

Interlude: Files and Directories

Thus far we have seen the development of two key operating system abstractions: the process, which is a virtualization of the CPU, and the address space, which is a virtualization of memory. In tandem, these two abstractions allow a program to run as if it is in its own private, isolated world; as if it has its own processor (or processors); as if it has its own memory. This illusion makes programming the system much easier and thus is prevalent today not only on desktops and servers but increasingly on all programmable platforms including mobile phones and the like.

In this section, we add one more critical piece to the virtualization puzzle: **persistent storage**. A persistent-storage device, such as a classic **hard disk drive** or a more modern **solid-state storage device**, stores information permanently (or at least, for a long time). Unlike memory, whose contents are lost when there is a power loss, a persistent-storage device keeps such data intact. Thus, the OS must take extra care with such a device: this is where users keep data that they really care about.

CRUX: HOW TO MANAGE A PERSISTENT DEVICE

How should the OS manage a persistent device? What are the APIs? What are the important aspects of the implementation?

Thus, in the next few chapters, we will explore critical techniques for managing persistent data, focusing on methods to improve performance and reliability. We begin, however, with an overview of the API: the interfaces you'll expect to see when interacting with a UNIX file system.

39.1 Files and Directories

Two key abstractions have developed over time in the virtualization of storage. The first is the **file**. A file is simply a linear array of bytes, each of which you can read or write. Each file has some kind of **low-level**

name, usually a number of some kind; often, the user is not aware of this name (as we will see). For historical reasons, the low-level name of a file is often referred to as its **inode number**. We'll be learning a lot more about inodes in future chapters; for now, just assume that each file has an inode number associated with it.

In most systems, the OS does not know much about the structure of the file (e.g., whether it is a picture, or a text file, or C code); rather, the responsibility of the file system is simply to store such data persistently on disk and make sure that when you request the data again, you get what you put there in the first place. Doing so is not as simple as it seems!

The second abstraction is that of a **directory**. A directory, like a file, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs. For example, let's say there is a file with the low-level name "10", and it is referred to by the user-readable name of "foo". The directory that "foo" resides in thus would have an entry ("foo", "10") that maps the user-readable name to the low-level name. Each entry in a directory refers to either files or other directories. By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.

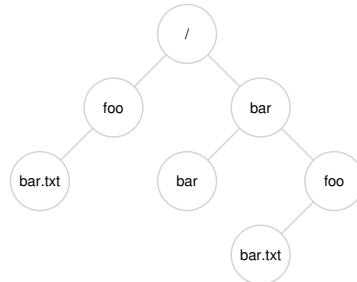


Figure 39.1: An Example Directory Tree

The directory hierarchy starts at a **root directory** (in UNIX-based systems, the root directory is simply referred to as `/`) and uses some kind of **separator** to name subsequent **sub-directories** until the desired file or directory is named. For example, if a user created a directory `foo` in the root directory `/`, and then created a file `bar.txt` in the directory `foo`, we could refer to the file by its **absolute pathname**, which in this case would be `/foo/bar.txt`. See Figure 39.1 for a more complex directory tree; valid directories in the example are `/`, `/foo`, `/bar`, `/bar/bar`, `/bar/foo` and valid files are `/foo/bar.txt` and `/bar/foo/bar.txt`. Directories and files can have the same name as long as they are in different locations in the file-system tree (e.g., there are two files named `bar.txt` in the figure, `/foo/bar.txt` and `/bar/foo/bar.txt`).

TIP: THINK CAREFULLY ABOUT NAMING

Naming is an important aspect of computer systems [SK09]. In UNIX systems, virtually everything that you can think of is named through the file system. Beyond just files, devices, pipes, and even processes [K84] can be found in what looks like a plain old file system. This uniformity of naming eases your conceptual model of the system, and makes the system simpler and more modular. Thus, whenever creating a system or interface, think carefully about what names you are using.

You may also notice that the file name in this example often has two parts: `bar` and `txt`, separated by a period. The first part is an arbitrary name, whereas the second part of the file name is usually used to indicate the **type** of the file, e.g., whether it is C code (e.g., `.c`), or an image (e.g., `.jpg`), or a music file (e.g., `.mp3`). However, this is usually just a **convention**: there is usually no enforcement that the data contained in a file named `main.c` is indeed C source code.

Thus, we can see one great thing provided by the file system: a convenient way to **name** all the files we are interested in. Names are important in systems as the first step to accessing any resource is being able to name it. In UNIX systems, the file system thus provides a unified way to access files on disk, USB stick, CD-ROM, many other devices, and in fact many other things, all located under the single directory tree.

39.2 The File System Interface

Let's now discuss the file system interface in more detail. We'll start with the basics of creating, accessing, and deleting files. You may think this is straightforward, but along the way we'll discover the mysterious call that is used to remove files, known as `unlink()`. Hopefully, by the end of this chapter, this mystery won't be so mysterious to you!

39.3 Creating Files

We'll start with the most basic of operations: creating a file. This can be accomplished with the `open` system call; by calling `open()` and passing it the `O_CREAT` flag, a program can create a new file. Here is some example code to create a file called "foo" in the current working directory.

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

The routine `open()` takes a number of different flags. In this example, the second parameter creates the file (`O_CREAT`) if it does not exist, ensures that the file can only be written to (`O_WRONLY`), and, if the file already exists, truncates it to a size of zero bytes thus removing any existing content (`O_TRUNC`). The third parameter specifies permissions, in this case making the file readable and writable by the owner.

ASIDE: THE `creat()` SYSTEM CALL

The older way of creating a file is to call `creat()`, as follows:

```
int fd = creat("foo"); // option: add second flag to set permissions
```

You can think of `creat()` as `open()` with the following flags: `O_CREAT` | `O_WRONLY` | `O_TRUNC`. Because `open()` can create a file, the usage of `creat()` has somewhat fallen out of favor (indeed, it could just be implemented as a library call to `open()`); however, it does hold a special place in UNIX lore. Specifically, when Ken Thompson was asked what he would do differently if he were redesigning UNIX, he replied: “I’d spell `creat` with an e.”

One important aspect of `open()` is what it returns: a **file descriptor**. A file descriptor is just an integer, private per process, and is used in UNIX systems to access files; thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. In this way, a file descriptor is a **capability** [L84], i.e., an opaque handle that gives you the power to perform certain operations. Another way to think of a file descriptor is as a pointer to an object of type `file`; once you have such an object, you can call other “methods” to access the file, like `read()` and `write()`. We’ll see just how a file descriptor is used below.

39.4 Reading and Writing Files

Once we have some files, of course we might like to read or write them. Let’s start by reading an existing file. If we were typing at a command line, we might just use the program `cat` to dump the contents of the file to the screen.

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

In this code snippet, we redirect the output of the program `echo` to the file `foo`, which then contains the word “hello” in it. We then use `cat` to see the contents of the file. But how does the `cat` program access the file `foo`?

To find this out, we’ll use an incredibly useful tool to trace the system calls made by a program. On Linux, the tool is called **strace**; other systems have similar tools (see **dtruss** on Mac OS X, or **truss** on some older UNIX variants). What `strace` does is trace every system call made by a program while it runs, and dump the trace to the screen for you to see.

TIP: USE `strace` (AND SIMILAR TOOLS)

The `strace` tool provides an awesome way to see what programs are up to. By running it, you can trace which system calls a program makes, see the arguments and return codes, and generally get a very good idea of what is going on.

The tool also takes some arguments which can be quite useful. For example, `-f` follows any fork'd children too; `-t` reports the time of day at each call; `-e trace=open,close,read,write` only traces calls to those system calls and ignores all others. There are many more powerful flags — read the man pages and find out how to harness this wonderful tool.

Here is an example of using `strace` to figure out what `cat` is doing (some calls removed for readability):

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)      = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

The first thing that `cat` does is open the file for reading. A couple of things we should note about this; first, that the file is only opened for reading (not writing), as indicated by the `O_RDONLY` flag; second, that the 64-bit offset be used (`O_LARGEFILE`); third, that the call to `open()` succeeds and returns a file descriptor, which has the value of 3.

Why does the first call to `open()` return 3, not 0 or perhaps 1 as you might expect? As it turns out, each running process already has three files open, standard input (which the process can read to receive input), standard output (which the process can write to in order to dump information to the screen), and standard error (which the process can write error messages to). These are represented by file descriptors 0, 1, and 2, respectively. Thus, when you first open another file (as `cat` does above), it will almost certainly be file descriptor 3.

After the `open` succeeds, `cat` uses the `read()` system call to repeatedly read some bytes from a file. The first argument to `read()` is the file descriptor, thus telling the file system which file to read; a process can of course have multiple files open at once, and thus the descriptor enables the operating system to know which file a particular read refers to. The second argument points to a buffer where the result of the `read()` will be placed; in the system-call trace above, `strace` shows the results of the read in this spot ("hello"). The third argument is the size of the buffer, which

in this case is 4 KB. The call to `read()` returns successfully as well, here returning the number of bytes it read (6, which includes 5 for the letters in the word “hello” and one for an end-of-line marker).

At this point, you see another interesting result of the `strace`: a single call to the `write()` system call, to the file descriptor 1. As we mentioned above, this descriptor is known as the standard output, and thus is used to write the word “hello” to the screen as the program `cat` is meant to do. But does it call `write()` directly? Maybe (if it is highly optimized). But if not, what `cat` might do is call the library routine `printf()`; internally, `printf()` figures out all the formatting details passed to it, and eventually calls `write` on the standard output to print the results to the screen.

The `cat` program then tries to read more from the file, but since there are no bytes left in the file, the `read()` returns 0 and the program knows that this means it has read the entire file. Thus, the program calls `close()` to indicate that it is done with the file “foo”, passing in the corresponding file descriptor. The file is thus closed, and the reading of it thus complete.

Writing a file is accomplished via a similar set of steps. First, a file is opened for writing, then the `write()` system call is called, perhaps repeatedly for larger files, and then `close()`. Use `strace` to trace writes to a file, perhaps of a program you wrote yourself, or by tracing the `dd` utility, e.g., `dd if=foo of=bar`.

39.5 Reading And Writing, But Not Sequentially

Thus far, we’ve discussed how to read and write files, but all access has been **sequential**; that is, we have either read a file from the beginning to the end, or written a file out from beginning to end.

Sometimes, however, it is useful to be able to read or write to a specific offset within a file; for example, if you build an index over a text document, and use it to look up a specific word, you may end up reading from some **random** offsets within the document. To do so, we will use the `lseek()` system call. Here is the function prototype:

```
off_t lseek(int fd, off_t offset, int whence);
```

The first argument is familiar (a file descriptor). The second argument is the `offset`, which positions the **file offset** to a particular location within the file. The third argument, called `whence` for historical reasons, determines exactly how the seek is performed. From the man page:

```
If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
```

As you can tell from this description, for each file a process opens, the OS tracks a “current” offset, which determines where the next read or

ASIDE: CALLING `lseek()` DOES NOT PERFORM A DISK SEEK

The poorly-named system call `lseek()` confuses many a student trying to understand disks and how the file systems atop them work. Do not confuse the two! The `lseek()` call simply changes a variable in OS memory that tracks, for a particular process, at which offset to which its next read or write will start. A disk seek occurs when a read or write issued to the disk is not on the same track as the last read or write, and thus necessitates a head movement. Making this even more confusing is the fact that calling `lseek()` to read or write from/to random parts of a file, and then reading/writing to those random parts, will indeed lead to more disk seeks. Thus, calling `lseek()` can certainly lead to a seek in an upcoming read or write, but absolutely does not cause any disk I/O to occur itself.

write will begin reading from or writing to within the file. Thus, part of the abstraction of an open file is that it has a current offset, which is updated in one of two ways. The first is when a read or write of N bytes takes place, N is added to the current offset; thus each read or write *implicitly* updates the offset. The second is *explicitly* with `lseek`, which changes the offset as specified above.

Note that this call `lseek()` has nothing to do with the **seek** operation of a disk, which moves the disk arm. The call to `lseek()` simply changes the value of a variable within the kernel; when the I/O is performed, depending on where the disk head is, the disk may or may not perform an actual seek to fulfill the request.

39.6 Writing Immediately with `fsync()`

Most times when a program calls `write()`, it is just telling the file system: please write this data to persistent storage, at some point in the future. The file system, for performance reasons, will **buffer** such writes in memory for some time (say 5 seconds, or 30); at that later point in time, the write(s) will actually be issued to the storage device. From the perspective of the calling application, writes seem to complete quickly, and only in rare cases (e.g., the machine crashes after the `write()` call but before the write to disk) will data be lost.

However, some applications require something more than this eventual guarantee. For example, in a database management system (DBMS), development of a correct recovery protocol requires the ability to force writes to disk from time to time.

To support these types of applications, most file systems provide some additional control APIs. In the UNIX world, the interface provided to applications is known as `fsync(int fd)`. When a process calls `fsync()` for a particular file descriptor, the file system responds by forcing all **dirty** (i.e., not yet written) data to disk, for the file referred to by the specified

file descriptor. The `fsync()` routine returns once all of these writes are complete.

Here is a simple example of how to use `fsync()`. The code opens the file `foo`, writes a single chunk of data to it, and then calls `fsync()` to ensure the writes are forced immediately to disk. Once the `fsync()` returns, the application can safely move on, knowing that the data has been persisted (if `fsync()` is correctly implemented, that is).

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
assert(fd > -1);
int rc = write(fd, buffer, size);
assert(rc == size);
rc = fsync(fd);
assert(rc == 0);
```

Interestingly, this sequence does not guarantee everything that you might expect; in some cases, you also need to `fsync()` the directory that contains the file `foo`. Adding this step ensures not only that the file itself is on disk, but that the file, if newly created, also is durably a part of the directory. Not surprisingly, this type of detail is often overlooked, leading to many application-level bugs [P+13].

39.7 Renaming Files

Once we have a file, it is sometimes useful to be able to give a file a different name. When typing at the command line, this is accomplished with `mv` command; in this example, the file `foo` is renamed `bar`:

```
prompt> mv foo bar
```

Using `strace`, we can see that `mv` uses the system call `rename(char *old, char *new)`, which takes precisely two arguments: the original name of the file (`old`) and the new name (`new`).

One interesting guarantee provided by the `rename()` call is that it is (usually) implemented as an **atomic** call with respect to system crashes; if the system crashes during the renaming, the file will either be named the old name or the new name, and no odd in-between state can arise. Thus, `rename()` is critical for supporting certain kinds of applications that require an atomic update to file state.

Let's be a little more specific here. Imagine that you are using a file editor (e.g., `emacs`), and you insert a line into the middle of a file. The file's name, for the example, is `foo.txt`. The way the editor might update the file to guarantee that the new file has the original contents plus the line inserted is as follows (ignoring error-checking for simplicity):

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```


What the editor does in this example is simple: write out the new version of the file under a temporary name (`foo.txt.tmp`), force it to disk with `fsync()`, and then, when the application is certain the new file metadata and contents are on the disk, rename the temporary file to the original file's name. This last step atomically swaps the new file into place, while concurrently deleting the old version of the file, and thus an atomic file update is achieved.

39.8 Getting Information About Files

Beyond file access, we expect the file system to keep a fair amount of information about each file it is storing. We generally call such data about files **metadata**. To see the metadata for a certain file, we can use the `stat()` or `fstat()` system calls. These calls take a pathname (or file descriptor) to a file and fill in a `stat` structure as seen here:

```
struct stat {
    dev_t      st_dev;      /* ID of device containing file */
    ino_t      st_ino;      /* inode number */
    mode_t     st_mode;     /* protection */
    nlink_t    st_nlink;    /* number of hard links */
    uid_t      st_uid;      /* user ID of owner */
    gid_t      st_gid;      /* group ID of owner */
    dev_t      st_rdev;     /* device ID (if special file) */
    off_t      st_size;     /* total size, in bytes */
    blksize_t  st_blksize;  /* blocksize for filesystem I/O */
    blkcnt_t   st_blocks;   /* number of blocks allocated */
    time_t     st_atime;    /* time of last access */
    time_t     st_mtime;    /* time of last modification */
    time_t     st_ctime;    /* time of last status change */
};
```

You can see that there is a lot of information kept about each file, including its size (in bytes), its low-level name (i.e., inode number), some ownership information, and some information about when the file was accessed or modified, among other things. To see this information, you can use the command line tool `stat`:

```
prompt> echo hello > file
prompt> stat file
File: 'file'
Size: 6 Blocks: 8          IO Block: 4096   regular file
Device: 811h/2065d Inode: 67158084    Links: 1
Access: (0640/-rw-r-----)  Uid: (30686/  remzi)  Gid: (30686/  remzi)
Access: 2011-05-03 15:50:20.157594748 -0500
Modify: 2011-05-03 15:50:20.157594748 -0500
Change: 2011-05-03 15:50:20.157594748 -0500
```

As it turns out, each file system usually keeps this type of information in a structure called an **inode**¹. We'll be learning a lot more about inodes when we talk about file system implementation. For now, you should just think of an inode as a persistent data structure kept by the file system that has information like we see above inside of it.

39.9 Removing Files

At this point, we know how to create files and access them, either sequentially or not. But how do you delete files? If you've used UNIX, you probably think you know: just run the program `rm`. But what system call does `rm` use to remove a file?

Let's use our old friend `strace` again to find out. Here we remove that pesky file "foo":

```
prompt> strace rm foo
...
unlink("foo")                                = 0
...
```

We've removed a bunch of unrelated cruft from the traced output, leaving just a single call to the mysteriously-named system call `unlink()`. As you can see, `unlink()` just takes the name of the file to be removed, and returns zero upon success. But this leads us to a great puzzle: why is this system call named "unlink"? Why not just "remove" or "delete". To understand the answer to this puzzle, we must first understand more than just files, but also directories.

39.10 Making Directories

Beyond files, a set of directory-related system calls enable you to make, read, and delete directories. Note you can never write to a directory directly; because the format of the directory is considered file system meta-data, you can only update a directory indirectly by, for example, creating files, directories, or other object types within it. In this way, the file system makes sure that the contents of the directory always are as expected.

To create a directory, a single system call, `mkdir()`, is available. The eponymous `mkdir` program can be used to create such a directory. Let's take a look at what happens when we run the `mkdir` program to make a simple directory called `foo`:

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                            = 0
...
prompt>
```

¹Some file systems call these structures similar, but slightly different, names, such as `dnodes`; the basic idea is similar however.

TIP: BE WARY OF POWERFUL COMMANDS

The program `rm` provides us with a great example of powerful commands, and how sometimes too much power can be a bad thing. For example, to remove a bunch of files at once, you can type something like:

```
prompt> rm *
```

where the `*` will match all files in the current directory. But sometimes you want to also delete the directories too, and in fact all of their contents. You can do this by telling `rm` to recursively descend into each directory, and remove its contents too:

```
prompt> rm -rf *
```

Where you get into trouble with this small string of characters is when you issue the command, accidentally, from the root directory of a file system, thus removing every file and directory from it. Oops!

Thus, remember the double-edged sword of powerful commands; while they give you the ability to do a lot of work with a small number of keystrokes, they also can quickly and readily do a great deal of harm.

When such a directory is created, it is considered “empty”, although it does have a bare minimum of contents. Specifically, an empty directory has two entries: one entry that refers to itself, and one entry that refers to its parent. The former is referred to as the “.” (dot) directory, and the latter as “..” (dot-dot). You can see these directories by passing a flag (`-a`) to the program `ls`:

```
prompt> ls -a
./ ../
prompt> ls -al
total 8
drwxr-x---  2 remzi remzi   6 Apr 30 16:17 ./
drwxr-x--- 26 remzi remzi 4096 Apr 30 16:17 ../
```

39.11 Reading Directories

Now that we’ve created a directory, we might wish to read one too. Indeed, that is exactly what the program `ls` does. Let’s write our own little tool like `ls` and see how it is done.

Instead of just opening a directory as if it were a file, we instead use a new set of calls. Below is an example program that prints the contents of a directory. The program uses three calls, `opendir()`, `readdir()`, and `closedir()`, to get the job done, and you can see how simple the interface is; we just use a simple loop to read one directory entry at a time, and print out the name and inode number of each file in the directory.

```

int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%d %s\n", (int) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}

```

The declaration below shows the information available within each directory entry in the `struct dirent` data structure:

```

struct dirent {
    char      d_name[256]; /* filename */
    ino_t     d_ino;       /* inode number */
    off_t     d_off;       /* offset to the next dirent */
    unsigned short d_reclen; /* length of this record */
    unsigned char d_type;   /* type of file */
};

```

Because directories are light on information (basically, just mapping the name to the inode number, along with a few other details), a program may want to call `stat()` on each file to get more information on each, such as its length or other detailed information. Indeed, this is exactly what `ls` does when you pass it the `-l` flag; try `strace` on `ls` with and without that flag to see for yourself.

39.12 Deleting Directories

Finally, you can delete a directory with a call to `rmdir()` (which is used by the program of the same name, `rmdir`). Unlike file deletion, however, removing directories is more dangerous, as you could potentially delete a large amount of data with a single command. Thus, `rmdir()` has the requirement that the directory be empty (i.e., only has `."` and `.."` entries) before it is deleted. If you try to delete a non-empty directory, the call to `rmdir()` simply will fail.

39.13 Hard Links

We now come back to the mystery of why removing a file is performed via `unlink()`, by understanding a new way to make an entry in the file system tree, through a system call known as `link()`. The `link()` system call takes two arguments, an old pathname and a new one; when you “link” a new file name to an old one, you essentially create another way to refer to the same file. The command-line program `ln` is used to do this, as we see in this example:

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

Here we created a file with the word “hello” in it, and called the file `file`². We then create a hard link to that file using the `ln` program. After this, we can examine the file by either opening `file` or `file2`.

The way `link` works is that it simply creates another name in the directory you are creating the link to, and refers it to the *same* inode number (i.e., low-level name) of the original file. The file is not copied in any way; rather, you now just have two human names (`file` and `file2`) that both refer to the same file. We can even see this in the directory itself, by printing out the inode number of each file:

```
prompt> ls -li file file2
67158084 file
67158084 file2
prompt>
```

By passing the `-li` flag to `ls`, it prints out the inode number of each file (as well as the file name). And thus you can see what `link` really has done: just make a new reference to the same exact inode number (67158084 in this example).

By now you might be starting to see why `unlink()` is called `unlink()`. When you create a file, you are really doing *two* things. First, you are making a structure (the inode) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth. Second, you are *linking* a human-readable name to that file, and putting that link into a directory.

After creating a hard link to a file, to the file system, there is no difference between the original file name (`file`) and the newly created file name (`file2`); indeed, they are both just links to the underlying meta-data about the file, which is found in inode number 67158084.

Thus, to remove a file from the file system, we call `unlink()`. In the example above, we could for example remove the file named `file`, and still access the file without difficulty:

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

The reason this works is because when the file system unlinks `file`, it checks a **reference count** within the inode number. This reference count

²Note how creative the authors of this book are. We also used to have a cat named “Cat” (true story). However, she died, and we now have a hamster named “Hammy.” Update: Hammy is now dead too. The pet bodies are piling up.

(sometimes called the **link count**) allows the file system to track how many different file names have been linked to this particular inode. When `unlink()` is called, it removes the “link” between the human-readable name (the file that is being deleted) to the given inode number, and decrements the reference count; only when the reference count reaches zero does the file system also free the inode and related data blocks, and thus truly “delete” the file.

You can see the reference count of a file using `stat()` of course. Let’s see what it is when we create and delete hard links to a file. In this example, we’ll create three links to the same file, and then delete them. Watch the link count!

```
prompt> echo hello > file
prompt> stat file
... Inode: 67158084      Links: 1 ...
prompt> ln file file2
prompt> stat file
... Inode: 67158084      Links: 2 ...
prompt> stat file2
... Inode: 67158084      Links: 2 ...
prompt> ln file2 file3
prompt> stat file
... Inode: 67158084      Links: 3 ...
prompt> rm file
prompt> stat file2
... Inode: 67158084      Links: 2 ...
prompt> rm file2
prompt> stat file3
... Inode: 67158084      Links: 1 ...
prompt> rm file3
```

39.14 Symbolic Links

There is one other type of link that is really useful, and it is called a **symbolic link** or sometimes a **soft link**. As it turns out, hard links are somewhat limited: you can’t create one to a directory (for fear that you will create a cycle in the directory tree); you can’t hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc. Thus, a new type of link called the symbolic link was created.

To create such a link, you can use the same program `ln`, but with the `-s` flag. Here is an example:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

As you can see, creating a soft link looks much the same, and the original file can now be accessed through the file name `file` as well as the symbolic link name `file2`.

However, beyond this surface similarity, symbolic links are actually quite different from hard links. The first difference is that a symbolic link is actually a file itself, of a different type. We've already talked about regular files and directories; symbolic links are a third type the file system knows about. A `stat` on the symlink reveals all:

```
prompt> stat file
... regular file ...
prompt> stat file2
... symbolic link ...
```

Running `ls` also reveals this fact. If you look closely at the first character of the long-form of the output from `ls`, you can see that the first character in the left-most column is a `-` for regular files, a `d` for directories, and an `l` for soft links. You can also see the size of the symbolic link (4 bytes in this case), as well as what the link points to (the file named `file`).

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r----- 1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

The reason that `file2` is 4 bytes is because the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file. Because we've linked to a file named `file`, our link file `file2` is small (4 bytes). If we link to a longer pathname, our link file would be bigger:

```
prompt> echo hello > alongerfilename
prompt> ln -s alongerfilename file3
prompt> ls -al alongerfilename file3
-rw-r----- 1 remzi remzi    6 May  3 19:17 alongerfilename
lrwxrwxrwx  1 remzi remzi   15 May  3 19:17 file3 -> alongerfilename
```

Finally, because of the way symbolic links are created, they leave the possibility for what is known as a **dangling reference**:

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

As you can see in this example, quite unlike hard links, removing the original file named `file` causes the link to point to a pathname that no longer exists.

39.15 Making and Mounting a File System

We've now toured the basic interfaces to access files, directories, and certain types of special types of links. But there is one more topic we should discuss: how to assemble a full directory tree from many underlying file systems. This task is accomplished via first making file systems, and then mounting them to make their contents accessible.

To make a file system, most file systems provide a tool, usually referred to as `mkfs` (pronounced "make fs"), that performs exactly this task. The idea is as follows: give the tool, as input, a device (such as a disk partition, e.g., `/dev/sda1`) a file system type (e.g., `ext3`), and it simply writes an empty file system, starting with a root directory, onto that disk partition. And `mkfs` said, let there be a file system!

However, once such a file system is created, it needs to be made accessible within the uniform file-system tree. This task is achieved via the `mount` program (which makes the underlying system call `mount()` to do the real work). What `mount` does, quite simply is take an existing directory as a target **mount point** and essentially paste a new file system onto the directory tree at that point.

An example here might be useful. Imagine we have an unmounted `ext3` file system, stored in device partition `/dev/sda1`, that has the following contents: a root directory which contains two sub-directories, `a` and `b`, each of which in turn holds a single file named `foo`. Let's say we wish to mount this file system at the mount point `/home/users`. We would type something like this:

```
prompt> mount -t ext3 /dev/sda1 /home/users
```

If successful, the mount would thus make this new file system available. However, note how the new file system is now accessed. To look at the contents of the root directory, we would use `ls` like this:

```
prompt> ls /home/users/  
a b
```

As you can see, the pathname `/home/users/` now refers to the root of the newly-mounted directory. Similarly, we could access files `a` and `b` with the pathnames `/home/users/a` and `/home/users/b`. Finally, the files named `foo` could be accessed via `/home/users/a/foo` and `/home/users/b/foo`. And thus the beauty of `mount`: instead of having a number of separate file systems, `mount` unifies all file systems into one tree, making naming uniform and convenient.

To see what is mounted on your system, and at which points, simply run the `mount` program. You'll see something like this:

```
/dev/sda1 on / type ext3 (rw)  
proc on /proc type proc (rw)  
sysfs on /sys type sysfs (rw)  
/dev/sda5 on /tmp type ext3 (rw)  
/dev/sda7 on /var/vice/cache type ext3 (rw)  
tmpfs on /dev/shm type tmpfs (rw)  
AFS on /afs type afs (rw)
```


This crazy mix shows that a whole number of different file systems, including ext3 (a standard disk-based file system), the proc file system (a file system for accessing information about current processes), tmpfs (a file system just for temporary files), and AFS (a distributed file system) are all glued together onto this one machine's file-system tree.

39.16 Summary

The file system interface in UNIX systems (and indeed, in any system) is seemingly quite rudimentary, but there is a lot to understand if you wish to master it. Nothing is better, of course, than simply using it (a lot). So please do so! Of course, read more; as always, Stevens [SR05] is the place to begin.

We've toured the basic interfaces, and hopefully understood a little bit about how they work. Even more interesting is how to implement a file system that meets the needs of the API, a topic we will delve into in great detail next.

References

[K84] “Processes as Files”

Tom J. Killian
USENIX, June 1984

The paper that introduced the /proc file system, where each process can be treated as a file within a pseudo file system. A clever idea that you can still see in modern UNIX systems.

[L84] “Capability-Based Computer Systems”

Henry M. Levy
Digital Press, 1984
Available: <http://homes.cs.washington.edu/~levy/capabook>
An excellent overview of early capability-based systems.

[P+13] “Towards Efficient, Portable Application-Level Consistency”

Thanumalayan S. Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
HotDep ’13, November 2013

Our own work that shows how readily applications can make mistakes in committing data to disk; in particular, assumptions about the file system creep into applications and thus make the applications work correctly only if they are running on a specific file system.

[SK09] “Principles of Computer System Design”

Jerome H. Saltzer and M. Frans Kaashoek
Morgan-Kaufmann, 2009
This tour de force of systems is a must-read for anybody interested in the field. It’s how they teach systems at MIT. Read it once, and then read it a few more times to let it all soak in.

[SR05] “Advanced Programming in the UNIX Environment”

W. Richard Stevens and Stephen A. Rago
Addison-Wesley, 2005
We have probably referenced this book a few hundred thousand times. It is that useful to you, if you care to become an awesome systems programmer.

Homework

In this homework, we'll just familiarize ourselves with how the APIs described in the chapter work. To do so, you'll just write a few different programs, mostly based on various UNIX utilities.

Questions

1. **Stat:** Write your own version of the command line program `stat`, which simply calls the `stat()` system call on a given file or directory. Print out file size, number of blocks allocated, reference (link) count, and so forth. What is the link count of a directory, as the number of entries in the directory changes? Useful interfaces: `stat()`.
2. **List Files:** Write a program that lists files in the given directory. When called without any arguments, the program should just print the file names. When invoked with the `-l` flag, the program should print out information about each file, such as the owner, group, permissions, and other information obtained from the `stat()` system call. The program should take one additional argument, which is the directory to read, e.g., `mys -l directory`. If no directory is given, the program should just use the current working directory. Useful interfaces: `stat()`, `opendir()`, `readdir()`, `getcwd()`.
3. **Tail:** Write a program that prints out the last few lines of a file. The program should be efficient, in that it seeks to near the end of the file, reads in a block of data, and then goes backwards until it finds the requested number of lines; at this point, it should print out those lines from beginning to the end of the file. To invoke the program, one should type: `mytail -n file`, where `n` is the number of lines at the end of the file to print. Useful interfaces: `stat()`, `lseek()`, `open()`, `read()`, `close()`.
4. **Recursive Search:** Write a program that prints out the names of each file and directory in the file system tree, starting at a given point in the tree. For example, when run without arguments, the program should start with the current working directory and print its contents, as well as the contents of any sub-directories, etc., until the entire tree, root at the CWD, is printed. If given a single argument (of a directory name), use that as the root of the tree instead. Refine your recursive search with more fun options, similar to the powerful `find` command line tool. Useful interfaces: you figure it out.

File System Implementation

In this chapter, we introduce a simple file system implementation, known as **vsfs** (the **Very Simple File System**). This file system is a simplified version of a typical UNIX file system and thus serves to introduce some of the basic on-disk structures, access methods, and various policies that you will find in many file systems today.

The file system is pure software; unlike our development of CPU and memory virtualization, we will not be adding hardware features to make some aspect of the file system work better (though we will want to pay attention to device characteristics to make sure the file system works well). Because of the great flexibility we have in building a file system, many different ones have been built, literally from AFS (the Andrew File System) [H+88] to ZFS (Sun's Zettabyte File System) [B07]. All of these file systems have different data structures and do some things better or worse than their peers. Thus, the way we will be learning about file systems is through case studies: first, a simple file system (vsfs) in this chapter to introduce most concepts, and then a series of studies of real file systems to understand how they can differ in practice.

THE CRUX: HOW TO IMPLEMENT A SIMPLE FILE SYSTEM

How can we build a simple file system? What structures are needed on the disk? What do they need to track? How are they accessed?

40.1 The Way To Think

To think about file systems, we usually suggest thinking about two different aspects of them; if you understand both of these aspects, you probably understand how the file system basically works.

The first is the **data structures** of the file system. In other words, what types of on-disk structures are utilized by the file system to organize its data and metadata? The first file systems we'll see (including vsfs below) employ simple structures, like arrays of blocks or other objects, whereas

ASIDE: MENTAL MODELS OF FILE SYSTEMS

As we’ve discussed before, mental models are what you are really trying to develop when learning about systems. For file systems, your mental model should eventually include answers to questions like: what on-disk structures store the file system’s data and metadata? What happens when a process opens a file? Which on-disk structures are accessed during a read or write? By working on and improving your mental model, you develop an abstract understanding of what is going on, instead of just trying to understand the specifics of some file-system code (though that is also useful, of course!).

more sophisticated file systems, like SGI’s XFS, use more complicated tree-based structures [S+96].

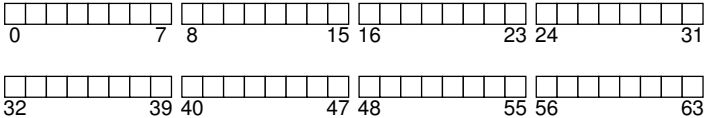
The second aspect of a file system is its **access methods**. How does it map the calls made by a process, such as `open()`, `read()`, `write()`, etc., onto its structures? Which structures are read during the execution of a particular system call? Which are written? How efficiently are all of these steps performed?

If you understand the data structures and access methods of a file system, you have developed a good mental model of how it truly works, a key part of the systems mindset. Try to work on developing your mental model as we delve into our first implementation.

40.2 Overall Organization

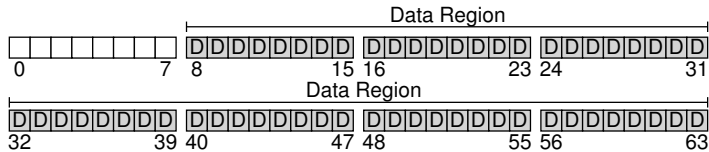
We now develop the overall on-disk organization of the data structures of the vsfs file system. The first thing we’ll need to do is divide the disk into **blocks**; simple file systems use just one block size, and that’s exactly what we’ll do here. Let’s choose a commonly-used size of 4 KB.

Thus, our view of the disk partition where we’re building our file system is simple: a series of blocks, each of size 4 KB. The blocks are addressed from 0 to $N - 1$, in a partition of size N 4-KB blocks. Assume we have a really small disk, with just 64 blocks:



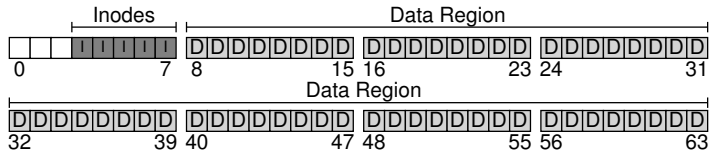
Let’s now think about what we need to store in these blocks to build a file system. Of course, the first thing that comes to mind is user data. In fact, most of the space in any file system is (and should be) user data. Let’s call the region of the disk we use for user data the **data region**, and,

again for simplicity, reserve a fixed portion of the disk for these blocks, say the last 56 of 64 blocks on the disk:



As we learned about (a little) last chapter, the file system has to track information about each file. This information is a key piece of **metadata**, and tracks things like which data blocks (in the data region) comprise a file, the size of the file, its owner and access rights, access and modify times, and other similar kinds of information. To store this information, file systems usually have a structure called an **inode** (we'll read more about inodes below).

To accommodate inodes, we'll need to reserve some space on the disk for them as well. Let's call this portion of the disk the **inode table**, which simply holds an array of on-disk inodes. Thus, our on-disk image now looks like this picture, assuming that we use 5 of our 64 blocks for inodes (denoted by I's in the diagram):

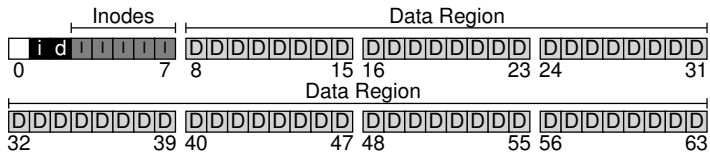


We should note here that inodes are typically not that big, for example 128 or 256 bytes. Assuming 256 bytes per inode, a 4-KB block can hold 16 inodes, and our file system above contains 80 total inodes. In our simple file system, built on a tiny 64-block partition, this number represents the maximum number of files we can have in our file system; however, do note that the same file system, built on a larger disk, could simply allocate a larger inode table and thus accommodate more files.

Our file system thus far has data blocks (D), and inodes (I), but a few things are still missing. One primary component that is still needed, as you might have guessed, is some way to track whether inodes or data blocks are free or allocated. Such **allocation structures** are thus a requisite element in any file system.

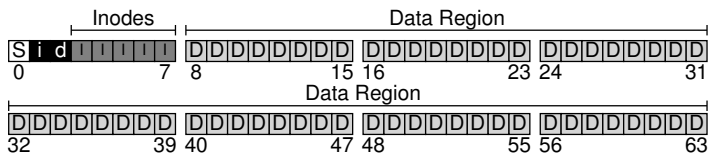
Many allocation-tracking methods are possible, of course. For example, we could use a **free list** that points to the first free block, which then points to the next free block, and so forth. We instead choose a simple and popular structure known as a **bitmap**, one for the data region (the **data bitmap**), and one for the inode table (the **inode bitmap**). A bitmap is a

simple structure: each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1). And thus our new on-disk layout, with an inode bitmap (i) and a data bitmap (d):



You may notice that it is a bit of overkill to use an entire 4-KB block for these bitmaps; such a bitmap can track whether 32K objects are allocated, and yet we only have 80 inodes and 56 data blocks. However, we just use an entire 4-KB block for each of these bitmaps for simplicity.

The careful reader (i.e., the reader who is still awake) may have noticed there is one block left in the design of the on-disk structure of our very simple file system. We reserve this for the **superblock**, denoted by an S in the diagram below. The superblock contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system (80 and 56, respectively in this instance), where the inode table begins (block 3), and so forth. It will likely also include a magic number of some kind to identify the file system type (in this case, vsfs).



Thus, when mounting a file system, the operating system will read the superblock first, to initialize various parameters, and then attach the volume to the file-system tree. When files within the volume are accessed, the system will thus know exactly where to look for the needed on-disk structures.

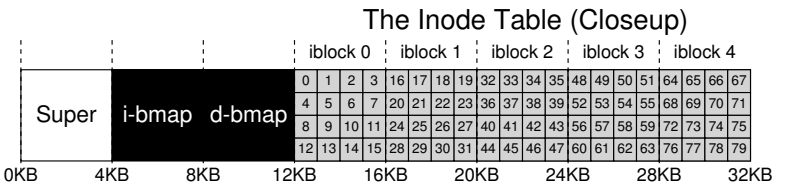
40.3 File Organization: The Inode

One of the most important on-disk structures of a file system is the **inode**; virtually all file systems have a structure similar to this. The name inode is short for **index node**, the historical name given to it in UNIX [RT74] and possibly earlier systems, used because these nodes were originally arranged in an array, and the array *indexed* into when accessing a particular inode.

ASIDE: DATA STRUCTURE — THE INODE

The **inode** is the generic name that is used in many file systems to describe the structure that holds the metadata for a given file, such as its length, permissions, and the location of its constituent blocks. The name goes back at least as far as UNIX (and probably further back to Multics if not earlier systems); it is short for **index node**, as the inode number is used to index into an array of on-disk inodes in order to find the inode of that number. As we'll see, design of the inode is one key part of file system design. Most modern systems have some kind of structure like this for every file they track, but perhaps call them different things (such as dnodes, fnodes, etc.).

Each inode is implicitly referred to by a number (called the **inumber**), which we've earlier called the **low-level name** of the file. In vsfs (and other simple file systems), given an i-number, you should directly be able to calculate where on the disk the corresponding inode is located. For example, take the inode table of vsfs as above: 20-KB in size (5 4-KB blocks) and thus consisting of 80 inodes (assuming each inode is 256 bytes); further assume that the inode region starts at 12KB (i.e. the superblock starts at 0KB, the inode bitmap is at address 4KB, the data bitmap at 8KB, and thus the inode table comes right after). In vsfs, we thus have the following layout for the beginning of the file system partition (in closeup view):



To read inode number 32, the file system would first calculate the offset into the inode region ($32 \cdot \text{sizeof}(\text{inode})$ or 8192), add it to the start address of the inode table on disk ($\text{inodeStartAddr} = 12KB$), and thus arrive upon the correct byte address of the desired block of inodes: 20KB. Recall that disks are not byte addressable, but rather consist of a large number of addressable sectors, usually 512 bytes. Thus, to fetch the block of inodes that contains inode 32, the file system would issue a read to sector $\frac{20 \times 1024}{512}$, or 40, to fetch the desired inode block. More generally, the sector address $iaddr$ of the inode block can be calculated as follows:

```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Inside each inode is virtually all of the information you need about a file: its *type* (e.g., regular file, directory, etc.), its *size*, the number of *blocks*

Size	Name	What is this inode field for?
2	mode	can this file be read /written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file.acl	a new permissions model beyond mode bits
4	dir.acl	called access control lists

Figure 40.1: Simplified Ext2 Inode

allocated to it, *protection information* (such as who owns the file, as well as who can access it), some *time* information, including when the file was created, modified, or last accessed, as well as information about where its data blocks reside on disk (e.g., pointers of some kind). We refer to all such information about a file as **metadata**; in fact, any information inside the file system that isn't pure user data is often referred to as such. An example inode from ext2 [P09] is shown in Figure 40.1¹.

One of the most important decisions in the design of the inode is how it refers to where data blocks are. One simple approach would be to have one or more **direct pointers** (disk addresses) inside the inode; each pointer refers to one disk block that belongs to the file. Such an approach is limited: for example, if you want to have a file that is really big (e.g., bigger than the size of a block multiplied by the number of direct pointers), you are out of luck.

The Multi-Level Index

To support bigger files, file system designers have had to introduce different structures within inodes. One common idea is to have a special pointer known as an **indirect pointer**. Instead of pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data. Thus, an inode may have some fixed number of direct pointers (e.g., 12), and a single indirect pointer. If a file grows large enough, an indirect block is allocated (from the data-block region of the disk), and the inode's slot for an indirect pointer is set to point to it. Assuming 4-KB blocks and 4-byte disk addresses, that adds another 1024 pointers; the file can grow to be $(12 + 1024) \cdot 4K$ or 4144KB.

¹Type info is kept in the directory entry, and thus is not found in the inode itself.

TIP: CONSIDER EXTENT-BASED APPROACHES

A different approach is to use **extents** instead of pointers. An extent is simply a disk pointer plus a length (in blocks); thus, instead of requiring a pointer for every block of a file, all one needs is a pointer and a length to specify the on-disk location of a file. Just a single extent is limiting, as one may have trouble finding a contiguous chunk of on-disk free space when allocating a file. Thus, extent-based file systems often allow for more than one extent, thus giving more freedom to the file system during file allocation.

In comparing the two approaches, pointer-based approaches are the most flexible but use a large amount of metadata per file (particularly for large files). Extent-based approaches are less flexible but more compact; in particular, they work well when there is enough free space on the disk and files can be laid out contiguously (which is the goal for virtually any file allocation policy anyhow).

Not surprisingly, in such an approach, you might want to support even larger files. To do so, just add another pointer to the inode: the **double indirect pointer**. This pointer refers to a block that contains pointers to indirect blocks, each of which contain pointers to data blocks. A double indirect block thus adds the possibility to grow files with an additional $1024 \cdot 1024$ or 1-million 4KB blocks, in other words supporting files that are over 4GB in size. You may want even more, though, and we bet you know where this is headed: the **triple indirect pointer**.

Overall, this imbalanced tree is referred to as the **multi-level index** approach to pointing to file blocks. Let's examine an example with twelve direct pointers, as well as both a single and a double indirect block. Assuming a block size of 4 KB, and 4-byte pointers, this structure can accommodate a file of just over 4 GB in size (i.e., $(12 + 1024 + 1024^2) \times 4 \text{ KB}$). Can you figure out how big of a file can be handled with the addition of a triple-indirect block? (hint: pretty big)

Many file systems use a multi-level index, including commonly-used file systems such as Linux ext2 [P09] and ext3, NetApp's WAFL, as well as the original UNIX file system. Other file systems, including SGI XFS and Linux ext4, use **extents** instead of simple pointers; see the earlier aside for details on how extent-based schemes work (they are akin to segments in the discussion of virtual memory).

You might be wondering: why use an imbalanced tree like this? Why not a different approach? Well, as it turns out, many researchers have studied file systems and how they are used, and virtually every time they find certain "truths" that hold across the decades. One such finding is that *most files are small*. This imbalanced design reflects such a reality; if most files are indeed small, it makes sense to optimize for this case. Thus, with a small number of direct pointers (12 is a typical number), an inode

ASIDE: LINKED-BASED APPROACHES

Another simpler approach in designing inodes is to use a **linked list**. Thus, inside an inode, instead of having multiple pointers, you just need one, to point to the first block of the file. To handle larger files, add another pointer at the end of that data block, and so on, and thus you can support large files.

As you might have guessed, linked file allocation performs poorly for some workloads; think about reading the last block of a file, for example, or just doing random access. Thus, to make linked allocation work better, some systems will keep an in-memory table of link information, instead of storing the next pointers with the data blocks themselves. The table is indexed by the address of a data block *D*; the content of an entry is simply *D*'s next pointer, i.e., the address of the next block in a file which follows *D*. A null-value could be there too (indicating an end-of-file), or some other marker to indicate that a particular block is free. Having such a table of next pointers makes it so that a linked allocation scheme can effectively do random file accesses, simply by first scanning through the (in memory) table to find the desired block, and then accessing (on disk) it directly.

Does such a table sound familiar? What we have described is the basic structure of what is known as the **file allocation table**, or **FAT** file system. Yes, this classic old Windows file system, before NTFS [C94], is based on a simple linked-based allocation scheme. There are other differences from a standard UNIX file system too; for example, there are no inodes per se, but rather directory entries which store metadata about a file and refer directly to the first block of said file, which makes creating hard links impossible. See Brouwer [B02] for more of the inelegant details.

can directly point to 48 KB of data, needing one (or more) indirect blocks for larger files. See Agrawal et. al [A+07] for a recent study; Figure 40.2 summarizes those results.

Of course, in the space of inode design, many other possibilities exist; after all, the inode is just a data structure, and any data structure that stores the relevant information, and can query it effectively, is sufficient. As file system software is readily changed, you should be willing to explore different designs should workloads or technologies change.

Most files are small	Roughly 2K is the most common size
Average file size is growing	Almost 200K is the average
Most bytes are stored in large files	A few big files use most of the space
File systems contains lots of files	Almost 100K on average
File systems are roughly half full	Even as disks grow, file systems remain ~50% full
Directories are typically small	Many have few entries; most have 20 or fewer

Figure 40.2: File System Measurement Summary

40.4 Directory Organization

In vsfs (as in many file systems), directories have a simple organization; a directory basically just contains a list of (entry name, inode number) pairs. For each file or directory in a given directory, there is a string and a number in the data block(s) of the directory. For each string, there may also be a length (assuming variable-sized names).

For example, assume a directory `dir` (inode number 5) has three files in it (`foo`, `bar`, and `foobar`), and their inode numbers are 12, 13, and 24 respectively. The on-disk data for `dir` might look like this:

inum	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

In this example, each entry has an inode number, record length (the total bytes for the name plus any left over space), string length (the actual length of the name), and finally the name of the entry. Note that each directory has two extra entries, . “dot” and .. “dot-dot”; the dot directory is just the current directory (in this example, `dir`), whereas dot-dot is the parent directory (in this case, the root).

Deleting a file (e.g., calling `unlink()`) can leave an empty space in the middle of the directory, and hence there should be some way to mark that as well (e.g., with a reserved inode number such as zero). Such a delete is one reason the record length is used: a new entry may reuse an old, bigger entry and thus have extra space within.

You might be wondering where exactly directories are stored. Often, file systems treat directories as a special type of file. Thus, a directory has an inode, somewhere in the inode table (with the type field of the inode marked as “directory” instead of “regular file”). The directory has data blocks pointed to by the inode (and perhaps, indirect blocks); these data blocks live in the data block region of our simple file system. Our on-disk structure thus remains unchanged.

We should also note again that this simple linear list of directory entries is not the only way to store such information. As before, any data structure is possible. For example, XFS [S+96] stores directories in B-tree form, making file create operations (which have to ensure that a file name has not been used before creating it) faster than systems with simple lists that must be scanned in their entirety.

40.5 Free Space Management

A file system must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it. Thus **free space management** is important for all file systems. In vsfs, we have two simple bitmaps for this task.

ASIDE: FREE SPACE MANAGEMENT

There are many ways to manage free space; bitmaps are just one way. Some early file systems used **free lists**, where a single pointer in the super block was kept to point to the first free block; inside that block the next free pointer was kept, thus forming a list through the free blocks of the system. When a block was needed, the head block was used and the list updated accordingly.

Modern file systems use more sophisticated data structures. For example, SGI's XFS [S+96] uses some form of a **B-tree** to compactly represent which chunks of the disk are free. As with any data structure, different time-space trade-offs are possible.

For example, when we create a file, we will have to allocate an inode for that file. The file system will thus search through the bitmap for an inode that is free, and allocate it to the file; the file system will have to mark the inode as used (with a 1) and eventually update the on-disk bitmap with the correct information. A similar set of activities take place when a data block is allocated.

Some other considerations might also come into play when allocating data blocks for a new file. For example, some Linux file systems, such as ext2 and ext3, will look for a sequence of blocks (say 8) that are free when a new file is created and needs data blocks; by finding such a sequence of free blocks, and then allocating them to the newly-created file, the file system guarantees that a portion of the file will be contiguous on the disk, thus improving performance. Such a **pre-allocation** policy is thus a commonly-used heuristic when allocating space for data blocks.

40.6 Access Paths: Reading and Writing

Now that we have some idea of how files and directories are stored on disk, we should be able to follow the flow of operation during the activity of reading or writing a file. Understanding what happens on this **access path** is thus the second key in developing an understanding of how a file system works; pay attention!

For the following examples, let us assume that the file system has been mounted and thus that the superblock is already in memory. Everything else (i.e., inodes, directories) is still on the disk.

Reading A File From Disk

In this simple example, let us first assume that you want to simply open a file (e.g., `/foo/bar`), read it, and then close it. For this simple example, let's assume the file is just 4KB in size (i.e., 1 block).

When you issue an `open("/foo/bar", O_RDONLY)` call, the file system first needs to find the inode for the file `bar`, to obtain some basic information about the file (permissions information, file size, etc.). To do so,

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
open(bar)			read			read				
			read			read				
			read			read				
read()			read			read				
read()			write			read				
read()			read			read				
read()			write			read				
read()			read			read				
read()			write			read				
read()			read			read				

Figure 40.3: File Read Timeline (Time Increasing Downward)

the file system must be able to find the inode, but all it has right now is the full pathname. The file system must **traverse** the pathname and thus locate the desired inode.

All traversals begin at the root of the file system, in the **root directory** which is simply called `/`. Thus, the first thing the FS will read from disk is the inode of the root directory. But where is this inode? To find an inode, we must know its i-number. Usually, we find the i-number of a file or directory in its parent directory; the root has no parent (by definition). Thus, the root inode number must be “well known”; the FS must know what it is when the file system is mounted. In most UNIX file systems, the root inode number is 2. Thus, to begin the process, the FS reads in the block that contains inode number 2 (the first inode block).

Once the inode is read in, the FS can look inside of it to find pointers to data blocks, which contain the contents of the root directory. The FS will thus use these on-disk pointers to read through the directory, in this case looking for an entry for `foo`. By reading in one or more directory data blocks, it will find the entry for `foo`; once found, the FS will also have found the inode number of `foo` (say it is 44) which it will need next.

The next step is to recursively traverse the pathname until the desired inode is found. In this example, the FS reads the block containing the inode of `foo` and then its directory data, finally finding the inode number of `bar`. The final step of `open()` is to read `bar`’s inode into memory; the FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user.

Once open, the program can then issue a `read()` system call to read from the file. The first read (at offset 0 unless `lseek()` has been called) will thus read in the first block of the file, consulting the inode to find the location of such a block; it may also update the inode with a new last-accessed time. The read will further update the in-memory open file table for this file descriptor, updating the file offset such that the next read will read the second file block, etc.

ASIDE: READS DON'T ACCESS ALLOCATION STRUCTURES

We've seen many students get confused by allocation structures such as bitmaps. In particular, many often think that when you are simply reading a file, and not allocating any new blocks, that the bitmap will still be consulted. This is not true! Allocation structures, such as bitmaps, are only accessed when allocation is needed. The inodes, directories, and indirect blocks have all the information they need to complete a read request; there is no need to make sure a block is allocated when the inode already points to it.

At some point, the file will be closed. There is much less work to be done here; clearly, the file descriptor should be deallocated, but for now, that is all the FS really needs to do. No disk I/Os take place.

A depiction of this entire process is found in Figure 40.3 (time increases downward). In the figure, the open causes numerous reads to take place in order to finally locate the inode of the file. Afterwards, reading each block requires the file system to first consult the inode, then read the block, and then update the inode's last-accessed-time field with a write. Spend some time and try to understand what is going on.

Also note that the amount of I/O generated by the open is proportional to the length of the pathname. For each additional directory in the path, we have to read its inode as well as its data. Making this worse would be the presence of large directories; here, we only have to read one block to get the contents of a directory, whereas with a large directory, we might have to read many data blocks to find the desired entry. Yes, life can get pretty bad when reading a file; as you're about to find out, writing out a file (and especially, creating a new one) is even worse.

Writing to Disk

Writing to a file is a similar process. First, the file must be opened (as above). Then, the application can issue `write()` calls to update the file with new contents. Finally, the file is closed.

Unlike reading, writing to the file may also **allocate** a block (unless the block is being overwritten, for example). When writing out a new file, each write not only has to write data to disk but has to first decide which block to allocate to the file and thus update other structures of the disk accordingly (e.g., the data bitmap and inode). Thus, each write to a file logically generates five I/Os: one to read the data bitmap (which is then updated to mark the newly-allocated block as used), one to write the bitmap (to reflect its new state to disk), two more to read and then write the inode (which is updated with the new block's location), and finally one to write the actual block itself.

The amount of write traffic is even worse when one considers a simple and common operation such as file creation. To create a file, the file system must not only allocate an inode, but also allocate space within

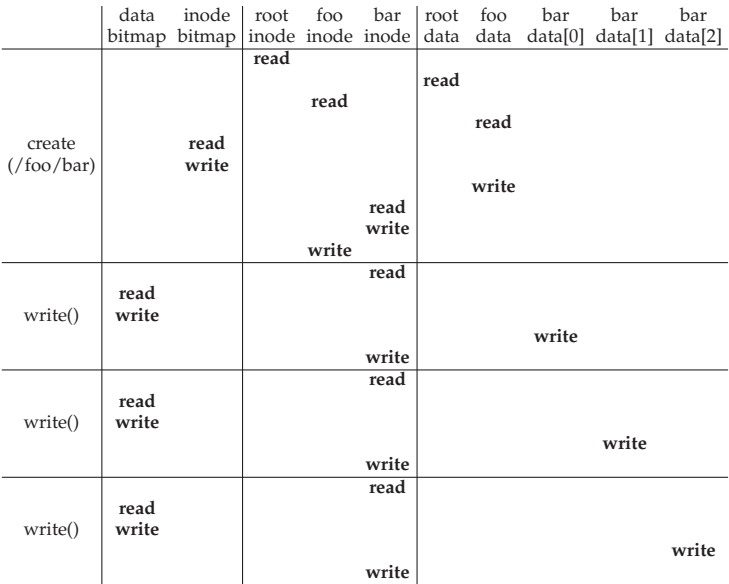


Figure 40.4: File Creation Timeline (Time Increasing Downward)

the directory containing the new file. The total amount of I/O traffic to do so is quite high: one read to the inode bitmap (to find a free inode), one write to the inode bitmap (to mark it allocated), one write to the new inode itself (to initialize it), one to the data of the directory (to link the high-level name of the file to its inode number), and one read and write to the directory inode to update it. If the directory needs to grow to accommodate the new entry, additional I/Os (i.e., to the data bitmap, and the new directory block) will be needed too. All that just to create a file!

Let’s look at a specific example, where the file `/foo/bar` is created, and three blocks are written to it. Figure 40.4 shows what happens during the `open()` (which creates the file) and during each of three 4KB writes.

In the figure, reads and writes to the disk are grouped under which system call caused them to occur, and the rough ordering they might take place in goes from top to bottom of the figure. You can see how much work it is to create the file: 10 I/Os in this case, to walk the pathname and then finally create the file. You can also see that each allocating write costs 5 I/Os: a pair to read and update the inode, another pair to read and update the data bitmap, and then finally the write of the data itself. How can a file system accomplish any of this with reasonable efficiency?

THE CRUX: HOW TO REDUCE FILE SYSTEM I/O COSTS

Even the simplest of operations like opening, reading, or writing a file incurs a huge number of I/O operations, scattered over the disk. What can a file system do to reduce the high costs of doing so many I/Os?

40.7 Caching and Buffering

As the examples above show, reading and writing files can be expensive, incurring many I/Os to the (slow) disk. To remedy what would clearly be a huge performance problem, most file systems aggressively use system memory (DRAM) to cache important blocks.

Imagine the open example above: without caching, every file open would require at least two reads for every level in the directory hierarchy (one to read the inode of the directory in question, and at least one to read its data). With a long pathname (e.g., `/1/2/3/ ... /100/file.txt`), the file system would literally perform hundreds of reads just to open the file!

Early file systems thus introduced a **fixed-size cache** to hold popular blocks. As in our discussion of virtual memory, strategies such as **LRU** and different variants would decide which blocks to keep in cache. This fixed-size cache would usually be allocated at boot time to be roughly 10% of total memory.

This **static partitioning** of memory, however, can be wasteful; what if the file system doesn't need 10% of memory at a given point in time? With the fixed-size approach described above, unused pages in the file cache cannot be re-purposed for some other use, and thus go to waste.

Modern systems, in contrast, employ a **dynamic partitioning** approach. Specifically, many modern operating systems integrate virtual memory pages and file system pages into a **unified page cache** [S00]. In this way, memory can be allocated more flexibly across virtual memory and file system, depending on which needs more memory at a given time.

Now imagine the file open example with caching. The first open may generate a lot of I/O traffic to read in directory inode and data, but subsequent file opens of that same file (or files in the same directory) will mostly hit in the cache and thus no I/O is needed.

Let us also consider the effect of caching on writes. Whereas read I/O can be avoided altogether with a sufficiently large cache, write traffic has to go to disk in order to become persistent. Thus, a cache does not serve as the same kind of filter on write traffic that it does for reads. That said, **write buffering** (as it is sometimes called) certainly has a number of performance benefits. First, by delaying writes, the file system can **batch** some updates into a smaller set of I/Os; for example, if an inode bitmap is updated when one file is created and then updated moments later as another file is created, the file system saves an I/O by delaying the write after the first update. Second, by buffering a number of writes in memory,

TIP: UNDERSTAND STATIC VS. DYNAMIC PARTITIONING

When dividing a resource among different clients/users, you can use either **static partitioning** or **dynamic partitioning**. The static approach simply divides the resource into fixed proportions once; for example, if there are two possible users of memory, you can give some fixed fraction of memory to one user, and the rest to the other. The dynamic approach is more flexible, giving out differing amounts of the resource over time; for example, one user may get a higher percentage of disk bandwidth for a period of time, but then later, the system may switch and decide to give a different user a larger fraction of available disk bandwidth.

Each approach has its advantages. Static partitioning ensures each user receives some share of the resource, usually delivers more predictable performance, and is often easier to implement. Dynamic partitioning can achieve better utilization (by letting resource-hungry users consume otherwise idle resources), but can be more complex to implement, and can lead to worse performance for users whose idle resources get consumed by others and then take a long time to reclaim when needed. As is often the case, there is no best method; rather, you should think about the problem at hand and decide which approach is most suitable. Indeed, shouldn't you always be doing that?

the system can then **schedule** the subsequent I/Os and thus increase performance. Finally, some writes are avoided altogether by delaying them; for example, if an application creates a file and then deletes it, delaying the writes to reflect the file creation to disk **avoids** them entirely. In this case, laziness (in writing blocks to disk) is a virtue.

For the reasons above, most modern file systems buffer writes in memory for anywhere between five and thirty seconds, representing yet another trade-off: if the system crashes before the updates have been propagated to disk, the updates are lost; however, by keeping writes in memory longer, performance can be improved by batching, scheduling, and even avoiding writes.

Some applications (such as databases) don't enjoy this trade-off. Thus, to avoid unexpected data loss due to write buffering, they simply force writes to disk, by calling `fsync()`, by using **direct I/O** interfaces that work around the cache, or by using the **raw disk** interface and avoiding the file system altogether². While most applications live with the trade-offs made by the file system, there are enough controls in place to get the system to do what you want it to, should the default not be satisfying.

²Take a database class to learn more about old-school databases and their former insistence on avoiding the OS and controlling everything themselves. But watch out! Those database types are always trying to bad mouth the OS. Shame on you, database people. Shame.

TIP: UNDERSTAND THE DURABILITY/PERFORMANCE TRADE-OFF

Storage systems often present a durability/performance trade-off to users. If the user wishes data that is written to be immediately durable, the system must go through the full effort of committing the newly-written data to disk, and thus the write is slow (but safe). However, if the user can tolerate the loss of a little data, the system can buffer writes in memory for some time and write them later to the disk (in the background). Doing so makes writes appear to complete quickly, thus improving perceived performance; however, if a crash occurs, writes not yet committed to disk will be lost, and hence the trade-off. To understand how to make this trade-off properly, it is best to understand what the application using the storage system requires; for example, while it may be tolerable to lose the last few images downloaded by your web browser, losing part of a database transaction that is adding money to your bank account may be less tolerable. Unless you're rich, of course; in that case, why do you care so much about hoarding every last penny?

40.8 Summary

We have seen the basic machinery required in building a file system. There needs to be some information about each file (metadata), usually stored in a structure called an inode. Directories are just a specific type of file that store name→inode-number mappings. And other structures are needed too; for example, file systems often use a structure such as a bitmap to track which inodes or data blocks are free or allocated.

The terrific aspect of file system design is its freedom; the file systems we explore in the coming chapters each take advantage of this freedom to optimize some aspect of the file system. There are also clearly many policy decisions we have left unexplored. For example, when a new file is created, where should it be placed on disk? This policy and others will also be the subject of future chapters. Or will they?³

³Cue mysterious music that gets you even more intrigued about the topic of file systems.

References

- [A+07] Nitin Agrawal, William J. Bolosky, John R. Douceur, Jacob R. Lorch
A Five-Year Study of File-System Metadata
FAST '07, pages 31–45, February 2007, San Jose, CA
An excellent recent analysis of how file systems are actually used. Use the bibliography within to follow the trail of file-system analysis papers back to the early 1980s.
- [B07] “ZFS: The Last Word in File Systems”
Jeff Bonwick and Bill Moore
Available: <http://opensolaris.org/os/community/zfs/docs/zfs.last.pdf>
One of the most recent important file systems, full of features and awesomeness. We should have a chapter on it, and perhaps soon will.
- [B02] “The FAT File System”
Andries Brouwer
September, 2002
Available: <http://www.win.tue.nl/~aeb/linux/fs/fat/fat.html>
A nice clean description of FAT. The file system kind, not the bacon kind. Though you have to admit, bacon fat probably tastes better.
- [C94] “Inside the Windows NT File System”
Helen Custer
Microsoft Press, 1994
A short book about NTFS; there are probably ones with more technical details elsewhere.
- [H+88] “Scale and Performance in a Distributed File System”
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West.
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988
A classic distributed file system; we’ll be learning more about it later, don’t worry.
- [P09] “The Second Extended File System: Internal Layout”
Dave Poirier, 2009
Available: <http://www.nongnu.org/ext2-doc/ext2.html>
Some details on ext2, a very simple Linux file system based on FFS, the Berkeley Fast File System. We’ll be reading about it in the next chapter.
- [RT74] “The UNIX Time-Sharing System”
M. Ritchie and K. Thompson
CACM, Volume 17:7, pages 365-375, 1974
The original paper about UNIX. Read it to see the underpinnings of much of modern operating systems.
- [S00] “UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD”
Chuck Silvers
FREENIX, 2000
A nice paper about NetBSD’s integration of file-system buffer caching and the virtual-memory page cache. Many other systems do the same type of thing.
- [S+96] “Scalability in the XFS File System”
Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, Geoff Peck
USENIX ’96, January 1996, San Diego, CA
The first attempt to make scalability of operations, including things like having millions of files in a directory, a central focus. A great example of pushing an idea to the extreme. The key idea behind this file system: everything is a tree. We should have a chapter on this file system too.

Homework

Use this tool, `vsfs.py`, to study how file system state changes as various operations take place. The file system begins in an empty state, with just a root directory. As the simulation takes place, various operations are performed, thus slowly changing the on-disk state of the file system. See the README for details.

Questions

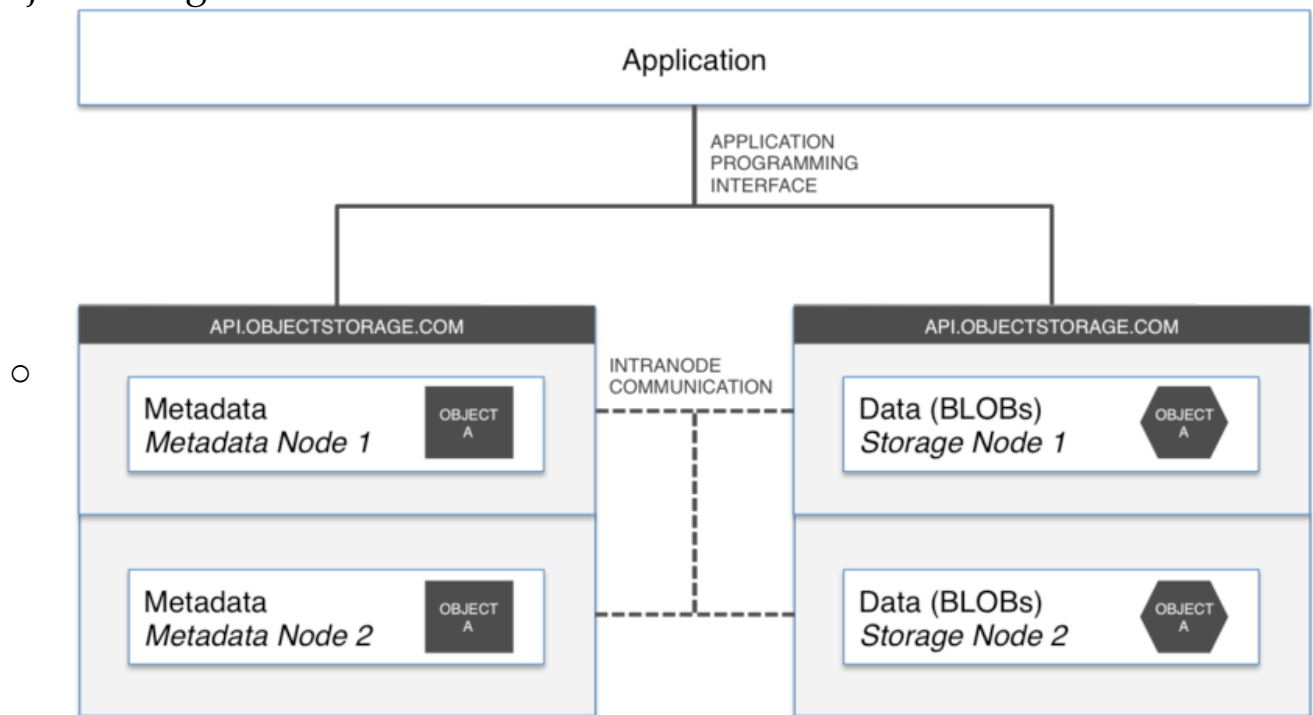
1. Run the simulator with some different random seeds (say 17, 18, 19, 20), and see if you can figure out which operations must have taken place between each state change.
2. Now do the same, using different random seeds (say 21, 22, 23, 24), except run with the `-r` flag, thus making you guess the state change while being shown the operation. What can you conclude about the inode and data-block allocation algorithms, in terms of which blocks they prefer to allocate?
3. Now reduce the number of data blocks in the file system, to very low numbers (say two), and run the simulator for a hundred or so requests. What types of files end up in the file system in this highly-constrained layout? What types of operations would fail?
4. Now do the same, but with inodes. With very few inodes, what types of operations can succeed? Which will usually fail? What is the final state of the file system likely to be?

Wikipedia

Wednesday, November 9, 2016

10:23 PM

- File format
 - A file format is a standard way that information is encoded for storage in a computer file. It specifies how bits are used to encode information in a digital storage medium
- Object storage



- Storage architecture that manages data as objects that include the data itself, a variable amount of metadata and a globally unique identifier.
- Object storage can be implemented at multiple levels, including the device level (object storage device), the system level, and interface level.
- Abstraction of storage
 - Abstracts lower layers of storage away and data is exposed and managed as objects instead of files or blocks.
 - Do not have to deal with constructing or managing logical volumes to utilize disk capacity
 - Allows the addressing and identification of individual objects by more than just file name and file path
 - Adds a unique identifier within a bucket, or across the entire system. to support much larger namespaces and eliminate

name collisions.

- Inclusion of rich custom metadata within the object
 - PURPOSE
 - To capture application/user-specific information for better indexing purposes
 - To support data management policies
 - To centralize management of storage across many individual nodes and clusters
 - To optimize metadata storage independently from data storage
 - Data is organized into flexible-sized data containers called objects, rather than fixed sized blocks of data
 - Each object has both data and metadata
 - Physically encapsulating both together benefits recoverability

- Key-value database

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623

- A data storage paradigm designed for storing, retrieving, and managing associative arrays known as a dictionary.
- Key-value systems treat the data as a single opaque collection which may have different fields for every record
 - + Offers considerable flexibility and more closely follows modern concepts like OOP
 - + Use far less memory to store the same database
 - * Relational databases (RDB's) conversely structure the database with well defined data types
- Filesystem in Userspace (FUSE)
 - Software interface that lets non-privileged users create their own file systems without editing kernel code
 - Achieved by running file system code in user space while the FUSE model provides an interface "bridge" to the actual kernel

FUSE model provides only a bridge to the actual kernel interfaces

- Often useful for writing virtual file systems

An Introduction to DOS FAT Volume and File Structure

Mark Kampe markk@cs.ucla.edu

1. Introduction

When the first personal computers with disks became available, they were very small (a few megabytes of disk and a few dozen kilobytes of memory). A file system implementation for such machines had to impose very little overhead on disk space, and be small enough to fit in the BIOS ROM. BIOS stands for **BASIC I/O Subsystem**. Note that the first word is all upper-case. The purpose of the BIOS ROM was to provide run-time support for a BASIC interpreter (which is what Bill Gates did for a living before building DOS). DOS was never intended to provide the features and performance of real timesharing systems.

Disk and memory size have increased in the last thirty years, People now demand state-of-the-art power and functionality from their PCs. Despite the evolution that the last decades have seen, old standards die hard. Much as European train tracks maintain the same wheel spacing used by Roman chariots, most modern OSs still support DOS FAT file systems. DOS file systems are not merely around for legacy reasons. The ISO 9660 CDROM file system format is a descendent of the DOS file system.

The DOS FAT file system is worth studying because:

- It is heavily used all over the world, and is the basis for more modern file system (like 9660).
- It provides reasonable performance (large transfers and well clustered allocation) with a very simple implementation.
- It is a very successful example of "linked list" space allocation.

2. Structural Overview

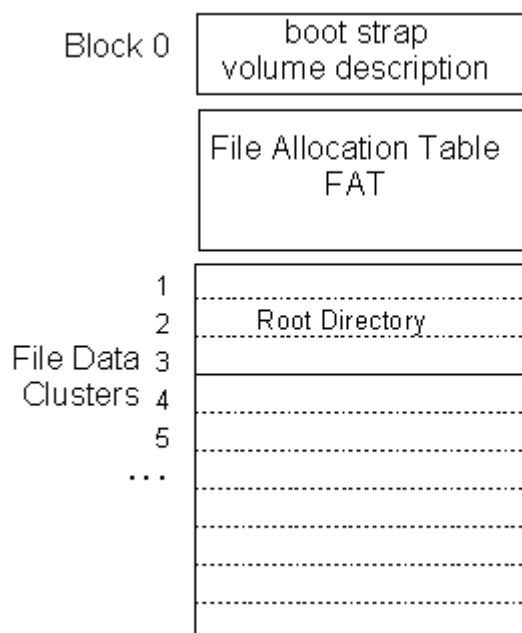
All file systems include a few basic types of data structures:

- bootstrap
code to be loaded into memory and executed when the computer is powered on. MVS volumes reserve the entire first track of the first cylinder for the boot strap.
- volume descriptors
information describing the size, type, and layout of the file system ... and in particular how to find the other key meta-data descriptors.
- file descriptors
information that describes a file (ownership, protection, time of last update, etc.) and points where the actual data is stored on the disk.
- free space descriptors
lists of blocks of (currently) unused space that can be allocated to files.
- file name descriptors
data structures that user-chosen names with each file.

DOS FAT file systems divide the volume into fixed-sized (physical) blocks, which are grouped into larger fixed-sized (logical) block clusters.

The first block of DOS FAT volume contains the bootstrap, along with some volume description information. After this comes a much longer **File Allocation Table** (FAT from which the file system takes its name). The File Allocation Table is used, both as a free list, and to keep track of which blocks have been allocated to which files.

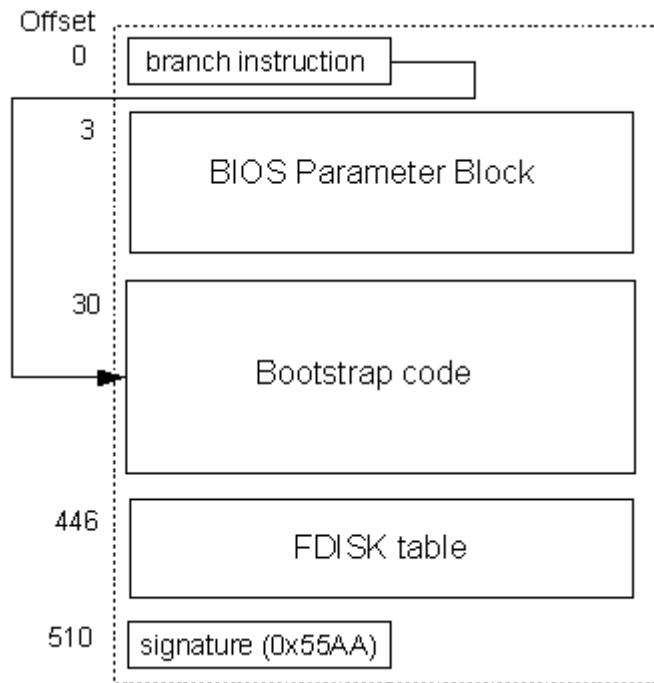
The remainder of the volume is data clusters, which can be allocated to files and directories. The first file on the volume is the root directory, the top of the tree from which all other files and directories on the volume can be reached.



3. Boot block BIOS Parameter Block and FDISK Table

Most file systems separate the first block (pure bootstrap code) from volume description information. DOS file systems often combine these into a single block. The format varies between (partitioned) hard disks and (unpartitioned) floppies, and between various releases of DOS and Windows ... but conceptually, the boot record:

- begins with a branch instruction (to the start of the real bootstrap code).
- followed by a volume description (BIOS Parameter Block)
- followed by the real bootstrap code
- followed by an optional disk partitioning table
- followed by a signature (for error checking).



3.1 BIOS Parameter Block

After the first few bytes of the bootstrap comes the BIOS parameter block, which contains a brief summary of the device and file system. It describes the device geometry:

- number of bytes per (physical) sector
- number of sectors per track
- number of tracks per cylinder
- total number of sectors on the volume

It also describes the way the file system is layed out on the volume:

- number of sectors per (logical) cluster
- the number of reserved sectors (not part of file system)
- the number of Alternate File Allocation Tables
- the number of entries in the root directory

These parameters enable the OS to interpret the remainder of the file system.

3.2 FDISK Table

As disks got larger, the people at MicroSoft figured out that their customers might want to put multiple file systems on each disk. This meant they needed some way of partitioning the disk into logical sub-disks. To do this, they added a small partition table (sometimes called the FDISK table, because of the program that managed it) to the end of the boot strap block.

This FDISK table has four entries, each capable of describing one disk partition. Each entry includes

- A partition type (e.g. Primary DOS partition, UNIX partition).
- An ACTIVE indication (is this the one we boot from).
- The disk address where that partition starts and ends.
- The number of sectors contained within that partition.

Partn	Type	Active	Start (C:H:S)	End (C:H:S)	Start (logical)	Size (sectors)
-------	------	--------	---------------	-------------	-----------------	----------------

1	LINUX	True	1:0:0	199:7:49	400	79,600
2	Windows NT		200:0:0	349:7:49	80,000	60,000
3	FAT 32		350:0:0	399:7:49	140,000	20,000
4	NONE					

In older versions of DOS the starting/ending addresses were specified as cylinder/sector/head. As disks got larger, this became less practical, and they moved to logical block numbers.

The addition of disk partitioning also changed the structure of the boot record. The first sector of a disk contains the Master Boot Record (MBR) which includes the FDISK table, and a bootstrap that finds the active partition, and reads in its first sector (Partition Boot Record). Most people (essentially everyone but Bill Gates :-)) make their MBR bootstrap ask what system you want to boot from, and boot the active one by default after a few seconds. This gives you the opportunity to choose which OS you want to boot. Microsoft makes this decision for you ... you want to boot Windows.

The structure of the Partition Boot Record is entirely operating system and file system specific ... but for DOS FAT file system partitions, it includes a BIOS Parameter block as described above.

4. File Descriptors (directories)

In keeping with their desire for simplicity, DOS file systems combine both file description and file naming into a single file descriptor (directory entries). A DOS directory is a file (of a special type) that contains a series of fixed sized (32 byte) directory entries. Each entry describes a single file:

- an 11-byte name (8 characters of base name, plus a 3 character extension).
- a byte of attribute bits for the file, which include:
 - Is this a file, or a sub-directory.
 - Has this file changed since the last backup.
 - Is this file hidden.
 - Is this file read-only.
 - Is this a system file.
 - Does this entry describe a volume label.
- times and dates of creation and last modification, and date of last access.
- a pointer to the first logical block of the file. (This field is only 16 bits wide, and so when Microsoft introduced the FAT32 file system, they had to put the high order bits in a different part of the directory entry).
- the length (number of valid data bytes) in the file.

Name (8+3)	Attributes	Last Changed	First Cluster	Length
.	DIR	08/01/03 11:15:00	61	2,048
..	DIR	06/20/03 08:10:24	1	4,096
MARK	DIR	10/15/04 21:40:12	130	1,800
README.TXT	FILE	11/02/04 04:27:36	410	31,280

If the first character of a files name is a NULL (0x00) the directory entry is unused. The special character (0xE5) in the first character of a file name is used to indicate that a directory entry describes a deleted file. (See the section on Garbage collection below)

Note on times and dates:

DOS stores file modification times and dates as a pair of 16-bit numbers:

- 7 bits of year, 4 bits of month, 5 bits of day of month
- 5 bits of hour, 6 bits of minute, 5 bits of seconds (x2).

All file systems use dates relative to some epoch (time zero). For DOS, the epoch is midnight, New Year's Eve, January 1, 1980. A seven bit field for years means that the the DOS calendar only runs til 2107. Hopefully, nobody will still be using DOS file systems by then :-)

5. Links and Free Space (File Allocation Table)

Many file systems have very compact (e.g. bitmap) free lists, but most of them use some per-file data structure to keep track of which blocks are allocated to which file. The DOS File Allocation Table is a relatively unique design. It contains one entry for each logical block in the volume. If a block is free, this is indicated by the FAT entry. If a block is allocated to a file, the FAT entry gives the logical block number of the **next** logical block in the file.

5.1 Cluster Size and Performance

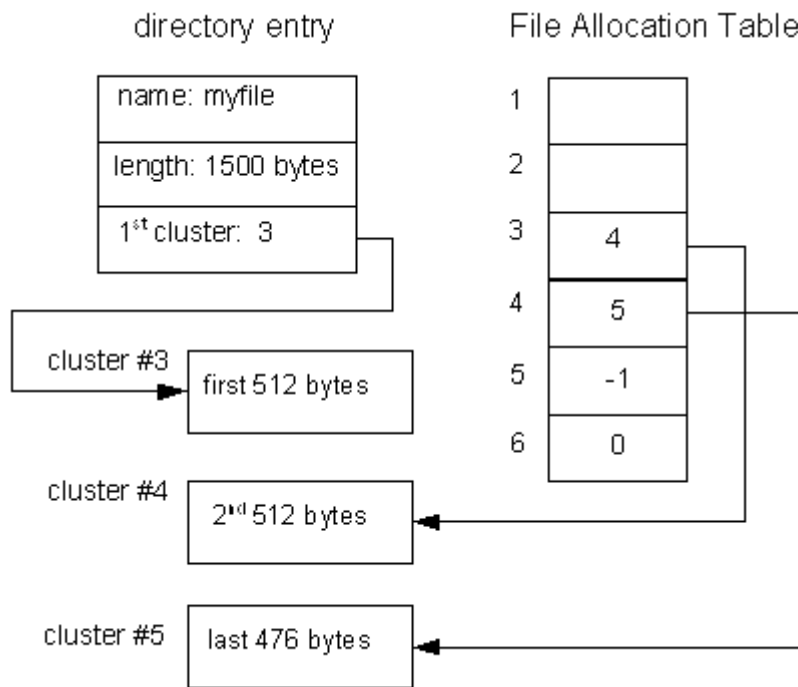
Space is allocated to files, not in (physical) blocks, but in (logical) multi-block clusters. The number of clusters per block is determined when the file system is created.

Allocating space to files in larger chunks improves I/O performance, by reducing the number of operations required to read or write a file. This comes at the cost of higher internal fragmentation (since, on average, half of the last cluster of each file is left unused). As disks have grown larger, people have become less concerned about internal fragmentation losses, and cluster sizes have increased.

The maximum number of clusters a volume can support depends on the width of the FAT entries. In the earliest FAT file systems (designed for use on floppies, and small hard drives). An 8-bit wide FAT entry would have been too small ($256 * 512 = 128K$ bytes) to describe even the smallest floppy, but a 16-bit wide FAT entry would have been ludicrously large (8-16 Megabytes) ... so Microsoft compromised and adopted 12-bit wide FAT entries (two entries in three bytes). These were called FAT-12 file systems. As disks got larger, they created 2-byte wide (FAT-16) and 4-byte wide (FAT-32) file systems.

5.2 Next Block Pointers

A file's directory entry contains a pointer to the first cluster of that file. The File Allocation Table entry for that cluster tells us the cluster number **next** cluster in the file. When we finally get to the last cluster of the file, its FAT entry will contain a -1, indicating that there is no next block in the file.



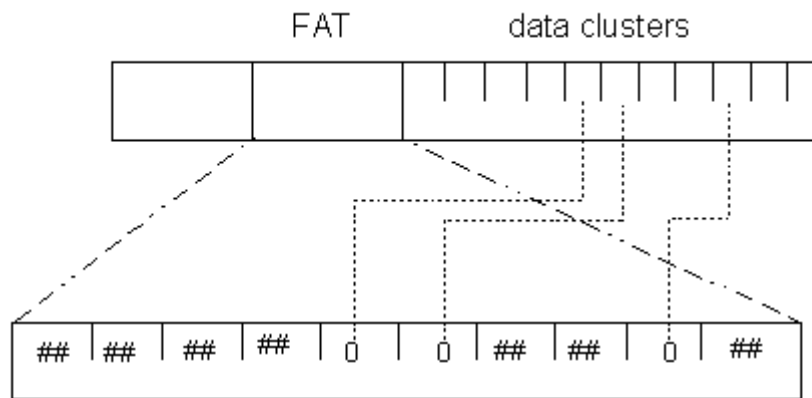
The "next block" organization of the FAT means that in order to figure out what physical cluster is the third logical block of a file, must know the physical cluster number of the second logical block. This is not usually a problem, because almost all file access is sequential (reading the first block, and then the second, and then the third ...).

If we had to go to disk to re-read the FAT each time we needed to figure out the next block number, the file system would perform very poorly. Fortunately, the FAT is so small (e.g. 512 bytes per megabyte of file system) that the entire FAT can be kept in memory as long as a file system is in use. This means that successor block numbers can be looked up without the need to do any additional disk I/O. It is easy to imagine

5.3 Free Space

The notion of "next block" is only meaningful for clusters that are allocated to a file ... which leaves us free to use the FAT entries associated with free clusters as a free indication. Just as we reserved a value (-1) to mean **end of file** we can reserve another value (0) to mean **this cluster is free**.

To find a free cluster, one has but to search the FAT for an entry with the value -2. If we want to find a free cluster near the clusters that are already allocated to the file, we can start our search with the FAT entry after the entry for the first cluster in the file.



Each FAT entry corresponds to one data cluster

A FAT entry of 0 means the corresponding data cluster is not allocated to any file.

5.4 Garbage Collection

Older versions of FAT file systems did not bother to free blocks when a file was deleted. Rather, they merely crossed out the first byte of the file name in the directory entry (with the reserved value 0xE5). This had the advantage of greatly reducing the amount of I/O associated with file deletion ... but it meant that DOS file systems regularly ran out of space.

When this happened, they would initiate garbage collection. Starting from the root directory, they would find every "valid" entry. They would follow the chain of next block pointers to determine which clusters were associated with each file, and recursively enumerate the contents of all sub-directories. After completing the enumeration of all allocated clusters, they inferred that any cluster not found in some file was free, and marked them as such in the File Allocation Table.

This "feature" was probably motivated by a combination of laziness and a desire for performance. It did, however, have an advantage. Since clusters were not freed when files were deleted, they could not be reallocated until after garbage collection was performed. This meant that it might be possible to recover the contents of deleted files for quite a while. The opportunity this created was large enough to enable Peter Norton to start a very successful company.

6. Descendents of the DOS file system

The DOS file system has evolved with time. Not only have wider (16- and 32-bit) FAT entries been used to support larger disks, but other features have been added. The last stand-alone DOS product was DOS 6.x. After this, all DOS support was under Windows, and along with the change to Windows came an enhanced version of the FAT file system called Virtual FAT (or simply VFAT).

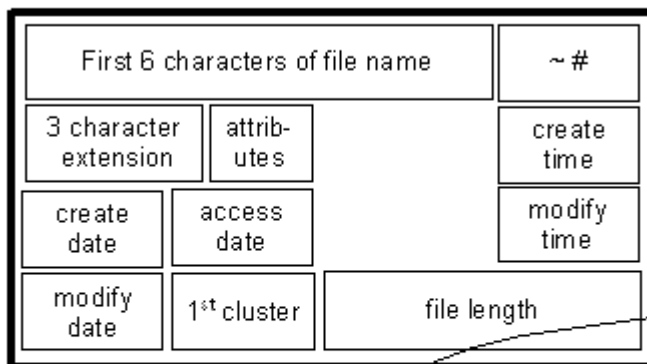
6.1 Long File Names

Most DOS and Windows systems were used for personal productivity, and their users didn't demand much in the way of file system features. Their biggest complaints were about the 8+3 file names. Windows users demanded longer and mixed-case file names.

The 32 byte directory entries didn't have enough room to hold longer names, and changing the format of DOS directories would break hundreds or even thousands of applications. It wasn't merely a matter of importing files from old systems to new systems. DOS diskettes are commonly used to carry files between various systems ... which means that old systems still had to be able to read the new directories. They had to find a way to support longer file names without making the new files unreadable by older systems.

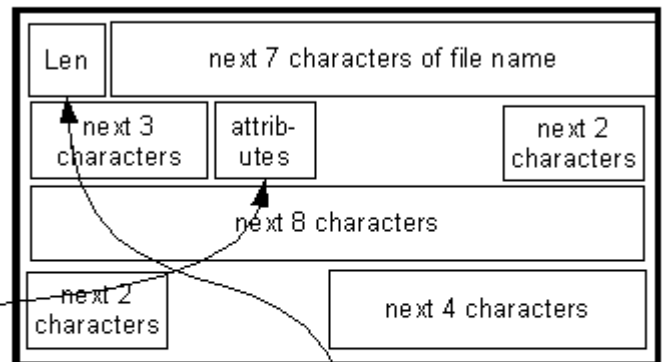
The solution they came up was to put the extended filenames in additional (auxiliary) directory entries. Each file would be described by an old-format directory entry, but supplementary directory entries (following the primary directory entry) could provide extensions to the file name. To keep older systems from being confused by the new directory entries, they were tagged with a very unusual set of attributes (hidden, system, read-only, volume label). Older systems would ignore new entries, but would still be able to access new files under their 8+3 names in the old-style directory entries. New systems would recognize the new directory entries, and be able to access files by either the long or the short names.

Primary (old style) directory entry



Attributes (Read Only, Hidden, System, Volume) identify this as an continuation directory entry. Older systems will ignore such entries.

Secondary (continuation) directory entry



Length field says how many more bytes of name are contained in this entry.

The addition of long file names did create one problem for the old directory entries. What would you do if you had two file names that differed only after the 8th character (e.g. datafileread.c and datafilewrite.c)? If we just used the first 8 characters of the file name in the old directory entries (datafile), we would have two files with the same name, and this is illegal. For this reason, the the short file names are not merely the first eight characters of the long file names. Rather, the last two bytes of the short name were merely made unambiguous (e.g. "~1", "~2", etc).

6.2 Alternate/back-up FATs

The File Allocation Table is a very concise way of keeping track of all of the next-block pointers in the file system. If anything ever happened to the File Allocation Table, the results would be disastrous. The directory entries would tell us where the first blocks of all files were, but we would have no way of figuring out where the remainder of the data was.

Events that corrupt the File Allocation Table are extremely rare, but the consequences of such an incident are catastrophic. To protect users from such eventualities, MicroSoft added support for alternate FATs. Periodically, the primary FAT would be copied to one of the pre-reserved alternat FAT locations. Then if something bad happened to the primary FAT, it would still be possible to read most files (files created before the copy) by using the back-up FAT. This is an imperfect solution, as losing new files is bad ... but losing all files is worse.

6.3 ISO 9660

When CDs were proposed for digital storage, everyone recognized the importance of a single standard file system format. Dueling formats would raise the cost of producing new products ... and this would be a lose for everyone. To respond to this need, the International Standards Organization chartered a sub-committee to propose such a standard.

The failings of the DOS file system were widely known by this time, but, as the most widely used file system on the planet, the committee members could not ignore it. Upon examination, it became clear that the most ideomatic features of the DOS file system (the File Allocation Table) were irrelevant to a CDROM file system (which is written only once):

- We don't need to keep track of the free space on a CD ROM. We write each file contiguously, and the next file goes immidiately after the last one.
- Because files can be written contiguously, we don't need any "next block" pointers. All we need to know about a file is where its first block resides.

It was decided that ISO 9660 file systems would (like DOS file systems) have tree structured directory hierarchies, and that (like DOS) each directory entry would describe a single file (rather than having some auxiliary data structure like and I-node to do this). 9660 directory entries, like DOS directory entries, contain:

- file name (within the current directory)
- file type (e.g. file or directory)
- location of the file's first block
- number of bytes contained in the file
- time and date of creation

They did, however, learn from DOS's mistakes:

- Realizing that new information would be added to directory entries over time, they made them variable length. Each directory entry begins with a length field (giving the number of bytes in this directory entry, and thus the number of bytes until the next directory entry).
- Recognizing the need to support long file names, they also made the file name field in each entry a variable length field.
- Recognizing that, over time, people would want to associate a wide range of attributes with files, they also created a variable length extened attributes section after the file name. Much of this section has been left unused, but they defined several new attributes for files:
 - file owner
 - owning group
 - permissions
 - creation, modification, effective, and expiration times
 - record format, attributes, and length information

But, even though 9660 directory entries include much more information than DOS directory entries, it remains that 9660 volumes resemble DOS file systems much more than they resemble any other file system format. And so, the humble DOS file system is reborn in a new generation of products.

7. Summary

DOS file systems are very simple, They don't support multiple links to a file, or symbolic links, or even multi-user access control. They are also and very economical in terms of the space they take up. Free block lists, and file block pointers are combined into a single (quite compact) File Allocation Table. File descriptors are incorporated into the directory entries. And for all of these limitations, they are probably the most widely used

file system format on the planet. Despite their primitiveness, DOS file systems were used as the basis for much newer CD ROM file system designs.

What can we infer from this? That most users don't need alot of fancy features, and that the DOS file system (primitive as it may be) covers their needs pretty well.

It is also noteworthy that when Microsoft was finally forced to change the file system format to get past the 8.3 upper case file name limitations, they chose to do so with a klugy (but upwards compatible) solution using additional directory entries per file. The fact that they chose such an implementation clearly illustrates the importance of maintaining media interchangeability with older systems. This too is a problem that all (successful) file systems will eventually face.

8. References

DOS file system information

- PC Guide's [Overview of DOS FAT file systems](#). (this is a pointer to the long filename article ... but the entire library is nothing short of excellent).
- Free BSD sources, PCFS implementation, [BIOS Parameter block format](#), and (Open Solaris) [FDISK table format](#).
- Free BSD sources, PCFS implementation, [Directory Entry format](#)

9660 file system information

- Wikipedia Introduction to [ISO 9660 file systems](#)
- Free BSD sources, ISOFS implementation, [ISO 9660 data structures](#).