Introduction

This chapter introduces vsfs, a simple file system implementation (the Very Simple File System). This file system is a simplified version of a standard UNIX file system that introduces essential on-disk structures, access methods, and regulations. A simple file system (vsfs) will be used in this chapter to introduce most fundamentals, followed by a series of real-world file system studies to see how they differ in practice.

IMPLEMENTING A SIMPLE FILE SYSTEM

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

Thinking About File Systems

We normally propose thinking about file systems from two separate angles; if you grasp both, you probably understand how a file system works.

The first is the file system data structures. What on-disk structures does the file system use to organize its data and metadata? Simple file systems (like vsfs) use arrays of blocks or other objects, but more complex file systems (like SGI's XFS) use tree-based structures.

The second feature of a file system is access. How does it map process calls like open(), read(), write(), etc. to its structures? Which structures are read during a system call? Who wrote? How well are these steps executed?

You have a decent mental representation of how a file system works if you comprehend its data structures and access techniques. As we begin our initial implementation, try to refine your mental model.

Organization

The vsfs file system data structures are now organized on disk. Simple file systems use only one block size, and that's exactly what we'll do here. Let's choose a standard size of 4 KB.
So, our file system partition looks like a series of 4 KB chunks. A partition of size N 4-KB blocks is ad-dressed from 0 to N 1. Assume we have a 64-block disk:

Consider what we need to store in these blocks to create a file system. Of course, the first thought is user data. In truth, most file system space is (and should be) user data. For simplicity, let us name the data area the final 56 of 64 blocks on the disk.

As we taught last chapter, the file system must keep track of each file. This metadata tracks things like which data blocks (in the data area) make up a file, its size, its owner and access rights, access and modification times, and other such items. File systems use an inode structure to hold this data (more on inodes later).
We'll also need to reserve disk space for inodes. Inode table: An array of on-disk inodes. So, assuming we use 5 of our 64 blocks for inodes (I's in the graphic), our on-disk image now looks like this:

Inodes are typically 128 or 256 bytes in size. A 4-KB block can hold 16 inodes, and our file system above has 80 inodes total. This reflects the maximum number of files in our simple file system created on a modest 64-block partition; however, the same file system built on a larger disk could simply allocate a larger inode table and so accommodate more files.
Our file system currently has data blocks (D) and inodes (I), but more is needed. As you might expect, some means to track whether inodes or data blocks are free or allocated is still required. Any file system must have such allocation mechanisms.
Of course, several approaches exist. For example, we may use a free list that points to the first free block, and so on. Instead, we use a bitmap for both the data region and the inode table (the inode bitmap). Simple structure: each bit indicates whether an object/block is free (0) or in use (1). So, our new on-disk layout, with inode I and data (d) bitmaps:

It seems a bit excessive to use a full 4-KB block for these bitmaps, when we only have 80 inodes and 56 data blocks. For convenience, we utilize a 4-KB block for each of these bitmaps.
The attentive reader (i.e., the reader who is still awake) may have noticed that our extremely rudimentary file system still lacks one block. The superblock, marked by a S in the picture below, gets this. There are 80

inodes and 56 data blocks in this file system, and the inode table starts at block 3. It'll probably also have a magic number to identify the file system type (in this case, vsfs).

The operating system will first read the superblock to initialize different parameters, before attaching the volume to the file-system tree. In this way, the system knows exactly where to look for the required on-disk structures.

File Organization: The Inode

The inode is a key component of almost every file system on disk. An inode is referred to as an index node in UNIX and probably older systems because they were originally placed in an array that was indexed into when accessing an inode.

Each inode is implicitly referred to by an i-number, which we previously referred to as the file's low-level name. Given an i-number, vsfs (and other simple file systems) should be able to find the appropriate inode on disk. Consider the vsfs inode table: 20KB (five 4KB blocks), 80 inodes (assuming each inode is 256 bytes), inode region starting 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, the file system partition starts with the following layout (in closeup view):

Assuming the file system's offset into the inode area is 32 (32 sizeof(inode) or 8192), the file system would then add this to the 12KB start address of the inode table on disk to get the right byte address of the desired block of inodes: 20KB. Remember that drives are not byte addressable, but rather include many 512-byte addressable sectors. So, to get the inode block containing inode 32, the file system would issue a read to sector 201024 (or 40). The inode block's 512 sector address sector can be calculated as follows:

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

It contains information on a file's type (file, directory, etc.), size (blocks allocated to it), protection (who owns the file and who may access it), time (when the file was created, modified, or last accessed), and location (where its data blocks are on disk) (e.g., pointers of some kind). Metadata is any information about a file that isn't pure user data. Figure 40.1 shows an ext2 inode.

One of the most crucial decisions in inode design is where data blocks are located. The inode could have one or more direct pointers (disk addresses) inside it, each pointing to a different file-related disk block. A file larger than the block size multiplied by the number of direct references in the inode, for example, is out of the question.

The Multi-Level Index

To enable larger files, file system designers had to modify inode structures. Indirect pointers are a popular idea. Instead of pointing to a block with user data, it points to a block with pointers to user data. An inode may have 12 direct pointers and 1 indirect pointer. The inode's indirect pointer slot is set to point to an indirect block (from the data-block section of the disk) if the file grows large enough. With 4-KB blocks and 4-byte disk addresses, the file can grow to (12 + 1024) 4K or 4144KB.

Unsurprisingly, you may want to support larger files using this method. To do so, simply add a double indirect pointer to the inode. An indirect block that contains pointers to data blocks is referenced by this pointer. A double indirect block allows files to grow by 1024 1024 or 1-million 4KB blocks, providing files larger than 4GB. We know what you're thinking: the triple indirect pointer.

The multi-level index approach to pointing to file blocks refers to this unbalanced tree. Let's look at a case with 12 direct pointers and a single and double indirect block. With a block size of 4 KB and 4-byte pointers, this structure can handle files up to 4 GB in size (12 + 1024 + 10242). Can you calculate the maximum file size that a triple-indirect block can handle? (Hint: large)

Many file systems, including Linux ext2 and ext3, NetApp's WAFL, and the original UNIX file system, use a multi-level index. Other file systems, such SGI XFS and Linux ext4, use extents instead than simple pointers (they are akin to segments in the discussion of virtual memory).

You may think, why employ such an unbalanced tree? Why not try a new tactic? But as it turns out, numerous academics have examined file systems and their use for decades, and they always come up with the same "truths". Most files are small, for example. If most files are indeed small, it makes sensible to optimize for this circumstance. An inode can directly point to 48 KB of data with 12 direct pointers, but larger files require one (or more) indirect blocks.

The inode is essentially a data structure, and any data structure that stores and queries the relevant information effectively is sufficient. In the event that workloads or technologies change, you should be open to experiment with new designs.

Directory Organization

Directories in vsfs (like in many file systems) are simply lists of (entry name, inode number) pairs. The data block(s) of the directory contain a string and a number for each file or directory. Each string may have a length (assuming variable-sized names).
Assume a directory dir (inode 5) contains three files (foo, bar, and foobar), with inode numbers 12, 13, and 24, respectively. Dir's on-disk data may look like:

In this example, each entry has an inode number, record length (the name's total bytes plus any leftover space), string length (the name's actual length), and name. The dot directory is simply the current directory (in this case, dir), whereas the dot-dot directory is the parent directory (in this case, the root).

Deleting a file (e.g., unlink()) can leave a gap in the directory, therefore it should be marked as well (e.g., with a reserved inode number such as zero). A new entry may reuse an older, larger entry, resulting in greater space.

It's natural to ask where directories are stored. File systems often regard directories as a particular file type. Thus, a directory has an inode (with the type field set to "directory" instead of "regular file"). The inode (and maybe indirect blocks) point to data blocks in the data block section of our simple file system. So our disk structure stays the same.

Also, keep in mind that this is not the only option to store such data. The same as previously. For example, XFS stores directories as B-trees, making file creation faster than systems with simple lists that must be scanned in whole.

Free Space Management

When a new file or directory is allocated, the file system must track which inodes and data blocks are free. All file systems need to manage free space. For this, vsfs provides two simple bitmaps.

For example, when creating a file, we must allocate an inode. The file system must next search the bitmap for a free inode, allocate it to the file, mark it as utilized (with a 1), and eventually update the on-disk bitmap with the right information. When a data block is allocated, comparable steps occur.

Other factors may be taken into account when allocating data blocks for a new file. When a new file is created, some Linux file systems, such ext2 and ext3, look for a series of free blocks (say 8) to allocate to the new file, ensuring that a section of the file is contiguous on the disk and therefore boosting efficiency. Pre-allocating space for data blocks is thus a typical approach.

Reading A File From Disk

Now that we know how files and directories are saved on disk, we can follow the flow of operations when reading or writing files. Pay attention to what happens on this access path to gain a better grasp of how a file system operates.

Assume that the file system has been mounted and that the superblock is already in memory. The inodes and directories are still there.

Reading A File From Disk

INSERT IMAGE

Assume you wish to open a file (e.g., /foo/bar), read it, and then close it. Assume the file is 12KB in size for this example (i.e., 3 blocks).

Opening a file requires the file system to locate the file's inode, which contains some basic information about the file (permissions information, file size, etc.). The file system needs to find the inode, but it only has the complete pathname. The pathname must be traversed to find the desired inode.

All traversals start at the file system's root, the directory /. The FS will so read the root directory's inode first. But where is it? That's the i-number we need. The i-number of a file or directory is usually found in its parent directory (by definition). So the root inode number must be "well known" to the FS when mounting the file system. Most UNIX file systems have a root inode of 2. To begin, the FS reads in the block containing inode number 2. (the first inode block).

Once the inode is read, the FS can check inside it for references to data blocks containing the root directory's contents. The FS will then utilize these on-disk pointers to search the directory for a foo entry. It finds the entry for foo by reading one or more directory data blocks; it also finds the inode number for foo (say 44) which it needs next.

Iterate over the pathname until you find the desired inode. In this case, the FS reads the block containing foo's inode, then its directory data, and finally bar's inode number. The FS 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.
It can then use the read() system function to read from the file. It will so read in the first block of the file, checking the inode to locate its location, and updating the inode with a new last-accessed time. The read will update the in-memory open file table for this file descriptor, reading the second

file block, etc.
The file will eventually close. The file descriptor should be deallocated, but for now, that is all the FS needs to do. No disk I/Os occur.

Illustration 40.3 (page 11) depicts the full process; time moves lower in the figure. In the illustration, the open causes several reads to identify the file's inode. The file system must then consult the inode, read the block, and update the inode's last-accessed-time field with a write. Spend some time understanding.

Also, the open generates I/O proportional to the pathname's length. We must read the inode and data of each additional directory in the path. In this case, we only need to read one block, but in a huge directory, we may need to read numerous data blocks to discover the necessary entry. Writing a file (and especially generating a new one) is much more difficult than reading one.

Writing A File To Disk

Writing to a file works similarly. First, open the file (as above). The application can then use write() to update the file's content. The file is then closed.

A block may be allocated when writing to a file (unless the block is being overwritten, for example). Copying a new file requires not just writing data to disk but also allocating a block and updating other disk structures (e.g., the data bitmap and inode). Each write to a file 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 write the inode (which is updated with the new block's location), and finally one to write the actual block itself (which is marked as used).

INSERT IMAGE

Even a simple and routine process like file creation generates a lot of write traffic. To create a file, the file system must first allocate an inode and then space within the new file's directory. One read (to discover a free inode), one write (to allocate it), one write (to initialize it) to the new inode itself, one read and write (to update the directory inode) to the data of the directory (to link the high-level name of the file to its inode number). Also, if the directory must expand to accommodate the new entry, more I/Os will be required (data bitmap and new directory block). All for a file!

Consider the case of the file /foo/bar, which receives three blocks. Figure 40.4 (page 13) depicts the open() (file creation) and three 4KB writes.

The figure groups reads and writes to the disk by the system call that caused them, and the figure shows them in rough order from top to bottom. Create the file takes a lot of work: In this scenario, 10 I/Os to walk the pathname and create the file. One pair to read and update the inode, another pair to read and update the data bitmap, and lastly the write of the data itself takes five I/Os. How can a file system do all of this efficiently?

HOW TO LOWER FILE SYSTEM I/O COSTS

Simply opening, reading, or writing a file results in a massive number of I/O operations distributed throughout the disk.
What can a file system do to reduce I/O costs?

Caching and Buffering

As shown above, reading and writing files requires several I/Os to the (slow) disk. Most file systems employ system memory (DRAM) to cache critical blocks to avoid a major performance issue.

Consider the open example: Without caching, every file open would necessitate two reads for every level (one to read the inode of the directory in question, and at least one to read its data). In this case, the file system would have to read hundreds of times simply to open the file!

This led to a fixed-size cache for popular blocks. Like in virtual memory, LRU and its variants pick which blocks to cache. An initial fixed-size cache of around 10% of total RAM is allocated upon boot.

But what if the file system doesn't use 10% of memory at any one time? With the fixed-size technique mentioned above, unused file cache pages cannot be reused and go to waste.

Modern systems, however, use dynamic partitioning. Modern operating systems combine virtual memory and file system pages into a single page cache. In this approach, memory can be shared between virtual memory and file system, depending on which needs it more.

Assume the file open example with caching now. When opening a file (or a directory), the first open may require a lot of I/O to read the directory's inode and data, but subsequent opens will primarily hit the cache and require no I/O.

Consider the impact of caching on writes. Unlike read I/O, which can be avoided with a big cache, write traffic must go to disk to be persistent. So a cache doesn't act as a write filter like it does for reads. However, write buffering (as it is frequently termed) provides many performance advantages. For example, if an inode bitmap is updated when one file is created and then updated again seconds later when another file is created, the file system saves an I/O by waiting the write after the initial update. (2) Buffering writing in memory allows the system to arrange subsequent I/Os, increasing performance. Finally, delaying some writes completely prevents them, such as when an application creates a file and subsequently deletes it. Laziness (in writing blocks to disk) is a virtue here.

Since most modern file systems buffer writes in memory for between five and thirty seconds, there is another trade-off: if the system fails before the updates have been written to disk, the modifications are lost.

Some apps (like databases) don’t like this trade-off. Because write buffering is a concern, they either force writes to disk using fsync(), use direct I/O interfaces that bypass the cache, or use a raw disk interface that completely bypasses it2. While most applications accept the file system’s trade-offs, there are enough controls to make the system do what you want it to do.

Summary

We've seen the basic tools for creating a file system. Each file requires metadata, which is kept in an inode structure. Directory files store name-inode-number mappings. File systems, for example, utilize a bitmap to track which inodes or data blocks are free or allocated.

These file systems use this freedom to optimize various aspects of their architecture. Clearly, we have left many policy options unexplored. For example, where should a new file be created on disk? This and other policies will be covered in later sections.