Exercises in System Level Programming (SLP) – Summer Term 2026

Exercise 10

Arne Vogel
Maxim Ritter von Onciul
Eva Dengler

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

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Faculty of Engineering

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

In-/Output

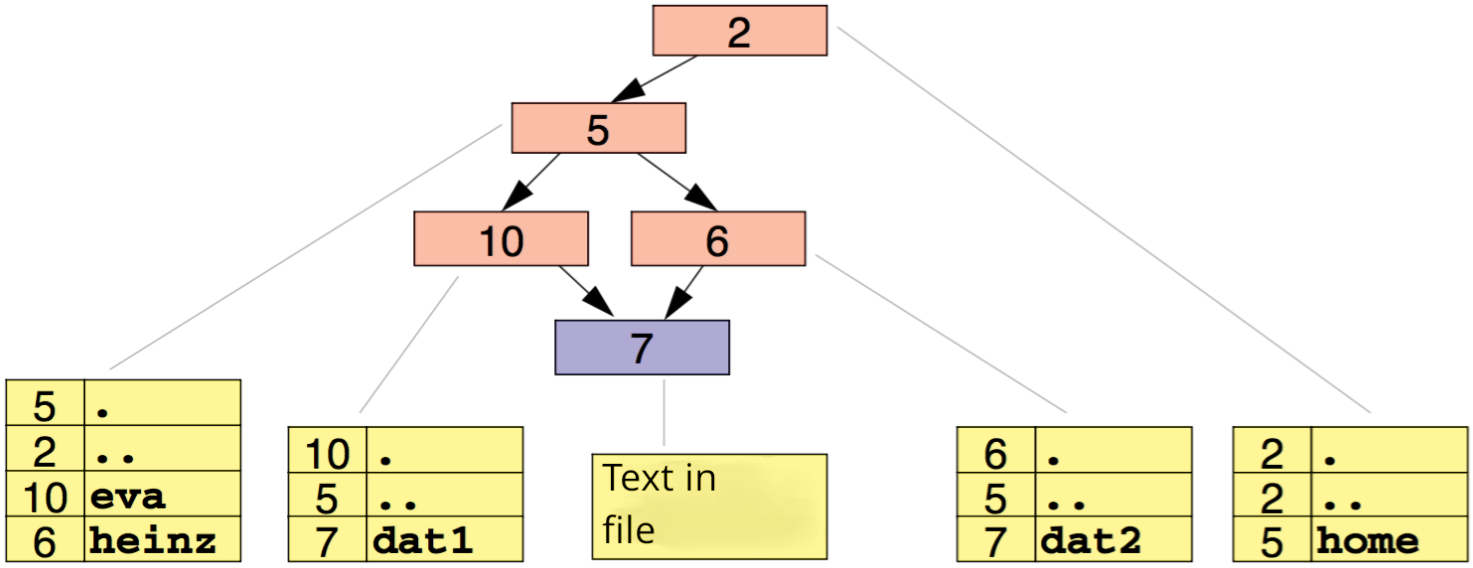
- File streams opened by default

  `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

File System: Files, Directories, Inodes



inode: Contains file attributes & references to data blocks

File: Block with arbitrary data

Directory: Special file with pairs of names & inode number

opendir, closedir, readdir

- `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

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 };
```

- 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 for `readdir(3)`

- 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);
```

Determine File Attributes: `stat(2)`

- `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)

The `struct stat`

- 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

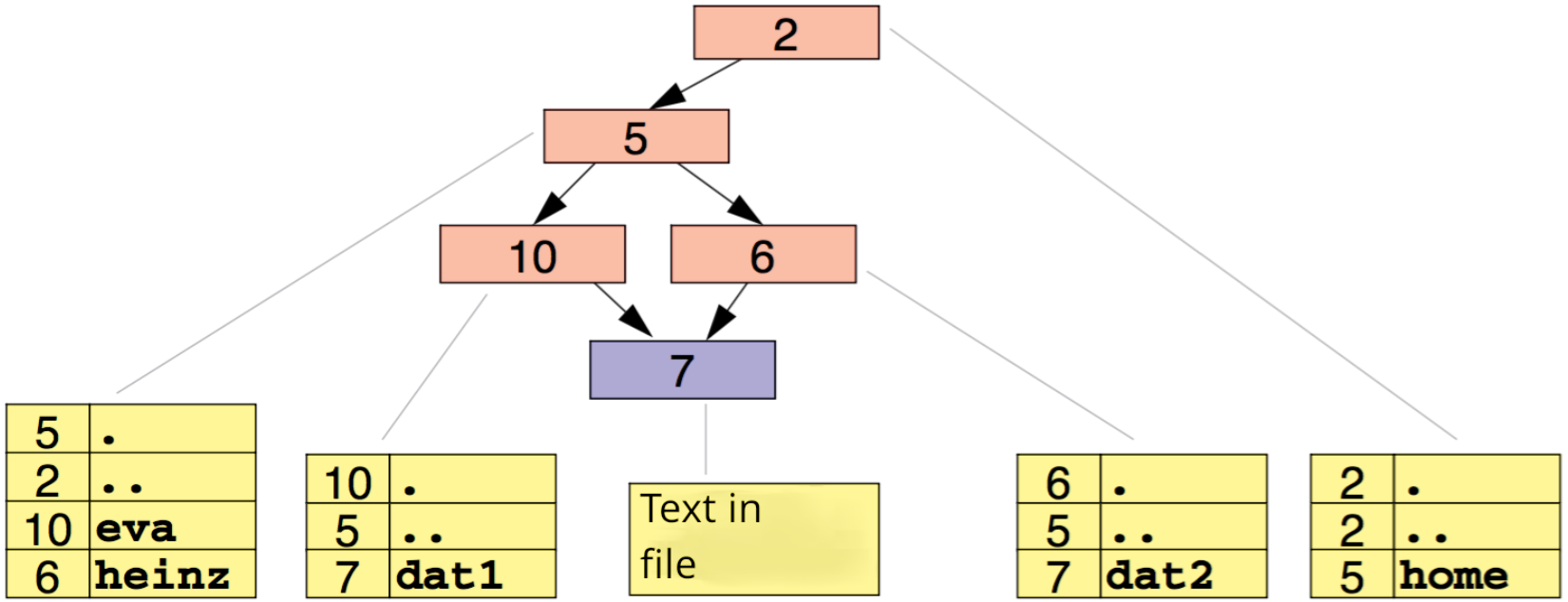
    `S_ISREG(m)` - is it a regular file?

    `S_ISDIR(m)` - is directory?

    `S_ISCHR(m)` - is character device?

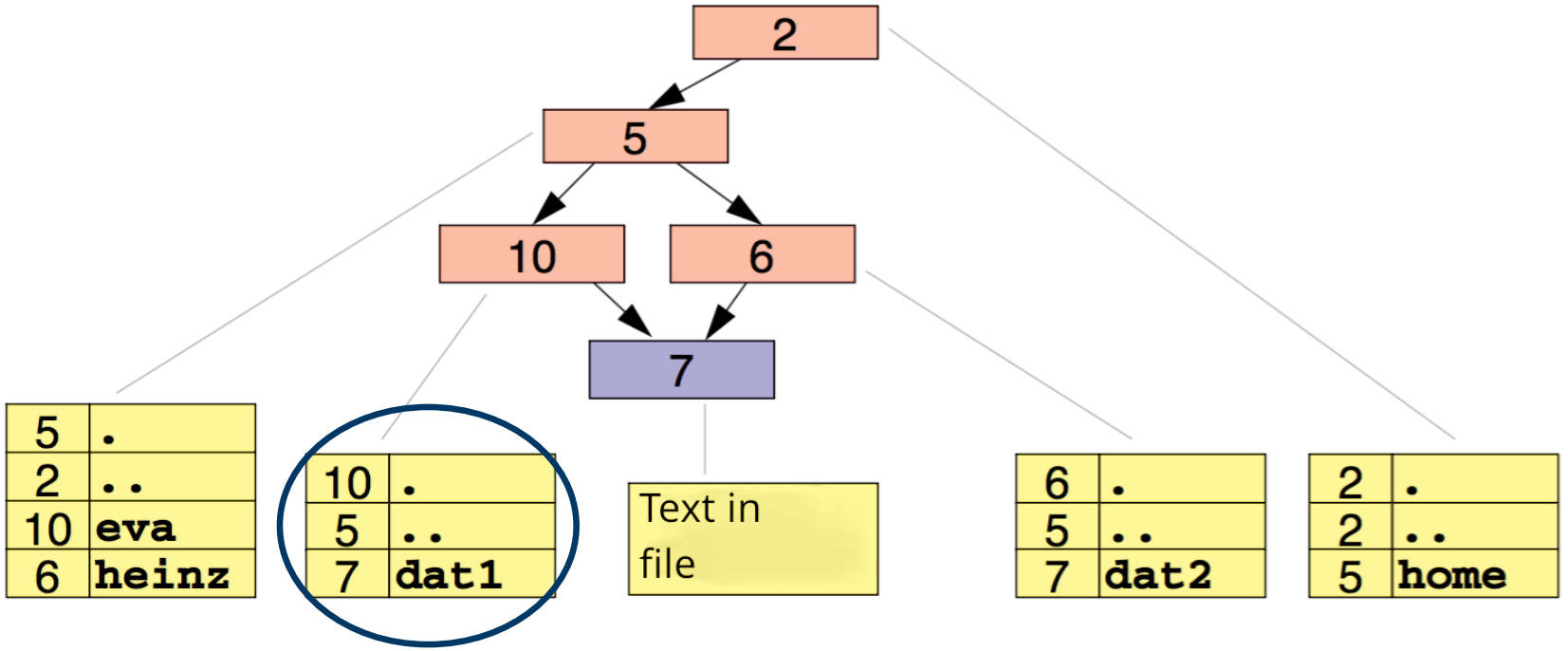
    `S_ISLNK(m)` - is symbolic link?

Example



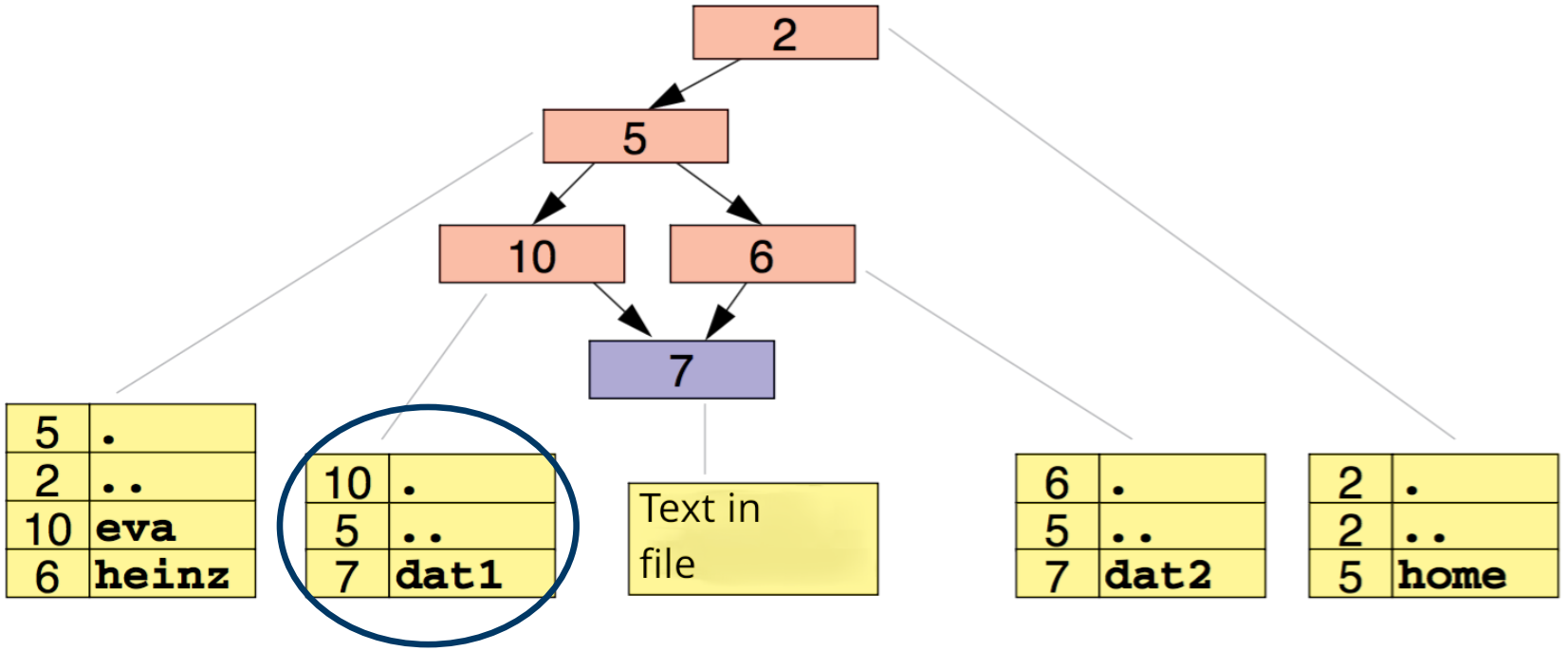

```
$> find /
/home
/home/eva
/home/eva/dat1
/home/heinz
/home/heinz/dat2
```

Example



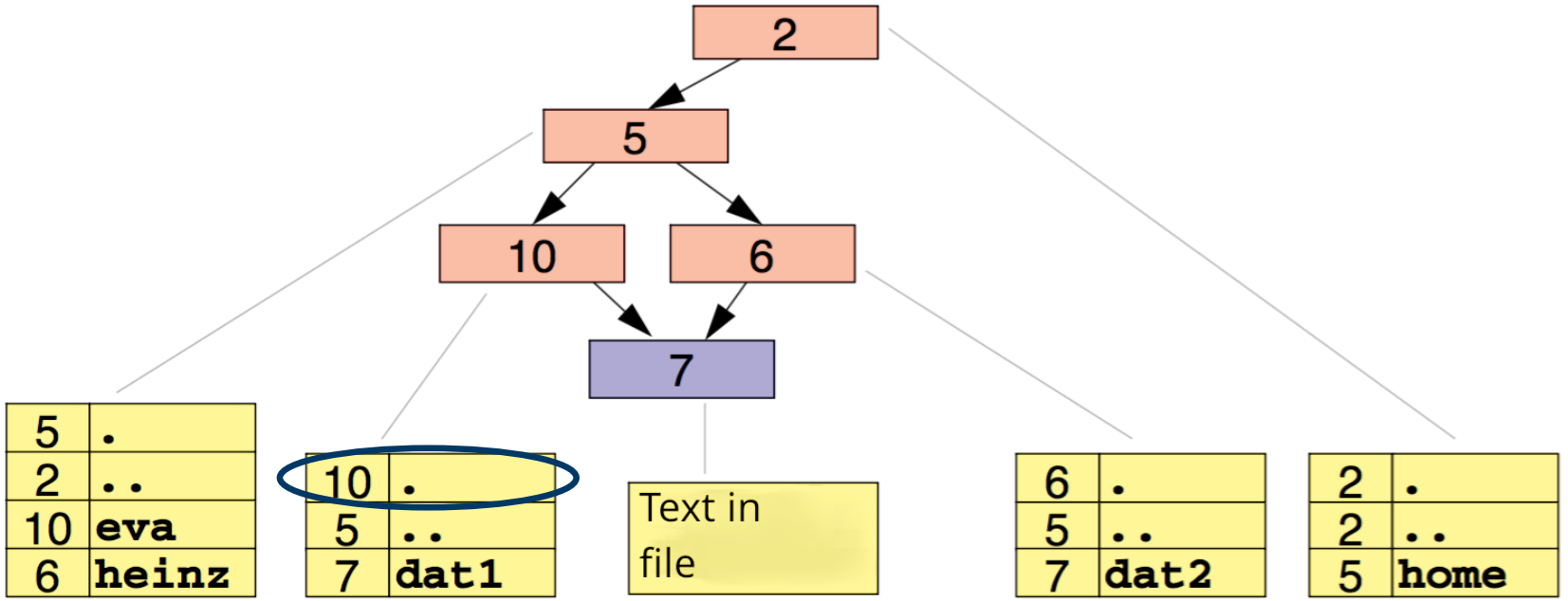

```
01 DIR *dir = opendir("/home/eva/");
02 if(dir == NULL) {
03     perror("opendir");
04     exit(EXIT_FAILURE);
05 }
```

Example



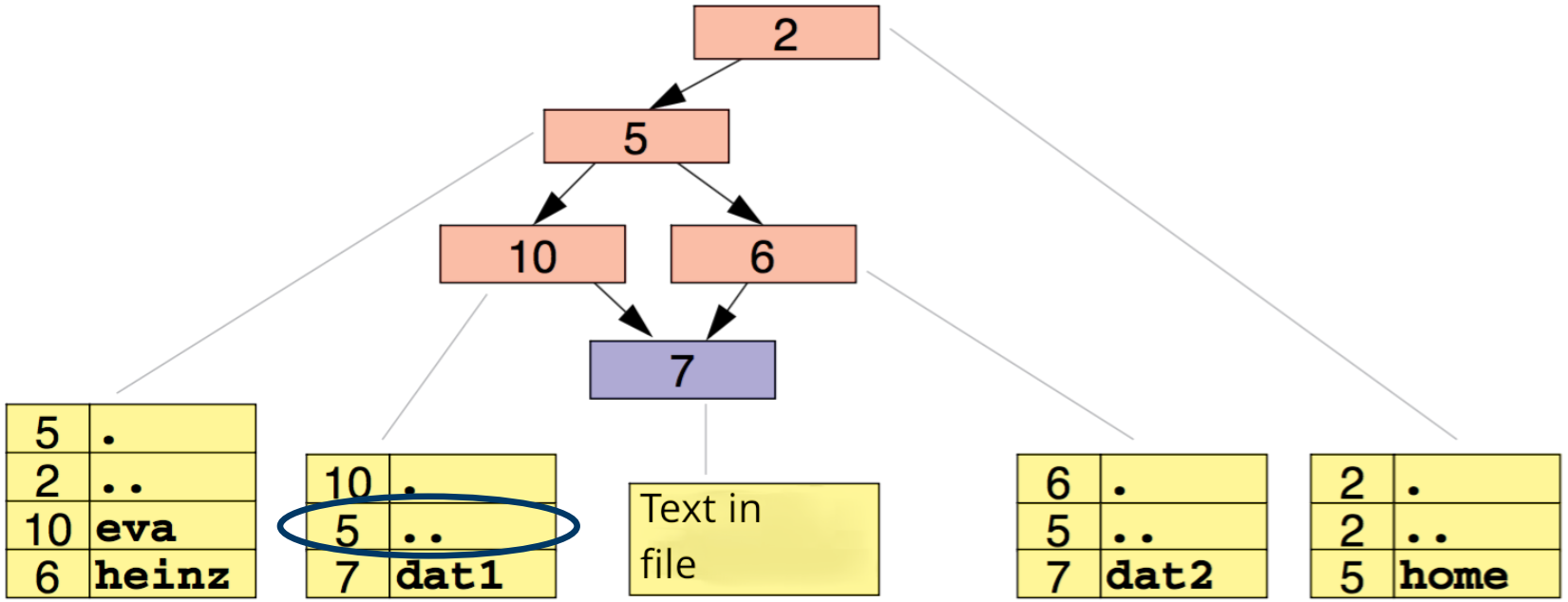

```
01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```

Example



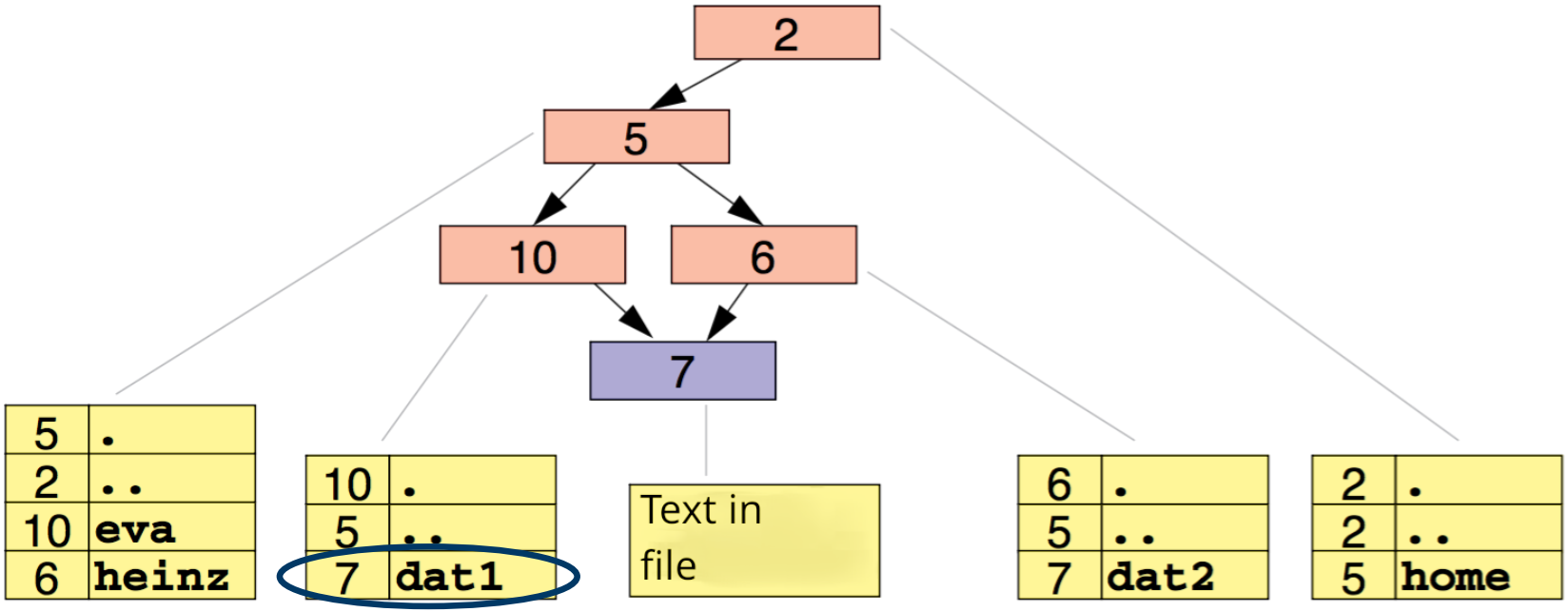

```
01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```

Example



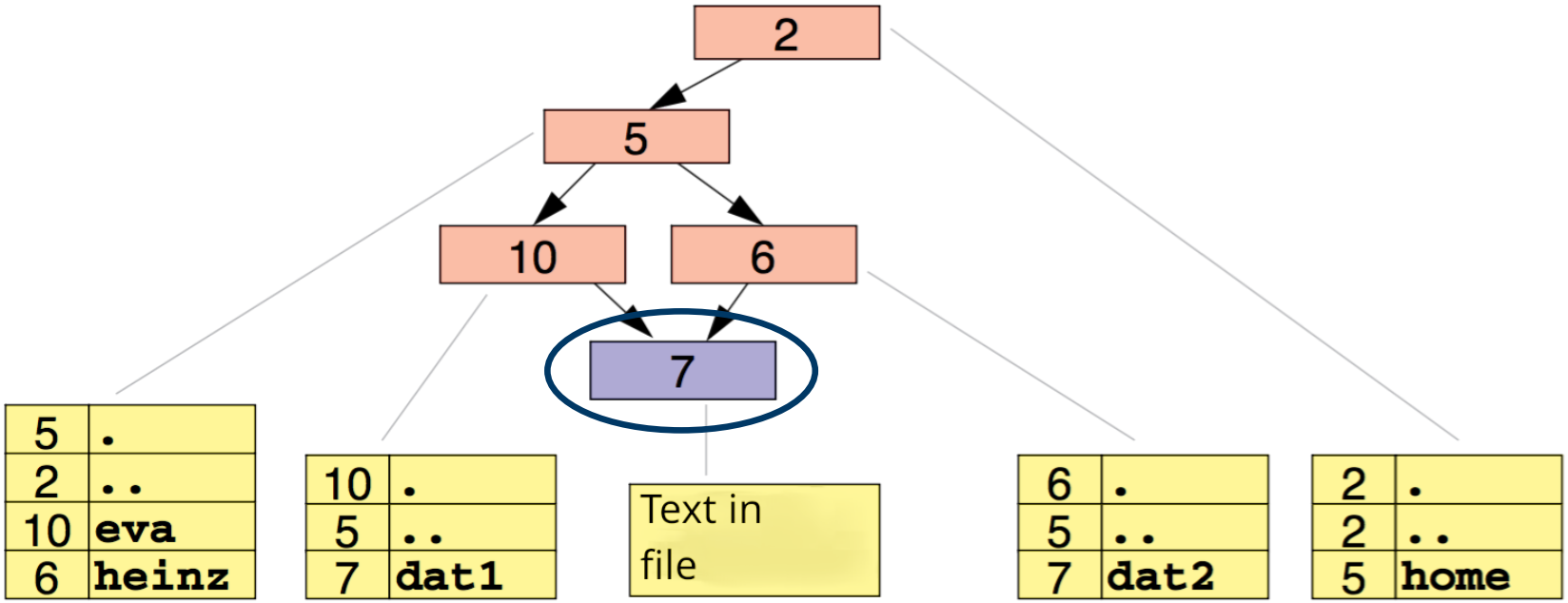

```
01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```

Example



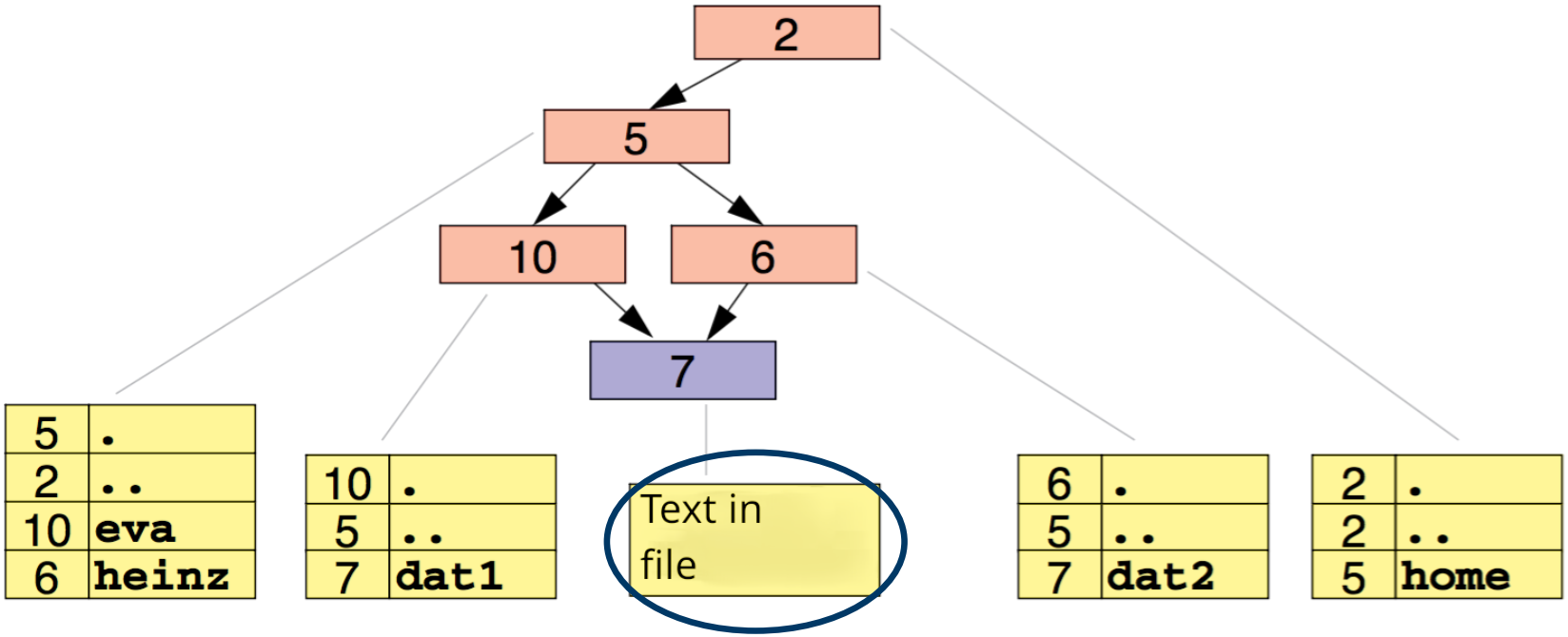

```
01 struct dirent *ent;
02 errno = 0;
03 while((ent = readdir(dir)) != NULL) {
04     //...
05     errno = 0;
06 }
07 if(errno != 0) { perror("readdir"); exit(EXIT_FAILURE); }
```

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 }
```

Example




```
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, f) != NULL) {
06     printf("%s", buf);
07 }
08
09 if(ferror(f) != 0) die("fgets");
10 if(fclose(f) != 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
```

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, f) != NULL) {
06     printf("%s", buf);
07 }
08
09 if(ferror(f) != 0) die("fgets");
10 if(fclose(f) != 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?

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)`

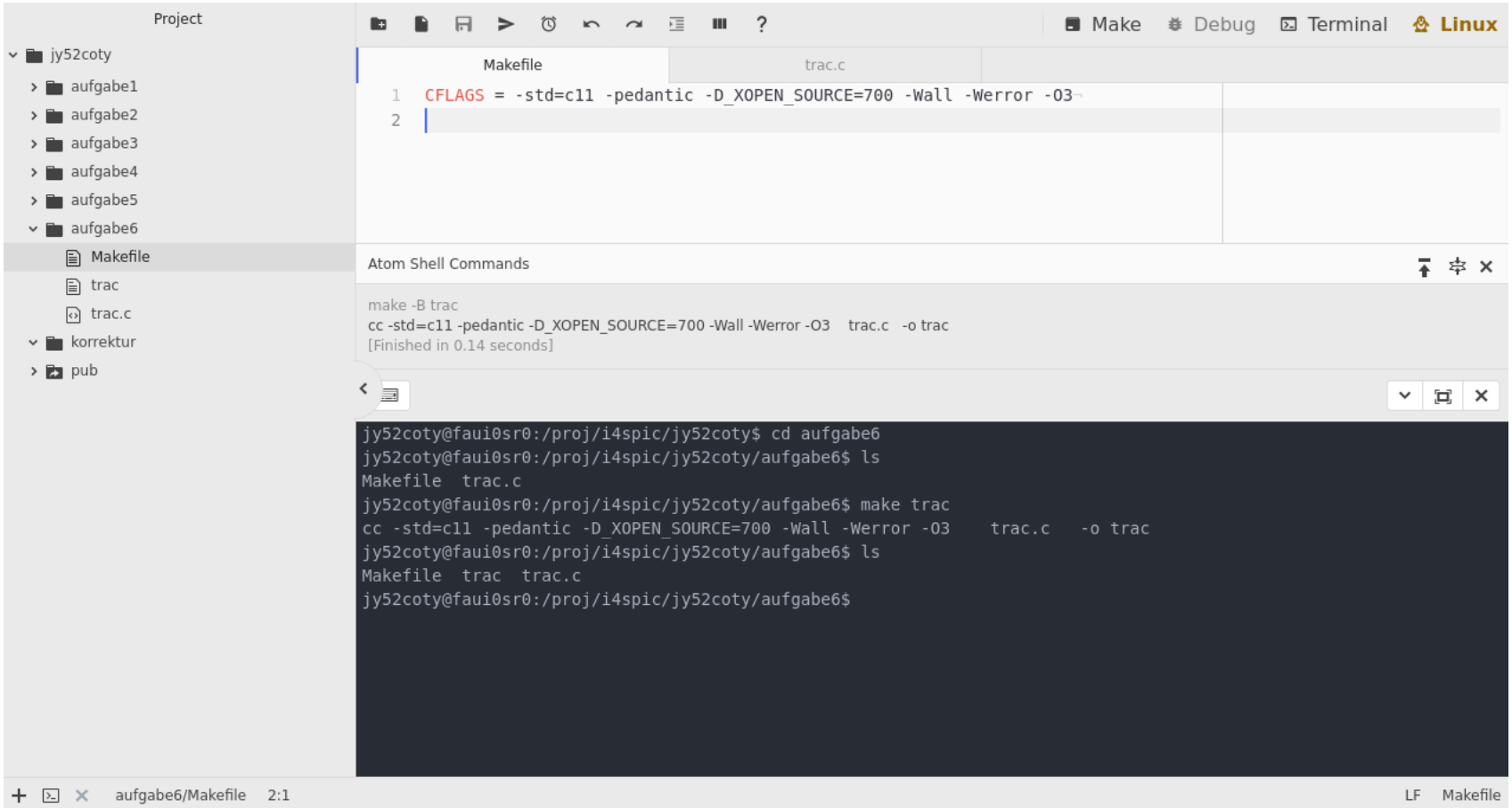
make

- 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)

make



- 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

Task: printdir

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 (malicious) SUP (Stupidest User Possible) ☺

Hands-on: sgrep

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

Hands-on: sgrep (1)

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

- Simplified version of command line tool `grep(1)`
- Searches multiple files for a given string

- Procedure:
  - Read files line by line
  - Search for the string in every line
  - Possibly print out the line on `stdout`
- Pay attention to sensible error handling
  - Report missing files and continue with next file
  - Display error messages using `stderr`

Hands-on: sgrep (2)

- 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`