1 UNIX file system

A file system is created with `mkfs`. It defines a number of parameters for the system, such as:
- bootblock - contains a primary boot program
- superblock - static parameters of the file system, like total size, block and fragment sizes of data blocks
- inodes - stands for index node
- data blocks - each block has typically a size of 4Kbyte or 8Kbyte
- fragment data block size - typically of size 512byte or 1024byte

The number of inodes determines the maximum number of files in the file system.

![Disk layout](image)

Figure 1: Disk layout

1.1 File system structure

Files have no structure at all, they are only flat sequences of bytes. Directories are files that contain information on how to find other files. Directories are arranged in a familiar tree structure. Different disks (machines) may have different filesystems and we need a way of accessing files located on different disks. One solution is to do it like Windows does it, where we give each disk a separate name, like C: or D:.

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.
1.2 Files and inodes

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 (three 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.

The file mode is normally referred to as permissions, but it also contains information about the type of file. Normally when you do a `ls -l`, you see something like this:

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

The first bits identify as one of:

- a regular file: -
- a directory: d
- a symbolic link: l

some other things, see the manual page for `stat`

Permissions are the low-order 9 bits of the mode bytes and they define who may do what with the file. The bits are normally presented like `rwx`, where r, w and x stand for read, write, and execute, respectively. 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)

\texttt{chmod} lets you change the permission mode of a file.

There are three timestamps defined:
• modification time - when the file was last changed
• access time - when the file was last read
• status time - when certain changes was made to the inode

Through pointers in the inode we can access the file’s data blocks. For reasons of disk space efficiency, there are four different types of pointers:
• direct (12)
• single indirect (1)
• double indirect (1)
• triple indirect (1)

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 sizes 4Kbyte or 8Kbyte, this means that files up to 48Kbyte or 96Kbyte, respectively, can be accessed entirely from the information in the inode.

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).

The double indirect pointer is necessary in order to create file 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.

1.3 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, and not part of the file.
(possibly some padding)

If you type ls -li in the directory in figure 6, you will see:

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

### 1.4 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.

The directory in figure 7 contains links to inodes 27 and 51, so we can refer to file 27 as "test" and file 51 as "bar.c".
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 figure 8, 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.

There is absolutely no difference between the two, and the links do not need to be in the same directory. To create a hard link you use `ln`.

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.

When you type `ls -li` in the directory in figure 9 you get:

```
51 lwxrwxrwx 1 (...) bar.c -> test
27 -rw-r--r-- 1 (...) test
```

Symbolic links are created with the shell command `ln` or the system call `symlink`. If we have a file that we know is a symbolic link, we can use the system call `readlink` to get the contents of the link, i.e. determine where it points.
1.5 Creating, using, and destroying stuff

1.5.1 Creating a file
To create a file we use the `open` system call with some arguments to tell it to create a file. When you do this a free inode is found and initialized and an entry is created in the current directory to point to the inode. Initially the file is *empty*, that is, there are no data blocks.

1.5.2 Creating a directory
To create a directory we can use either the shell command `mkdir` or the system call with the same name. When creating a directory the following happens: A file is created (i.e. an inode is allocated), and it is identified as a directory. Then a link to the inode is created in the current directory, and in the new directory two entries are created:

- `"."` points to the directory’s own inode
- `".."` points to the parent’s inode

Figure 10 shows the situation.

1.5.3 Removing files
To remove a file we use the shell command `rm` or the system call `unlink`. When you do this the following happens: First the directory entry is freed and the record pointer of the previous entry is reset. Then the file reference counter is decreased by one, and if it now is zero, the data blocks and the inode are freed.

1.5.4 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 in may vary among systems or among disks within a single system. It is also inconvenient. The easy and portable method is to use the three standard functions `opendir`, `readdir`, and `closedir`. This is how they are used in C:

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

d = opendir( );
while (f=readdir(d)) {
    (use f)
}
closedir(d);
```
1.5.5 Reading an inode

We can read an inode with one of the `stat` system calls. There is more than one `stat` function. Use the right one! We can now extend the C example:

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