREG(m) - is it a regular file?
    - S\_ISDIR(m) - is directory?
    - S\_ISCHR(m) - is character device?
    - S\_ISLNK(m) - is symbolic link?

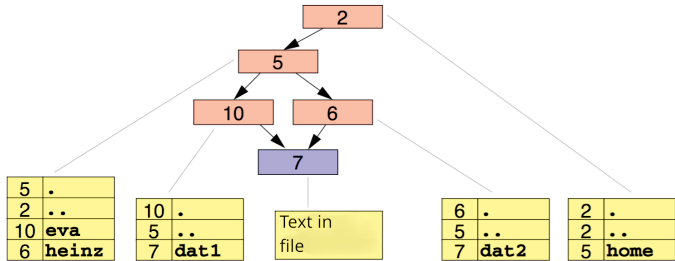
7

8

Example

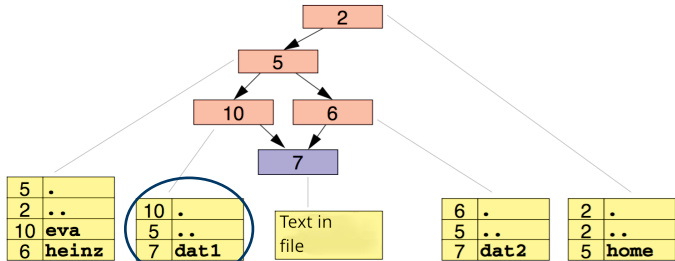


Example



```

01 $> find /
02 /home
03 /home/eva
04 /home/eva/dat1
05 /home/heinz
06 /home/heinz/dat2
  
```



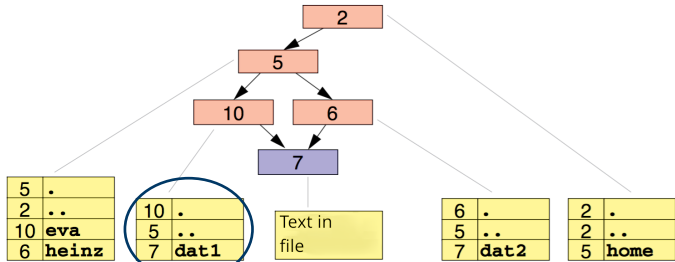
```

01 DIR *dir = opendir("/home/eva/");
02 if(dir == NULL) {
03     perror("opendir");
04     exit(EXIT_FAILURE);
05 }
  
```

Example

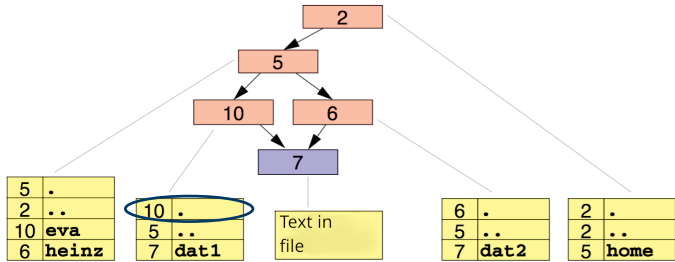


Example



```

01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
  
```



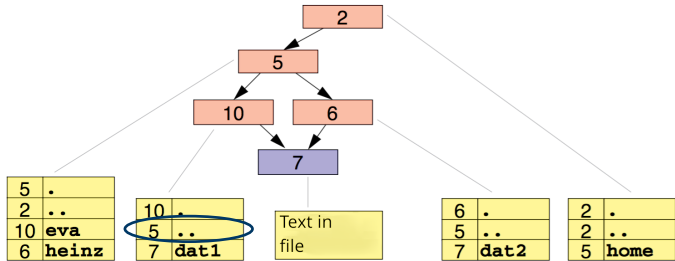
```

01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
  
```

Example



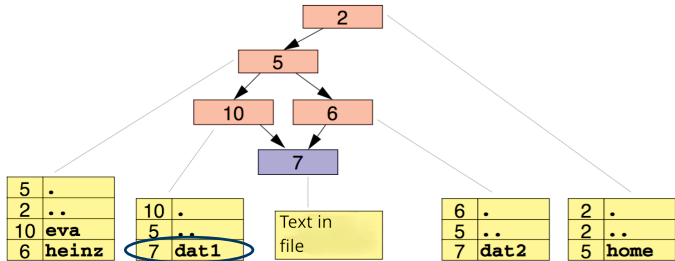
Example



```

01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }

```



```

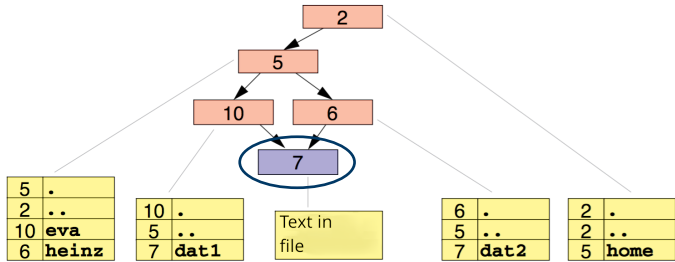
01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }

```

Example



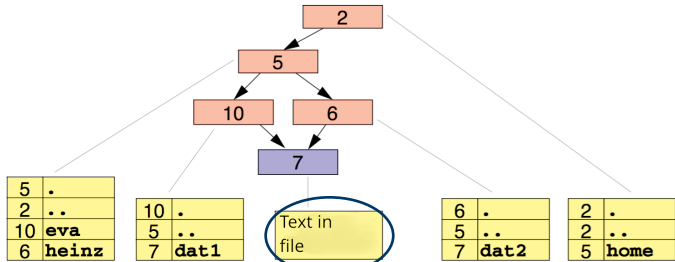
Example



```

01 char path[len];
02 strcpy(path, "/home/eva/");
03 strcat(path, ent->d_name); // d_name = "dat1"
04
05 struct stat buf;
06 if(lstat(path, &buf) == -1) {
07     perror("lstat"); exit(EXIT_FAILURE);
08 }

```



```

01 FILE *file = fopen(path, "r");
02 if(file == NULL) {
03     perror("fopen");
04     exit(EXIT_FAILURE);
05 }

```

## Discussion: Error Handling stdout (1)



Minimal implementation of 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(fcloses(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
```

10

## Discussion: Error Handling stdout (1)



Minimal implementation of 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(fcloses(file) != 0) die("fclose");
```

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

- Why is not the whole file written?
- Why is no error message printed?

10

## Discussion: Error Handling 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 can be redirected to a file
- Writing to a file can go wrong
  - No room for the data
  - Missing permissions
  - Broken drive
- Error handling for *relevant* outputs
  - What is important?
  - Error handling for printf(3) difficult
- For our exercises: No error handling required for printf(3)

11

## make



- make: Build management tool
- Automatically builds program from its source files
- Only rebuilds parts of the program that have changed

```
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
```

- Object file `trac.o` is built from source file `trac.c` (compiler)
- Binary `trac` is built from object file `trac.o` (linker)

12

## make



The screenshot shows the SPiC-IDE interface. On the left, a project tree shows a directory named 'aufgabe6' containing a 'Makefile'. The main editor displays the following Makefile content:

```
1 CFLAGS = -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror -O3
2
```

Below the editor, the 'Atom Shell Commands' panel shows the command 'make -B trac' and its output:

```
cc -std=c11 -pedantic -D_XOPEN_SOURCE=700 -Wall -Werror -O3 -o trac -o trac
(Refreshed in 0.14 seconds)
```

The terminal window at the bottom shows the execution of 'make trac' in the directory 'aufgabe6', resulting in the compilation of 'trac.c' into the binary 'trac'.

## Task: printdir

- SPiC-IDE detects makefiles (Make Button)
  - ⇒ alternatively: `make <binary>`
- make has built in rules (sufficient for SLP)
  - ⇒ only compiler flags (CFLAGS) have to be provided

13

## Assignment: printdir



- Iteration over all directories given as parameters
  - Output of all contained entries with size and name
  - Output of amount of regular files and their total size (per directory)
  - Relevant functions:
    - `opendir(3)`
    - `readdir(3)`
    - `stat(2)`
    - string functions
  - Error handling:
    - Informative error messages
    - Catch each wrong user input
- ⇒ Always assume the (vicious) SUP (Stupidest User Possible) ☹

14

## Hands-on: sgrep

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



```
01 # Usage: ./sgrep <text> <files...>
02 $ ./sgrep "SLP" exam.tex task.tex
03 Exam of lecture SLP
04 SLP task
05 SLP is cool
```

- Simple version of command line tool `grep(1)`
- Searches in multiple files for a given string
- Procedure:
  - Read files line by line
  - Search for the string in the line
  - Possibly print out the line on `stdout`
- Pay attention to sensible error handling
  - Report about missing files and continue with next file
  - Write error messages to `stderr`



- Helpful functions:
  - `fopen(3)` ⇒ Opens a file
  - `fgets(3)` ⇒ Reads one line
  - `fputs(3)` ⇒ Prints one line
  - `fclose(3)` ⇒ Closes a file
  - `strstr(3)` ⇒ Searches for a sub-string

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

```
01 # Usage: ./sgrep [-i] <text> <files...>
02 $ ./sgrep -i "SLP" exam.tex task.tex
03 exam.tex:13: Exam of lecture SLP
04 task.tex:32: SLP task
05 task.tex:56: SLP is cool
```

- Extensions
  - Implement `strstr(3)` on your own
  - Print file name and line number in front of every line
  - Ignore capitalization with the option `-i`