6 Basic File Management

Storage devices provide persistent storage for a computer. Information can be written to a storage device, and then the computer can be power cycled without destroying the information on the storage device. They are fundamental to most information technology tasks.

However, storage devices are not particularly programmer friendly. Data is written to, and read from, the device in fixed-length blocks of bytes. When, say, a 32-bit integer is to be written to a storage device, the programmer would have to create a buffer in the executable memory that is the same size as the storage device block, write the 4 bytes that represent the integer into the buffer, and then write the buffer contents to the storage device. Programmers need another layer of abstraction on top of storage devices that will simplify the task of writing/reading data to/from the storage device. This storage abstraction will handle the details of creating temporary in-memory buffers, filling them up, and then "automatically" writing the buffer contents to disk when the buffer is full. Similarly, for input operations, this abstraction enables a program to read a only few bytes from a block storage device. That is, the abstraction software reads a full block from the device into the executable memory, and then retrieves the few bytes that the program requires for its current operations.

As you know from your previous programming experience, files are such an abstraction. Operating systems began supporting files before the 1960s – about a half a century ago. Beyond device abstraction, files were the first really significant abstraction of another abstraction in operating systems – an approach that is common in modern operating systems. The file manager is the part of the OS that creates the file abstraction on top of the executable memory and device abstractions.

6.1 Files

A **file** is a named sequence of bytes (see Figure 6-1). Each byte in the file is uniquely referenced by its location in the sequence. File implementations are optimized for **sequential access**: after reading byte i, the file abstraction is designed to minimize the effort required to read or write byte i+1. The **file pointer** is a variable associated with a file that references a single byte position in the file. File operations are applied to file at the location specified by the current setting of the file pointer.

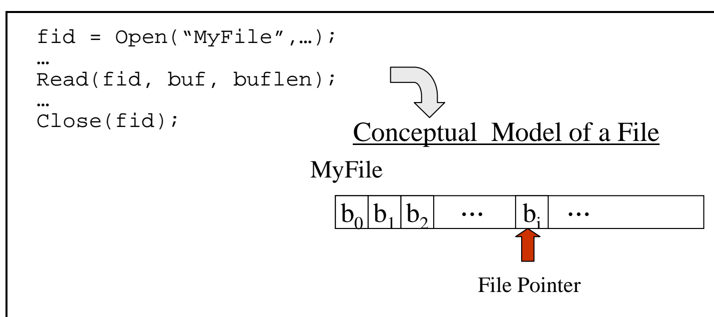

Figure 6-1: Conceptual Model of a File

Creative Commons Attribution-Share Alike 3.0 U.S. License

Programs represent data using various data types – including characters, integers, floating point numbers, and programmer-defined data types such as C structs or C++/Java objects. Thus some of the instances of data types are use only a single byte (for a character), while an instance of programmer-defined data types can require an unbounded number of bytes for storage. For example, consider the C code fragment shown in Figure 6-2. Let's suppose that in the machine that is executing this code fragment, a character is stored in a byte, an integer uses 4 bytes, and a floating point number requires 8 bytes. If the data structures in the code fragment were written to the screen, then the user would see the character stored in c[0], then the integer stored in j[0], and so on. Let's suppose that instead, the program writes the stream of data structure values directly to a storage device that has a block size of 512 bytes. When the program writes c[0], if we were to write that byte to the disk storage device, then a 512 byte block would have the character stored in c[0] written into the first byte of the block, and the remaining 511 bytes would have undefined values written to them (for example, a byte containing zero).

```
 …
char c[N];
int i, j[N], fid;
float x[N];
struct foobar (
    char f1;
    int f2;
    float f3;
} s[N];
…
for(i=0; i<N; i++) {
    fid = open("file_name", O_RDWR);
    Write(fid, &c[i], sizeof(char));
    Write(fid, &j[i], sizeof(int));
    Write(fid, &x[i], sizeof(float));
    Write(fid, &(s[i].f1), sizeof(char));
    Write(fid, &(s[i].f2, sizeof(int));
    Write(fid, &(s[i].f3), sizeof(float));
…
```

Figure 6-2: Sample Code to Write a Data Structure to a Storage Device

In the example code fragment, the first time through the loop, the program will write 1 byte for `c[0]`, 4 bytes for `j[0]`, 8 bytes for `x[0]`, and 13 bytes for `s[0]`, or 26 bytes. This will not fill up a block, so these 26 bytes would have to be placed in a 512-byte buffer that would be filled before the buffer would be copied into a block on the disk. After the second iteration, the buffer would have 2*26 = 52 bytes, and so on until the loop has been executed 19 times the buffer will have 19*26 = 494 bytes filled. On the 20th iteration, c[19] fills the 495th byte, j[19] fills byte numbers 496-499, x[19] fills byte numbers 500-507, s[19].f1 fills byte 508, s[19].f2 fills bytes 509-512, at which point the buffer is full so it can be written to an available disk block. After the buffer has been copied, s[19].f3 is copied into the first 8 bytes, c[20] is copied into the 9th byte, and so on. Hopefully it is apparent that it is laborious to keep track of the buffer as it is being filled, copying the buffer to a disk block, and so on. This example hints at why files are a useful abstraction. Figure 6-3 illustrates the contents of the first couple of blocks that would be written to the storage device by this code fragment.

Creative Commons Attribution-Share Alike 3.0 U.S. License

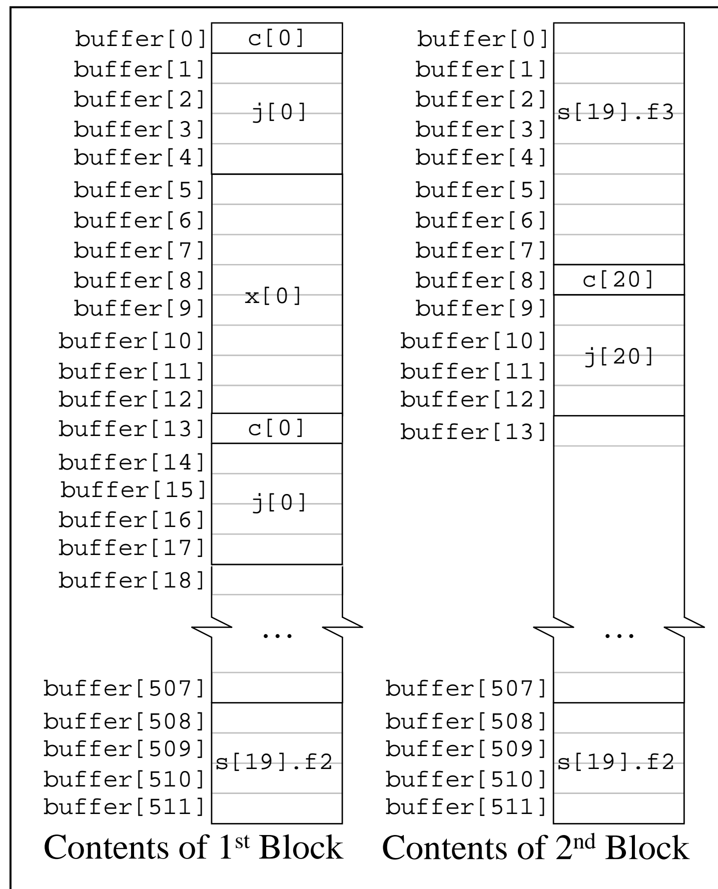

Figure 6-3: Storing the Output Byte Stream into Dev ice Blocks

> The **file manager** is the component of the OS that implements the file abstraction. It exports a simple system call interface – the same one used for all abstract devices, shown in Section 4.3.1. It uses the device abstraction that is implemented by the device manager.

All operating systems have a file manager, but an ST file manager is not intended to handle thousands of files. This is because the capacity of the ST SCC storage devices is usually limited, so the computer's secondary storage device capacity – its **storage system** – is typically quite small compared to file systems on conventional desktop and server machines. The SCC storage system sometimes a "RAMdisk," meaning a block of executable memory that is reserved to save file images – of course the contents of a RAMdisk are not preserved through machine power cycles, so you should think of a RAMdisk as a temporary **file cache** (or place to temporarily store copies of files) that persists as long as the computer remains in operation. There are also various kinds of small capacity, small form factor persistent storage device such as a Compact Flash (CF) or Secure Digital (SD) memory cards. There are other removable media that can keep a copy of a file through power cycling, however their memory capacities are much smaller than hard disks (usually less than 1% the size of hard disks).

Creative Commons Attribution-Share Alike 3.0 U.S. License

File managers (large and small) include a mechanism to organize a collection of files on associated storage device(s). Operating systems use directories as the abstraction for organizing files. A **directory** is a logical collection of files. Since a ST file manager is only intended to handle small numbers of files, it does not necessarily provide a comprehensive and scalable directory mechanism (for example it may not provide the conventional hierarchical directories that you are used to seeing on your favorite notebook or desktop computer).

Briefly a file manager is responsible for the following tasks:

- Implementing the file abstraction
    - Reading and writing blocks of data from/to storage devices
    - Buffering blocks in executable memory
    - Marshalling and unmarshalling byte streams
    - Implementing device independence
    - Adding/deleting blocks to/from a file
- Providing directories for organizing collections of files
- Managing files in the host distributed system

6.2 File Implementation

In the general view, a file is an abstraction of the abstract storage devices, so we can think of it as a *second-level abstraction* built on other OS abstractions rather than on a physical resource. Since the file manager uses the services of the device manager, logically it needs to be able to use the system call interface that the device manager exports. As a practical matter, the device manager would normally be designed to export an internal API that is used by other OS software, such as the file manager, when it wants to read/write devices.

The original Unix design[1] created the idea of *using the same API* for files and devices. This idea has persevered in Unix systems since the early 1970s, and is used in many other systems (such as Windows). In our hypothetical ST system call interface, we will follow this idea, meaning the file manager exports the same system calls as the device manager, where the abstract device name can also be a file name.

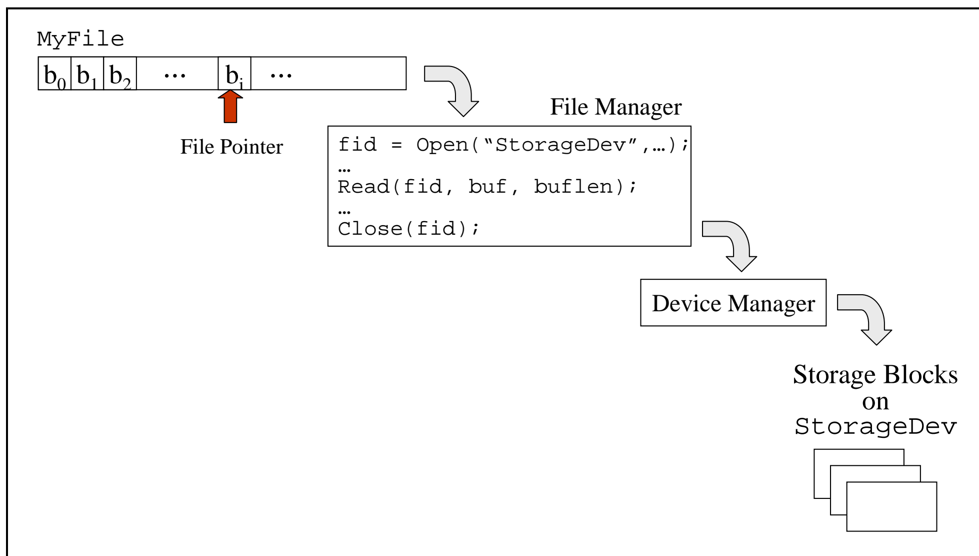

Figure 6-4: The File Manager's Job

[1] See Section 3 of the classic Unix paper [Ritchie and Thompson, 1974].

Creative Commons Attribution-Share Alike 3.0 U.S. License

The file manager implements the named byte stream abstraction by providing functions to open, close, read, write, poll, and seek on the byte stream. It also provides an `Ioctl()` function to perform file-specific operations. As suggested by Figure 6-4, these functions reference the underlying storage by relying on the appropriate storage device manager to handle the details of device I/O, thereby enabling the file manager design to be relatively independent of the nature of the underlying storage device. (For example, the storage device could be any kind of hard disk, a floppy disk, a CD-ROM, or a tape drive.) The file manager functions have a lot of work to do: the collective functions have to define and manage data structures to describe all of the characteristics of a file, including its name, length, access permissions, owner, current status, and so on. Then, for example, the file `Read()` function can use this information to determine exactly which storage device, and block number on the device, that it should be reading in order to return byte values to the calling application program.

6.2.1 File Systems and External File Descriptors

A **file system** is a unified collection of files on a storage device. When a storage device is formatted, areas of the device are reserved for particular purposes – generally for on-device descriptions of each file is stored in the file system (which we will refer to as **external file descriptors** hereafter), and for the files themselves. The external file descriptor definition will influence (and be influenced by) the design of the file system. Mainstream operating systems often allow multiple file systems to exist on a single storage device (such as a hard disk device). The disk is partitioned into multiple logical disks, and then each logical disk can have its own file system. Removable media (such as floppy disks, CD-ROMs, and flash drives) are generally formatted with their own file system. Each file system can be independently **mounted** or dismounted (logically attached so that the OS can manage the file system or disconnected from it). As suggested by Figure 6-5, there is a fixed area on the storage device where external file descriptors are stored. Whenever a file is added to the file system, an external file descriptor is created to describe the properties of the new file, for example, its name, owner, length, and so on. The external file descriptor also specifies the blocks on the storage device that are used to save the byte stream on the storage device. (In the simplified picture, it appears as though files are stored in contiguous blocks, although that is not normally the case as you will see in Section 0.)

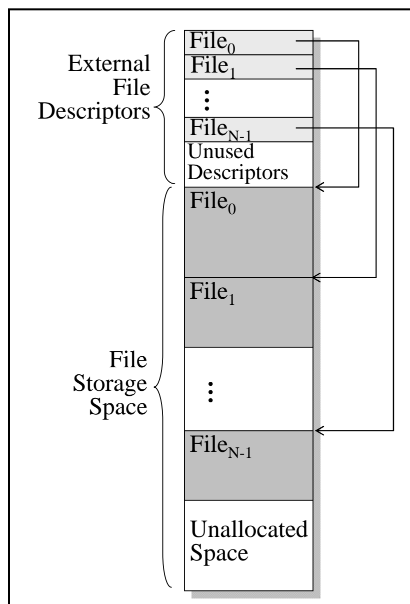

Figure 6-5: File System on a Storage Device

Creative Commons Attribution-Share Alike 3.0 U.S. License

To give you the flavor of an external file descriptor, Figure 6-6 shows the fields in the, now ancient, Version 6 Unix external file descriptor – called an **inode** (this version of Unix was the first publicly available description of the Unix code[2]). The `i_uid` and `i_gid` fields identify the owner user ID and owner group ID, respectively. The i_size0 and i_size1 fields specify the file length. The `i_atime[2]` is the time that the file was last accessed (in Unix standard time format, which is two integers representing the time since January 1, 1970 – the first integer is the number of seconds since "the beginning of time," and the second integer is the number of microseconds since the last second changed. Similarly, `i_mtime[2]` is the time that the file was last modified. Notice that these fields all describe the static properties of the file when it is stored on the device. Today, the Linux Version 2.6.23 external file descriptor has more than 35 fields, many of which are themselves structs. The exact number of fields in the descriptor depends on several compile time variables, but the fields in the old Version 6 Unix generally still exist, albeit in different formats.[3]

```
/*
 * Inode structure as it appears on
 * the disk…
 */
struct        inode
{
    …
    char      i_uid;
    char      i_gid;
    char      i_size0;
    char      *i_size1;
    int       i_addr[8];
    int       i_atime[2];
    int       i_mtime[2];
};
```

Figure 6-6: Example Fields in the Linux External File Descriptor

Observe that each the *type* of a file system is reflected by the details of how external file descriptors are defined, how they are saved on the storage device, how blocks are associated with a file, and a myriad of other details. In general, we would not expect any particular OS to be able to read and write files, say on removable storage like a floppy diskette, that had been created by a different OS. We will return to discuss this aspect of file manager design in Section 6.4. For now, let's assume that the file manager defines a file system layout and an external file descriptor that can be used with the given file system.

[2] This is from John Lion's description of Version 6 Unix [Lions, 1977].
[3] See the file named `include/linux/fs.h` in the Linux source code, e.g., at http://lxr.linux.no/.

Creative Commons Attribution-Share Alike 3.0 U.S. License

6.2.2 Internal File Descriptors

A **file session** refers to the collective interactions between an AUC and a file; the session begins when the AUCH makes an `Open()` system call and terminates with the `Close()` system call. On initiation, the file manager will need the information that is in the external file descriptor along with additional information that describes the file session activity. For example, an open file has states that are different from closed files, each session that references a file has a file pointer that references the "current" byte in the byte stream, the OS should know the user who is responsible for the AUC that is accessing the file, and so on. Therefore, when a file is opened the file manager creates an **internal file descriptor** to keep track of the file session. The file manager first reads the external file descriptor from the file systems and then it makes a copy of the information from the external file descriptor in the internal file descriptor (see Figure 6-7). It also adds session-specific dynamic information to other fields in the internal file descriptor, for example the file pointer, the user that is currently accessing the file, the mode of access (such as read-only, write-only, or read and write), and so on.

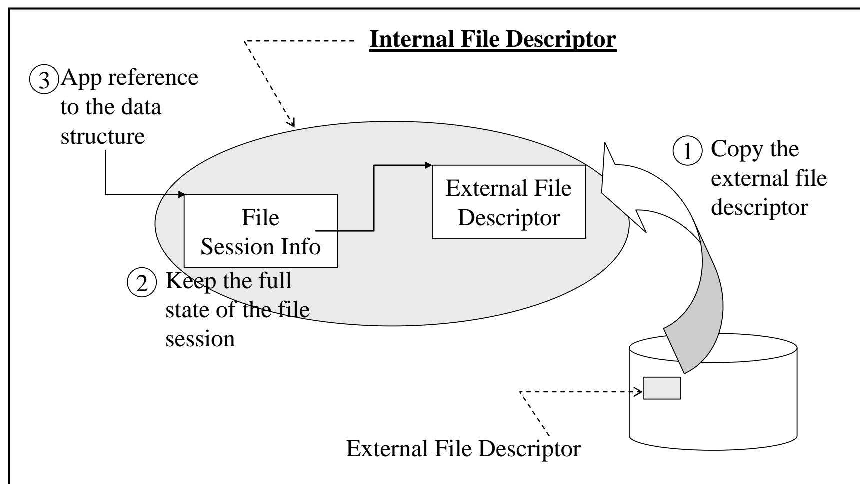

Figure 6-7: Building the Internal File Descriptor

During the time that the file is open, the static characteristics of the file could change, for example its length would change if it data were to be added to the file with a `Write()` function. Since device reads and writes are so expensive (compared to executable memory operations), the external file descriptor is not updated during the file session. Instead, the copy of the external file descriptor information that is stored in the internal file descriptor is updated, but the original version of the external file descriptor (residing on the storage device) will become outdated during the file session. When the file manager closes the file (ends the file session), the external file descriptor on the storage device is updated from the copy in the internal file descriptor (then the internal file descriptor is destroyed).

Creative Commons Attribution-Share Alike 3.0 U.S. License

When the file is opened, the internal file descriptor is built using the information in the external file description along with information that is specific to the particular file I/O session. For example, in a Unix style system, the `open()` system call sets up entries in three different OS tables (see Figure 6-8). Unix keeps a File Descriptor Table for each AUC (a process in Unix). The `open()` call allocates an entry in this table, returning the index of the entry as a result of the call. Recall that in Unix each process is initialized with the first three such indices for the File Descriptor Table set to reference the standard input pseudo file (stream `STDIN = 0`), the standard output (`STDOUT = 1`), and the standard error stream (`STDERR = 2`). When the kth file is opened, the file manager also creates an entry in the File Structure Table in which it will store dynamic information about the file session; the File Descriptor Table entry references this new File Structure Table entry. The open() call fetches a copy of the external file descriptor from the storage device, allocates a new entry in the inode Table, and then copies the external file descriptor into the new entry. The File Structure Table entry is then modified to reference this copy of the inode. Once the file has been opened, the other file functions use the internal file descriptor to adjust the file pointer, to find information in the file, to allocate/deallocate blocks to/from the file, and so on.

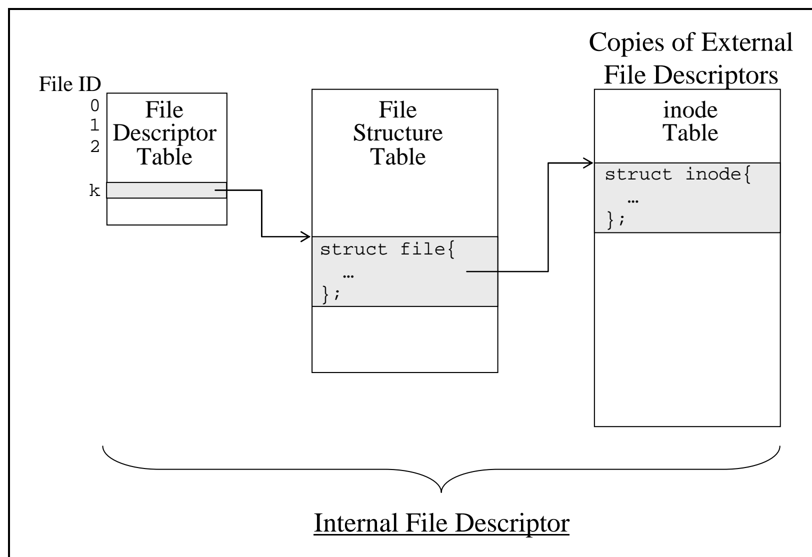

Figure 6-8: Unix Style Internal File Descriptors

6.2.3 The `Read()` and `Write()` Operations

File `Read()` and `Write()` enable an application program to read or write a collection of *k* bytes from/to the designated file at its current file position. The first argument of these two system calls is the **file handle** (for Unix, this is just the index into the calling process' File Descriptor Table; in Windows it is an abstract address that refers to Windows' equivalent of the File Descriptor Table). Both `Read()` and `Write()` use the file handle to find the internal file descriptor that provides the details of the file contents. For example, these access functions use the file pointer that is stored in the internal file descriptor to determine which byte(s) to read or write, and the external file descriptor copy in the internal file descriptor to determine which disk address it should use to read or write the next device block that contains the target byte(s).

Creative Commons Attribution-Share Alike 3.0 U.S. License

Marshalling and Unmarshalling

Let's first focus on the way `Read()` behaves: First, recall that the file is logically organized as a byte stream, b_0, b_1, b_2,... where bytes b_0 to b_{k-1} are stored in logical block B_0; bytes b_k to b_{2k-1} are stored in block B_1, and so on. After the file has been opened, the file pointer is set to zero, meaning that the next (first) read operation will begin reading the byte stream at $byte_0$. Thus there are two stages to a read operation:

1. The file manager must determine *which* device block contains the targeted bytes, and then *make a copy* of the block in the memory (that is, it must read the target block from the storage device)
2. The file manager must extract the targeted bytes from the in-memory copy of the device block and pass this information back to the calling AUC (this is called **unmarshalling** the data).

For example, if the first read call is `Read(devID, dataBuff, 5)`, the file manager must first read a copy of B_0 into the memory (Step 1), and then copy bytes b_0 to b_4 into the `dataBuff` array (Step 2).

The application programmer need never know which block is being read, or even the number of bytes in a block. When the application program attempts to read byte b_k, the file will determine that the next targeted byte is in block B_1. Therefore it uses the information in the external file descriptor to determine the block address of the B_1, and then issues a read command to the storage device for that block (see Figure 6-9).

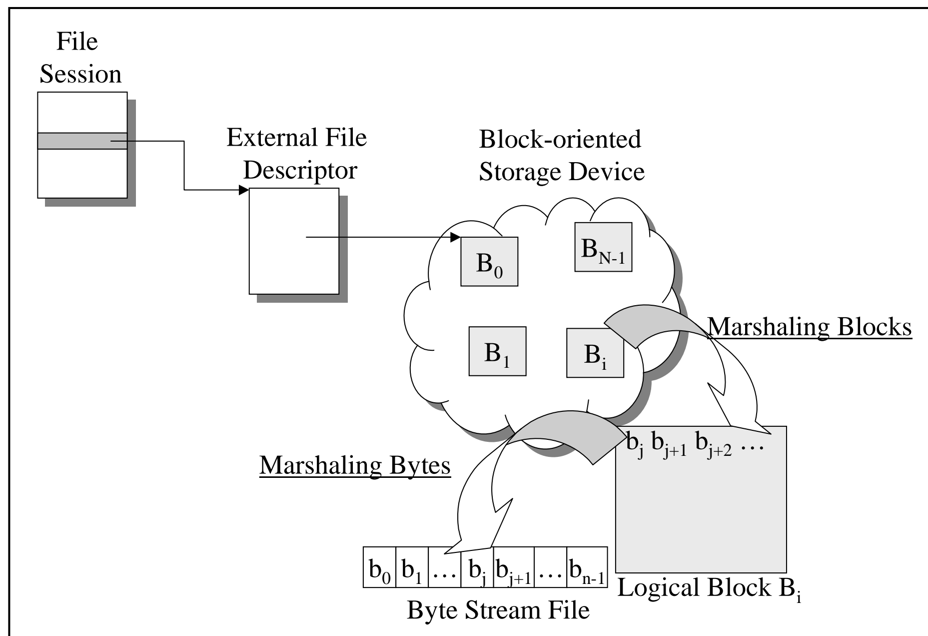

Figure 6-9: Marshalling Blocks and Bytes

The `write()` operation works in the opposite order. That is, the application fills a buffer in the file manager in anticipation of writing the block to the storage device after it is full. The action of converting a byte stream into a block sequence is known as **marshalling** the byte stream. When the file is opened for writing, the first block of the file is copied into memory so that the string will be overwritten on the existing data. When the write operations cause the file position to move from the first block into the second, the first block can be written back to the storage device and the second block is copied into memory to be altered by output operations.

Creative Commons Attribution-Share Alike 3.0 U.S. License

File seek operations potentially cause a series of storage device block read operations. If the application calls for a seek to file position *i*, then the file manager must determine which logical block, *j*, contains position *i*. Assume all blocks contain the same number, *k*, of bytes (except the last block in the file). The simplest implementation is to determine *j* by computing $\lfloor i/k \rfloor$. The file manager then can read the jth block in preparation for subsequent I/O commands. Notice that "reading the jth block" only requires one device read operation if the block manager uses index tables, but can still require several reads if the block management strategy uses a linked list; we will consider this in some detail in the next section.

Buffering

Notice that in reading bytes from a file (Figure 6-9) the file manager *could* choose to load two blocks at a time, three blocks, or even the entire file into the executable memory. A similar decision to write more than one block at a time could be made during `Write()` operations. Reading or writing multiple blocks, known as file buffering, can be useful in situations in which it is possible to overlap the device I/O with the processor operation. The benefit arises in that read operations can perform block input before the application software on the processor issue a `Read()` call. If the file manager is successful at doing this, then the application program never has to wait for a device input operation, since it was done earlier in anticipation of the Read() call.

A similar situation can occur on output. However, instead of *reading ahead* to load blocks of data into the memory before the actual `Read()` calls for the data, the file manager *writes behind* the actual Write() calls. That is, if the file manager fills up a logical block in the memory (as the result of Write() calls), rather than blocking the application to wait for the write to complete, the file manager returns control to the application program immediately, and then overlaps the block write operation with subsequent application activity.

File buffering is not an intellectually difficult concept, but it is one of the most effective techniques for overlapping processor and device operation. This means that the file manager is likely to make heavy use of the storage device's nonblocking `Read()` and `Write()` functions. Notice that application level polling will play an important role in effective buffering in an ST systems, since the devices do not issue interrupts when they complete an operation.

6.3 Block Management

Block-oriented devices are typically designed to enable software to access the blocks in any order they wish. This is contrasted with sequentially-accessed devices such as various forms of tape devices (and even CD-ROMs). In sequentially-accessed devices the physical records are conceptually arranged as an array of records. Record number i can only be accessed after first accessing record number i-1. Historically, sequentially-accessed devices were a mainstream storage device, but in contemporary systems, particularly SCCs, there is no sequential constraint on access – such devices are sometimes called **direct** or **random access devices**. Disk drives are the most well-known of this class of devices.

File systems build a file by writing information to multiple blocks on a device. In sequentially-accessed devices, the logically adjacent blocks are also physically adjacent: for example, logical record 3 might be stored in physical block 107, logical record 4 in physical block 108, and so on. In a device that supports random access, logical record 3 might be stored in physical block 107, but logical record 4 could be stored in any arbitrary physical block such as number 882.

Block management is the logical part of the file manager that keeps track of the set of blocks that are used (from a random block access device) with a file, including the order in which they are used. Conceptually, the block management map is a table of the form shown in Figure 6-10. That is, the map indicates the physical block address for each logical record/block used for file storage. Notice that when a block gets added to a file, then a new row is added to the table; and when a block is removed, a row is removed from the table. While it would be possible to implement the map as a table as shown in the figure, operating systems tend to use alternative data structures that are more flexible with respect to the amount of space that must be reserved for the map. The general approaches are to store the information in the table in a list or some form of index.

Creative Commons Attribution-Share Alike 3.0 U.S. License

Logical Block Number	Physical Device Block
0	
1	
…	
N-1	

Figure 6-10: Mapping Logical to Physical Blocks

File managers generally make an assumption that the data in the file's byte stream is packed as tightly as possible in the storage blocks. For example, if blocks contain 512 bytes, then the first 512 bytes are assured of being stored in the first block; the second 512 bytes are in the second block, and so on. This means that the file manager can easily determine a logical block number, j, by taking the integer resulting from dividing the byte number by the block size, that is,

$$\text{block number} = \lfloor \text{byte address/block size} \rfloor$$

Since there may be a remainder from this division, the file manager will generally use $\lceil$file length/block size$\rceil$ blocks on the device to store the file.

The file descriptor storage details will be designed to implement different data structures for each of the block allocation strategies.

6.3.1 List-oriented Block Management

Linked lists of blocks use explicit pointers among an arbitrary set of physical blocks making up the file. Logical block $i + 1$ need not be physically located near logical block i, since i will contain a header with a link that addresses the physical block that contains logical block $i + 1$. The internal file descriptor for a file will include a pointer to the first device block used in the file and a copy of the file position pointer (see Figure 6-11). The entry may also contain other data used to manage the open file.

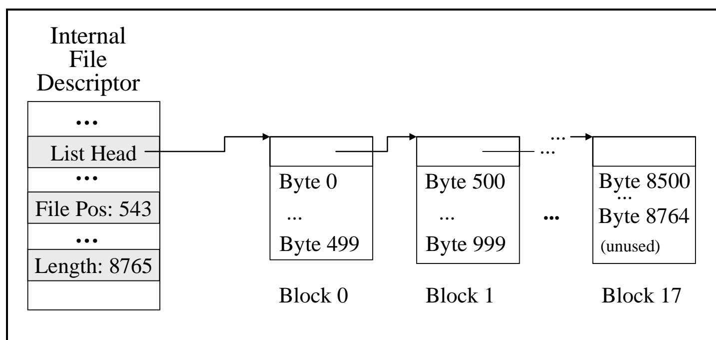

Figure 6-11: File Status Entry for Linked Lists

Each block in the file contains overhead information used by the file manager. In the example, 12 bytes are reserved for this information, thereby allowing the block to be written to a storage device that supports 512-byte physical blocks. The unused bytes in the Block 17 are said to have been lost to **internal fragmentation**, meaning that when the file is stored in multiple blocks, the last block will usually have some unused bytes (unless the file is a multiple of the effective block length).

Creative Commons Attribution-Share Alike 3.0 U.S. License

Linked list implementations are especially good in cases where the size of files is not known ahead of time, and where some files may be large and some small. If a table were used, then each external file descriptor would need to reserve enough space to hold the logical-physical block map for the largest file, meaning that small files would waste a lot of space. With a linked list implementation, the space to store the map exactly fits the number of blocks used in the file.

On the negative side of things, if application software tends to randomly access the bytes in the stream (by using the Seek() system call), access will be slow, particularly as the file size grows. Each file `Seek()` will be costly in this block allocation scheme, since it requires list traversal. This in turn requires each block in the list to be read from the device so that the link field can be obtained and the next block can be referenced. Doubly linked lists can be used to enhance performance during `Seek()` operations. When a `Seek()` is issued, the file manager calculates whether to move forward on the list, move backward on the list, or go to the front or back of the list to begin searching for the target block. Even with doubly linked lists, `Seek()` can be slow.

6.3.2 Index-oriented Block Management

Because the linked-list strategy can cause seek operations to be read-intensive, the block management data structure can be changed to be more efficient for random access, that is to avoid list traversal. Suppose that the link field (and other related information) is removed from data blocks, and placed into an index table (see Figure 6-12). Next, the overall table can be split up and stored in disk blocks: small files will use only a single disk block to store the index, but large files can use as many disk blocks as they need to store the entire index.

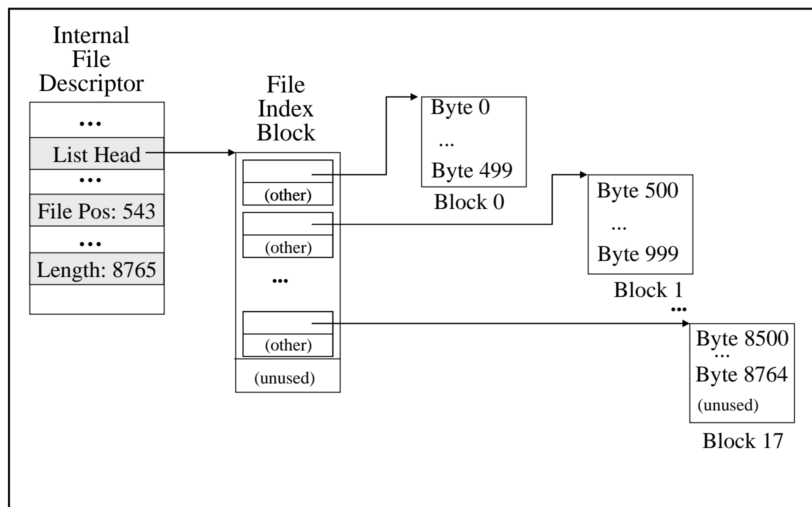

Figure 6-12: File Status Entry for Indexed Allocation

As is the case with linked lists, there is storage loss due to fragmentation. In addition to the internal fragmentation in the last block, there is loss in the last (or only) block that stores the File index. In the usual case, the index table will not completely fill the file index block, thereby introducing storage loss due to **table fragmentation**. You initial reaction is probably something like "that can't be very much space…" but in cases where the block size is large, the table fragmentation can be significant (but not more than one block in the informal design described here).

Creative Commons Attribution-Share Alike 3.0 U.S. License

6.3.3 Keeping Track of Unallocated Blocks

When the file system is created by formatting, none of the blocks belong to any file, that is they are all unallocated. The block manager part of the file manager must find some way to keep track of these unallocated blocks. If the allocated blocks are managed using a list data structure, it is natural to consider managing the unallocated blocks as a list data structure, called a **free list**. Note that an empty disk has a very large list that will take a long time to traverse should that be necessary. Remember that to traverse from one block to the next it will be necessary to issue a device block read to chase each link in the list. For a 500 GB disk, this can take many minutes, if not hours.

Normally, any block (on a randomly accessed device) can be allocated to a file, since there is no dependence on sequential blocks. Therefore, the block manager could, say, allocate blocks from the front of the list and place deallocated blocks at the tail of the list. Initially, blocks with consecutive addresses will be allocated consecutively. If a file manager requests multiple disks at a time, it will receive contiguously addressed blocks … until the list has been cycled once and the block manager begins to allocate blocks that had previously been allocated then returned to the free list. Contiguous block allocations will likely not have related block addresses.

Disk I/O times are dominated by **disk seek time**, meaning the time to align the disk's read-write head over the target track on the disk surface. If a block manager could easily allocate blocks to the same file so that adjacent logical blocks were on tracks that were relatively close to one another, then accessing the file would take far less time than if the blocks were on distant tracks. However, the block manager will be unable to perform this optimization without traversing the entire free list – a very expensive operation. (Of course the free list could be kept as a sorted list, but then block deallocation would be very expensive.)

It is common for block managers to keep a **bitmap** to represent the availability of disk blocks. The general idea is that an array of bytes is allocated as part of the block manager's information. The block is periodically written to the storage device in order to ensure persistence. Each bit in a byte represents the state of a corresponding device block. Suppose the least significant bit in the byte[0] of the bitmap array represents the state of the physical device block with address 0, the second least significant bit in byte[0] represents the state of $block_1$, and so on. Whenever the file system is created, the entire bitmap is set to zero, that is every bit in every byte is zero. Now whenever a block is allocated to any file, the corresponding bit in the bitmap is set to one. The block manager can now consult the bitmap to see which blocks close to some target block are available.

6.4 Virtual File System Architectures

It is apparent that different file systems have different external file descriptors, and hence different code to manipulate the file descriptors, and likely have different block management strategies. However, it is entirely conceivable that all of the file system could export the same OS system call interface for different file systems using a refinement of the device management strategy for exporting the same interface to a spectrum of different devices (see Section 5.3). The virtual file system architecture is used to accomplish this for disparate file systems.

The technique for the virtual file system switch came from work done many years ago on remote file systems (more on the reasons for this in the next section).[4] The core idea is to split the file manager into two parts (see Figure 6-13): a part that encapsulates the details of a particular file system (the file system dependent part), and a part that implements the aspects of the file manager that are common to all file systems (the file system independent part). The file system independent part focuses on implementing the OS system call interface, and on implementing a virtual (or abstract) file manager that relies on its own virtual external file descriptor, popularly called **vnodes** in the Unix world.[5] This virtual file manager does not manipulate real devices, but instead manipulates virtual devices with specific file formats that are implemented by the collective file system dependent parts.

[4] I first heard about this design in a technical product announcement for Systems V Unix in about 1983. See [Rifkin, 1986] for a description of the System V design.
[5] The vnode work appeared in an early version of Sun Solaris at about the same time as the System V technique was being developed, see [Kleiman, 1986].

Creative Commons Attribution-Share Alike 3.0 U.S. License

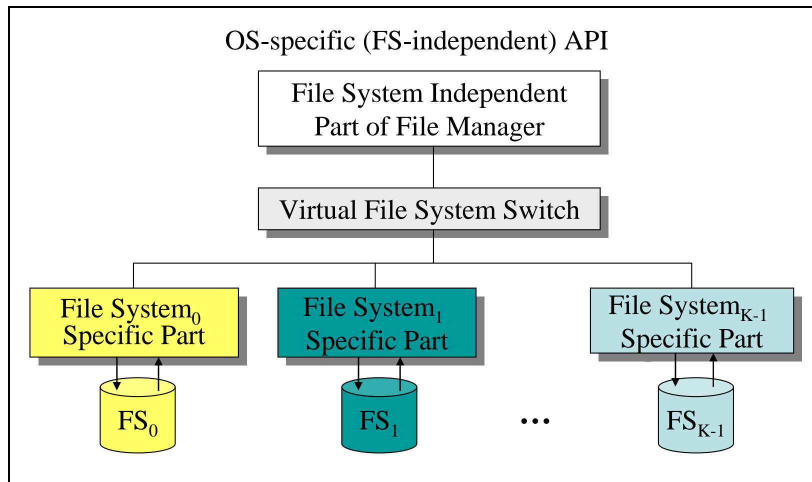

Figure 6-13: Virtual File Systems

The design specifies a standard interface between the file system independent part and any file system specific module. Generally, the file system independent part determines the type of the file system for a particular device, then calls the associated file system specific part to manage the actual on-device information such as the external file descriptor and the file contents. That is, the highest level abstraction is the file system independent file manager; it uses a file system specific adaptor that encapsulates the details for a particular file system format; the file system specific adaptor uses the device driver abstraction to read and write information from the physical device. Whew, that's a lot of abstraction!

There are three primary times when the two parts of the file manager interact:

- **Mount time**. The file system can be mounted at the time the system is booted (the usual case for permanently attached devices), or when a device is dynamically attached to the system (usually as some form of removable media). The file system independent part collects file system specific general parameters (stored in the file system's **superblock**); these parameters include items such as the number of bytes per block on the storage device. The file system specific module can either directly provide superblock parameters, or if they are stored on the device as part of the formatting operation, the module reads the information from the device's equivalent of a superblock. The file system independent module then saves this information in its virtual form.
- **Open time**. When a file is opened, the file system independent module builds its virtual internal file descriptor (analogous to a Unix vnode) by asking the file system specific module to provide the external file descriptor information. In most cases, the module just reads this data from the device's external file descriptor.
- **Read/write time**. The file system independent module exports the normal file system read and write system calls to the application program. These implementations use the virtual internal file descriptor to read and write in-memory buffers. When it is necessary to read/write device blocks, file system specific read/write functions are called to handle the device operation.

The file system switch architecture is widely used in small, medium, and large operating systems. In a distributed environment, this is very important, since there can be many different kinds of file systems on the entire distributed system.

6.5 Files in a Distributed System

Small, communicating computers are intended to execute in a distributed computing environment. ST machines are the smallest (at least the least functional) of the SCCs, so they are not likely to have very much memory in their storage devices. Instead, they rely on **caching** (making a temporary copy of) information from larger machines in the distributed system on an as-needed basis. Once the SCC has

Creative Commons Attribution-Share Alike 3.0 U.S. License

completed its processing of a file, it will generally write the file back to the larger machine for longer term storage or archiving. Notice that if the SCC file manager uses the virtual file system switch architecture described in the previous section, then it can read and write a variety of file system formats that may be stored on different remote machines in the distributed system.

This implies that the ST SCC will be much more useful if it has some form of file manager that enables it to cache files onto its own storage devices. Once the files have been loaded onto the SCC, the ST file manager (described in the previous sections) will enable the application to read and write those files. File caching can be manual or automatic. Manual caching just means that the SCC issues a message to the large machine that requests it to sent a copy of a particular file in response to the message. This is the technique used in the HTTP (web browsing) and FTP protocols.

Automatic caching is implemented by creating software on both the client SCC and the large machine file server that is designed to enable the SCC application to use files on the large machine as if they were stored on the SCC's storage devices. Based on what you have learned about file management earlier in this chapter, you could infer that the SCC must be able to mount a file system that is physically stored on the file server machine: this is known as a **remote mount** operation. Notice that the virtual file switch architecture has been designed to incorporate mounting: it can be enhanced so that there is a file system dependent part for remote file systems (this is the reason that the early Unix developers designed the virtual file switch – it was to handle all three facets (mount, open, and read/write) on files that were stored on a file server, but which could be accessed from a remote client machine.

The remote mount operation enables the file manager to obtain the information it needs in order to retrieve the general information it needs about a file system when the file system is located on a remote machine. After a file system has been remote mounted, the SCC application can use an `Open()` system call to begin reading and writing the file that is located on the file server.

There are two general strategies for opening a remote command: (1) the entire file may be copied from the file server onto the client SCC, or (2) the file server may send blocks from a file to the client SCC whenever its file manager intends to load a local SCC buffer. The Andrew File System (AFS) uses the first approach,[6] and the Sun Network File System (NFS) uses the second approach.[7]

6.6 Directories

Contemporary computers incorporate hierarchical directories into the file manager. In graphic user interfaces, a directory is typically represented as a "folder," while a file is a "document." Each directory (folder) can contain multiple files (documents). Further, a directory can also contain other directories (informally called sub directories). Pictorially, the collection of files in directories is often represented as a tree structure with the root of the tree at the top instead of the bottom (see Figure 6-14). In this informal diagram, directories are strings that begin with an upper case letter, and files are strings that begin with a lower case letter. The "(Root)" directory is unique in that it has no predecessor, whereas all other files and directories in the hierarchy have exactly one predecessor. A File has no descendant, and a directory can have zero or more descendants which are either files or directories.

In the file hierarchy, every file or directory has a unique pathname from the root to the target file or directory. In POSIX style systems, the root directory has the name "/" (in Windows systems, it is named "\"). In the example, the unique pathname from the root to the "todo" directory is "`/Home/Jabba/todo`". Similarly, the pathname to the "CS3753" directory is "`/Home/Jabba/Public_html/CS3753`" often written with a trailing "/" to show that it is a directory rather than a file ("`/Home/Jabba/Public_html/CS3753/`").

---

[6] AFS was developed as part of the Andrew distributed system, and subsequently became open software that can be used with Unix systems [Howard, et al., 1988].

[7] NFS was developed as part of Sun's operating system product, and is probably the most widely used network file system [Sandberg, et al., 1985].

Creative Commons Attribution-Share Alike 3.0 U.S. License

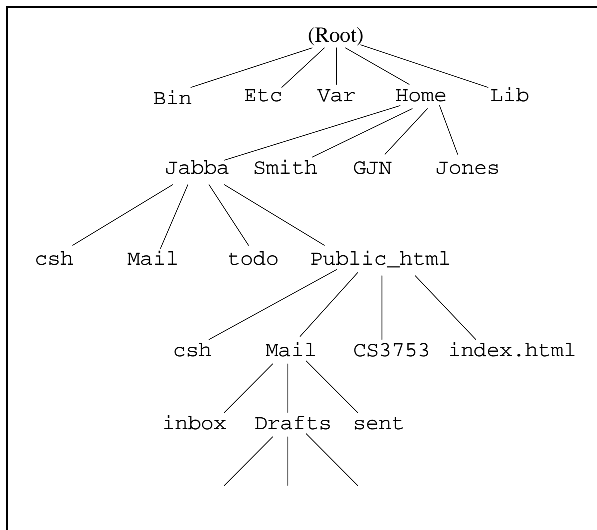

Figure 6-14: Files and Directories

Recall that the mount operation is used to logically attach a file system to a computer. Just as the computer has a single root directory, *every file system also has a unique root directory*. The POSIX mount command attaches a file system to a leaf node in the computer's file system, for example, it we had a file system we could mount it at "`/Home/Jones/`". Then a pathname of the form "`/Home/Jones/Blather/…`" would lead to a file in the `Blather` sub directory in the mounted file system. Now we can appreciate the beauty of the remote mount for remote file systems. If the file system mounted on "`/HOME/Jones/`" was on a remote machine, then a network file system would mount the remote file system at the Jones mount point, thereby making all files in the remote file system have pathnames on the local machine.

Despite the obvious importance of directories, it is interesting to note that in modern operating systems, they are usually not implemented as OS software (they are system software, but not part of the OS). In an ST OS, the may be no significant distinction between OS and other system software, so we will provide a brief description of directory implementations.

A directory is typically implemented using an ordinary file. The contents of the file are a list of directory entries, each of which references the inode for the corresponding file or directory. For example, the Linux Version 2.6.23 directory entry is defined as

```
struct dirent {
        long            d_ino;
        __kernel_off_t  d_off;
        unsigned short  d_reclen;
        char            d_name[256];
};
```

Where the name of the file is given as a string in the `d_name[]` array and the `d_ino` field references the inode (in the inode table) for the designated file or directory. In the case of Unix style systems, essentially all of the information about a file is kept in its file descriptors rather than in its parent directory entry. Microsoft DOS (or "FAT") files are 32-byte entries that more information for each file in the directory than does the Unix style directory entry.

Creative Commons Attribution-Share Alike 3.0 U.S. License

6.7 Summary

Files are the main means of managing and storing large amounts of data on a computer system. ST OS files are usually the simplest form of file, namely the byte stream file. Larger machine operating systems sometimes include support for files that contain structured records, but that would be overkill in a ST SCC.

The file manager implements files and directories. Each file is represented by file descriptors – an external file descriptor for the static structural description of the file (kept on the storage device with the file), and the internal file descriptor that incorporates the information in the external file descriptor as well as information that describes a file session (for an open file).

Besides implementing the open operation to prepare a file for use, the file manager implements information marshalling and unmarshalling in the write and read routines, respectively. Generally, the file manager incorporates a buffering strategy in order to read blocks from the storage device before they are needed by the application program, and to write blocks back to the storage device after they have been filled (again without making the application program wait for the write operation to complete).

The file manager block management algorithms handle the means managing unused blocks of storage, and for allocating blocks to files when they are needed or returning unused blocks to the system-wide free list.

Since the 1990 timeframe, operating systems have made wide use of the virtual file switch architecture. This divides the file manager into file system dependent and independent parts. The advantage of doing this is that it makes it possible for a single OS to be able to read and write files written by a spectrum of other operating systems.

SCCs exist as components of distributed systems. Since they are likely to be limited in the amount of persistent storage space they can have, they can make good use of remote file systems. This enables the SCC to access a large number of files that are stored on a file server without having to keep local copies of each file.

Directory management programs are middleware system software that use the basic byte stream file as a container for information that is used to organize the system's files into a hierarchy. The hierarchy also accommodates remote mount operations to simplify remote file reference and use.

There are *many* details in a working file manager. The combination of the device manager and file manager are a major portion of the code that makes up an OS, particularly an ST OS. You can learn more about these details by studying the examples in the supplemental materials you are using with this book.

6.8 References

1. Howard, John, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West, "Scale and Performance in a Distributed File System," ACM Transactions on Computer Systems, 6, 1 (February 1988), pp. 51-81.
2. Kleiman, Steven R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Summer Conference Proceedings*, June 1986, pp. 238-247.
3. Lions, John, *Lions' Commentary on UNIX 6th Edition with Source Code*, ed. By Peter Salus, Peer-to-Peer Communications, 1996.
4. Rifkin, Andrew P. et al., "RFS Architecture Overview," *USENIX Summer Conference Proceedings*, June 1986, pp. 248-259.
5. Sandberg, R., D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network File System," *USENIX Proceedings*, June 1985, pp. 119-130.
6. Ritchie, D., and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM*, 7, 2 (July 1974), pp. 1897-1920.

Creative Commons Attribution-Share Alike 3.0 U.S. License