
File System Overview

In this note, 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 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) to ZFS (Sun's Zettabyte File System). 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.

an inode. The name inode is short for **index node**, the historical name given to it by UNIX inventor Ken Thompson [RT74].

Each inode is implicitly referred to by a number (the **i-number**). In vsfs (and many but not all other file systems, as we will see), given an i-number, you should directly be able to calculate where on the disk the corresponding inode is. For example, imagine that the inode region of vsfs was 16KB in size (4 4KB blocks) and consisted of 128 inodes (assume each inode is 128 bytes); further assume that the inode region starts at 4KB (the first block is the superblock). In vsfs, we thus have the following layout for the beginning of the file system partition:

0	4KB	8KB	12KB	16KB	20KB
SuperBlock	InodeBlk0	InodeBlk1	InodeBlk2	InodeBlk3	Data
	inodes 0-31	32-63	64-95	96-128	

Thus, to read inode number 64, the file system would first calculate the offset into the inode region ($64 \cdot \text{sizeof}(\text{inode})$ or 8192, add it to the start address of the inode table on disk (`inode_startAddr = 4KB`), and thus arrive upon the correct address of the desired block of inodes: 12KB or 12288. More generally, the address `iaddr` of the inode block can be calculated as follows:

```
boffset = (i_number * sizeof(inode_t)) / blocksize; // use integer divide
iaddr   = (boffset * blocksize) + inode_startAddr;
```

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* 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 30.1.

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
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

Table 30.1: The ext2 inode

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. Of course, 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 (say 12), and then a single indirect pointer. If a file grows large enough, an indirect block is allocated, and the inode's slot for an

ASIDE: 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. Of course, 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).

indirect pointer is set to point to it. Assuming that a block is 4KB and that each disk pointer is 4b, that adds another 1024 pointers and thus the file can grow to be $(12 + 1024) \cdot 4$ or 4144KB in size (just over 4MB).

Of course, 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 I 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. Many file systems use such an approach, including commonly-used file systems such

as Linux ext2 [P09] and ext3 as well as the original UNIX file system. Other file systems (e.g., SGI XFS) use **extents** instead of simple pointers; see the Aside for details.

30.3 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. Thus, 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 file 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 block 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

where 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 example, the root).

Note that 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 a fair amount of space within.

ASIDE: FREE SPACE MANAGEMENT

There are many ways to manage free space, where bitmaps are just one way. Some early file systems would use **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 uses some form of a **binary tree** to compactly represent which chunks of the disk are free. As with any data structure, different time-space trade-offs are possible.

30.4 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 readily find space for it. Thus, **free space management** is an important aspect of any file system. In vsfs, we have two simple bitmaps for this task.

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. Thus, 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.

30.5 Examples: 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. Let us assume that the file system has been mounted and thus that the

superblock is already in memory; everything else (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.txt`, 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.txt", O_RDONLY)` call, the file system first needs to find the inode for the file `bar.txt` and make sure it is OK that you are opening it (i.e., that you have the correct permissions). To do so, the file system must be able to find the inode. Unfortunately, all the FS has right now is the full pathname. Thus, it must **traverse** the pathname and in doing so locate the inode of the desired file.

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 would next read the block containing the inode of `foo` and then read in its

directory data, finally finding the inode number of `bar.txt`. The final step of `open()`, then, is to read its inode into memory; the FS can then do a final permissions check, allocate a file descriptor for this process in the per-process open-file table, and return it to the user.

To summarize, to simply open a file `/foo/bar.txt`, the file system reads the following blocks from disk:

- the inode of the root directory
- the data of the root directory
- the inode of `foo`
- the data of the `foo` directory
- the inode of the file `bar.txt`

It only gets worse with longer pathnames!

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. The read will further update the open file table for this file descriptor, updating the file pointer such that the next read will read the next block of the file.

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.

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). Thus, each write to a file logically generates two

I/Os: one to the data bitmap to mark the newly-allocated block as used, and one to 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 the directory containing the new file. The total amount of I/O traffic to do so is quite high: 1 write to the inode bitmap, 1 write to the new inode itself, 1 to the data of the directory, and 1 to the directory inode (4 total). If the directory needs to grow to accommodate the new entry, an additional two I/Os (to the data bitmap to allocate the block and to the new block to record the entry) will be needed. All of that work just to create a file!

30.6 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 **fix-sized 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 fix-sized cache would usually be allocated at boot time to be roughly 10% of the total memory of the system. Modern systems integrate virtual memory pages and file system pages into a **unified page cache** [S00]. In this way, memory can be allocated much 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 also 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, 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 5 and 30 seconds, which represents another trade-off. If the system crashes before the updates have been propagated to disk, the updates are lost. However, by buffering writes, performance can be greatly increased, by batching writes together, scheduling them, and avoiding some altogether.

30.7 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 an inode. Directories are just a

specific type of file that store name-to-i-number mappings. And other structures are needed too, e.g., bitmaps 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 notes.

References

[P09] "The Second Extended File System: Internal Layout"
Dave Poirier, 2009
Available: <http://www.nongnu.org/ext2-doc/ext2.html>

[RT74] "The UNIX Time-Sharing System"
M. Ritchie and K. Thompson
CACM, Volume 17:7, pages 365-375, 1974

[S00] "UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD"
Chuck Silvers
FREENIX, 2000