

1 Sample Driver Uses

These examples are derived from the MINIX OS. MINIX was the inspiration for the Linux system and as such, represents a sort of simplified Linux.

1.1 Initialization

During boot, the OS calls init routines for installed drivers. The drivers then read the physical device data from the hardware (or BIOS) and store it in a data structure in kernel memory. Then a jump table is initialized to provide linkage between the generic kernel APIs and the specific routines in the driver. This indirection allows for drivers to be loaded piecemeal without having to recompile the entire kernel.

1.2 Mounting, open

The device is accessed only at the first physical read to the device. Then the driver prepares for subsequent accesses (data structures needed only for reading and writing are created only when necessary). The minor device number is used to identify exactly which device (or partition) is being used. The driver then installs an interrupt handler to respond to the drive controller. The driver then queries the controller for all the specifics (drive geometry, volume label, etc.). The driver then sets the comb to a starting position.

1.3 Data Transfer

The driver first sets up data structures for the transfer, including source/destination and size of the transfer. The driver then waits for previously scheduled requests to finish. Then the driver gains control of the bus and sends the request to the actual device. At this point the driver waits to be activated by the interrupt that signals when the operation is done. When the interrupt is received, the driver updates status flags and signals the user process.

1.4 Unmount

Usually there's not much to do at the hardware level (apart from ejecting the CD or tape). Mostly unmounting is just cleaning up the data structures associated with the now unmounted drive.

2 File Systems

What is a filesystem? A filesystem is essentially that part of the operating system that translates nice, convenient file names into a collection of blocks stored on disk (and vice versa).

2.1 Filesystem Design Goals

A filesystem is an engineered system, and as a consequence has multiple design goals to fulfill. Namely:

1. **Persistence:** Data should survive the system powering down
2. **Quantity:** Lots of data should be storable. Much more than in addressable memory (e.g. much greater than 2^{32} bytes).
3. **Sharing:** The filesystem should allow controlled sharing of data, and conversely reliable control of access to data.

3 Files

So, you all know what a file is. It's basically a collection of bytes, or at least that's what it is on UNIX and Windows machines. A sequence of bytes is a simple organization and is flexible insofar as you can embed any more sophisticated structure in it. However, other systems have more structured files. On old IBM machines, files are actually collections of records, and records are collections of fields. On that system files resemble tables in a database, and the filesystem was more DB-like. With structured files, one of the fields in every record is labeled as the 'key', which is used (just like the primary key in modern DBs) for searching.

3.1 File Contents

So, if a file's just a collection of raw bits, how do you interpret them? How do you know that the file in question is an image, rather than a sound clip or your tax return? One answer is context. When you're using Photoshop, chances are that you're manipulating images rather than tax returns. Another, more flexible answer is to add file type information to the file's metadata. On Linux, this is the so-called 'magic number', which is just an integer identifying a particular file as a text file rather than an executable. On DOS, all file information was embedded in the file name (the three letter extension), so executables ended in `.EXE` and GIF files in `.GIF`. Of course, this means that any user can mess up file mappings with a careless rename.

At the coarsest granularity, you want to be able to distinguish between ASCII/text and binary files. Even after that, you still need to figure out exactly what kind of binary files you're dealing with. At the very least you have to be able to distinguish between executable files (basically compiled code) and just raw data files. Even though most OSes have a preferred executable format, many modern OSes have to support multiple executable formats. For instance, Linux made a switch from the old `a.out` format to the newer, fancier `ELF` format. And during that transition, Linux systems basically had to be able to support both, even though they each required different linkage support. MacOS recently switched processors from PowerPC to Intel x86. Even though both binaries are technically `mach-o`, the system needs to distinguish between the two because under the hood MacOSX runs a binary translation layer (called Rosetta) that translates, at runtime, from PowerPC to x86! This is basically recompilation on-the-fly!

3.2 Other Metadata

Apart from magic numbers carrying file type information, the filesystem and OS need other metadata. What other kind of metadata? How about:

1. File Size
2. Permissions
3. Ownership (user and group on UNIX)
4. Dates (creation, modification, etc.)
5. And much more...

So, you have all this metadata. Where do you store it? You could store it in the **file header**, but that means that every user program on the system would have to remember that the first N bytes of data in the file are actually filesystem metadata, which is annoying. Additionally, the user code would have to play nice with the filesystem meaning that it would have to ensure that it didn't 'accidentally' mess around with vital meta-data (like file size, or permissions). From a security and reliability standpoint this is bad. So file headers are out, perhaps you could store the information in the **directory**? That avoids some of the issues with the file header, but it bloats each file's entry in the directory. For a directory with just a few files that's no problem. But consider a directory with hundreds or thousands of files. The directory entry itself could be bloated by megabytes at that point. And that would mean that it would take a lot longer to read in a directory, which means that all lookups would be slowed down (perhaps considerably). Ok, so headers are bad, and directories are inefficient. Where else can we store this vital metadata? The answer is: somewhere else! The filesystem will simply allocate a block to store file information. On UNIX systems this additional meta-data block is called an **inode**. The inode contains all the meta-data you need to keep yourself organized. In addition to the fields listed above, inodes also contain pointers to the actual data blocks of the file, the number of blocks used by the file, and the number of links to the file (more on links later). The inode also contains **flags** which specify deeper file semantics. For example flags can specify whether or not a file is read-only, append-only, a system file, an archive, a compressed file, a temporary file, locked by a user, etc. Flags allow the filesystem to support richer semantics than would be possible with a magic number and UNIX-style permissions. For systems that support structured files (with records and fields), the metadata would also contain information about record and field length, key length and key position.

4 Directories

Up til now we've just discussed directories indirectly (no pun intended). We assumed that there was some kind of directory thing out there but we never discussed it in detail. Well, not all filesystems support directories. Particularly filesystems catering to systems with small amounts of persistent

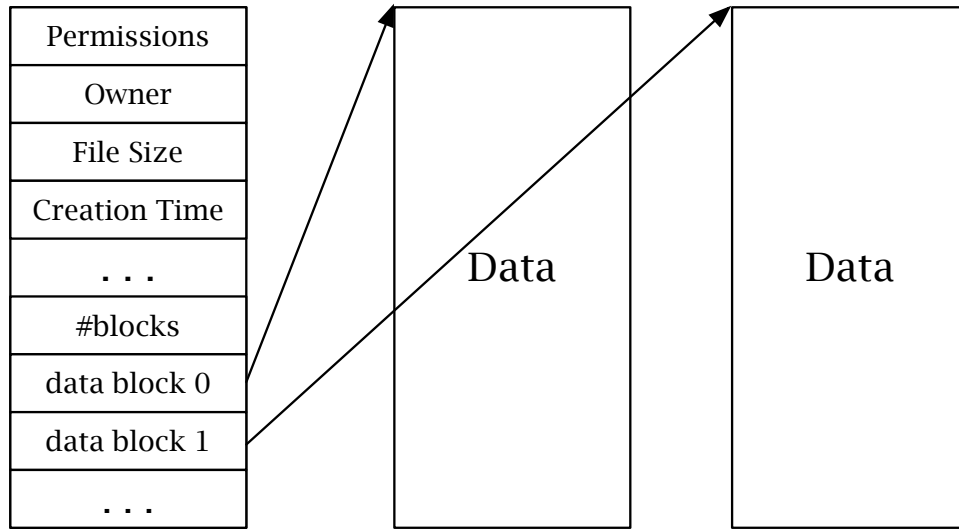


Figure 1: A Unix Inode, more or less

storage (where every byte counts). In such circumstances using an entire block just to hold organizational information is wasteful. Of course, PCs have had disk space coming out of their ears for years now, so most filesystems support directories. A directory is basically an organizational tool that the user can employ to better structure their data. Users abstractly place files in directories and directories in directories to organize their files hierarchically. This leads to a tree-like representation of file organization. However, UNIX systems support **links**. A link is a file that ‘points’ to another file. A symbolic link, is just a file whose contents are the path to the file it’s referencing. A hard link contains a reference to the actual inode on disk for the file it’s referencing. As such, hard links can only be created to a file on the same volume. However, hard links are faster, because the shell doesn’t have to re-traverse the directory tree to locate the linked-to file. With links, the tree is converted into a directed graph (which looks mostly like a tree, just with an occasional freak edge that points off somewhere else). More recently, people have started playing around with the notion of namespaces or views (this all started with Plan 9 in the 90s). In this case, the user organizes a ‘view’ of their data which may or may not bear any relation to the actual directory structure on disk.

On most systems, a directory is essentially just another file. It just happens to only contain a list of references to other files. On UNIX, a directory has an inode, so directories can enjoy all the same permission and ownership issues of normal files. With directories, however, the directory flag has been set, so that a program knows that the data blocks referenced by the inode are, in fact, lists of directory entries. These entries will contain the name of the file, as well as a reference to the inode for that file.

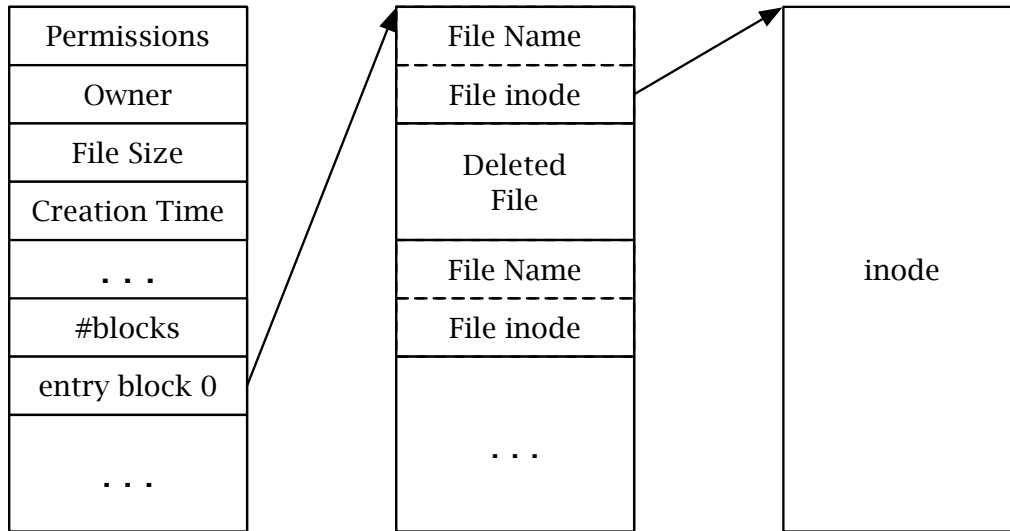


Figure 2: A Directory Inode

4.1 Mounting

Mounting is the name of the process of introducing a new volume (basically a partition or a device) into your filesystem. On windows, each volume is its own separate tree, rooted at some semi-arbitrary single-letter label (like `C :`, for instance). UNIX, is far more elegant in this regard. There is only one tree, and it's rooted at `' / '`. So mounting on UNIX systems is the process of rooting a particular volume at some directory in the global file tree. Some standard places have emerged over the years, `/mnt` was used to mount removable devices in the old days, most modern Linux systems place mounted drives in the `/media` directory. Mac OSX has a volume manager daemon running that likes to put everything under `/VOLUME` and then remount important drives (like the root drive) at the appropriate place. All of the startup mount points are specified in `/etc/fstab`.

4.2 Naming

Directories not only organize the actual files, but they allow the user to more easily organize the file namespace. For instance, `/home/trekp/foo` and `/usr/src/foo` are different files, even though they're both named `foo`. They are different to the filesystem because they're located in different directories, and therefore have different **absolute paths**. The path to a file is considered part of the file's name. That's why you have to specify the `PATH` environment variable for running executables, your shell has to be able to construct the full path name of a binary before it can ask the OS to run it. The `'/'` characters on UNIX systems allow for the filesystem to distinguish directories from one another and more efficiently navigate to a specific file.

5 Filesystem Implementation

Now that we have a rough idea of what a filesystem has to do, now we can address how best to go about it. Filesystem implementation is all about performance and scalability. You want to design everything so that it's fast for small files and humongous files, small directories and gigantic directories, and work on many different size devices.

5.1 File Creation

Obviously, a filesystem must support file creation. To create a file requires three basic steps:

1. Find free space and claim it
2. Write the file data into that claimed space
3. Update file metadata and directory metadata

For step one, filesystems need to have efficient ways of managing free space. They must also be able to layout files in such a way as writing to them is also efficient. And meta-data must be conveniently located so the continual updates to the meta-data don't slow the system down.

5.2 Directories

Filesystems must also support efficient directories. Care must be taken when laying out directories and their files so that a simple `ls` doesn't take forever. On UNIX, a directory is just like any other file, except that the data blocks hold a sequence of variable length records detailing the files contained within the directory. Each entry contains the inode # of the file, the length of the record, the length of the file name, a file type, and the file name itself. The **root** directory is always stored at inode 2 on the volume. So, from any volume you can retrieve the root directory, and from that directory you can follow any (legal) path, directory by directory, until you find the file specified. That's how paths work!

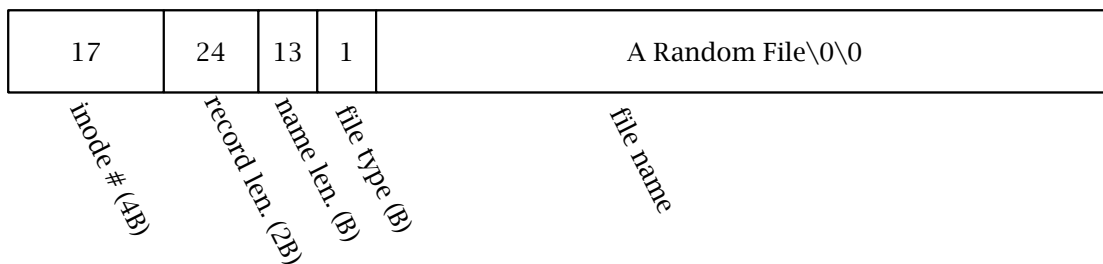


Figure 3: A Sample UNIX Directory Entry

As you may imagine, constantly going out to the disk to look up directory information can be time-consuming. And if you went back to the disk every time a user asked for a file, then it would be unimaginably slow. The reason things aren't that slow is that any OS worth its salt caches the directory inodes and entry blocks in main memory. This means that subsequent lookups will actually just be memory operations (which are 100K to 1M times faster than disk). Still, you do want your directory layout on the disk itself to be as fast as possible. A complex solution (employed by `hfs+` actually) is to treat directories as interior nodes on a B-tree, and layout the tree on disk efficiently (a problem the DB people have spent a lot of time on). A simpler solution is to store the directories on the inner tracks so that hopping from one directory to another is way cheaper than one file to another. This makes sense as every new file lookup may involve many directory queries, but only one regular file query.

5.3 Free Space Management

A filesystem needs to keep track of free space on disk in an accurate and efficient manner. You need to be able to deallocate and allocate space rapidly.

5.3.1 Free List

One solution is to maintain a linked list of all the free blocks on disk. A simple route would be to have a link (the block number, basically) to the next free block at the beginning of a free block. This way you can just wander from one free block to the next allocating a block at a time until you've allocated enough space.

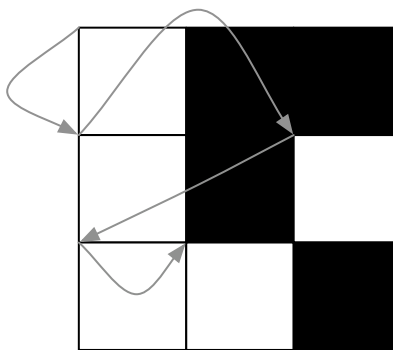


Figure 4: Free List, one link per block

A more efficient route would be to have a list of blocks that contain links to free blocks. That way you would only have to load in a single link block to get access to many blocks worth of free space. Each link block will have a link to the next link block (if more is needed), but this will result in much shorter chains than the simple free list described above. Of course, the link blocks can only be used as free space when all their links have been used up, so there is some space overhead for the increase in efficiency.

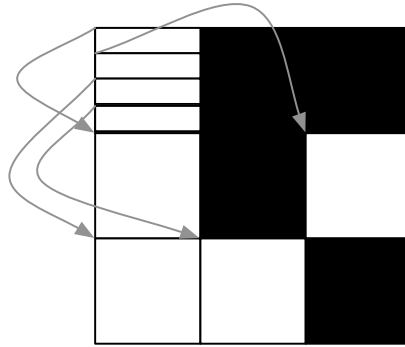


Figure 5: Free List, link blocks

In both cases, caching the first few free blocks (or link blocks) in memory greatly speeds up allocation.

5.3.2 Allocation Bitmap

Another approach is to have a bitmap that reflects the allocation state of the block group. The idea is simple, a bitmap is an array of bits. One bit for each block. If the block is free, the bit is 0; if the block is being used, the bit is 1. For example, the bit map for the block groups pictured above is: 011010001. In terms of meta-data per block, this is about as efficient as you can get. To allocate space, you walk the string of bits flipping as you go until you've grabbed enough blocks. (Bitmaps are what the ext2 filesystem in Linux uses, by the way). Bitmaps can be space-hogs though. Consider a 100 GB disk with 512 B blocks. With one bit per block that comes to 25MB needed for storing the bitmap. This is ok on disk (25MB is .025% of 100GB), but bad in main memory.

5.4 File Management

Once you've figured out how to manage free space, you can start worrying about how to allocate files within that free space. There are, of course, numerous options. Each with their own tradeoffs.

5.4.1 Continuous Allocation

The first strategy would be to just allocate blocks one after another, so that a file existed as a contiguous chunk of disk space. Of course, this is a problem if the file needs to grow over time, as there may not be any free blocks after the last one in the file. Additionally, contiguous allocation causes fragmentation. As blocks are deallocated and smaller files are allocated over those free blocks, leftover space begins to accumulate. This can lead to a situation where the disk has enough free space to accommodate a new file, but the blocks aren't contiguous so you can't create the file. The figure below illustrates the situation where subsequent file creation leads to fragmentation.

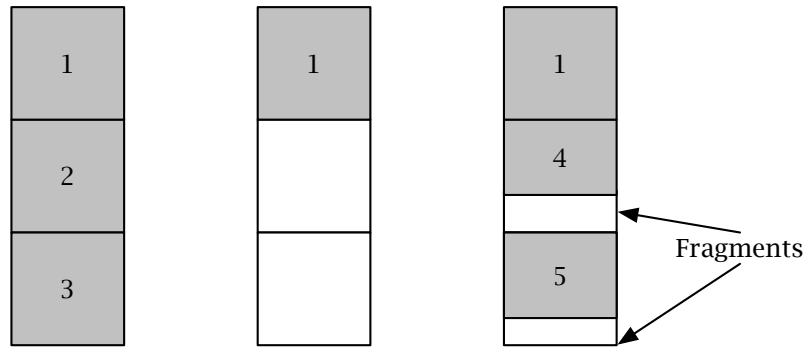


Figure 6: Fragmentation

This, by the way, is what defrag-ing your disk is all about. When you defrag, you collapse scattered file contents into a contiguous chunk and just pile up all the free blocks at the end into one contiguous free region. Of course, defragging takes a long time and doesn't guarantee that you won't have more fragmentation in the future. In general it would just be better to avoid fragmentation in the first place.

5.4.2 Linked Block List

Another strategy is to have a link to the next block contained within the current block. This alleviates both the growth and fragmentation problem, but it means that random access to parts of the file is difficult. For instance, if you wanted to go straight to the middle of the file (block 5, say), you'd have to start at block 0, then go to blocks 1,2,3, and 4 before you'd know which disk block contained block 5 of the file.

5.4.3 File Allocation Table

Another strategy is a File Allocation Table (or FAT), which is a data structure that collapses the linked list of blocks into a table. Each entry in the table contains the block number of the next entry for a file. This speeds up random access because all the pointer traversing happens in memory (no disk reads necessary until you locate the block you want). However, FATs can grow to be huge. Unlike bitmaps, each entry in a FAT needs to accommodate a block number (usually 4 bytes) so a FAT for a mostly full 100GB hard drive could occupy 800MB of space! It seems unreasonable to waste most of your RAM just maintaining file information.

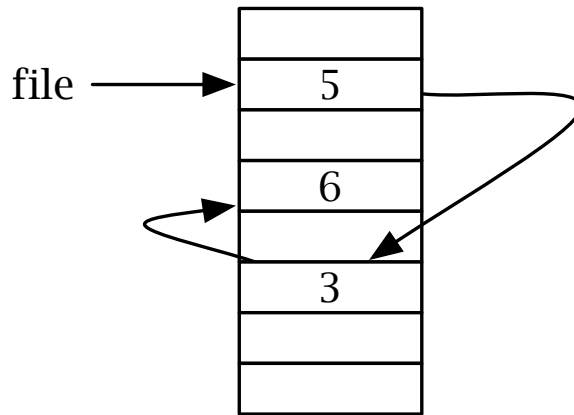


Figure 7: A File Allocation Table

5.4.4 Index Node

The last strategy we'll discuss is the index node (inode). An index node is a block that holds pointers to all the blocks used by a file. On UNIX systems, all the file metadata is contained in the inode as well, so there's no space overhead for the metadata. Of course, because an inode must fit into a single block, this puts an upper limit on file size. Consider a system with 512B blocks. If each block entry is 4B large, and each inode consists solely of block pointers, then a file can be no larger than $(512B/4B) * 512B = 65536B = 64K$. So, modern UNIX systems use a hierarchical inode structure, where the index contains pointers to data blocks and blocks of pointers (the so-called indirect blocks). On Linux, for example, the first 12 pointers point directly to data blocks. This works just fine for small files. If you need more space, then pointer 13 points to a block that contains references to more data blocks (the indirect block). If you need even more space, then pointer 14 points to a block that contains pointers to indirect blocks (the doubly-indirect block). If you need even MORE space, then pointer 15 points to a block that contains pointers to doubly indirect blocks (the triply-indirect block). This compromise means that small files (files that fit into 12 or fewer blocks) use only one block for indexing, but large files can be accommodated as well. How large? Well, using our previous example of 512B block size and 4B per block pointer, we have $12 * 512B = 6144B$ for the direct blocks. $(512/4) * 512B = 64K$ for the indirect block. $(512/4) * 64K = 8MB$ for the doubly-indirect block. And last, $(512/4) * 8MB = 1GB$ for the triply-indirect block. Of course, real systems have larger block sizes and can accommodate considerably larger files.

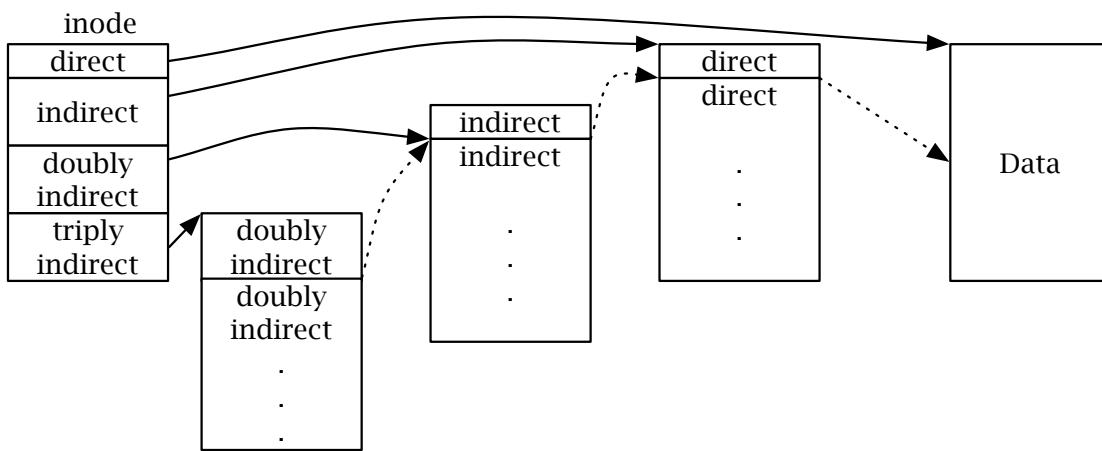


Figure 8: A Hierarchical Inode