8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

5.3 FILE SYSTEM IMPLEMENTATION

Now it is time to turn from the user's view of the file system to the implementer's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementers are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

5.3.1 File System Layout

File systems usually are stored on disks. We looked at basic disk layout in Chap. 2, in the section on bootstrapping MINIX 3. To review this material briefly, most disks can be divided up into partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR** (**Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table may be marked as active. When the computer is booted, the BIOS reads in and executes the code in the MBR. The first thing the MBR program does is locate the active partition, read in its first block, called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the some time in the future, so reserving a boot block is a good idea anyway.

The above description must be true, regardless of the operating system in use, for any hardware platform on which the BIOS is to be able to start more than one operating system. The terminology may differ with different operating systems. For instance the master boot record may sometimes be called the **IPL** (**Initial Program Loader**), **Volume Boot Code**, or simply **masterboot**. Some operating

systems do not require a partition to be marked active to be booted, and provide a menu for the user to choose a partition to boot, perhaps with a timeout after which a default choice is automatically taken. Once the BIOS has loaded an MBR or boot sector the actions may vary. For instance, more than one block of a partition may be used to contain the program that loads the operating system. The BIOS can be counted on only to load the first block, but that block may then load additional blocks if the implementer of the operating system writes the boot block that way. An implementer can also supply a custom MBR, but it must work with a standard partition table if multiple operating systems are to be supported.

On PC-compatible systems there can be no more than four **primary partitions** because there is only room for a four-element array of partition descriptors between the master boot record and the end of the first 512-byte sector. Some operating systems allow one entry in the partition table to be an **extended partition** which points to a linked list of **logical partitions**. This makes it possible to have any number of additional partitions. The BIOS cannot start an operating system from a logical partition, so initial startup from a primary partition is required to load code that can manage logical partitions.

An alternative to extended partitions is used by MINIX 3, which allows a partition to contain a **subpartition table**. An advantage of this is that the same code that manages a primary partition table can manage a subpartition table, which has the same structure. Potential uses for subpartitions are to have different ones for the root device, swapping, the system binaries, and the users' files. In this way, problems in one subpartition cannot propagate to another one, and a new version of the operating system can be easily installed by replacing the contents of some of the subpartitions but not all.

Not all disks are partitioned. Floppy disks usually start with a boot block in the first sector. The BIOS reads the first sector of a disk and looks for a magic number which identifies it as valid executable code, to prevent an attempt to execute the first sector of an unformatted or corrupted disk. A master boot record and a boot block use the same magic number, so the executable code may be either one. Also, what we say here is not limited to electromechanical disk devices. A device such as a camera or personal digital assistant that uses nonvolatile (e.g., flash) memory typically has part of the memory organized to simulate a disk.

Other than starting with a boot block, the layout of a disk partition varies considerably from file system to file system. A UNIX-like file system will contain some of the items shown in Fig. 5-8. The first one is the **superblock**. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched.

Next might come information about free blocks in the file system. This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file and where its blocks are located. After that might come the root directory, which contains the top of the file system tree. Finally, the remainder of the disk typically contains all the other directories and files.

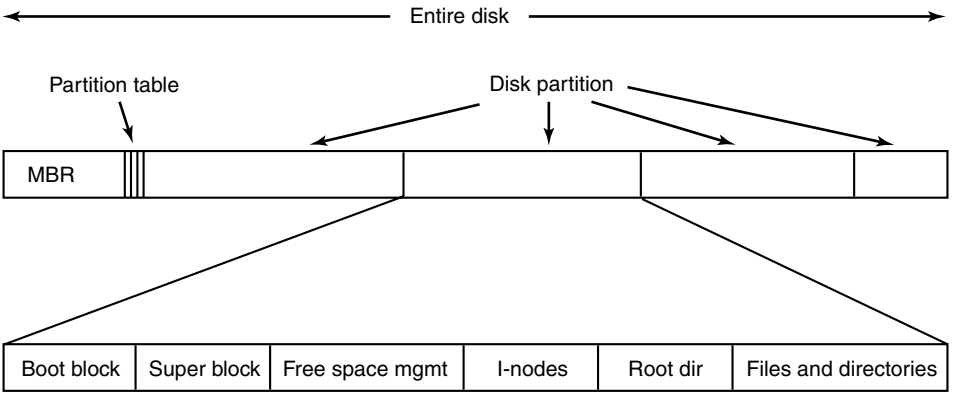

Figure 5-8. A possible file system layout.

5.3.2 Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. Contiguous disk space allocation has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.

Second, the read performance is excellent because the entire file can be read from the disk in a single operation. Only one seek is needed (to the first block). After that, no more seeks or rotational delays are needed so data come in at the full bandwidth of the disk. Thus contiguous allocation is simple to implement and has high performance.

Unfortunately, contiguous allocation also has a major drawback: in time, the disk becomes fragmented, consisting of files and holes. Initially, this fragmentation is not a problem since each new file can be written at the end of disk, following the previous one. However, eventually the disk will fill up and it will become necessary to either compact the disk, which is prohibitively expensive, or to reuse the free space in the holes. Reusing the space requires maintaining a list of holes, which is doable. However, when a new file is to be created, it is necessary to know its final size in order to choose a hole of the correct size to place it in.

As we mentioned in Chap. 1, history may repeat itself in computer science as new generations of technology occur. Contiguous allocation was actually used on magnetic disk file systems years ago due to its simplicity and high performance (user friendliness did not count for much then). Then the idea was dropped due to the nuisance of having to specify final file size at file creation time. But with the advent of CD-ROMs, DVDs, and other write-once optical media, suddenly contiguous files are a good idea again. For such media, contiguous allocation is feasible and, in fact, widely used. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system. It is thus important to study old systems and ideas that were conceptually clean and simple because they may be applicable to future systems in surprising ways.

Linked List Allocation

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 5-9. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

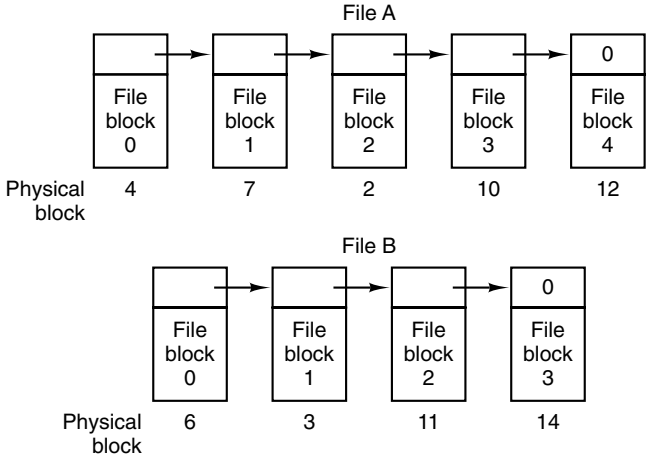

Figure 5-9. Storing a file as a linked list of disk blocks.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block of each file). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there.

On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the $n - 1$ blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied to a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

Linked List Allocation Using a Table in Memory

Both disadvantages of the linked list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 5-10 shows what the table looks like for the example of Fig. 5-9. In both figures, we have two files. File *A* uses disk blocks 4, 7, 2, 10, and 12, in that order, and file *B* uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 5-10, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., −1) that is not a valid block number. Such a table in main memory is called a **FAT** (**File Allocation Table**).

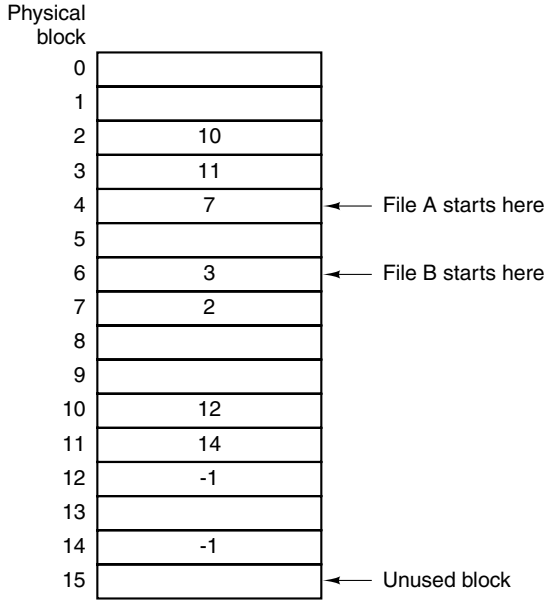

Figure 5-10. Linked list allocation using a file allocation table in main memory.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed

without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

The primary disadvantage of this method is that the entire table must be in memory all the time. With a 20-GB disk and a 1-KB block size, the table needs 20 million entries, one for each of the 20 million disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 60 MB or 80 MB of main memory all the time, depending on whether the system is optimized for space or time. Conceivably the table could be put in pageable memory, but it would still occupy a great deal of virtual memory and disk space as well as generating paging traffic. MS-DOS and Windows 98 use only FAT file systems and later versions of Windows also support it.

I-Nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node** (**index-node**), which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 5-11. Given the i-node, it is then possible to find all the blocks of the file. The big advantage of this scheme over linked files using an in-memory table is that the i-node need only be in memory when the corresponding file is open. If each i-node occupies n bytes and a maximum of k files may be open at once, the total memory occupied by the array holding the i-nodes for the open files is only kn bytes. Only this much space need be reserved in advance.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 1 GB or 10 GB or 100 GB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of an **indirect block** containing more disk block addresses. This idea can be extended to use **double indirect blocks** and **triple indirect blocks**, as shown in Fig. 5-11.

5.3.3 Implementing Directories

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry. Finding a directory entry means, of course, that the root directory must be

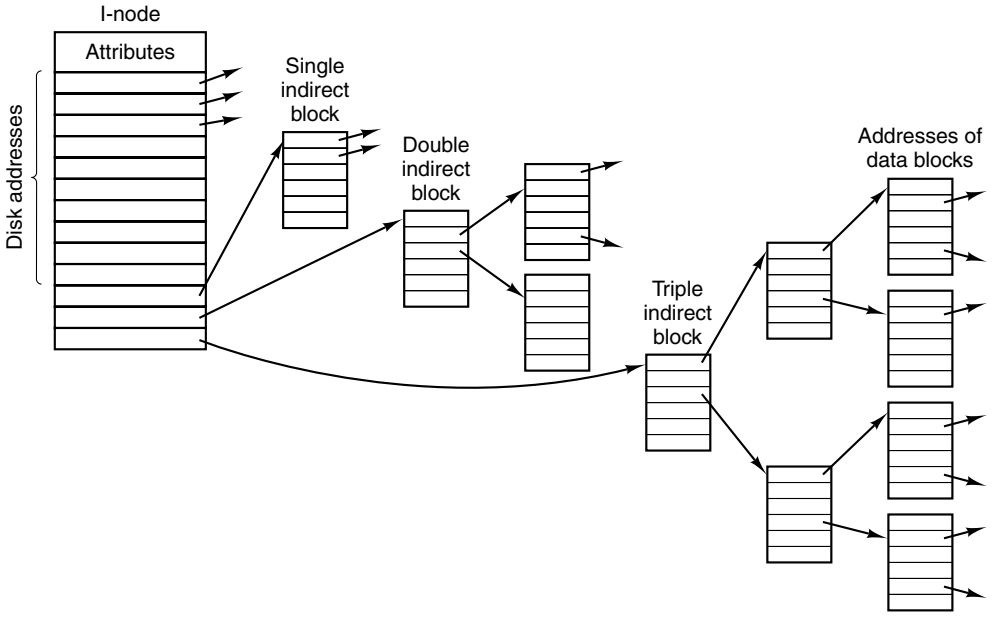

Figure 5-11. An i-node with three levels of indirect blocks.

located first. The root directory may be in a fixed location relative to the start of a partition. Alternatively, its position may be determined from other information, for instance, in a classic UNIX file system the superblock contains information about the size of the file system data structures that precede the data area. From the superblock the location of the i-nodes can be found. The first i-node will point to the root directory, which is created when a UNIX file system is made. In Windows XP, information in the boot sector (which is really much bigger than one sector) locates the **MFT** (**Master File Table**), which is used to locate other parts of the file system.

Once the root directory is located a search through the directory tree finds the desired directory entry. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (contiguous allocation), the number of the first block (both linked list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data.

A closely related issue is where the attributes should be stored. Every file system maintains file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. In its simplest form, a directory consists of a list of fixed-size entries, one per file, containing a (fixed-length) file name, a structure of the

file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are, as we saw in Fig. 5-5(a).

For systems that use i-nodes, another possibility for storing the attributes is in the i-nodes, rather than in the directory entries, as in Fig. 5-5(b). In this case, the directory entry can be shorter: just a file name and an i-node number.

Shared Files

In Chap. 1 we briefly mentioned **links** between files, which make it easy for several users working together on a project to share files. Figure 5-12 shows the file system of Fig. 5-6(c) again, only with one of *C*'s files now present in one of *B*'s directories as well.

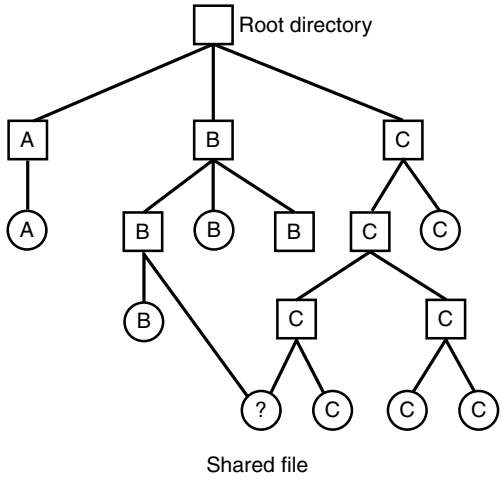

Figure 5-12. File system containing a shared file.

In UNIX the use of i-nodes for storing file attributes makes sharing easy; any number of directory entries can point to a single i-node. The i-node contains a field which is incremented when a new link is added, and which is decremented when a link is deleted. Only when the link count reaches zero are the actual data and the i-node itself deleted.

This kind of link is sometimes called a **hard link**. Sharing files using hard links is not always possible. A major limitation is that directories and i-nodes are data structures of a single file system (partition), so a directory in one file system cannot point to an i-node on another file system. Also, a file can have only one owner and one set of permissions. If the owner of a shared file deletes his own directory entry for that file, another user could be stuck with a file in his directory that he cannot delete if the permissions do not allow it.

An alternative way to share files is to create a new kind of file whose data is the path to another file. This kind of link will work across mounted file systems. In fact, if a means is provided for path names to include network addresses, such a link can refer to a file on a different computer. This second kind of link is called a **symbolic link** in UNIX-like systems, a **shortcut** in Windows, and an **alias** in Apple's Mac OS. Symbolic links can be used on systems where attributes are stored within directory entries. A little thought should convince you that multiple directory entries containing file attributes would be difficult to synchronize. Any change to a file would have to affect every directory entry for that file. But the extra directory entries for symbolic links do not contain the attributes of the file to which they point. A disadvantage of symbolic links is that when a file is deleted, or even just renamed, a link becomes an orphan.

Directories in Windows 98

The file system of the original release of Windows 95 was identical to the MS-DOS file system, but a second release added support for longer file names and bigger files. We will refer to this as the Windows 98 file system, even though it is found on some Windows 95 systems. Two types of directory entry exist in Windows 98. We will call the first one, shown in Fig. 5-13, a base entry.

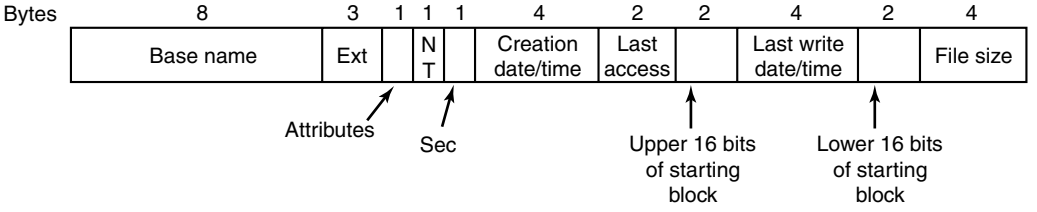

Figure 5-13. A Windows 98 base directory entry.

The base directory entry has all the information that was in the directory entries of older Windows versions, and more. The 10 bytes starting with the *NT* field are additions to the older Windows 95 structure, which fortunately (or more likely deliberately, with later improvement in mind) were not previously used. The most important upgrade is the field that increases the number of bits available for pointing to the starting block from 16 to 32. This increases the maximum potential size of the file system from 2^{16} blocks to 2^{32} blocks.

This structure provides only for the old-style 8 + 3 character filenames inherited from MS-DOS (and CP/M). How about long file names? The answer to the problem of providing long file names while retaining compatibility with the older systems was to use additional directory entries. Fig. 5-14 shows an alternative form of directory entry that can contain up to 13 characters of a long file name. For files with long names a shortened form of the name is generated automatically

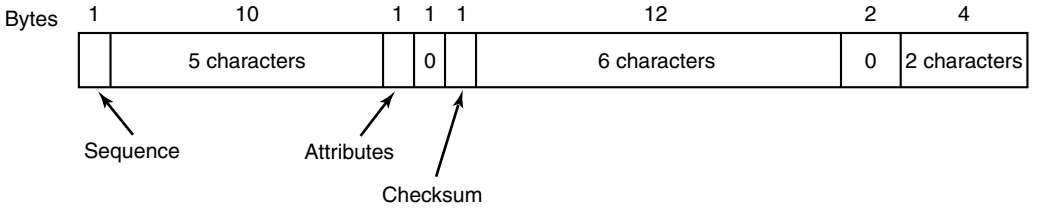

Figure 5-14. An entry for (part of) a long file name in Windows 98.

and placed in the *Base name* and *Ext* fields of an Fig. 5-13-style base directory entry. As many entries like that of Fig. 5-14 as are needed to contain the long file name are placed before the base entry, in reverse order. The *Attributes* field of each long name entry contains the value 0x0F, which is an impossible value for older (MS-DOS and Windows 95) files systems, so these entries will be ignored if the directory is read by an older system (on a floppy disk, for instance). A bit in the *Sequence* field tells the system which is the last entry.

If this seems rather complex, well, it is. Providing backward compatibility so an earlier simpler system can continue to function while providing additional features for a newer system is likely to be messy. A purist might decide not to go to so much trouble. However, a purist would probably not become rich selling new versions of operating systems.

Directories in UNIX

The traditional UNIX directory structure is extremely simple, as shown in Fig. 5-15. Each entry contains just a file name and its i-node number. All the information about the type, size, times, ownership, and disk blocks is contained in the i-node. Some UNIX systems have a different layout, but in all cases, a directory entry ultimately contains only an ASCII string and an i-node number.

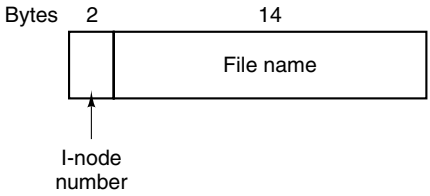

Figure 5-15. A Version 7 UNIX directory entry.

When a file is opened, the file system must take the file name supplied and locate its disk blocks. Let us consider how the path name */usr/ast/mbox* is looked up. We will use UNIX as an example, but the algorithm is basically the same for all hierarchical directory systems. First the system locates the root directory. The

i-nodes form a simple array which is located using information in the superblock. The first entry in this array is the i-node of the root directory.

The file system looks up the first component of the path, *usr*, in the root directory to find the i-node number of the file */usr/*. Locating an i-node from its number is straightforward, since each one has a fixed location relative to the first one. From this i-node, the system locates the directory for */usr/* and looks up the next component, *ast*, in it. When it has found the entry for *ast*, it has the i-node for the directory */usr/ast/*. From this i-node it can find the directory itself and look up *mbox*. The i-node for this file is then read into memory and kept there until the file is closed. The lookup process is illustrated in Fig. 5-16.

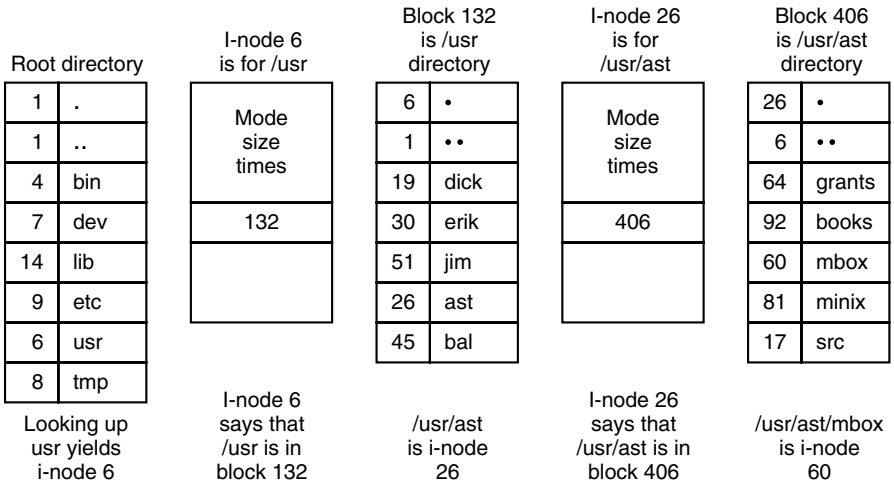

Figure 5-16. The steps in looking up */usr/ast/mbox*.

Relative path names are looked up the same way as absolute ones, only starting from the working directory instead of starting from the root directory. Every directory has entries for . and .. which are put there when the directory is created. The entry . has the i-node number for the current directory, and the entry for .. has the i-node number for the parent directory. Thus, a procedure looking up *../dick/prog.c* simply looks up .. in the working directory, finds the i-node number for the parent directory, and searches that directory for *dick*. No special mechanism is needed to handle these names. As far as the directory system is concerned, they are just ordinary ASCII strings, just the same as any other names.

Directories in NTFS

Microsoft's **NTFS** (**New Technology File System**) is the default file system. We do not have space for a detailed description of NTFS, but will just briefly look at some of the problems NTFS deals with and the solutions used.

One problem is long file and path names. NTFS allows long file names (up to 255 characters) and path names (up to 32,767 characters). But since older versions of Windows cannot read NTFS file systems, a complicated backward-compatible directory structure is not needed, and filename fields are variable length. Provision is made to have a second 8 + 3 character name so an older system can access NTFS files over a network.

NTFS provides for multiple character sets by using Unicode for filenames. Unicode uses 16 bits for each character, enough to represent multiple languages with very large symbol sets (e.g., Japanese). But using multiple languages raises problems in addition to representation of different character sets. Even among Latin-derived languages there are subtleties. For instance, in Spanish some combinations of two characters count as single characters when sorting. Words beginning with "ch" or "ll" should appear in sorted lists after words that begin with "cz" or "lz", respectively. The problem of case mapping is more complex. If the default is to make filenames case sensitive, there may still be a need to do case-insensitive searches. For Latin-based languages it is obvious how to do that, at least to native users of these languages. In general, if only one language is in use, users will probably know the rules. However, Unicode allows a mixture of languages: Greek, Russian, and Japanese filenames could all appear in a single directory at an international organization. The NTFS solution is an attribute for each file that defines the case conventions for the language of the filename.

More attributes is the NTFS solution to many problems. In UNIX, a file is a sequence of bytes. In NTFS a file is a collection of attributes, and each attribute is a stream of bytes. The basic NTFS data structure is the **MFT** (**Master File Table**) that provides for 16 attributes, each of which can have a length of up to 1 KB within the MFT. If that is not enough, an attribute within the MFT can be a header that points to an additional file with an extension of the attribute values. This is known as a **nonresident attribute**. The MFT itself is a file, and it has an entry for every file and directory in the file system. Since it can grow very large, when an NTFS file system is created about 12.5% of the space on the partition is reserved for growth of the MFT. Thus it can grow without becoming fragmented, at least until the initial reserved space is used, after which another large chunk of space will be reserved. So if the MFT becomes fragmented it will consists of a small number of very large fragments.

What about data in NTFS? Data is just another attribute. In fact an NTFS file may have more than one data stream. This feature was originally provided to allow Windows servers to serve files to Apple MacIntosh clients. In the original MacIntosh operating system (through Mac OS 9) all files had two data streams, called the resource fork and the data fork. Multiple data streams have other uses, for instance a large graphic image may have a smaller thumbnail image associated with it. A stream can contain up to 2^{64} bytes. At the other extreme, NTFS can handle small files by putting a few hundred bytes in the attribute header. This is called an **immediate file** (Mullender and Tanenbaum, 1984).

We have only touched upon a few ways that NTFS deals with issues not addressed by older and simpler file systems. NTFS also provides features such as a sophisticated protection system, encryption, and data compression. Describing all these features and their implementation would require much more space than we can spare here. For a more throrough look at NTFS see Tanenbaum (2001) or look on the World Wide Web for more information.

5.3.4 Disk Space Management

Files are normally stored on disk, so management of disk space is a major concern to file system designers. Two general strategies are possible for storing an n byte file: n consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same trade-off is present in memory management systems between pure segmentation and paging.

As we have seen, storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it will probably have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises of how big the blocks should be. Given the way disks are organized, the sector, the track and the cylinder are obvious candidates for the unit of allocation (although these are all device dependent, which is a minus). In a paging system, the page size is also a major contender. However, having a large allocation unit, such as a cylinder, means that every file, even a 1-byte file, ties up an entire cylinder.

On the other hand, using a small allocation unit means that each file will consist of many blocks. Reading each block normally requires a seek and a rotational delay, so reading a file consisting of many small blocks will be slow.

As an example, consider a disk with 131,072 bytes/track, a rotation time of 8.33 msec, and an average seek time of 10 msec. The time in milliseconds to read a block of k bytes is then the sum of the seek, rotational delay, and transfer times:

$$10 + 4.165 + (k/131072) \times 8.33$$

The solid curve of Fig. 5-17 shows the data rate for such a disk as a function of block size.

To compute the space efficiency, we need to make an assumption about the mean file size. An early study showed that the mean file size in UNIX environments is about 1 KB (Mullender and Tanenbaum, 1984). A measurement made in

2005 at the department of one of the authors (AST), which has 1000 users and over 1 million UNIX disk files, gives a median size of 2475 bytes, meaning that half the files are smaller than 2475 bytes and half are larger. As an aside, the median is a better metric than the mean because a very small number of files can influence the mean enormously, but not the median. A few 100-MB hardware manuals or a promotional videos or to can greatly skew the mean but have little effect on the median.

In an experiment to see if Windows NT file usage was appreciably different from UNIX file usage, Vogels (1999) made measurements on files at Cornell University. He observed that NT file usage is more complicated than on UNIX. He wrote:

> *When we type a few characters in the* notepad *text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional open and close sequences.*

Nevertheless, he observed a median size (weighted by usage) of files just read at 1 KB, files just written as 2.3 KB and files read and written as 4.2 KB. Given the fact that Cornell has considerable large-scale scientific computing and the difference in measurement technique (static versus dynamic), the results are reasonably consistent with a median file size of around 2 KB.

For simplicity, let us assume all files are 2 KB, which leads to the dashed curve in Fig. 5-17 for the disk space efficiency.

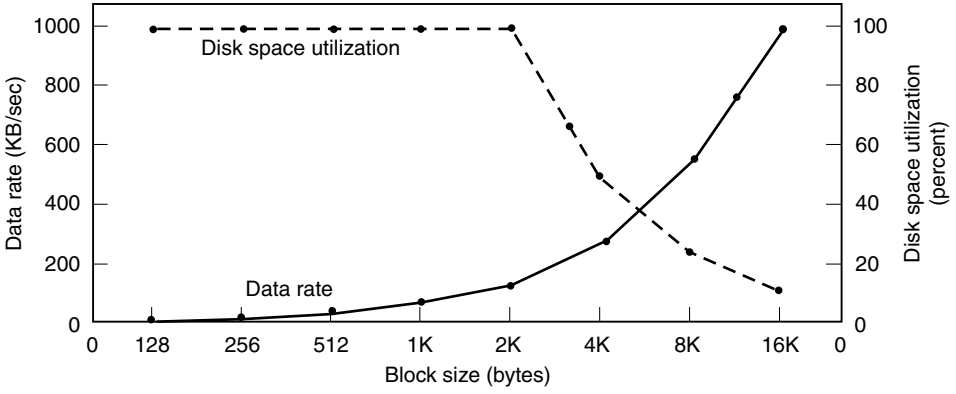

Figure 5-17. The solid curve (left-hand scale) gives the data rate of a disk. The dashed curve (right-hand scale) gives the disk space efficiency. All files are 2 KB.

The two curves can be understood as follows. The access time for a block is completely dominated by the seek time and rotational delay, so given that it is going to cost 14 msec to access a block, the more data that are fetched, the better. Hence the data rate goes up with block size (until the transfers take so long that the transfer time begins to dominate). With small blocks that are powers of two

and 2-KB files, no space is wasted in a block. However, with 2-KB files and 4 KB or larger blocks, some disk space is wasted. In reality, few files are a multiple of the disk block size, so some space is always wasted in the last block of a file.

What the curves show, however, is that performance and space utilization are inherently in conflict. Small blocks are bad for performance but good for disk space utilization. A compromise size is needed. For this data, 4 KB might be a good choice, but some operating systems made their choices a long time ago, when the disk parameters and file sizes were different. For UNIX, 1 KB is commonly used. For MS-DOS the block size can be any power of two from 512 bytes to 32 KB, but is determined by the disk size and for reasons unrelated to these arguments (the maximum number of blocks on a disk partition is 2^{16}, which forces large blocks on large disks).

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 5-18. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is needed for the pointer to the next block). A 256-GB disk needs a free list of maximum 1,052,689 blocks to hold all 2^{28} disk block numbers. Often free blocks are used to hold the free list.

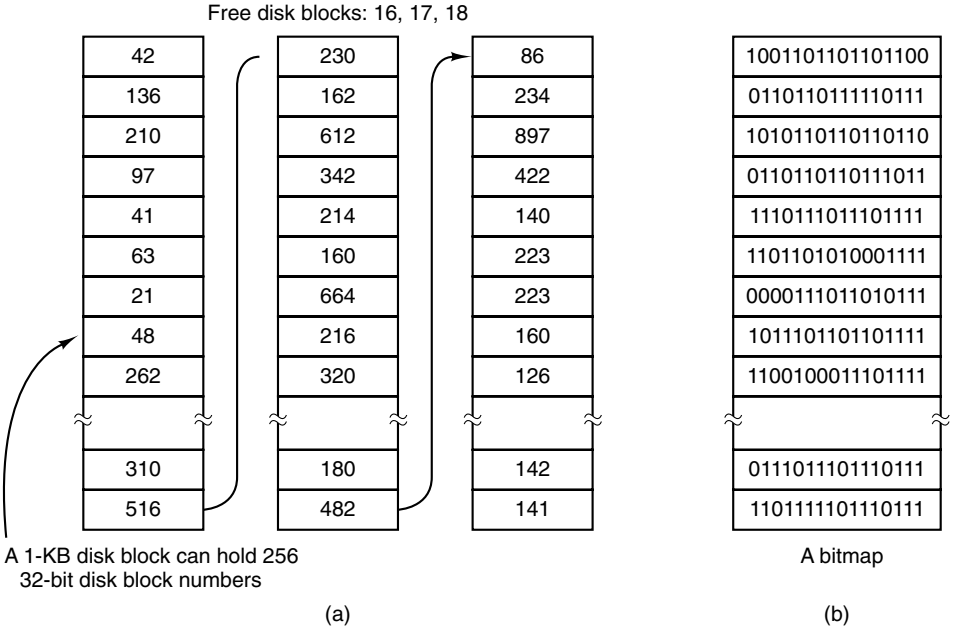

Figure 5-18. (a) Storing the free list on a linked list. (b) A bitmap.

The other free space management technique is the bitmap. A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). A 256-GB disk has 2^{28} 1-KB blocks and thus requires 2^{28} bits for the map, which requires 32,768 blocks. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, versus 32 bits in the linked list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked list scheme require fewer blocks than the bitmap. On the other hand, if there are many blocks free, some of them can be borrowed to hold the free list without any loss of disk capacity.

When the free list method is used, only one block of pointers need be kept in main memory. When a file is created, the needed blocks are taken from the block of pointers. When it runs out, a new block of pointers is read in from the disk. Similarly, when a file is deleted, its blocks are freed and added to the block of pointers in main memory. When this block fills up, it is written to disk.

5.3.5 File System Reliability

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss. Inexpensive personal computers can even be replaced within an hour by just going to the dealer (except at universities, where issuing a purchase order takes three committees, five signatures, and 90 days).

If a computer's file system is irrevocably lost, whether due to hardware, software, or rats gnawing on the backup tapes, restoring all the information will be difficult and time consuming at best, and in many cases will be impossible. For the people whose programs, documents, customer files, tax records, databases, marketing plans, or other data are gone forever, the consequences can be catastrophic. While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information. In this section we will look at some of the issues involved in safeguarding the file system.

Floppy disks are generally perfect when they leave the factory, but they can develop bad blocks during use. It is arguable that this is more likely now than it was in the days when floppy disks were more widely used. Networks and large capacity removable devices such as writeable CDs have led to floppy disks being used infrequently. Cooling fans draw air and airborne dust in through floppy disk drives, and a drive that has not been used for a long time may be so dirty that it ruins the next disk that is inserted. A floppy drive that is used frequently is less likely to damage a disk.

Hard disks frequently have bad blocks right from the start: it is just too expensive to manufacture them completely free of all defects. As we saw in Chap. 3, bad blocks on hard disks are generally handled by the controller by replacing bad

sectors with spares provided for that purpose. On these disks, tracks are at least one sector bigger than needed, so that at least one bad spot can be skipped by leaving it in a gap between two consecutive sectors. A few spare sectors are provided on each cylinder so the controller can do automatic sector remapping if it notices that a sector needs more than a certain number of retries to be read or written. Thus the user is usually unaware of bad blocks or their management. Nevertheless, when a modern IDE or SCSI disk fails, it will usually fail horribly, because it has run out of spare sectors. SCSI disks provide a "recovered error" when they remap a block. If the driver notes this and displays a message on the monitor the user will know it is time to buy a new disk when these messages begin to appear frequently.

A simple software solution to the bad block problem exists, suitable for use on older disks. This approach requires the user or file system to carefully construct a file containing all the bad blocks. This technique removes them from the free list, so they will never occur in data files. As long as the bad block file is never read or written, no problems will arise. Care has to be taken during disk backups to avoid reading this file and trying to back it up.

Backups

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo a deathbed conversion. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, usually to tape. Modern tapes hold tens or sometimes even hundreds of gigabytes and cost pennies per gigabyte. Nevertheless, making backups is not quite as trivial as it sounds, so we will examine some of the related issues below.

Backups to tape are generally made to handle one of two potential problems:

1. Recover from disaster.
2. Recover from stupidity.

The first one covers getting the computer running again after a disk crash, fire, flood, or other natural catastrophe. In practice, these things do not happen very often, which is why many people do not bother with backups. These people also tend not to have fire insurance on their houses for the same reason.

The second reason is that users often accidentally remove files that they later need again. This problem occurs so often that when a file is "removed" in Windows, it is not deleted at all, but just moved to a special directory, the **recycle bin**, so it can be fished out and restored easily later. Backups take this principle further and allow files that were removed days, even weeks ago, to be restored from old backup tapes.

Making a backup takes a long time and occupies a large amount of space, so doing it efficiently and conveniently is important. These considerations raise the

following issues. First, should the entire file system be backed up or only part of it? At many installations, the executable (binary) programs are kept in a limited part of the file system tree. It is not necessary to back up these files if they can all be reinstalled from the manufacturers' CD-ROMs. Also, most systems have a directory for temporary files. There is usually no reason to back it up either. In UNIX, all the special files (I/O devices) are kept in a directory */dev/*. Not only is backing up this directory not necessary, it is downright dangerous because the backup program would hang forever if it tried to read each of these to completion. In short, it is usually desirable to back up only specific directories and everything in them rather than the entire file system.

Second, it is wasteful to back up files that have not changed since the last backup, which leads to the idea of **incremental dumps**. The simplest form of incremental dumping is to make a complete dump (backup) periodically, say weekly or monthly, and to make a daily dump of only those files that have been modified since the last full dump. Even better is to dump only those files that have changed since they were last dumped. While this scheme minimizes dumping time, it makes recovery more complicated because first the most recent full dump has to be restored, followed by all the incremental dumps in reverse order, oldest one first. To ease recovery, more sophisticated incremental dumping schemes are often used.

Third, since immense amounts of data are typically dumped, it may be desirable to compress the data before writing them to tape. However, with many compression algorithms, a single bad spot on the backup tape can foil the decompression algorithm and make an entire file or even an entire tape unreadable. Thus the decision to compress the backup stream must be carefully considered.

Fourth, it is difficult to perform a backup on an active file system. If files and directories are being added, deleted, and modified during the dumping process, the resulting dump may be inconsistent. However, since making a dump may take hours, it may be necessary to take the system offline for much of the night to make the backup, something that is not always acceptable. For this reason, algorithms have been devised for making rapid snapshots of the file system state by copying critical data structures, and then requiring future changes to files and directories to copy the blocks instead of updating them in place (Hutchinson et al., 1999). In this way, the file system is effectively frozen at the moment of the snapshot, so it can be backed up at leisure afterward.

Fifth and last, making backups introduces many nontechnical problems into an organization. The best online security system in the world may be useless if the system administrator keeps all the backup tapes in his office and leaves it open and unguarded whenever he walks down the hall to get output from the printer. All a spy has to do is pop in for a second, put one tiny tape in his pocket, and saunter off jauntily. Goodbye security. Also, making a daily backup has little use if the fire that burns down the computers also burns up all the backup tapes. For this reason, backup tapes should be kept off-site, but that introduces more security

risks. For a thorough discussion of these and other practical administration issues, see Nemeth et al. (2001). Below we will discuss only the technical issues involved in making file system backups.

Two strategies can be used for dumping a disk to tape: a physical dump or a logical dump. A **physical dump** starts at block 0 of the disk, writes all the disk blocks onto the output tape in order, and stops when it has copied the last one. Such a program is so simple that it can probably be made 100% bug free, something that can probably not be said about any other useful program.

Nevertheless, it is worth making several comments about physical dumping. For one thing, there is no value in backing up unused disk blocks. If the dumping program can get access to the free block data structure, it can avoid dumping unused blocks. However, skipping unused blocks requires writing the number of each block in front of the block (or the equivalent), since it is no longer true that block *k* on the tape was block *k* on the disk.

A second concern is dumping bad blocks. If all bad blocks are remapped by the disk controller and hidden from the operating system as we described in Sec. 5.4.4, physical dumping works fine. On the other hand, if they are visible to the operating system and maintained in one or more "bad block files" or bitmaps, it is absolutely essential that the physical dumping program get access to this information and avoid dumping them to prevent endless disk read errors during the dumping process.

The main advantages of physical dumping are simplicity and great speed (basically, it can run at the speed of the disk). The main disadvantages are the inability to skip selected directories, make incremental dumps, and restore individual files upon request. For these reasons, most installations make logical dumps.

A **logical dump** starts at one or more specified directories and recursively dumps all files and directories found there that have changed since some given base date (e.g., the last backup for an incremental dump or system installation for a full dump). Thus in a logical dump, the dump tape gets a series of carefully identified directories and files, which makes it easy to restore a specific file or directory upon request.

In order to be able to properly restore even a single file correctly, all information needed to recreate the path to that file must be saved to the backup medium. Thus the first step in doing a logical dump is doing an analysis of the directory tree. Obviously, we need to save any file or directory that has been modified. But for proper restoration, all directories, even unmodified ones, that lie on the path to a modified file or directory must be saved. This means saving not just the data (file names and pointers to i-nodes), all the attributes of the directories must be saved, so they can be restored with the original permissions. The directories and their attributes are written to the tape first, and then modified files (with their attributes) are saved. This makes it possible to restore the dumped files and directories to a fresh file system on a different computer. In this way, the dump and restore programs can be used to transport entire file systems between computers.

A second reason for dumping unmodified directories above modified files is to make it possible to incrementally restore a single file (possibly to handle recovery from accidental deletion). Suppose that a full file system dump is done Sunday evening and an incremental dump is done on Monday evening. On Tuesday the directory */usr/jhs/proj/nr3/* is removed, along with all the directories and files under it. On Wednesday morning bright and early, a user wants to restore the file */usr/jhs/proj/nr3/plans/summary* However, is not possible to just restore the file *summary* because there is no place to put it. The directories *nr3/* and *plans/* must be restored first. To get their owners, modes, times, etc., correct, these directories must be present on the dump tape even though they themselves were not modified since the previous full dump.

Restoring a file system from the dump tapes is straightforward. To start with, an empty file system is created on the disk. Then the most recent full dump is restored. Since the directories appear first on the tape, they are all restored first, giving a skeleton of the file system. Then the files themselves are restored. This process is then repeated with the first incremental dump made after the full dump, then the next one, and so on.

Although logical dumping is straightforward, there are a few tricky issues. For one, since the free block list is not a file, it is not dumped and hence it must be reconstructed from scratch after all the dumps have been restored. Doing so is always possible since the set of free blocks is just the complement of the set of blocks contained in all the files combined.

Another issue is links. If a file is linked to two or more directories, it is important that the file is restored only one time and that all the directories that are supposed to point to it do so.

Still another issue is the fact that UNIX files may contain holes. It is legal to open a file, write a few bytes, then seek to a distant file offset and write a few more bytes. The blocks in between are not part of the file and should not be dumped and not be restored. Core dump files often have a large hole between the data segment and the stack. If not handled properly, each restored core file will fill this area with zeros and thus be the same size as the virtual address space (e.g., 2^{32} bytes, or worse yet, 2^{64} bytes).

Finally, special files, named pipes, and the like should never be dumped, no matter in which directory they may occur (they need not be confined to */dev/*). For more information about file system backups, see Chervenak et al. (1998) and Zwicky (1991).

File System Consistency

Another area where reliability is an issue is file system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can

be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or blocks containing the free list.

To deal with the problem of inconsistent file systems, most computers have a utility program that checks file system consistency. For example, UNIX has *fsck* and Windows has *chkdsk* (or *scandisk* in earlier versions). This utility can be run whenever the system is booted, especially after a crash. The description below tells how *fsck* works. *Chkdsk* is somewhat different because it works on a different file system, but the general principle of using the file system's inherent redundancy to repair it is still valid. All file system checkers verify each file system (disk partition) independently of the other ones.

Two kinds of consistency checks can be made: blocks and files. To check for block consistency, the program builds two tables, each one containing a counter for each block, initially set to 0. The counters in the first table keep track of how many times each block is present in a file; the counters in the second table record how often each block is present in the free list (or the bitmap of free blocks).

The program then reads all the i-nodes. Starting from an i-node, it is possible to build a list of all the block numbers used in the corresponding file. As each block number is read, its counter in the first table is incremented. The program then examines the free list or bitmap, to find all the blocks that are not in use. Each occurrence of a block in the free list results in its counter in the second table being incremented.

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 5-19(a). However, as a result of a crash, the tables might look like Fig. 5-19(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they do waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

Another situation that might occur is that of Fig. 5-19(c). Here we see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 5-19(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.

The appropriate action for the file system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files. In this way, the information content of the files is unchanged (although almost assuredly one is garbled), but the file system structure is at least made consistent. The error should be reported, to allow the user to inspect the damage.

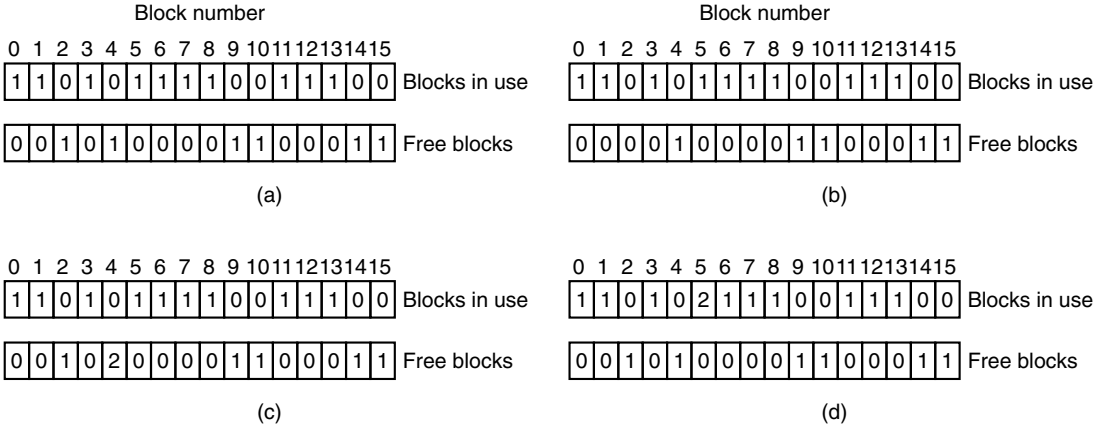

Figure 5-19. File system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

In addition to checking to see that each block is properly accounted for, the file system checker also checks the directory system. It, too, uses a table of counters, but these are per file, rather than per block. It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every file in every directory, it increments a counter for that file's usage count. Remember that due to hard links, a file may appear in two or more directories. Symbolic links do not count and do not cause the counter for the target file to be incremented.

When it is all done, it has a list, indexed by i-node number, telling how many directories contain each file. It then compares these numbers with the link counts stored in the i-nodes themselves. These counts start at 1 when a file is created and are incremented each time a (hard) link is made to the file. In a consistent file system, both counts will agree. However, two kinds of errors can occur: the link count in the i-node can be too high or it can be too low.

If the link count is higher than the number of directory entries, then even if all the files are removed from the directories, the count will still be nonzero and the i-node will not be removed. This error is not serious, but it wastes space on the disk with files that are not in any directory. It should be fixed by setting the link count in the i-node to the correct value.

The other error is potentially catastrophic. If two directory entries are linked to a file, but the i-node says that there is only one, when either directory entry is removed, the i-node count will go to zero. When an i-node count goes to zero, the file system marks it as unused and releases all of its blocks. This action will result in one of the directories now pointing to an unused i-node, whose blocks may soon be assigned to other files. Again, the solution is just to force the link count in the i-node to the actual number of directory entries.

These two operations, checking blocks and checking directories, are often integrated for efficiency reasons (i.e., only one pass over the i-nodes is required). Other checks are also possible. For example, directories have a definite format, with i-node numbers and ASCII names. If an i-node number is larger than the number of i-nodes on the disk, the directory has been damaged.

Furthermore, each i-node has a mode, some of which are legal but strange, such as 0007, which allows the owner and his group no access at all, but allows outsiders to read, write, and execute the file. It might be useful to at least report files that give outsiders more rights than the owner. Directories with more than, say, 1000 entries are also suspicious. Files located in user directories, but which are owned by the superuser and have the SETUID bit on, are potential security problems because such files acquire the powers of the superuser when executed by any user. With a little effort, one can put together a fairly long list of technically legal but still peculiar situations that might be worth reporting.

The previous paragraphs have discussed the problem of protecting the user against crashes. Some file systems also worry about protecting the user against himself. If the user intends to type

```
rm *.o
```

to remove all the files ending with *.o* (compiler generated object files), but accidentally types

```
rm * .o
```

(note the space after the asterisk), *rm* will remove all the files in the current directory and then complain that it cannot find *.o*. In some systems, when a file is removed, all that happens is that a bit is set in the directory or i-node marking the file as removed. No disk blocks are returned to the free list until they are actually needed. Thus, if the user discovers the error immediately, it is possible to run a special utility program that "unremoves" (i.e., restores) the removed files. In Windows, files that are removed are placed in the recycle bin, from which they can later be retrieved if need be. Of course, no storage is reclaimed until they are actually deleted from this directory.

Mechanisms like this are insecure. A secure system would actually overwrite the data blocks with zeros or random bits when a disk is deleted, so another user could not retrieve it. Many users are unaware how long data can live. Confidential or sensitive data can often be recovered from disks that have been discarded (Garfinkel and Shelat, 2003).

5.3.6 File System Performance

Access to disk is much slower than access to memory. Reading a memory word might take 10 nsec. Reading from a hard disk might proceed at 10 MB/sec, which is forty times slower per 32-bit word, and to this must be added 5–10 msec

to seek to the track and then wait for the desired sector to arrive under the read head. If only a single word is needed, the memory access is on the order of a million times as fast as disk access. As a result of this difference in access time, many file systems have been designed with various optimizations to improve performance. In this section we will cover three of them.

Caching

The most common technique used to reduce disk accesses is the **block cache** or **buffer cache**. (Cache is pronounced "cash" and is derived from the French *cacher*, meaning to hide.) In this context, a cache is a collection of blocks that logically belong on the disk but are being kept in memory for performance reasons.

Various algorithms can be used to manage the cache, but a common one is to check all read requests to see if the needed block is in the cache. If it is, the read request can be satisfied without a disk access. If the block is not in the cache, it is first read into the cache, and then copied to wherever it is needed. Subsequent requests for the same block can be satisfied from the cache.

Operation of the cache is illustrated in Fig. 5-20. Since there are many (often thousands of) blocks in the cache, some way is needed to determine quickly if a given block is present. The usual way is to hash the device and disk address and look up the result in a hash table. All the blocks with the same hash value are chained together on a linked list so the collision chain can be followed.

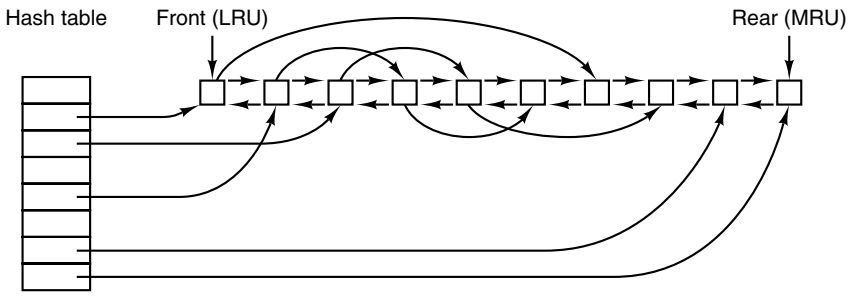

Figure 5-20. The buffer cache data structures.

When a block has to be loaded into a full cache, some block has to be removed (and rewritten to the disk if it has been modified since being brought in). This situation is very much like paging, and all the usual page replacement algorithms described in Chap. 4, such as FIFO, second chance, and LRU, are applicable. One pleasant difference between paging and caching is that cache references are relatively infrequent, so that it is feasible to keep all the blocks in exact LRU order with linked lists.

In Fig. 5-20, we see that in addition to the collision chains starting at the hash table, there is also a bidirectional list running through all the blocks in the order of

usage, with the least recently used block on the front of this list and the most recently used block at the end of this list. When a block is referenced, it can be removed from its position on the bidirectional list and put at the end. In this way, exact LRU order can be maintained.

Unfortunately, there is a catch. Now that we have a situation in which exact LRU is possible, it turns out that LRU is undesirable. The problem has to do with the crashes and file system consistency discussed in the previous section. If a critical block, such as an i-node block, is read into the cache and modified, but not rewritten to the disk, a crash will leave the file system in an inconsistent state. If the i-node block is put at the end of the LRU chain, it may be quite a while before it reaches the front and is rewritten to the disk.

Furthermore, some blocks, such as i-node blocks, are rarely referenced twice within a short interval. These considerations lead to a modified LRU scheme, taking two factors into account:

1. Is the block likely to be needed again soon?
2. Is the block essential to the consistency of the file system?

For both questions, blocks can be divided into categories such as i-node blocks, indirect blocks, directory blocks, full data blocks, and partially full data blocks. Blocks that will probably not be needed again soon go on the front, rather than the rear of the LRU list, so their buffers will be reused quickly. Blocks that might be needed again soon, such as a partly full block that is being written, go on the end of the list, so they will stay around for a long time.

The second question is independent of the first one. If the block is essential to the file system consistency (basically, everything except data blocks), and it has been modified, it should be written to disk immediately, regardless of which end of the LRU list it is put on. By writing critical blocks quickly, we greatly reduce the probability that a crash will wreck the file system. While a user may be unhappy if one of his files is ruined in a crash, he is likely to be far more unhappy if the whole file system is lost.

Even with this measure to keep the file system integrity intact, it is undesirable to keep data blocks in the cache too long before writing them out. Consider the plight of someone who is using a personal computer to write a book. Even if our writer periodically tells the editor to write the file being edited to the disk, there is a good chance that everything will still be in the cache and nothing on the disk. If the system crashes, the file system structure will not be corrupted, but a whole day's work will be lost.

This situation need not happen very often before we have a fairly unhappy user. Systems take two approaches to dealing with it. The UNIX way is to have a system call, sync, which forces all the modified blocks out onto the disk immediately. When the system is started up, a program, usually called *update*, is started up in the background to sit in an endless loop issuing sync calls, sleeping for 30

sec between calls. As a result, no more than 30 seconds of work is lost due to a system crash, a comforting thought for many people.

The Windows way is to write every modified block to disk as soon as it has been written. Caches in which all modified blocks are written back to the disk immediately are called **write-through caches**. They require more disk I/O than nonwrite-through caches. The difference between these two approaches can be seen when a program writes a 1-KB block full, one character at a time. UNIX will collect all the characters in the cache and write the block out once every 30 seconds, or whenever the block is removed from the cache. Windows will make a disk access for every character written. Of course, most programs do internal buffering, so they normally write not a character, but a line or a larger unit on each write system call.

A consequence of this difference in caching strategy is that just removing a (floppy) disk from a UNIX system without doing a sync will almost always result in lost data, and frequently in a corrupted file system as well. With Windows, no problem arises. These differing strategies were chosen because UNIX was developed in an environment in which all disks were hard disks and not removable, whereas Windows started out in the floppy disk world. As hard disks became the norm, the UNIX approach, with its better efficiency, became the norm, and is also used now on Windows for hard disks.

Block Read Ahead

A second technique for improving perceived file system performance is to try to get blocks into the cache before they are needed to increase the hit rate. In particular, many files are read sequentially. When the file system is asked to produce block k in a file, it does that, but when it is finished, it makes a sneaky check in the cache to see if block $k + 1$ is already there. If it is not, it schedules a read for block $k + 1$ in the hope that when it is needed, it will have already arrived in the cache. At the very least, it will be on the way.

Of course, this read ahead strategy only works for files that are being read sequentially. If a file is being randomly accessed, read ahead does not help. In fact, it hurts by tying up disk bandwidth reading in useless blocks and removing potentially useful blocks from the cache (and possibly tying up more disk bandwidth writing them back to disk if they are dirty). To see whether read ahead is worth doing, the file system can keep track of the access patterns to each open file. For example, a bit associated with each file can keep track of whether the file is in "sequential access mode" or "random access mode." Initially, the file is given the benefit of the doubt and put in sequential access mode. However, whenever a seek is done, the bit is cleared. If sequential reads start happening again, the bit is set once again. In this way, the file system can make a reasonable guess about whether it should read ahead or not. If it gets it wrong once it a while, it is not a disaster, just a little bit of wasted disk bandwidth.

Reducing Disk Arm Motion

Caching and read ahead are not the only ways to increase file system performance. Another important technique is to reduce the amount of disk arm motion by putting blocks that are likely to be accessed in sequence close to each other, preferably in the same cylinder. When an output file is written, the file system has to allocate the blocks one at a time, as they are needed. If the free blocks are recorded in a bitmap, and the whole bitmap is in main memory, it is easy enough to choose a free block as close as possible to the previous block. With a free list, part of which is on disk, it is much harder to allocate blocks close together.

However, even with a free list, some block clustering can be done. The trick is to keep track of disk storage not in blocks, but in groups of consecutive blocks. If sectors consist of 512 bytes, the system could use 1-KB blocks (2 sectors) but allocate disk storage in units of 2 blocks (4 sectors). This is not the same as having a 2-KB disk blocks, since the cache would still use 1-KB blocks and disk transfers would still be 1 KB but reading a file sequentially on an otherwise idle system would reduce the number of seeks by a factor of two, considerably improving performance. A variation on the same theme is to take account of rotational positioning. When allocating blocks, the system attempts to place consecutive blocks in a file in the same cylinder.

Another performance bottleneck in systems that use i-nodes or anything equivalent to i-nodes is that reading even a short file requires two disk accesses: one for the i-node and one for the block. The usual i-node placement is shown in Fig. 5-21(a). Here all the i-nodes are near the beginning of the disk, so the average distance between an i-node and its blocks will be about half the number of cylinders, requiring long seeks.

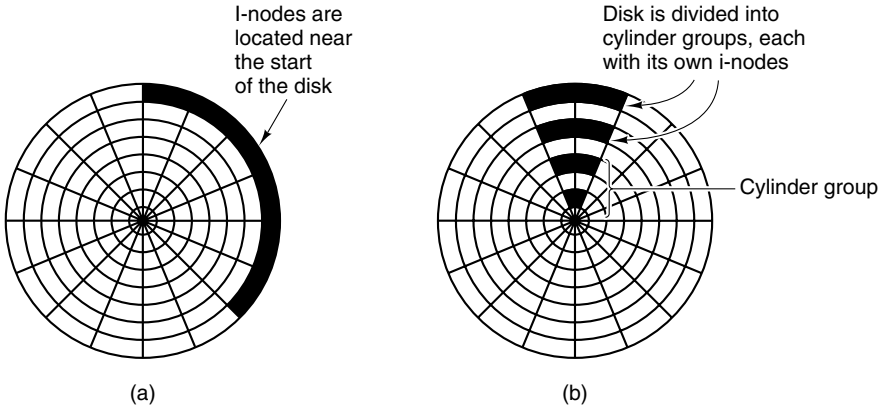

Figure 5-21. (a) I-nodes placed at the start of the disk. (b) Disk divided into cylinder groups, each with its own blocks and i-nodes.

One easy performance improvement is to put the i-nodes in the middle of the disk, rather than at the start, thus reducing the average seek between the i-node

and the first block by a factor of two. Another idea, shown in Fig. 5-21(b), is to divide the disk into cylinder groups, each with its own i-nodes, blocks, and free list (McKusick et al., 1984). When creating a new file, any i-node can be chosen, but an attempt is made to find a block in the same cylinder group as the i-node. If none is available, then a block in a nearby cylinder group is used.

5.3.7 Log-Structured File Systems

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, disks are becoming much bigger and cheaper (but not much faster), and memories are growing exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time. The combination of these factors means that a performance bottleneck is arising in many file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the **Log-structured File System**). In this section we will briefly describe how LFS works. For a more complete treatment, see Rosenblum and Ousterhout (1991).

The idea that drove the LFS design is that as CPUs get faster and RAM memories get larger, disk caches are also increasing rapidly. Consequently, it is now possible to satisfy a very substantial fraction of all read requests directly from the file system cache, with no disk access needed. It follows from this observation, that in the future, most disk accesses will be writes, so the read-ahead mechanism used in some file systems to fetch blocks before they are needed no longer gains much performance.

To make matters worse, in most file systems, writes are done in very small chunks. Small writes are highly inefficient, since a 50-μsec disk write is often preceded by a 10-msec seek and a 4-msec rotational delay. With these parameters, disk efficiency drops to a fraction of 1 percent.

To see where all the small writes come from, consider creating a new file on a UNIX system. To write this file, the i-node for the directory, the directory block, the i-node for the file, and the file itself must all be written. While these writes can be delayed, doing so exposes the file system to serious consistency problems if a crash occurs before the writes are done. For this reason, the i-node writes are generally done immediately.

From this reasoning, the LFS designers decided to re-implement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a log. Periodically, and also when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log. A single segment may thus contain i-nodes, directory blocks, data blocks, and other kinds of blocks all mixed together. At the start of each segment is a

segment summary, telling what can be found in the segment. If the average segment can be made to be about 1 MB, almost the full bandwidth of the disk can be utilized.

In this design, i-nodes still exist and have the same structure as in UNIX, but they are now scattered all over the log, instead of being at a fixed position on the disk. Nevertheless, when an i-node is located, locating the blocks is done in the usual way. Of course, finding an i-node is now much harder, since its address cannot simply be calculated from its i-node number, as in UNIX. To make it possible to find i-nodes, an i-node map, indexed by i-node number, is maintained. Entry *i* in this map points to i-node *i* on the disk. The map is kept on disk, but it is also cached, so the most heavily used parts will be in memory most of the time in order to improve performance.

To summarize what we have said so far, all writes are initially buffered in memory, and periodically all the buffered writes are written to the disk in a single segment, at the end of the log. Opening a file now consists of using the map to locate the i-node for the file. Once the i-node has been located, the addresses of the blocks can be found from it. All of the blocks will themselves be in segments, somewhere in the log.

If disks were infinitely large, the above description would be the entire story. However, real disks are finite, so eventually the log will occupy the entire disk, at which time no new segments can be written to the log. Fortunately, many existing segments may have blocks that are no longer needed, for example, if a file is overwritten, its i-node will now point to the new blocks, but the old ones will still be occupying space in previously written segments.

To deal with this problem, LFS has a **cleaner** thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

The bookkeeping here is nontrivial, since when a file block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nevertheless, it is possible to do the administration, and the performance results show that all this complexity is worthwhile. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good as or better than UNIX for reads and large writes.

5.4 SECURITY

File systems generally contain information that is highly valuable to their users. Protecting this information against unauthorized usage is therefore a major concern of all file systems. In the following sections we will look at a variety of issues concerned with security and protection. These issues apply equally well to timesharing systems as to networks of personal computers connected to shared servers via local area networks.

5.4.1 The Security Environment

People frequently use the terms "security" and "protection" interchangeably. Nevertheless, it is frequently useful to make a distinction between the general problems involved in making sure that files are not read or modified by unauthorized persons, which include technical, administrative, legal, and political issues on the one hand, and the specific operating system mechanisms used to provide security, on the other. To avoid confusion, we will use the term **security** to refer to the overall problem, and the term **protection mechanisms** to refer to the specific operating system mechanisms used to safeguard information in the computer. The boundary between them is not well defined, however. First we will look at security to see what the nature of the problem is. Later on in the chapter we will look at the protection mechanisms and models available to help achieve security.

Security has many facets. Three of the more important ones are the nature of the threats, the nature of intruders, and accidental data loss. We will now look at these in turn.

Threats

From a security perspective, computer systems have three general goals, with corresponding threats to them, as listed in Fig. 5-22. The first one, **data confidentiality**, is concerned with having secret data remain secret. More specifically, if the owner of some data has decided that these data are only to be made available to certain people and no others, the system should guarantee that release of the data to unauthorized people does not occur. As a bare minimum, the owner should be able to specify who can see what, and the system should enforce these specifications.

The second goal, **data integrity**, means that unauthorized users should not be able to modify any data without the owner's permission. Data modification in this context includes not only changing the data, but also removing data and adding false data as well. If a system cannot guarantee that data deposited in it remain unchanged until the owner decides to change them, it is not worth much as an information system. Integrity is usually more important than confidentiality.