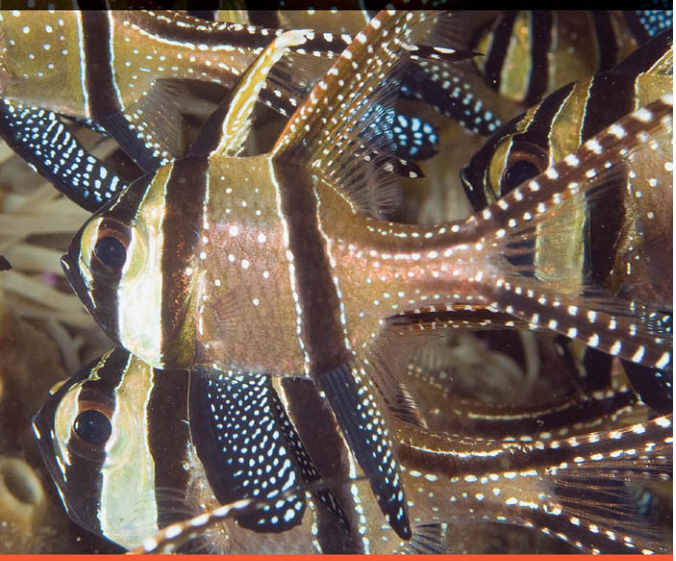
OPERATING SYSTEMS IN DEPTH



THOMAS W. DOEPPNER

File Systems

CHAPTER 6

The purpose of a file system is to provide easy-to-use permanent storage with modest functionality. Unlike database systems with sophisticated search and access capabilities, file systems typically present files as simple, unstructured collections of bytes and provide tree-structured naming. We use files to hold the programs we run, the source code of programs we're developing, documents we're reading or composing, email messages we've sent and received, data we've collected from an experiment, and the like.

There are few aspects of operating systems that aren't affected by file systems. The performance of the file system is a major factor in the performance of the entire system — most operations performed on computers involve files. How well a computer survives crashes is completely dependent on its file system's ability to handle them. Much of computer security is the security of files.

Thus there are four top criteria for file systems:

- *Easy access*: files should be easy to use. This means that the abstraction of a file presented by the file system should be simple, understandable, and useful.
- *High performance*: access to files should be quick and economical. The file-system implementation should let us exploit the speed of the underlying hardware to its fullest, yet have no undue wastage of space.
- *Permanence*: file systems should be dependable. Data stored in them should be accessible as long as they are needed. System crashes and hardware problems should not cause data loss.
- *Security*: data stored in file systems should be subject to strict access controls. (We cover security in Chapter 8.)

We start by looking at a rather ancient Unix file system, S5FS, which, in a simple, elegant fashion, provides all the basic functionality needed by application programs and the virtual-memory system to store and retrieve data. Files can be accessed with equal ease both sequentially and randomly.

The reasons why S5FS is not the last word in file systems are what make file systems interesting. Disks do not provide uniform access times the way primary storage does — much care is required to get top performance from them. After a crash, whatever damage was done to on-disk data structures must be repaired for the system to return to normal operation. S5FS's elegance and simplicity don't take any of this into account, but it gives us a starting point for discussing more recent file systems that do.

We tackle performance first, starting with techniques used in a somewhat later Unix file system known, not too surprisingly, as the "Fast File System" (FFS). We look at later improvements to this system as well as techniques used in other systems such as Microsoft's NTFS. Finally, we cover an interesting approach known as log-structured file systems that first appeared in a research system but have been adapted for more recent commercial systems from Network Appliance (WAFL) and Sun (ZFS).

We then cover approaches for making file systems resilient to crashes. These range from insuring that what's on disk is always consistent, just in case there's a crash, to providing enough extra information on disk to make possible post-crash recovery.

Among S5FS's shortcomings were its directories, which didn't actually provide all the functionality needed by applications — component names were limited to eight characters. Removing this limitation exacerbated performance problems; we examine these problems and see how other systems have dealt with them using hashing and balanced-tree techniques.

Next we take on the use of multiple disk drives, not only to provide additional space, but also to improve performance and failure tolerance. In particular, we look at an approach known as RAID: redundant array of inexpensive disks.

We then look at a relatively new alternative to disk technology — flash memory. Its characteristics have ramifications for the file systems that use it.

We finish the chapter by looking at additional details of the file systems we've used as examples: FFS, NTFS, WAFL, and ZFS.

6.1 THE BASICS OF FILE SYSTEMS

6.1.1 UNIX'S S5FS

We start with a file system that became obsolete well over twenty years ago — the file system of sixth-edition and some later Unix systems known as the System 5 File System (S5FS). Because of its simplicity and elegance, it is suitable for implementation in student operating-system projects. As a file system for real operating systems, however, it has a number of deficiencies and

does poorly on all but the first of the criteria listed above. We use it here as a starting point to discuss how modern file-systems handle these concerns.

The Unix file abstraction is simple — a revolutionary feature in Unix's early days. Files are arrays of bytes. User applications need not know how files are physically represented on disks. Other file systems of the time forced the user to be aware of the *record size* — the amount of data transferred to or from the application in each file request — as well as the *block size* — the unit by which data is transferred to or from the disk. Unix just had bytes. Applications read or wrote as many bytes as necessary. It was up to the file system to implement such requests efficiently using the available disk storage. Rather than making programs allocate space for files before using them, Unix files grow implicitly: writing beyond the current end of a file makes it bigger. Files are named by their paths in a single, system-wide directory hierarchy.

The architecture of the underlying storage medium is, of course, pretty important in file-system design. We assume here that it's a disk organized as a collection of sectors, each of the same size — 512 bytes is typical. As detailed in Section 6.1.2 below, disks are accessed by moving disk heads to the appropriate cylinder and then waiting for the disk to rotate until the desired disk sector is under one of the heads. Thus the time required to access a sector depends on the distance of the current position of the disk heads from the desired sector. One of the things that make S5FS so simple is that it does not take this distance into account when allocating space for files. It considers a disk to be a sequence of sectors each of which can be accessed in the same amount of time as all the others; the only optimization done is to minimize the number of disk accesses. No attempt is made to minimize the time spent waiting for disk heads to be properly positioned (the *seek time*) or the time waiting for the desired sector to be under the disk head (the *rotational latency*).

This simplifies things a lot. All sectors are equal; thus we can think of a disk as a large array. Accessing this array is a bit expensive compared to accessing primary storage, but the design of on-disk data structures needn't differ substantially from the data structures in primary storage. (This, of course, is a simplification used in S5FS; it doesn't apply to all file systems!)

Figure 6.1 shows the basic format of a file system on disk.[1] The first disk block contains a *boot block*; this has nothing to do with the file system, but contains the first-level boot program that reads the operating system's binary image into primary memory from the file system. The next block, the *superblock*, describes the layout of the rest of the file system and contains the heads of the free lists. Following this are two large areas. The first is the *i-list*, an array of index nodes (*inodes*) each of which, if in use, represents a file. Second is the *data region*, which contains the disk blocks holding or referring to file contents.

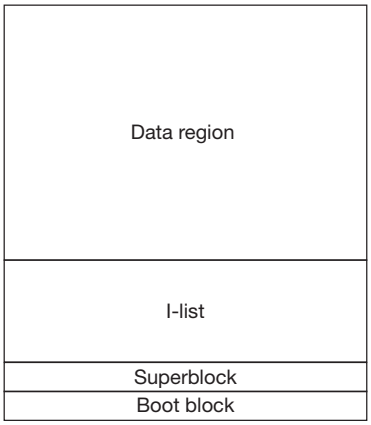


FIGURE 6.1 S5FS layout.

[1] Note that one physical disk is often partitioned to hold multiple file-system instances. Thus what is shown in Figure 6.1 is a single file-system instance within a region of a partitioned disk.

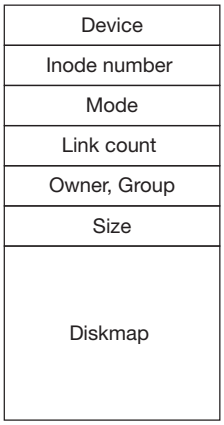


FIGURE 6.2 S5FS inode.

Each file is described by an inode; thus a file is referred to internally by its inode, which is done via the inode's index in the i-list. As we saw in Chapter 1, directories are otherwise normal files containing pairs of directory-component names and inode numbers. Thus following a path in the directory hierarchy involves reading the contents of a directory file (contained in sectors taken from the data region) to look up the next component of the path. Associated with that component in the file is an inode number that is used to identify the inode of the file containing the next directory in the path.

The layout of an inode is shown in Figure 6.2. What's important to us at the moment is the disk map (Figure 6.3), which refers to the disk blocks containing the file's data. The disk map maps logical blocks numbered relative to the beginning of a file into physical blocks numbered relative to the beginning of the file system. Each block is 1024 (1 K) bytes long. (It was 512 bytes long in the original Unix file system.) The data structure allows fast access when a file is accessed sequentially and, with the help of caching, reasonably fast access when the file is used for paging and other "random" access.

The disk map contains 13 pointers to disk blocks. The first ten of these pointers point to the first 10 blocks of the file, so that the first 10 KB of a file are accessed directly. If the file is larger than 10 KB, then pointer number 10 points to a disk block called an *indirect block*. This block contains up to 256 (4-byte) pointers to *data blocks* (i.e., 256 KB of data). If the file is bigger than this (256 KB + 10 KB = 266 KB), then pointer number 11 points to a *double indirect block* containing 256 pointers to indirect blocks, each of which contains 256 pointers to data blocks (64 MB of data). If the file is bigger than this (64 MB + 256 KB + 10 KB), then pointer number 12 points to a *triple indirect block* containing up to 256 pointers to double indirect blocks, each of which contains up to 256 pointers pointing to single indirect blocks, each of which contains

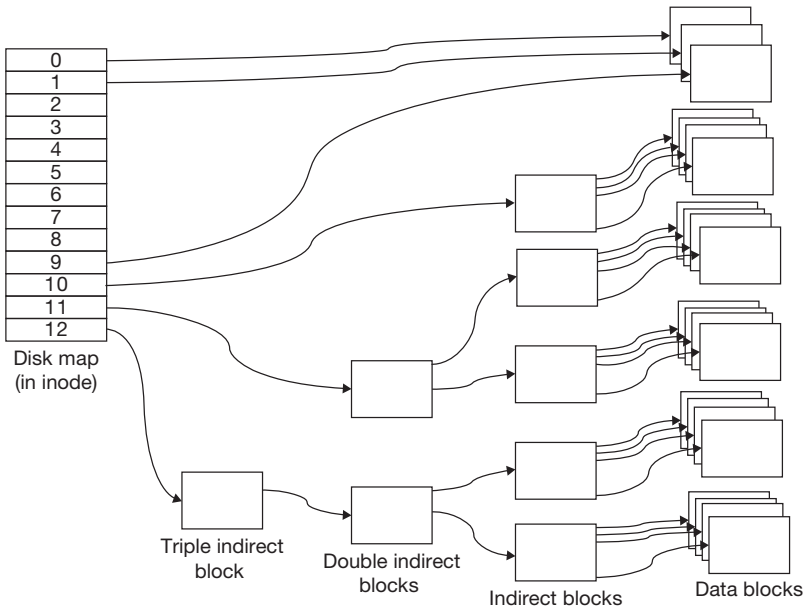


FIGURE 6.3 S5FS disk map. Each of the indirect blocks (including double and triple indirect blocks) contains up to 256 pointers.

up to 256 pointers pointing to data blocks (potentially 16 GB, although the real limit is 2 GB, since the file size, a signed number of bytes, must fit in a 32-bit word).

Sequential access to a file with this scheme is fairly efficient in terms of the number of disk accesses. When a file is opened, its inode is fetched from disk and remains in primary storage until after the file is closed. Thus the addresses of the first ten blocks of each file are available (in the inode's disk map) without further disk I/O.

Accessing blocks of a file beyond the first ten requires additional disk accesses to fetch their addresses, but only one disk access per 256 data blocks is required for the portion of the file mapped by indirect blocks. Beyond that, additional accesses are required for double and triple indirect blocks, but only one access of double indirect blocks per 2^{16} data blocks is required and only one access of a triple indirect block per 2^{24} data blocks is required.

For random access, however, the overhead can be a lot greater. Accessing a block might require fetching three blocks (triple, double, and single indirect) to obtain its address. While this is possible with truly random access, more typically a program fetches data blocks sequentially starting at some specific file location. Thus additional fetches are required to obtain the indirect blocks necessary to get started, but, since these blocks are cached, the addresses of subsequent blocks are available without additional disk I/O.

One interesting attribute of the disk-map data structure is that it allows the efficient representation of sparse files, i.e., files whose content is mainly zeros. Consider, for example, creating an empty file and then writing one byte at location 2,000,000,000. Only four disk blocks are allocated to represent this file: a triple indirect block, a double indirect block, a single indirect block, and a data block. All pointers in the disk map, except the last one, are zero. Furthermore, all bytes up to the last one read as zero. This is because a zero pointer is treated as if it points to a block containing all zeros: a zero pointer to an indirect block is treated as if it pointed to an indirect block filled with zero pointers, each of which is treated as if it pointed to a data block filled with zeros. However, one must be careful about copying such a file, since doing so naively creates a file for which disk blocks are allocated for all the zeros!

The issues involved in organizing free storage on disk are similar to those in organizing the blocks of individual files: we want to minimize the number of disk accesses required to manage free storage. (We'll see soon that there are other concerns as well, but minimizing disk accesses was the primary one in S5FS.) Ideally, whenever a free disk block is required for a file, its address should be available without a disk access. Similarly, freeing a block from a file should not require disk I/O. We can't achieve these ideals, of course, but we can do a lot better than a naive implementation in which a disk access is required for each block allocation and liberation.

Free disk blocks are represented as a linked list (see Figure 6.4); the superblock, which is kept in primary storage while a file system is being used, contains the addresses of up to 100 free disk blocks. The last of these disk blocks contains 100 pointers to additional free disk blocks. The last of these pointers points to another block containing up to n free disk blocks, etc., until all free disk blocks are represented.

Thus most requests for a free block can be satisfied by merely getting an address from the superblock. When the last block reference in the superblock is consumed, however, a disk read must be done to fetch the addresses of up to 100 additional free disk blocks. When a block from a file is freed, the disk block's address is simply added to the list of free blocks in the superblock. If this list is full, then its contents are written out to disk in the block being freed and are replaced with the address of this block.

Inodes are handled differently. Rather than keep track of free inodes in a linked list, they're simply marked free or not free on disk, with no further on-disk organization (see Figure 6.5). The superblock, however, contains a cache of the indices of free inodes (recall that inodes are organized as an array on disk — the i-list). Thus when the system creates a file and therefore must allocate an inode, the index of a free inode is taken from the cache and the on-disk copy of

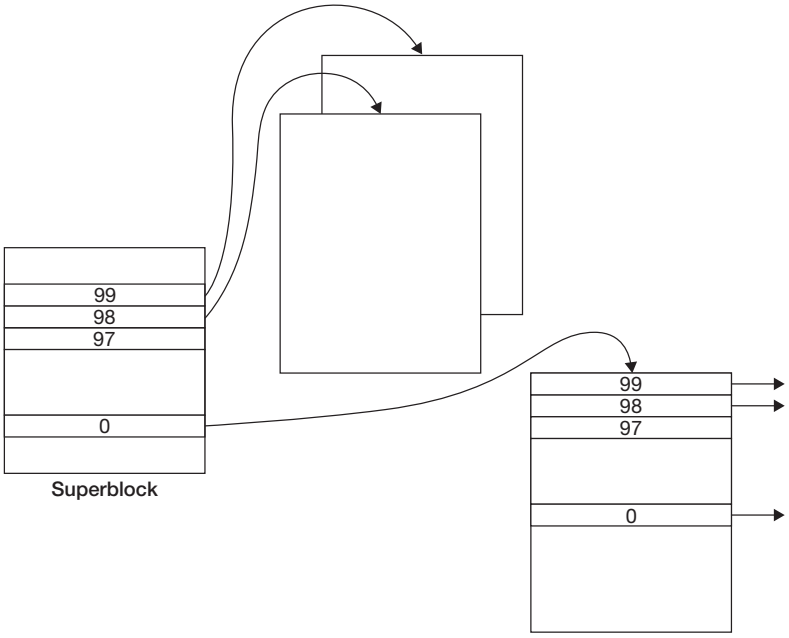


FIGURE 6.4 S5FS free list.

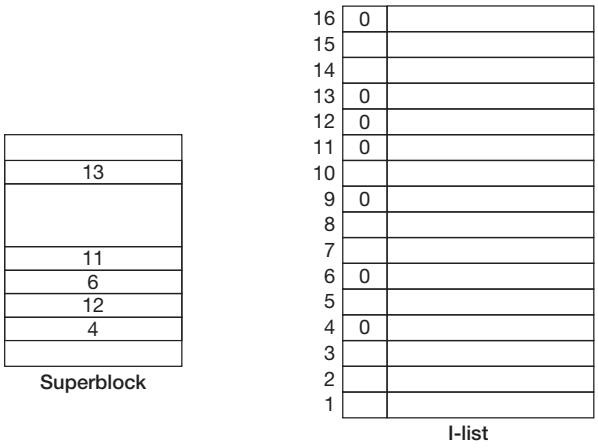


FIGURE 6.5 S5FS free inode list.

the inode is marked as allocated. If the cache is empty, then the i-list is scanned for sufficient free inodes to refill it. To aid this scan, the cache contains the index of the first free inode in the i-list. Freeing an inode involves simply marking the on-disk inode as free and adding its index to the cache, if there's room.

Why are inodes handled this way? In particular, why use a technique that requires a disk write every time an inode is allocated or freed? There are two reasons. First, inodes are allocated and freed much less often than data blocks, so there's less need for a relatively complex technique

Component name	Inode number

Directory entry

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

FIGURE 6.6 S5FS directory.

to minimize I/O. The second reason, as discussed further in Section 6.2 below, has to do with crash tolerance. If the system crashes, information that's been cached in primary storage but is not yet on disk is lost. Though this is definitely a problem for data blocks and the various indirect blocks, it has especially severe consequences for inodes. Thus leaving changes to inode allocation information uncached and caching only a list of some free ones ensures that nothing about the allocated/unallocated state of inodes is lost in a crash.

S5FS directories are implemented exactly as discussed in Section 1.3.5.5. They are otherwise ordinary files whose inodes are marked as being directories. Their contents are represented just like those of other files, though they have a special interpretation: a sequence of component-name/inode-number pairs (see Figure 6.6). The component-name is in a fixed-width field 14 characters long. Thus searching a directory is a simple matter of reading the file block by block, scanning the component-name field for a match and then returning the associated inode number. That the component-name field is fixed-width improves the efficiency of the search, though it definitely limits the possible file names.

6.1.2 DISK ARCHITECTURE

To understand file-system performance, let's first take a careful look at how disk drives work. The typical disk drive (see Figure 6.7) consists of a number of platters, each with one or two recording surfaces (on top and bottom). Each surface is divided into a number of concentric tracks; each track is divided into a number of sectors. All the sectors are the same size. Early disk drives, keeping things simple, had the same number of sectors in each track. This was, of course, a waste of space in outer tracks. Modern disk drives put more sectors in outer tracks than inner ones, thus keeping the bit density roughly constant. The data stored on disks is read and written using a set of read/write heads, one head per surface. The heads are connected to the ends of arms that move in tandem across the surfaces. Only one head is active at a time — the desired one must be selected explicitly. The set of tracks selected by the disk heads at any one moment is called a cylinder.

The data in a particular disk sector can be read or written only when the sector moves under one of the read/write heads. Thus the steps to perform a read or write are:

1. Position the read/write heads over the correct cylinder. The time required for this positioning is known as the *seek time*. The time required to select the head that's positioned over the desired track is assumed to be zero.

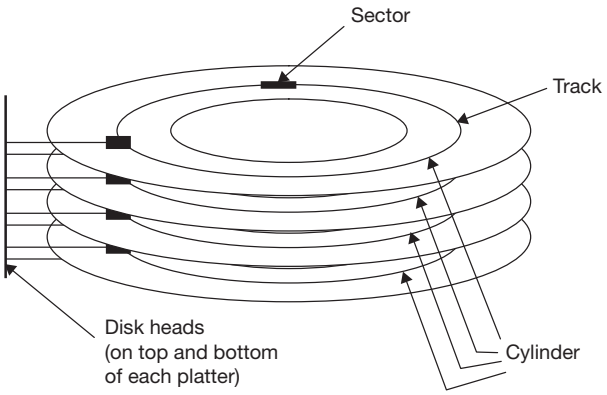


FIGURE 6.7 Disk architecture.

2. Rotate the disk platter until the desired sector is underneath the read/write head. The time required for this is known as the *rotational latency*.
3. Rotate the disk platter further so the head can read or write the entire sector, transferring data between it and the computer's memory. The time required for this is known as the *transfer time*.

The seek time is usually the dominant factor. Back in the days when S5FS was popular and FFS was being developed (the early 1980s), average seek times (i.e., the time required to move from one randomly selected cylinder to another) were typically in the neighborhood of 30 milliseconds. At the time of this writing, average seek times are from 2 to 10 milliseconds. It takes less time to move the disk heads a shorter distance than a longer distance, but the relationship is not linear: we must take into account the time required for acceleration and deceleration of the disk heads and other factors as well. A typical disk drive might have 25,000 cylinders. The time to move the heads one cylinder might be .2 milliseconds, yet the average seek time might still be 4 milliseconds. For our purposes, however, it suffices to say that closer means faster.

Rotational latency times depend on the speed at which disks spin. In the early 1980s this was pretty much always 3600 RPM. Today's spin rates range from 5200 to 15,000 RPM. Assuming that the average rotational latency is the time required for half a revolution, rotational latency has gone from 8.3 milliseconds in the early 1980s to as little as 2 milliseconds today.

The transfer time depends both on rotational latency and on the number of sectors per track: the more sectors on a track, the smaller the fraction of a full revolution the platter must spin to pass one complete sector underneath a head. Since modern disks have more sectors in outer tracks than in inner tracks, the transfer time depends upon which track the sector is in. A typical drive might have 500 sectors in the inner tracks and 1000 in the outer tracks, with a sector size of 512 bytes. Thus at 10,000 RPM, the transfer rate can be as high as almost 85 MB/second, though this rate can be maintained only for as much data as there is in a track. Transferring data that is spread out on multiple tracks requires additional positioning time.

Many disk controllers automatically cache the contents of the current track: as soon as a disk head is selected after a seek to a new cylinder, a buffer in the controller begins to fill with the contents of the sectors passing under the head. Thus after one complete revolution, the entire contents of the track are in the buffers and each sector can be read without further delay.

This form of caching is innocuous in the sense that it can only help performance and has no effect on the semantics of disk operations. Another form of caching — *write-behind caching* — is used by many modern disk controllers (particularly SATA) to cache disk writes, allowing the writer to proceed without having to wait for the data to be written to the actual disk. This can

speed things up a fair amount, but can be a problem if, for example, there is a power failure and the data does not actually get to disk. Worse yet, data may be written to disk in a different order from what the writer intended (we discuss the consequences of this in Section 6.2.1). Some file systems (such as ZFS (Section 6.6.6)) cope with this correctly, others do not.

Keep in mind that typical processor speeds in the early 1980s were over a thousand times lower than they are today. Even though disk technology has improved since then, processor technology has improved even more and thus the discrepancy between processor speed and disk speed has increased dramatically.

We now see where we must work so as to improve file access times. The biggest issue is seek time: we need to minimize how often and how far the disk heads move while satisfying disk requests. A lesser but still important issue is rotational latency: we need to position file data on disk so as to minimize it.

6.1.2.1 The Rhinopias Disk Drive

We now specify a disk drive to use as a running example so that we can be specific in examining file-system performance. We want our drive to be representative of drives available today, though it will undoubtedly be obsolete soon. We also need to give it a name: we'll call it the Rhinopias drive. Our drive has the following characteristics:[2]

Rotation speed	10,000 RPM
Number of surfaces	8
Sector size	512 bytes
Sectors/track	500–1000; 750 average
Tracks/surface	100,000
Storage capacity	307.2 billion bytes
Average seek time	4 milliseconds
One-track seek time	.2 milliseconds
Maximum seek time	10 milliseconds

From this information we compute its maximum transfer rate, which occurs when we transfer consecutive sectors from one track, as 85.33 million bytes/sec. Of course, this occurs only on outer tracks. A better figure might be the maximum transfer rate on the average track of 750 sectors: 64 million bytes/sec.

What is the maximum transfer rate when we're transferring data that occupies more than one track? Let's first consider data that resides in a single cylinder. Though we can switch read/write heads quickly to go from one track to another in a cylinder, this requires some time, enough that by the time the drive switches from the read/write head for surface 1 to the head for surface 2, the disk has rotated some distance into the next sector. Thus if we want to access that next sector, we have to wait for the disk to spin almost one complete revolution.

What current disk drives (and thus our Rhinopias drive) do to avoid this problem is a trick called *head skewing*. Sector 1 on track 2 is not in the same relative position on the track as sector 1 of track 1; instead, it's offset by one sector. Thus, after accessing sectors 1 through 750 on one 750-sector track, by the time the drive switches to the disk head of the next track, the disk has rotated so that sector number 1 is just about to pass under the head. Thus to compute the time required to transfer all the data in one cylinder, we must add to the transfer time for all but the last

[2] We use the prefixes kilo-, mega-, and giga- to refer to powers of two: 2^{10}, 2^{20}, and 2^{30}. The terms thousand, million, and billion refer to powers of ten: 10^3, 10^6, and 10^9.

track the time required for the disk to rotate by one sector. For the average track, this is 1/750 of a disk rotation time (8 microseconds), which reduces the transfer rate to about 63.9 megabytes/sec from the 64 megabytes/sec of single-track transfers.

In addition, if our data resides in multiple cylinders, we need to factor into the computation the time required to move from one cylinder to the next. With average (750-sector) tracks, each cylinder contains 6000 sectors, or 3,072,000 bytes. After transferring that many bytes at 63.9 million bytes/sec, we must wait .2 milliseconds to move the read/write heads to the next cylinder. During that time the disk rotates almost 25 sectors. To avoid waiting for the disk to make an almost complete rotation to bring sector 1 under the read/write head, our drive uses *cylinder skewing*, which is similar to head skewing: sector 1 of the next cylinder is offset by 25 sectors. Thus by the time the heads reach the next cylinder, sector 1 is just about to rotate beneath them. So, to compute the average maximum transfer rate for data occupying more than one cylinder, we simply factor in the one-track seek time, giving us a rate of 63.7 million bytes/sec. (The rate would be 56.8 million bytes/sec if cylinder skewing were not used.)

6.1.3 PROBLEMS WITH S5FS

So what's wrong with S5FS? Lots, it turns out. Its file-allocation strategy results in slow file access. Its small block size contributes to slow file access. The segregation of inodes from other file-system information contributes even more to slow file access. Its limitations on directory-component names are inconvenient, and its lack of resilience in the face of crashes is a real killer.

Its performance problems are due to too many long seeks combined with too short data transfers. In particular, it does nothing whatsoever to minimize seek time. Consider its segregation of inodes from other file data. When accessing a file, first the inode is fetched, and then there is a long seek to the data region to fetch the file's data and indirect blocks. These blocks are, in general, not organized for fast sequential access. Instead, they're allocated when needed from the free list, which, though it's ordered in some fashion when the file system is created, is pretty well randomized after the file system has been used for a while. Thus successive data blocks of a file are in random locations within the data region. On average, the read/write heads must travel halfway across the data region to go from one file block to the next — and then, after the heads travel such a long distance, they transfer only 512 bytes of data.

To see the effect of all this, let's implement S5FS on the Rhinopias drive and analyze how well it performs. In particular, let's compute how fast a file can be read from disk to primary storage. How long does it take to do each of the seeks required to access each file block? If the data region occupied the entire disk, we could use the 4-millisecond average seek time. But since the data region is smaller than this, that would be a bit pessimistic. So let's be generous and say that the average seek between blocks takes 2 milliseconds. Once the heads are positioned over the desired cylinder, we have to wait the average rotational latency of 3 milliseconds. The time required to transfer one 512-byte sector is negligible in comparison, so every 5 milliseconds 512 bytes can be transferred. This gives us an average maximum transfer rate of 102.4 thousand bytes/sec. This is a remarkable 0.16% of Rhinopias's maximum speed! And note that the speed would have been considerably lower on the disk drives used in S5FS's heyday.

This performance is so terrible that one really wonders why this file system was used at all. It turns out that, though S5FS is amazingly slow by today's standards, the processors on the systems it was used on were also amazingly slow: instruction execution was measured not in billions per second, as is common today, but in tens or hundreds of thousands per second. Thus the data-transfer rate of S5FS was not badly out of line with application requirements. Furthermore, the typical file size was much smaller than it is today: the average Unix file contained just a few thousand bytes. So, though definitely slow, S5FS wasn't out of line in early Unix systems and was certainly usable.

6.1.4 IMPROVING PERFORMANCE

It should be clear from the above discussion that to improve file-system performance we need both to minimize the setup delays of seek time and rotational latency and to maximize the amount of useful data transferred after these delays. In this section we discuss a few techniques for making such improvements, starting with the fairly straightforward and continuing to the quite remarkable. We mention some of the file systems that employ each of these techniques, and discuss these file systems in more detail in Section 6.5.

One approach to improving file-system performance is to take advantage of buffering. We've already mentioned and will continue to discuss the buffering of file-system data by the operating system. But what about buffering implemented in the disk hardware? Rhinopias's user manual (if it had one) would show that it, like other disk drives, has a *pre-fetch buffer* that stores the contents of the current track in a buffer managed by the disk controller. Of course, the buffer doesn't fill until the disk makes a complete rotation while the read/write heads are properly positioned and the appropriate head selected. We can certainly take advantage of this to improve latency for reads, but it provides no improvement for writes.

6.1.4.1 Larger Blocks and Improved Layouts

In a first attempt to improve file-system performance, we might simply try to group file-system data so as to reduce seek time and to use a larger block size so as to maximize the useful data transferred. These were the major innovations of the FFS (sometimes called UFS) system that supplanted S5FS on Unix systems starting in the early to mid-1980s (McKusick, Joy, et al. 1984). The Ext2 file system of Linux is similar to FFS, and we discuss the two together.

Block Size Increasing the file-system block size seems easy. Simply switching from a 512-byte block to an 8-kilobyte block sounds straightforward, but doing so causes problems: the average space wasted per file due to internal fragmentation (Section 3.3.1) is half a block. Thus if there are 20,000 files, the wasted disk space goes from 5.12 million bytes to 80 million bytes — a fair amount of space, particularly if your disk holds only 60 million bytes (see Figure 6.8).

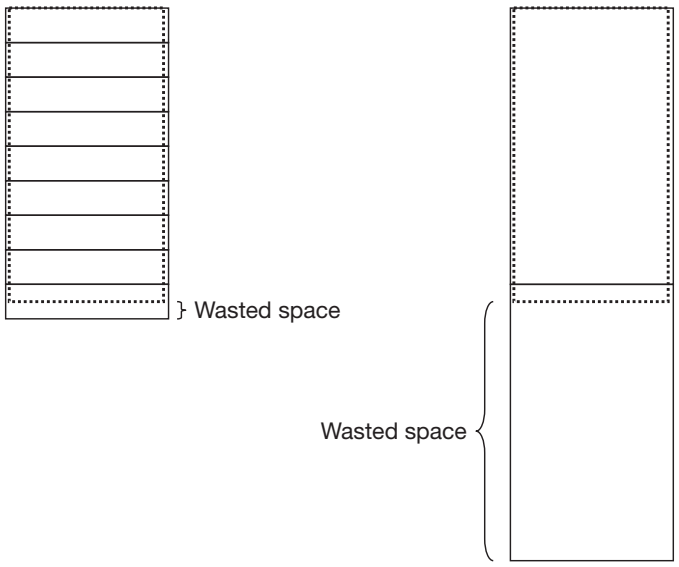


FIGURE 6.8 Wasted space with 512-byte blocks (left) and 8-kilobyte blocks (right).

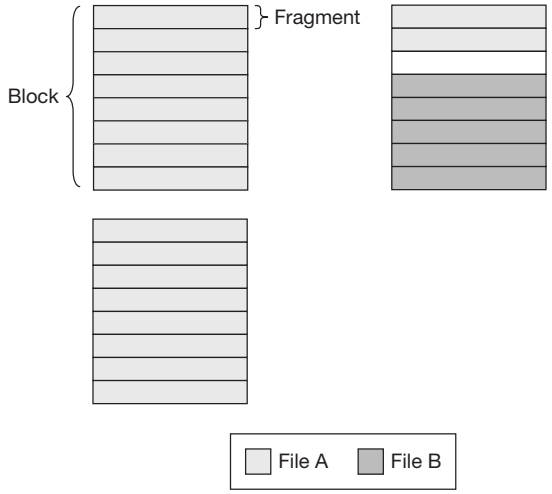


FIGURE 6.9 File A has two full blocks plus a partial block containing two fragments, for a total of 18 fragments. File B has five fragments in a partial block.

FFS used a rather complicated strategy to avoid this problem: files were allocated in blocks except for what would otherwise be the last block, for which a partial block is used. To obtain such partial blocks, blocks are divided into pieces somewhat confusingly called *fragments*, each typically two sectors (1024 bytes) in length. Thus if a file's length is 17.5 fragments (17,920 bytes) and the block size is 8 fragments, two blocks would be allocated to hold its first 16,384 bytes and a partial block consisting of two contiguous fragments would be allocated to hold its last 1536 bytes (see Figure 6.9). Thus the wastage due to external fragmentation would be 512 bytes, rather than the 6656 bytes it would have been if only whole blocks were used. Of course, in S5FS the wastage would have been zero, though remember that the number of seeks required to fetch the blocks of the file is three, whether the file is allocated wholly in blocks or with whole blocks followed by a partial block. S5FS would require 35 seeks to fetch the contents of this file.

The file-system block size is equal to some small power-of-two multiple of the fragment size — in the first version of FFS it could be 2, 4, or 8 fragments, in later versions up to 64 fragments. This block size is a parameter and is fixed for each on-disk instance of the file system.

This block-size strategy doesn't sound complicated, but that's just because we're not finished yet. FFS must keep track not only of free whole blocks but also of each of the various sizes of partial blocks. We discuss how it does this soon, when we talk about optimizing block placement. But for the time being let's assume FFS can handle the partial blocks and examine their use.

Suppose our 17.5-fragment file grows by 513 bytes. Another fragment is required to hold it, since its length is now 18 fragments plus one byte. FFS can't simply allocate another fragment from any place in the file system, because that would yield a file with two partial blocks: a two-fragment one and a one-fragment one.

Instead, we'd like the last portion of the file to reside in a single three-fragment partial block — a block consisting of three contiguous fragments — that can be fetched after just one seek. This is easily done if the fragment following our two-fragment partial block is free: FFS simply allocates it and extends the two-fragment block to be a three-fragment block (Figure 6.10). But if this next fragment is not free, then FFS must find three free contiguous fragments, copy the original two-fragment block into its first two fragments, replace the original partial block in the file's disk map with this new one, and return the original block to free storage. Of course if, for example, our 17.5-fragment file grows by seven fragments, FFS expands its partial block into an (8-fragment) whole block and gives it a new one-fragment partial block (Figure 6.11). Now things are getting complicated — and expensive.

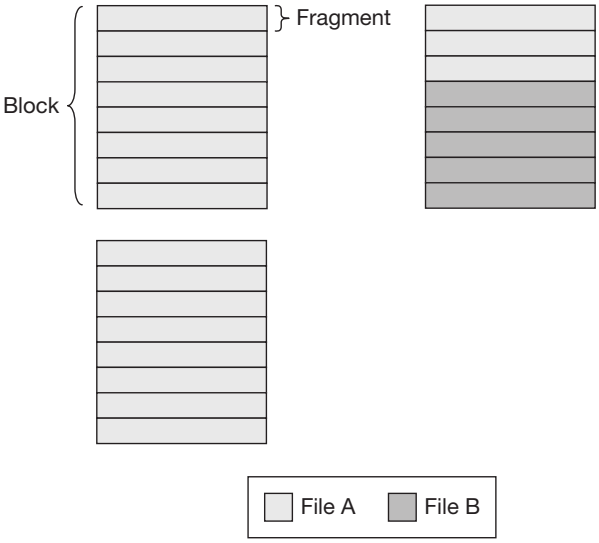


FIGURE 6.10 File A grows by one fragment into an adjacent free fragment.

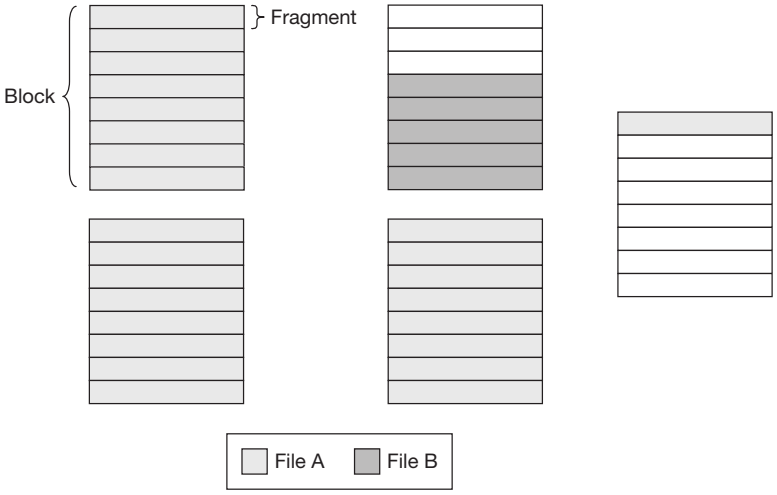


FIGURE 6.11 File A grows by six more fragments. Its partial block is copied into a full block and its last fragment is in a new partial block.

To reduce the expense, FFS tries to arrange things so there are always free fragments for partial blocks to expand into. One way for FFS to do this, in allocating a partial block for a file, is actually to give the file a whole block starting on a block boundary, and then free the unneeded fragments. Assuming FFS always does this, there is always room to extend partial blocks. Of course, this also defeats the entire purpose of using partial blocks: it's effectively reserving each partial block's expansion fragments, and this isn't a whole lot different from simply allocating whole blocks.

All this begins to make sense, however, if FFS uses a two-policy allocation strategy. As long as plenty of whole blocks are available, when a partial block is needed, FFS allocates a whole block and then frees the unneeded fragments. This is known as the *optimization-for-time policy*. However, once an FFS instance starts running low on whole blocks but instead has lots of

short runs of free fragments, it switches to allocating partial blocks containing just the number of fragments needed and pays the price of copying when files grow. This is known as the *optimization-for-space policy*. A parameter in each on-disk FFS instance specifies at what point the allocation strategy switches from one policy to the other.

Reducing Seek Time Reducing seek time is conceptually straightforward — simply keep the data items that are likely to be accessed in close time proximity close to one another on the disk. Thus the blocks of each file should be close to one another. Since files' inodes are accessed in conjunction with their data blocks, this means keeping data blocks close to the corresponding inodes. Since the inodes of a directory are often examined all at once (for example, when listing the attributes of all the files in a directory), this also means keeping the inodes of one directory close to one another.

One way to do this is to divide the disk into a number of regions, each comprised of relatively few adjacent cylinders. Items that are to be kept close together are placed in the same region. Of course, we've got to be careful or we'll end up attempting to put everything in the same region. Thus, in addition to identifying items that are to be stored in close to one another, we need to identify items that are to be spread out.

FFS and Ext2 both use this region approach; in the former the regions are called *cylinder groups* (see Figure 6.12), in the latter *block groups*. Each group contains space set aside for inodes as well as space for data and indirect blocks. In general, inodes are stored in the same group as the inode of the directory containing them. Data blocks are stored in the same group as their inode. To spread things out, inodes of subdirectories are put in different groups from their parents. Only those direct blocks are stored in the inode's group. The remaining portion of files requiring indirect blocks is divided into pieces (two megabytes each in some versions of FFS), each of which is stored in a separate group.

The result of all this is that the inodes of files within a directory can be examined quickly, requiring no worse than short seeks, except when a subdirectory is encountered. Then a long seek is required to reach its inode. Accessing the blocks of files requires long seeks only every two megabytes: otherwise the seeks are short.

An important part of this procedure for placing pieces of files into cylinder groups is identifying cylinder groups with plenty of free space. The intent is to spread out the allocations so that, for example, the cylinder group assigned to an inode has enough space to hold the first two megabytes of the file, as well as inodes and initial portions of other files in the directory. In FFS this is accomplished by using quadratic hashing, which both quickly finds cylinder groups with sufficient free space and spreads out the allocations so that the cylinder groups are used uniformly. The approach works well as long as roughly only 90% of disk space is in use. So, FFS keeps its last 10% of disk space as a reserve, making it available only to privileged users.

Suppose we stop here, having improved on S5FS by increasing the block size and improving locality. How well have we done? To put things in the most favorable light, let's look at the average

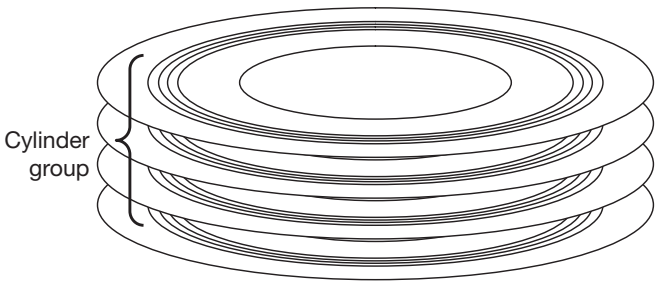


FIGURE 6.12 Four adjacent cylinders forming a cylinder group in FFS.

maximum transfer speed for a single file, one that fits entirely within a cylinder group. Let's further assume that our Rhinopias disk is configured with cylinder groups of 20 cylinders each, or roughly 60 megabytes for the average cylinder group. We'll have to make an educated guess on the average seek time within a cylinder group: since the one-track seek time is .2 milliseconds and the time required to seek 500 cylinders (halfway across the disk) is only 20 times greater, the time required to seek 10 cylinders is probably fairly close to the one-track seek time — let's say .3 milliseconds.

Once the disk head is positioned over the correct cylinder, we have to wait on average half a rotation for the desired block to come under the disk head — this takes an average wait of 3 milliseconds. The time for one 8-kilobyte block (16 sectors) to pass under a disk head, assuming 800 sectors/track, is .12 milliseconds. Thus, on average, in every 3.42 milliseconds we can transfer one 8-kilobyte block. This gives us a transfer speed of 2.40 million bytes/sec. This is around 3.7% of Rhinopias's capacity — more than 20 times the 0.16% we computed for S5FS, but still pretty miserable.

Reducing Rotational Latency It should be clear from the above analysis that the major cause of our performance problems is now the rotational latency — we've done a pretty good job of reducing seek delays. Not much can be done to reduce the rotational delay that occurs just after a seek completes. But if there are a number of blocks of the file in the cylinder, perhaps we can arrange them so that, after an initial delay, they can all be read in without further rotational delays.

In our discussion on reducing seek time, the order in which blocks of a file are requested didn't matter much, particularly for files fitting in one cylinder group. But when we discuss rotational delays, order is important. If blocks of a file are requested randomly, nothing can be done. But if blocks are requested sequentially, we can usefully access a number of them in a single disk rotation.

Let's assume that blocks of a file are being requested sequentially. How should we arrange blocks on a track (this will generalize to cylinders) so that they can be read with minimal rotational delay? It might seem obvious at first glance that we should place our blocks in consecutive locations on the track. Thus, after accessing one block, we can immediately access the next (see Figure 6.13). However, this doesn't necessarily work all that well, particularly on early 21st-century systems.

Suppose the block size is 8 kilobytes and each track holds 50 blocks. Two separate disk operations are required to access the two consecutive blocks. The operating system starts the first operation, waits until an interrupt notifies it of completion, and then starts the second. However, some period of time elapses from when the first operation completes to when the second is started. During this time the disk rotates; how far depends upon both its rotation speed and the speed of the processor. Even though there is space between disk sectors and thus between the last sector of the first block and the first sector of the second block, the disk might rotate far enough that by the time the second operation is issued, the read/write head is beyond the beginning of the second block. Thus, to read the entire block, we must wait for the disk to make an almost complete revolution, bringing the beginning of the block to the read/write head. How long does this take? If the disk is spinning at 10,000 RPM, a single revolution takes 6 milliseconds — a significant amount of time; in fact, we'd probably be better off if the second block were on a different cylinder within the cylinder group.

Suppose our processor takes 100 microseconds to field a disk-operation-completion interrupt and start the next operation. (This is a long time by today's standards, but on processors of the early 1980s, it was the time to execute at most 100 instructions.) With 50 blocks/track (16 sectors/block), each block (including the inter-sector gaps) takes 120 microseconds to pass under the read/write head. Thus by the time the second operation starts, the read/write head is near the end of the second block and the disk must rotate roughly one complete revolution to read the entire block.

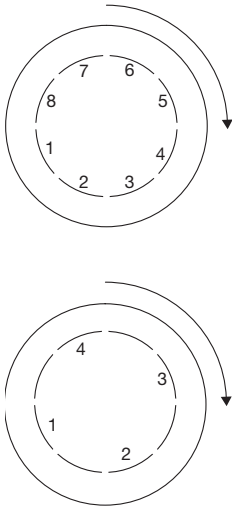


FIGURE 6.13 Above, a disk surface with consecutive block access; below, block interleaving is used.

To alleviate this problem, FFS used a technique known as *block interleaving*. Rather than placing successive blocks of a file in contiguous disk locations, it places the second block some number of block locations after the first. In our example, if successive blocks from the file are separated by a single block on disk, then the read/write head is positioned to begin a read of the second block after only 20 microseconds after the operation is started — a factor-of-300 improvement over the wait time in the approach where we don't skip a block.

So, what's the average maximum transfer speed of FFS with block interleaving? As earlier, let's consider a 2-megabyte file fitting in a cylinder group. Since each track holds 50 blocks, a Rhinopias cylinder has 400. Thus, in this best case, all the blocks of the file are in the same cylinder. To transfer these blocks, we first have a .3-millisecond seek to that cylinder. Then there's a 3-millisecond rotational delay. At this point, the disk heads are positioned to read the first block without further delay. Simplifying things slightly: with each disk rotation 25 blocks are transferred (since every other block is skipped). Thus 2 megabytes are transferred after 10.24 revolutions of the disk, which takes 61.44 milliseconds. Adding in the initial seek and rotational delays of 3.3 milliseconds yields an effective transfer speed of 32.4 million bytes/second, about 50% of Rhinopias's maximum speed. The good news is that we have a 13-fold improvement over the previous figure. The bad news is that, even in this best-case analysis, we can achieve only 50% of Rhinopias's capacity.

Clustering and Extents With today's faster processors it should be possible both to respond to a disk interrupt and to start the next operation so quickly that block interleaving is not needed. However, the balance could change if, for example, disk technology improves so that disks spin much faster. A better disk-access strategy is to recognize ahead of time that more than one block is to be accessed and issue one request for all of them. Alternatively, we might do away with fixed-size blocks entirely and allocate disk space in variable-size but generally larger units.

Both Ext2 and later versions of FFS use multiple-block-at-a-time strategies known as *block clustering*. In Ext2's preallocation approach, contiguous disk blocks are allocated eight at a time when the first of each eight file-system blocks is written. The preallocated blocks are available for other files but are not taken back unless there is a shortage of disk space. In FFS, as implemented on Solaris (McVoy and Kleiman 1991), disk-block allocation is delayed until a certain number of file-system blocks, say eight, are ready to be written, and at this point contiguous disk blocks are allocated for them. Both strategies result in an effective block size that's eight times the actual block size, but with minimal internal fragmentation costs.

An approach that actually predates block clustering is *extent-based allocation*. Rather than make any pretense of allocation in fixed-size blocks, files are treated as collections of large contiguous regions of disk space called *extents*. In principle, a large file can consist of a single such extent, thus allowing fast access with a minimum of metadata. However, since extents are of variable size, external fragmentation becomes a problem — a heavily used file system's free space might mostly reside in small pieces, thus making it impossible to allocate large extents. Thus, in the worst case, an extent-based file system might require just as much metadata as a block-based one and have even worse performance problems because file data is widely scattered and in small pieces.

Windows's NTFS is an extent-based file system. Like most such systems, it represents a file's disk space with a *run list* consisting of items describing each successive extent; each item contains an extent's starting disk address and length (Figure 6.14). If the run list becomes so long that it no longer fits in a file record (the NTFS equivalent of an inode: see Section 6.1.5 below), additional file records are allocated for the additional run-list segments and a separate list is maintained that indexes these segments. Thus, for example, a highly fragmented file might have 100 extents. Rather than searching linearly through a 100-element run list to find a particular extent, a top-level run list (referred to in NTFS as an "attribute-list attribute") with just a few entries is searched to find which expansion file record contains the entry for the extent in question.

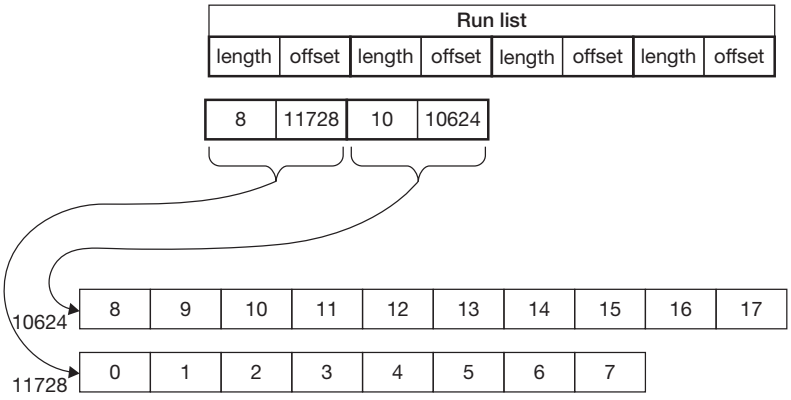


FIGURE 6.14 NTFS uses run lists to refer to multiple extents. The figure shows two: one starting at location 11728 and containing 8 blocks, the other starting at location 10624 and containing 10 blocks.

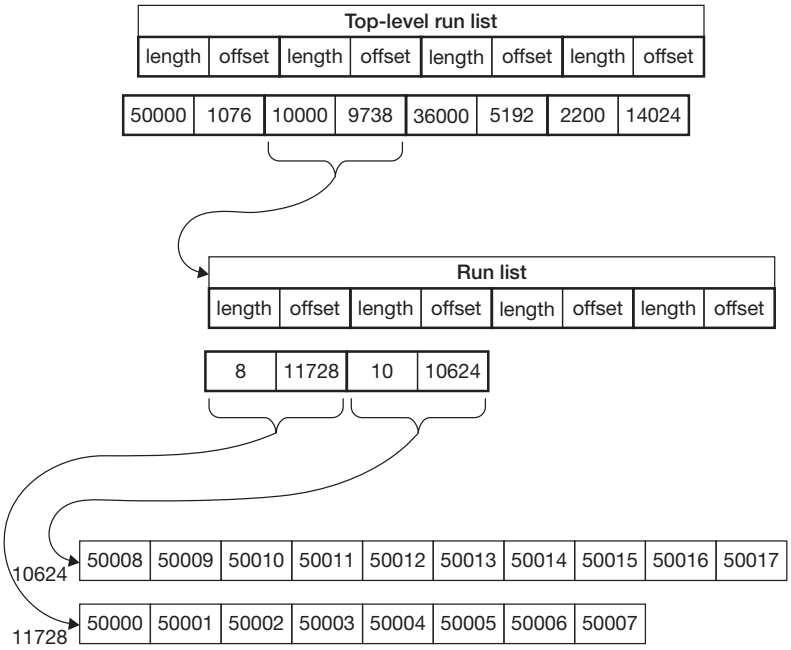


FIGURE 6.15 A two-level run list in NTFS. To find block 50011, we search the top-level run list for a pointer to the file record containing the run list that refers to this block. In this case it's pointed to by the second entry, which contains a pointer to the run list that refers to extents containing file blocks starting with 50,000 and continuing for 10,000 blocks. We show the first two references to extents in that run list. The block we're after is in the second one.

Then the run list in this file record, containing perhaps just 10 entries, is searched. This two-level approach reduces the search time considerably (see Figure 6.15).

NTFS requires active avoidance of fragmentation problems. It uses a defragmenter program to rearrange free and allocated space on disk so that files are comprised of few but large extents and free space similarly is comprised of a small number of large regions. Overall good performance is achieved if the defragmenter is run often enough, which could be anywhere between once a day and never depending upon how full the disk is.

6.1.4.2 Aggressive Caching and Optimized Writes

The file-system layout policies discussed so far have organized disk space primarily to optimize the sequential reading of files (see Figure 6.16). But it's not clear that this is necessarily the

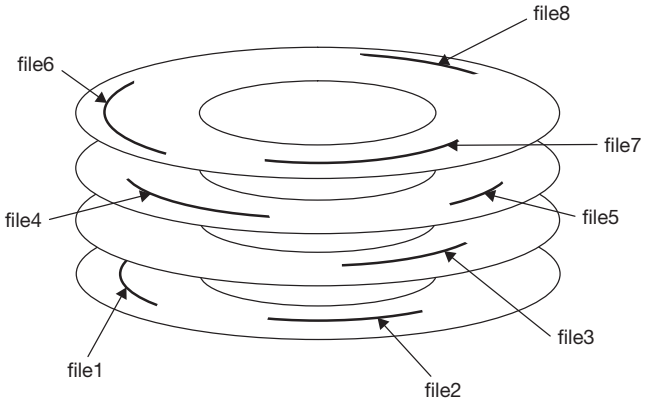


FIGURE 6.16 Files organized for optimal read access using single extents.

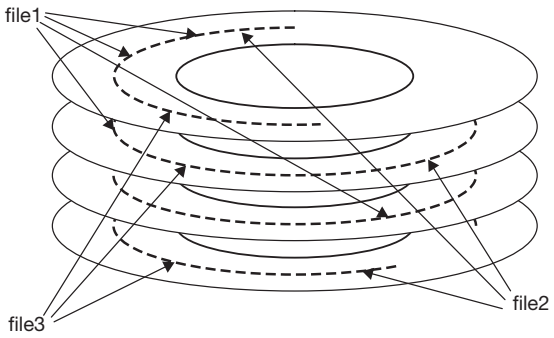


FIGURE 6.17 Portions of files are placed in the log (shown here as occupying most of one cylinder) in the order in which they were written.

best strategy. Suppose lots of primary memory is available for caching file-system blocks. For active files, most reads might well be satisfied from the cache. However, so that our data doesn't disappear if the system crashes, modifications are written to disk. Perhaps what we really want to optimize is writes.

Rather than trying to aggregate the data of any one file for fast read access, let's simply maintain a sequential log of file updates (Figure 6.17). We buffer the updates and write batches of them to disk in the order in which they arrive. Thus we're assured of fast updating, rather than hoping for fast reading.

On the one hand, this is optimal use of the disk: since we are usually writing to contiguous locations, the greatest seek delays we'll normally have are in moving the heads from one cylinder to the next as cylinders are filled. If we write in units of whole tracks and provide sufficient buffering, we minimize the effects of rotational delays. On the other hand, though, the approach looks remarkably stupid, since reading a file would seem to require searching through the update log to track down the blocks of the file and thus would require many expensive disk operations.

The approach becomes a lot less stupid, though, if we take advantage of a large file-system cache in primary memory and either access file blocks from it directly or use it to find them quickly. This is the basis of what are called *log-structured file systems*. Pioneering work on such file systems is described in (Rosenblum and Ousterhout 1992), which discusses the *Sprite File System*. This system is logically similar to FFS in its use of inodes and indirect blocks, but is radically different in most other respects.

To illustrate the utility of the log-structured approach, we adapt an example from (Rosenblum and Ousterhout 1992). Suppose we are creating two single-block files *dir1/file1* and *dir2/file2*. In FFS we'd allocate and initialize the inode for *file1* and write it to disk before we update *dir1* to refer to it. We'd then add the appropriate entry to *dir1* and write the modified block that contains the entry to disk — let's assume we can do this without adding a new block to *dir1*. Finally we write data to *file1*, which requires us to allocate a disk block, fill it with the data being written, write this block to disk, and modify *file1*'s inode to refer to the new block.

Doing all this requires six disk writes, including those to modify the bit maps of free blocks and free inodes (note that we write *file1*'s inode twice — once when it's allocated and once when we add a block to it). Since we're also creating *dir2/file2*, we need six more disk writes. Most of these writes potentially go to scattered disk locations — each requires a seek (see Figure 6.18).

The Sprite File System accomplishes all this with a single, long disk write. However, this write includes some additional data so that the files can be read back again. Here's what's done.

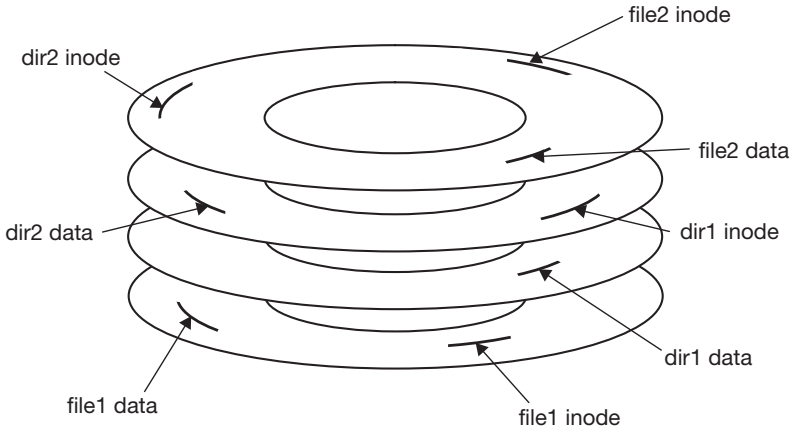


FIGURE 6.18 Creating two files in FFS requires writing each of the blocks shown here, with the inodes for *file1* and *file2* being written twice. Not shown are the blocks containing the free vector, which must be written twice, once for *file1*'s data block and again for *file2*'s data block.

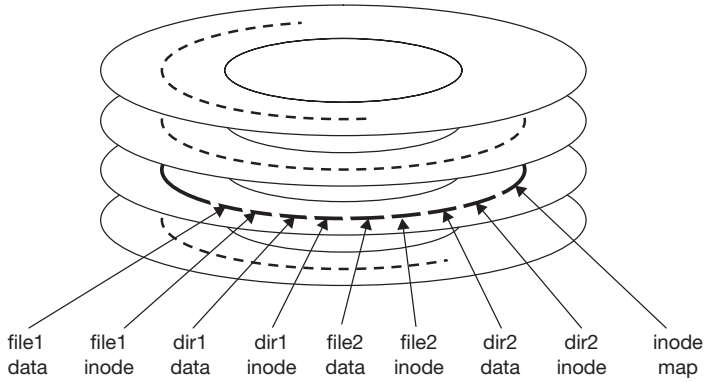


FIGURE 6.19 In Sprite FS, everything required to create *file1* and *file2* is added to the log in a single write. Here, again, the log occupies most of one cylinder.

Creating *file1* in *dir1* requires modifying the contents of *dir1* to refer to *file1*. Rather than simply modifying the current block of *dir1* that holds this information, we allocate and write out a new block to replace it. So that this new block can be found, we have to modify the disk map in *dir1*'s inode. However, rather than simply modifying the old inode, we allocate and write out a new one to replace the old.

Of course, we need to keep track of where this new version of *dir1*'s inode is — a new data structure, called an *inode map*, is used for this. The inode map is a simple array that's indexed by inode number; each entry points to the current location of the inode and contains some other information about it. The map's current contents are in primary memory. As it's updated it's written out to disk in pieces, along with other update information. The location of these pieces is recorded in a *checkpoint region* that is stored at a fixed location on disk. There are actually two copies of this region at two different fixed locations; the two alternate between being the current copy and the copy being updated.

So, the single, long disk write contains the data block for *file1*, *file1*'s inode, the modified data block containing the *dir1* entry referring to *file1*, and the modified *dir1* inode. It also contains similar information for *dir2* and *file2*, as well as the pieces of the inode map necessary to locate the new and modified inodes (see Figure 6.19). What's missing is anything recording block allocation — such information turns out to be unnecessary, as we discuss shortly.

First, though, to make sure we understand how everything works, let's go over what must be done to read the data in *file1*. We first find its inode by following the path *dir1/file1*. Some unspecified directory contains the inode number of *dir1*; suppose this number is 1137. We look this up in the inode map, which is in primary memory. The entry there gives us the disk location of the inode for *dir1*. Since we have a large cache, we probably can find this inode there, but if not, we fetch it from disk. The inode's disk map refers us to the block containing the *file1* entry, which contains *file1*'s inode number. We look this number up in the inode map to find the inode's address. We fetch this inode, probably from the cache. Finally, the inode contains the disk map, which refers us to the file's data.

Let's compare this with what it would take to read *file1* in FFS. The only additional steps required in Sprite FS are the two in which it reads the inode map. Since the inode map is in primary memory, reading from it doesn't add to the cost of reading from the file. What if *file1* has multiple data blocks that we want to read sequentially and quickly? If they were produced sequentially (and quickly) on Sprite FS, they'd be in contiguous locations on disk. But if they were produced in some other fashion, they might be in scattered locations. In FFS, they would normally be near one another. However, if the file is in active use, the disk locations of the blocks don't matter, since they'll be accessed directly from the cache.

The next issue in Sprite FS is disk-space management. Sprite FS requires large contiguous regions of free space so that it can write out large amounts of data at once. The larger a free region is, the more such writes Sprite FS can handle without seek delays, and thus the better its performance.

When we initialize a disk for Sprite FS, some small beginning portion will be allocated and the rest will be free, thus giving us a large contiguous block of free storage. As we use the file system, we allocate more and more disk space from the beginning of the free-storage area. As discussed above, many of these allocations will supplant storage previously in use, leaving a lot of garbage blocks within the allocated region of the disk. Somehow we have to identify these garbage regions and consolidate them as free space.

If we can identify the garbage regions, there are two ways we might consolidate them. One is simply to link them together. This is easy to do, but doesn't provide the necessary large contiguous regions of free space. The other way is to rearrange the various regions of disk space, moving all the allocated portions to one end and leaving one large, contiguous free region. This has the desired result but is rather complicated to do. What Sprite FS does is a combination of the two. It divides the disk into large segments, each either 512 kilobytes or one megabyte in length. The live (that is, non-garbage) blocks are identified in a number of segments and are consolidated into a smaller number of segments by moving some of them from one segment to another. Those segments made entirely free are linked together.

Each segment contains a *segment summary* identifying everything in the segment. Thus, say, if a data block is moved from one segment to another, the segment summary identifies the inode number and block number of the file the block belongs to; this makes it possible to modify the appropriate disk map to contain the block's new location. To determine whether a block is live or garbage, the segment summary is consulted to find where the block belongs. If the disk map still refers to the block, it's live, otherwise it's garbage.

Though Sprite FS was strictly a research file system and was never put into widespread use, it demonstrated the feasibility and desirability of the log-structured approach. It influenced the design of many subsequent file systems, including WAFL and ZFS, which we discuss in Sections 6.2.2.3 and 6.5.

6.1.5 DYNAMIC INODES

In describing S5FS we introduced the notion of an inode, a notion used in one form or another in almost all file systems. In many file systems inodes are allocated as in S5FS: from a static array.

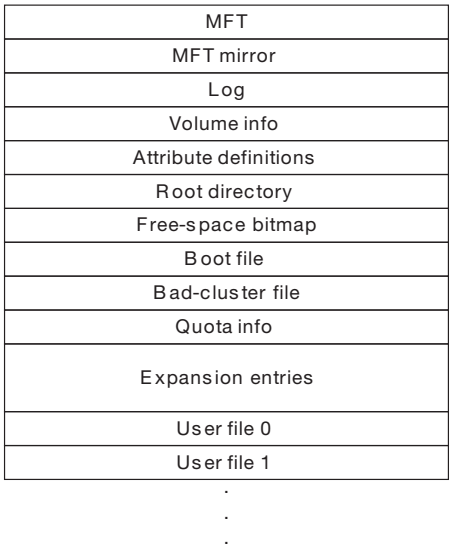


FIGURE 6.20 NTFS's Master File Table (MFT). Each entry is one kilobyte long and refers to a file. Since the MFT is a file, it has an entry in itself (the first entry). There's a copy of the MFT called the MFT mirror. Then there are a number of other entries for system files and metadata files, and then the entries for ordinary files.

In FFS, Ext2, and Ext3, this array is broken up into pieces, but is still static. Thus their inodes are still identified by their indices within this array.

This causes two sorts of problems. The first is that in laying out a file system on disk, you have to predict the maximum number of files the system will have — that is, you need to determine the size of this inode array. If your guess is too large, the space is wasted for unneeded inodes. If your guess is too low, eventually you can no longer create files in your file system, even though plenty of disk space is available for file contents. The second problem is that you might want to add more disk space to the file system. The original assumptions about the number of inodes needed might no longer be valid if the file system becomes larger.

What's clearly needed is a means for allocating inodes from the same pool of disk blocks from which everything else is allocated. This doesn't seem like a big deal, but we often still want to be able to refer to an inode by a simple index rather than by a disk address — this facilitates having standard inode numbers for certain key files, such as roots of file systems, as well as making it possible to use fewer bits for the inode number than for general disk addresses.

A simple approach that maintains the notion of an index is to treat the array of inodes as a file. This inode file is represented just as any other file and thus can grow and shrink as needed. Of course, like any other file, an inode is required to refer to it. And this inode, like any other inode, is in the inode file. This might seem like a recipe for infinite recursion, but the inode for the inode file is simply kept at a fixed location on disk, with a backup copy at another fixed location in case of problems with the first copy.

NTFS is organized in this way. It uses the term *file record* rather than inode, and the inode file is called the *master file table* (MFT). Thus the master file table is a file containing an array of file records (see Figure 6.20).

6.2 CRASH RESILIENCY

Computer crashes are annoying. For file systems, we'd like to make sure they're merely annoying and not devastating — in particular, that no data is lost. Early versions of S5FS and other file systems were capable of devastation — a system crash could destroy not only what you'd spent the last three hours adding to a file, but its prior contents as well. Though now you should still back up any files you value, modern file systems have made great strides in minimizing data loss from crashes.

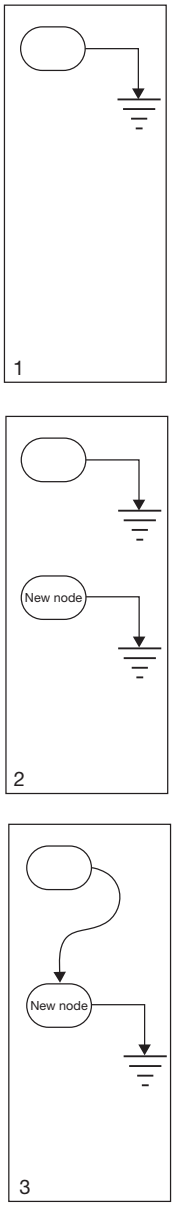


FIGURE 6.21 Adding a node to a linked list.

In this section we discuss the problems that bedeviled S5FS and examine how they have been mitigated in modern versions of FFS as well as in recent file systems, including Ext3, NTFS, WAFL, and ZFS.

6.2.1 WHAT GOES WRONG

How do computer crashes take such a toll on file systems? We might expect some ambiguity about the effect of a write that was taking place just as the system crashed, but how could S5FS lose entire files?

The culprits are the metadata structures — those data structures that form the structure of the file system itself. Examples are inodes, indirect blocks, directories, and the like. If these are damaged, then some user data, even though on the disk, could be inaccessible. Worse yet, the metadata structures could become so scrambled that one data item might appear in multiple files, "good" data might appear in free space, etc. Thus making certain that anything survives a crash entails special care of these metadata structures. It was lack of care of such structures that caused files in early S5FS to disappear.

A simple example of the sort of problem we're dealing with is adding an item to the end of a singly linked list. Figure 6.21 shows an apparently safe way of doing this. A list item consists of a data field and a link field; the last item's link field contains 0. To add a new item to an existing list, we first write the new item to disk, ensuring that its link field contains zero. Then we write out a new version of what had been the end of the list, changing its link field to contain the address of the new item. If the system crashes before this write takes place, the worst that could happen, it seems, is that our new item isn't appended to the list: when the system comes back up from the crash, the linked list should be as it was before we tried to add the item.

But things aren't this simple. As discussed in Chapter 4, most operating systems employ caches for file-system data and metadata. Active portions of files are in primary storage; modified data and metadata are not written to disk immediately. This caching is done so that many file operations can take place at memory speed rather than at disk speed. In the event of a crash, however, we have two problems.

The first is that many file modifications are only in the cache and were never written out to disk: when the cache's contents are lost in the crash, the modifications are lost too. Thus the application might have written to the file and received every indication that the operation succeeded (for example, the system call returned with a success status), but nevertheless, after recovery from the crash, the modifications don't appear in the file (Figure 6.22). From one point

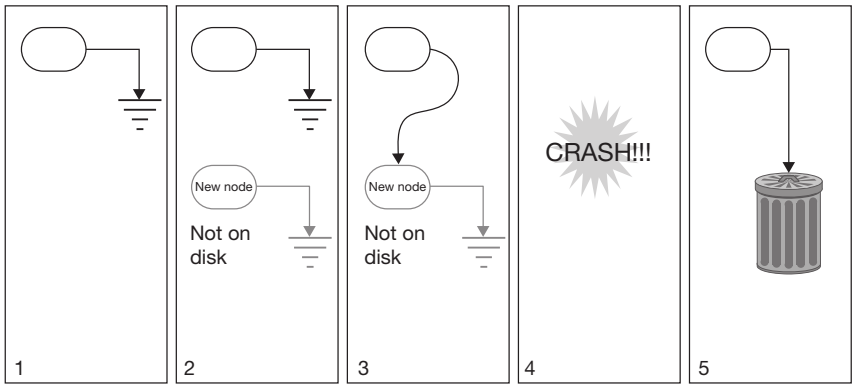


FIGURE 6.22 Adding a node to a linked list via a cache.

of view, this isn't a big deal, since it is equivalent to the system's crashing slightly earlier. From another, though, the application might have printed out a message such as "file successfully modified," leading the user to believe that the file really was modified. If the data involved recorded something such as the hand-in of a homework assignment or the purchase of a stock before its price went up, this could be pretty disastrous.

Things get worse with the second problem: the modifications that were written from the cache to the disk weren't necessarily the oldest ones. Modifications can be written to disk in a different order from that in which they were sent to the cache. This would happen if, for example, a least-recently-used policy is used to determine when data and metadata are written from the cache to disk. It is this sort of problem that can cause the destruction of entire files — not just the loss of the most recent updates.

To see how this could happen, let's take another look at our linked-list example (Figure 6.21). We first allocate a new disk block and modify a cached copy of the block to contain the new item with a zeroed link field. We then modify the cached copy of the disk block containing what had been the last item in the list so that it points to the new item. So far, all changes are in the cache; if the system were to crash, the version of the data structure on disk would be unchanged.

However, suppose the cached copy of the old item (what used to be the last item of the list) is written out to disk and then, before the cached copy of the new item can be written out, the system crashes. When the system comes back up, the former last item of our list now points to what is probably garbage. We have a badly damaged list structure that could cause much harm, particularly if it's metadata. Are metadata really stored in the form of such singly linked lists? Most definitely — disk maps, run lists of extents, and directories contain them.

Our general problem is that the metadata structures on disk at any particular moment may not be *well formed*: they may not be how they are supposed to be. In a well formed linked list, each node is supposed either to point to another node or to have a null pointer. In a well formed directory hierarchy (in most systems), the directories and their hard links form a directed acyclic graph (DAG) — if there is a cycle, the directory hierarchy is not well formed. If a disk block appears in more than one file or in both a file and in free space, then the disk map, free space, or both are not well formed. If a disk block is unaccounted for — if it's not in free space and it's not in a file — then free space is not well formed, since it is supposed to include all disk blocks that are not otherwise used.

If all the metadata structures are well formed, the file system is *consistent*. Thus the problem is that, because of caching, what's on the disk is not only not the current version of the file system, it's not necessarily even a consistent version of the file system. What applications perceive as the file system, which we call the *cache view*, is a combination of what is in the cache and what is on disk.

Our discussion above has focused on metadata. While ordinary data is pretty important too, metadata problems cause more widespread damage than data problems. If user data is inconsistent after a crash, there is clearly trouble, but it's localized to a particular file and does not affect the integrity of the entire file system. Thus file-system crash protection has traditionally focused on metadata without any real concern for data issues, a simplification that aids performance by allowing aggressive caching.

Another class of problems arises from media failures of various sorts. A disk sector can spontaneously go bad. A disk head can collide with a disk surface and destroy a good portion of it. A power glitch can cause a disk controller to write out random bits. These issues have nothing to do with caching and may or may not be easy to detect (a head crash has one positive attribute: it's pretty obvious something has happened). Bad sectors can be isolated and effectively removed from the file system. If the file system is mirrored, new sectors can be allocated and loaded with the copy.

6.2.2 DEALING WITH CRASHES

It is clear what needs to be done to make sure we can recover from crashes:[3] simply make certain that what is on disk is either consistent or contains enough information to make it consistent easily. This is relatively simple if we can ignore caching. We thus begin by assuming there is no cache and examine what can be done, and then we look at the effects of adding the cache.

Performing an update system call, such as *write*, *create*, *delete*, and *rename*, to a file requires a number of disk operations that affect both data and metadata. One approach to making file systems crash-tolerant is to make sure that each of these operations takes the file system from one consistent state to another. The idea is that at all times the file system is consistent and that if there is a crash, what is on disk is a valid file system. This doesn't quite work, however, because in some cases multiple disk writes are necessary to move from one consistent state to another.

We saw an example of this already in adding an item to a singly linked list (see Figures 6.21 and 6.22). Even when we do the disk writes in the correct order — first writing the contents of the node that is to be added, then writing out the new value of the node that points to it — a crash between the two writes would leave the file system in an inconsistent state, since there would be a block that is neither part of a file nor in free space. This isn't so bad, however: other than reducing the amount of available space a bit, we haven't damaged the file system. Furthermore, we can clean this up relatively easily by garbage-collecting these orphaned disk blocks. We'll consider this an *innocuous inconsistency*.

Another situation arises in renaming a file. This involves removing the entry for the file in one directory and putting it in another. We should avoid doing this in just that order, since a system crash between the two operations would make the file not appear in any directory afterwards. The correct way is first to add the file to the new directory, then remove it from the old. If the system crashes between these two operations, the file will be in both directories. This isn't exactly what was intended, but the file system remains consistent and the extra link can easily be removed. (See Exercise 13 for the issue of the file's link count, stored in the inode or equivalent.)

Some operations may require many steps. Writing a large amount of data to a file can require any number of separate disk operations. Similarly, deleting a large file can require many steps as each of the file's blocks is freed. If a crash occurs in the middle of such a multi-step operation, the disk will end up in a state in which only part of the write was completed or only part of the delete was completed — the file still exists, but it's smaller than it was. The file system is still consistent, but the user or application must check after the system comes back up to see how far the operation actually got.

This approach — performing multi-step file-system updates so that the on-disk file system never has anything worse than innocuous inconsistencies — is called the *consistency-preserving approach*. It's the basis of the soft-updates technique we describe below. Another approach, known as the *transactional approach*, treats a sequence of file-system updates as *atomic transactions*. This means that if the system crashes during such a sequence, the effect (perhaps after some recovery operations) is as if the operations either took place completely or didn't happen at all. Thus there's no such thing as a partially completed operation, though there is some uncertainty about whether or not an operation actually happened.

Transactions are standard fare in databases, where there is much concern that a sequence of steps have a well defined effect regardless of whatever else may be going on, including system crashes. For example, what happens when I transfer $1000 from one bank account into another? The expected outcome is clearly that the first account, assuming it has sufficient funds, ends up with $1000 less than it started and the other ends up with $1000 more.

[3] Note that by "recover from crashes" we mean bringing the file system's metadata to a consistent state. As discussed above, file systems generally make no attempt to make data consistent after a crash, and recent updates can disappear because of a crash.

A more complete definition of transactions is that they have the "ACID" property:

- *Atomic*: all or nothing. Either all of a transaction happens or none of it does. For example, if the system crashes in the middle of my $1000 transfer, I don't want to end up with the money taken out of the first account but not deposited in the other. At some point in the progress of a transaction we say it is *committed*: the full effect of the transaction is applied to the database or file system if the system crashes after this point, and no part of the transaction is applied if it crashes before this point.
- *Consistent*: transactions applied to a consistent database or file system leave it in a consistent state. In the example, regardless of whether or not the system crashes, my bank-account information should still be well formed.
- *Isolated*: a transaction has no effect on other transactions until it is committed and is not affected by any concurrent uncommitted transactions. This means that if I have $50 in my bank account and someone is in the act of transferring $1000 into it, I can't transfer the $1000 out until the first transaction commits. This is important since the first transaction might not commit — it could be aborted so that no money is deposited into my account. This requirement might also be construed as meaning that (again assuming my account balance is $50) it would be OK to make two concurrent transactions, each transferring all $50 out of it. However, this would violate the consistency requirement — a negative balance is not well formed.
- *Durable*: once a transaction has fully taken place, its effect on the database or file system persists until explicitly changed by some other transaction. If I deposit $1000 in your bank account, the money shouldn't spontaneously disappear sometime later.

Implementing transaction processing may seem at first glance a tall order, but, at least conceptually, it's not all that complicated. There are two basic strategies. In *journaling,*[4] the steps of a transaction are first written onto disk into a special journal file. Once the steps are safely recorded, the operation is said to be *committed* and the steps are applied to the actual file system. If the system crashes after the operation is committed but before the steps have been fully applied to the file system, a simple recovery procedure takes place when the system comes back up in which the steps are applied from the journal so as to perform the operation fully. This is known as *redo* or *new-value journaling*. If the system crashes after the operation has started but before all the steps have been recorded in the journal, crash recovery simply disposes of those steps that were recorded and the file system is left unmodified.

A variant of this approach, known as *undo* or *old-value journaling*, involves storing in the journal the old versions of blocks to be modified. Once these are safely on disk, the file system is updated. If the system crashes before this update is completed, the file system is restored to its previous state by writing back the contents of the journal.

A combination of both variants is used in some systems (for example, NTFS): both the old and the new values of the items being modified are put in the journal. Section 6.6.4 discusses why NTFS employs this combination.

In both redo and undo journaling, it might seem that we're doubling the amount of work: blocks are written not only to the file system, but to the journal as well. However, the techniques discussed in Section 6.1.4.2 to improve file-system performance — writing large amounts of data to contiguous blocks — can minimize the time required for this extra work. So, rather than write each item to the journal as it is produced, we delay until a fair number can be written at once, and then write them to contiguous locations. Of course, we must keep track of when we've written items to the journal and make sure we don't update the file system until afterwards.

[4] This is also called *logging*; we prefer *journaling* so as to avoid confusion with log-structured file systems.

What should transactions be in a file system? In other words, how do we decide where one transaction ends and another begins? One possibility is that each file-system system call is a separate transaction. This would be convenient, providing a well defined notion that makes sense to the application programmer, but would make it difficult to batch the writing of updates to the journal. The problem is the wide range in the number of disk operations used in system calls — writing a small number of bytes to an existing location within a file might take one operation, but deleting a large file could require millions of operations. Most system calls involve a small number of disk operations and thus their transactions would have so few steps that we'd get little benefit from batching their steps when journaling them. Others would be so large that the journal might not have enough space for all the steps of their transactions. What's generally done is a compromise: group together a number of short system calls into a single transaction, but split a long system call into a number of transactions. In all cases the transactions obey the ACID rules and, in particular, take the file system from consistent state to another.

The alternative approach to journaling is known as *shadow paging*. With shadow paging, new space is allocated within the database or file system to hold the modifications, but the original versions of the modified items are retained as well. The new versions are not integrated into the database or file system and the old versions continue to be part of it until the transaction is committed. At this point, a single write to disk effectively removes the old versions and integrates the new versions.

Consider a file-system example (Walker, Popek, et al. 1983). Suppose we are modifying a block of a file in an S5FS- or FFS-like file system. We allocate a new block, copy the contents of the old block into it, and then write the new data into it. If the original block is referred to by an indirect block, we allocate another new block, copy the original indirect block into it, and set it to point to our new data block. We then ensure that the changes are written to disk.

At this point, if the system were to crash, the effect would be that this update transaction hasn't happened, since the changes haven't been integrated into the file system. But now we can do this integration with a single disk write: we modify an in-memory copy of the block containing the file's inode so that it points to the newly modified indirect block, and then write the copy to disk, overwriting the original copy. This single write effectively removes the old information from the file system and integrates in the new. Some background cleanup may have to be done to add to the free list the disk blocks that were orphaned by a crash.

Why choose one of the transaction approaches over the other? An early comparison of the two (Brown, Kolling, et al. 1985) argues in favor of journaling: the argument is that, even though journaling requires more disk operations than shadow paging, most operations are either reads that are normally satisfied from the cache or writes that go to contiguous locations in the journal file. Shadow paging requires fewer disk operations, but these operations require more time. We refute this argument in Section 6.2.2.3 below.

6.2.2.1 Soft Updates

At first glance, it would seem that implementing the consistency-preserving approach should be straightforward: simply make sure that writes from cache to disk are done in the same order as the cache was updated. This would be easy if each metadata item occupied a separate block on disk. However, these items are typically small and thus are stored many per disk block so as to maximize the use of disk bandwidth. For example, in S5FS and FFS, multiple inodes are packed into a single block; directories consist of many blocks, but each block consists of a number of entries (each containing a component-name/inode-number pair). Of two cache pages, for instance, each containing a copy of a disk block, one might contain a metadata item that must be written out before a metadata item in the other, and vice versa. Thus there could well be circular dependencies on write order among cache pages.

Before we discuss how to handle these circular dependencies, let's reexamine the ordering constraints for changes to metadata. We must ensure that at all times the metadata on disk forms a consistent file system containing all the files and file data that applications believe should have been there at some point in the very recent past. Thus, assuming we start with a consistent file system on disk, after each write to disk it should still be consistent.

An easy way to prevent circular dependencies is the following: when two metadata items are to be modified and one must be written to disk before the other, write out the first synchronously (i.e., wait for the disk write to complete) immediately after modifying it; then modify the other but leave it in the cache, to be written out sometime later.

For example, Figure 6.23 shows what must happen when a new file is created and added to a directory. The file's inode is initialized, the data block containing the new directory entry is modified, and the directory inode is modified. Assuming each modification is in a separate block, all three blocks must be written to disk. The correct order for the writes would be first the new file's inode, then the directory entry, then the directory's inode. However, if the directory's inode and the file's inode share the same disk block (Figure 6.24), there is a circular dependency: the block containing the inodes must be written first, since it contains the file's inode, but it also must be written last, since it contains the directory's inode. Thus if both the two inodes and the data block are modified in the cache but are unmodified on disk, it is not possible to update the disk in a consistency-preserving order. To avoid this problem, a thread performing such a sequence of operations must, before updating the directory inode in the cache, first add the file inode to the cached block and wait for it to be updated on disk, and then wait for the data block to be written to disk.

This approach was used in early versions of FFS and provided good crash resiliency. But this came at the cost of significant performance loss due to the synchronous writes: the loss of performance in FFS due to such synchronous writes was between a factor of six and a factor of twenty over a version of FFS called Async FFS in which all updates of metadata are deferred and can be done in any order (Ganger, McKusick, et al. 2000).

Eliminating circular dependencies without requiring synchronous writes involves somehow keeping track of the metadata updates individually so they can be written out in the correct order. The *soft updates* technique does this while still maintaining the most recent updates in the cache (Ganger, McKusick, et al. 2000), so that the cumulative effect of changes to files appears in the cache, while the individual updates are listed separately. In this approach, when a cache page is to be written out, any updates that have been applied to it but must not be written out yet due

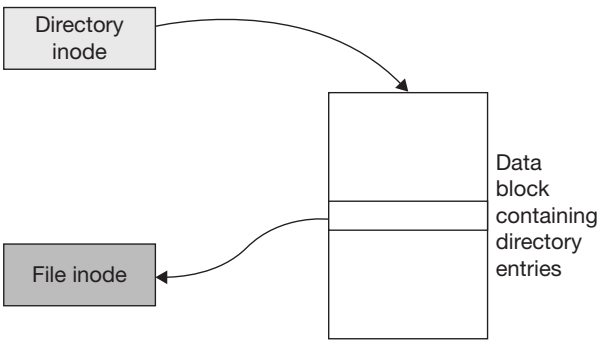


FIGURE 6.23 A new file is created in a directory: the file's inode is initialized, the data block containing the new directory entry is modified, and the directory inode is modified. All three modifications must be written to disk.

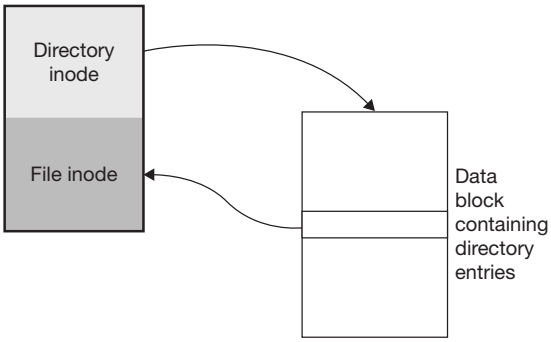


FIGURE 6.24 Here the directory inode and the file inode share the same disk block. Thus it is impossible to write the modifications out so that the file inode is written before the directory-entry block, yet have the directory-entry block written before the directory inode — we have a circular dependency.

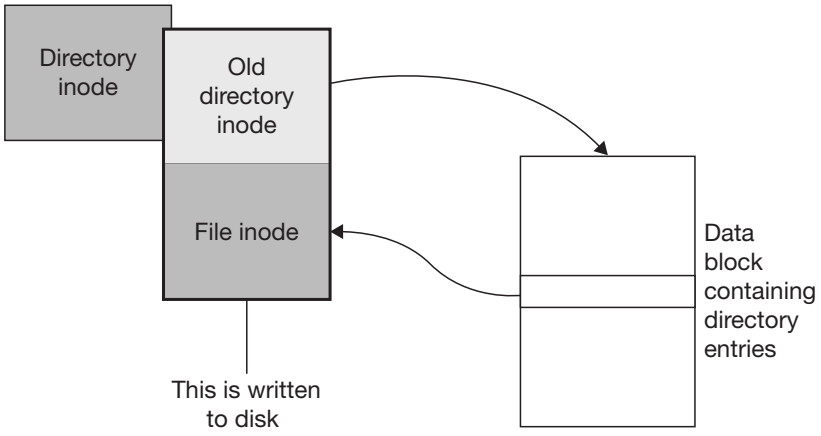


FIGURE 6.25 In the soft-updates approach, the circular dependency is broken by retaining the old value of the directory inode and including it in the block when the new file inode is written out.

to ordering constraints are undone: the cache page is restored to its state before the update was applied. The page is then written out to disk; the update is redone on completion. While the page is in this undone state, it's locked so as to prevent access until the updates are redone. Using this technique on FFS yielded performance almost as good as that of Async FFS, yet with the benefits of correctly ordered updates. Figure 6.25 shows how the circular dependency is broken.

(Ganger, McKusick, et al. 2000) identified three general problems necessitating ordering constraints in the cache.

1. The system might write out a new value for a metadata item so that it points to an item that isn't properly initialized on disk. A crash would thus make the just-written item point to garbage. The pointed-to item should have been written out first, as in our linked-list example. *Thus a constraint is added that the pointed-to item must be written out first.*
2. The system might modify a pointer in a metadata item so as to orphan an item that shouldn't be orphaned. That is, it might remove the last pointer to an item containing data still in use. A crash would cause the permanent loss of the orphaned item, since nothing on disk refers to it. This might happen when we're renaming a file and remove the on-disk directory entry containing the file's previous name before adding an on-disk directory entry with the new name. *A constraint is added specifying that the new directory entry must be written out before the old directory entry.*
3. The system might write out a new value for a metadata item for which on-disk items still point to and depend upon its old value. This could happen in FFS if, in the cache, we delete a file and reuse its inode for a new file, and the new inode contents are the only thing written to disk. Thus there are directory entries on disk for the old file that refer to the new file's inode. A crash would cause the replacement of the old file's contents with those of the new file. *A constraint is added specifying that metadata items containing pointers to the old use of the metadata item must be written out (with null values) before the new contents of the metadata item are written out.*

Thus, subject to the above constraints, the writing back to disk of cached file-system updates can be deferred. And, as explained above, if a single cache page contains a number of updated metadata items, it might be written back several times so as not to update a particular metadata item on disk before its constraints allow.

The soft-updates technique has a few drawbacks. More disk writes may be necessary than with Async FFS so as to maintain ordering constraints, and extra memory is required to represent the constraints and to hold old metadata values. (Ganger, McKusick, et al. 2000) found these problems to be minimal. In addition, after a crash, some disk blocks may be free but not accounted for as free space. While these blocks must be reclaimed, this can be done as background activity after the system restarts.

6.2.2.2 Journaled File Systems[5]

Many file systems use journaling for crash tolerance. Many of these, such as Ext3 and VxFS, use redo journaling. NTFS, for reasons discussed in Section 6.6.4 below, uses a combination of both redo and undo journaling. We use Ext3 to introduce journaling: it's essentially the same file system as Ext2 but with journaling added, much as soft updates were added to FFS. Recall that Ext2 is similar to FFS.

Ext3 uses journaling to protect metadata and, optionally, user data as well. Its approach is to treat all operations on the file system in a specific time interval as one large transaction. As with Ext2 and FFS, disk blocks are cached in memory. No modified blocks are written to the file system or the journal on disk until the transaction is about to commit. A special journal thread performs the commit, writing all modified blocks to the journal. Once they are safely copied to the journal, a *checkpointing* phase begins in which the file system itself is updated: the cached blocks are released to the normal buffer-cache writeback mechanisms (as in Ext2 and the original FFS), which write them to the file system in due course. As soon as all modified blocks have reached the file system, the copies of them in the journal are no longer needed, and the space they occupy is released.

Lots of details need to be filled in to get a really clear understanding of how Ext3 works. A major issue is whether user data is journaled along with metadata. Let's proceed by assuming at first that it is, and then examine what must be done if not. The portion of Ext3 that does journaling is split off into a separate subsystem known as JFS that makes the journaling code usable by other subsystems as well. Thus JFS provides the mechanism for journaling, while Ext3 proper determines what is to be journaled.

The example in Figures 6.26 through 6.28 shows what a transaction consists of. Ext3 provides JFS with what we call subtransactions — sequences of operations that take the file system from one consistent state to another. JFS determines how many of these subtransactions to group into a transaction. Note that no distinction is made as to which thread or process produced the subtransactions; JFS simply groups them together until it has a large enough transaction or until a time limit (five seconds, say) is exceeded. A separate kernel thread, known as *journald* (the *journal daemon*), then wakes up and does the commit processing: it writes the updated blocks of the transaction to the journal. Note that transactions are formed sequentially in time: all subtransactions within the JFS-determined transaction interval are grouped into the same transaction. A new transaction is started as soon as the previous one begins to commit.

In the terminology of the Ext3 implementation, the subtransactions are referred to via *handles* provided by JFS. For example, when Ext3 performs a *write* system call, it splits the transfer into separate pieces along block boundaries. As it processes each piece, it obtains a handle from JFS to refer to the piece. Thus the file-system updates required to add one block to the file system form a subtransaction, or at least part of one: JFS can arrange to group subtransactions together by supplying the same handle for the second piece of a single *write* system call as for the first piece, and similarly for subsequent pieces. Thus one handle can refer to a subtransaction forming some or all of the steps of one system call. JFS combines any number of these subtransactions into a single transaction.

[5] Some of the material in this section is based on a presentation by Stephen Tweedie, the developer of ext3. See (Tweedie 1998).

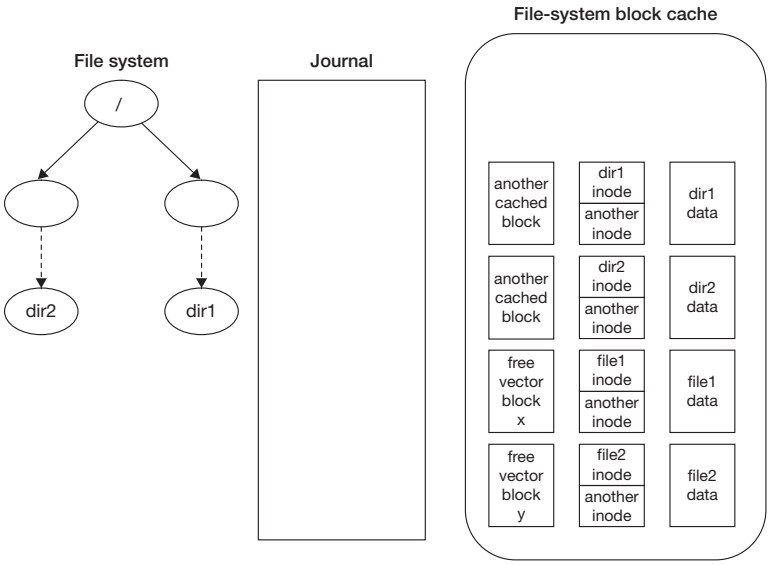


FIGURE 6.26 Journaling in Ext3, part 1. One process has created *file1* in directory *dir1*, another has created *file2* in directory *dir2*. The *journald* thread has yet to begin commit processing, so these operations have so far affected only the cache. Each process has performed two system calls: one to create the file, another to write data to it. Creating a file involves making an entry in the parent directory and allocating an inode. Note that a disk block contains more than one inode. Writing to a file involves allocating a disk block and modifying the inode's disk map. Thus the cached blocks containing directory data, the files' inodes, the files' data, and portions of the free vector are modified. Let's assume that JFS has grouped these into two subtransactions, one for the steps required to create and write to each file. These modified cached blocks must stay in the cache and must not be written to the file system until they've been copied to the journal.

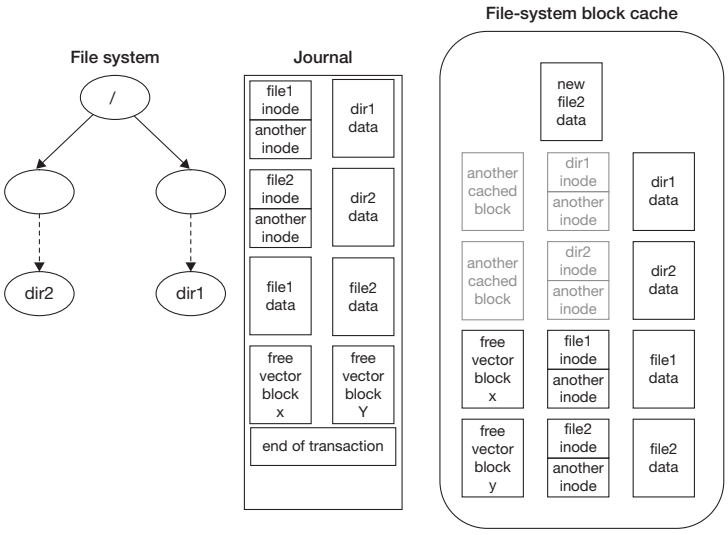


FIGURE 6.27 Journaling in Ext3, part 2. The *journald* thread has written the modified cache blocks to the journal, thus committing the transaction. Note that the unmodified cached blocks can be removed from the cache (i.e., their storage can be reused for other items). However, in the meantime, new data has been written to *file2*. Since the original copy of its data block is involved in the earlier transaction, a copy of that block is made for a new transaction that will include this update.

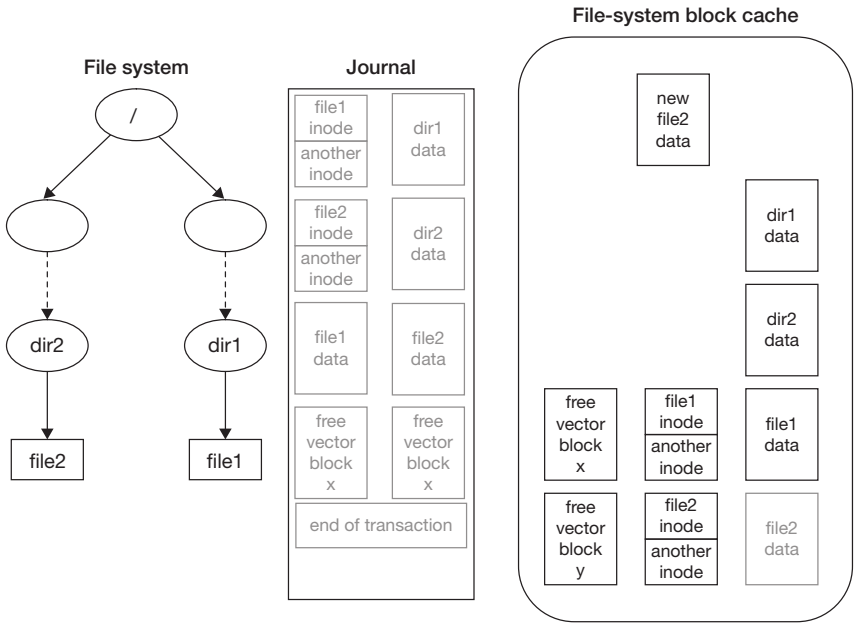


FIGURE 6.28 Journaling in Ext3, part 3. The transaction has been checkpointed, i.e., the cached blocks have been written to the file system, and thus it's now safe to remove the transaction from the journal. The old *file2* data is removed from the cache, since it's superseded by the new data.

The steps of the subtransactions (and hence transactions) are represented by the new contents of blocks and are stored in memory as part of the usual file-system block cache. Thus file-system updates are applied to the blocks in the cache. These modified blocks are written out to the journal as part of commit processing, and are later written to the file system as part of checkpointing. Normally, there's at most one copy of any particular disk block in the cache. Thus if a block contains a number of metadata items, their updates are effectively batched: the block is written out only once each to the journal and the file system, even if multiple items have been modified. This is particularly important for the free-space bitmap, in which each bit in a block is a separate metadata item.

As we've mentioned, when JFS determines it has reached a transaction boundary, it closes the current transaction and wakes up the *journald* thread to begin commit processing. This thread writes all the modified disk blocks to the journal and then writes a special *end-of-transaction record* to make it clear that the entire transaction is in the journal. It then starts checkpointing by linking together the cached copies of the modified blocks in the dirty-block list of the buffer cache, so they'll be eventually written to the file system. Each of these blocks refers back to the transaction it was involved in, so once they all have been written to the file system, the journal space their copies were occupying can be freed.

What if the contents of a block involved in the previous transaction are modified by the current one before it is written to the journal as part of committing the previous transaction? If there is only one cached copy of the block, we have a problem: what is to be written to the journal should be the contents of that block as it was at the end of the previous transaction, yet we also want to modify it to contain the block's new value. To get around this dilemma, Ext3 makes a copy of the block for use by the *journald* thread to write to the journal. The original cached copy can then be modified as part of the current transaction.

The journal contains a sequence of disk-block copies along with their location in the file system. When the system reboots after a crash, recovery involves simply finding all committed transactions (that is, ones with commit records at their ends) and copying their modified disk blocks from the journal to the indicated positions in the file system. If the system crashes in the middle of recovery, no harm is done: copying a disk block to the file system is *idempotent*, meaning that doing

it twice has the same effect as doing it once. When the system reboots the second and perhaps subsequent times, the blocks of the committed transactions are simply copied to the file system again.

By journaling both metadata and user data, we're assured that after crash recovery the state of the file system is what it was at the end of the last committed transaction. But journaling user data could cause a significant slow-down, since everything is written to disk twice. So, what happens if only metadata is journaled?

Suppose we've just created a new file and have written a few million bytes of data to it. The transaction containing the new metadata for this file has committed, but the system crashes before any of the file's data has been written to the file system. After the system comes back up, the file's metadata is recovered from the journal. This metadata correctly refers to the disk blocks that should be holding the file's data, but the data never made it to disk. Thus what we find instead are the previous contents of these disk blocks, what they contained before they were last freed. They might well contain sensitive information from someone else's file — information we are not supposed to be able to read.

Unless your goal is to be an information thief, this is not good. We've got to make sure that the old contents of these disk blocks are not visible when the blocks are reused. Some extremely secure systems write zeros to disk blocks when they are freed to make certain there is no information leakage. Since this would slow down the file system further, we'd like to avoid it. Instead, let's simply make certain that the new data is written to the disk blocks before the metadata is committed. So, in Ext3, all the cached copies of newly allocated data blocks related to the current transaction are written to disk by the *journald* thread before it writes the metadata to the journal. Thus if the system crashes as in the scenario described above, after crash recovery the data blocks referred to by the recovered metadata will contain their intended new contents, not their prior contents.

We've solved one problem, but have we created others? We're trying to get the effect of journaling user data without the expense of writing it to the journal — we're simply writing it to the file system. However, suppose the system crashes after we've written the user data to the file system, but before the metadata has been committed in the journal. Since the transaction didn't commit, none of the metadata changes appear in the file system after crash recovery. However, the changes to user data, having been applied directly to the file system, do appear.

So, suppose just before the crash we deleted file A and then created file B and wrote some data to it. As it turns out, the data blocks that were freed when we deleted A were allocated to hold the new data for B. Unfortunately, the system crashed before the transaction containing the operations on A and B could be committed. So, after crash recovery, A still exists and B was never created. However, the data we wrote to B did make it to disk — it resides in the disk blocks that are now still part of file A. For those interested in information thievery, this is further good news.

To avoid this problem, Ext3 allocates new blocks from only those blocks that were free in the most recent committed version of the free vector. Thus a block that has been freed in the current, uncommitted transaction is not a candidate for reallocation until the transaction commits.

But there's another problem, shown in Figures 6.29 through 6.32. Suppose we create a file X in directory A. We then delete X as well as the entire directory A. These operations end up in the same transaction, which is committed but not yet checkpointed.[6] We now create a new file Y and write to it. The data block allocated to hold the data used to be a metadata block[7] containing the entry for file X in directory A — it was freed when A was deleted. The transaction containing this operation on file Y is committed and thus the new contents of Y were written to the file system beforehand. Furthermore, checkpointing of this transaction completes before that of the first transaction and thus Y appears on disk in the file system.

[6] Remember that at most one copy of a cached block is in the journal. Thus when a transaction involves multiple operations on the same block, only the final contents of the block appear in the journal.

[7] Note that the contents of directories are metadata, despite residing in data blocks. Thus changes to directories are journaled even though the data blocks of non-directory files are not.

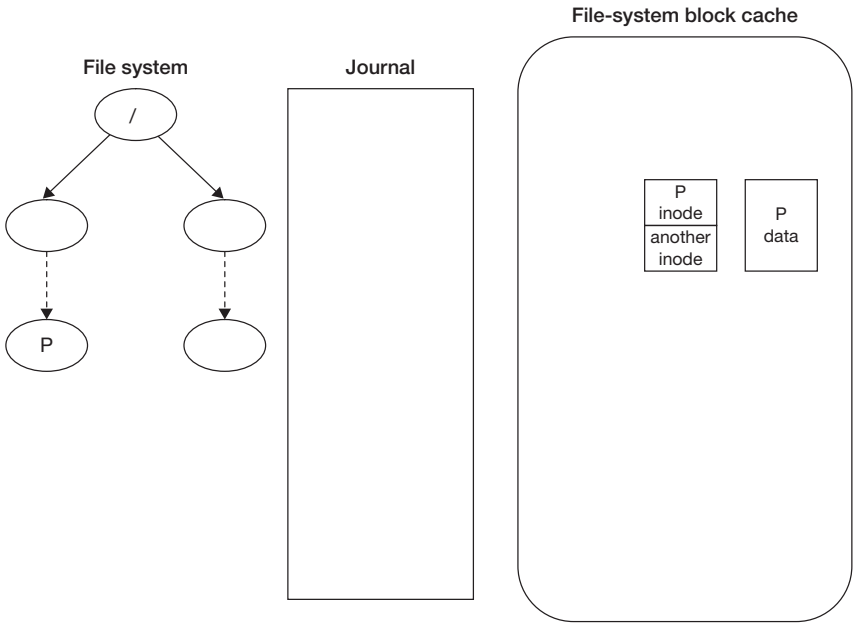


FIGURE 6.29 The need for revoke records in Ext3, part 1. Directory P is empty and we're about to create a subdirectory within it.

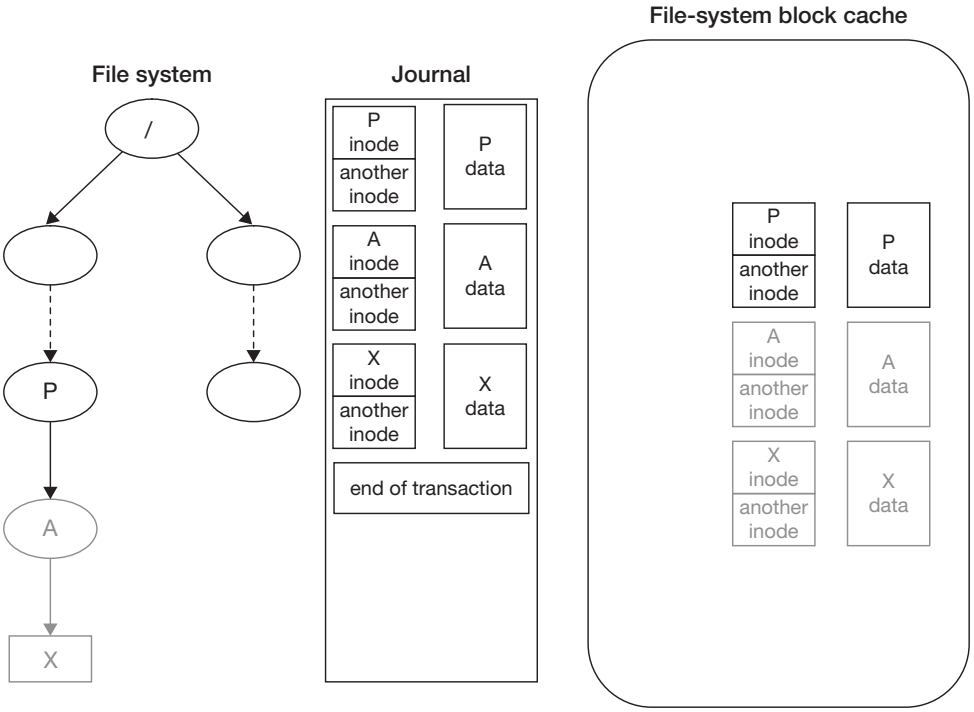


FIGURE 6.30 The need for revoke records in Ext3, part 2. Directory A was created and file X created in it, then both were deleted. These operations become part of the same transaction, which is committed but not checkpointed. However, since it was committed, the blocks occupied by A and X are added to the free vector and available for use.

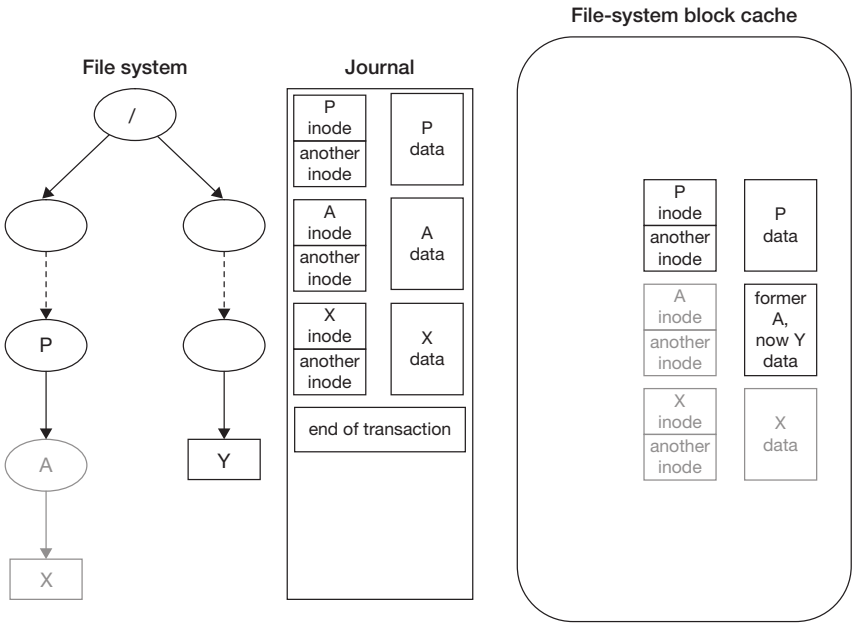


FIGURE 6.31 The need for revoke records in Ext3, part 3. File Y is created and written to. Its data block is the one that formerly held the contents of directory A.

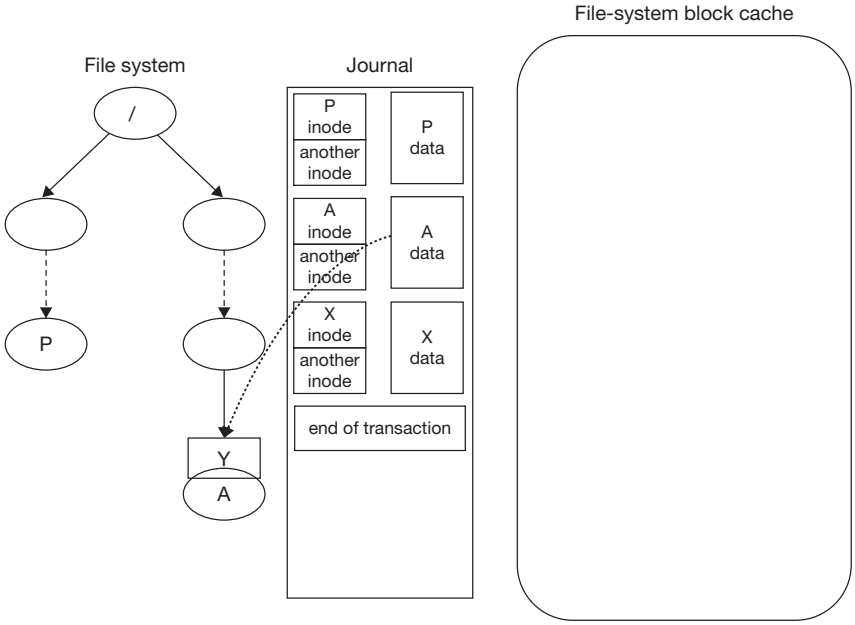


FIGURE 6.32 The need for revoke records in Ext3, part 4. The system has crashed and come back up. The contents of the journal are applied to the file system. In particular, the journaled contents of the block that originally contained the entries of directory A are copied back into their original location, which now is the first data block of Y. Thus Y's previous contents are destroyed.

Now the system crashes. Since checkpointing of the first transaction wasn't completed before the crash, the journal still contains its components and in particular the metadata block from directory A (with the entry for X deleted). Even though this block is subsequently deleted, when the journal is played back as part of crash recovery, the journaled contents of this block are written to the file system, overwriting the data that was written into file Y. Thus when the system returns to normal operation, everything appears fine except that file Y has in it, not the data that was written to it, but what used to be the metadata block from the old directory A.

So a problem again arises from our attempt to avoid journaling user data. If we were journaling user data, this problem clearly wouldn't happen: the journal would contain the new data written to Y, which would be copied to the file system during crash recovery after the old metadata is written there.

We can attack this from a different angle: the problem occurs because we've modified and then deleted a metadata block. Since the metadata block is being deleted, there's no real point to keeping the earlier entry containing the modified block in the journal. However, rather than actually deleting this entry, Ext3 writes a special *revoke* record to the journal indicating that the earlier metadata update should be ignored during crash recovery.

It's a bit disconcerting, though, to assert that something is correct merely because we've dealt with all the problems we can think of. Short of a formal proof, let's try to be a bit more convincing that Ext3's metadata-only journaling actually works. What do we mean by "works" in this case? We certainly want the file system's metadata to be in a consistent state after a crash. This would be guaranteed if all writes to disk were done as part of transactions, but, of course, writes to user data are not. If any particular disk block were always used for user data or always used for metadata, it would be clear that the consistency of metadata is protected by the transactions, but we know that one block might be used for metadata at one moment and for user data an instant later. However, if all blocks being used as metadata within a transaction are modified only via the journal during the transaction, we're assured that if the system crashes before the transaction commits, then the blocks will revert to their pre-transaction contents. But if a block is used and modified both as user data and metadata within the same transaction, then some changes to it are applied directly to the on-disk file system, while others are applied via the journal. By allocating new blocks from only those free at the end of the previous transaction, we make certain that no block is used both as user data and metadata in the same transaction.

What can we say about the state of user data after crash recovery? Since it's not journaled, we can't say it's consistent. However, we would like to say that the contents of any user-data block are relatively recent and were written there by the application program. In other words, we'd like to say that no opportunities for information thievery exist. Thus user-data blocks are initialized before they are linked into a file, and once a disk block is allocated to hold user data, the only writes to it are those containing data provided by the application program. By writing the contents of newly allocated user blocks to disk before the current transaction is committed, we guarantee their proper initialization. The revoke records guarantee that nothing in the journal will overwrite a user-data block during crash recovery.

How does journaling compare with soft updates? (Seltzer, Ganger, et al. 2000) reports that neither is a clear winner over the other.

6.2.2.3 Shadow-Paged File Systems

Shadow-paged file systems are a cross between consistency-preserving and journaled file systems: at all times the on-disk file system is consistent, yet operations are performed in a transactional fashion. Such systems, also called *copy-on-write file systems*, are simpler to explain (and implement!) than the metadata-only journaling used in systems such as Ext3. We examine the approach used in both Network Appliance's WAFL (Write-Anywhere File Layout) and Sun's ZFS (Zettabyte File System — so called because it can handle storage of size greater than 10^{21}

bytes, or a *zettabyte*). An additional feature of shadow paging is that it makes possible convenient "snapshots" of the past contents of a file system.

To accomplish all this, all in-use blocks of the file system are organized as one large tree — the *shadow-page tree*. The root of this tree (called the *überblock* in ZFS and *fsinfo* in WAFL) refers to the inode of the file containing all the inodes (see Section 6.1.5), which in turn refers to all the inodes currently in use, each of which refers, perhaps via indirect blocks, to the contents of all the files — see Figure 6.33. When a node of the tree (a disk block) is to be modified, a copy is made and the copy is modified — the original stays as it was (Figure 6.34). To link the modified copy into the tree, the parent node must be modified so as to point to it. But rather than directly modify the parent, it too is copied and the copy is modified. To link it into the tree, its parent must be modified; this copy-on-write approach continues all the way up to the root (Figure 6.35).

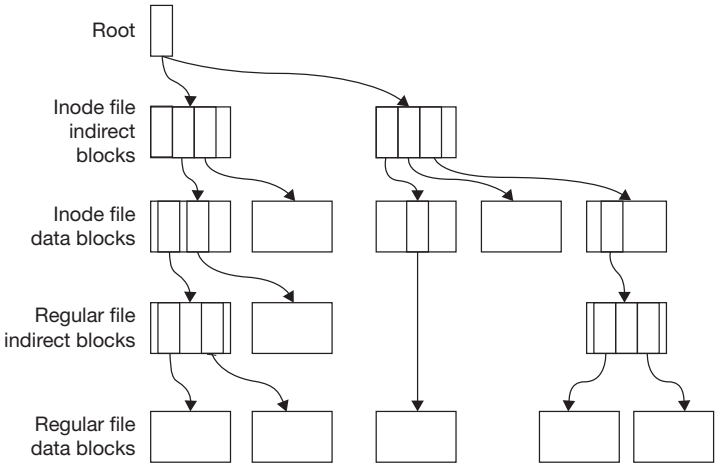


FIGURE 6.33 Shadow-page tree.

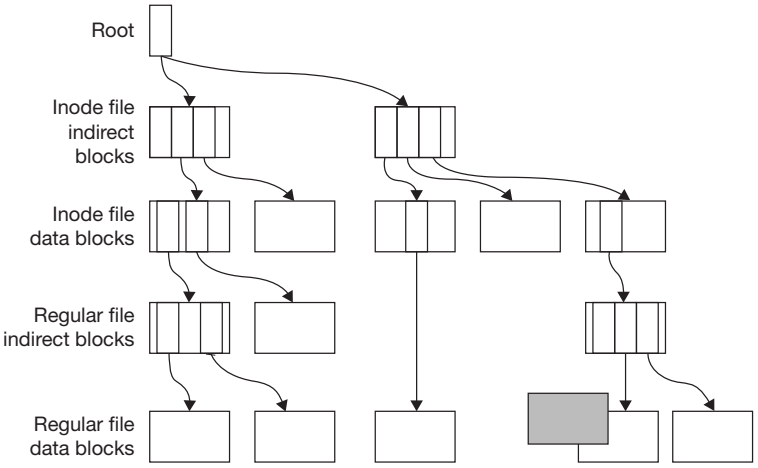


FIGURE 6.34 A leaf node in a shadow-page tree is modified, step 1.

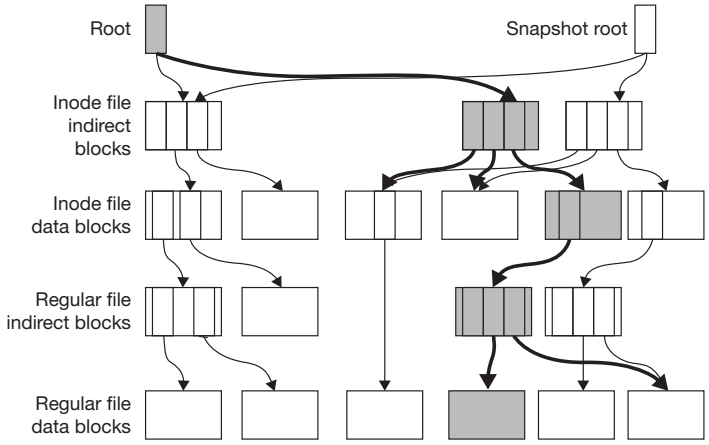


FIGURE 6.35 A leaf node in a shadow-page tree is modified, step 2. Copies are made of the leaf node and its ancestors all the way up to the root. A copy of the old root is maintained, pointing to a snapshot of the old version of the file system.

The root itself is at a known location on disk and is modified directly, in a single disk write. Thus, note two things:

1. If the system crashes after any non-root nodes have been modified but before the root is modified, then when the system comes back up the root refers to the unmodified file-system tree. No changes take effect on disk until the überblock is modified. Thus we have a transaction — the modified copies of the nodes are the shadow pages.
2. A copy of the unmodified root refers to the file-system tree as it was before the modifications took place. Thus we have a snapshot of the earlier state of the file system. Such snapshots can be kept around to let us recover inadvertently deleted files and can also provide a consistent copy of the file system for backup to a tape drive.

Note also that the transactions don't require extra disk writes, as the journaling approach does. As mentioned in Section 6.2.2 above, (Brown, Kolling, et al. 1985) argued that though fewer disk writes are required, those that were necessary were expensive because they are not to contiguous locations. But file-system technology has improved greatly since 1985. In particular, log-structured file systems have shown that we can simply group a bunch of seemingly unrelated file pages together and write them as a sequence of contiguous blocks. This and other techniques are used in both WAFL and ZFS to provide good performance, as we discuss in Sections 6.6.5 and 6.6.6 below.

6.3 DIRECTORIES AND NAMING

Naming in file systems is pretty straightforward: files are named by their path names in a basically tree-structured naming hierarchy. We could certainly think of other naming techniques — for example, names could be completely unstructured with no notion of directories — but the organizing power of directories has proven itself over the past few decades and seems here to stay.

We note briefly that file systems are not database systems. File systems organize files for easy browsing and retrieval based on *names*. They do not provide the sophisticated search facilities of database systems. The emphasis in file systems is much more on efficient management of individual files than on information management in general. Thus, by database standards, the organization of file systems only on the basis of file names is pretty simplistic. But implementing such naming well is extremely important to the operating system.

The implementation of a file-system name space has all the requirements of the rest of the file system: it must be fast, miserly with space, crash tolerant, and easy to use. The key components are directories and various means for piecing different name spaces together — that is, the notion of

connecting one directory hierarchy to another, such as the mounting that takes place in Unix. File-system naming can be extended into other domains and applications. Unix extends such naming to devices and processes (see Section 6.3.2.2). And, as we explain below (Section 6.3.2.1), NTFS's notion of *reparse points* allows naming to be extended to non-file-system applications.

6.3.1 DIRECTORIES

We begin with directories. Their role is clearly pretty important. Every time we open a file, we must follow its pathname through a sequence of directories, looking up successive components in each. Creating a file involves verifying all but the last component of the pathname, then inserting this last component into the last directory. Deleting entails the lookup work of opening a file, then removing the last component from the last directory. Listing the contents of a directory is another common operation.

As we did with file systems in general, let's start with a simple design intended for small systems: S5FS's directories. We've seen them already, in Section 6.1.1. All entries had the same fixed length and were added sequentially in the order in which they were created. Directories were searched sequentially — this wasn't a performance problem, since most directories were rather small. Listing the contents of a directory was straightforward: you simply read through the directory sequentially, enumerating its contents and sorting them if required.

Deleting directory entries was a bit of a problem if it was subsequently necessary to compact the now smaller directories and free unneeded space. Deleting one directory entry was easy: the slot was simply marked free and made available for the next entry to be added. But suppose you create a directory and insert two disk blocks' worth of entries into it. You then delete half the entries in the first block and half the entries in the second block. To give that space back to the file system, it would be necessary to copy the active entries from one of the blocks into the free space of the other. Doing this so as to be immune from any problems resulting from an untimely system crash was considered too difficult back then. The easy way to avoid problems from crashes was to avoid doing anything complicated. Thus no directory space was given back to the file system unless the entire directory was deleted. Since the space for deleted entries was reused when new entries were added, this simple approach normally worked fine.[8]

The first version of FFS, in 4.2BSD Unix, made one major change to the simple S5FS directory format: variable-length entries were used to support component names much longer than the fourteen-character maximum of S5FS (see Figure 6.36). Everything else was the same: directories were searched sequentially. Since entries were now of variable length, they were added using a first-fit approach; a new block was appended to the end if more space was needed. Directories were still not compacted after entries were deleted, and space was not released to the file system unless the entire directory was removed. To help further with crash tolerance, though directory entries were of variable length and successive entries were in successive locations, no entry was allowed to cross a 512-byte block boundary. This ensured that a directory could be modified with exactly one 512-byte (one-sector) disk write.

Despite or perhaps because of this simplicity, directory operations turned out to be a major bottleneck in 4.2BSD Unix. The operating-system kernel was spending a remarkable 25% of its time dealing with path names (Leffler, McKusick, et al. 1989). This performance issue was mitigated in the next release, 4.3BSD, through caching: recent component-name-to-inode

[8] FFS is certainly not the only file system whose directories have this problem. For example, storage for directory entries in the operating system used by NASA on its Mars Rovers is also not freed when entries are deleted. This property almost caused the demise of NASA's Spirit Rover (traversing Mars) in January, 2004. Its file system ran out of space due to deleted but unfreed directory entries. Fortunately, NASA was able to send it instructions to free the space. (From http://www.wired.com/news/space/0,2697,64752,00.html?tw=wn_tophead_3)

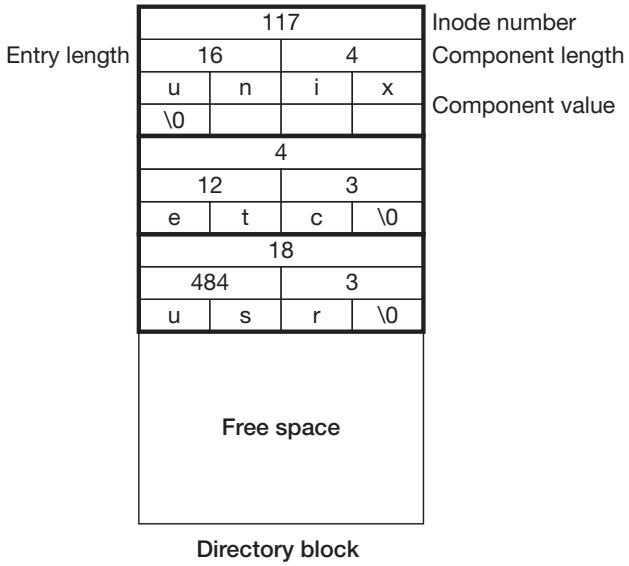


FIGURE 6.36 A 512-byte directory block in FFS. Each entry contains the inode number, the length of the path-name component (not counting the null character at the end), and the length of the entire entry. The latter was always a multiple of four bytes. If it was longer than necessary for the given component, then it was the last entry in the block and the extra space was free space.

translations were stored in a cache that was consulted before the directory itself was searched. This reduced the kernel time spent on path names to 10% — better but still fairly high.

The directory-related performance issues of BSD Unix make it clear that directory implementation plays a crucial role not just in file-system performance, but in overall system performance. Linear-searched, unordered directories simply aren't sufficient for any but the smallest systems. The directory in which I maintain the material for this book contains a few tens of entries — linear directories are fine. However, my department's mail server has over two thousand entries in the directory for everyone's incoming email. Every time a new message arrives for me, the mail-handling program would have to perform a linear search to find my mailbox if this directory were unordered. The comparable directory for a large Internet service provider (ISP) would be considerably larger. Adding an entry to linear directories requires linear ($O(n)$) time as well, meaning that the time required to create all the entries is quadratic ($O(n^2)$) — a search must be done to ensure each entry doesn't already exist as well as to find space for it. Similarly, deleting all the entries in a directory requires quadratic time.

To deal with these problems, we can continue with the strategy taken in 4.3BSD Unix and employ even better caching. We can also adopt strategies developed in the database world for providing efficient indexing for large databases by using better on-disk data structures. Both approaches typically employ more efficient search techniques and are based on either hashing or some variant of B trees (usually B+ trees, explained in Section 6.3.1.2 below).

The use of caches to improve directory performance has the advantage that no changes are required to on-disk data structures. Thus we can continue to use the current disk contents unmodified — which makes moving to a system with enhanced directory performance much more attractive. Furthermore, a cache-based system doesn't need to be crash-tolerant; the cache is simply recreated each time the system starts up.

But a cache-based system won't show any performance improvement the first time each directory is accessed after a reboot. In fact, some approaches might show considerably worse performance with the first access to a directory if the entire directory must be fetched so as to construct the cache. Furthermore, if we use a disk-based approach in conjunction with journaling or shadow paging, we'll get crash tolerance for free (after suitable integration). Thus we have a tradeoff between cache-based and disk-based approaches.

Almost all approaches appear in practice. Recent versions of FFS use an approach called *dirhash*, which is a cache-based system using hashing. WAFL also uses caching and hashing. ZFS uses disk-based hashing, while NTFS uses disk-based B+ trees.

Below we discuss first hashing, then B+ trees.

6.3.1.1 Hashing

We have a number of disk blocks at our disposal in which we'd like to create directory entries. So, given a name (say a component from some pathname), we want to determine in which block to create an entry for it. And, once we've done this, if we're given the name again, we need to find this block again so we can retrieve the entry. There might of course be multiple entries in a block. For now we'll perform a linear search within a block to find the entry we're after.

How do we determine which block to use? In hashing we use a *hash function* that computes an index from the name. Thus if h is our hash function and *comp* is the name, $h(comp)$ is the index identifying the block containing the entry for that name. In standard hashing terminology, all names that have the same image under h, i.e., are hashed to the same value, are called *hash synonyms*. The things indexed by these values, the disk blocks in our example, are known as *buckets* — all hash synonyms are thrown into the same bucket.

Choosing the right hash function is pretty important. We won't go into how this is done, but let's assume our hash function does what it's supposed to do — the hashes of the names are uniformly distributed across the buckets.

If we have a static environment in which nothing is added to or deleted from our directories, then hashing clearly allows us to search directories in constant time: a fixed amount of time to hash a name and identify the block the corresponding entry should be in, then a linear search of a fixed-size directory block. But how do we handle adding and deleting items? If we add an item and there's room for it in the appropriate bucket, it's easy — we simply put the item in that bucket. But what if the bucket (that is, directory block) is full? We could use an approach called chaining in which we create a new block and link it to the first. But if we continue with this approach we will have chains of unbounded length and our search times will be linear again. So what we really want to do is to increase the number of buckets and modify our hash function so that it hashes names to the new set of buckets.

One approach for doing this is *extensible hashing*. It uses a sequence of hash functions h_0, h_1, h_2, . . ., where h_i hashes names into 2^i buckets, and for any name *comp*, the low-order i bits of $h_i(comp)$ are the same in $h_{i+1}(comp)$. Thus if $h_2(\text{"}Adam\text{"})$ is 1, $h_3(\text{"}Adam\text{"})$ is either 1 (001 in binary) or 5 (101 in binary). Suppose at the moment we have four buckets and are thus using hash function h_2 (see Figure 6.37). We now add the name *Fritz*, which should go into bucket 2, but doesn't fit. To accommodate *Fritz* (and future additions), let's double the number of buckets. This requires a new hash function, h_3, which hashes into eight buckets. Then we take bucket 2 and rehash all the names that were in it. Assuming these names are reasonably well distributed, roughly half will stay in bucket 2 under h_3 and half will go into bucket 6. Since the other buckets haven't overflowed, we won't rehash and redistribute their contents until they do.

Now we have a problem: though we've switched from hash function h_2 to h_3, we've gone from four buckets to only five, rather than the eight required by h_3. We add a level of indirection to accommodate this — an array of *indirect buckets*. (This array of indirect buckets is usually called a *directory* in the hashing and database literature, but that name is too confusing to use in our context.) The value produced by the hash function is used to index into this array; each entry in the array points to the appropriate bucket. If another bucket overflows, we simply create a new bucket, rehash the contents of the old using h_3, and update the appropriate indirect buckets as shown in Figure 6.38.

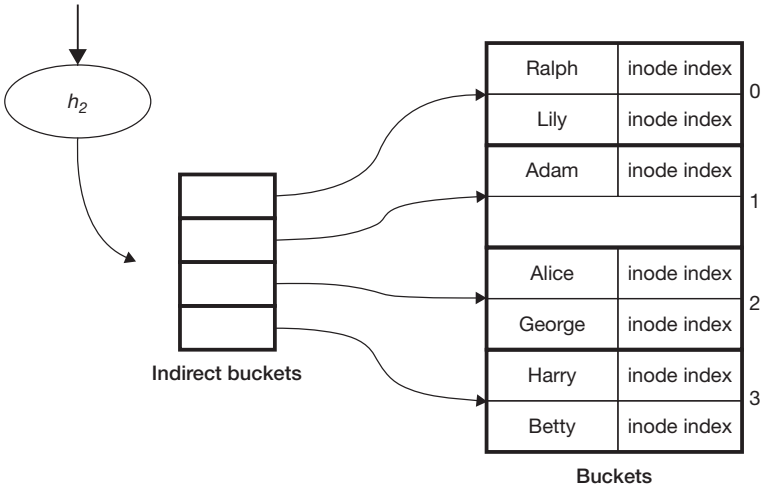


FIGURE 6.37 Extensible hashing example, part 1. Here we have four buckets and hence are using h_2, which actually computes indexes into the array of indirect buckets, which, in turn, lead to the appropriate bucket. Each of our buckets holds two items. We are about to add an entry for *Fritz*. However, h_2(*Fritz*) is 2 and the bucket that leads to it is already full.

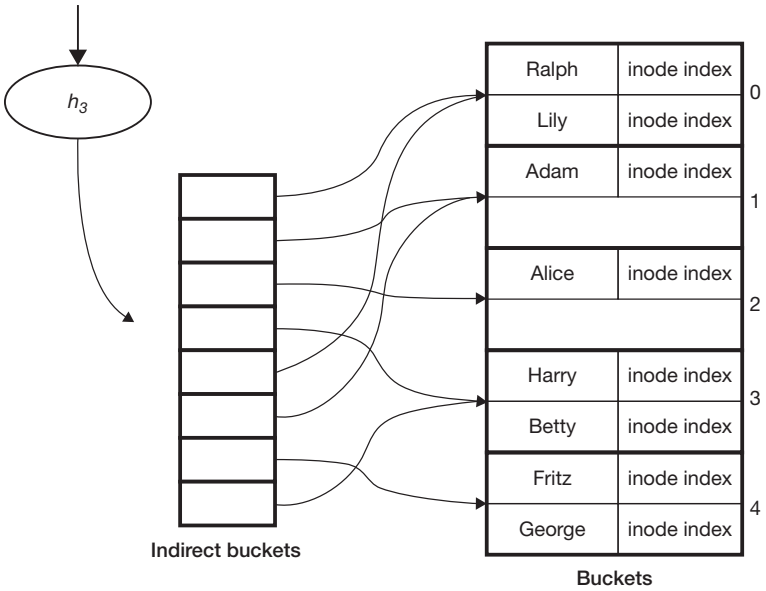


FIGURE 6.38 Extensible hashing example, part 2. Here we've added an entry for *Fritz* to the directory of Figure 6.37. Since the bucket *Fritz* was to go in under h_2 was full, we've switched to h_3, which maps names into eight buckets. However, rather than double the number of buckets and rehash the contents of all the old buckets, we take advantage of the array of indirect buckets. We double their number, but, initially, the new ones point to the same buckets as the old ones do: indirect buckets 0 and 4 point to bucket 0, indirect buckets 1 and 5 point to bucket 1, and so forth. For *Fritz*'s sake we add a new (direct) bucket which we label 4. We rehash *Fritz* and the prior contents of bucket 2 under h_3, with the result that *Fritz* and *George* end up in the bucket referred to by indirect bucket 6, while *Alice* stays in the bucket referred to by indirect bucket 2. Thus, we set indirect bucket 6 to refer to the new bucket 4. If, for example, we add another name that would go into bucket 0, we'd have to add another bucket to hold it and rehash the current contents of bucket 0.

The time required to add a new entry is now constant except when a bucket overflows. Coping with overflows is a bit expensive, but doesn't happen very often. Directory searches are made slightly more expensive by the addition of looking up hash values in the indirect buckets.

Deleting directory entries requires us to determine whether it's possible to combine buckets and thus save space. We leave the details of this to Exercise 15.

6.3.1.2 B+ Trees

Hashing works well if the hash function does a good job of hashing the actual names being used into uniformly distributed values. A different approach is to maintain some form of balanced search tree, thereby guaranteeing that searches can always be done in logarithmic time. The down side is that maintaining the data structure is a bit complicated.

The basic principle of how B+ trees work is fairly simple. Names are in the leaves of the tree and are in lexicographical order. Each level of the tree partitions the names into ranges; each level's partition is a refinement of its predecessor. Thus a search takes us from the root, which partitions the names into broad ranges, down to a leaf, which leads us to a narrow range or an individual name.

At each node the ranges are defined by a number of *separator names*. If the node has p ranges, then p-1 names are used to divide them. For each of the p ranges there is a link either to the root of a subtree that further divides the range or to a leaf. For example, if the node divides the space into three ranges r_1, r_2, and r_3, then all names in range r_1 are less than all names in r_2, which are less than all names in r_3. If the name we are after falls within range r_2, we follow the r_2 pointer. If it leads to the root of a subtree, then we compare our name with the subtree's separators and follow the pointer from the selected range. Otherwise if it points to a leaf, if the name we're after is present, it's in that leaf (see the example of Figure 6.39).

B+ trees must be *balanced* — that is, the distance from the root node of the tree to any particular leaf must be the same as that from the root to any other leaf. Another important aspect of B+ trees is the limit on the degree of each node. If the names we are searching for are of fixed length, then we can require that each B+ tree must be of some *order*, say p, such that each of its nodes must have between $p/2$ and p descendants, inclusive, except for the root node, which may have fewer. Thus, for all but the smallest trees, if there are n leaves, the number of tree levels is between $ceil(log_p(n))$, where $ceil(m)$ is the smallest integer greater than or equal to m, and $2+ceil(log_p(n))$.

However, in most file-system directory applications, names are of variable length. We thus modify this node-degree requirement to say that each node fits in a block and each such block must be at least half full. We assume that each block can hold at least three entries (in practice, an 8-kilobyte disk block can hold a few hundred typical entries).

One final requirement of B+ trees is that their leaves are linked together. Thus one can enumerate the values contained in the leaves in lexicographic order simply by starting at the leftmost leaf and following the links to the others.

When B+ trees are used in file-system directories, their leaves usually don't contain individual directory entries, but are disk blocks containing some number of directory entries, all in lexicographic order. Thus the parent B+-tree node of each leaf specifies the range of names found in that leaf. When doing a search, we descend the tree to a leaf, and then search the leaf sequentially.

As with hashing, the hard part is insertion and deletion. We sketch how do to them here, and then work out the details in the exercises. If we insert a new name and there's room in the appropriate directory block, then things are easy. *Igor*, for example, is easily added to the

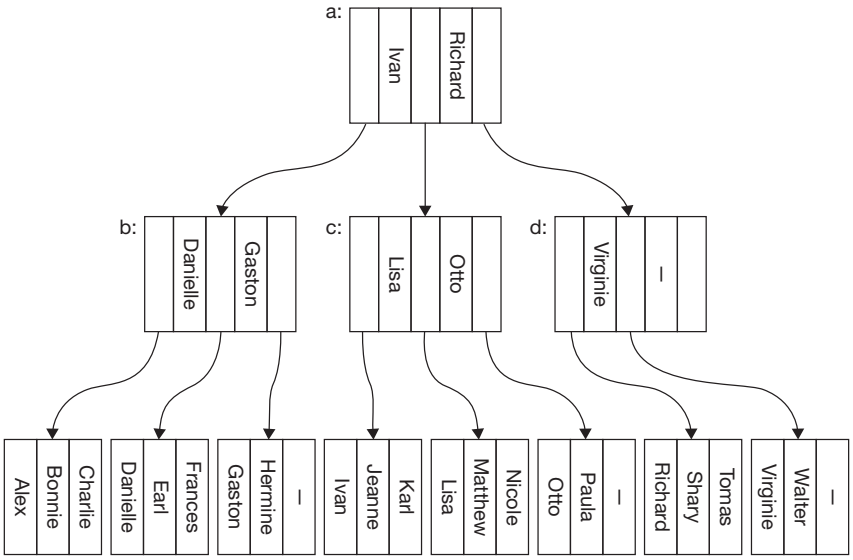


FIGURE 6.39 A B+ tree representing a directory. To simplify the figure, all entries occupy the same amount of space.

example of Figure 6.39. But if there's no room, we need to add a new directory block and fit this block into the search tree.

Adding a directory block requires a bit of work. Suppose we add the name *Lucy*. Since the directory block it should go into does not have enough free space to hold it, we create a new block and move everything greater than *Lucy*, namely *Matthew* and *Nicole*, from the old block to the new one, and put *Lucy* in the old one. We now have two blocks where we used to have one. If the parent node, *c*, of the just-copied block had room for it, we could simply modify *c*'s ranges to refer to the new block as well as the old one and we'd be done — we wouldn't have increased the length of any search path, so the tree would still be balanced.

However, in our case, this parent-node *c* has no room. So we split it into two nodes, *c* and *c′*, and add a range for the new node to the appropriate parent — *c′* in this case. We again have two nodes where we used to have one. So we must go to the parent of *c* and *c′* to accommodate *c′*. As before, if this parent has room, we're done, otherwise we have to split it and continue on.

If we reach the root and have to split it, we create a new root node with two ranges, each referring to one of the halves of the former root. We've thus increased the number of levels in the tree by one, yet maintained its balance. In our example, we have to split node *a* into nodes *a* and *a′*, and thus create a new root, *r* — see Figure 6.40.

When deleting an entry, we must make certain we maintain the invariant that all nodes are at least half full. Let's suppose we remove *Paula* and *Otto* in Figure 6.40. This causes the directory block containing them to become empty. We could simply leave the empty block in the directory — this is what S5FS and FFS do, after all. But it's certainly cleaner and more space efficient to remove it, so let's do so (though note that many systems don't bother rebalancing after a deletion, but simply delete the storage when a block is completely free).

We free the block and remove its range from the parent node that points to it, *c′*. If this node were more than half full, we'd be done. In our case it's not, so we examine the parent's adjacent siblings (there must be at least one — just node *d* in our case). Can we combine the contents of nodes *c′* and *d* to form a single node? If we can't, we would move just enough entries from *d* to *c′* so that it is at least half full (since each node must have room for at least three

entries, this is always possible), adjust the ranges in the grandparent (*a′*), and we'd be done. If we can combine the two nodes, as we can here, we do so, free one of them, and remove its entry in the grandparent (*a′*).

We now check this grandparent node to make sure it still is at least half full. If not, we repeat the above procedure on it and one of its adjacent siblings, working our way up towards the root as necessary. If we reduce the number of children of the root to one, we can eliminate the root and make its child the new root, thereby reducing the number of tree levels by one. The result for our example is shown in Figure 6.41.

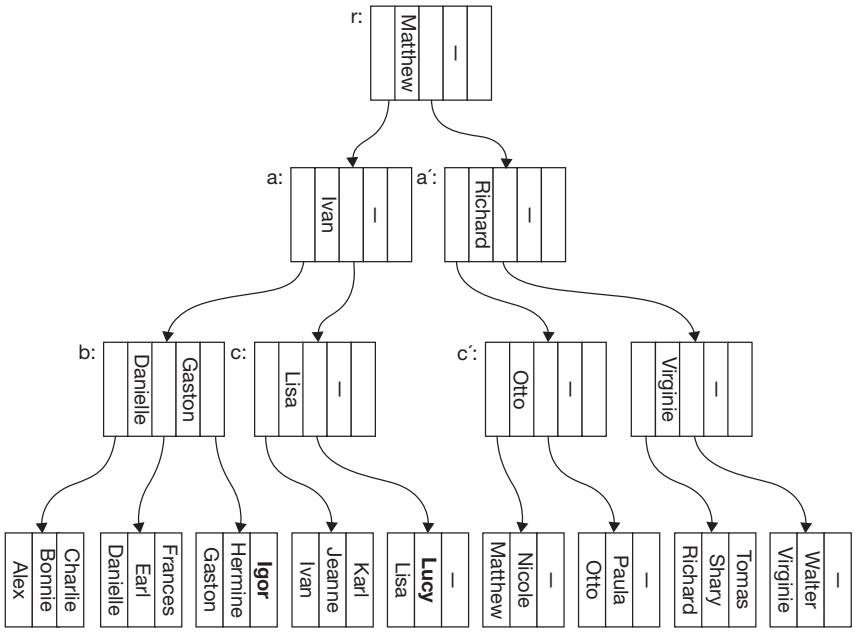


FIGURE 6.40 The result of adding entries for *Igor* and *Lucy* to the example of Figure 6.39.

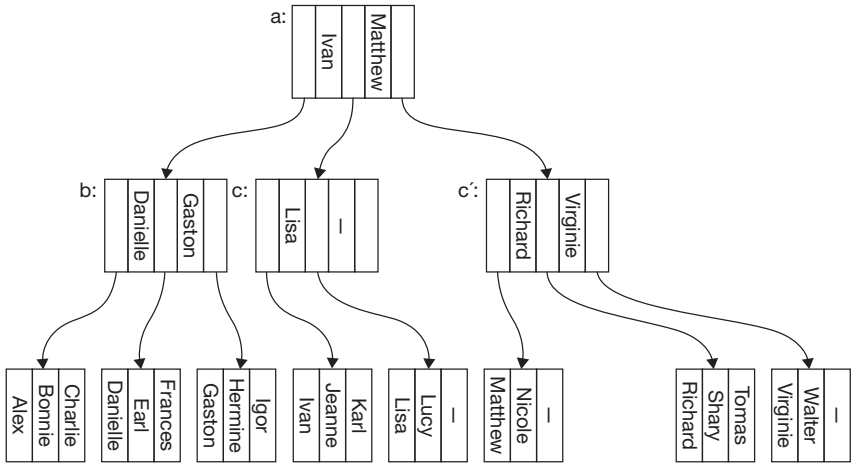


FIGURE 6.41 The directory after removing *Paula* and *Otto*.

6.3.2 NAME-SPACE MANAGEMENT

Let's now focus on the name spaces formed by directories. We have two concerns. One is that we have a number of such name spaces, one for each file-system instance. How do we distinguish them? How might we unify them? Another is whether we can take advantage of file-system name spaces to name things other than files.

6.3.2.1 Multiple Name Spaces

Microsoft systems use a simple approach to managing multiple name spaces: distinguish the name spaces by the "drive" their file system resides on. Thus *A:\x\y* is a file on drive A, while *B:\x\y* is a different file on drive B. A drive might be a region of a disk drive or perhaps all of a CD-RW or diskette, etc.

Alternatively, we might piece together the various directory hierarchies into a single unified hierarchy, as is done in Unix (and, recently, in Windows). We can superimpose the root directory of one file-system instance onto a directory of another, a procedure known as *mounting*. The original contents of the mounted-on directory become invisible; any attempt to access that directory accesses instead the directory mounted on it. Thus a path starting at the root of the mounted-on file-system instance can cross seamlessly into the file-system instance that was mounted upon it. For example, if, in Figure 6.42, the root directory of file system 2 is superimposed (mounted) on the directory */a/c* in file system 1, then we can refer to file system 2's */x/z* as */a/c/x/z* with the path starting in file-system 1.

This notion of mounting is used on Unix systems to tie all the file-system instances into a single unified name space. The file-system instance containing what is to be the root of this name space is called the *root file system*. If the file-system instance containing users' home directories is in file-system instance Z, it might be mounted on the root file system's */home* directory. So, for example, if my home directory is contained in Z at */twd* and I have a file in it called *memo.txt*, I could find it at */home/twd/memo.txt* in the unified hierarchy.

Both Unix (and hence Linux) and Windows support mounting, but they use different strategies for implementing it. In the Unix strategy one issues *mount* requests that tie the root of the file-system instance being mounted to the mounted-on directory. Any attempt to access the mounted-on directory gets the mounted root instead. This is accomplished within the operating system by linking together the cached copies of the file-system instance's inodes, without any changes to the on-disk copies of these inodes.

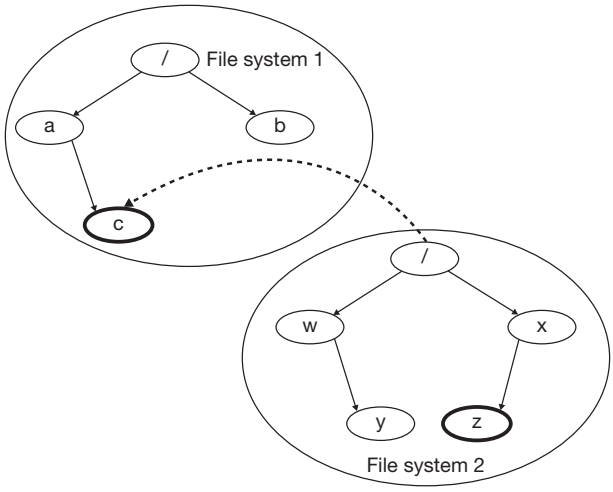


FIGURE 6.42 Mounting file system 2 on file system 1's directory */a/c*.

In a generalization of this approach, called *stackable file systems*, access to the mounted-on directory invokes other code layered on top of it. This has been used to implement "union mounts" in which the roots of multiple file-system instances are mounted on the same directory, so that one sees the union of their name spaces. Another application is the *portal* — access to the mounted-on directory results in, for example, the invocation of code to access some remote service. Portals were implemented in Free BSD Unix, but don't appear to be widely used.

Mounting is implemented in Windows via a mechanism called *reparse points* that, like stackable file systems, supports a number of other applications (unlike stackable file systems, these other applications are actually used). Marking a file or directory as a reparse point indicates that it requires special treatment. For example, if we want it to be a mount point, then its contents should be ignored and another directory, whose name is supplied as *reparse data*, should be used instead.

A file or directory marked as a reparse point contains, in addition to reparse data, a *reparse tag* indicating which subsystem is responsible for interpreting it. When such a file or directory is encountered when the system is following a path, an error is returned to the system code. The system code then examines the reparse tag and then directs further processing to the appropriate subsystem, which uses the reparse data. Applications include standard notions such as hard and symbolic links (known as *junctions* in Windows) as well as more novel ideas such as support for hierarchical storage management (HSM). This latter allows one to mark a file, via a reparse point, as having been moved to offline storage. An attempt to access the file is translated into a request to HSM to fetch the file from the offline store.

6.3.2.2 Naming Beyond the File System

Among the many important ideas popularized, if not introduced, by Unix is the notion of naming things other than files via the directory hierarchy. Earliest Unix used the file system to name devices. For example, */dev/disk1* might refer to a disk device, not the files within it. */dev/mt* might refer to a magnetic tape unit, and so forth. The */dev* directory is where, by convention, devices appeared in the name space. In principle they could appear in any directory. An entry in the name space referring to a device was called a *special file*.[9]

There were two important uses for this idea. The first was that it provided a simple means for referring to a device as a whole. For example, if you want to mount the file-system instance contained in */dev/disk1* on the directory */mnt*, you can do it simply with the following command:

```
mount /dev/disk1 /mnt
```

The other use was to simplify using devices. If, for example, you want to write to the tape drive, you could simply open */dev/mt* as if it were a file, then write to it via the file descriptor. Of course, within the kernel, the *write* system call code would have to call the tape driver rather than the file system. An inode for */dev/mt* would be stored on disk, but rather than representing a file, it would indicate which driver to use to access the tape drive and which tape drive to access.[10]

More recent versions of Unix have expanded the use of the file-system name space to refer to processes as well. A special file system, *procfs*, can be mounted whose files represent processes. This is particularly handy for debuggers, which can "open" a process, read and write its address space, and control its execution (Killian 1984). This idea of using the file system to name pretty much everything was further elaborated in *Plan 9*,[11] an operating system designed by the original developers of Unix after they grew tired of Unix (Pike, Presotto, et al. 1995).

---

[9] Unix distinguishes between block-special and character-special files. The former are accessed via the buffer cache, the latter are not. Disk and tape devices are typically named with two special files, one block-special, the other character-special.

[10] Similar functionality could certainly be obtained with reparse points, though this notion of special files predated reparse points by around 30 years.

[11] Plan 9 is actually short for "Plan 9 from Bell Labs," a name derived from that of the classic B movie "Plan 9 from Outer Space."

Another trend has been to extend naming outside of the file system. On the one hand, this makes names transient, since, if they aren't in the file system, they won't survive crashes. On the other hand, they can be recreated each time the system reboots. Such naming was used with great success in SpringOS, a research operating system developed at Sun Microsystems Laboratories (Radia et al. 1993).

This approach was also adopted in Windows NT, where it's used to refer to objects maintained by the operating system, including those representing I/O devices. For example, *\Device\CdRom0* refers to a CD-ROM device. However, unlike Unix's special files, this name is employed internally and cannot currently be used by applications. An application instead would refer to drive D, which is mapped by the kernel to *\??\CdRom0*. The *\??* directory contains names used in the older DOS operating system and provided backwards compatibility in Windows.

Names are also used outside of the file system by Windows to support its *registry*, which is a simple database providing names and values useful for configuring and maintaining state information for system and user applications. Though it's outside the file system, both the registry and its naming hierarchy are maintained on disk. For details, see (Russinovich and Solomon 2005).

6.4 MULTIPLE DISKS

What are the benefits of more than one disk drive?

1. You have more disk storage than you would with just one disk drive.
2. You can store data redundantly, so if one drive has a problem, you can find the data on another.
3. You can spread data across multiple disk drives; then, taking advantage of parallel transfers to and from the drives, you can access the data much faster than if it were on a single drive (this assumes there is sufficient I/O bandwidth).

The first point may seem so obvious as to be flippant, but putting it into practice is not straightforward. With two disks, we could have two separate file-system instances, one on each disk. However, this does us no good if we want to handle a really large file, one that is too big to fit on one disk. Or we might want to have within a single directory files whose total size is larger than can fit on one disk.

Thus it's often useful to have one file system spanning two or more disks. However, the disk addresses used within a file system are usually assumed to refer to one particular disk or subdisk; i.e., the disk or subdisk forms an address space. There is no means of directly referring to disk blocks on another disk or subdisk.

One way to get around this problem is to provide an interface that makes a number of disks appear as one. The software that does this is often called a *logical volume manager* (LVM). It provides this sort of functionality by presenting to the file system a large address space, but mapping the first half of the addresses to one disk and the second half to another.

A logical volume manager can also easily provide redundancy: with two disk drives, each holding a separate, identical copy of the file system, writes can go to the LVM and the LVM can direct them to the same block address on each disk. Reads can go to either disk; a reasonable LVM would direct them to the less busy one. This technique is known as *mirroring*.

Parallel access can be provided by putting our file system on multiple disk drives in yet another way, using a technique known as *striping*. Let's say we have four disk drives, each on a separate controller. When we write to disk, we arrange to write four sectors at a time: the first goes to the first disk, the second to the second disk, and so forth. Of course, we'll also read the file four sectors at a time. We get a clear fourfold speedup over a file system on just a single disk drive. The reason for the term "striping" becomes apparent if we depict our file as a matrix that's four sectors wide. Each column of sectors resides on a separate disk; each row represents logically contiguous sectors. If we think of each of the rows as being colored differently, then we have stripes (see Figure 6.43).

	Disk 1	Disk 2	Disk 3	Disk 4
Stripe 1	Unit 1	Unit 2	Unit 3	Unit 4
Stripe 2	Unit 5	Unit 6	Unit 7	Unit 8
Stripe 3	Unit 9	Unit 10	Unit 11	Unit 12
Stripe 4	Unit 13	Unit 14	Unit 15	Unit 16
Stripe 5	Unit 17	Unit 18	Unit 19	Unit 20

FIGURE 6.43 Disk striping: each stripe is written across all disks at once. The size of a "unit" may be anywhere from a bit to multiple tracks. If it's less than a sector in size, then multiple stripes are transferred at once, so that the amount of data per transfer per disk is an integer multiple of a sector.

Striping requires some careful design to be really effective. Let's first discuss the terminology. The two factors normally specified are the *stripe length*, which is the number of disks to which data is written, and the *striping unit*, which is the "maximum amount of logically contiguous data that is stored on a single disk" (Chen and Patterson 1990). This latter term requires some explanation. It's not necessarily the amount of data written to one disk at a time, but is rather an indication of how data is spread across the disks. It can be as small as one bit, in which case we have what is known as "bit interleaving": the first bit of a file goes to the first bit of the first block of the first disk, the second bit of the file goes to the first bit of the first block of the second disk, and so forth until we have one bit on each disk. Then the next bit of the file goes to the second bit of the first block of the first disk, the bit after that goes to the second bit of the first block of the second disk, and so forth — the bits are distributed in round-robin style across all the disks.

If the striping unit is a byte, we have byte interleaving: the bytes of a file are spread out in round-robin style across the disks. Similarly, we might have sector as well as block interleaving. We can think of the stripe unit as the width of the stripe (as opposed to the length, which is the number of disks), so that with bit interleaving the stripes are one bit wide and with block interleaving they are one block wide.

For a given request size and a fixed number of disks, low striping unit values allow us to spread each request across all the disks. This achieves high parallelism for the request, even with a relatively small amount of data to transfer to or from each disk. With large striping unit values, many requests might be for data that resides completely on one disk; thus the data is transferred without the benefits of parallelism.

It might seem that maximal parallelism is the way to go, but suppose we have a number of requests for data at uniformly distributed locations waiting to be served. With a small striping unit, the transfer time for each request is kept small due to the parallel transfer from the multiple disks. However, the positioning delays are not reduced:[12] for each request, all disks must first seek to the proper cylinder, then wait for the desired sector to come by. Thus for each request we

[12] In fact, rotational delays could be worse: another striping issue is whether or not the disks rotate synchronously, so that the first sector on all disks rotates under each disk's heads at the same time. This is important if we're concerned about the time to write to or read from all disks in parallel. If they are rotating synchronously, then the effective average rotational delay for all disks is what we expect for a single disk — half the maximum delay. But if they're not rotating synchronously, the effective average rotational delay — the time we must wait for the appropriate sector to rotate underneath the disk heads on all disks — is much closer to the maximum delay.

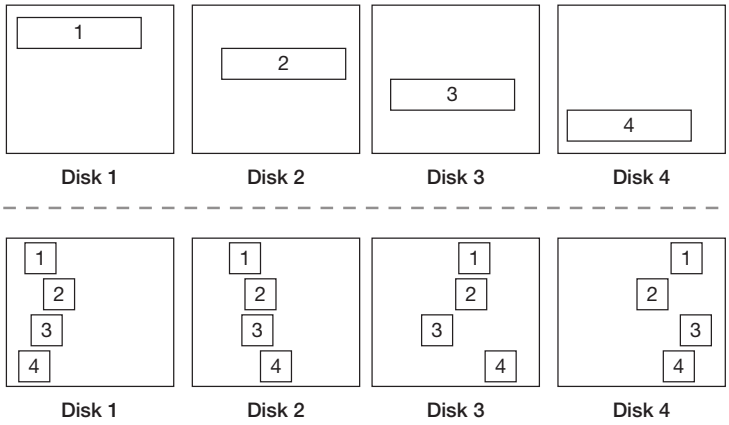


FIGURE 6.44 The top half of the figure shows four disks with a four-sector striping unit. Suppose we have requests for the four data areas shown, each four sectors in length. The four requests can be handled in roughly the time it takes to handle one, since the positioning for each of the requests can be done simultaneously, as can the data transfer. The bottom half of the figure shows four disks with a one-sector striping unit. We have the same four requests, but in this case each is spread across all four disks, one sector per disk. Handling the requests requires first positioning the heads on all four disks for the first, then positioning the heads on all four disks for the second, and so forth. Thus the total positioning delays are four times those in the top half of the figure, which has a larger striping unit.

have to wait a long time for positioning, then wait a brief time for the parallel transfer of the data. The total time to handle all the requests is the sum of all the positioning delays — a large number — plus the transfer delay — a small number.

We might be better off with a large striping unit that causes each request to be handled completely on a separate disk. Thus multiple requests could be handled at once, each on a separate disk, with the total positioning delays being reduced by a factor equal to the number of disks. Though individual transfers would not benefit from parallel access, the total amount of data being transferred would be much increased — the positioning for the multiple requests is done in parallel, as are the data transfers. Thus the total time spent transferring data using a large striping unit would be the same as with a small striping unit, but the positioning time would be dramatically reduced. See the example in Figure 6.44.

The number of requests waiting to be handled is known in the striping literature (Chen and Patterson 1990) as the *concurrency factor*. The larger the concurrency factor, the less important striping is: performance is better with a large striping unit than with a small one.[13]

Thus striping gives us what is in effect a disk with much faster data transfers than ordinary ones, but with no improvement in seek and rotational delays. This "effective disk" is also more prone to failure than the usual sort.[14]

[13] Note that this line of reasoning assumes that the requests are to uniformly distributed locations on the disks. If there is locality of reference, the argument is not as strong.

[14] Assuming disk failures are independent and identically distributed (not necessarily a valid assumption), if the probability of one disk's failing is f, then the probability of an n-disk system failing is $(1 - (1 - f)^n)$.

6.4.1 REDUNDANT ARRAYS OF INEXPENSIVE DISKS (RAID)

Let's look at the use of multiple disks from a different point of view — cost. Unless you get an incredibly good deal, *N* disk drives cost more than just one for all values of *N* greater than 1. But what if you want to purchase a "disk system" to handle a zillion bytes of data with an average access time of two milliseconds and a transfer rate of 1 billion bytes/second? You might be able to find one disk that satisfies your constraints, but it's going to be pretty expensive (it's called a SLED — single large expensive disk — in (Patterson, Gibson, et al. 1988)). But you could put together a large number (depending on how big a zillion actually is) of cheap, off-the-shelf disks that meet your constraints for far less money. You'd of course use striping to get the desired transfer rate. Since you have so many disks, the odds of at least one of them failing are not insignificant, so it makes sense to have some extra ones. So, what we end up with is RAID: a redundant array of inexpensive disks (Patterson, Gibson, et al. 1988).[15]

(Patterson, Gibson, et al. 1988) defined various RAID organizations or *levels* on the basis of the striping unit and how redundancy is handled. Though they didn't define RAID level 0, the term now commonly refers to pure striping as discussed above — no redundant disks. What were defined in the paper are levels 1 through 5.

RAID level 1 (Figure 6.45) is disk mirroring, as discussed above. Two identical copies are kept of the file system, each on a separate disk. Each write request is applied to both copies; reads go to either one.

RAID level 2 (Figure 6.46) is striping using bit interleaving and some redundancy. Rather than an extra copy of each disk, an error-correcting code (ECC) is used. Such codes let us provide a few extra "check bits" along with each set of data bits. In the case of a single-error-correcting code, if any one bit is modified (due to noise, etc.), whether a data bit or a check bit, we can compute which bit it was and hence correct it. We won't go into any of the details; (Patterson, Gibson, et al. 1988) refers its readers to (Hamming 1950), which describes a code that uses four check bits per ten data bits to correct any single-bit error.

Suppose we have ten disks holding data. We would need an additional four disks to hold check bits so that all one-bit errors can be corrected. Since the striping unit is one bit, as we distribute each set of ten data bits on the ten data disks, we compute the check bits and distribute them across the four check disks. Consider any sector on any of the disks. Each bit in that sector is part of a different 10-data-bit-plus-4-check-bit stripe. If any number of them is modified due to a problem, we can determine which they were and repair them. Thus we can recover even if we lose an entire disk.

In discussing the performance of RAID disks, we need to distinguish between large writes, which have enough data to fill a block in each data disk, and small writes, which have just a block's worth of data. For large writes, performance is good. Since RAID level 2 has fewer check disks per data disk than RAID level 1, it makes more efficient use of space. However, performance for small writes is miserable: to write one block's worth of data, one must first read a block from each of the data disks, then bit-interleave the new data on these blocks, and then write them all out to the data disks along with the new check blocks to the check disks.

The use of sophisticated ECC in RAID level 2 is overkill, since most disk controllers already employ it at the sector level on disk drives. Thus a problem with a particular sector can be resolved by the disk controller. However, if there's a major problem with that sector, it's likely that the controller won't be able to repair the error, but merely identify the sector as bad. If we lose an entire disk drive, it should be pretty clear which one it was.

RAID level 3 (Figure 6.47) is an improvement on level 2 for the (usual) case in which the disk controller does sector-level detection of errors. If there's an error, we don't need to identify

[15] Common usage today is that the "I" in RAID stands for *independent*, perhaps because all disks are now inexpensive.

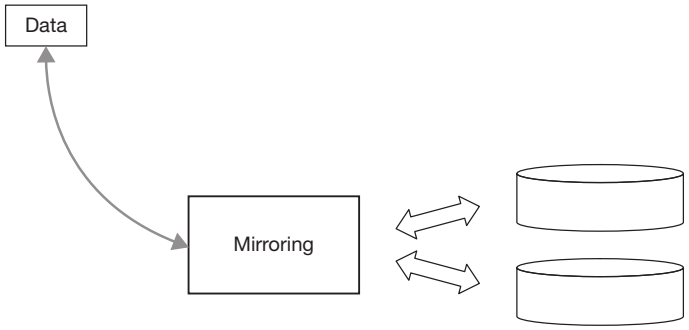


FIGURE 6.45 RAID level 1: disk mirroring.

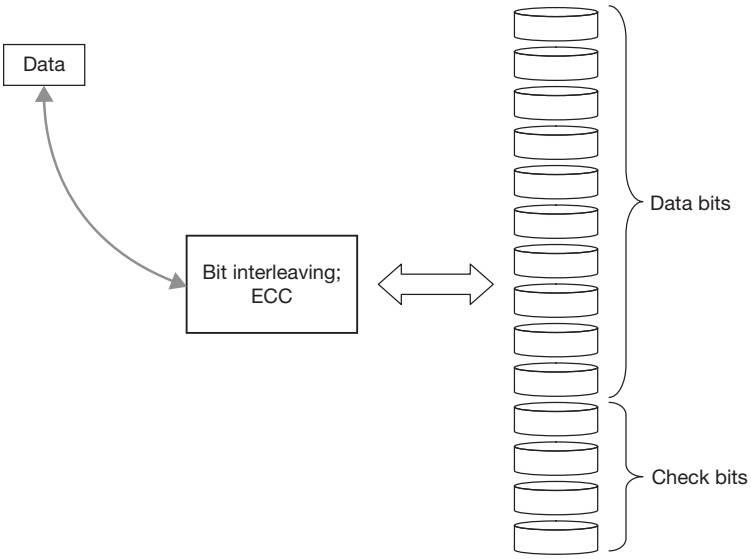


FIGURE 6.46 RAID level 2: bit interleaving with an error-correcting code.

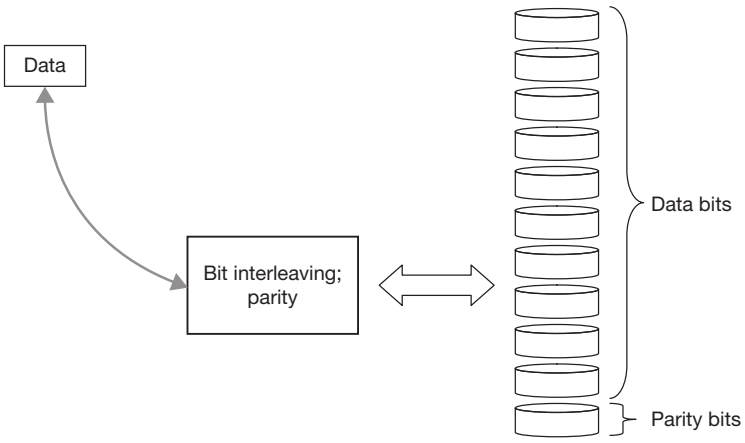


FIGURE 6.47 RAID level 3: bit interleaving with parity bits.

where it is, since that's done for us already, but we do need to repair it. Now, however, no matter how many data disks we have, we need only one check disk. For a system with 10 data disks and a striping unit of 1 bit, for each 10 bits distributed to the data disks, the check bit is the exclusive OR of the data bits. Thus if any of the bits are modified, recomputing the exclusive OR will show that there's a problem, though with only this information we won't know which bit was in error. But since we'll find out from the disk controller that a particular sector is suspect, we'll know which bits might be in error — those that reside in that sector. Since each of these bits resides in a separate 10 data-bit plus 1 check-bit stripe, we can restore it to its correct value as necessary.

Bit interleaving has some convenient properties if we need to compute check bits for an error-correcting code, but otherwise it's expensive, particularly if done in software. RAID level 4 (Figure 6.48) is just like level 3, except block interleaving rather than bit interleaving is used — the striping unit is increased to be equal to the block size.

RAID level 4 is not great at handling write requests of just one block. It's bad enough that modifying one block requires us to modify the corresponding block on the check disk as well,[16] but modifying any block on any of the data disks requires an update operation to modify a check block on the check disk. Thus the check disk becomes a bottleneck.

RAID level 5 (Figure 6.49) fixes this problem: rather than dedicating one disk to holding all the check blocks, all disks hold check blocks (and data) — the check blocks rotate through all the disks. As we go from one stripe to the next, the check block moves to a successive disk. Performance is thus good both for small writes and large writes, and there is no bottleneck disk for the small writes, as in RAID level 4.

It would seem that RAID level 5 is the ultimate, certainly more desirable than the other RAID levels. But there is at least one other concern: expansion. Suppose we decide to add another disk or two to a RAID level-5 system we've been using for a couple of years. Doing this would involve reorganizing the contents of all the disks so as to extend the check-block pattern to the new disk. Adding a disk to a RAID level-4 system involves simply recomputing the contents of the check disk to take into account the new data disk, a much easier job.

The one advantage of RAID level 5 over RAID level 4 is its performance for small writes. But, as we've seen, relatively recent file systems such as WAFL and ZFS really don't have small

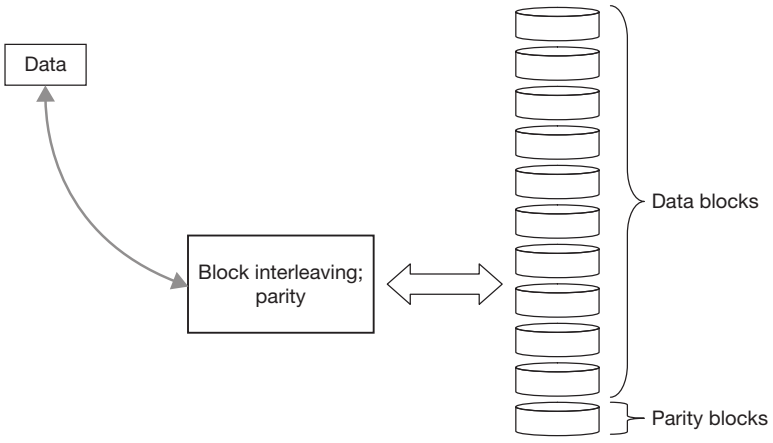


FIGURE 6.48 RAID level 4: block interleaving with parity blocks.

[16] You might think that we have to read all the other data blocks in the striping row in order to recompute the check block, but this is not necessary — see Exercise 17.

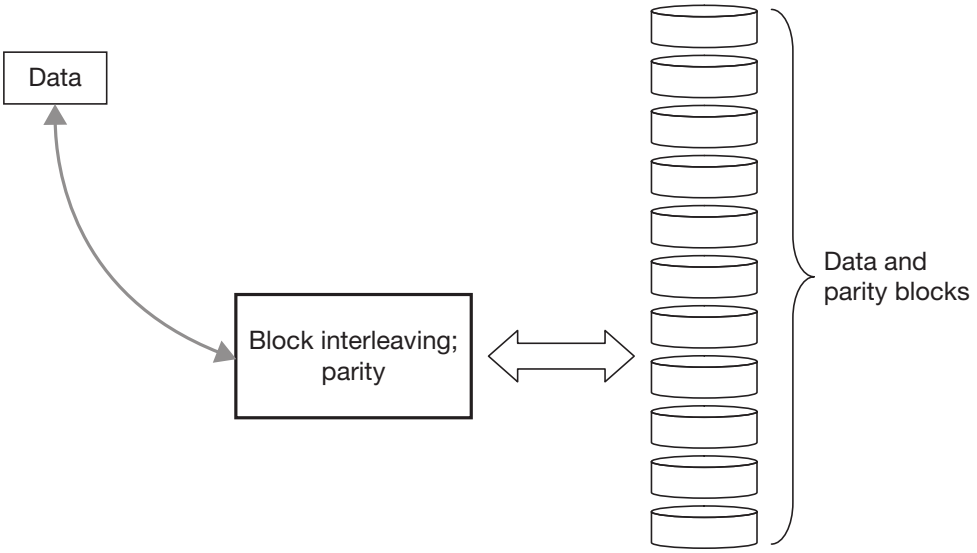


FIGURE 6.49 RAID level 5: block interleaving with parity blocks. Rather than dedicating one disk to hold all the parity blocks, the parity blocks are distributed among all the disks. For stripe 1, the parity block might be on disk 1; for stripe 2 it would be on disk 2, and so forth. If we have eleven disks, then for stripe 11 the parity block would be back on disk 1.

writes. Updates are collected until a large number can be written out, and thus most writes contain a number of stripes. However, other recent file systems, in particular journaled systems such as Ext3 and NTFS, must still make relatively small updates as part of checkpointing. For them, RAID level 5 makes the most sense.

Since the seminal paper on RAID (Patterson, Gibson, et al. 1988) the hierarchy has been extended. RAID level 6 is similar to RAID level 5, except that it provides additional parity blocks allowing it to handle two failures. *Cascaded RAID* combines two RAID architectures into one. RAID 1+0 (sometimes called RAID level 10) does striping across mirrored drives. RAID 0+1 has two striped sets, one mirroring the other.

6.5 FLASH MEMORY

Up to this point we've assumed that file systems reside on disks — an assumption that has been valid for around 50 years. Flash memory is a relatively new technology providing the functionality needed to hold file systems: it is both writable and readable, and it retains its contents even when power is shut off. Though more expensive than disk technology, it is cheaper than DRAM technology (on which primary storage is built). It provides generally faster access to data than disks do, although it is slower than DRAM, and unlike disks, it provides uniform random access to data. However, writing takes significantly more time than reading, and there is a limit to how often each location can be written to. In this section we explore how flash memory is used both to supplant and augment disk devices for support of file systems.

6.5.1 FLASH TECHNOLOGY

There are two sorts of flash devices, *nor* and *nand* — the names referring to how the individual cells are organized. The earlier was *nor*, which is byte-addressable and thus can be used to hold code such as BIOS for direct execution. The more recent is *nand*, which is page-addressable and less expensive. Thus *nand* flash is used in conjunction with file systems; we focus our attention on it. (Wikipedia 2009) provides a general and concise discussion of flash technology.

Nand flash devices are organized into blocks that are subdivided into pages. Block sizes range from 16k bytes to 512k bytes, with page sizes ranging from 512 bytes to 4k bytes. Writing to a block is complicated, involving *erasing* and *programming*. *Erasing* a block sets it to all ones. One can then *program* an individual page by changing some of its ones to zeros. One can perform multiple programming operations on a page, but the only way to change zeros to ones is to erase the entire page. Any particular block can be erased no more than around a hundred thousand times.

These characteristics provide challenges to implementing a file system on flash. The propensity of modern file systems to reuse blocks exacerbates the problem of blocks "wearing out" because of too many erasures. To minimize the effects of wear, a flash file system must avoid reusing blocks and distribute its use of blocks across the entire device, a notion known as *wear-leveling*.

A relatively easy approach to an effective flash file system is to create a layer on top of the flash device that takes care of wear leveling and provides a disk-like (or block) interface. Then one simply places a standard disk file system on top of it. Such layers were formalized in 1994 and are called the *flash translation layer* (FTL) (Intel 1998). It is implemented in firmware within a device controller and maps virtual disk blocks to real device blocks. In broad outline, when the file system modifies what it thinks is a disk block, FTL allocates a new device block, copies the modified block into it, then maps the disk block into the new device block. The old device block is eventually erased and added to a list of available blocks.

The reality is a bit more complicated, primarily because the file systems FTL was designed to support were Microsoft's FAT-16 and FAT-32, which are similar to S5FS, particularly in supporting a single and rather small block size. Thus FTL maps multiple virtual disk blocks to a single device block.

6.5.2 FLASH-AWARE FILE SYSTEMS

Using an existing file system on a flash device is convenient, but the file system was presumably designed to overcome problems with accessing and modifying data on disk devices, which are of no relevance to flash devices. We can make better use of the technology by designing a file system specifically for flash memory. Flash devices that do not have built-in controllers providing FTL or similar functionality are known as *memory technology devices* (MTD).

File systems, regardless of the underlying hardware, must manage space. This certainly entails differentiating free and allocated space, but also entails facilitating some sort of organization. For disks, this organization is for supporting locality — blocks of files must be close to one another to allow quick access. But file systems for flash devices must organize space to support wear-leveling; locality is not relevant.

File systems also must provide some sort of support for atomic transactions so as to prevent damage from crashes. What must be done in this regard for file systems on flash memory is no different than for file systems on disk.

Consider a log-structured file system. Files are updated not by rewriting a block already in the file, but by appending the update to the end of the log. With such an approach, wear-leveling comes almost free. All that we have to worry about is maintaining an index and dealing with blocks that have been made partially obsolete by subsequent updates. Transaction support also comes almost free — the log acts as a journal.

The JFFS and JFFS2 file systems (Woodhouse 2001) use exactly this approach. They keep an index in primary storage that must be reconstructed from the log each time the system is mounted. Then a garbage collector copies data out of blocks made partially obsolete and into new blocks that go to the end of the log, thereby freeing the old blocks.

UBI/UBIFS started out as JFFS3, but diverged so greatly that it was given a new name. (The design documentation of UBIFS is a bit difficult to follow; (Bityuckiy 2005) describes an

early version when it was still known as JFFS3 and helps explain what finally went into UBIFS.) UBI (for *unsorted block images*) is similar to FTL in that it transparently handles wear-leveling (Linux–MTD 2009). However, UBI makes no attempt to provide a block interface suitable for FAT-16 and FAT-32; its virtual blocks are the same size as physical blocks. Layered on top of it is UBIFS (the *UBI file system*), which is aware that it is layered on UBI and not on a disk. Any number of UBIFS instances can be layered on one instance of UBI (on a single flash device); all share the pool of blocks provided by UBI.

UBIFS (Hunter 2008) is log-structured, like JFFS, but its index is kept in flash, thereby allowing quicker mounts. Its index is maintained as a B+-tree and is similar to those used in shadow-paged file systems (Section 6.2.2.3), such as WAFL's. So that the index does not have to be updated after every write, a journal is kept of all updates since the index was last modified. Commits are done when the journal gets too large and at other well defined points (such as in response to explicit synch requests) — they cause the index tree to be updated, so that each modified interior node is moved to a new block.

6.5.3 AUGMENTING DISK STORAGE

Flash fits nicely in the storage hierarchy, being cheaper and slower than primary storage (DRAM) and faster and more expensive than disk storage. (Leventhal 2008) describes how it is used to augment disk-based file systems, in particular ZFS. One attractive use is as a large cache. Though the write latency to flash is rather high, the file system can attempt to predict which of its data is most likely to be requested and write it to flash, from where it can be quickly retrieved. (Leventhal 2008) reports that in an enterprise storage system, a single server can easily accommodate almost a terabyte of flash-based cache. This approach works so well that the disk part of the storage hierarchy can be implemented with less costly, lower-speed disks that consume far less power than high-performance disks.

Another use of flash is to hold the journal for ZFS, thus keeping uncommitted updates in nonvolatile storage. In the past this has been done with battery-backed-up DRAM, but flash provides a cheaper alternative. To get around its poor write latency, it can be combined with a small amount of DRAM to hold updates just long enough for them to be transferred to flash. This DRAM requires some protection from loss of power, but only for the time it takes to update the flash. (Leventhal 2008) suggests that a "supercapacitor" is all that is required.

6.6 CASE STUDIES

Here we summarize and fill in some of the details about the file systems mentioned earlier in this chapter. Our intent is to give a balanced presentation of all of them and to point out important features that aren't covered above. This section may be skipped by those interested only in file-system concepts and not in further details of existing systems.

6.6.1 FFS

FFS, the Fast File System originally developed at the University of California, Berkeley (McKusick, Joy, et al. 1984), has been the primary file system of BSD Unix starting with 4.2 and continuing with FreeBSD. Its design was the basis for Linux's Ext2 and Ext3. It's been supported in a number of commercial versions of Unix, including Sun's Solaris, and has come to be known as UFS, the Unix File System. A number of improvements have been made to it over the years and it's probably the best studied and best published file system in existence (for example, (Ousterhout, Da Costa, et al. 1985), (McVoy and Kleiman 1991), and (Smith and Seltzer 1996)).

As we discussed in Section 6.1.4, the design of FFS was a response to the problems of S5FS and features better organization of file data and metadata to minimize positioning delays,

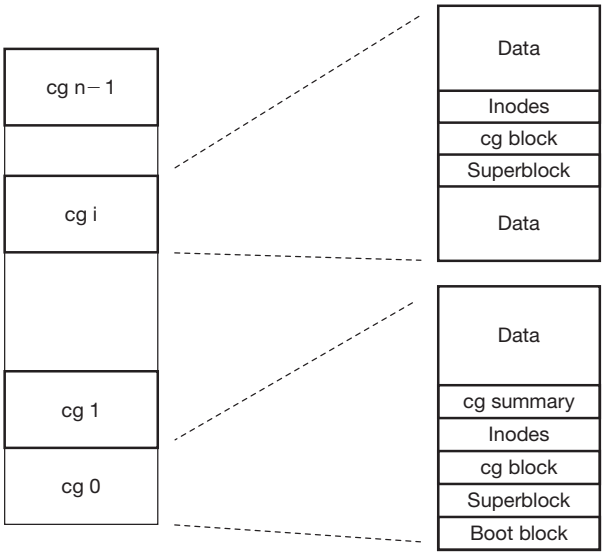


FIGURE 6.50 FFS layout.

as well as a larger block size to improve transfer speed. The key features discussed above include the use of cylinder groups to help localize file allocation, what was at the time a large block size made palatable through the use of fragments, and a file representation based on inodes allocated from a static array. Among the later improvements is the use of block clustering to improve data-transfer bandwidth.

Figure 6.50 shows the on-disk organization of FFS. Allocation information for each cylinder group, including free-fragment bitmaps, is maintained in a *cylinder-group block* (*cg block*) data structure. Though residing on disk, it's cached in memory when used. To facilitate finding space in cylinder groups, a *cylinder-group summary* (*cg summary*) structure is stored in cylinder group 0 and brought into kernel memory when the file system is in use. The cylinder-group summary contains the amount of space available in each cylinder group and thus can be scanned before going to find the space in the cylinder group itself.

Lastly, the *superblock* resides in cylinder 0 and contains all the file-system parameters such as size, number of cylinder groups, and so forth. Just in case a disk problem destroys the superblock, copies of it are maintained in each of the other cylinder groups, in a rotating track position (track 0 in cylinder group 0, track 1 in cylinder group 1, etc., wrapping around at the maximum track number of a cylinder).

FFS's approach to crash tolerance is careful ordering of updates to disk. Initially this was achieved with the aid of synchronous writes of key metadata. As explained in Section 6.2.2.1, more recent versions of FFS employ soft updates to accomplish the same thing without the need for synchronous writes.

6.6.2 EXT3

Ext3 is the third in a sequence of extended file systems for Linux, each moving farther away from the original Linux file system, Andrew Tanenbaum's Minix file system (Tanenbaum 1987). It's become the mainstream file system of Linux, though certainly not its only file system — others with either journaling or other support for crash tolerance include ReiserFS (see Section 6.6.3

below), XFS (Sweeney et al. 1996), originally developed at SGI, and Episode (Chutani, Anderson, et al. 1992), developed at Transarc (a company later acquired by IBM).

Ext2 is based on the original design of FFS. It was later updated to include clustering, as explained in Section 6.1.4.1 above. Other improvements, such as extent-based allocation, either have been made or are being discussed for it (see (Ts'o and Tweedie 2002)).

Ext3 is Ext2 with the addition of journaling (see the discussion of Ext3's journaling in Section 6.2.2.2). Since its on-disk format is essentially the same as Ext2's, one can switch from Ext2 to Ext3 (and back again) without having to convert what's on disk. This adds greatly to its appeal.

A recent addition to Ext3 is a new directory format that, imitating how Ext3 added journaling, is fairly compatible with the old format. This format replaces the sequentially searched directories of previous versions of Ext3 with one based on a combination of hashing and balanced search trees called *H Trees* (Phillips 2001). It's a search tree, though searching is based on the hashed values of component names rather than the actual names. Balancing is not done explicitly; instead, the hash function is relied on to keep things distributed. The leaves of the search tree are the original sequentially structured directory of Ext2. Since each tree node has a fanout in the hundreds, three levels of search tree (in addition to the leaf nodes) are far more than sufficient to handle any reasonably sized directory.

The search-tree nodes are encoded on disk as free space within Ext2 directory blocks. Thus an Ext2 or a non-upgraded Ext3 system will ignore the search tree during directory searches, though the tree might be badly damaged if new entries are added!

6.6.3 REISER FS

The Reiser file system, so called because it's the work of Hans Reiser and a few associates, was one of the more popular Linux file systems, probably second to the more conservative Ext3. It's radically different from Ext3, featuring a fair degree of extensibility as well as a unique structure. We discuss its version 4, which was to have replaced version 3 until development was halted after Reiser's 2008 murder conviction.

The entire file system is organized as a single balanced search tree, essentially a B+ tree, with exactly four levels (see Figure 6.51). It may seem like a tall order to make everything — directory entries as well as all file blocks —accessible via the same search tree, but this is accomplished using some magic in the search key. This key is 192 bits long and is structured depending on what is being searched for. When looking up a directory entry, the key consists of the object ID of the directory and the hashed name of the directory entry. When looking up a location within a file, the key consists of the file's object ID and the starting offset within that file.

The tree nodes are of fixed size, 4 kilobytes, and are made up of smaller units called *items*. For the top three tree levels — the interior nodes — items point to nodes at the next lower level and contain a *left delimiting key*, which is the smallest key of any item in the subtree rooted by the item.

Since Reiser FS uses a B+ tree, the leaves contain the data, both that of ordinary files and that of directories, as well as what in other file systems would be stored in inodes: file owner, permissions, link count, etc. The data of small files is divided into pieces called, strangely, *tails*, each of which is contained in an item; multiple such items are packed into a single 4-kilobyte leaf node. Larger files are divided into extents; the third tree level, known as twigs, may contain extent pointers, which contain both a pointer to a node and a length — the number of 4-kilobyte (leaf) nodes in the extent.

With fixed-height trees, balancing is not an issue. Instead, the concern is wasted space within the tree nodes. For example, three adjacent nodes, each one-third full, can be combined into a single node, freeing up the other two. Checking on every operation to see if such combining is possible is expensive. Reiser FS does such combining only when nodes are actually written to disk or when there is a severe shortage of space — an approach called *dancing trees*.

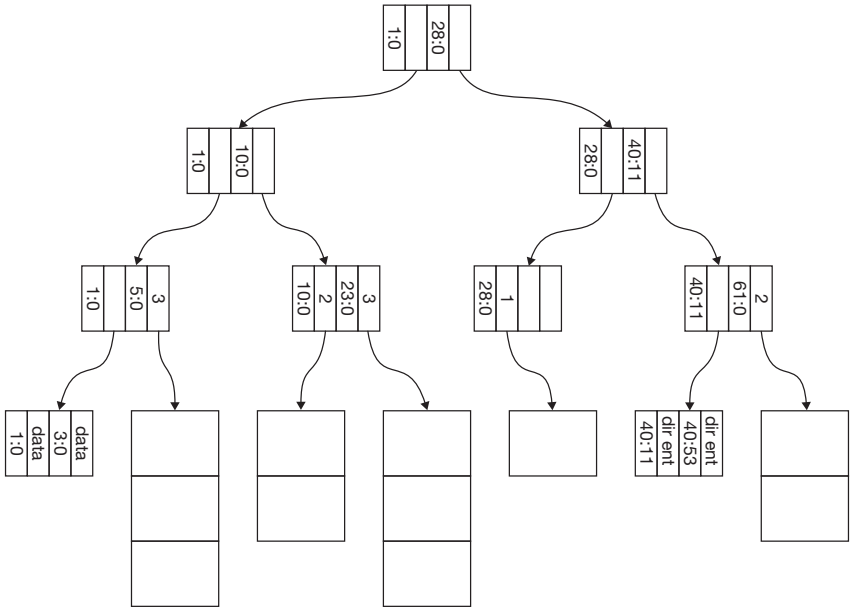


FIGURE 6.51 A Reiser4 tree. To keep things simple, each of the nodes contains at most two items. The leftmost leaf contains data starting at location 0 for each of two small files: one with object ID 1 and the other with object ID 3. Its sibling leaf is referred to by the parent via an *extent pointer*, and is three blocks long. The next three leaves to the right are also extents. The next leaf contains two directory entries, both of the directory whose object ID is 40. One entry's hashed component name is 11, the other's is 53. The rightmost leaf is a two-block extent.

Crash tolerance is achieved using a combination of journaling and shadow paging called *wandering logs*. The concern is that, when updating a file-system block, shadow paging avoids the cost of writing both to the log and to the original location at the expense of moving the block to a new location whose position might harm read performance. Reiser FS uses heuristics to determine which policy makes sense for each update. As part of committing a transaction, both shadow-paged and journaled blocks are written to contiguous locations, as in log-structured file systems. The journaled blocks are linked together, along with the address of their original positions, to form the journal (hence the term "wandering log"). The journal is in turn linked to the file system's superblock. In a crash, each member of the list of journaled blocks is written to its original location upon recovery.

Reiser FS uses *plugins* to achieve extensibility. Every item in the tree has a plugin ID identifying the module that provides its semantics. Plugin modules are provided to define the semantics for standard files, directories, and hash functions, but alternative versions are easily added.

One important aspect of extensibility is Reiser FS's support for *attributes*, a form of metadata. Unlike much metadata we've discussed before, however, attributes are used not for the system's layout of data, but to provide information about data. Attributes are used for the sort of information about a file, such as owner, protection information, and so forth, that often resides in inodes. With the use of plugins, arbitrary additional attributes can be added, along with restrictions implied by the attributes. For example, additional security attributes may be defined; who is allowed to use a file, and under what circumstances, can be governed by these attributes.

6.6.4 NTFS

NTFS is the standard file system on Microsoft's larger operating systems. WinFS is a database system that's layered on top of NTFS. Though in many cases users are dealing directly with WinFS, it is still NTFS that manages everything on disk. We discuss NTFS and leave WinFS for database textbooks.

We've already discussed much of the technology used in NTFS. Files are represented using extents. Directories are built on B+ trees. NTFS uses the term *file record* rather than inode and, as discussed in Section 6.1.5, maintains its array of one-kilobyte file records — what it calls the *master file table* (MFT) — in what is essentially a file. It provides four options for using multiple disks (see Section 6.4):[17]

- spanned volumes, in which regions from two different disks are combined to form what appears to be one single region
- RAID level 0 (striping)
- RAID level 1 (mirroring)
- RAID level 5

It provides crash tolerance using journaling, though unlike Ext3's redo journaling (Section 6.2.2.2), it employs a combination of both redo and undo journaling. Thus what are written to the journal are both the old contents and new contents of modified disk blocks.

Why do things this way? The idea behind redo journaling is that all steps of a transaction are written to the journal before the file system is itself modified. Though straightforward, this means that all file-system changes must remain in the cache and not go to disk until the transaction commits. This could be a problem if the operating system needs some of the cache's memory for other purposes. With undo journaling, a cached change may be written to disk and removed from the cache after it has been written to the journal but before all the transaction commits, thus allowing some cache memory to be freed, if necessary. (Why bother with redo journaling at all? Why not employ strictly undo journaling? See Exercise 11.)

Let's look again at the master file table (Figure 6.20). Each non-free entry describes a file in terms of a set of attributes. Some of these attributes are important special cases: a name attribute and a data attribute contain a file's name relative to its parent directory and its contents, respectively. There's also an attribute containing some of the usual attributes of a file, such as access and creation times and link count, and called, fittingly, *standard information*. Directories contain attributes comprising their B+ trees. The reparse information described in Section 6.3.2.1 is contained in another attribute. Files may be given a unique object ID, contained in yet another attribute.

If an attribute's value takes up little space, it's stored directly in the file-table entry. Otherwise the entry contains an extent table, referring to extents containing the attribute's value. This provides an efficient representation for small attributes, particularly for small data attributes. Thus a small file fits entirely within its file record (see Figure 6.52).

Indexing is a notion used in databases in which one finds data items by their attributes, much as one finds items in a book by looking them up in the index. Directories provide an index of files based on the last components of their name attributes. NTFS also maintains an index based on object ID in a special hidden file, implemented using a B+ tree, whose components are, of course, attributes. This is used by system components and applications that need a way to refer to files other than by names.

Multiple data streams allow one to associate data with a file in addition to the normal data. These are essentially a means for adding new attributes of arbitrary length to files. For example,

[17] This "volume aggregation" is actually done not by NTFS proper, but by a lower-level module on which NTFS is layered.

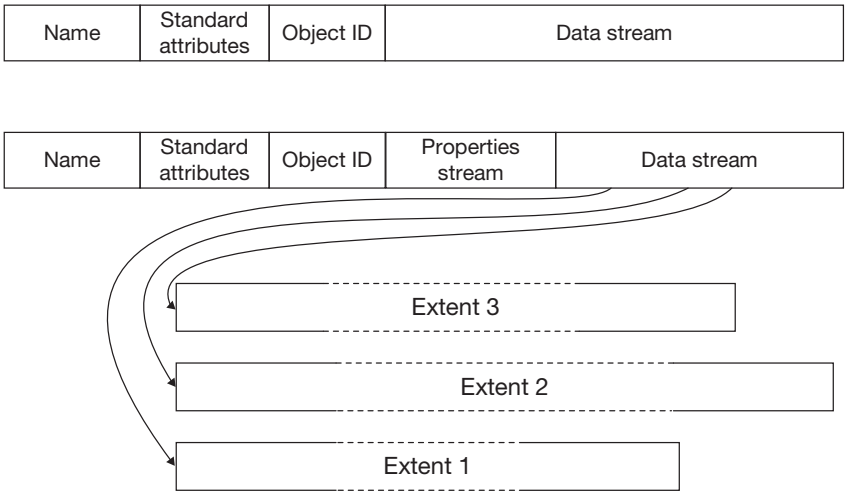


FIGURE 6.52 Two NTFS file records. The top one is for a small file fitting completely within the record. The bottom one is for a file containing two streams, one for some application-specific properties and one for normal data. The latter is too big to fit in the file record and is held in three extents.

many Windows applications support a notion of *properties* with which one can tag a file with a description of the file's contents. In NTFS, such properties are implemented as separate data streams (see Figure 6.52). The standard data stream containing the "normal" file data is unnamed and referred to implicitly simply by using the file name. The other streams are referred to by appending a colon and the stream's name to the file name. For example, the information associated with the file *\MyStuff\book.doc* is referred to as *\MyStuff\book.doc:information*.

One option for an NTFS file is for it to be compressed. Two techniques are employed for doing this. One is simply not to allocate space for long runs of zeros within files. Such runs are left out of the runlist of extents. Thus when the file system encounters a gap in the runlist, it assumes this missing extent contains all zeros.

The second technique is to use a compression algorithm[18] to compress the data in individual extents. Thus an extent that logically covers, for example, 16 kilobytes of data might be contained in 4 kilobytes of actual disk space. The compression and decompression are performed automatically by the file system.

6.6.5 WAFL

WAFL (for "Write-Anywhere File Layout"), from Network Appliance, runs as part of a special-purpose operating system on a special-purpose file-server computer, or *filer*, whose sole purpose is to handle file requests via such protocols as CIFS and NFS (see Chapter 9). For some details, see (Hitz, Lau, et al. 1994).[19]

WAFL is one of the first commercially available file systems to employ shadow paging and is perhaps the first file system to be structured as a tree with its data blocks as the leaves

[18] Microsoft's documentation does not reveal which compression algorithm is used.

[19] This article is somewhat out of date — our discussion is based partly on recent personal communication with George Cabrera of Network Appliance.

and most of its metadata as interior nodes, thus allowing periodic snapshots of the file system (see Section 6.2.2.3). It takes advantage of techniques pioneered by log-structured file systems (see Section 6.1.4.2): file updates are written to consecutive disk locations in batches, minimizing seek and rotational delays. This batching is further capitalized on to work well with RAID level 4, so as to provide the advantages of parallelism and redundancy shared with RAID level 5 while avoiding the latter's problems with adding new disks, and also avoiding the bottlenecks of typical RAID level-4 implementations (see Section 6.4.1).

How does all this work? Let's look at how the shadow-paging tree of Section 6.2.2.3 is implemented in WAFL. The root, rather than being called the überblock as in ZFS (Section 6.6.6), is given the rather unimaginative name *fsinfo*. Nevertheless, it contains the inode for what's called the inode file, containing inodes for all other files (see Section 6.1.5), including certain files that are actually metadata. Among the latter are files containing the free-space map.

As discussed in Section 6.2.2.3, all tree nodes (which are, of course, disk blocks) are treated in a copy-on-write fashion: when one is to be changed, a new copy is made and the copy is modified. Thus when a leaf node is modified, creating a copy entails modifying its parent's pointer to it, and hence the parent too is copied. This propagates all the way up to the root. Rather than actually writing all these changes to disk with each write to a file, the changes are collected into batches that are written out as *consistency points* at least every ten seconds. These changes are collected between disk writes in non-volatile RAM (NVRAM), which is battery-backed-up storage that retains its contents across power failures and other system crashes. Thus no file updates are lost, even if the system crashes. (This property is essential for WAFL to be a server for early versions of NFS, as discussed in Chapter 10.)

The disk space into which these batched changes are written is allocated so as to maximize speed. On a single-disk system, this would mean that contiguous blocks are allocated. WAFL, however, uses RAID 4 and thus the blocks are allocated in stripes across the disks; if multiple stripes are required, they occupy contiguous disk blocks on each of the disks. Since data is written not in single blocks at a time, but in multiples of stripes, the single parity disk of RAID 4 is not a bottleneck: it is written to roughly as often as the other disks (see Figure 6.53).

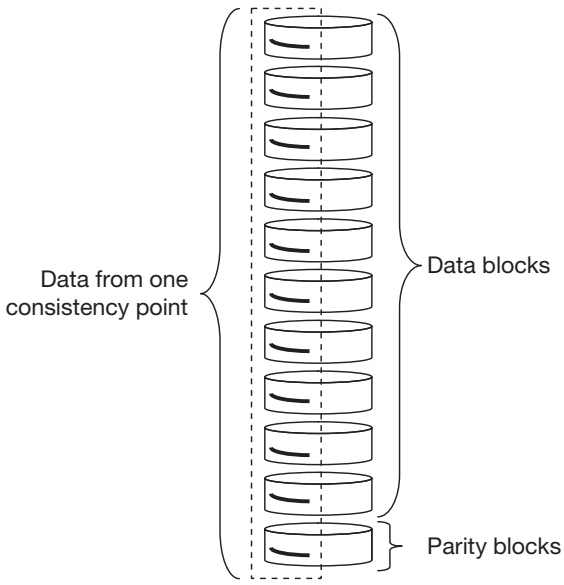


FIGURE 6.53 WAFL writes out changes in batches called consistency points. Each batch occupies some integral number of stripes — thus the parity disk is written to no more often than the data disks.

Normally the original copies of blocks modified at consistency points are freed, but the system administrator can arrange to retain a "snapshot" of the file system: the blocks being modified are not freed but kept, including the prior version of the modified fsinfo node — the root (see Figure 6.35). The administrator establishes a "snap schedule" indicating when such snapshots are made, perhaps hourly or nightly. Up to 255 may exist at any one time.

How is free space managed? Recall that with log-structured file systems (Section 6.1.4.2), free space was reclaimed as a background activity, as part of "log cleanup." Though WAFL is similar to log-structured file systems in the way it writes updates to contiguous locations, it doesn't require such a log-cleanup activity. Instead, since the free-space map is maintained in a number of files, these files are updated just like any other file: modifications are done in a copy-on-write fashion and written to disk as part of consistency points. Thus at each consistency point, the on-disk free-space map is updated.

With all the snapshots, how does WAFL determine what's free and what isn't? Each snapshot, including the most recent one reflecting the current status, has a bitmap indicating which blocks are in use. A summary bitmap, which is the logical OR of all the snapshot bitmaps, then indicates what's really free.

6.6.6 ZFS

ZFS (for "Zettabyte File System"), an open-source file system, was developed at Sun Microsystems for their Solaris operating system (Bonwick, Ahrens, et al. undated technical report: probably 2003) and has since been adopted by Apple for Mac OS X. It provides a robust approach to handling failures and providing good performance, along with much flexibility. Its name refers, indirectly, to the size of the addresses it uses, 128 bits.[20]

Its flexibility is made possible by its layered architecture, shown in Figure 6.54. At the top is the *ZFS POSIX Layer* (ZPL), which converts standard POSIX-style (Unix-style) operations into operations on *objects* within *datasets*, abstracting files and disks. The *Data Management Unit* (DMU) batches these operations into transactions and manages the metadata. Both ZPL and DMU operate in a large virtual address space and are oblivious to physical addresses. The *Storage Pool Allocator* (SPA) provides a malloc-like interface for allocating space (using slab allocation) and is responsible for mapping the virtual address space into actual disk addresses from any number of disks. In doing so, it may employ spanning, mirroring, or various forms of RAID, depending on the configuration.

ZFS is distinguished from the other file systems we have looked at by going well beyond them in the sorts of problems it can handle. In particular,

- Suppose there is a power failure while writing to a set of RAID disks. Perhaps a stripe consists of fifteen disks and writes to seven complete successfully, but writes to eight do not. When the system comes back up, there is not enough information available to recover either the new data that was not written or the old data that was overwritten.
- Suppose an obscure bug in the RAID firmware causes garbage to be written to disk (with correct parity!) on rare occasions.
- Perhaps a system administrator types the wrong command and accidentally scribbles on one disk, replacing data with garbage (again with correct parity).
- Suppose one's 1.6-TB file system is no longer large enough and must be made one terabyte larger.

[20] A zettabyte (10^{21} bytes) is actually far less than 2^{128} bytes (which is approximately 10^{36} bytes). It is, however, the smallest numerical prefix such that 64 bits aren't enough to address files of that size.

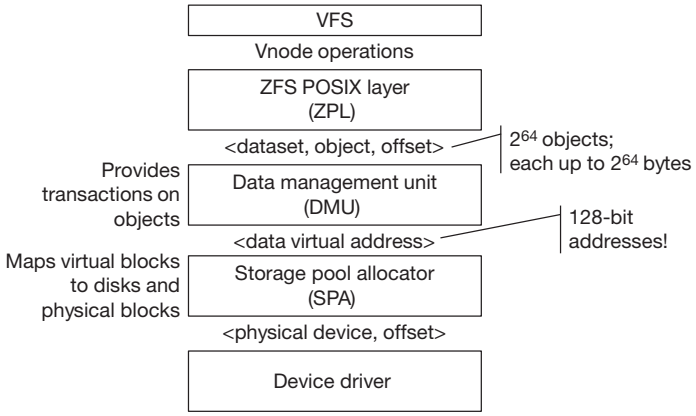


FIGURE 6.54 ZFS's layered architecture.

The DMU layer handles the first three of these problems by adding redundancy to its meta-data. As described in Section 6.2.2.3, the data management unit organizes all of its storage into a tree, headed by the überblock. This tree consists of a number of subtrees, each representing a separate logical file system. Creating a new file system is thus trivial — a new subtree is created and the file system uses the storage provided by the storage pool allocator, in common with all the other logical file systems of the pool. Snapshots may be taken of individual file systems, in a similar fashion as in WAFL.

Suppose, for example, a firmware problem causes the disk controller to write garbage data to a disk. Such errors are not covered by the parity provided with each sector by hardware, since the bad data we're concerned about here is, from the disk drive's point of view, good data that was written purposely. To cope with such errors, DMU maintains a checksum on the data in each block of its storage tree. For all but the überblock, the checksum is stored in the block's parent. Thus, if for some reason a block is inadvertently scribbled on, the checksum will detect the problem. If there is sufficient redundancy in the storage pool, the block can be automatically repaired.

If the only copy of a block is lost (perhaps because it was scribbled on), there is, of course, no way to recover it. This is bad, but it's even worse if the block contains metadata, particularly at the higher levels of the tree. ZFS protects against loss of metadata by allowing for three copies, known as *ditto blocks*, of each metadata block — each pointer to metadata is actually three addresses, one for each copy (see Figure 6.55).

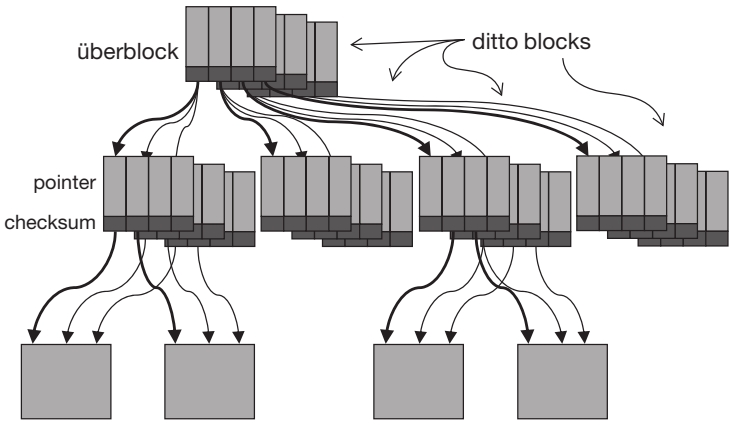


FIGURE 6.55 ZFS's shadow-page tree.

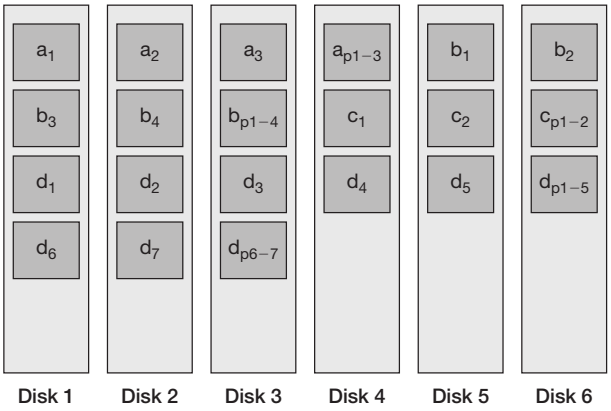


FIGURE 6.56
A RAID-Z layout.

Since all physical storage management is handled in the SPA layer, the DMU is completely independent of concerns such as mirroring and higher RAID levels. An administrator can add another physical disk drive to the set used by SPA, which can then start providing more storage to DMU. No reorganization is necessary at the DMU level, since the SPA level takes care of everything.

A RAID-4 and -5 shortcoming is the cost of small writes. Updates to data occupying a partial stripe on, say, two disks, requires first reading then writing parity information on a third disk. ZFS avoids this shortcoming by using a variant of RAID known as RAID-Z. Rather than use fixed-length stripes, each block forms a stripe, regardless of its length, and each has its own parity. Figure 6.56 shows an example where blocks *a*, *b*, *c*, and *d* on are striped on six disks. Block *a* consists of three sectors and is written to disks 1 through 3, followed by a parity sector written to disk 4. Similarly, block *b* consists of four sectors and is written, along with a parity sector, to the disks; block *c* consists of two sectors and is written with a parity sector. Block *d* consists of seven sectors but, so that there is enough parity to recover on the loss of a disk, has two parity sectors.

Thus small writes are handled efficiently — it's not necessary to read the parity sector before modifying it, since the block's parity depends only on the block itself. The down side is that, to determine the block boundaries (perhaps for recovery purposes), one must traverse the metadata (the shadow-page tree). But, with the use of ditto blocks (as described above), at least one copy of each block of metadata is highly likely to exist.

Another feature of RAID-Z is that it is easy to handle the addition of another disk drive. In Figure 6.57, the example of Figure 6.56 is augmented with a new disk. None of the existing stripes are affected. However, when a new stripe, block *e*, is written, it is striped over all seven drives.

The final area of improvement in ZFS is in caching. Many systems employ LRU caches: they hold the *n* least-recently-used disk blocks. The intent is that *n* is large enough so that the cache holds the blocks needed by the current processes. But suppose a thread reads an *n*-block file sequentially. It fills the cache with the blocks of the file, thereby removing the blocks that are useful to the other processes. But since the file is being read sequentially, none of the blocks are accessed a second time — thus they are wasting space in the cache.

The approach used by ZFS (though developed at IBM) that avoids this problem is *adaptive replacement caching* (ARC) (Megiddo and Modha 2003a) (Megiddo and Modha 2003b). To see how it works, let's first look at a non-adaptive version of it. Suppose we have a cache that holds up to *n* blocks. Let's reserve half its capacity for blocks that have been referenced just once,

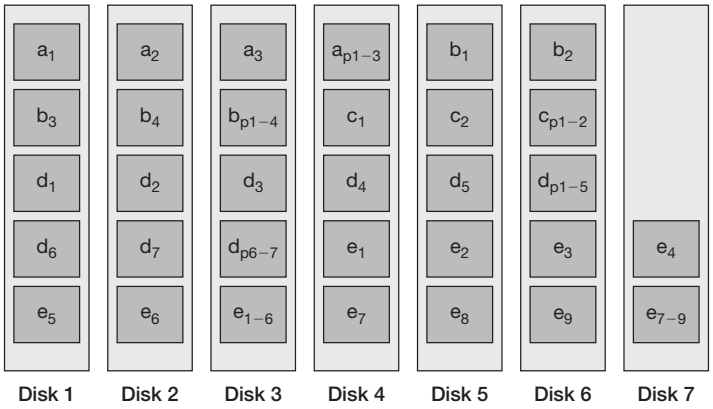


FIGURE 6.57 The RAID-Z layout augmented with a new disk and a new stripe.

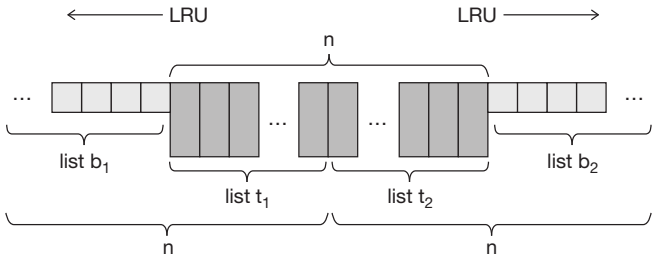


FIGURE 6.58 The adaptive replacement cache (ARC).

and the other half for blocks that have been referenced at least one more time since entering the cache. Both halves of the cache are organized in an LRU fashion. Thus in the example in the previous paragraph, a thread that reads a file sequentially can use no more than half the cache space, since the other half of the cache is reserved for blocks that have been referenced at least twice.

This might be an improvement over a strict LRU cache, but perhaps performance would be better if the cache weren't split in half, but apportioned in some other way. The idea behind ARC is that the portions of the cache used for the two sorts of blocks vary depending on use. If a number of processes are reusing the same set of n blocks and one process is reading a large file sequentially, we'd like to dedicate the entire cache to blocks that have been referenced more than once. On the other hand, if the system is starting up and a set of processes are making their initial references to file-system blocks, we would like to dedicate the entire cache to blocks that have been referenced just once, since none have yet been referenced more than once.

To do all this, ARC maintains, in addition to the n-block cache, two lists of block numbers, each in LRU order (see Figure 6.58). The first list, t_1, refers to blocks that have just entered the cache after being referenced. The second list, t_2, refers to blocks that were re-referenced while in t_1 (or again while in t_2). (Note that t here stands for top and b for bottom.) Thus the maximum total number of block numbers in both lists is n.

So as to determine how many blocks should be in each list, ARC keeps two additional lists of the block numbers of the most recent blocks evicted from the cache. One list, b_1, refers to blocks that have been evicted from the single-reference portion of the cache (and hence were referred to by t_1); the other list, b_2, refers to blocks that have been evicted from the multiple-reference

portion of the cache (and hence were referred to by t_2). The maximum number of entries in the combined t_1 and b_1 lists is n, as for the combined t_2 and b_2 lists.

The adaptive part of the algorithm works as follows. A reference to a block in b_1 is taken to mean that more cache space should be available for blocks that have been referenced just once. So, the maximum size of list t_1 is increased by one[21] (up to n) and the size of t_2 is correspondingly decreased. Similarly, a reference to a block in b_2 is taken to mean that more cache space should be available for blocks that have been referenced more than once. So, the maximum size of list t_2 is increased by one (up to n) and the size of t_1 is correspondingly decreased.

A pseudocode version of the general algorithm is as follows. Note that a cache hit means that the referenced block is not necessarily in the cache, but that it is referred to in one of the four lists. LRU(x) refers to the least-recently-used element of list x; MRU(x) refers to the most-recently-used element of list x.

```
cache miss:
   if t1 is full
         evict LRU(t1) and make it MRU(b1)
   referenced block becomes MRU(t1)
cache hit:
   if in t1 or t2, block becomes MRU(t2)
   otherwise
         if block is referred to by b1
               increase t1 space at expense of t2
         otherwise
               increase t2 space at expense of t1
         if t1 is full
               evict LRU(t1) and make it MRU(b1)
         if t2 is full
               evict LRU(t2) and make it MRU(b2)
         insert block as MRU(t2)
```

6.7 CONCLUSIONS

File systems provide persistent storage. For most computer systems, they provide all the persistent storage, except for that provided by specialized backup devices. Not only must they perform well, in terms of both time and space, but they must also be tolerant of both hardware and software failures.

We started this chapter by studying one of the simplest file systems, S5FS. It served to illustrate all the problems that later file systems have strived to overcome, yet provided an application interface that is almost identical to that provided by modern file systems. Modern systems employ a number of techniques to improve on S5FS. Much was achieved by reorganizing on-disk data structures to take into account disk architecture and to maximize speed of access and transfer. Further progress was made by taking advantage of techniques such as RAID to utilize multiple disks, improving both performance and tolerance of media failures.

While S5FS was notorious for losing data in the event of a crash, modern systems rarely suffer from such problems. Two approaches are used, consistency-preserving updates and transactional updates.

[21] The actual algorithm increases the size of the list by an amount proportional to relative sizes of t_2 and t_1.

Whether used on a personal computer or on a large server, a file system must be easy to administer and be sufficiently scalable to handle what are often growing demands. This has been made possible by techniques based on the dynamic-inode approach as well as support for ever larger files and disks.

File-system technology remains an active area of work, with much competition from both the open-source community and makers of proprietary systems.

6.8 EXERCISES

1. Assume we are using the Rhinopias disk drive discussed in Section 6.1.2.1.

a) What is the maximum transfer rate for a single track?

b) What is the maximum transfer rate for accessing five tracks consecutively (assume head skewing)?

2. Explain why the typical transfer rate achieved on S5FS is far less than the maximum transfer rate of the disk.

3. How do block-based file systems differ from extent-based systems? What is block clustering?

***4.** Assume we're using the Rhinopias disk drive.

a) Suppose we are using S5FS and we've doubled the block size from 512 bytes to 1024 bytes. Is the maximum expected transfer rate roughly doubled? Explain.

b) We've done the same with FFS. Is the maximum expected transfer rate roughly doubled? Explain. Assume that we're using two-way block interleaving: when we allocate successive blocks on a track, one block is skipped. Thus, in the best case, every other block of a track is allocated to a file.

c) Why should we care about the block size in FFS?

5. Explain what is meant by innocuous inconsistency. How does it differ from non-innocuous or real inconsistency?

6. The layout of disk blocks is an important factor in file-system performance. FFS uses the technique of block interleaving to reduce rotational delays. This technique provides a large improvement over the use of consecutive blocks. However, many of today's most advanced file systems, such as WAFL and ZFS, do not use block interleaving.

a) Explain how block interleaving helps FFS reduce the time required to read from disk.

b) Explain how WAFL's design provides fast access without the need for block interleaving.

7. Would log-structured file systems (Section 6.1.4.2) have been feasible in the early 1980s when FFS was designed? Explain. (Consider the cost of the resources required.)

8. What approach did the original FFS use for crash tolerance? What impact did this have on performance? How has this been improved in recent versions?

****9.** Section 6.2.2.1 covers soft updates and mentions that FFS using soft updates may require more disk writes than Async FFS. Give an example in which more disk writes would occur with soft updates than with Async FFS.

****10.** One early application of shadow paging in a file system was in an FFS-like system. All operations between *open* and *close* of a file were treated as part of a single transaction. Thus if the system crashed before the file was closed, the effect was as if nothing had happened

to the file since the *open*. This was done by making a copy of the file's inode when it was opened, and applying all changes to copies of blocks being modified (i.e., using copy-on-write techniques). When the file was closed, the reference to its inode in the directory it appeared in was replaced with a reference to the new, modified inode. Thus in a single disk write, all modifications took effect at once.

There are a few problems with this approach. It doesn't handle files that appear in multiple directories. It also doesn't handle concurrent access to the file by multiple processes. Let's ignore both of these problems (they're fixable, but we won't fix them here!). There's a performance problem: the original blocks of the file were allocated on disk so that the file could be accessed sequentially very quickly. But when we make copies of the disk blocks being modified, these copies aren't necessarily going to be near the original copies. Thus writing these blocks might incur long seek delays. For this reason shadow paging wasn't used in file systems until relatively recently. How might these problems be fixed?

**11. NTFS does a combination of both redo and undo journaling. Section 6.6.4 explains why it's useful to have undo journaling. Why is redo journaling also done?

12. NTFS has a feature called data streams. Describe this feature and give an example of how it might be useful.

**13. Explain how renaming a file can be done using the consistency-preserving approach so that a crash will not result in the loss of the file. Be sure to take into account the link count stored in the file's inode. Note that there may have to be a temporary inconsistency; if so, explain why it will not result in lost data.

*14. A defragmenter is often used with NTFS to improve its performance. Explain how a similar tool might be devised for S5FS to improve its performance. Would the use of this tool bring S5FS's performance up to acceptable levels?

15. Hashing is often used for looking up directories entries. Explain the advantages of extensible hashing over simple static hashing.

**16. Section 6.3.1.1 discussed extensible hashing. Show how deletions of directory entries can be implemented so that buckets are combined where possible, allowing the bucket and indirect bucket arrays to be reduced in size.

*17. RAID level 4, in handling a one-block write request, must write to both a data disk and the check disk. However, it need not read data blocks from the other data disks in order to recompute the check block. Explain why not.

*18. We have an operating system whose only file system is S5FS. Suppose we decide to employ a RAID level 4 or RAID level 5 disk system. Can S5FS take advantage of all the features of these systems? Explain. Assume that the block size of S5FS is equal to the disk-sector size.

19. Explain why Ext3 and NTFS use RAID level 5 rather than RAID level 4. Explain what features of WAFL make it particularly well suited for use with RAID level 4.

**20. An issue with ZFS is reclamation of data blocks. With multiple snapshots in use for each file system, it's not necessarily trivial to determine which blocks are actually free. For example, there might be, in addition to the current version of the file system, three snapshots of it. If we delete the snapshot that's neither the most recent nor the least recent, how can we determine which blocks must be deleted and which must remain? To answer this question, let's go through the following steps.

a) Explain why reference counts are not part of the solution.

b) Suppose each block pointer, in addition to containing a checksum, also contains the time at which the block was created (note that blocks are never modified in place, thus the creation time for a particular file-system block on disk doesn't change). Explain how this information can be used to determine that a block was created after a particular snapshot.

c) Suppose a block is freed from the current file system (but might still be referenced by a snapshot). Explain how it can be determined whether the most recent snapshot is referring to it.

d) If such a block cannot be reclaimed because some snapshot is referring to it, a reference to it is appended to the file-system's *dead list*. When a new snapshot is created, its dead list is set to be that of the current file system, and the current file-system's dead list is set to empty. Let's assume no snapshots have ever been deleted. We say that a snapshot is *responsible for* the existence of a block if the block's birth time came after the creation of the previous snapshot and the block was freed before the birth time of the next snapshot (or of the file system if this is the most recent snapshot). In other words, the only reason the block cannot be reclaimed is because this snapshot exists. Are all such blocks in the next snapshot's dead list (or in the current file system's dead list if this is the most recent snapshot)? Explain.

e) Suppose a snapshot is deleted (the first one to be deleted). How can we determine which blocks to reclaim?

f) We'd like this algorithm to work for subsequent deletions of snapshots. Part d describes an invariant: all blocks for which a snapshot is responsible appear in the next snapshot's dead list (or in the dead list of the file system if it's the most recent snapshot). What else must be done when a snapshot is deleted so that this invariant is maintained (and the algorithm continues to work)?

6.9 REFERENCES

Bityuckiy, A. B. (2005). JFFS3 Design Issues. http://www.linux-mtd.infradead.org/doc/JFFS3design.pdf.

Bonwick, J., M. Ahrens, V. Henson, M. Maybee, M. Shellenbaum (undated technical report, probably 2003). *The Zettabyte File System*, Sun Microsystems.

Brown, M. R., K. N. Kolling, E. A. Taft (1985). The Alpine File System. *ACM Transactions on Computer Systems* **3**(4): 261–293.

Chen, P. M. and D. A. Patterson (1990). Maximizing Performance in a Striped Disk Array. *Proceedings of the 17th Annual Symposium on Computer Architecture*, ACM: 322–331.

Chutani, S., O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, R. N. Sidebotham (1992). The Episode File System. *Proceedings of the USENIX Winter 1992 Technical Conference*: 43–60.

Ganger, G. R., M. K. McKusick, C. A. N. Soules, Y. N. Patt (2000). Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems* **18**(2): 127–153.

Hamming, R. W. (1950). Error Detecting and Correcting Codes. *The Bell System Technical Journal* **29**(2): 147–160.

Hitz, D., J. Lau, M. Malcolm (1994). *File System Design for an NFS File Server Appliance*. Proceedings of USENIX Winter 1994 Technical Conference.

Hunter, A. (2008). A Brief Introduction to the Design of UBIFS. http://www.linux-mtd.infradead.org/doc/ubifs_whitepaper.pdf.

Intel (1998). Understanding the Flash Translation Layer (FTL) Specification. http://www.eetasia.com/ARTICLES/2002MAY/2002MAY07_MEM_AN.PDF

Killian, T. J. (1984). Processes as Files. *USENIX Conference Proceedings*. Salt Lake City, UT, USENIX Association: 203–207.

Leffler, S. J., M. K. McKusick, M. J. Karels, J. S. Quarterman (1989). *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley.

Leventhal, A. (2008). Flash Storage Memory. *Communications of the ACM* **51**(7): 47–51.

Linux-MTD. (2009). UBI — Unsorted Block Images. http://www.linux-mtd.infradead.org/doc/ubi.html.

McKusick, M., W. N. Joy, S. J. Leffler, R. S. Fabry (1984). A Fast File System for UNIX. *ACM Transactions on Computer Systems* **2**(3): 181–197.

McVoy, L. W. and S. R. Kleiman (1991). Extent-like Performance from a UNIX File System. *Proceedings of the Winter 1991 USENIX Technical Conference*. Dallas, USENIX: 12.

Megiddo, N. and D. S. Modha (2003a). ARC: A Self-tuning, Low Overhead Replacement Cache. *Proceedings of FAST 2003: 2nd USENIX Conference on File and Storage Technologies*.

Megiddo, N. and D. S. Modha (2003b). One Up on LRU. *login* **28**(4): 7–11.

Ousterhout, J. K., H. Da Costa, D. Harrison, J. Kunze, M. Kupfer, J. Thompson (1985). A Trace-Driven Analysis of the UNIX 4.2 BSD File System. *Proceedings of the 10th Symposium on Operating Systems Principles*: 15–24.

Patterson, D. A., G. Gibson, R. H. Katz (1988). A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proceedings of the 1980 ACM SIGMOD International Conference on Management of Data*: 109–116.

Phillips, D. (2001). A Directory Index for Ext2. *5th Annual Linux Showcase and Conference*, USENIX: 173–182.

Pike, R., D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, P. Winterbottom (1995). Plan 9 from Bell Labs. *Computing Systems* **8**(3): 221–254.

Radia, S., M. N. Nelson, M. L. Powell (1993). The Spring Name Service. Sun Microsystems Laboratories Technical Report SMLI-93-16.

Rosenblum, M. and J. K. Ousterhout (1992). The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems* **10**(1): 26–52.

Russinovich, M. E. and D. A. Solomon (2005). *Microsoft Windows Internals*, Microsoft Press.

Seltzer, M. I., G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, C. A. Stein (2000). Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *Proceedings of the 2000 USENIX Annual Technical Conference*.

Smith, K. A. and M. Seltzer (1996). A Comparison of FFS Disk Allocation Policies. *Proceedings of the USENIX 1996 Annual Technical Conference*. San Diego.

Sweeney, A., D. Doucette, W. Hu, C. Anderson, M. Nishimoto, G. Peck (1996). Scalability in the XFS File System. *Proceedings of the USENIX 1996 Annual Technical Conference*.

Tanenbaum, A. S. (1987). *Operating Systems: Design and Implementation*. Prentice Hall.

Ts'o, T. Y. and S. Tweedie (2002). Planned Extensions to the Linux Ext2/Ext3 Filesystem. *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*.

Tweedie, S. C. (1998). Journaling the Linux ext2fs File System. *Proceedings of the 4th Annual LinuxExp,* Durham, N.C.

Walker, B., G. Popek, R. English, C. Kline, G. Thiel (1983). The LOCUS Distributed Operating System. *ACM Symposium on Operating Systems Principles*: 49–70.

Wikipedia. (2009). Flash Memory. *Wikipedia*, http://en.wikipedia.org/wiki/Flash_memory.

Woodhouse, D. (2001). JFFS: The Journalling Flash File System. http://sourceware.org/jffs2/jffs2-html.