System-Level Programming

26 File Systems – UNIX

Peter Wägemann

Lehrstuhl für Informatik 4
Systemsoftware

Friedrich-Alexander-Universität
Erlangen-Nürnberg (FAU)

Summer Term 2025

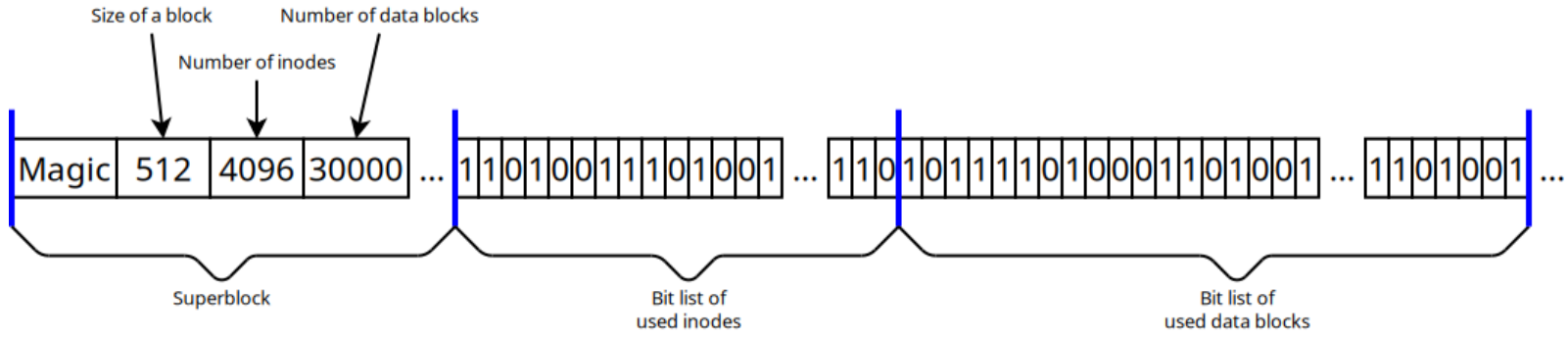
http://sys.cs.fau.de/lehre/ss25

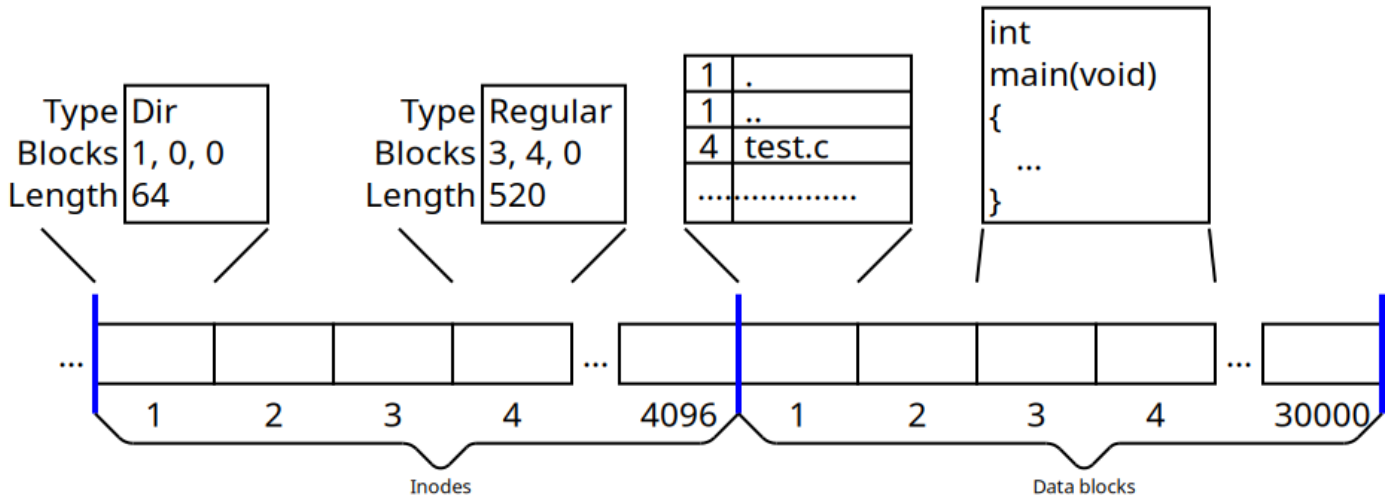
File System – Example Linux/UNIX

- File
  - simple, unstructured set of bytes
  - arbitrary content; content is transparent for the operating system
  - can be dynamically extended
- File attributes
  - the operating system manages a set of attributes for each file (permissions, size, time of access, data blocks, ...)
  - the attributes are saved in a special management structures, called the *file header*
    – Linux/UNIX: *Inode*
    – Windows NTFS: *Master File Table* entry
- Namespace
  - flat namespace: inodes are enumerated
  - hierarchical namespace: directory structure maps the file and path names to the inode numbers

File-System Structure

- Structure on medium (simplified)

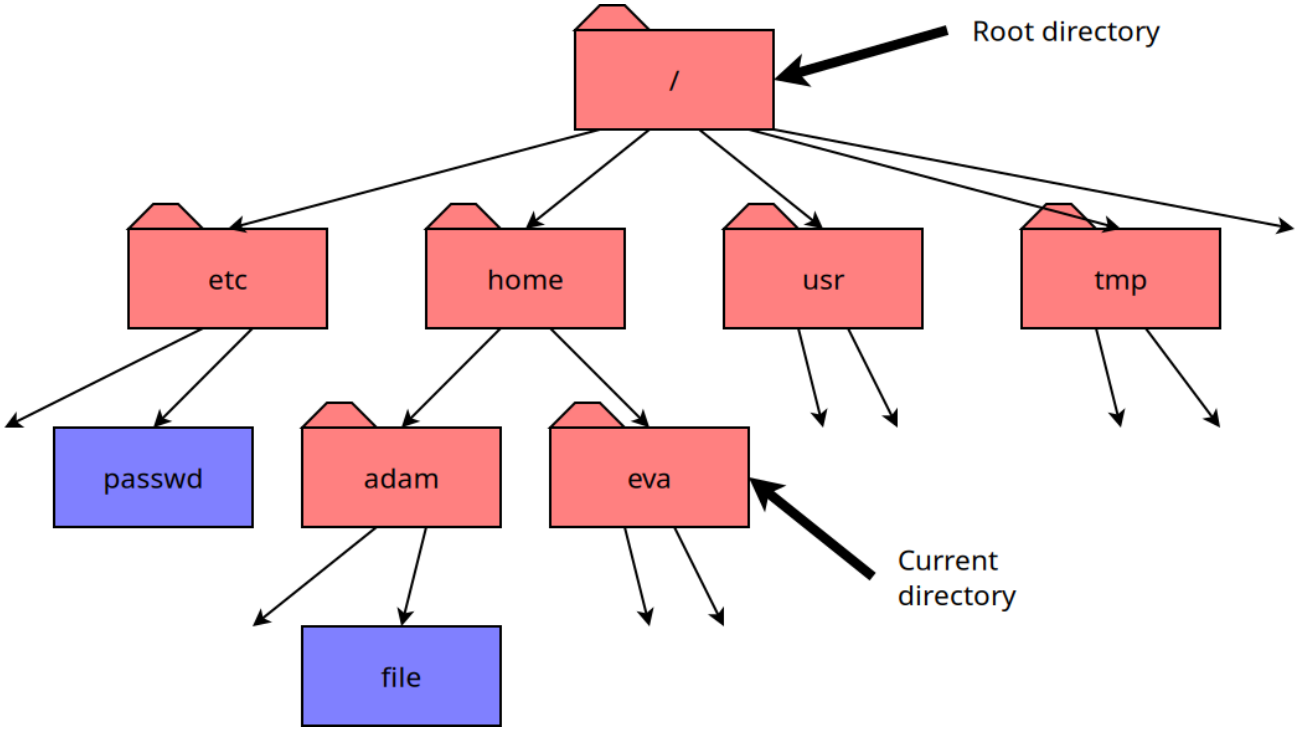




- `mkfs` creates an empty structure; `fsck` verifies the structure

Path Names

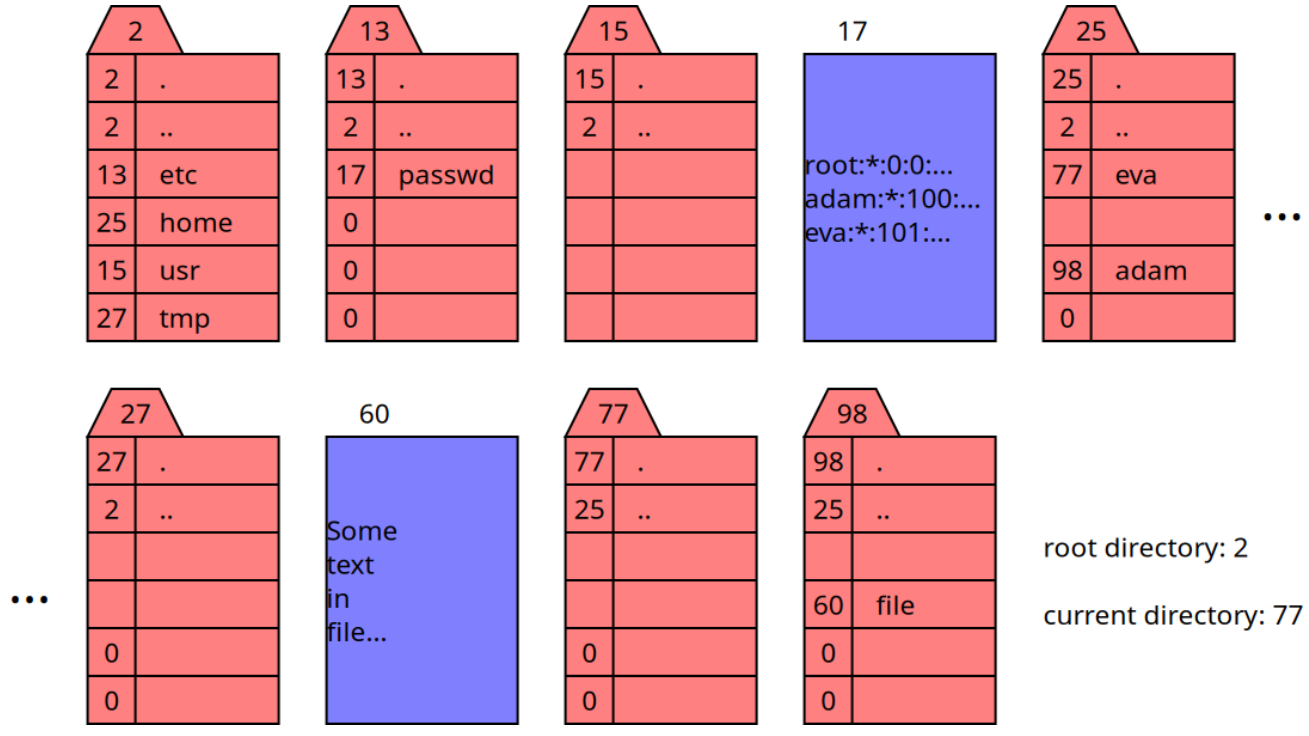
- Tree structure



- Paths
  - e. g., `/home/adam/datei`, `/tmp`, `../adam/datei`
  - `/` represents separator (*forward slash*)
  - beginning `/` stands for the root directory; else, implicit start in the current working directory

Path Names (2)

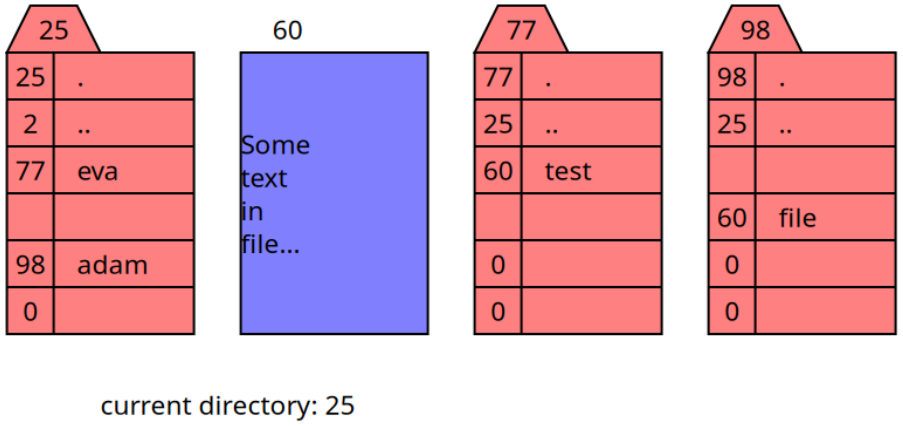
- Actual “tree structure”

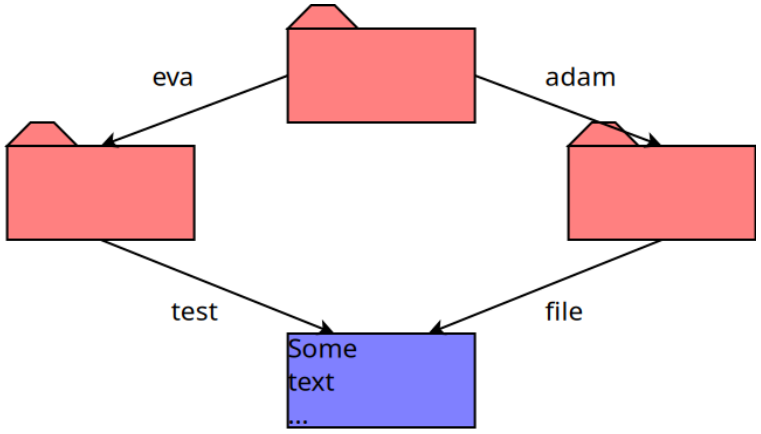


- Example, resolving the path “`../adam/datei`”:
  - 77 + “`../adam/datei`” $\rightsquigarrow$ 25 + “`adam/datei`”
  - 25 + “`adam/datei`” $\rightsquigarrow$ 98 + “`datei`”
  - 98 + “`datei`” $\rightsquigarrow$ 60

Path Names (3)

- There can exist multiple references (**hard links**) to the same file:

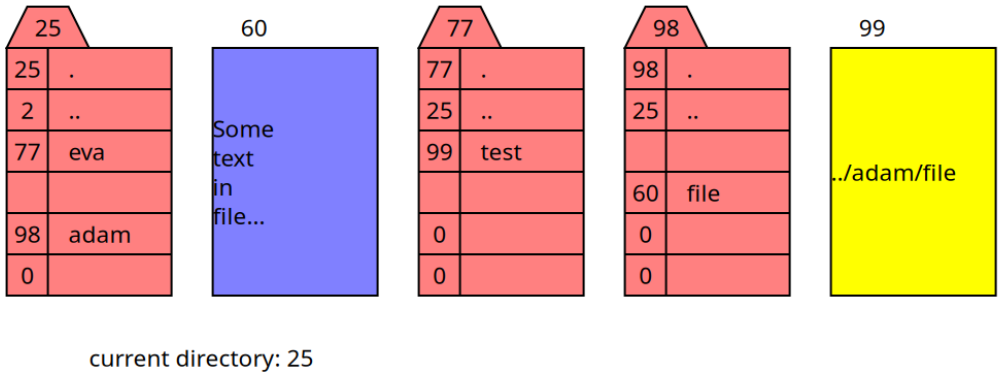


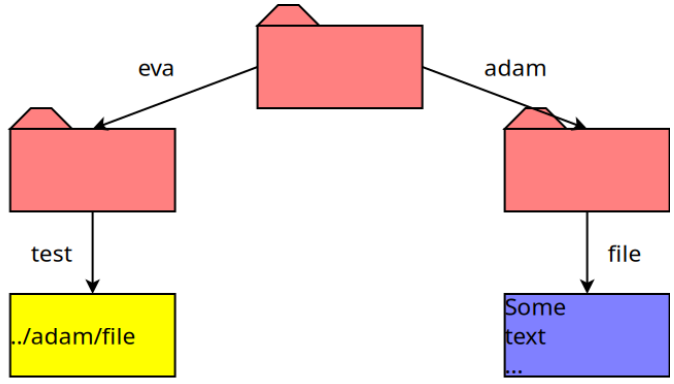


- Example, resolving the path "`adam/datei`":
  - 25 + "`adam/datei`" $\rightsquigarrow$ 98 + "`datei`"
  - 98 + "`datei`" $\rightsquigarrow$ 60
- Example, resolving the path "`eva/test`":
  - 25 + "`eva/test`" $\rightsquigarrow$ 77 + "`test`"
  - 77 + "`test`" $\rightsquigarrow$ 60
- A file is deleted, if no more references exist that refer to it.

Path Names (4)

- There can exist multiple symbolic references (**symbolic links**) to a file or directory:





- Example, resolving the path "`eva/test`":
  - 25 + "`eva/test`" $\rightsquigarrow$ 77 + "`test`"
  - 77 + "`test`" $\rightsquigarrow$ 99 $\rightsquigarrow$ 77 + "`../adam/datei`"
  - 77 + "`../adam/datei`" $\rightsquigarrow$ 25 + "`adam/datei`"
  - 25 + "`adam/datei`" $\rightsquigarrow$ 98 + "`datei`"
  - 98 + "`datei`" $\rightsquigarrow$ 60
- A symbolic name persists even if the file or directory does not yet exist or does not longer exist.

Owners and Permissions

- Owner
  - Each owner is represented by a unique user identification number (UID).
  - A user can belong to one or more user groups, each of which is represented by a unique group identification number (GID).
  - A file or directory is mapped to exactly one user and one group.
- Permissions on files
  - Read, write, execute (only by the owner)
  - Can be individually modified for the owner, for members of the group or for all others.
- Permission on directories
  - Read, write (creating and deleting of files/directories), right of pass-through
  - Write permission can be restricted for each file individually.

Inodes

- Attributes (permissions, owners, etc.) of a file, or of a directory are stored in the so called **inodes** (simplified):

```
int st_mode;        /* type and permissions */
int st_nlink;       /* number of Hard Links */
int st_uid;         /* owner */
int st_gid;         /* group */
long st_size;       /* length of the file in bytes */
int st_block[...]; /* list of the (indirect) blocks */
time_t st_atime;    /* last time of reading */
time_t st_mtime;    /* last time of modification */
time_t st_ctime;    /* last change of attributes */
```

- **File system** assigns each inode a number and a storage location (disk/partition):

```
int st_ino;         /* inode number */
int st_dev;         /* Disk/partition number */
```


Programming Interface for Inodes

- `stat`, `lstat` return file attributes from an inode
- Function interface:

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
```

- Arguments:
  - `path`: path name
  - `buf`: pointer to a buffer, which will be filled with the information from the inode
- Return value:
  - 0 if OK
  - -1 if an error occurred (`errno` variable contains error number)
- Example:

```
struct stat buf;
stat("/etc/passwd", &buf); /* Error handling...! */
printf("Number of the inode: %d\n", buf.st_ino);
```

Programming Interface for Directories

- Handling links or directories
  - Creating (an empty directoy)

```
int mkdir(const char *path, mode_t mode);
```

  - Deleting (an empty directory)

```
int rmdir(const char *path);
```

  - Creating a hard link

```
int link(const char *existing, const char *new);
```

  - Creating a symbolic link

```
int symlink(const char *existing, const char *new);
```

  - Deleting a link (and possibly the file)

```
int unlink(const char *path);
```

  - Reading a symbolic link

```
int readlink(const char *path, char *buf, int size);
```


Programming Interface for Directories (2)

- Reading directories (interface of the Linux kernel)
  - open(2), getdents(2), close(2)
  - Linux-specific
- Reading directories (interface of the C library)
  - Opening a directory

```
DIR *opendir(const char *path);
```

  - Reading an entry

```
struct dirent *readdir(DIR *dirp);
```

  - Closing a directory

```
int closedir(DIR *dirp);
```

  - Structure `struct dirent` (simplified)

```
struct dirent {
    int d_ino;                  /* Number of the inode */
    char d_name[NAME_MAX + 1]; /* Name */
};
```

Directories (3): `opendir/closedir`

- Function interface:

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *path);
int closedir(DIR *dirp);
```

- Argument of `opendir`:
  - `path`: name of the directory
- Return value of `opendir`:
  - Pointer to a structure of type `DIR` if OK
  - `NULL` if an error occurred (`errno` variable contains error number)

Directories (4): `readdir`

- Function interface:

```
#include <sys/types.h>
#include <dirent.h>

struct dirent *readdir(DIR *dirp);
```

- Argument:
  - `dirp`: pointer to `DIR` data structure
- Return value:
  - Pointer to data structure of type `struct dirent`, if OK
  - NULL, if directory has been written entirely (`errno`-variable remains unchanged)
  - NULL, if an error occurred (`errno` variable contains error number)
- Note: The memory for `struct dirent` can possibly be overwritten by the next `readdir` call!

Directories (5): Example

- Output all file names in the current directory (".")

```
#include <sys/types.h>
#include <dirent.h>

DIR *dirp;
struct dirent *de;
int ret;

dirp = opendir(".");                    // opening cur. dir
if (dirp == NULL) ...                   // error

while (1) {
  errno = 0;
  de = readdir(dirp);                   // reading entry
  if (de == NULL && errno != 0) ...     // error
  if (de == NULL) break;                // end reached

  printf("%s\n", de->d_name);
}

ret = closedir(dirp);                   // closing directory
if (ret < 0) ...                        // error
```

Programming Interface for Files

- Function interface:

```
#include <sys/types.h>
#include <fcntl.h>

int open(const char *path, int flags, ...);

int close(int fd);

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```


Files (2): Example

- Copy program

```
#include <fcntl.h>

int ret;

int src_fd = open("src", O_RDONLY);
if (src_fd < 0) ...          // error
int dst_fd = open("dst", O_CREAT | O_TRUNC | O_WRONLY, 0777);
if (dst_fd < 0) ...          // error

while (1) {
  char buf[1024];
  len = read(src_fd, buf, sizeof(buf));
  if (len < 0) ...           // error
  if (len == 0) break;
  ret = write(dst_fd, buf, len);
  if (ret < 0) ...           // error
}

ret = close(dst_fd);
if (ret < 0) ...             // error
ret = close(src_fd);
if (ret < 0) ...             // error
```

Files (3)

- A call to the `write` function has to
  - verify the file descriptor whether file is opened and/or writable
  - verify the length and address of the buffer
  - determine the block(s) of the medium which have to be written to
  - read the block(s) from the medium (unless the whole block needs to be written)
  - overwrite the required bytes in the block(s)
  - transfer the block(s) back to the medium
  - modify the attributes (last change, length of the file)
- `write` is therefore a system call
- ⇒ `write` is a costly operation, (`read` analogously)!
- ⇒ Improvement: read/write many bytes at once (ideally: multiples of the block size)
- ⇒ Use `fopen`, `fclose`, `fread`, `fwrite`, `getchar`, `putchar`, `fscanf`, `fprintf`, ... functions from the C library!

Special Files

- Peripheral devices (disks, printers, CD, terminal, scanner, ...) are represented as special files (`/dev/sda`, `/dev/lp0`, `/dev/cdrom0`, `/dev/tty`, ...)
- Their inode contains
  - Type:
    - Block-oriented devices (drives, CD, DVD, SSD, ...)
    - Character-oriented devices (printer, terminal, scanner, ...)
  - Instead of block numbers:
    - Major number: type of the device (disk, printer, ...)
    - Minor number: number of the device (3^{rd} printer, 5^{th} terminal, ...)
- Opening a device creates a possibly exclusive connection to the device, provided by drivers
- Devices can be accessed with `read`, `write`, and `ioctl` operations

Special Files (2): Example

- Output to a printer

```
#include <linux/lp.h>
int fd, ret;

/* Establish connection to printer 0. */
fd = open("/dev/lp0", O_WRONLY);
if (fd < 0) ...

/* Get status of the printer. */
ret = ioctl(fd, LPGETSTATUS, &state);
if (ret < 0) ...
if (state & LP_POUTPA) {
  fprintf(stderr, "Out of paper!\n"); exit(1);
}

/* Write to the printer. */
ret = write(fd, "Hallo, Drucker!\n\f", 17);
if (ret < 0) ...

/* Close connection. */
ret = close(fd);
if (ret < 0) ...
```

Partitions

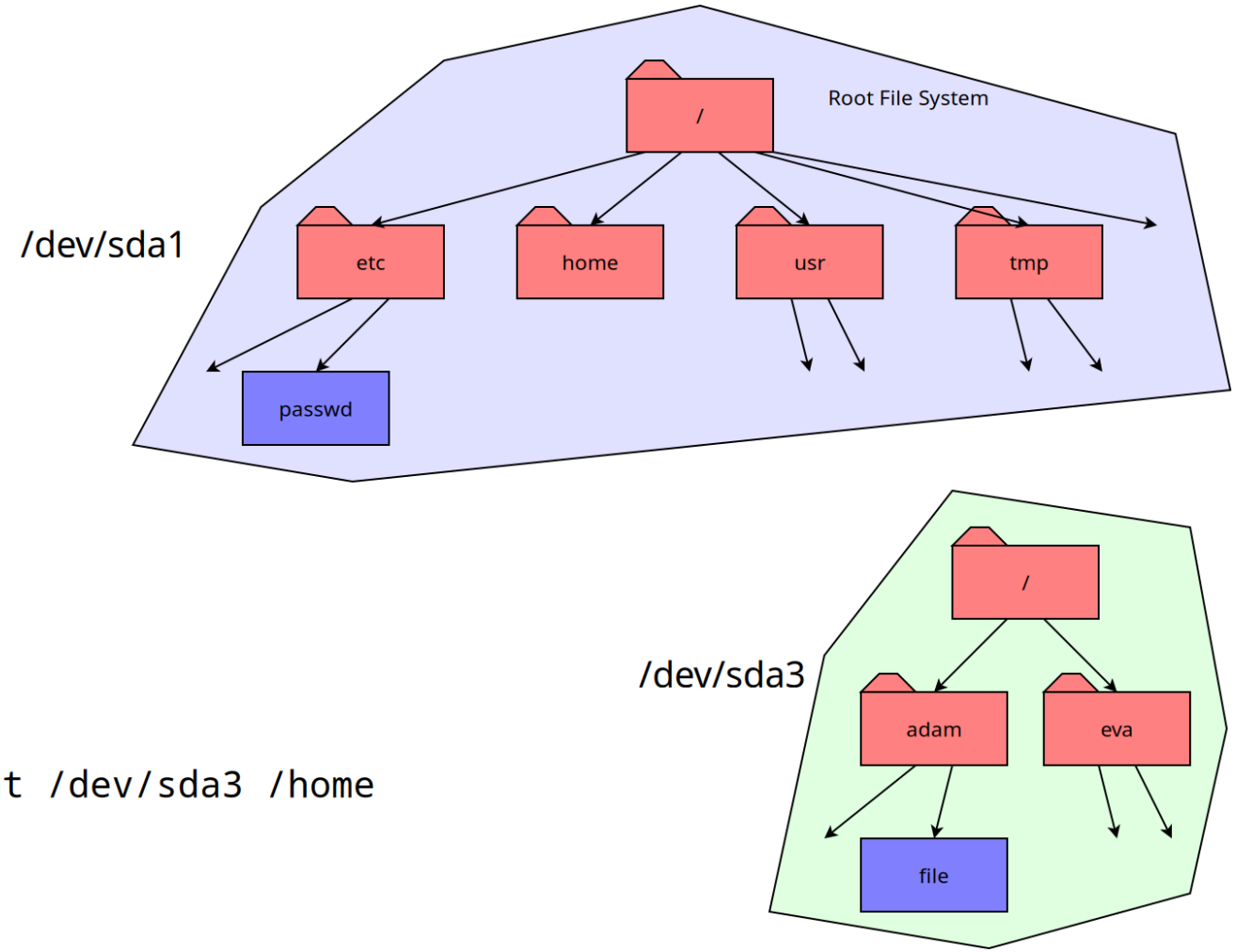
- Each disk can contain a file system as a whole
  - the disk then corresponds to a single partition
- However, each disk can be divided into multiple partitions
  - first block of the disk contains partition table
  - partition table contains information about
    - how many partitions exist
    - how large these partitions are
    - where they start
- Each partition
  - is represented by a special file; e. g.,
    - `/dev/sda`, `/dev/sdb` (whole disk)
    - `/dev/sda1`, `/dev/sda2`, `/dev/sdb1` (parts of the disk)
  - contains a single file system

Partitions (2)

- Trees for each partition can be combined to a single homogeneous file tree (boundaries not visible for the user!)
  - "mounting" of file trees
- A single specified file system is the *root file system*, whose root directory is the root directory for the whole system
  - Other file systems can be mounted to the existing file system with the `mount` operation and removed from it with the `umount` operation.
  - With the help of *network file system* (NFS) even directories of other computers can be mounted to the local file system.
    ⇒ *"hidden" boundaries* between file systems of different computers

Mounting to the File Tree

- Example




```
mount /dev/sda3 /home
```

Mounting to the File Tree (2)

- After execution of the mount operation