



Where Are You, Exactly?



A GPS Introduction

Find it on the O'Reilly Network

Published on [The O'Reilly Network](http://www.oreillynet.com/) (<http://www.oreillynet.com/>)
http://www.oreillynet.com/pub/a/bsd/2001/02/28/FreeBSD_Basics.html



Understanding Unix Filesystems

02/28/2001

In [last week's article](#), we viewed a PC's BIOS partition table and its Unix partition table using the `fdisk` and `disklabel` utilities. Let's continue this week by looking at the `newfs` utility and `inode` tables. The `newfs` utility actually formats your slice with the filesystems you previously specified with the `disklabel` utility. Let's start by taking a closer look at formatting and filesystems in general so we can gain a better appreciation of `newfs`.

There are two types of hard-drive formatting. When you purchased your hard drive, it most likely was already "low level" formatted for you by the manufacturer. Low-level formatting creates the tracks and sectors on the drive; the intersection of these tracks and sectors creates the units of storage known as physical blocks, which are 512 bytes in size.

The second type of formatting is called "high-level" formatting. This type of formatting installs a particular file system onto a slice of your physical drive using a utility such as DOS's `format` or FreeBSD's `newfs`. Some examples of file systems are FAT16, FAT32, NTFS, and FFS. Different file systems may vary in performance, but they usually have two features in common:

- They require some type of table to map block addresses to the files contained within the blocks
- They may also use a "logical" block addressing scheme to try to optimize read/write performance

Let's pretend you're a file system for a moment. Your goal is to quickly store (write) and find (read) data given the following physical limitations:

- You've been assigned an area called a "cylinder."
- Your cylinder has 255 horizontal lines running through it (tracks) and also 63 vertical lines running through it (sectors).
- Where these lines intersect, a storage unit (block) has been created for you to place files into; every block is the same size (512 bytes).

So, how many storage blocks do you have on your cylinder?

Also in FreeBSD Basics:

[Dividing Your Data](#)

[Monitoring Unix Logins](#)

[Securing BSD Daemons](#)

[Understanding BSD Daemons](#)

[Cracking Passwords to Enhance Security](#)

$$255 * 63 = 16,065$$

If you put one file in each storage block, you can save up to 16,065 files. If you create a table for yourself and number it from 1 to 16,065, you can simply record the name of each file next to a free number as you save the file to the block represented by that number. If you delete a file, you have to remember to remove its name from your table. If you want to move a file, you can look it up in your table, erase it from the old location, and write its name next to its new block number. You would also quickly learn that you didn't have to go to all the trouble of physically moving a file from one storage block to another storage block; it is much easier to simply change the entry for that file in your table.

In its simplest form, this is how all file systems keep track of your files. If this was a Unix file system, that table would be called the inode table.

Unfortunately, simplicity results in lousy hard disk usage. The ability to save files in 512-byte storage units would be great if every file created by users was 512 bytes in size. But, as you know, files vary greatly in size, from just a few bytes to several kilobytes.

Still thinking as a file system, how would you save a file that was 10 bytes in size? If you simply place this file by itself into a storage block, you've wasted 412 bytes of hard disk space. Save enough small files, and you end up wasting a lot of your disk space. What would happen if you saved 16,065 one-byte files? You would use up all your blocks with 16,065 bytes worth of data. Even though there may be several MB of disk space on your cylinder, you have run out of the blocks to place files into. This is called running out of inodes (or inode table entries), and it is not a good thing.

Continuing to think like a filesystem, it would make sense not to devote an entire physical block to one small file. However, you now have to re-think how you're going to organize your table to deal with the fact that there may be more than one file in a block. If you simply start stuffing in as many files as will fit into a 512-byte block, how are you going to keep track of where one file ends and another file begins? What if you remove a 10-byte file and replace it with an 8-byte file -- how will you keep track of that extra two bytes in case you want to stuff in two more 1-byte files? You should be able to see that such a scheme would quickly become unworkable.

Most filesystems use the concept of "fragments." A fragment is a logical division of a block. Each fragment will be assigned an address so it can have an entry in the filesystem table. As a simple example, a filesystem may choose to divide each physical block into four "fragments." This effectively multiplies your number of blocks by four while reducing the block size by four. For example, if you started with 16,065 physical blocks that were each 512 bytes in size, a fragment size of 4 would give you 64,260 logical blocks that were each 128 bytes in size. Each fragment can be treated as a storage block and only store one file, meaning the table now has 64,260 entries, but you still don't have to worry about keeping track of multiple files per logical storage unit.

Now let's look at the other end of the scale. What happens if you need to store files larger than your physical or logical block size? You're obviously going to need to use more than one block to store that file. Pretend you need to save a 1,000-byte file. This will require two physical blocks ($512 * 2$), so you will need to make two entries in your table for this one file. You'll also have to re-think how you are going to make those entries, as order is now important. It's not enough to know that this file lives in, say, blocks 3 and 4; you also need to know that the first 512 bytes of that file lives in block 3, and the remainder of that file lives in block 4.

If you are a filesystem that chose to use fragments, your job is actually harder when you need to save a large file. If you save that same 1,000-byte file with a fragment size of 4, you'll have to make eight entries in your table and ensure that you remember which order to keep those eight entries in.

Up to this point, we've only looked at the considerations for saving or writing files. Another important consideration for a filesystem is its read performance. The whole point of saving files to disk in the first place is so that users can access files when they need to. In order for a user to access a file, the filesystem must find out which block or blocks that file has been stored in, and then copy the contents of those disk blocks to RAM so the user can actually manipulate the data within the file.

by Dru Lavigne

There are several things a filesystem can do to increase read performance. One is to actually increase the logical block size, meaning that several physical blocks are grouped together into one large virtual block. When a user wishes to access a file that is contained on a physical block, the entire virtual block is loaded into RAM. This saves a lot of transferring of individual blocks from disk to RAM and then back to disk. If several blocks have already been pre-loaded into RAM, it is quite likely that all the data that user required was transferred to RAM in one transfer.

Let's see if we can summarize the considerations of a file system:

- There is a finite number of storage blocks, and every storage block has a numbered entry in a table, which in Unix is called the inode table.
- If you run out of inode numbers, you have run out of storage blocks; this means that you can no longer create and store files, regardless of how much disk space is left on your drive.
- If you need to store a lot of small files, you should fragment your block size to create more inode numbers.
- If you wish to increase read performance, you should create a large virtual block size so more blocks are loaded into RAM per disk transfer.

Now, let's see how this applies to FreeBSD. When you use `newfs`, you are formatting your slice with FFS, or the Berkeley Fast File System. Let's take a peek at the manpage for `newfs` to see what the defaults are for this file system:

```
man 8 newfs
```

We're interested in some of the switches; let's start with the `b` switch:

<code>-b</code>	block-size	The block size of the file system, in bytes. It must be a power of 2. The default size is 8192 bytes, and the smallest allowable size is 4096 bytes.
-----------------	------------	--

Notice that this is the block size for the file system; the physical block size is always 512 bytes. 8192 bytes is actually 16 physical blocks, so this switch deals with the virtual block size or how many blocks are loaded into RAM at once. Remember that this is done to increase read performance and is one of the reasons why FFS is "fast."

Now let's look at the `f` switch:

`-f` `frag-size` The fragment size of the file system in bytes. It must be a power of two ranging in value between `blocksize/8` and `blocksize`. The default is 1024 bytes.

FFS uses fragments, but it fragments the virtual block size, not the physical block size. The default fragment size is actually two physical blocks or 1024 bytes. Now the `i` switch:

`-i` number of bytes per inode Specify the density of inodes in the file system. The default is to create an inode for every $(4 * \text{frag-size})$ bytes of data space. If fewer inodes are desired, a larger number should be used; to create more inodes a smaller number should be given. One inode is required for each distinct file, so this value effectively specifies the average file size on the file system.

By default, there is one inode for every 4048 bytes worth of disk space and every file requires its own inode number. Since an inode represents the storage unit for one file, this default assumes an average file size of 4048 bytes.

There are two other switches worth mentioning at this point, as they affect the read/write performance of the FFS filesystem. The first one is the `m` switch:

`-m` free space % The percentage of space reserved from normal users; the minimum free space threshold. The default value used is defined by `MINFREE` from `<ufs/ffs/fs.h>`, currently 8%. See `tunefs(8)` for more details on how to set this option.

All filesystems suffer when you start to get low on available storage blocks. Filesystems like to store files from the same directory in contiguous (next to each other) blocks; this gets harder to do as the number of available blocks decreases and the filesystem has to start searching for empty blocks. The designers of FFS noticed that performance drastically decreases when a filesystem gets around 90% full. When a filesystem reaches the default free space threshold (8%), meaning the filesystem is 92% full, it won't let regular users save any more files. Instead the users will receive an error message and will probably start complaining to the administrator to rectify the situation. The superuser will still be able to save files and use up the remaining blocks, but his time would be better spent in coming up with a plan to save the filesystem before it runs out of storage blocks.

The second switch that deals with free space is the `o` switch:

`-o` optimization preference ("space" or "time") The file system can either be instructed to try to minimize the time spent allocating blocks, or to try to minimize the space fragmentation on the disk. If the value of `minfree` (see above) is less than 8%, the default is to optimize for space; if the value of `minfree` is greater than or equal to 8%, the default is to optimize for time. See `tunefs(8)` for more details on how to set this option.

Most filesystems use an algorithm to determine which blocks should be used to store which files. This switch tells the FFS algorithm to optimize itself for speed or "time." However, when the filesystem meets the `minfree` threshold, finding free blocks or "space" becomes much more important than speed.

From the default parameters, you can see that the FFS has been optimized for speed, while still creating a fair number of inodes to record where files have been stored. For most installations, the default values will be fine; in later articles, we'll look more in-depth on how to determine if the default values are not sufficient for your system and what to do about it.

At this point, you've probably learned more about filesystems than you care to admit. Let's take a break until next week when we take a look at cylinder groups, superblocks, and what type of information is actually recorded in an inode entry.

[Dru Lavigne](#) is a networking instructor at a private technical college in Kingston, ON, and is notorious for seeing how many operating systems she can convince to multi-boot on her test machine.

Read more from [FreeBSD Basics](#).

Discuss this article in the [Operating Systems Forum](#).

Return to the [BSD DevCenter](#).

oreillynet.com Copyright © 2000 O'Reilly & Associates, Inc.