Tutorial 13 - Filesystems

Gidon Rosalki

2025-06-25

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

This content will not appear in the exam this year!

1 File

A file is a logical unit of information, and not an address in the disk. It contains information like ADT (abstract data type), has a name, content (typically a sequence of bytes), metadata / attributes such as the creation date, size, and so on, and we can apply operations to it such as read, rename, and more. Furthermore, files are persistent, and can survive power outages, and outlive processes. This means that processes can use files, die, and then another process can use the same files.

1.1 Metadata

File metadata includes the following information:

- Size
- Owner
- Permissions, generally read, write, and execute
- Timestamps: This is the creation time, and the last time that the content metadata were modified / accessed. These are used by Makefiles, so if a file has been modified since `make` was last run, then `make` knows to recompile it. Otherwise, there is no reason so to do
- Location: This is the location on the disk (since it is a block device), and details where the file's blocks reside on the disk
- Type: Binary, text, whether or not it is a file or a directory

1.2 inode

The inode is a data structure in the Unix File System, that stores a file (or directory) metadata. It includes a hierarchical index of disk blocks, where the file is located:

- A few (around 12) **direct pointers**, which list the first blocks of the file (useful for small files)
- A single **indirect pointer**, which points to a block of additional direct pointers (useful for medium files)
- A **double indirect pointer**: Points to a block of indirect pointers (large files)
- **Triple indirect pointer**: Points to a block of double indirect pointers (huge files)

This results in the following inode structure:

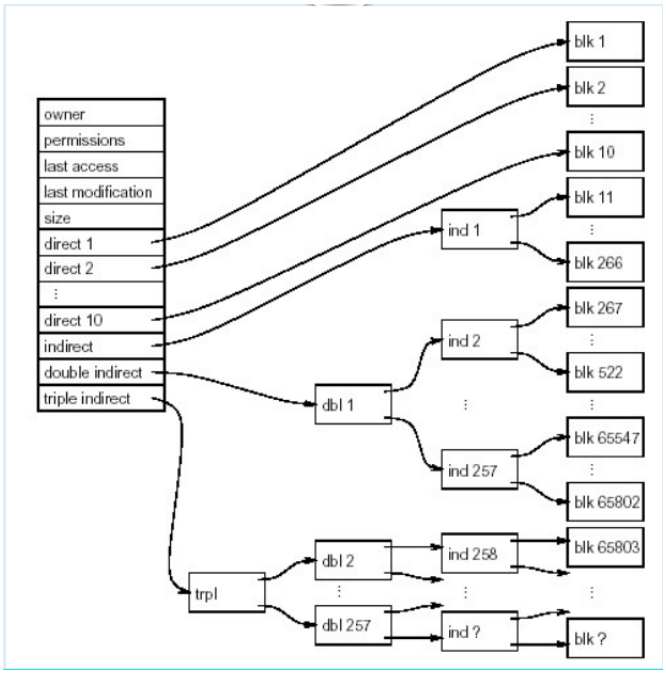

Figure 1: inode file structure

So here we can see that for all files, it will begin by filling the direct pointers, then the indirect, and so on down the hierarchy.

Assuming 32 bit inodes, where each pointer is 4 bytes, the blocks are 4096 bytes, then the 12 direct pointers provide access to up to 48KB. The indirect block contains 1024 bytes of additional pointers, resulting in a data size of 4MB, the double indirect block has 1024 pointers to indirect blocks, so it points to 4GB of data, and finally the triple indirect block allows files of up to 4TB.

1.3 Directories

Directories are stored like files. In the inode metadata, their type is assigned to be directory. The content (data blocks) will encode a mapping of the contained file / directory to its inode. This can be shown using the `-i` tag to the `ls` command.

An interesting quirk is that files do not store their names, only directories store the names of the files within them.

1.4 Superblocks

So, how do we allocate blocks for files and directories? Well, we use the superblock. This is located on the main memory, and it manages the allocation of blocks on the file system area. This block contains the size of the file system, a list of free blocks available on the FS, a list of unused inodes, and more. Using this information, it is possible to allocate disk blocks for saving file data or file metadata. It is presumably stored in a location that is known prior, since we need this information to be persistent.

An inode's space is allocated when creating a new FS, hence the number of inodes (number of ifles + folders) is limited. The superblock caches a short list of free inodes, and when a process needs a new inode, the kernel can use this list to allocate one. When an inode is freed, its location is written in the superblock, but only if there is room in the list. If the superblock list of free inodes is empty, then the kernel will search the disk, and add other free inodes to the superblock list.

When a process writes data to a file, the kernel must allocate disk blocks from the FS for a direct, or indirect block. When the kernel wants to allocate a block from the FS, it allocates the next available block in the superblock list.

1.5 Storing and accessing data

1.5.1 Opening a file

Storing data in a file involves the following operations:

- Allocation of disk blocks to the file
- Read (not always) and write operations
- Optimisation: Avoiding disk access by caching data in memory

When you open a file, a file descriptor (FD) is an abstract handle used to access a file or other input/output resources, such as a network socket. Threads share the FD table (and the file offset), and once an FD exists, it will always point to the same file. Remember, in Unix "everything" is a file. Successful calls to `open` return an FD, which is a nonnegative int, which is an index to a per process array called the "file descriptor table".

There are several predefined FDs:

File	FD	POSIX symbolic constant
Standard input	0	STDIN_FILENO
Standard output	1	STDOUT_FILENO
Standard Error	2	STDERR

Table 1:

When opening a file, (so we just called `fd = open("myfile", R)`, the FS reads the current directory, and finds the named file is represented internally by a specific inode number in the list of files maintained by the disk. The relevant entry is then read from the disk, and copied into the kernel's open file tables. The user's access rights are then checked, and the user's variable `fd` is made to point to the allocated entry in the open files table. `fd` serves as a handle ,telling to which file the user tries to access with subsequent read and write system calls, without allowing the user code to actually access kernel data.

We need the open file table to check permissions (once), and store the file offset (how far into the file we have read). All the operations on the file are performed through the file descriptor.

1.5.2 Reading from a file

Consider `read(fd, buf, 199)`. The argument `fd` identifies the open file, by pointing into the kernel's open files table. The system then gains access to the list of blocks that contain the file's data, and the file system reads the relevant disk block into its buffer cache. Finally, 100 bytes are copied into the users memory, at the address indicated by `buf`.

1.5.3 Writing to a file

Assume the following scenario: We want to write 100 bytes, starting with byte 2000 in the file, and each disk block is 1024 bytes. Therefore, the data we want to write spans the end of the second block through the beginning of the third block. When modifying a block, we need to read it in its entirety into the buffer cache, and then overwrite part of it with the new data.

The buffer cache is important to improve performance, since if we write to a buffer, we will probably want to write to it again soon. However, this can cause reliability issues, since it delays the writeback to the disk, and therefore data can be lost to sudden shutdowns, or perhaps USB removal.

1.5.4 Location in the file

The `read` system call is given a buffer address for placing the data in the user's memory, without an indication of the offset in the file from which the data should be taken. The OS maintains the current offset into the file, which is updated after each operation, and if random access is required, the process can set the file pointer to any desired value by using the `seek` system call.

1.5.5 OS file tables

- The file descriptor table: This is separate for each process. Each entry points to an entry in the open files table, and the index of this slot is the `fd` that was returned by `open`
- The open files table: An entry in this table is allocated every time a file is opened. Each entry contains a pointer to the inode table, and a position within the file. There can be multiple open file entries pointing to the same inode.
- The inode table: Each file may appear at most once in this table.