Chapter 8

Files and Other Persistent Storage

8.1 Introduction

In this chapter, you will study two different kinds of service, each of which can be provided by either an operating system or middleware, and each of which can take several different forms. *Persistence* services provide for the retention of data for periods of time that are long enough to include system crashes, power failures, and similar disruptions. *Access* services provide application programs the means to operate on objects that are identified by name or by other attributes, such as a portion of the contents. In principle, these two kinds of service are independent of one another: persistent objects can be identified by numeric address rather than by name, and naming can be applied to non-persistent objects. However, persistence and access services are often provided in concert, as with named files stored on disk. Therefore, I am addressing both in a single chapter.

Any kind of object that stores data can be persistent, whether the object is as simple as a sequence of bytes or as complex as an application-specific object, such as the representation of a retirement portfolio in a benefits management system. In contemporary mainstream systems, the three most common forms of persistent storage are as follows:

- A *file*, which is an array of bytes that can be modified in length, as well as read and written at any numerically specified position. (Historically, the word has had other meanings, but this definition has become dominant.) File storage is normally provided by operating systems and will serve as my primary example in this chapter.

- A *table*, which in a relational database system is a multiset of *rows* (also known as *tuples* or *records*). Each row provides an appropriately typed value for each of the table's *columns*. For example, a table of chapters might have a title column, which holds character strings, and a number column, which holds integers. Then an individual row within that table might contain the title `"Files and Other Persistent Storage"` and the number `8`. Database storage is normally provided by middleware, rather than by an operating system.

- A *persistent object*, which is an application-specific object of the sort associated with object-oriented programming languages. For example, Java objects can be made persistent. Persistent objects are normally supported by middleware using one of the previous two types of persistent storage. Unfortunately, there are many competing approaches to supporting persistent objects; even the Java API does not yet have a single standardized approach. Therefore, I will not discuss persistent objects any further.

Access services can also take a variety of forms, from a single directory of unique names to the sort of sophisticated full-text search familiar from the web. I will concentrate on two access options that are popular in operating systems and middleware:

- Hierarchical *directories* map names into objects, each of which can be a subdirectory, thereby forming a tree of directories (or nested file folders). In some variants, objects can have be accessible through multiple names, either directly (multiple names refer to one object) or indirectly (one name refers to another name, which refers to an object). Operating systems generally use hierarchical directories to provide access to files.

- *Indexes* provide access to those objects that contain specified data. For example, an index on a table of orders could be used to find those rows that describe orders placed by a particular customer. Relational database middleware commonly uses indexes to provide access to rows. Files can also be indexed for fast searching.

The design of persistent storage mechanisms is influenced not only by the service being provided, but also by the underlying hardware technology. For many years the dominant technology has been moving-head magnetic disk drives. Although solid-state flash memory is playing a rapidly increasing role, disk drives are likely to remain important for years to come. Therefore,

Section 8.2 summarizes the key performance characteristics of disk drives; this summary serves as background for the design decisions explained in the remainder in the chapter.

Then, in Section 8.3, I will present an external view of a persistence service, looking at the file operations made available by POSIX operating systems. This material is of practical value (you are more likely to use a file system than to design one) and serves to motivate the examination of file system design in subsequent sections. Only once you understand what requirements a persistence service needs to meet will it make sense to consider the internal mechanisms it uses to do so.

Moving into the underlying mechanisms for persistence, Sections 8.4 and 8.5 examine the techniques used to allocate disk space and the metadata used to package the allocated space into usable objects. For simplicity, these two sections make reference only to file systems. However, the techniques used to provide space for a database table are fundamentally no different than for a file.

Next, I turn in Section 8.6 to the primary mechanisms for locating data: directories and indexes. Initially, I explain how these mechanisms are used in the traditional context of file directories and database indexes, and I point out that they are variations on the common theme of providing access through search keys. I then give a brief example of how these mechanisms can be merged to provide index-based file access. Before leaving the high-level view of access services, I explain one topic of particular interest to system administrators and application programmers: the ways in which multiple names can refer to the same file. Moving into the internals, I then present the data structures commonly used to store the directories or indexes for efficient access.

Persistent storage needs to retain its integrity in the face of system crashes. For example, no storage space should ever be both assigned to a file and marked as free for other use, even if the system crashed just as the space was being allocated. Similar properties are needed for directories and indexes; if a crash occurs while a file is being renamed, the file should have either its old name or its new name, but not both or neither. Because Chapter 5 covered the use of logs to provide durable atomic transactions, you have already seen the primary mechanism used to ensure integrity in contemporary persistent storage systems. Nonetheless, I devote Section 8.7 to the topic of metadata integrity so that I can sketch the alternative approaches to this problem.

Many operating systems allow file systems of varying designs to be mixed together. A Linux system might use one disk partition to store a Linux-

specific file system, while another partition holds a file system designed for Microsoft Windows or Mac OS X. This mixing of file systems provides a valuable case study of *polymorphism*, that is, the use of multiple implementations for a common interface. I devote Section 8.8 to this topic.

Finally, I give some attention to security issues in Section 8.9 before closing with the usual selection of exercises, projects, and bibliographic notes.

8.3 POSIX File API

All UNIX-like systems (including Linux and Mac OS X) support a rather complicated set of procedures for operating on files, which has evolved over the decades, eventually becoming part of the POSIX standard. For most everyday purposes, programmers can and should ignore this API, instead using one of the cleaner, higher-level APIs built on top of it, such as those included in the Java and C++ standards. Nonetheless, I will introduce the POSIX API here, because in many important systems, it forms the interface between the operating system kernel and software running in user-level application processes, even if the latter is encapsulated in libraries.

8.3.1 File Descriptors

Files are referred to in two different ways: by character-string *pathnames* (such as `microshell.c` or `/etc/passwd`) and by integer *file descriptors* (such as 0, 1, or 17). A pathname is a name of a file, optionally includ-ing a sequence of directories used to reach it. A file descriptor, on the other hand, provides no information about the file's name or location; it is just a featureless integer.

Many operations require file descriptors; in particular, to read data from a file or write data into a file requires a file descriptor. If a process happens to have inherited a file descriptor when it was forked from its parent (or happens to have received the file descriptor in a message from another process), then it can read or write the file without ever knowing a name for it. Otherwise, the process can use the `open` procedure to obtain a file descriptor for a named file. When the process is done with the file descriptor, it can `close` it. (When a process terminates, the operating system automatically closes any remaining open file descriptors.)

File descriptors can refer not only to open files, but also to other sources and destinations for input and output, such as the keyboard and display screen. Some procedures will work only for regular files, whereas others work equally well for hardware devices, network communication ports, and so forth. I will flag some places these distinctions matter; however, my primary focus will be on regular files in persistent storage.

By convention, all processes inherit at least three file descriptors from their parent. These file descriptors, known as the *standard input*, *standard output*, and *standard error output*, are numbered 0, 1, and 2, respectively. Rather than remembering the numbers, you should use the symbolic names defined in `unistd.h`, namely, `STDIN_FILENO`, `STDOUT_FILENO`, and `STDERR_FILENO`.

When you run a program from a shell and don't make special arrangements, standard input generally is your keyboard, while the standard output and error output are both directed to the shell's window on your display screen. You can redirect the standard input or output to a file by using the shell's `<` and `>` notations. For example, the shell command

```
ps l >my-processes
```

runs the `ps` program with the `l` option to generate a list of processes, as you saw in Chapter 7. However, rather than displaying the list on your screen, this command puts the list into a file called `my-processes`. The `ps` program doesn't need to know anything about this change; it writes its output to the standard output in either case. Only the shell needs to do something different, namely, closing the preexisting standard output and opening the file in its place before executing the `ps` program. If the `ps` program has any error messages to report, it outputs them to the standard error output, which remains connected to your display screen. That way, the error messages aren't hidden in the `my-processes` file.

Figure 8.2 contains a program illustrating how the shell would operate in the preceding example, with a child process closing its inherited standard output and then opening `my-processes` before executing `ps`. The most complicated procedure call is the one to `open`. The first argument is the name of the file to open. Because this character string does not contain any slash characters (`/`), the file is found in the process's current directory. (Every process has a current *working directory*, which can be changed using the `chdir` procedure.) If the name contained one or more slashes, such as `alpha/beta/gamma` or `/etc/passwd`, then the operating system would traverse one or more directories to find the file to open. In particular, `alpha/beta/gamma` would start with the current directory, look for subdirectory `alpha`, look in `alpha` for `beta`, and finally look in `beta` for the file `gamma`. Because `/etc/passwd` starts with a slash, the search for this file would begin by looking in the root directory for `etc` and then in that directory for `passwd`. In Section 8.6, I will discuss file naming further, including related aspects of the POSIX API, such as how a file can be given an additional name or have a name removed.

```
#include <unistd.h>
#include <stdio.h>
#include <iostream>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/stat.h>
using namespace std;

int main(){
  pid_t returnedValue = fork();
  if(returnedValue < 0){
    perror("error forking");
    return -1;
  } else if (returnedValue == 0){
    if(close(STDOUT_FILENO) < 0){
      perror("error closing standard output");
      return -1;
    }
    // When there is no error, open returns the smallest file
    // descriptor not already in use by this process, so having
    // closed STDOUT_FILENO, the open should reuse that number.
    if(open("my-processes", O_WRONLY | O_CREAT | O_TRUNC,
            S_IRUSR | S_IWUSR) < 0){
      perror("error opening my-processes");
      return -1;
    }
    execlp("ps", "ps", "l", NULL);  // ps with option letter l
    perror("error executing ps");
    return -1;
  } else {
    if(waitpid(returnedValue, 0, 0) < 0){
      perror("error waiting for child");
      return -1;
    }
    cout << "Note the parent still has the old standard output."
         << endl;
  }
}
```

Figure 8.2: This C++ program, `file-processes.cpp`, illustrates how the shell runs the command `ps l >my-processes`. After forking, the child process closes the inherited standard output and in its place opens `my-processes` before executing `ps`.

The second argument to `open` specifies the particular way in which the file should be opened. Here, the `O_WRONLY` indicates the file should be opened for writing only (as opposed to `O_RDONLY` or `O_RDWR`), the `O_CREAT` indicates that the file should be created if it doesn't already exist (rather than signaling an error), and the `O_TRUNC` indicates that the file should be truncated to zero length before writing; that is, all the old data (if any) should be thrown out. Because the `O_CREAT` option is specified, the third argument to `open` is needed; it specifies the access permissions that should be given to the file, if it is created. In this case, the access permissions are read and write for the owning user only, that is, `rw-------`.

Even setting aside `open` and `close`, not all operations on files involve reading or writing the contents of the file. Some operate on the *metadata attributes*—attributes describing a file—such as the access permissions, time of last modification, or owner. A variety of procedures, such as `chmod`, `utime`, and `chown`, allow these attributes to be set; I won't detail them. I will, however, illustrate one procedure that allows the attributes of a file to be retrieved. The C++ program in Figure 8.3 uses the `fstat` procedure to retrieve information about its standard input. It then reports just a few of the attributes from the larger package of information. After printing the owner and modification time stamp, the program checks whether the standard input is from a regular file, as it would be if the shell was told to redirect standard input, using `<`. Only in this case does the program print out the file's size, because the concept of size doesn't make any sense for the stream of input coming from the keyboard, for example. If this program is compiled in a file called `fstater`, then the shell command

```
./fstater </etc/passwd
```

would give you information about the `/etc/passwd` file, which you could verify using the command `ls -ln /etc/passwd`.

Moving on to actually reading or writing the contents of a file, the low-level POSIX API provides three different choices, outlined here:

	Explicit Positions	**Sequential**
Memory Mapped	`mmap`	—
External	`pread/pwrite`	`read/write`

A file (or a portion thereof) can be mapped into the process's address space using the `mmap` procedure, allowing normal memory loads and stores to do the reading and writing. Alternatively, the file can be left outside the address space, and individual portions explicitly read or written using procedures

```
#include <unistd.h>
#include <time.h>
#include <sys/stat.h>
#include <stdio.h>
#include <iostream>
using namespace std;

int main(){
  struct stat info;
  if(fstat(STDIN_FILENO, &info) < 0){
    perror("Error getting info about standard input");
    return -1;
  }
  cout << "Standard input is owned by user number "
       << info.st_uid << endl;
  cout << "and was last modified " << ctime(&info.st_mtime);
  if(S_ISREG(info.st_mode)){
    cout << "It is a " << info.st_size << "-byte file." << endl;
  } else {
    cout << "It is not a regular file." << endl;
  }
  return 0;
}
```

Figure 8.3: This C++ program, `fstater.cpp`, describes its standard input, using information retrieved using `fstat`. That information includes the owner, last modification time, and whether the standard input is from a regular file. In the latter case, the size of the file is also available.

that copy from the file into memory or from memory into the file. One version of these procedures (`pread` and `pwrite`) needs to be told what position within the file to read or write, whereas the other version (`read` and `write`) operates sequentially, with each operation implicitly using the portion of the file immediately after the preceding operation. I'll discuss all three possibilities at least briefly, because each has its virtues. Because `mmap` is the simplest procedure, I will start with it.

8.3.2 Mapping Files Into Virtual Memory

The use of `mmap` is illustrated by the C++ program in Figures 8.4 and 8.5, which copies the contents of one file to another. The program expects to be given the names of the input and output files as `argv[1]` and `argv[2]`, respectively. It uses the `open` procedure to translate these into integer file descriptors, `fd_in` and `fd_out`. By using `fstat` (as in Figure 8.3), it finds the size of the input file. This size (`info.st_size`) plays three roles. One is that the program makes the output file the same size, using `ftruncate`. (Despite its name, `ftruncate` does not necessarily make a file shorter; it sets the file's size, whether by truncating it or by padding it out with extra bytes that all have the value zero.) Another use of the input file's size is for the two calls to `mmap`, which map the input and output files into virtual memory, with read-only and write-only protections, respectively. The returned values, `addr_in` and `addr_out`, are the virtual addresses at which the two files start in the process's address space. The third use of the input file size is to tell the library procedure `memcpy` how many bytes to copy from `addr_in` to `addr_out`. The `memcpy` procedure is a loop that executes load and store instructions to copy from one place in virtual memory to another. (This loop could be written explicitly in C++, but would be less clear and likely less efficient as well, because the library routine is very carefully tuned for speed.)

Of course, I haven't explained all the arguments to `mmap`, or many other details. My intent here is not to provide comprehensive documentation for these API procedures, nor to provide a complete tutorial. Instead, the example should suffice to give you some feel for file I/O using `mmap`; files are opened, then mapped into the virtual address space, and then accessed as any other memory would be, for example, using `memcpy`.

The underlying idea behind virtual memory-based file access (using `mmap`) is that files are arrays of bytes, just like regions of virtual address space; thus, file access can be treated as virtual memory access. The next style of file I/O to consider accepts half of this argument (that files are arrays of bytes)

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <stdio.h>
#include <string.h>
#include <iostream>
using namespace std;

int main(int argc, char *argv[]){
  if(argc != 3){
    cerr << "Usage: " << argv[0] << " infile outfile" << endl;
    return -1;
  }
  int fd_in = open(argv[1], O_RDONLY);
  if(fd_in < 0){
    perror(argv[1]);
    return -1;
  }
  struct stat info;
  if(fstat(fd_in, &info) < 0){
    perror("Error stating input file");
    return -1;
  }
  void *addr_in =
    mmap(0, info.st_size, PROT_READ, MAP_SHARED, fd_in, 0);
  if(addr_in == MAP_FAILED){
    perror("Error mapping input file");
    return -1;
  }
```

Figure 8.4: This is the first portion of `cpmm.cpp`, a C++ program using virtual memory mapping to copy a file. The program is continued in the next figure.

```
  int fd_out =
    open(argv[2], O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
  if(fd_out < 0){
    perror(argv[2]);
    return -1;
  }
  if(ftruncate(fd_out, info.st_size) < 0){
    perror("Error setting output file size");
    return -1;
  }
  void *addr_out =
    mmap(0, info.st_size, PROT_WRITE, MAP_SHARED, fd_out, 0);
  if(addr_out == MAP_FAILED){
    perror("Error mapping output file");
    return -1;
  }
  memcpy(addr_out, addr_in, info.st_size);
  return 0;
}
```

Figure 8.5: This is the second portion of `cpmm.cpp`, a C++ program using virtual memory mapping to copy a file. The program is continued from the previous figure.

but rejects the other half (that they should therefore be treated the same as memory). In Section 8.3.4, you will see a third style of I/O, which largely rejects even the first premise.

8.3.3 Reading and Writing Files at Specified Positions

Although convenient, accessing files as virtual memory is not without disadvantages. In particular, writing files using `mmap` raises three problems:

- The process has no easy way to control the time at which its updates are made persistent. Specifically, there is no simple way for the process to ensure that a data structure is written to persistent storage only after it is in a consistent state, rather than in the middle of a series of related updates.
- A process can write a file only if it has read permission as well as write permission, because all page faults implicitly read from the file, even if the page faults occur in the course of writing data into the file's portion of virtual memory.
- Mapping a file into a range of addresses presumes you know how big the file is. That isn't well suited to situations in which you don't know in advance how much data will be written.

For these and other reasons, some programmers prefer to leave files separate from the virtual memory address space and use procedures in the POSIX API that explicitly copy data from a file into memory or from memory into a file. The `pread` and `pwrite` procedures take as arguments a file descriptor, a virtual address in memory, a number of bytes to copy, and a position within the file. Each procedure copies bytes starting from the specified position in the file and the specified address in memory—`pread` from the file to the memory and `pwrite` from the memory to the file. These procedures are somewhat tricky to use correctly, because they may copy fewer bytes than requested, and because they may signal error conditions that go away upon retrying the operation. Therefore, they always need to be put in carefully designed loops. For this reason, I will not devote space to an example here.

8.3.4 Sequential Reading and Writing

Both `mmap` and the `pread`/`pwrite` pair rely on the ability to access arbitrary positions within a file; that is, they treat the file as an array of bytes. As such, neither interface will work for other sources of input and destinations

for output, such as keyboards and network connections. Instead, one needs to use a sequential style of I/O, where each read or write operation takes place not at a specified position, but wherever the last one left off.

Sequential I/O is also quite convenient for many purposes, even when used with files. For example, suppose you give the following command in a shell:

```
(ls; ps) > information
```

This opens the file named `information` for writing as the standard output and then runs two programs in succession: `ls` to list the files in the current directory and `ps` to list processes. The net result is that `information` contains both listings, one after the other. The `ps` command does not need to take any special steps to direct its output to the position in the file immediately after where `ls` stopped. Instead, by using the sequential I/O features of the POSIX API, each of the two processes naturally winds up writing each byte of output to the position after the previously written byte, whether that previous byte was written by the same process or not.

A process can perform sequential I/O using the `read` and `write` procedures, which are identical to `pread` and `pwrite`, except that they do not take an argument specifying the position within the file. Instead, each implicitly is directed to read or write at the current *file offset* and to update that file offset. The file offset is a position for reading and writing that is maintained by the operating system.

For special files such as keyboard input, sequential input is intrinsic, without needing an explicit file offset. For regular files in persistent storage, however, the file offset is a numeric position within the file (of the same kind `pread` and `pwrite` take as arguments) that the operating system keeps track of behind the scenes. Whenever a file is opened, the operating system creates an *open file description*, a capability-like structure that includes the file offset, normally initialized to 0. Any file descriptors descended from that same call to `open` share the same open file description. For example, in the previous example of `ls` and `ps` writing to the `information` file, each of the two processes has its own file descriptor, but they are referring to the same open file description, and hence share the same file offset. If a process independently calls `open` on the same file, however, it will get a separate file offset.

A process implicitly increases the file offset whenever it does a `read` or `write` of length more than zero. It can also explicitly change the file offset using the `lseek` procedure. The `lseek` procedure can set the file offset anywhere within the file (for a regular file). As such, a process can use

the combination of `lseek` and `read` or `write` to simulate `pread` or `pwrite`. However, this simulation is prone to races if multiple threads or processes share the same open file description, unless they use some synchronization mechanism, such as a mutex.

Normally `lseek` is used only infrequently, with sequential access predominating. For example, a process may read a whole file sequentially, using `read`, and then use `lseek` to set it back to the beginning to read a second time. The conceptual model is based on a tape drive, where ordinary reads and writes progress sequentially through the tape, but rewinding or skipping forward are also possible.

The `read` and `write` procedures share the same difficulty as `pread` and `pwrite`: the necessity of looping until all bytes have been transferred. It is much easier to use the I/O facilities defined in the standard libraries for higher level programming languages, such as Java or C++. Behind the scenes, these libraries are using `read` and `write` and doing the looping (and other details) for you.

8.4 Disk Space Allocation

A file system is analogous to a virtual memory system, in that each uses a level of indirection to map objects into storage locations. In virtual memory, the mapping is from virtual addresses within address spaces to physical addresses within memory. In a file system, the mapping is from positions within files to locations in persistent storage. For efficiency, the mapping is done at a coarse granularity, several kilobytes at a time. In virtual memory, each page is mapped into a page frame; in a file system, each block of a file is mapped into a storage block. (You will see that blocks are typically several kilobytes in size, spanning multiple sectors.)

When discussing virtual memory, I remarked that the operating system was free to assign any unused page frame of physical memory to hold each page of virtual memory. However, although any allocation policy would be correct, some might cause cache memory to perform better.

Persistent storage faces a similar allocation problem, but the performance issues are considerably more pronounced if the persistent storage hardware is a disk drive, as I will assume in this section. A file system has the freedom to store data in any otherwise unused disk block. The choices it makes determine how accesses to files translate into accesses to disk. You have already seen that the pattern of disk access can make a huge performance difference (three orders of magnitude). Thus, I will examine

allocation policies here more closely than I examined placement policies in Chapter 6.

Before I get into allocation policies themselves and their embodiment in allocation mechanisms, I will look at the key objectives for allocation: minimizing wasted space and time. As you will see in Sections 8.4.1 and 8.4.2, these goals can be expressed as minimizing fragmentation and maximizing locality.

8.4.1 Fragmentation

The word *fragmentation* is used in two different senses. First, consider the definition I will *not* be using. For some authors, fragmentation refers to the degree to which a file is stored in multiple noncontiguous regions of the disk. A file that is stored in a single contiguous sequence of disk blocks (called an *extent*) is not fragmented at all, by this definition. A file stored in two separate extents would be slightly fragmented. If the file's blocks are individual scattered across the disk, then the file is maximally fragmented, by this definition. A *defragmentation* program moves files' blocks around on disk so as to leave each file in a single extent. To allow future allocations to be non-fragmented, the defragmentation program also arranges the files so that the free space on the disk is clustered together.

The contiguity and sequentiality issues mentioned in the preceding paragraph are important for speed of access; I will discuss them in Section 8.4.2 under the broader heading of locality. However, I will not refer to them as fragmentation, because I will use another definition that is well established in the operating systems field. By this alternative definition, fragmentation concerns space efficiency. A highly fragmented disk is one in which a large proportion of the storage capacity is unavailable for allocation to files. I will explain in the remainder of this subsection the phenomena that cause space to be unusable.

One source of waste is that space is allocated only in integer multiples of some file system block size. For example, a file system might allocate space only in units of 4 KB. A file that is too big to fit in a single 4-KB unit will be allocated 8 KB of space—even if it is only a single byte larger than 4 KB. The unused space in the last file block is called *internal fragmentation*. The amount of internal fragmentation depends not only on the desired file sizes, but also on the file system block size. As an analogy, consider parallel parking in an area where individual parking spaces are marked with painted lines, and where drivers actually respect those lines. The amount of wasted space depends on the cars being parked, but it also depends on how far

apart the lines are painted. Larger parking spaces will generally result in more wasted space.

The file system block size is always some multiple of the underlying disk drive's sector size; no file system ever subdivides the space within a single disk sector. Generally the file system blocks span several consecutive disk sectors; for example, eight disk sectors of 512 bytes each might be grouped into each 4-KB file system block. Larger file system blocks cause more internal fragmentation, but are advantageous from other perspectives. In particular, you will see that a larger block size tends to reduce external fragmentation. Additionally, a larger block size implies that there are fewer blocks to keep track of, which reduces bookkeeping overhead.

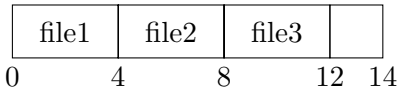
Once a space allocation request has been rounded up to the next multiple of the block size, the operating system must locate the appropriate number of unused blocks. In order to read or write the file as quickly as possible, the blocks should be in a single consecutive extent. For the moment, I will consider this to be an absolute requirement. Later, I will consider relaxing it.

Continuing with my earlier example, suppose you need space for a file that is just one byte larger than 4 KB and hence has been rounded up to two 4-KB blocks. The new requirement of contiguity means that you are looking for somewhere on the disk where two consecutive 4-KB blocks are free. Perhaps you are out of luck. Maybe the disk is only half full, but the half that is full consists of every even-numbered file system block with all the odd-numbered ones available for use. This situation, where there is lots of space available but not enough grouped together in any one place, is *external fragmentation*. So long as you insist on contiguous allocation, external fragmentation is another cause of wasted space: blocks that are free for use, but are too scattered to be usable.

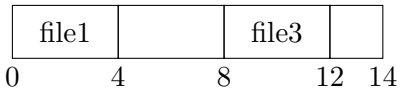
On the surface, it appears that external fragmentation would result only from very strange circumstances. My example, in which every second file system block is occupied, would certainly fit that description. To start with, it implies that you allocated lots of small files and now suddenly want to allocate a larger file. Second, it implies that you either were really dumb in choosing where those small files went (skipping every other block), or had phenomenally bad luck in the user's choice of which files to delete.

However, external fragmentation can occur from much more plausible circumstances. In particular, you can wind up with only small gaps of space available even if all the allocations have been for much larger amounts of space and even if the previous allocations were done without leaving silly gaps for no reason.

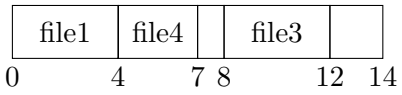
For a small scenario that illustrates the phenomenon, consider a disk that has room for only 14 file system blocks. Suppose you start by allocating three four-block files. At this point, the space allocation might look as follows:



Suppose file2 is now deleted, resulting in a four-block gap, with another two blocks free at the end of the disk:



If, at this point, a three-block file (file4) is created, it can go into the four-block gap, leaving one block unused:



Now there are three unused blocks, but there is no way to satisfy another three-block allocation request, because the three unused blocks are broken up, with one block between files 4 and 3, and two more blocks at the end of the disk.

Notice that you wound up with a one-block gap not because a one-block file was created and later deleted (or because of stupid allocation), but because a four-block file was replaced by a three-block file. The resulting gap is the difference in the file sizes. This means that even if a disk is used exclusively for storing large files, it may still wind up with small gaps, which cannot hold any large files. This is the fundamental problem of external fragmentation.

Returning to the parallel parking analogy, consider an area where no parking spaces are marked on the pavement, leaving drivers to allocate their own spaces. Even if they are courteous enough not to leave any pointless gaps, small gaps will arise as cars of varying sizes come and go. A large car may vacate a space, which is then taken by a smaller car. The result is a gap equal to the difference in car sizes, too small for even the smallest cars to use. If this situation happens repeatedly at different spots along a block, there may be enough total wasted space to accommodate a car, but not all in one place.

Earlier, I mentioned that increasing the file system block size, which increases internal fragmentation, decreases external fragmentation. The reason for this is that with a larger block size, there is less variability in the amount of space being allocated. Files that might have different sizes when rounded up to the next kilobyte (say, 14 KB and 15 KB) may have the same size when rounded to the next multiple of 4 KB (in this case, 16 KB and 16 KB). Reduced variability reduces external fragmentation; in the extreme case, no external fragmentation at all occurs if the files are all allocated the same amount of space.

Suppose you relax the requirement that a file be allocated a single extent of the disk. Using file metadata, it is possible to store different blocks of the file in different locations, much as a virtual memory address space can be scattered throughout physical memory. Does this mean that external fragmentation is a nonissue? No, because for performance reasons, you will still want to allocate the file contiguously as much as possible. Therefore, external fragmentation will simply change from being a space-efficiency issue (free space that cannot be used) to a time-efficiency issue (free space that cannot be used without file access becoming slower). This gets us into the next topic, locality.

8.4.2 Locality

Recall that disks provide their fastest performance when asked to access a large number of consecutive sectors in a single request at a location nearby to the previous access request. Most file system designers have interpreted these conditions for fast access as implying the following locality guidelines for space allocation:

1. The space allocated for each file should be broken into as few extents as possible.

2. If a file needs to be allocated more than one extent, each extent should be nearby to the previous one.

3. Files that are commonly used in close succession (or concurrently) should be placed near one another.

The connection between fast access and these three guidelines is based on an implicit assumption that the computer system's workload largely consists of accessing one file at a time and reading or writing each file in its entirety, from beginning to end. In some cases, this is a reasonable approximation to the truth, and so the preceding locality guidelines do result in

good performance. However, it is important to remember that the guidelines incorporate an assumption about the workload as well as the disk performance characteristics. For some workloads, a different allocation strategy may be appropriate. In particular, as computing workloads are consolidated onto a smaller number of computers (using techniques such as virtualization, as discussed in Section 7.5.2), file accesses become more jumbled.

As an example of a different allocation strategy that might make sense, Rosenblum and Ousterhout suggested that blocks should be allocated space on disk in the order they are written, without regard to what files they belong to or what positions they occupy within those files. By issuing a large number of consecutive writes to the disk in a single operation, this allows top performance for writing. Even if the application software is concurrently writing to multiple files, and doing so at random positions within those files, the write operations issued to disk will be optimal, unlike with the more conventional file layout. Of course, read accesses will be efficient only if they are performed in the same order as the writes were. Fortunately, some workloads do perform reads in the same order as writes, and some other workloads do not need efficient read access. In particular, the efficiency of read access is not critical in a workload that reads most disk blocks either never or repeatedly. Those blocks that are never read are not a problem, and those that are read repeatedly need only suffer the cost of disk access time once and can thereafter be kept in RAM.

Returning to the more mainstream strategy listed at the beginning of this subsection, the primary open question is how to identify files that are likely to be accessed contemporaneously, so as to place them nearby to one another on disk. One approach, used in UNIX file systems, is to assume that files are commonly accessed in conjunction with their parent directory or with other (sibling) files in the same directory. Another approach is to not base the file placement on assumptions, but rather on observed behavior. (One assumption remains: that future behavior will be like past behavior.) For example, Microsoft introduced a feature into Windows with the XP version, in which the system observes the order of file accesses at system boot time and also at application startup time, and then reorganizes the disk space allocation based on those observed access orders. Mac OS X does something similar as of version 10.3: it measures which files are heavily used and groups them together.

8.4.3 Allocation Policies and Mechanisms

Having seen the considerations influencing disk space allocation (fragmentation and locality), you are now in a better position to appreciate the specific allocation mechanism used by any particular file system and the policy choices embodied in that mechanism. The full range of alternatives found in different file systems is too broad to consider in any detail here, but I will sketch some representative options.

Each file system has some way of keeping track of which disk blocks are in use and which are free to be allocated. The most common representation for this information is a *bitmap*, that is, an array of bits, one per disk block, with bit i indicating whether block i is in use. With a bitmap, it is easy to look for space in one particular region of the disk, but slow to search an entire large disk for a desired size extent of free space.

Many UNIX and Linux file systems use a slight variant on the bitmap approach. Linux's ext3fs file system can serve as an example. The overall disk space is divided into modest-sized chunks known as *block groups*. On a system with 4-KB disk blocks, a block group might encompass 128 MB. Each block group has its own bitmap, indicating which blocks within that group are free. (In Exercise 8.8, you can show that in the example given, each block group's bitmap fits within a single block.) Summary information for the file system as a whole indicates how much free space each block group has, but not the specific location of the free space within the block groups. Thus, allocation can be done in two steps: first find a suitable block group using the summary information, and then find a suitable collection of blocks within the block group, using its bitmap.

I remarked earlier that UNIX and Linux file systems generally try to allocate each file near its parent directory. In particular, regular files are placed in the same block group as the parent directory, provided that there is any space in that group. If this rule were also followed for subdirectories, the result would be an attempt to cram the entire file system into one block group. Therefore, these file systems use an alternative rule to choose a block group for a subdirectory.

When creating a subdirectory, early versions of ext3fs and similar file systems selected a block group containing a lot of free space. This spread the directories, with their corresponding files, relatively evenly through the whole disk. Because each new directory went into a block group with lots of free space, there was a good chance that the files contained in that directory would fit in the same block group with it. However, traversing a directory tree could take a long time with these allocation policies, because

each directory might be nowhere near its parent directory.

Therefore, more recent versions of ext3fs and similar file systems have used a different allocation policy for directories, developed by Orlov. A subdirectory is allocated in the parent directory's block group, provided that it doesn't get too crowded. Failing that, the allocation policy looks through the subsequent block groups for one that isn't too crowded. This preserves locality across entire directory trees without stuffing any block group so full of directories that the corresponding files won't fit. The result can be significant performance improvements for workloads that traverse directory trees.

Once a file system decides to locate a file within a particular block group, it still needs to allocate one or more extents of disk blocks to hold the file's data. (Hopefully those extents will all lie within the chosen block group, although there needs to be a way for large files to escape from the confines of a single block group.)

The biggest challenge in allocating extents is knowing how big an extent to allocate. Some older file systems required application programmers to specify each file's size at the time the file was created, so that the system could allocate an extent of corresponding size. However, modern systems don't work this way; instead, each file grows automatically to accommodate the data written into it.

To meet this challenge, modern operating systems use a technique known as *delayed allocation*. As background, you need to understand that operating systems do not normally write data to disk the moment an application program issues a write request. Instead, the data is stored in RAM and written back to disk later. This delay in writing yields two options for when the disk space is allocated: when the data goes into RAM or later when it gets written to disk.

Without delayed allocation, the operating system needs to choose a disk block to hold the data at the time it goes into RAM. The system tags the data in RAM with the disk block in which that data belongs. Later, the system writes the data out to the specified location on disk. This approach is simple, but requires the operating system to allocate space for the first block of data as soon as it is generated, before there is any clue how many more blocks will follow.

Delayed allocation puts off the choice of disk block until the time of actually writing to disk; the data stored in RAM is tagged only with the file it should be written to and the position within that file. Now the operating system does not need to guess how much data a program is going to write at the time when it generates the first block. Instead, it can wait and see

how much data gets written and allocate an extent that size.

Once the operating system knows the desired extent size, it needs to search the data structure that records the available space. Bitmaps (whether in individual block groups or otherwise) are not the only option for tracking free space. The XFS file system, which was particularly designed for large file systems, takes an alternative approach. It uses balanced search trees, known as B-trees, to track the free extents of disk space. One B-tree stores the free extents indexed by their location while another indexes them by their size. That way, XFS can quickly locate free space near a specified location on disk or can quickly locate a desired amount of space. Technically, the trees used by XFS are a slight variant of B-trees, known as B^+-trees. I'll describe this data structure in Section 8.5.1.

With free extents indexed by size in a B^+-tree, the XFS allocator can naturally use a *best-fit* policy, where it finds the smallest free extent bigger than the desired size. (If the fit is not exact, the extra space can be broken off and left as a smaller free extent.) With a bitmap, on the other hand, the most natural allocation policy is *first-fit*, the policy of finding the first free extent that is large enough. Each policy has its merits; you can compare them in Exercise 8.9.

8.5 Metadata

You have seen that a file system is analogous to a virtual memory system. Each has an allocation policy to select concrete storage locations for each chunk of data. Continuing the analogy, I will now explain the *metadata* that serves as the analog of page tables. Recall that in a system with separate address spaces, each process has its own page table, storing the information regarding which page frame holds that process's page 0, page 1, and so forth. Similarly, each file has its own metadata storing the information regarding which disk block holds that file's block 0, block 1, and so forth. You will see that, as with page tables, there are several choices for the data structure holding this mapping information. I discuss these alternative structures in Section 8.5.1.

Metadata is data about data. Information regarding where on disk the data is stored is one very important kind of metadata. However, I will also more briefly enumerate other kinds. First, in Section 8.5.2, I will revisit access control, a topic I considered from another perspective in Chapter 7. In Section 8.5.2, the question is not how access control information is enforced during access attempts, but how it is stored in the file system. Second, I

will look in Section 8.5.3 at the other more minor, miscellaneous kinds of metadata (beyond data location and access control), such as access dates and times.

Some authors include file names as a kind of metadata. This makes sense in those file systems where each file has exactly one name. However, most modern file systems do not fit this description; a file might have no names, or might have multiple names. Thus, you are better off thinking of a name not as a property of a file, but as a route that can lead to a file. Similarly, in other persistence services, data may be accessed through multiple routes, such as database indexes. Therefore, I will not include naming in this section on metadata, instead including it in Section 8.6 on directories and indexing.

8.5.1 Data Location Metadata

The simplest representation for data location metadata would be an array of disk block numbers, with element i of the array specifying which disk block holds block i of the file. This would be analogous to a linear page table. Traditional UNIX file systems (including Linux's ext2fs and ext3fs) use this approach for small files. Each file's array of disk block numbers is stored in the file's metadata structure known as its *inode* (short for *index node*). For larger files, these file systems keep the inodes compact by using indirect blocks, roughly analogous to multilevel page tables. I discuss the traditional form of inodes and indirect blocks next. Thereafter, I discuss two alternatives used in some more modern file systems: extent maps, which avoid storing information about individual blocks, and B^+-trees, which provide efficient access to large extent maps.

Inodes and Indirect Blocks

When UNIX was first developed in the early 1970s, one of its many innovative features was the file system design, a design that has served as the model for commonly used UNIX and Linux file systems to the present day, including Linux's ext3fs. The data-location metadata in these systems is stored in a data structure that can better be called expedient than elegant. However, the structure is efficient for small files, allows files to grow large, and can be manipulated by simple code.

Each file is represented by a compact chunk of data called an inode. The inode contains the file's metadata if the file is small or an initial portion of the metadata if the file is large. By allowing large files to have more metadata elsewhere (in indirect blocks), the inodes are kept to a small fixed

size. Each file system contains an array of inodes, stored in disk blocks set aside for the purpose, with multiple inodes per block. Each inode is identified by its position in the array. These inode numbers (or *inumbers*) are the fundamental identifiers of the files in a file system; essentially, the files are identified as file 0, file 1, and so forth, which indicate the files with inodes in position 0, 1, and so forth. Later, in Section 8.6, you'll see how file names are mapped into inode numbers.

Each inode provides the metadata for one file. The metadata includes the disk block numbers holding that file's data, as well as the access permissions and other metadata. These categories of metadata are shown in Figure 8.6. In this simplified diagram, the inode directly contains the mapping information specifying which disk block contains each block of the file, much like a linear page table. Recall, however, that inodes are a small, fixed size, whereas files can grow to be many blocks long. To resolve this conflict, each inode directly contains the mapping information only for the first dozen or so blocks. (The exact number varies between file systems, but is consistent within any one file system.) Thus, a more realistic inode picture is as shown in Figure 8.7.

Before I go into detail on how further disk blocks are indirectly accessed, I should emphasize one aspect of the inode design. The low-numbered blocks of a file are mapped in the exact same way (directly in the inode) regardless of whether they are the only blocks in a small file or the first blocks of a large file. This means that large files have a peculiar asymmetry, with some blocks more efficiently accessible than others. The advantage is that when a file grows and transitions from being a small file to being a large one, the early blocks' mapping information remains unchanged.

Because most files are small, the inodes are kept small, a fraction of a

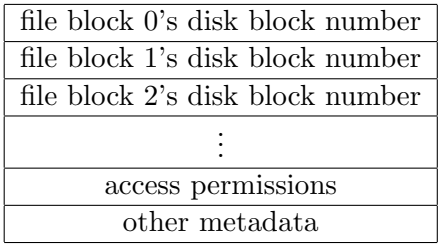


Figure 8.6: This initial approximation of an inode shows the principle categories of metadata. However, this diagram is unrealistic in that the list of disk block numbers seems to be unlimited, whereas actual inodes have only a limited amount of space.

file block 0's disk block number
⋮
file block 11's disk block number
indirect access to file block 12 through the end of the file
access permissions
other metadata

Figure 8.7: In this limited-size inode, blocks from number 12 to the end of the file are indirectly referenced.

block in size. (If inodes were full blocks, the overhead for single-block files would be 100 percent.) For those files large enough to overflow an inode, however, one can be less stingy in allocating space for metadata. Therefore, if the system needs more metadata space, it doesn't allocate a second inode; it allocates a whole additional disk block, an *indirect block*. This provides room for many more block numbers, as shown in Figure 8.8. The exact number of additional block numbers depends on how big blocks and block numbers are. With 4-KB blocks and 4-byte block numbers, an indirect block could hold 1 K block numbers (that is, 1024 block numbers), as shown in the figure. This kind of indirect block is more specifically called a *single indirect block*, because it adds only a single layer of indirection: the inode points to it, and it points to data blocks.

In this example with 4-KB blocks, the single indirect block allows you to accommodate files slightly more than 4 MB in size. To handle yet-larger files, you can use a multilevel tree scheme, analogous to multilevel page tables.

Inode

file block 0's disk block number
⋮
file block 11's disk block number
indirect block's block number
access permissions
other metadata

Indirect block

file block 12's disk block number
⋮
file block 1035's disk block number

Figure 8.8: If an inode were used with a single indirect block, the block numbers would be stored as shown here. Note that the indirect block is actually considerably larger than the inode, contrary to its appearance in the figure.

The inode can contain a block number for a double indirect block, which contains block numbers for many more single indirect blocks, each of which contains many data block numbers. Figure 8.9 shows this enhancement to the inode design, which retains the dozen direct blocks and the original single indirect block, while adding a double indirect block.

Because the double indirect block points at many indirect blocks, each of which points at many data blocks, files can now grow quite large. (In Exercise 8.10, you can figure out just how large.) However, many UNIX file systems go one step further by allowing the inode to point to a triple indirect block as well, as shown in Figure 8.10. Comparing this with multilevel page tables is illuminating; the very unbalanced tree used here allows a small, shallow tree to grow into a large, deeper tree in a straightforward way. Later you'll see that B^+-trees grow somewhat less straightforwardly, but without becoming so imbalanced.

Having presented this method of mapping file blocks into disk blocks, I will shortly turn to an alternative that avoids storing information on a per-block basis. First, however, it is worth drawing one more analogy with page tables. Just as a page table need not provide a page frame number for every page (if some pages are not in memory), an inode or indirect block need not provide a disk block number for every block of the file. Some entries can be left blank, typically by using some reserved value that cannot be mistaken for a legal disk block number. This is valuable for *sparse files*, also known as files with *holes*. A sparse file has one or more large portions containing nothing but zeros, usually because those portions have never been written. By not allocating disk blocks for the all-zero file blocks, the file system can avoid wasting space and time.

Extent Maps

You have seen that traditional inodes and indirect blocks are based around the notion of a *block map*, that is, an array specifying a disk block number for each file block. A block map is completely general, in that each file block can be mapped to any disk block. File block n can be mapped somewhere totally different on disk from file block $n-1$. Recall, however, that file system designers prefer not to make use of this full generality. For performance reasons, consecutive file blocks will normally be allocated consecutive disk blocks, forming long extents. This provides the key to a more efficient data structure for storing the mapping information.

Suppose you have a file that is 70 blocks long and that occupies disk blocks 1000–1039 and 1200–1229. A block map would contain each one

Inode

file block 0's disk block number
⋮
file block 11's disk block number
single indirect block's number
double indirect block's number
access permissions
other metadata

Single indirect block

file block 12's disk block number
⋮
file block 1035's disk block number

Double indirect block

indirect block 1's block number
⋮
indirect block 1024's block number

Indirect block 1

file block 1036's disk block number
⋮
file block 2059's disk block number

Indirect blocks 2–1024: similar to indirect block 1

Figure 8.9: If an inode were used with single and double indirect blocks, the block numbers would be stored as shown here.

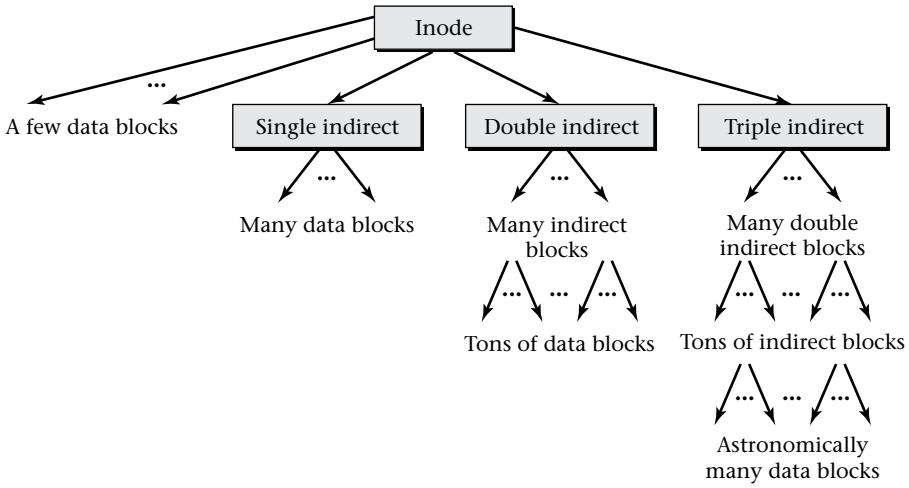


Figure 8.10: The full structure of a file starts with an inode and continues through a tree of single, double, and triple indirect blocks, eventually reaching each of the data blocks.

of those 70 disk block numbers. An *extent map*, on the other hand, would contain only two entries, one for each of the file's extents, just as the opening sentence of this paragraph contains two ranges of block numbers. Each entry in the extent map needs to contain enough information to describe one extent. There are two alternatives for how this can be done:

- Each entry can contain the extent's length and starting disk block number. In the example, the two extent map entries would be $(40, 1000)$ and $(30, 1200)$. These say the file contains 40 blocks starting at disk block 1000 and 30 blocks starting at disk block 1200.

- Each entry can contain the extent's length, starting file block number, and starting disk block number. In the example, the two extent map entries would be $(40, 0, 1000)$ and $(30, 40, 1200)$. The first entry describes an extent of 40 blocks, starting at position 0 in the file and occupying disk blocks starting with number 1000. The second entry describes an extent of 30 blocks, starting at position 40 in the file and occupying disk blocks starting with number 1200.

The first approach is more compact. The second approach, however, has the advantage that each extent map entry can be understood in isolation, without needing to read the preceding extent map entries. This is particularly useful if the extent map is stored in a B^+-tree, as I will discuss subsequently. For simplicity, I will assume the second approach in the remainder of my discussion, though there are systems that use each.

At first, it may not be obvious why extent maps are a big improvement. A typical block map system might use a 4-byte block number to refer to each 4-KB block. This is less than one-tenth of one percent space overhead, surely affordable with today's cheap disk storage. What reason do file system designers have to try to further reduce such an already small overhead? (I will ignore the possibility that the extent map takes more space than the block map, which would happen only if the file is scattered into lots of tiny extents.)

The key fact is that disk space efficiency turns into time efficiency, which is a much more precious commodity. Indirect blocks result in extra disk I/O operations. Consider, for example, reading a file that is stored in a single 20-block extent. With the block map approach, the file system would need to do at least two disk read operations: one to read the single indirect block and one to read the data blocks. This assumes the inode is already cached in memory, having been read in along with other inodes in its disk block, and that the file system is smart enough to read all 20 data blocks in a single

operation. With an extent map, the entire mapping information would fit in the inode; if you again assume the inode is cached, a single read operation suffices. Thus, the system can read files like this twice as fast. Admittedly, this is a somewhat artificial best-case example. However, even with realistic workloads, a significant speedup is possible.

Several modern file systems use extent maps, including Microsoft Windows' NTFS, Mac OS X's HFS Plus, and XFS, which was ported into Linux from SGI's IRIX version of UNIX. For files that have only a handful of extents (by far the most common case), all three store the sequence of extent map entries in the inode or (in Windows and Mac OS X) in the corresponding inode-like structure. The analogs of inodes in NTFS are large enough (1 KB) that they can directly store entire extent maps for most files, even those with more than a few extents. The other two file systems use smaller inodes (or inode-like structures) and so provide an interesting comparison of techniques for handling the situation where extra space is needed for a large extent map.

HFS Plus takes an approach quite reminiscent of traditional UNIX inodes: the first eight extent map entries are stored directly in the inode-like structure, whether they are the only ones or just the first few of a larger number. Any additional entries are stored elsewhere, in a single B^+-tree that serves for all the files, as I will describe subsequently. XFS, on the other hand, stores all the extent map entries for a file in a file-specific B^+-tree; the space in the inode is the root node of that tree. When the tree contains only a few extents, the tree is small enough that the root of the tree is also a leaf, and so the extents are directly in the inode, just as with HFS Plus. When the extent map grows larger, however, all the entries move down into descendant nodes in the tree, and none are left in the inode, unlike HFS Plus's special treatment of the first eight.

B-Trees

The *B-tree* data structure is a balanced search tree structure generally configured with large, high-degree nodes forming shallow, bushy trees. This property makes it well suited to disk storage, where transferring a large block of data at once is efficient (hence, large nodes), but performing a succession of operations is slow (hence, a shallow tree). You may have encountered B-trees before, in which case my summary will be a review, with the exception of my description of specific applications for which this structure is used.

Any B-tree associates search keys with corresponding values, much like a dictionary associates words with their definitions or a phone book associates

names with phone numbers. The keys can be textual strings organized in alphabetic order (as in these examples) or numbers organized by increasing value; all that is required is that there is some way to determine the relative order of two keys.

The B-tree allows entries to be efficiently located by key, as well as inserted and deleted. Thus far, the same could be said for a hash table structure, such as is used for hashed page tables. Where B-trees (and other balanced search trees) distinguish themselves is that they also provide efficient operations based on the ordering of keys, rather than just equality of keys. For example, if someone asks you to look up "Smit" in a phone book, you could reply, "There is no Smit; the entries skip right from Smirnoff to Smith." You could do the same with a B-tree, but not with a hash table.

This ability to search for neighbors of a key, which need not itself be present in the tree, is crucial when B-trees are used for extent maps. Someone may want information about the extent containing file block 17. There may be no extent map entry explicitly mentioning 17; instead, there is an entry specifying a 10-block extent starting with file block 12. This entry can be found as the one with the largest key that is less than or equal to 17.

B-trees can play several different roles in persistence systems. In Section 8.6, you'll see their use for directories of file names and for indexes of database contents; both are user-visible data access services. In the current section, B-trees play a more behind-the-scenes role, mapping positions within a file to locations on disk. Earlier, in Section 8.4.3, you saw another related use, the management of free space for allocation. The data structure fundamentals are the same in all cases; I choose to introduce them here, because extent maps seem like the simplest application. Free space mapping is complicated by the dual indexing (by size and location), and directories are complicated by the use of textual strings as keys.

You are probably already familiar with binary search trees, in which each tree node contains a root key and two pointers to subtrees, one with keys smaller than the root key, and one with keys larger than the root key. (Some convention is adopted for which subtree contains keys equal to the root key.) B-tree nodes are similar, but rather than using a single root key to make a two-way distinction, they use N root keys to make an $N+1$ way distinction. That is, the root node contains N keys (in ascending order) and $N+1$ pointers to subtrees, as shown in Figure 8.11. The first subtree contains keys smaller than the first root key, the next subtree contains keys between the first and second root keys, and so forth. The last subtree contains keys larger than the last root key.

If a multi-kilobyte disk block is used to hold a B-tree node, the value

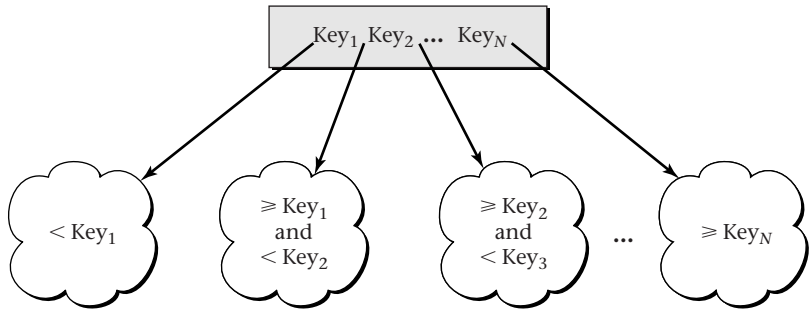


Figure 8.11: A B-tree node contains N keys and $N + 1$ pointers to the subtrees under it. Each subtree contains keys in a particular range.

of N can be quite large, resulting in a broad, shallow tree. In fact, even if a disk block were only half full with root keys and subtree pointers, it would still provide a substantial branching factor. This observation provides the inspiration for the mechanism used to maintain B-trees as entries are inserted.

Each node is allowed to be anywhere between half full and totally full. This flexibility means one can easily insert into a node, so long as it is less than full. The hard case can be handled by splitting nodes. As a special exception, the root node is not required to be even half full. This exception allows you to build a tree with any number of entries, and it adds at most one level to the height of the tree.

Consider, for example, inserting one more entry into an already full node. After insertion, you have $N+1$ keys but only room for N. The node can be replaced with two nodes, one containing the $N/2$ smallest keys and the other the $N/2$ largest keys. Thus, you now have two half-full nodes. However, you have only accounted for N of the $N+1$ keys; the median key is still left over. You can insert this median key into the parent node, where it will serve as the divider between the two half-full nodes, as shown in Figure 8.12.

When you insert the median key into the parent node, what if the parent node is also full? You split the parent as well. The splitting process can continue up the tree, but because the tree is shallow, this won't take very long. If the node being split has no parent, because it is the root of the tree, it gains a new parent holding just the median key. In this way the tree grows in height by one level.

In Bayer and McCreight's 1972 paper introducing B-trees, they suggested that each node contain key/value pairs, along with pointers to subtrees. Practical applications today instead use a variant, sometimes called B^+-

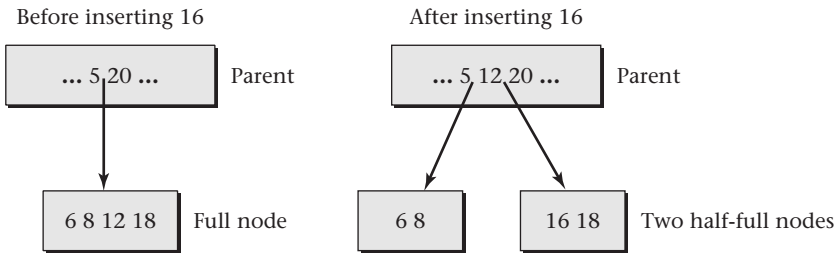


Figure 8.12: Inserting 16 into the illustrated B-tree, which has node-capacity 4, causes a node to split, with the median key moving into the parent.

trees. In a B^+-tree, the nonleaf nodes contain just keys and pointers to subtrees, without the keys having any associated values. The keys in these nodes are used solely for navigation to a subtree. The leaves contain the key/value pairs that are the actual contents of the data structure. For example, a small B^+-tree of extent map entries might be organized as shown in Figure 8.13.

This sort of B^+-tree can store the extent map for a single file, as is done in XFS. For Mac OS X's HFS Plus, a slightly different approach is needed, because all files' extent maps are combined into a single B^+-tree. (Recall, though, that the first eight extents of each file are not included in this tree.)

Each entry in this file system's B^+-tree describes an extent map entry for some position within some file. That is, the entry contains a file number (analogous to an inode number), a starting block number within the file, a length in blocks, and a starting disk block number. The concatenation of file number and starting file block number serves as the key. That way, all the entries for a particular file appear consecutively in the tree, in order by their position within the file.

The insertion algorithm for B^+-trees is a slight variant of the one for pure B-trees; you can work through the differences in Exercise 8.13.

8.5.2 Access Control Metadata

The complexity of the data structures storing access control information is directly related to the sophistication of the protection system. Recall that the POSIX specification, followed by UNIX and Linux, provides for only fixed-length access control lists (ACLs), with permissions for a file's owner, owning group, and others. This information can be stored compactly in the file's inode. Microsoft Windows, on the other hand, allows much more general ACLs. Thus, the designers of NTFS have faced a more interesting

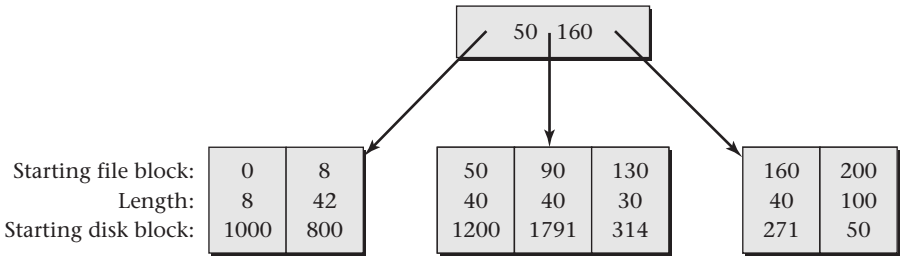


Figure 8.13: This small B^+-tree extent map contains information that can be used to find each extent's range of file block numbers and range of disk block numbers. Because the tree is a B^+-tree rather than a B-tree, all the extents are described in the leaves, with the nonleaf node containing just navigational information.

challenge and, in fact, have revisited their design decision, as you will see.

For POSIX-compliant access control, an inode can contain three numbers: one identifying the file's owning user, one identifying the file's owning group, and one containing nine bits, representing the `rwx` permissions for the owning user, the owning group, and other users. This third number, containing the nine permission bits, is called the file's *mode*. Rather than waste all but nine bits in the mode, the others are used to encode additional information, such as whether the file is a regular file, a directory, an I/O device, and so forth. Figure 8.14 shows how the permissions can be determined by extracting an inode's mode using the `stat` system call. (This system call differs only slightly from `fstat`, which you saw earlier. The file is specified by name, rather than by a numerical file descriptor.) If you compile this C++ program and call the resulting executable `stater`, then a command like `./stater somefile` should produce information you could also get with `ls -l somefile`.

Early versions of NTFS stored the full ACL for each file independently. If the ACL was small enough to fit in the inode-like structure, it was stored there. Otherwise, it was stored in one or more extents of disk blocks, just like the file's data, and the inode-like structure contained an extent map for the ACL.

As of Windows 2000, Microsoft redesigned NTFS to take advantage of the fact that many files have identical ACLs. The contents of the ACLs are now stored in a centralized database. If two files have identical ACLs, they can share the same underlying representation of that ACL.

```
#include <unistd.h>
#include <time.h>
#include <sys/stat.h>
#include <stdio.h>
#include <iostream>
using namespace std;

static void print_bit(int test, char toPrint){
  if(test)
    cout << toPrint;
  else
    cout << '-';
}

int main(int argc, char *argv[]){
  if(argc != 2){
    cerr << "Usage: " << argv[0] << " filename" << endl;
    return -1;
  }
  struct stat info;
  if(stat(argv[1], &info) < 0){
    perror(argv[1]);
    return -1;
  }
  print_bit(info.st_mode & S_IRUSR, 'r');
  print_bit(info.st_mode & S_IWUSR, 'w');
  print_bit(info.st_mode & S_IXUSR, 'x');
  print_bit(info.st_mode & S_IRGRP, 'r');
  print_bit(info.st_mode & S_IWGRP, 'w');
  print_bit(info.st_mode & S_IXGRP, 'x');
  print_bit(info.st_mode & S_IROTH, 'r');
  print_bit(info.st_mode & S_IWOTH, 'w');
  print_bit(info.st_mode & S_IXOTH, 'x');
  cout << endl;
  return 0;
}
```

Figure 8.14: This C++ program, `stater.cpp`, uses `stat` to retrieve access control metadata for whichever file is specified by the command-line argument `argv[1]`.

8.5.3 Other Metadata

Because files can be of any length, not just a multiple of the block size, each inode (or equivalent) contains the file's size in bytes. (The program in Figure 8.3 on page 340 showed how you can retrieve this information.) Other metadata is much more system-specific. For example, POSIX specifies that each file has three time stamps, recording when the file was last accessed, last written, and last modified in any way. Modification includes not only writing the data, but also making changes in permission and other metadata attributes. NTFS records whether the file should be hidden in ordinary directory listings. HFS Plus has many metadata attributes supporting the graphical user interface; for example, each file records its icon's position.

One metadata attribute on POSIX systems connects with file linking, that is, the use of multiple names for one file, which is the topic of Section 8.6.3. Each file's inode contains a count of how many names refer to the file. When that count reaches zero and the file is not in use by any process, the operating system deletes the file. The operation users normally think of as deleting a file actually just removes a name; the underlying file may or may not be deleted as a consequence.

8.6 Directories and Indexing

Having seen how file systems provide the storage for files, you are now ready to consider how those systems allow files to be located by name. As a similar question regarding database systems, you can consider how those systems provide indexed lookup. In Section 8.6.1, I set the stage for this discussion by presenting a common framework for file directories and database indexes, showing the ways in which they differ. In Section 8.6.2, I show how the separation between file directories and database indexes is currently weakening with the introduction of indexing mechanisms for locating files. Having shown the basic principles of both directories and indexes, I use Section 8.6.3 to dig into one particular aspect of file directories in more detail: the ways in which multiple names can refer to a single file. Finally, in Section 8.6.4, I take you behind the scenes to look at typical data structures used for directories and indexes.

8.6.1 File Directories Versus Database Indexes

Traditionally, file systems include *directories*, which provide access to files by name. Databases, on the other hand, include *indexes*, which provide

access to entries in the database based on a portion of the contents. This clean distinction between file systems and databases is currently blurring, as alternative file-access techniques based on indexes become available. In particular, Apple introduced such a feature in Mac OS X version 10.4 under the name Spotlight. I describe Spotlight in Section 8.6.2. Microsoft subsequently included a related feature in Windows Vista. This trend makes it even more important to see what directories and indexes have in common and what distinguishes them.

Both directories and indexes provide a mapping from keys to objects. The keys in a directory are names, which are external to the object being named. You can change the contents of a file without changing its name or change the name without changing the contents. In contrast, the keys in an index are attributes of the indexed objects, and so are intrinsic to those objects. For example, an index on a database table of chapters might allow direct access to the row with title `"Files and Other Persistent Storage"` or with the number `8`. If the row were updated to show a change in this chapter's title or number, the index would need to be updated accordingly. Similarly, any update to the index must be in the context of a corresponding change to the indexed row; it makes no sense to say that you want to look up the row under chapter number `1`, but there find that the real chapter number is still `8`.

Each name in a directory identifies a single file. Two files may have the same name in different directories, but not in the same directory. Database indexes, on the other hand, can be either for a unique attribute or a non-unique one. For example, it may be useful to index a table of user accounts by both the unique login name and the non-unique last name. The unique index can be used to find the single record of information about the user who logs in as `"jdoe"`, whereas the non-unique index can be used to find all the records of information about users with last name `"Doe"`. An index can also use a combination of multiple attributes as its key. For example, a university course catalog could have a unique index keyed on the combination of department and course number.

The final distinction between file directories and database indexes is the least fundamental; it is the kind of object to which they provide access. Traditionally, directories provide access to entire files, which would be the analog of tables in a relational database. Indexes, on the other hand, provide access not to entire tables, but rather to individual rows within those tables. However, this distinction is misleading for two reasons:

- Database systems typically have a meta-table that serves as a catalog

of all the tables. Each row in this meta-table describes one table. Therefore, an index on this meta-table's rows is really an index of the tables. Access to its rows is used to provide access to the database's tables.

- As I mentioned earlier, operating system developers are incorporating indexes in order to provide content-based access to files. This is the topic of Section 8.6.2.

8.6.2 Using Indexes to Locate Files

As I have described, files are traditionally accessed by name, using directories. However, there has been considerable interest recently in using indexes to help users locate files by content or other attributes. Suppose that I could not remember the name of the file containing this book. That would not be a disaster, even leaving aside the possibility that the world might be better off without the book. I could search for the file in numerous ways; for example, it is one of the few files on my computer that has hundreds of pages. Because the Mac OS X system that I am using indexes files by page count (as well as by many other attributes), I can simply ask for all files with greater than 400 pages. Once I am shown the five files meeting this restriction, it is easy to recognize the one I am seeking.

The index-based search feature in Mac OS X, which is called Spotlight, is not an integral component of the file system in the way directories and filenames are. Instead, the indexing and search are provided by processes external to the operating system, which can be considered a form of middleware.

The file system supports the indexing through a generic ability to notify processes of events such as the creation or deletion of a file, or a change in a file's contents. These events can be sent to any process that subscribes to them and are used for other purposes as well, such as keeping the display of file icons up to date. The Spotlight feature uses it to determine when files need reindexing. When I save out a new version of my book, the file system notifies Spotlight that the file changed, allowing Spotlight to update indexes such as the one based on page count. Unlike file directories, which are stored in a special data structure internal to the file system, the indexes for access based on contents or attributes like page counts are stored in normal files in the `/.Spotlight-V100` directory.

Apple refers to the indexed attributes (other than the actual file contents) as metadata. In my book example, the number of pages in a docu-

ment would be one piece of metadata. This usage of the word "metadata" is rather different from its more traditional use in file systems. Every file has a fixed collection of file system metadata attributes, such as owner, permissions, and time of last modification. By contrast, the Spotlight metadata attributes are far more numerous, and the list of attributes is open-ended and specific to individual types of files. For example, while the file containing my book has an attribute specifying the page count, the file containing one of my vacation photos has an attribute specifying the exposure time in seconds. Each attribute makes sense for the corresponding file, but would not make sense for the other one.

As you have seen, the metadata attributes that need indexing are specific to individual types of files. Moreover, even common attributes may need to be determined in different ways for different types of files. For example, reading a PDF file to determine its number of pages is quite different from reading a Microsoft Word file to determine its number of pages—the files are stored in totally different formats. Therefore, when the indexing portion of Spotlight receives notification from the file system indicating that a file has changed, and hence should be indexed, it delegates the actual indexing work to a specialist indexing program that depends on the type of file. When you install a new application program on your system, the installation package can include a matching indexing program. That way you will always be able to search for files on your system using relevant attributes, but without Apple having had to foresee all the different file types.

8.6.3 File Linking

Indexed attributes, such as page counts, are generally not unique. My system may well have several five-page documents. By contrast, you have already seen that each name within a directory names a single file. Just because each pathname specifies a single file does not mean the converse is true, however. In this subsection, I will explain two different ways in which a file can be reachable through multiple names.

The most straightforward way in which multiple names can reach a single file is if the directory entry for each of the names specifies the same file. Figure 8.15 shows a directory with two names, both referring to the same file. In interpreting this figure, you should understand that the box labeled as the file does not denote just the data contained in the file, but also all of the file's metadata, such as its permissions. In the POSIX API, this situation could have arisen in at least two different ways:

- The file was created with the name `alpha`, and then the procedure call

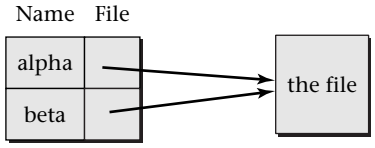


Figure 8.15: A directory can contain two names for one file.

`link("alpha", "beta")` added the name `beta`.

- The file was created with the name `beta`, and then the procedure call `link("beta", "alpha")` added the name `alpha`.

No matter which name is the original and which is added, the two play identical roles afterward, as shown in Figure 8.15. Neither can be distinguished as the "real" name. Often people talk of the added name as a *link* to the file. However, you need to understand that *all* file names are links to files. There is nothing to distinguish one added with the `link` procedure.

POSIX allows a file to have names in multiple directories, so long as all the directories are in the same file system. In the previous illustration (Figure 8.15), `alpha` and `beta` in the current directory named one file. Instead, I could have had directory entries in multiple directories all pointing at the same file. For example, in Figure 8.16, I show a situation where `/alpha/beta` is a name for the same file as `/gamma/delta`.

To keep the directory structure from getting too tangled, POSIX systems ordinarily do not allow a directory to have more than one name. One exception is that each directory contains two special entries: one called `.` that is an extra link to that directory itself and one called `..` that is an extra link to its parent directory.

Just as `link` adds a name for a file, `unlink` removes a name. For example, `unlink("/alpha/beta")` would eliminate one of the two routes to the file in Figure 8.16 by removing the `beta` entry from the directory `alpha`. As mentioned earlier, removing a name only implicitly has anything to do with removing a file. The operating system removes the file when it no longer has any names and is no longer in use by any process. (An open file can continue to exist without any names, as you can demonstrate in Exploration Project 8.10.)

POSIX also supports another alternative for how multiple names can lead to one file. One name can refer to another name and thereby indirectly refer to the same file as the second name. In this situation, the first name is called a *symbolic link*. Figure 8.17 shows an example, where `alpha`

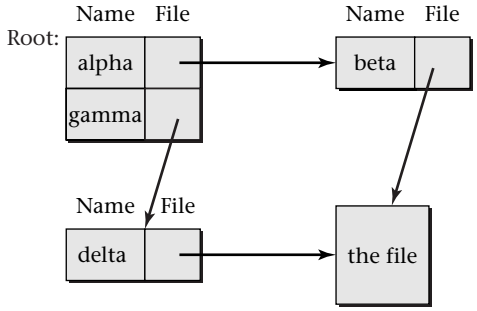


Figure 8.16: A file can have two different names, each in its own directory. In this example, the two pathnames `/alpha/beta` and `/gamma/delta` both lead to the same file.

is specified as a symbolic link to `beta`, and thereby refers to whatever file `beta` does. (Symbolic links are also sometimes called *soft links*. Ordinary links are called *hard links* when it is important to emphasize the difference.) In this figure, I show that a directory can map each name to one of two options: either a pointer to a file (which could be represented as an inode number) or another name. The code that looks up filenames, in procedures such as `open`, treats these two options differently. When it looks up `alpha` and finds `beta`, it recursively looks up `beta`, so as to find the actual file. The symbolic link shown in Figure 8.17 could be created by executing `symlink("beta", "alpha")`.

Symbolic links are somewhat tricky, because they can form long chains, dangling references, or loops. In the preceding example, you could form a longer chain by adding `gamma` as a symbolic link to `alpha`, which is already a symbolic link to `beta`. The code for looking up files needs to traverse such chains to their end. However, there may not be a file at the end of the chain. If you were to execute `unlink("beta")`, then you would have a dangling reference: `gamma` would still be a symbolic link to `alpha`, which would still be a

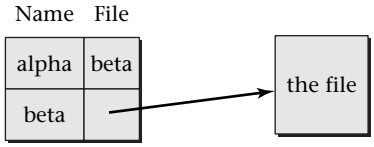


Figure 8.17: A symbolic link allows a file name to refer to a file indirectly, by way of another file name.

symbolic link to `beta`, which wouldn't exist any more. Worse, having deleted `beta`, you could reuse that name as a symbolic link to `alpha`, creating a loop. All POSIX procedures that look up files must return a special error code, `ELOOP`, if they encounter such a situation. In addition to returning `ELOOP` for true loops, these procedures are allowed to return the same error code for any chain of symbolic links longer than some implementation-defined maximum.

Symbolic links are more flexible than hard links. You can create a symbolic link that refers to a directory. You can also create a symbolic link that refers to a file stored in a separate file system. For example, you could have a symbolic link in your main file system, stored on your local disk drive, that refers to a file stored in an auxiliary file system on a network file server. Neither of these options is possible with a hard link.

You can create either a symbolic link or an ordinary hard link from within a shell by using the `ln` command. This command runs a program that will invoke either the `link` procedure or the `symlink` procedure. You can explore this command and the results it produces in Exploration Projects 8.9 and 8.11.

Some file systems outside the UNIX tradition store the metadata for a file directly in that file's directory entry, rather than in a separate structure such as an inode. This tightly binds the name used to reach the file together with the identity of the file itself. In effect, the name becomes an attribute of the file, rather than just a means of accessing the file. In systems of this kind, symbolic links can still be used, but there is no easy analog for hard links. This leads to an interesting situation when one of these systems needs to be retrofitted for POSIX compliance.

For example, Apple's HFS Plus was developed before Mac OS became based on UNIX, which happened in Mac OS X. The underlying design assumes that each file has exactly one name and fuses together the directory and metadata structures. Yet Mac OS X is a UNIX system and so needs to support files with multiple names (created with `link`) or no names (if still in use when unlinked). To accommodate this, Apple puts any file that is in either of these situations into a special invisible directory with a random number as its name. Any other names for the file are provided by a special kind of symbolic link, which is made completely invisible to the POSIX API, even to those procedures that normally inspect symbolic links rather than simply following them to their targets.

8.6.4 Directory and Index Data Structures

The simplest data structure for a directory or index is an unordered linear list of key/value pairs. Whereas this is never used for a database index, it is the most traditional approach for directories in UNIX-family file systems and remains in use in many systems to this day. With this structure, the only way to find a directory entry is through linear search. (For a database, unordered linear search is available without any index at all by searching the underlying rows of the database table.)

For small directories, a linear search can perform quite reasonably. Therefore, system administrators often design directory trees so that each directory remains small. For example, my home directory is not `/home/max`, but rather `/home/m/a/max`, where the `m` and `a` come from the first two letters of my username. That way, the `/home` directory has only 26 entries, each of which in turn has 26 entries, each of which has only one small fraction of the thousands of users' home directories. As you will see shortly, this kind of directory tree is no longer necessary with a modern file system. On a modern system, my files could be in `/home/max`, and similarly for the thousands of other users, without a major slowdown—unless, of course, someone listed the contents of `/home`.

A second alternative structure is a *hash table*. A hash table is a numerically indexed array of key/value pairs where software can directly access entry number i without looking at the preceding entries. The trick is to know (most of the time) which entry would contain a particular key; this knowledge comes from using a hash function of the key as the entry number. So long as no two keys collide and are assigned the same location, looking up a particular entry (such as the one for `max` inside the `/home` directory) is a constant-time operation, independent of the table size. All that is necessary is to hash the key into a numerical hash code and use that code to directly access the appropriate entry. If it contains the desired key (`max`), the lookup is complete. If it contains no key at all, the lookup is also complete and can report failure. If, due to a collision, the entry contains some other key than the one being looked for, the system must start searching through alternative locations. That searching, however, can be kept very rare, by ensuring that the table is never very full.

Hash tables are occasionally used for database indexes; in particular, they are an option in PostgreSQL. However, as I mentioned in Section 8.5.1, they have the disadvantage relative to B^+-trees of not supporting order-based accesses. For example, there is no way to use a hash table index to find all rows in an accounting table for payments made within a particular range

of dates. Hash indexes may also not perform as well as B^+-tree indexes; the PostgreSQL documentation cites this as a reason to discourage their use.

Hash tables are also occasionally used for indexing file system directories. In particular, the FFS file system used in BSD versions of UNIX supports a directory hashing extension. This feature builds a hash table in memory for large directories at the time they are accessed. However, the on-disk data structure remains an unsorted linear list.

B^+-trees are the dominant structure for both database indexes and contemporary file systems' directories. I already discussed the structure of B^+-trees in Section 8.5.1 and showed how they provide highly efficient access. As examples, B^+-trees are used for directories in Microsoft's NTFS, in SGI's XFS, and (in a different form) in Apple's HFS Plus.

In most systems, each index or directory is represented by its own B^+-tree. HFS Plus instead puts all the directories' entries together in one big B^+-tree. The keys in this tree are formed by concatenating together the identifying number of the parent directory with the name of the particular child file (or subdirectory). Thus, all the entries within a single directory appear consecutively within the tree.

8.7 Metadata Integrity

When a system crashes, any data held in the volatile main memory (RAM) is lost. In particular, any data that the file system was intending to write to persistent storage, but was temporarily buffering in RAM for performance reasons, is lost. This has rather different implications depending on whether the lost data is part of what a user was writing into a file or is part of the file system's metadata:

- Some user data is noncritical, or can be recognized by a human as damaged and therefore restored from a backup source. Other user data is critical and can be explicitly flushed out to persistent storage under control of the application program. For example, when a relational database system is committing a transaction and needs to ensure that all the log entries are in persistent storage, it can use the POSIX API's `fsync` procedure to force the operating system to write the log file to persistent storage.

- If the last few metadata operations before a crash are cleanly lost in their entirety, this can often be tolerated. However, users cannot tolerate a situation where a crash in the middle of metadata updates

> results in damage to the integrity of the metadata structures themselves. Without those structures to organize the storage blocks into meaningful files, the storage contents are just one big pile of bits. There wouldn't even be any individual files to check for damage.

Therefore, all file systems contain some mechanism to protect the integrity of metadata structures in the face of sudden, unplanned shutdowns. (More extreme hardware failures are another question. If your machine room burns down, you better have an off-site backup.)

Metadata integrity is threatened whenever a single logical transformation of the metadata from one state to another is implemented by writing several individual blocks to persistent storage. For example, extending a file by one data block may require two metadata blocks be written to storage: one containing the inode (or indirect block) pointing at the new data block and another containing the bitmap of free blocks, showing that the allocated block is no longer free. If the system crashes when only one of these two updates has happened, the metadata will be inconsistent. Depending on which update was written to persistent storage, you will either have a lost block (no longer free, but not part of the file either) or, more dangerously, a block that is in use, but also still "free" for another file to claim.

Although having a block "free" while also in use is dangerous, it is not irreparable. If a file system somehow got into this state, a consistency repair program could fix the free block bitmap by marking the block as not free. By contrast, if the situation were to progress further, to the point of the "free" block being allocated to a second file, there would be no clean repair. Both files would appear to have equal rights to the block.

Based on the preceding example, I can distinguish three kinds of metadata integrity violation: irreparable corruption, noncritical reparable corruption, and critical reparable corruption. Irreparable corruption, such as two files using the same block, must be avoided at all costs. Noncritical reparable corruption, such as a lost block, can be repaired whenever convenient. Critical reparable corruption, such as a block that is both in use and "free," must be repaired before the system returns to normal operation.

Each file system designer chooses a strategy for maintaining metadata integrity. There are two basic strategies in use, each with two main variants:

- Each logical change to the metadata state can be accomplished by writing a single block to persstent storage.
  - The single block can be the commit record in a write-ahead log, as I discussed in Section 5.4. Other metadata blocks may be written

as well, but they will be rolled back upon reboot if the commit record is not written. Thus, only the writing of the commit block creates a real state change. This approach is known as *journaling*.

  - Alternatively, if the system always creates new metadata structures rather than modifying existing ones, the single block to write for a state change is the one pointing to the current metadata structure. This approach is known as *shadow paging*.

- Each logical change to the metadata state can be accomplished by writing multiple blocks to persistent storage. However, the order of the updates is carefully controlled so that after a crash, any inconsistencies in the metadata will always be of the reparable kind. A consistency repair program is run after each crash to restore the metadata's integrity by detecting and correcting violations of the metadata structures' invariant properties.

  - The update order can be controlled by performing each metadata update as a *synchronous write*. That is, the file system actually writes the updated metadata block to persistent storage immediately, rather than buffering the write in RAM for later.
  - The update order can be controlled by buffering the updated metadata blocks in RAM for later writing, but with specific annotations regarding the dependencies among them. Before writing a block to persistent storage, the system must write the other blocks upon which it depends. If the same blocks are updated repeatedly before they are written to storage, cyclic dependencies may develop, necessitating additional complications in the mechanism. This approach is known as using *soft updates*.

The strategy of update ordering through synchronous writes was once quite popular. Linux's ext2fs uses this approach, for example. However, performance considerations have removed this approach from favor, and it is unlikely ever to return. The problem is not only that synchronous writes slow normal operation. Far more fatally, as typical file systems' sizes have grown, the consistency repair process necessary after each crash has come to take unacceptably long. Because synchronous writes are expensive, even systems of this kind use them as sparingly as possible. The result is that while all inconsistencies after a crash will be reparable, some may be of the critical kind that need immediate repair. Thus, the time-consuming consistency repair process must be completed before returning the crashed system to service.

Contemporary file systems have almost all switched to the journaling strategy; examples include Linux's ext3fs, Microsoft Windows' NTFS, and Mac OS X's HFS Plus. After rebooting from a crash, the system must still do a little work to undo and redo storage-block updates in accordance with the write-ahead log. However, this is much faster, as it takes time proportional to the amount of activity logged since the last checkpoint, rather than time proportional to the file system size.

Shadow paging has been less widely adopted than journaling. Three examples are the WAFL file system used in Network Appliance's storage servers, the ZFS file system developed by Sun Microsystems, and the btrfs file system for Linux. Network Appliance's choice of this design was motivated primarily by the additional functionality shadow paging provides. Because storage blocks are not overwritten, but rather superseded by new versions elsewhere, WAFL naturally supports *snapshots*, which keep track of prior versions of the file system's contents. Although shadow paging has not become as widespread as journaling, there is more hope for shadow paging than for either form of ordered updates (synchronous writes and soft updates). Increases in the demand for snapshots, the capacity of storage devices, and the utilization of solid-state storage are causing shadow paging to increasingly challenge journaling for dominance.

The soft updates strategy is generally confined to the BSD versions of UNIX. Its main selling point is that it provides a painless upgrade path from old-fashioned synchronous writes. (The in-storage structure of the file system can remain identical.) However, it shares the biggest problem of the synchronous write strategy, namely, the need for post-crash consistency repair that takes time proportional to the file system size.

Admittedly, soft updates somewhat ameliorate the problem of consistency repair. Because soft updates can enforce update ordering restrictions more cheaply than synchronous writes can, file systems using soft updates can afford to more tightly control the inconsistencies possible after a crash. Whereas synchronous write systems ensure only that the inconsistencies are reparable, soft update systems ensure that the inconsistencies are of the noncritical variety, safely reparable with the system up and running. Thus, time-consuming consistency repair need not completely hold up system operation. Even still, soft updates are only a valiant attempt to make the best of an intrinsically flawed strategy.

Because the only strategy of widespread use in contemporary designs is journaling, which I discussed in Section 5.4, I will not go into further detail here. However, it is important that you have a high-level understanding of the different strategies and how they compare. If you were to go further

and study the other strategies, you would undoubtedly be a better-educated computer scientist. The notes section at the end of this chapter suggests further reading on shadow paging and soft updates, as well as on a hybrid of shadow paging and journaling that is known as a *log-structured file system*.

8.8 Polymorphism in File System Implementations

If you have studied modern programming languages, especially object-oriented ones, you should have encountered the concept of *polymorphism*, that is, the ability of multiple forms of objects to be treated in a uniform manner. A typical example of polymorphism is found in graphical user interfaces where each object displayed on the screen supports such operations as "draw yourself" and "respond to the mouse being clicked on you," but different kinds of objects may have different methods for responding to these common operations. A program can iterate down a list of graphical objects, uniformly invoking the draw-yourself operation on each, without knowing what kind each is or how it will respond.

In contemporary operating systems, the kernel's interface to file systems is also polymorphic, that is, a common, uniformly invokable interface of operations that can hide a diversity of concrete implementations. This polymorphic interface is often called a *virtual file system* (*VFS*). The VFS defines a collection of abstract datatypes to represent such concepts as directory entry, file metadata, or open file. Each datatype supports a collection of operations. For example, from a directory entry, one can find the associated file metadata object. Using that object, one can access or modify attributes, such as ownership or protection. One can also use the file metadata object to obtain an open file object, which one can then use to perform read or write operations. All of these interface operations work seamlessly across different concrete file systems. If a file object happens to belong to a file on an ext3fs file system, then the write operation will write data in the ext3fs way; if the file is on an NTFS file system, then the writing will happen the NTFS way.

Operating systems are typically written in the C programming language, which does not provide built-in support for object-oriented programming. Therefore, the VFS's polymorphism needs to be programmed more explicitly. For example, in Linux's VFS, each open file is represented as a pointer to a structure (containing data about the file) that in turn contains a pointer to a structure of file operations. This latter structure contains a pointer to the procedure for each operation: one for how to read, one for how to write,

and so forth. As Figure 8.18 shows, invoking the polymorphic `vfs_write` operation on a file involves retrieving that file's particular collection of file operations (called `f_op`), retrieving the pointer to the particular `write` operation contained in that collection, and invoking it. This is actually quite similar to how object-oriented programming languages work under the hood; in C, the mechanism is made visible. (The `vfs_write` procedure writes a given count of bytes from a buffer into a particular position in the file. This underlies the POSIX `pwrite` and `write` procedures I described earlier.)

8.9 Security and Persistent Storage

When considering the security of a persistent storage system, it is critical to have a clear model of the threats you want to defend against. Are you concerned about attackers who will have access to the physical disk drive, or those who can be kept on the other side of a locked door, at least until the drive is taken out of service? Will your adversaries have sufficient motivation and resources to use expensive equipment? Are you concerned about authorized users misusing their authorization, or are you concerned only about outsiders? Are you concerned about attackers who have motivations to modify or delete data, or only those whose motivation would be to breach confidentiality?

As I explained in Section 7.6, if unencrypted data is written to a disk drive and an attacker has physical access to the drive, then software-based protection will do no good. This leads to two options for the security conscious:

- Write only encrypted data to the disk drive, and keep the key else-

```
ssize_t vfs_write(struct file *file, const char *buf,
                  size_t count, loff_t *pos){
  ssize_t ret;

  ret = file->f_op->write(file, buf, count, pos);
  return ret;
}
```

Figure 8.18: Linux's `vfs_write` procedure, shown here stripped of many details, uses pointers to look up and invoke specific code for handling the write request.

where. This leads to the design of *cryptographic file systems*, which automatically encrypt and decrypt all data.

- Keep the attacker from getting at the drive. Use physical security such as locked doors, alarm systems, and guards to keep attackers away. This needs to be coupled with careful screening of all personnel authorized to have physical access, especially those involved in systems maintenance.

Keeping security intact after the disk is removed from service raises further issues. Selling used disks can be a very risky proposition, even if the files on them have been deleted or overwritten.

File systems generally delete a file by merely updating the directory entry and metadata to make the disk blocks that previously constituted the file be free for other use. The data remains in the disk blocks until the blocks are reused. Thus, deletion provides very little security against a knowledgeable adversary. Even if no trace remains of the previous directory entry or metadata, the adversary can simply search through all the disk blocks in numerical order, looking for interesting data.

Even overwriting the data is far from a sure thing. Depending on how the overwriting is done, the newly written data may wind up elsewhere on disk than the original, and hence not really obscure it. Even low-level software may be unable to completely control this effect, because disk drives may transparently substitute one block for another. However, carefully repeated overwriting by low-level software that enlists the cooperation of the disk drive controller can be effective against adversaries who do not possess sophisticated technical resources or the motivation to acquire and use them.

For a sophisticated adversary who is able to use magnetic force scanning tunneling microscopy, even repeatedly overwritten data may be recoverable. Therefore, the best option for discarding a drive containing sensitive data is also the most straightforward: physical destruction. A disk shredder in operation is an awesome sight to behold. If you've never seen one, you owe it to yourself to watch one of the videos available on the web.

Having talked about how hard it is to remove all remnants of data from a drive, I now need to switch gears and talk about the reverse problem: data that is too easily altered or erased. Although magnetic storage is hard to get squeaky clean, if you compare it with traditional paper records, you find that authorized users can make alterations that are not detectable by ordinary means. If a company alters its accounting books after the fact, and those books are real books on paper, there will be visible traces. On the

other hand, if an authorized person within the company alters computerized records, who is to know?

The specter of authorized users tampering with records opens up the whole area of auditability and internal controls, which is addressed extensively in the accounting literature. Recent corporate scandals have focused considerable attention on this area, including the passage in the United States of the Sarbanes-Oxley Act, which mandates tighter controls. As a result of implementing these new requirements, many companies are now demanding file systems that record an entire version history of each file, rather than only the latest version. This leads to some interesting technical considerations; the end-of-chapter notes provide some references on this topic. Among other possibilities, this legal change may cause file system designers to reconsider the relative merits of shadow paging and journaling.

Authorized users cooking the books are not the only adversaries who may wish to alter or delete data. One of the most visible form of attack by outsiders is vandalism, in which files may be deleted wholesale or defaced with new messages (that might appear, for example, on a public web site). Vandalism raises an important general point about security: security consists not only in reducing the risk of a successful attack, but also in mitigating the damage that a successful attack would do. Any organization with a significant dependence on computing should have a contingency plan for how to clean up from an attack by vandals.

Luckily, contingency planning can be among the most cost-effective forms of security measures, because there can be considerable sharing of resources with planning for other contingencies. For example, a backup copy of data, kept physically protected from writing, can serve to expedite recovery not only from vandalism and other security breaches, but also from operational and programming errors and even from natural disasters, if the backup is kept at a separate location.