39

File System Implementation

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 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?

39.1 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 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.
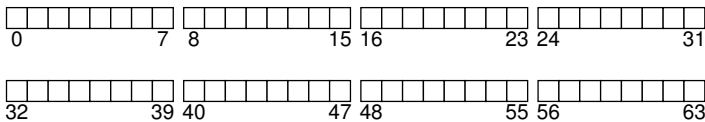
39.2 Overall Organization

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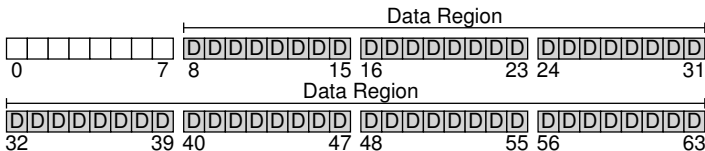
Thus, our view of the disk partition where we're building our file system is simple: a series of blocks, each of size 4 KB. The blocks are addressed from 0 to $N - 1$, in a partition of size N 4-KB blocks. Assume we have a really small disk, with just 64 blocks:

```
┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
0               7 8             15 16            23 24            31

┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
└─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘ └─┴─┴─┴─┴─┴─┴─┴─┘
32             39 40            47 48            55 56            63
```
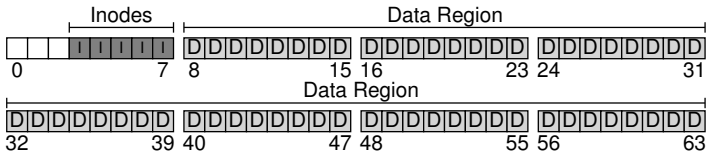
Let's now think about what we need to store in these blocks to
build a file system. Of course, the first thing that comes to mind is
user data. In fact, most of the space in any file system is (and should
be) user data. Let's call the region of the disk we use for user data
the **data region**, and, again for simplicity, reserve a fixed portion of
the disk for these blocks, say the last 56 of 64 blocks on the disk:

```
                          Data Region
┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
└─┴─┴─┴─┴─┴─┴─┴─┘ │D│D│D│D│D│D│D│D│ │D│D│D│D│D│D│D│D│ │D│D│D│D│D│D│D│D│
0               7 8             15 16            23 24            31
                          Data Region
┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐ ┌─┬─┬─┬─┬─┬─┬─┬─┐
│D│D│D│D│D│D│D│D│ │D│D│D│D│D│D│D│D│ │D│D│D│D│D│D│D│D│ │D│D│D│D│D│D│D│D│
32             39 40            47 48            55 56            63
```

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 system 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):
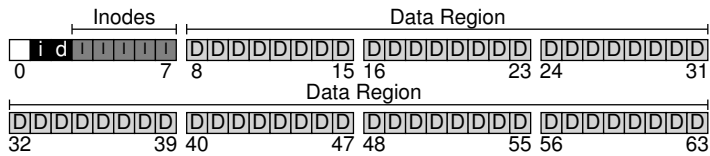

We should note here that inodes are typically not that big, for example 128 or 256 bytes. Assuming 256 bytes per inode, a 4-KB block can hold 16 inodes, and our file system above contains 80 total inodes. In our simple file system, built on a tiny 64-block partition, this number represents the maximum 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.
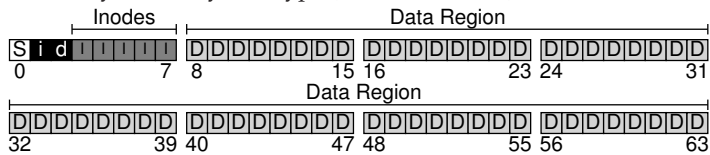
Many allocation-tracking methods are possible, of course. For example, we could use a **free list** that points to the first free block, which then points to the next free block, and so forth. We instead choose a simple and popular structure known as a **bitmap**, one for the data region (the **data 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

Inodes | Data Region

```
| i | d | | | | | | | |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D|
0               7 8              15 16             23 24             31
```
Data Region
```
|D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D|
32            39 40             47 48             55 56             63
```

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).

Inodes | Data Region

```
|S| i | d | | | | | | | |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D|
0               7 8              15 16             23 24             31
```
Data Region
```
|D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D| |D|D|D|D|D|D|D|D|
32            39 40             47 48             55 56             63
```

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.
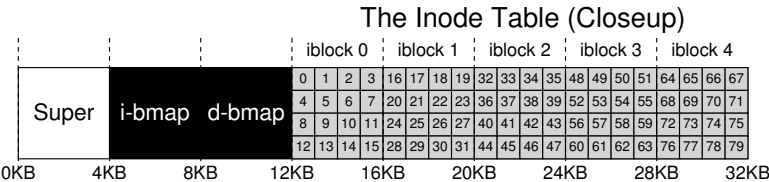
39.3 File Organization: The Inode

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 by UNIX inventor Ken Thompson [RT74], used because these nodes were originally arranged in an array, and the array *indexed* into when accessing a particular inode.

Each inode is implicitly referred to by a number (called the **i-number**), 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 in-

> 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.).

ode 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 ad-
dress 4KB, the data bitmap at 8KB, and thus the inode table comes
right after). In vsfs, we thus have the following layout for the begin-
ning of the file system partition (in closeup view):

The Inode Table (Closeup)

To read inode number 33, the file system would first calculate the
offset into the inode region ($32 \cdot sizeof(inode)$ or 8192), add it to the
start address of the inode table on disk ($\texttt{inodeStartAddr} = 12KB$),
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 $\texttt{iaddr}$ 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* 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 below in Figure 39.1.

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.

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 39.1: The ext2 inode

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).

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 (from the data-block region of the disk), and the inode's slot for an indirect pointer is set to point to it. Assuming that a block is 4KB and that each disk pointer is 4 bytes, that adds another 1024 pointers; the file can grow to be $(12 + 1024) \cdot 4K$ or 4144KB.

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**.

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

Table 39.2: File System Measurement Summary

Overall, this imbalanced tree is referred to as the **multi-level index** approach to pointing to file blocks. Let's examine an example with 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., $(12 + 1024 + 1024^2) \times 4 \ KB$). 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 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; Table 39.2 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.

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? Turns out that 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.

39.4 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. Of-
ten, 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 binary tree form, making file create operations (which have to en-
sure that a file name has not been used before creating it) faster than
systems with simple lists that must be scanned in their entirety.

39.5 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 **binary 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 on the disk and contiguous, thus improving performance. Such a **pre-allocation** policy is thus a commonly-used heuristic when allocating space for data blocks.

39.6 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, 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 would next read the block containing the inode of foo and then read in its directory data, finally finding the inode number of bar. 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.

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

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

Table 39.3: File Read Timeline (Time Increasing Downward)

file offset such that the next read will read the second file block, etc.

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 39.3 (time increases in the downward direction). As you can see from 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.

You might also notice 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 what we are looking for. 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.

	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[1]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
			write							
write()	read				read					
	write							write		
					write					
write()	read				read					
	write								write	
					write					
write()	read				read					
	write									write
					write					

Table 39.4: File Creation Timeline (Time Increasing Downward)

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 three 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), 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: 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 (to the data bitmap to allocate the block and to the new block to record the entry) will be needed too. All of 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 39.4 shows what happens during the open() (which creates the file), as well as during each of three 4-KB calls to write().

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 obvious high costs of doing so many I/Os?

39.7 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 **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 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; for this reason, more paranoid applications will force writes to disk explicitly (e.g., by calling `fsync()`). However, by buffering writes, performance can be greatly increased, by batching writes together, scheduling them, and avoiding some altogether.

39.8   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 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

[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. Satya-narayanan, 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.

The possible operations are:

- mkdir() - creates a new directory

- creat() - creates a new (empty) file

- open(), write(), close() - appends a block to a file

- link() - creates a hard link to a file

- unlink() - unlinks a file (removing it if linkcnt==0)

To understand how this homework functions, you must first understand how the on-disk state of this file system is represented. The state of the file system is shown by printing the contents of four different data structures:

```
inode bitmap - indicates which inodes are allocated
inodes       - table of inodes and their contents
data bitmap  - indicates which data blocks are allocated
data         - indicates contents of data blocks
```

The bitmaps should be fairly straightforward to understand, with a "1" indicating that the corresponding inode or data block is allocated, and a "0" indicating said inode or data block is free.

The inodes each have three fields: the first field indicates the type of file (e.g., `f` for a regular file, `d` for a directory); the second indicates which data block belongs to a file (here, files can only be empty, which would have the address of the data block set to -1, or one block in size, which would have a non-negative address); the third shows the reference count for the file or directory. For example, the following inode is a regular file, which is empty (address field set to -1), and has just one link in the file system:

```
[f a:-1 r:1]
```

If the same file had a block allocated to it (say block 10), it would be shown as follows:

```
[f a:10 r:1]
```

If someone then created a hard link to this inode, it would then become:

```
[f a:10 r:2]
```

Finally, data blocks can either retain user data or directory data. If filled with directory data, each entry within the block is of the form (name, inumber), where "name" is the name of the file or directory, and "inumber" is the inode number of the file. Thus, an empty root directory looks like this, assuming the root inode is 0:

```
[(.,0) (..,0)]
```

If we add a single file "f" to the root directory, which has been allocated inode number 1, the root directory contents would then become:

```
[(.,0) (..,0) (f,1)]
```

If a data block contains user data, it is shown as just a single character within the block, e.g., [h]. If it is empty and unallocated, just a pair of empty brackets ([]) are shown.

An entire file system is thus depicted as follows:

```
inode bitmap  11110000
inodes        [d a:0 r:6] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
data bitmap   11100000
data          [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] ...
```

This file system has eight inodes and eight data blocks. The root directory contains three entries (other than "." and ".."), to "y", "z", and "f". By looking up inode 1, we can see that "y" is a regular file (type f), with a single data block allocated to it (address 1). In that data block 1 are the contents of the file "y": namely, "u". We can also see that "z" is an empty regular file (address field set to -1), and that "f" (inode number 3) is a directory, also empty. You can also see from the bitmaps that the first four inode bitmap entries are marked as allocated, as well as the first three data bitmap entries.

The simulator can be run with the following flags:

```
prompt> vsfs.py -h
Usage: vsfs.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -i NUMINODES, --numInodes=NUMINODES
                        number of inodes in file system
  -d NUMDATA, --numData=NUMDATA
                        number of data blocks in file system
  -n NUMREQUESTS, --numRequests=NUMREQUESTS
                        number of requests to simulate
  -r, --reverse         instead of printing state, print ops
  -p, --printFinal      print the final set of files/dirs
  -c, --compute         compute answers for me
```

A typical usage would simply specify a random seed (to generate a different problem), and the number of requests to simulate. In this default mode, the simulator prints out the state of the file system at each step, and asks you which operation must have taken place to take the file system from one state to another. For example:

```
prompt> vsfs.py -n 6 -s 16
...
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
inodes        [d a:0 r:3] [f a:-1 r:1] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0) (y,1)] [] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
inodes        [d a:0 r:3] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
```

```
inodes        [d a:0 r:4] [f a:1 r:2] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (m,1)] [u] [] [] [] [] [] []

Which operation took place?

inode bitmap  11000000
inodes        [d a:0 r:4] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

Which operation took place?

inode bitmap  11100000
inodes        [d a:0 r:5] [f a:1 r:1] [f a:-1 r:1] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (z,2)] [u] [] [] [] [] [] []

Which operation took place?

inode bitmap  11110000
inodes        [d a:0 r:6] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
data bitmap   11100000
data          [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] ...
```

When run in this mode, the simulator just shows a series of states, and asks what operations caused these transitions to occur. Running with the −c flag shows us the answers. Specifically, file "/y" was created, a single block appended to it, a hard link from "/m" to "/y" created, "/m" removed via a call to unlink, the file "/z" created, and the directory "/f" created:

```
prompt> vsfs.py −n 6 −s 16 −c
...
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []

creat("/y");

inode bitmap  11000000
inodes        [d a:0 r:3] [f a:-1 r:1] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0) (y,1)] [] [] [] [] [] [] []
```

```
fd=open("/y", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

inode bitmap  11000000
inodes        [d a:0 r:3] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

link("/y", "/m");

inode bitmap  11000000
inodes        [d a:0 r:4] [f a:1 r:2] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (m,1)] [u] [] [] [] [] [] []

unlink("/m")

inode bitmap  11000000
inodes        [d a:0 r:4] [f a:1 r:1] [] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1)] [u] [] [] [] [] [] []

creat("/z");

inode bitmap  11100000
inodes        [d a:0 r:5] [f a:1 r:1] [f a:-1 r:1] [] [] [] [] []
data bitmap   11000000
data          [(.,0) (..,0) (y,1) (z,2)] [u] [] [] [] [] [] []

mkdir("/f");

inode bitmap  11110000
inodes        [d a:0 r:6] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] ...
data bitmap   11100000
data          [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)] [] ...
```

You can also run the simulator in "reverse" mode (with the "-r" flag), printing the operations instead of the states to see if you can predict the state changes from the given operations:

```
prompt> ./vsfs.py -n 6 -s 16 -r
Initial state

inode bitmap  10000000
inodes        [d a:0 r:2] [] [] [] [] [] [] []
data bitmap   10000000
data          [(.,0) (..,0)] [] [] [] [] [] [] []

creat("/y");
```

```
   State of file system (inode bitmap, inodes, data bitmap, data)?

fd=open("/y", O_WRONLY|O_APPEND); write(fd, buf, BLOCKSIZE); close(fd);

   State of file system (inode bitmap, inodes, data bitmap, data)?

link("/y", "/m");

   State of file system (inode bitmap, inodes, data bitmap, data)?

unlink("/m")

   State of file system (inode bitmap, inodes, data bitmap, data)?

creat("/z");

   State of file system (inode bitmap, inodes, data bitmap, data)?

mkdir("/f");

   State of file system (inode bitmap, inodes, data bitmap, data)?
```

A few other flags control various aspects of the simulation, including the number of inodes ("-i"), the number of data blocks ("-d"), and whether to print the final list of all directories and files in the file system ("-p").

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?