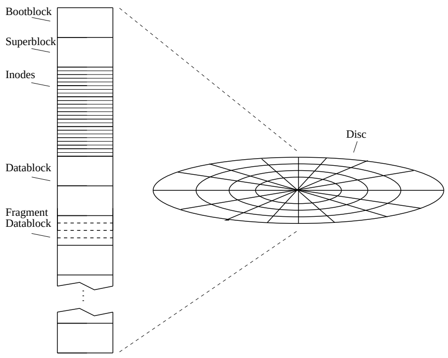
Labinstruction 3, UNIX file system

- UNIX file system
- Mounting
- Files & inodes
- Directories
- Labinstruction

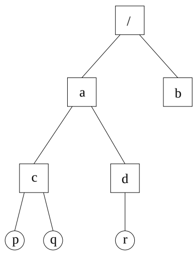
UNIX file system

- file system is created with `mkfs`
- `mkfs` defines a number of parameters for the system:
  - bootblock – contains a primary boot program.
  - superblock – static parameters of the file system, like total size, block and fragment sizes of data blocks.
  - inodes – (512 bytes)
  - data blocks – (4Kbyte, 8Kbyte)
  - fragment data block size – (512byte, 1024byte)
- number of inodes == maximum number of files.



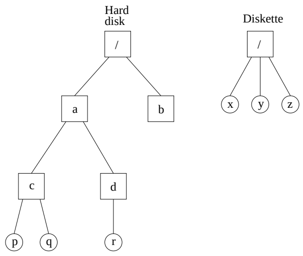
File system structure

- files have no structure at all - i.e. only flat sequence of bytes
- directories are files that contain information on how to find other files.
- directories arranged in familiar "tree" structure:



Mounting

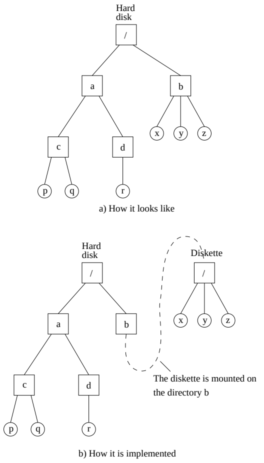
- Different disks (machines) might have different filesystems.
- We need a way of accessing files located on different disks.
- One solution is to do it in a DOS-like manner, where we give each disk a separate name, like `H:` (harddisk) and `D:` (diskette drive).



- For example, to copy the file `x` to the directory `d`, one would type: `cp D:/x H:/a/d/x`

Mounting, continued

- In UNIX a part (or all) of a disk's file system can be *mounted* in another disk's file system.



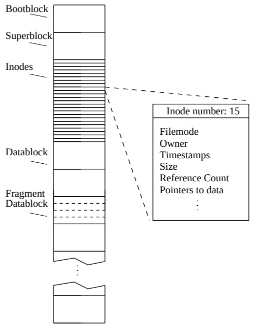
- The user sees a single file tree and no longer has to be aware of which file resides on which device.
- The root file system / is always available on a machine while other parts can be integrated (mounted) into the file system.
- See `df(1)` and `mount(1M)` for more information.

Files & inodes

Def: A file consists of exactly one *inode*, and zero or more *data blocks*.

An inode is a structure used to maintain information about the file. It includes fields for the following:

- file mode
- owner (user and group)
- timestamps (3 different)
- size (bytes, blocks)
- reference count
- pointers to data



Important: A file does *not* have a name. The file is uniquely identified by its inode number.

File mode

Normally referred to as permissions, but also contains information about the type of file.

Normally you see this, when you do `ls -l` as:

```
lrwxrwxrwx   1 user    group   file1
-rw-r--r--   1 ebbe    docs    file2
```

The first bits identify the file as one of:

- a regular file: -
- a directory: `d`
- a symbolic link: `l`
- etc – (see `stat(2)`, `stat(5)` and `mknod(2)`)

Permissions are the low-order 9 bits of the mode bytes, that defines who may do what with the file. The bits are normally presented like `rwx` where `r`, `w` and `x` stands for read, write and execute. For each file, this is defined for:

- the owner (the first 3 bits)
- the owner's group (the next 3 bits)
- everyone else (the last 3 bits)

`chmod(1)` and `chmod(2)` lets you change the permission mode of a file.

Timestamps

There are 3 timestamps defined:

- modification time - when file was last changed
- access time - when file was last read
- status time - when certain changes are made to inode

Pointers to data

Through pointers in the inode we can access the file's data blocks.

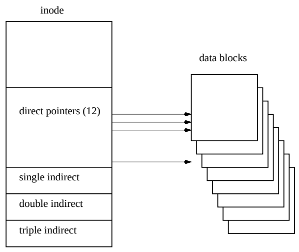
For reasons of disk space efficiency, there are 4 different types of pointers:

- direct (12)
- single indirect (1)
- double indirect (1)
- triple indirect (1)

Direct pointers

If the file consists of 12 or fewer data blocks, we can access them directly from the 12 direct pointers in the inode.

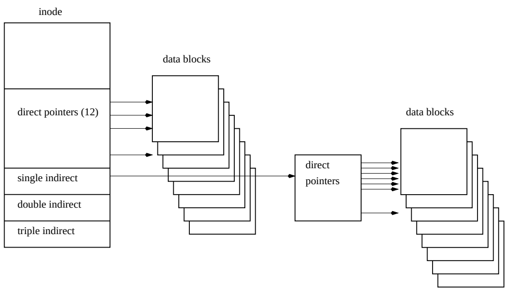
For block size 4 Kbyte or 8 Kbyte, this means files up to 48 Kbyte resp 96 Kbyte can be accessed entirely from the information in the inode.



Single indirect pointer

The single indirect pointer is necessary in order to create files of more than 12 data blocks.

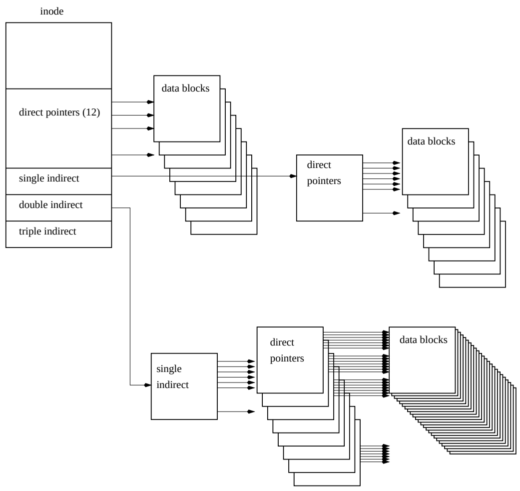
The single indirect pointer points to a single data block, whose contents are treated as direct pointers to data blocks. We can now use a couple of hundred data blocks (files of a few MB).



Double indirect pointer

The double indirect pointer is necessary in order to create files of more than a few MB.

The double indirect pointer points to a data block whose contents are treated as single indirect pointers. Each of these pointers points to a data block, whose contents are treated as direct pointers to data blocks. This is enough to reach the filesize limit on most systems.

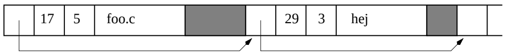


Directories

Directories are files, but we treat them differently. A directory can be identified by its mode bytes.

A directory is a file that consists of a number of records, each of which contains the following fields:

- a pointer to the next record
- a number identifying an *inode* (i.e. another file)
- a number identifying the *length* of the record
- a string containing the *name* of the record (max 255 chars). It is this name we usually refer to as a filename. Note that it is part of the directory, not part of the file.
- (possibly some padding)

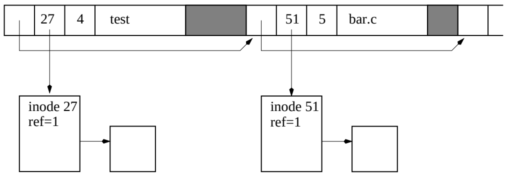


`ls` in this directory shows:

```
$ ls -li
17 -rw-r--r--  1    (...)    foo.c
29 -rwxr-xr-x  1    (...)    hej
```

Links

By associating a name in a directory with a file, we get what is known as a *link*. Don't confuse these links with *symbolic links*.

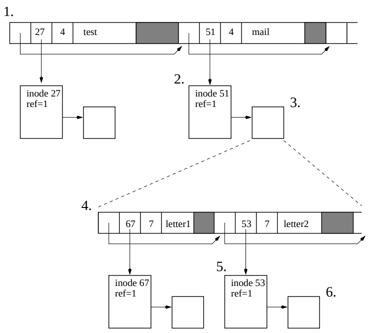


This directory contains links to inodes 27 and 51, so we can refer to file 27 as "test" and file 51 as "bar.c".

```
$ ls -li
51 -rw-r--r--  1    (...)    bar.c
27 -rw-r--r--  1    (...)    test
```

Example, find a file

The file `mail/letter2` shall be found.



1. Search the current directory to find the directory entry with the name `mail`.
2. When found, get the inode the directory entry is pointing at.
3. Identify the inode as a directory inode and get the directory data it is pointing at.
4. Search the directory data to find the directory entry with the name `letter2`.
5. When found, get the inode the directory entry is pointing at.
6. Finally, get the data that the inode is pointing at.

Creating a file

To create a regular file, we use `open(2)` and set the flags argument to `O_CREAT`.

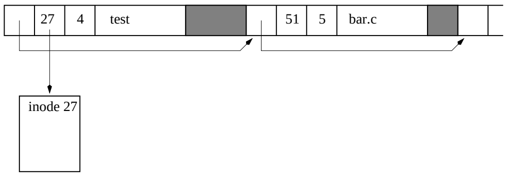
```
int fd;

fd=open("test", O_CREAT);
close(fd);
```

What happens?

- A free inode is found and initialized
- an entry is created in the current directory to point to the inode.

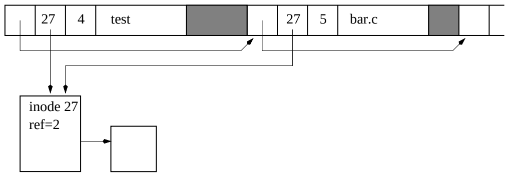
We now have a file with one link to it.



Note that the file is *empty*, i.e. there are initially no data blocks.

Hard links

We can have more than one link to a file. This kind of link is known as a *hard link*. Do not confuse it with a symbolic link.



In this case, the reference counter in inode 27 will be 2 because there are two links to the inode, and we can use either name ("test" or "bar.c") to refer to the same file.

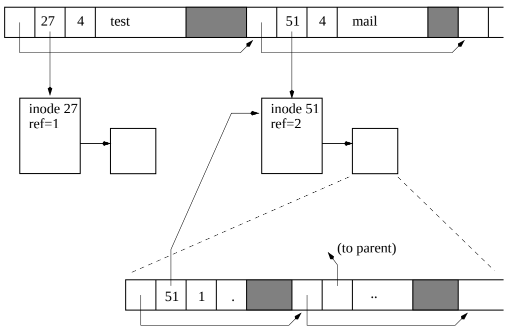
```
$ ls -li
27 -rw-r--r--  2    (...)    bar.c
27 -rw-r--r--  2    (...)    test
```

- There is absolutely *no* difference between the two.
- The links do not need to be in the same directory.
- To create a hard link use `ln(1)` or `link(2)`.

Directory links

When we use `mkdir(1)` or `mkdir(2)` to create a directory, the following happens:

- a file is created (i.e. an inode is allocated), and it is identified as a directory
- a link to the inode is created in the current directory.
- in the new directory, two entries are created:
  - "." points to the directory's own inode
  - ".." points to the parent's inode

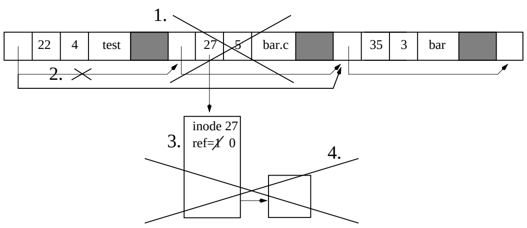


Removing files

To remove a file we use `rm(1)` or `unlink(2)`.

What happens is this:

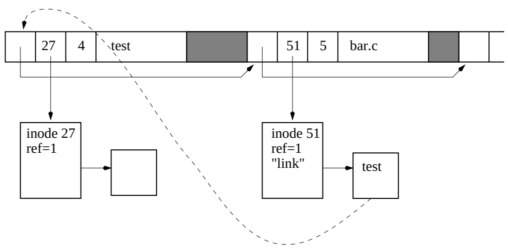
1. the directory entry is freed
2. the record pointer of the previous entry is reset
3. the file reference counter is decremented by one
4. if the reference counter now is at zero, the data blocks and inode are freed.



Symbolic links

Symbolic links are quite different from *hard links*.

A symbolic link is actually a separate file, whose contents is the *name* of another file or directory. A mode bit indicates that a file is to be interpreted as a symbolic link.




```
$ ls -li
51 lrwxrwxrwx  1    (...)    bar.c -> test
27 -rw-r--r--  1    (...)    test
```

Symbolic links are created with `ln(1)` or `symlink(2)`.

If we have a file that we know is a symbolic link, we can use `readlink(2)` to get the contents of the link, i.e. determine where it points.

Reading a directory

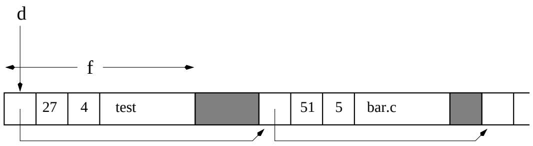
It is possible to open a directory directly like any other file, and read the data structures it contains.

This is not the recommended method, since the actual structures and the order they appear may vary among systems, or among disk within a single system. It is also inconvenient.

The easy and portable method is to use the 3 standard functions: `opendir(3)`, `readdir(3)` and `closedir(3)`. This is how they are used:

```
DIR *d;
struct dirent *f;

d=opendir(   );
while (f=readdir(d)) {
   (use f)
}
closedir(d);
```



Reading an inode

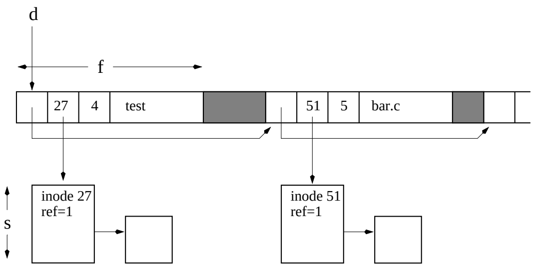
An inode can be read using the `stat(2)` system calls.

We call `stat(2)` with a pointer to a `struct stat`, and `stat` fills in the fields with the data we requested.

There is more than one `stat(2)` function. Use the right one! We can now extend the example:

```
struct stat s;

d=opendir(   );
while (f=readdir(d)) {
   (use f)
   stat(f->d_name, &s)
   (use s)
}
closedir(d);
```



Assignment

- **df** - Log in to the server machine Rama and use `df(1)` to get some statistics about the file system.
- **ls** - use `ls(1)` to display and determine information about files.
- **ln** - use `ln(1)` to create links and investigate the behaviour of links.
- **programming** - use `stat(2)` to report information from an inode. For example: mode (permisssions), number of links, owner's name and size in bytes. The information shall be printed nicely.
- **programming** - extend the first programming assignment by reading whole directories and traverse the filesystem in a recursively manner.
  First, report information about the current directory and secondly, do the same for its subdirectories.