17

File Systems

This chapter discusses the purpose of file systems and the type of objects that can be kept in such systems, as well as the details of the software that manages data files on a local disk.

17.1 What Is A File System?

A *file system* consists of software that manages permanent data objects, objects whose values persist longer than the processes that create and use them. Permanent data is kept in *files* on secondary storage devices like disks. These files are organized into *directories*. Conceptually, each file consists of a sequence of data objects (e.g., a sequence of integers). The file system provides operations that *create* or *delete* a file, *open* a file given its name, *read* the next object from an open file, *write* an object onto an open file, or *close* a file. If the file system allows random access, it may also provide a way to *seek* to a specified location in a file.

File system software does much more than manipulate individual files on secondary storage – it provides an abstract name space and high-level operations to manipulate objects in that space. The abstract space consists of the set of valid file names. It can be as simple as, ‘‘the set of strings formed from at least one, but fewer than nine, alphabetic characters,’’ or as complex as, ‘‘the set of strings that form a valid encoding of the network, machine, user, subdirectory, and file identifiers.’’ In some systems, the syntax of names in the abstract space conveys information about their type (e.g., text files end in ‘‘.TXT’’). In others, names give information about the organization of the file system (e.g., file names that begin with ‘‘M1_d0:’’ reside on disk 0 of machine 1).

Names in the abstract space need not correspond to conventional files on secondary storage. They may refer to devices, services that the system supplies, or files that reside on another machine. For example, the name *console* may correspond to the *CONSOLE* device; the name *printer* may correspond to a service that prints a copy of data written to it; and the name *foo:bar* may correspond to disk file *bar* accessed through network *foo*.

Allowing names to refer to devices and services is convenient because it permits programs to perform a variety of useful functions that depend only on the file names accessed. Consider, for example, a general-purpose utility program, *CP*, that reads data from one file and writes it to another (i.e. copies the contents of one file to another). If the abstract space contains names of devices and remote files, *CP* can be used: to print the contents of a file on the console terminal; to accept input from the console and write it on a disk file; to copy a file from a remote machine to the local one; or to copy the contents of one disk file to another.

What high-level operations should the file system support to make it possible to write programs like *CP*? The answer depends largely on the type of objects that the name space includes and the structure of data files on the disk. If the file name space includes devices, the set of operations that programs perform on files must map into the set of operations they can perform on devices. If the file system treats data files as a sequence of bytes, operations that transfer bytes may suffice; but if data files have more structure, operations that transfer records may be required.

Our choice of high-level operations is motivated by a desire to make devices and files compatible and to keep the software small. It uses the following principle:

> *The file system considers each object to be a sequence of zero or more bytes; any further structure must be enforced by user-level programs.*

Treating files as streams of bytes makes the file system primitives easier to implement and remember, keeps them applicable to devices and services as well as conventional files, and allows programs to impose whatever structure they desire on the file. For example, if integers are two bytes long, programs can create a file of integers by always transferring two bytes at a time.

Having decided to treat files as streams of bytes, we are ready to design a set of high-level operations for files. Our system will use exactly the same high-level operations that were used for devices. Thus, the file system will support *read*, *write*, *putc*, *getc*, *init*, *open*, *close*, *seek*, and *control* primitives. The semantics of these operations will depend on the type of the file, just as their interpretation depended on the type of device. (To be honest, we should admit that the set of device-independent operations were chosen with both devices and files in mind.)

When applied to conventional disk files, the high-level operations produce the following effects. *Open*ing a named file connects an executing process with the data on disk and establishes a pointer to the first byte. Operations *getc* and *read* retrieve bytes of data from the file and advance the pointer. Operations *putc* and *write* change bytes in the file and move the pointer along, extending the file if new data is written beyond the end. The *seek* operation moves the pointer to a specified position in the file. Finally, *close* detaches the running process from the disk file.

17.2 Disk And File Servers

Machines that connect to a network can have a file system even if they do not have a local secondary storage device. File systems on diskless machines pass requests across the network to a *server* machine. The server contains special-purpose software that interprets the requests and sends data back as needed. The server may provide a *pseudo-disk* for each workstation, leaving most of the work of managing files and directories to the individual machine, or it may perform most of the file system services itself, relieving the individual workstations of that responsibility. Whether the server simulates a disk (i.e., provides a large array of disk blocks that must be accessed by number), or a complete file system (i.e., provides *read* and *write* operations on named files) is determined largely by the hardware environment and expected use of the system. The latter style is more popular because it means each individual machine needs less software.

Neither disk servers or file servers are difficult to implement given reliable network software and the software to manage local disk files. The questions of how to name and address the server, where to store directories, and what pieces of the file system name space are shared among all users are the most difficult to answer because they depend on the hardware configuration and intended use of the system. Rather than tackle the issues of how to assign names or how to access files over a network, we will turn our attention to the more fundamental piece of the file system, the efficient, general-purpose access mechanism that manages information on a local disk.

17.3 A Local File System

Files are called *local* to a given machine if they reside on a disk that is connected to the machine. The design of software that manages such files is nontrivial; it has been the subject of much research. Local file software must support the high-level operations as defined above: *read*, *write*, *putc*, *getc*, *seek*, *open*, and *close*. Although these operations seem simple, complexity arises from the details of buffer and index management and concurrency control.

To what extent should the system support concurrent operations? Large systems usually allow arbitrary numbers of processes to read and write arbitrary numbers of files concurrently. The chief difficulty with multiple access lies in specifying exactly what it means to have multiple processes interacting on a single file. When will readers be able to access changed portions? If two processes attempt to write to the same disk block, which will be accepted? How can a process lock pieces of a file to avoid interference?

The generality of allowing multiple processes to read and write a file is usually not necessary on small systems. Thus, to limit the software complexity and make better use of disk space, small systems constrain the ways in which files can be accessed. Instead of allowing files to grow incrementally as needed, they may require the user to preallocate space for the file. They may also limit the number of files that a given process can access simultaneously, or the number of processes that can simultaneously access a given file.

Our goal is to design file system software that makes it convenient to create and extend files without making the system unnecessarily large or slow. As a compromise between generality and efficiency, we will allow multiple processes to access the file system concurrently; and we will allow a single process to access an arbitrary number of files concurrently; but we will restrict access of each file to at most one process. Finally, because preallocating space makes programming difficult, we will allow files to grow dynamically. The most significant consequence of this design is that good data structures will be needed to allocate disk space and access files.

17.4 Data Structures For The File System

To support concurrent file growth and random movement, the file system allocates disk blocks dynamically and uses an *index* mechanism to locate them quickly. Our design partitions the disk into three separate areas as shown in Figure 17.1.

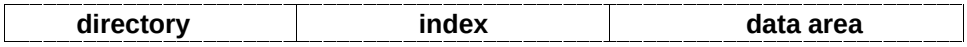

Figure 17.1 The disk partitioned into three areas

Physical disk blocks in the data area are referred to as *data blocks* because they hold all the data that has been written onto files. The file system allocates unused data blocks from a free list when they are needed and returns them to the free list when a file is deleted. The data blocks allocated to a given file contain no pointers to link them together or to relate them to the file; such information resides only in the file's index.

Separate from the data area, the index area on each disk contains a set of *index blocks* or *i-blocks*. Each file on the disk has its own index, which consists of a singly-linked list of i-blocks. Each i-block contains pointers to a set of data blocks as shown in Figure 18.2.

Figure 17.2 Part of a file's index with pointers to data blocks

Unlike data blocks, index blocks are smaller than physical disk blocks. Thus, there are several index blocks in each disk block. A layer of software handles the details of reading and writing index blocks, making it possible to think of them as randomly accessible items.

The third area of the disk holds a directory that contains pointers to the lists of free i-blocks and free disk blocks, the names of all files, and pointers to each file's index list. Each directory entry also contains an integer that gives the size of the file measured in bytes.

17.5 Implementation Of The Index Manager

Manipulation of the index area introduces a common problem with disk software. The index area contains a set of fixed-size i-blocks mapped onto a contiguous area of the disk. Because i-blocks are smaller than physical disk blocks, the system packs eight i-blocks into each physical block. The hardware transfers entire disk blocks, however, making it impossible to transfer a single i-block without transferring others that reside in the same block. So, to write an i-block, the software must read the entire physical disk block in which it resides, copy the new i-block into its correct place in the physical block, and write the resulting physical block back to disk. Similarly, reading an i-block requires the software to read the physical disk block in which it resides.

Before looking at the procedures that read and write i-blocks, we must understand a few more details. File *iblock.h* is a good place to start – it defines the contents of an i-block with structure *iblk*.

```
/* iblock.h - ibtodb, ibdisp */

typedef int             IBADDR;         /* iblocks addressed 0,1,2,...  */

#define IBLEN           29              /* # d-block ptrs in an i-block */
#define IBNULL          -1              /* null pointer in i-block list */
#define IBAREA          1               /* start of iblocks on disk     */
#define IBWDIR          TRUE            /* ibnew: write directory       */
#define IBNWDIR         FALSE           /* ibnew: don't write directory */

struct  iblk            {               /* index block layout           */
        long            ib_byte;        /* first data byte indexed by   */
                                        /*  this index block            */
        IBADDR          ib_next;        /* address of next i-block      */
        DBADDR          ib_dba[IBLEN];  /* ptrs to data blocks indexed  */
};

#define ibtodb(ib)      (((ib)>>3)+IBAREA)/* iblock to disk block addr. */
#define ibdisp(ib)      (((ib)&07)*sizeof(struct iblk))
```

Each i-block contains an array of pointers to data blocks. The array contains addresses of 29 (*IBLEN*) data blocks, each of which is 512 bytes long. Thus, a single i-block indexes 29 *X* 512 or 14,848 bytes. An i-block also contains a value that specifies which bytes of the file it indexes, and a pointer to the next i-block on the index list. Pointers to i-blocks are given by integers starting at zero.

How does the software know where to find an i-block given its address? The answer is that it must know where the index area starts on the disk and how many i-blocks are contained in each physical disk block. In our design the directory occupies disk block zero, and the index area lies just beyond, so it starts at disk block one. Thus, i-blocks zero through seven lie in physical block one. In-line procedure *ibtodb* contains code that converts an i-block address into the correct physical disk block address. Procedure *ibdisp* converts an i-block address into the byte displacement within its physical block.

17.6 Operations On I-Blocks

17.6.1 Clearing An I-Block

The file system initialization routine links all i-blocks into the free list when it builds an empty file system. As the file system allocates i-blocks from the free list, it needs to clear out old information. Clearing is performed by procedure *ibclear*. It consists of making all data block pointers null, so they cannot be confused with valid pointers, and setting the offset field to the value specified. In the code, shown below, file *iblock.h* is not included explicitly. As we will see later, file *file.h* includes it.

```
/* ibclear.c - ibclear */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  ibclear  --  clear in-core copy of an iblock
 *------------------------------------------------------------------------
 */
ibclear(ibptr, ibbyte)
struct  iblk    *ibptr;
long    ibbyte;
{
        int     i;

        ibptr->ib_byte = ibbyte;
        for (i=0 ; i<IBLEN ; i++)
                ibptr->ib_dba[i] = DBNULL;
        ibptr->ib_next = IBNULL;
}
```

17.6.2 Reading An I-Block

To read an i-block, the system maps its address to a physical disk block address, reads the physical disk block, and copies the appropriate area from the physical block into the desired location. File *ibget.c* contains the code.

```
/* ibget.c - ibget */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>
#include <mark.h>
#include <bufpool.h>

/*------------------------------------------------------------------------
 *  ibget  --  get an iblock from disk given its number
 *------------------------------------------------------------------------
 */
ibget(diskdev, inum, loc)
int     diskdev;
IBADDR  inum;
struct  iblk    *loc;
{
        char    *from, *to;
        int     i;
        char    *buff;

        buff = getbuf(dskdbp);
        read(diskdev, buff, ibtodb(inum));
        from = buff + ibdisp(inum);
        to = (char *)loc;
        for (i=0 ; i<sizeof(struct iblk) ; i++)
                *to++ = *from++;
        freebuf(buff);
}
```

Ibget allocates space for the physical disk block from the system buffer pool, *dskdbp*. After it reads the physical block and extracts the desired i-block, it releases the storage with *freebuf*.

17.6.3 Writing An I-Block

I-blocks from several files may occupy the same physical disk block. Because processes may try to write i-blocks into the same physical disk block concurrently, writing an i-block is more complicated than reading one. To prevent interference, writing processes must obtain exclusive use of the index area. File *ibput.c* shows how the calling process waits for access using the exclusion semaphore that was created at system startup by the disk driver initialization routine.

```
/* ibput.c - ibput */

#include <conf.h>
#include <kernel.h>
#include <io.h>
#include <disk.h>
#include <file.h>
#include <mark.h>
#include <bufpool.h>

/*------------------------------------------------------------------------
 *  ibput  --  write an iblock back to disk given its number
 *------------------------------------------------------------------------
 */
ibput(diskdev, inum, loc)
int     diskdev;
IBADDR  inum;
struct  iblk    *loc;
{
        DBADDR  dba;
        char    *buff;
        char    *to, *from;
        int     i;
        int     ibsem;

        dba = ibtodb(inum);
        buff = getbuf(dskdbp);
        ibsem = ((struct dsblk *)devtab[diskdev].dvioblk)->dibsem;
        wait(ibsem);
        read(diskdev, buff, dba);
        to = buff + ibdisp(inum);
        from = (char *)loc;
        for (i=0 ; i<sizeof(struct iblk) ; i++)
                *to++ = *from++;
        write(diskdev, buff, dba);
        signal(ibsem);
        return(OK);
}
```

As expected, *ibput* allocates space for a physical disk block from the system buffer pool, reads the appropriate physical block, copies in the changed i-block, and writes the physical block back to disk. It need not free the buffer, because the driver will free it when the output operation completes.

17.6.4 Allocating I-Blocks From The Free List

The file system allocates an i-block from the free list whenever it needs one for an index, and it returns i-blocks to the free list when it deallocates a file. Procedure *ibnew* obtains the next free i-block and returns its identifier. The code is found in file *ibnew.c*.

```
/* ibnew.c - ibnew */

#include <conf.h>
#include <kernel.h>
#include <io.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  ibnew  --  allocate a new iblock from free list on disk
 *------------------------------------------------------------------------
 */
ibnew(diskdev, writedir)
int     diskdev;
Bool    writedir;
{
        struct  dir     *dirptr;
        struct  iblk    iblock;
        IBADDR  inum;
        int     i;
        int     sem;

        sem =  ((struct dsblk *)devtab[diskdev].dvioblk)->dflsem;
        dirptr = dsdirec(diskdev);
        wait(sem);
        inum = dirptr->d_filst;
        ibget(diskdev, inum, &iblock);
        dirptr->d_filst = iblock.ib_next;
        if (writedir)
                write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(sem);
        ibclear(&iblock, 0L);
        ibput(diskdev, inum, &iblock);
        return(inum);
}
```

Concurrency control complicates what is otherwise a simple procedure. *Ibnew* first ob-

tains exclusive use of the free list by waiting for the ‘‘free list’’ mutual exclusion semaphore. (Recall that the disk driver initialization routine, *dsinit*, created the semaphore and placed its id in *dflsem*.) After obtaining access, *ibnew* retrieves the id of the first free i-block from the directory. It must then update the free list by reading the first i-block and making the directory entry point to its successor. After changing the directory, *ibnew* writes a copy back to disk. Finally, *ibnew* clears the allocated i-block and writes it to disk.

17.6.5 Returning I-Blocks To The Free List

When a file is deleted, its index and data blocks must be returned to their appropriate free lists. Procedure *iblfree* performs this task. It takes an i-block address as an argument and releases the i-blocks on that list. Thus, a single call to *iblfree* will release all the space used by a file.

```
/* iblfree.c - iblfree */

#include <conf.h>
#include <kernel.h>
#include <io.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  iblfree  --  free a list of iblocks given the number of the first
 *------------------------------------------------------------------------
 */
iblfree(diskdev, iblist)
int     diskdev;
IBADDR  iblist;
{
        IBADDR  ilast;
        struct  iblk    iblock;
        struct  dir     *dirptr;
        int     sem;
        DBADDR  dba;
        int     j;

        if (iblist == IBNULL)
                return(OK);
        dirptr = dsdirec(diskdev);
        ibget(diskdev, iblist, &iblock);
        for (ilast=iblist ; iblock.ib_next!=IBNULL ;) {
                for (j=0 ; j<IBLEN ; j++)
                        if ( (dba=iblock.ib_dba[j]) != DBNULL)
                                lfsdfree(diskdev, dba);
                ilast = iblock.ib_next;
                ibget(diskdev, ilast, &iblock);
        }
        for (j=0 ; j<IBLEN ; j++)
                if ( (dba=iblock.ib_dba[j]) != DBNULL)
                        lfsdfree(diskdev, dba);
        sem = ( (struct dsblk *)devtab[diskdev].dvioblk)->dflsem;
        wait(sem);
        iblock.ib_next = dirptr->d_filst;
        dirptr->d_filst = iblist;
        ibput(diskdev, ilast, &iblock);
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(sem);
```

```
        return(OK);
}
```

Iblfree moves along the list of i-blocks, calling procedure *lfsdfree* to free each data block. After all data blocks have been released, *iblfree* links the nodes on its argument list into the free list. To do so, it adds a pointer from the tail of the argument list to the head of the free list, and makes the free list point to the head of its argument list. After updating the free list pointer, *iblfree* writes a copy of the modified directory back to disk. Although these operations would be simpler if the lists resided in memory, the details are not difficult to follow.

17.7 The Directory Structure

Before plunging into the details of the file system software, we need to consider the format of data in the directory. Obviously, the directory must contain an entry for each file. The file entry includes the file's name and the address of the first i-block on the file's index list. The directory also contains the total number of i-blocks on the disk, as well as pointers to the lists of free blocks. Structure *dir*, found in file *dir.h*, defines the directory layout in detail.

```
/* dir.h */

#define FDNLEN  10                      /* length of file name + 1      */
#define NFDES   28                      /* number of files / directory  */

struct  fdes    {                       /* description of each file     */
        long    fdlen;                  /* length in bytes              */
        IBADDR  fdiba;                  /* first index block            */
        char    fdname[FDNLEN];         /* zero terminated file name    */
};

struct  dir     {                       /* directory layout             */
        int     d_iblks;                /* i-blocks on this disk        */
        DBADDR  d_fblst;                /* pointer to list of free blks */
        IBADDR  d_filst;                /* pointer to list of free iblks*/
        int     d_id;                   /* disk identification integer  */
        int     d_nfiles;               /* current number of files      */
        struct  fdes    d_files[NFDES]; /* description of the files     */
};

struct  freeblk {                       /* shape of block on free list  */
        DBADDR  fbnext;                 /* address of next free block   */
};

extern struct fdes *dfdsrch();
```

Because index blocks each contain a ‘‘next’’ pointer field, linking them into the free list is easy. Normally, data blocks do not contain pointers, however, so linking them onto a free list is not as simple. We have chosen to link free data blocks together in a singly-linked list by storing pointers at the beginning of each. Structure *freeblk* in file *dir.h* documents this decision. Whenever the software manipulates data blocks on the free list, it assumes they have the shape declared by this structure. For example, if disk block addresses are two bytes long, data blocks on the free list will contain a pointer to the next block in the first two bytes.

17.8 Using The Device Switch Table For Files

The file system software must establish connections between running processes and disk files, so that operations like *read* and *write* can be mapped onto the correct file. Exactly how the system performs this mapping depends on both the size and generality needed. To keep our system small, we will avoid introducing new software by using the device switch machinery already in place.

Just as with video windows, described in Chapter 14, a set of *pseudo-devices* have been added to the device switch table such that each pseudo-device can be used to control a file. Just like conventional devices, these file pseudo-devices have a set of driver routines that perform *read*, *write*, *getc*, *putc*, *seek*, and *close* operations. When a process opens a disk file, the file system searches for a currently unused pseudo-device, sets up the control block for that ''device,'' and returns the device identifier to the caller. After the file has been opened, the process uses the device identifier to *read* or *write* it. The device switch maps high-level operations to driver routines for the pseudo-device, exactly as they map high-level operations onto device drivers for other devices. Finally, the process finishes accessing the file, and calls *close* to break the connection and make the pseudo-device available for use with another file. The details will become clear as we review the code.

Just like other drivers, the file pseudo-device driver keeps a control block for each pseudo-device. File *file.h* contains the pertinent declarations.

```
/* file.h */

/* Local disk layout: disk block 0 is directory, then index area, and   */
/* then data blocks.  Each disk block (512 bytes) in the index area     */
/* contains 8 iblocks, which are 64 bytes long.  Iblocks are referenced */
/* relative to 0, so the disk block address of iblock k is given by     */
/* truncate(k/8)+1.  The offset of iblock k within its disk block is    */
/* given by 64*remainder(k,8).  The directory entry points to a linked  */
/* list of iblocks, and each iblock contains pointers to IBLEN (29) data*/
/* blocks. Index pointers contain a valid data block address or DBNULL. */

#define EOF             -2              /* value returned on end-of-file*/
#define FLREAD          001             /* fl_mode bit for "read"       */
#define FLWRITE         002             /* fl_mode bit for "write"      */
#define FLRW            003             /* fl_mode bits for read+write  */
#define FLNEW           010             /* fl_mode bit for "new file"   */
#define FLOLD           020             /* fl_mode bit for "old file"   */

struct  flblk   {                       /* file "device" control block  */
        int     fl_id;                  /* file's "device id" in devtab */
        int     fl_dev;                 /* file is on this disk device  */
        int     fl_pid;                 /* process id accessing the file*/
        struct  fdes    *fl_dent;       /* file's in-core dir. entry    */
        int     fl_mode;                /* FLREAD, FLWRITE, or both     */
        IBADDR  fl_iba;                 /* address of iblock in fl_iblk */
        struct  iblk    fl_iblk;        /* current iblock for file      */
        int     fl_ipnum;               /* current iptr in fl_iblk      */
        long    fl_pos;                 /* current file position (bytes)*/
        Bool    fl_dch;                 /* has fl_buff been changed?    */
        char    *fl_bptr;               /* ptr to next char in fl_buff  */
        char    fl_buff[DBUFSIZ];       /* current data block for file  */
};

#ifdef  Ndf
extern  struct  flblk   fltab[];
#endif
```

Most of the fields in the control block make sense without explanation. Field *fl_id*, for example, contains the file's device id. Field *fl_mode* tells whether the file has been opened for reading, writing, or both. (Symbolic constants *FLREAD* and *FLWRITE* identify the individual mode bits.) Field *fl_dent* points to the file's directory entry; it is the only link between the control block and the file name.

Other fields of the file control block contain information that identifies a position in the file and the data found at that position. Opening a file establishes a ''cursor'' that points to the beginning of the file. As processes *read* or *write* data, the cursor moves along through the file. At any time, field *fl_pos* gives the *current cursor position*, measured in bytes from the beginning of the file.

The control block contains enough information to enable upper-half routines to easily retrieve or modify data found at the current cursor position. Field *fl_iblk* contains a copy of the i-block from the file's index list that indexes the current position. Field *fl_ipnum* identifies which pointer in the index block corresponds to the current position. The data block in buffer *fl_buff* is the data block that includes the current position. Finally, field *fl_bptr* points to the character in the buffer that is found at the current position.

17.9 Establishing A Pseudo-Device

Making a connection between a running program and a named file involves searching the directory to see if the name is valid, allocating a pseudo-device, and initializing the control block and cursor to the beginning of the file. We will examine procedures that perform each of these tasks.

In PC-Xinu the user can specify, when opening a file, whether the file should be old or new (i.e., whether it must or must not exist) and whether it will be read or written. Such specifications are made by passing a *mode string* to the procedure that opens files. To parse the mode string and convert it to a single integer, the open routine calls *dfckmd*, shown below. *Dfckmd* returns an integer with mode bits set according to the symbolic constants defined in *file.h*.

```
/* dfckmd.c - dfckmd */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  dfckmd  --  parse file mode argument and generate actual mode bits
 *------------------------------------------------------------------------
 */
dfckmd(mode)
char    *mode;
{
        int     mbits;
        char    ch;

        mbits = 0;
        while (ch = *mode++)
                switch (ch) {

                    case 'r':   if (mbits&FLREAD) return(SYSERR);
                                mbits |= FLREAD;
                                break;

                    case 'w':   if (mbits&FLWRITE) return(SYSERR);
                                mbits |= FLWRITE;
                                break;

                    case 'o':   if (mbits&FLOLD || mbits&FLNEW)
                                        return(SYSERR);
                                mbits |= FLOLD;
                                break;

                    case 'n':   if (mbits&FLOLD || mbits&FLNEW)
                                        return(SYSERR);
                                mbits |= FLNEW;
                                break;

                    default:    return(SYSERR);
                }
        if (mbits&FLREAD == mbits&FLWRITE)      /* default: allow R + W */
                mbits |= (FLREAD| FLWRITE);
        return(mbits);
}
```

Dfckmd scans the mode string looking for occurrences of the characters ‘‘n’’ (new), ‘‘o’’ (old), ‘‘r’’ (read), and ‘‘w’’ (write). As it finds each character, it sets the appropriate bit of the mode integer. If it finishes without detecting an error, it returns the resulting integer. Note that the mode string may specify a new file or an old file, but not both. If neither new nor old is specified, *dfckmd* assumes that the user does not care whether the file currently exists, or whether it must be created.

Once the mode string has been parsed, the directory can be searched. Procedure *dfdsrch* uses the mode integer created by *dfckmd*. It searches for the specified file name, creating a new file if the mode bits allow creation and the name does not exist. The code is shown below.

```
/* dfdsrch.c - dfdsrch */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  dfdsrch  --  search disk directory for position of given file name
 *------------------------------------------------------------------------
 */
struct  fdes    *dfdsrch(dsptr, filenam, mbits)
struct  dsblk   *dsptr;
char    *filenam;
int     mbits;
{
        struct  dir     *dirptr;
        struct  fdes    *fdptr;
        int     len;
        int     i;
        int     inum;

        if ( (len=strlen(filenam))<=0 || len>=FDNLEN)
                return((struct fdes *)SYSERR);
        if ( (dirptr=dsdirec(dsptr->dnum)) == (struct dir *) NULL )
                return((struct fdes *)SYSERR);
        for (i=0 ; i<dirptr->d_nfiles ; i++)
                if (strcmp(filenam, dirptr->d_files[i].fdname) == 0)
                        if ( (mbits&FLNEW) != 0)
                                return((struct fdes *)SYSERR);
                        else
                                return(&dirptr->d_files[i]);
        wait(dsptr->ddirsem);
        if ( (mbits&FLOLD) || dirptr->d_nfiles >= NFDES) {
                signal(dsptr->ddirsem);
```

```
                return((struct fdes *)SYSERR);
        }
        inum = ibnew(dsptr->dnum, IBNWDIR);
        fdptr = &(dirptr->d_files[dirptr->d_nfiles++]);
        fdptr->fdlen = 0L;
        strcpy(fdptr->fdname, filenam);
        fdptr->fdiba = inum;
        write(dsptr->dnum, dskbcpy(dirptr), DIRBLK);
        signal(dsptr->ddirsem);
        return(fdptr);
}
```

Although the code may seem confusing, it is easy to understand. *Dfdsrch* first checks to see that the name is valid. It then searches the directory. If a match is found, it uses the mode bits to determine whether an old file is allowed. If no match is found, *dfdsrch* uses the mode bits to determine whether a new file is allowed. If a new file is needed, *dfdsrch* creates one by adding the new name to the directory and allocating an i-block for the file from the free list.

Once a file has been found or created, a connection to it can be established by allocating a pseudo-device and initializing its control block. Procedure *dfalloc* allocates a pseudo-device.

```
/* dfalloc.c - dfalloc */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  dfalloc  --  allocate a device table entry for a disk file; return id
 *------------------------------------------------------------------------
 */
#ifdef  Ndf
dfalloc()               /* assume exclusion for dir. provided by caller */
{
        int     i;

        for (i=0 ; i<Ndf ; i++)
                if (fltab[i].fl_pid == 0) {
                        fltab[i].fl_pid = getpid();
                        return(i);
                }
```

```
        return(SYSERR);
}
#endif
```

Allocation is straightforward. *Dfalloc* searches the array of pseudo-device control blocks looking for an unused device. It checks the process id field in each pseudo-device control block, because the process id is nonzero whenever the device is in use. When it finds an unused device, *dfalloc* marks the device busy by storing the caller's process id in the process id field. It then returns the index of the file control block it reserved.

Procedure *dsopen* uses the three procedures described above to make a connection between a running program and a disk file. It calls *dfckmd* to check the mode string and convert it to an integer, *dfdsrch* to search the directory, and *dfalloc* to allocate a pseudo-device for the file. Finally, *dsopen* fills in the file control block by setting the current position to zero and reading the first i-block from the file's index list. It returns the file's device id to the caller, so it can be used in operations like *read* and *write*. File *dsopen.c* contains the code.

```
/* dsopen.c - dsopen */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  dsopen  --  open/create a file on the specified disk device
 *------------------------------------------------------------------------
 */
#ifdef  Ndf
dsopen(devptr, filenam, mode)
struct  devsw   *devptr;
char    *filenam;
char    *mode;
{
        struct  flblk   *flptr;
        struct  fdes    *fdptr;
        DBADDR  dba;
        int     mbits, findex;
        int     id;
        int     ps;

        if ( (mbits=dfckmd(mode)) == SYSERR )
                return(SYSERR);
        disable(ps);
        if( (int)(fdptr=dfdsrch(devptr->dvioblk,filenam,mbits)) == SYSERR
                || (findex=dfalloc()) == SYSERR ) {
                restore(ps);
                return(SYSERR);
        }
        flptr = &fltab[findex];
        flptr->fl_dev = devptr->dvnum;
        flptr->fl_dent = fdptr;
        flptr->fl_mode = mbits & FLRW;
        flptr->fl_iba = fdptr->fdiba;
        ibget(flptr->fl_dev, flptr->fl_iba, &(flptr->fl_iblk));
        flptr->fl_pos = 0L;
        flptr->fl_dch = FALSE;
        dba = flptr->fl_iblk.ib_dba[flptr->fl_ipnum = 0];
        if (dba != DBNULL) {
                read(flptr->fl_dev, flptr->fl_buff, dba);
                flptr->fl_bptr = flptr->fl_buff;
```

```
        } else
                flptr->fl_bptr = &flptr->fl_buff[DBUFSIZ];
        id = flptr->fl_id;
        restore(ps);
        return(id);
}
#endif
```

As the name *dsopen* implies, this procedure is associated with the disk driver. The relationship between files and the disk driver is similar to the relationship between windows and the tty driver discussed in Chapter 14. In particular:

> *Because the directory maps names to disk files, the open operation is associated with the disk driver, not with the individual files.*

To connect to a file on disk device *i*, the user calls *open*, passing *i* as the device argument, the filename as the second argument, and the mode string as the third. *Open* uses the device switch table to pass the call to *dsopen*. If *dsopen* is successful in opening the file, it returns the device number of the file pseudo-device, which is used by the calling process for accessing the file data.

17.10 Pseudo-Device Driver Routines

Like any device driver, pseudo-devices need routines that handle high-level operations like *read* or *write*. Of course, there are no lower-half routines for file pseudo-devices because the driver never receives interrupts from a real hardware device. Instead of starting hardware, the upper-half file routines carry out requests by performing input and output operations on the disk device.

Although the pseudo-device drivers do not contend with real hardware, they are quite complex because they deal with the details of buffer and index management. To help manage the complexity, some of the work has been pushed into separate procedures; we will consider these first. Procedures *lfsnewd* and *lfsdfree* allocate and release data blocks. Procedure *lfsnewd* allocates a data block from the free list almost exactly the same way *ibnew* allocated an index block. The code, found in file *lfsnewd.c*, needs little further explanation.

```
/* lfsnewd.c - lfsnewd */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

#define  DFILLER  '+'

/*------------------------------------------------------------------------
 *  lfsnewd  --  allocate a new data block from free list on disk
 *------------------------------------------------------------------------
 */
lfsnewd(diskdev, flptr)
int     diskdev;
struct  flblk   *flptr;
{
        struct  dir     *dirptr;
        struct  freeblk *fbptr;
        char    *buf;
        int     sem;
        DBADDR  dba;
        int     i;

        dirptr = dsdirec(diskdev);
        fbptr = (struct freeblk *) (buf = flptr->fl_buff);
        sem = ((struct dsblk *)devtab[diskdev].dvioblk)->dflsem;
        wait(sem);
        dba = dirptr->d_fblst;
        read(diskdev, fbptr, dba);
        dirptr->d_fblst = fbptr->fbnext;
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(sem);
        for (i=0 ; i<DBUFSIZ ; i++)
                *buf++ = DFILLER;
        write(diskdev, dskbcpy(fbptr), dba);
        return(dba);
}
```

Companion procedure *lfsdfree* reverses the action of *lfsnewd* by returning a data block to the free list. As expected, it makes the data block point to the current list and makes the list head point to the block being deallocated. Again, the code is similar to the code used to deallocate i-blocks.

```
/* lfsdfree.c - lfsdfree */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfsdfree  --  free a data block given its address
 *------------------------------------------------------------------------
 */
lfsdfree(diskdev, dba)
int     diskdev;
DBADDR  dba;
{
        struct  dir     *dirptr;
        int     dirsem;
        struct  freeblk *buf;

        dirptr = dsdirec(diskdev);
        dirsem = ((struct dsblk *)(devtab[diskdev].dvioblk))->dflsem;
        buf = (struct freeblk *)getbuf(dskdbp);
        wait(dirsem);
        buf->fbnext = dirptr->d_fblst;
        dirptr->d_fblst = dba;
        write(diskdev, buf, dba);
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        signal(dirsem);
        return(OK);
}
```

Lfsdfree calls the routine *dskbcpy* to copy the in-core directory into a buffer for writing to disk. Recall that the *dswrite* routine queues the write request and returns immediately to the caller. Since *dsinter* ultimately calls *freebuf* to deallocate the buffer, *dskbcpy* must allocate a buffer from the disk data buffer pool before copying the directory block into it for writing. The *dskbcpy* routine, shown below, is used for the same purpose in other routines described in this chapter.

```
/* dskbcpy.c - dskbcpy */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <mark.h>
#include <bufpool.h>

/*------------------------------------------------------------------------
 *  dskbcpy  --  copy data into a new disk buffer and return its address
 *------------------------------------------------------------------------
 */
char *dskbcpy(oldbuf)
char    *oldbuf;
{
        int     i;
        char    *newbuf, *to;

        newbuf = to = getbuf(dskdbp);
        for (i=0 ; i<DBUFSIZ ; i++)
                *to++ = *oldbuf++;
        return(newbuf);
}
```

17.10.1 Index Management Routines

The file system cannot afford to write a data block every time the program changes one character. Instead, it waits until the block has been filled and a new one is needed before writing the block to disk. The system must also write charged blocks when the user closes the file (even if the block has not been filled), and when the user positions the file pointer outside the current block by calling *seek*. Of course, the file system should not copy the buffer to disk unless it has been changed, because disk accesses require much time. To eliminate unnecessary writes, it uses a Boolean variable (field *fl_dch* in the file control block), clearing the variable whenever a new block has been read and setting it whenever the contents are changed.

Procedure *lfsflush* writes data from the buffer to disk if the data has been changed. It computes the disk block address of the current buffer by looking in the i-block (field *fl_iblk*) and writes a copy of the buffer to that address. The code is found in file *lfsflush.c*.

```
/* lfsflush.c - lfsflush */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfsflush  --  flush data and i-block for a file
 *------------------------------------------------------------------------
 */
lfsflush(flptr)
struct  flblk   *flptr;
{
        DBADDR  dba;

        if ( flptr->fl_dch == FALSE )
                return;
        dba = flptr->fl_iblk.ib_dba[flptr->fl_ipnum];
        write(flptr->fl_dev, dskbcpy(flptr->fl_buff), dba);
        flptr->fl_dch = FALSE;
        return;
}
```

Another procedure that helps manage the index is named *lfsetup*. *Lfsetup* positions a file at a specified location by finding the correct index block and data block. It starts with the current i-block and moves along the index list until it finds the correct i-block. The index list is singly-linked, so if the desired position lies before the region covered by the current index block, *lfsetup* moves to the file's first index block before it begins the search. Once the correct index block has been read into memory, *lfsetup* determines which data block pointer to use by moving to the correct entry in the i-block. After extracting the data block address, it reads a copy of the data block into the buffer. Finally, it positions the buffer pointer to the desired byte within the buffer. The code is shown below.

```
/* lfsetup.c - lfsetup */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfsetup  --  set up appropriate iblock and data block in memory
 *------------------------------------------------------------------------
 */
lfsetup(diskdev, flptr)
int     diskdev;
struct  flblk   *flptr;
{
        struct  iblk    *ibptr;
        int     displ, i;
        long    ibrange;
        IBADDR  nextib;
        DBADDR  dba;

        ibrange = (long) (IBLEN * DBUFSIZ);
        ibptr = &flptr->fl_iblk;
        if (flptr->fl_pos < ibptr->ib_byte) {
                flptr->fl_iba = (flptr->fl_dent)->fdiba;
                ibget(diskdev, flptr->fl_iba, ibptr);
        }
        while (ibptr->ib_byte+ibrange <= flptr->fl_pos) {
                if (ibptr->ib_next == IBNULL) {
                        ibptr->ib_next = ibnew(diskdev, IBWDIR);
                        ibput(diskdev, flptr->fl_iba, ibptr);
                        flptr->fl_iba = ibptr->ib_next;
                        ibclear(ibptr, (long)ibptr->ib_byte+ibrange);
                        ibput(diskdev, flptr->fl_iba, ibptr);
                } else {
                        flptr->fl_iba = ibptr->ib_next;
                        ibget(diskdev, flptr->fl_iba, ibptr);
                }
        }
        displ = (int) (flptr->fl_pos - ibptr->ib_byte);
        for (flptr->fl_ipnum=0 ; displ>=DBUFSIZ ; displ-=DBUFSIZ)
                flptr->fl_ipnum++;
        flptr->fl_bptr = flptr->fl_buff + displ;
        if ( (dba=ibptr->ib_dba[flptr->fl_ipnum]) == DBNULL) {
```

```
                ibptr->ib_dba[flptr->fl_ipnum] = lfsnewd(diskdev,flptr);
                ibput(diskdev, flptr->fl_iba, ibptr);
        } else
                read(diskdev, flptr->fl_buff, dba);
        flptr->fl_dch = FALSE;
}
```

17.10.2 The Pseudo-Device Seek Routine

The *seek* operation moves the current file position to a given location without transferring data. Procedure *lfseek*, shown below, implements the *seek* operation. First, it checks to see if the buffer has been modified and writes a copy to disk if it has been. It then verifies that the specified position is valid. If the request is valid, *lfseek* updates the current file position (*fl_pos*), and calls *lfsetup* to locate the correct i-block.

```
/* lfseek.c - lfseek */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfseek  --  seek to a specified position of a file
 *------------------------------------------------------------------------
 */
lfseek(devptr, offset)
struct  devsw   *devptr;
long    offset;
{
        struct  flblk   *flptr;
        int     retcode;
        int     ps;

        disable(ps);
        flptr = (struct flblk *)devptr->dvioblk;
        if (flptr->fl_mode & FLWRITE) {
                if (flptr->fl_dch)
                        lfsflush(flptr);
        } else if (offset > (flptr->fl_dent)->fdlen) {
                        restore(ps);
                        return(SYSERR);
        }
        flptr->fl_pos = offset;
        lfsetup(flptr->fl_dev, flptr);
        restore(ps);
        return(OK);
}
```

17.10.3 The Pseudo-Device Getc Routine

Once a file has been opened and the index is in place, input from the file is trivial. It consists of extracting the next character from the buffer and advancing the buffer pointer. Procedure *lfgetc*, shown below, performs the operation.

```
/* lfgetc.c - lfgetc */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfgetc  --  get next character from (buffered) disk file
 *------------------------------------------------------------------------
 */
lfgetc(devptr)
struct  devsw   *devptr;
{
        struct  flblk   *flptr;
        char    nextch;
        int     ps;

        disable(ps);
        flptr = (struct flblk *)devptr->dvioblk;
        if (flptr->fl_pid!=currpid || !(flptr->fl_mode&FLREAD)) {
                restore(ps);
                return(SYSERR);
        }
        if (flptr->fl_pos >= (flptr->fl_dent)->fdlen) {
                restore(ps);
                return(EOF);
        }
        if (flptr->fl_bptr >= &flptr->fl_buff[DBUFSIZ]) {
                if (flptr->fl_dch)
                        lfsflush(flptr);
                lfsetup(flptr->fl_dev, flptr);
        }
        nextch = *(flptr->fl_bptr)++;
        flptr->fl_pos++;
        restore(ps);
        return( ((int)nextch) & 0xff );
}
```

When called, *lfgetc* checks to see that the invoking process owns the pseudo-device and that the file has been opened for reading. It then checks to see if the current position lies beyond the end of the file and returns a value indicating *end-of-file* (*EOF*) if it does. If end of file has not been reached, *lfgetc* checks to see if the buffer pointer points beyond

the current buffer and moves to the next data block if necessary. Finally, after it has positioned the buffer pointer correctly, *lfgetc* returns the next character from the buffer.

17.10.4 The Pseudo-Device Putc Routine

Procedure *lfputc* writes a single character to an open file. Like *lfgetc*, it checks the file status and moves to a new buffer if the current one is full. It then deposits the character and sets *fl_dch* to indicate that the buffer has been changed. Note that *lfputc* merely accumulates characters in the buffer; it does not write the buffer to disk each time it changes. The buffer will only be copied to disk when the current position moves to another disk block.

```
/* lfputc.c - lfputc */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfputc  --  put a character onto a (buffered) disk file
 *------------------------------------------------------------------------
 */
lfputc(devptr, ch)
struct  devsw   *devptr;
char    ch;
{
        struct  flblk   *flptr;
        int     ps;

        disable(ps);
        flptr = (struct flblk *) devptr->dvioblk;
        if (flptr->fl_pid != currpid || !(flptr->fl_mode&FLWRITE)) {
                restore(ps);
                return(SYSERR);
        }
        if (flptr->fl_bptr >= &flptr->fl_buff[DBUFSIZ]) {
                if (flptr->fl_dch)
                        lfsflush(flptr);
                lfsetup(flptr->fl_dev, flptr);
        }
        flptr->fl_pos++;
        if ( flptr->fl_pos > (flptr->fl_dent)->fdlen )
```

```
                (flptr->fl_dent)->fdlen = flptr->fl_pos;
        *(flptr->fl_bptr)++ = ch;
        flptr->fl_dch = TRUE;
        restore(ps);
        return(OK);
}
```

17.10.5 The Pseudo-Device Read Routine

Procedure *lfread* implements the *read* operation. It reads zero or more characters into a specified buffer by calling *lfgetc* repeatedly, as shown below. This implementation was chosen because it was easy to code; the exercises suggest redesigning it to improve the efficiency.

```
/* lfread.c - lfread */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfread  --  read from a previously opened disk file
 *------------------------------------------------------------------------
 */
lfread(devptr, buff, count)
struct  devsw   *devptr;
char    *buff;
int     count;
{
        int     done;
        int     ichar;

        if (count < 0)
                return(SYSERR);
        for (done=0 ; done < count ; done++)
                if ( (ichar=lfgetc(devptr)) == SYSERR)
                        return(SYSERR);
                else if (ichar == EOF ) {       /* EOF before finished */
                        if (done == 0)
                                return(EOF);
                        else
                                return(done);
                } else
                        *buff++ = (char) ichar;
        return(done);
}
```

17.10.6 The Pseudo-Device Write Routine

Procedure *lfwrite* implements the *write* operation by writing a sequence of zero or more bytes. Like *lfread*, it calls a single-character (*lfputc*, in this case) routine to perform the output operation. File *lfwrite.c* contains the code.

```
/* lfwrite.c - lfwrite */

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  lfwrite  --  write 'count' bytes onto a local disk file
 *------------------------------------------------------------------------
 */
lfwrite(devptr, buff, count)
struct  devsw   *devptr;
char    *buff;
int     count;
{
        int     i;

        if (count < 0)
                return(SYSERR);
        for (i=count; i>0 ; i--)
                if (lfputc(devptr, *buff++) == SYSERR)
                        return(SYSERR);
        return(count);
}
```

17.10.7 The Pseudo-Device Close Routine

When a process finishes using a file, it must call *close* to flush unwritten buffers out to disk and detach the file from the process. Procedure *lfclose* implements the *close* operation. It performs exactly as expected: it calls *lfsflush* to write data from the buffer if necessary, and assigns the process id field in the file control block zero to indicate that the pseudo-device can be used again. After it writes the data blocks to disk, *lfclose* writes a copy of the directory to disk in case the file length recorded in the directory entry was changed. The details are shown below in file *lfclose.c*.

```
/* lfclose.c - lfclose */

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <disk.h>
#include <file.h>

/*------------------------------------------------------------------------
 *  lfclose  --  close a file by flushing output and freeing device slot
 *------------------------------------------------------------------------
 */
lfclose(devptr)
struct  devsw   *devptr;
{
        struct  dsblk   *dsptr;
        struct  dir     *dirptr;
        struct  flblk   *flptr;
        int     diskdev;
        int     ps;

        disable(ps);
        flptr = (struct flblk *) devptr->dvioblk;
        if (flptr->fl_pid != currpid) {
                restore(ps);
                return(SYSERR);
        }
        diskdev = flptr->fl_dev;
        dsptr = (struct dsblk *)devtab[diskdev].dvioblk;
        dirptr = (struct dir *) dsptr->ddir;
        if ( (flptr->fl_mode & FLWRITE) && flptr->fl_dch)
                lfsflush(flptr);
        flptr->fl_pid = 0;
        dsptr->dnfiles--;
        write(diskdev, dskbcpy(dirptr), DIRBLK);
        restore(ps);
        return(OK);
}
```

17.10.8 The Pseudo-Device Initialization Routine

Although procedure *dsopen* initializes most of the entries in the file control block when it opens a file, some initialization is required at system startup. As in most drivers, the initialization routine assigns field *fl_id* in the file control block the device id that the high-level routines call to access the file. It also initializes field *fl_pid* to zero, indicating that the pseudo-device is not in use. The pseudo-device initialization routine also places a pointer to the file control block in the device switch table (field *dvioblk*), so the upper-half routines can find the correct control block. These details are all handled by procedure *lfinit*, as shown below.

```
/* lfinit.c - lfinit */

#include <conf.h>
#include <kernel.h>
#include <disk.h>
#include <file.h>

#ifndef Ndf
#define Ndf     1
#endif
struct  flblk   fltab[Ndf];

/*------------------------------------------------------------------------
 *  lfinit  --  mark disk file 'device' available at system startup
 *------------------------------------------------------------------------
 */
lfinit(devptr)
struct  devsw   *devptr;
{
        struct  flblk   *flptr;

        devptr->dvioblk = (char *)(flptr = &fltab[devptr->dvminor]);
        flptr->fl_pid = 0;
        flptr->fl_id = devptr->dvnum;
        return(OK);
}
```

17.11 Summary

The file system manages an abstract name space in which objects correspond to disk files, devices, or operating system services. This chapter concentrated on the part of the file system software that manages data files on secondary storage. To keep the interface to files the same as the interface to devices, the software is organized into a pseudo-device driver that supports *read*, *write*, *getc*, *putc*, *seek* and *close* operations. When a process opens a file, the software establishes a connection to it through the device switch table, allowing the high-level I/O routines to access the file driver, just as they access hardware device drivers.

Our design allows multiple files to grow concurrently by using an index to keep track of the data blocks associated with each file. The index for a file consists of a singly-linked list of nodes called i-blocks, where each i-block contains pointers to a set of data blocks. When a file is opened, the driver software reads its first index block into memory. It also reads a copy of the first data block if the file is nonempty. Subsequent accesses or changes affect the buffer in memory. Only when the file position moves outside the current buffer does the file system copy the buffer back to disk and read another.

Concurrency control, as well as the details of index and buffer management, make the file system software large and complex. The large volume of detail has been handled by dividing the driver into small pieces. We have also chosen to make the system simpler by limiting concurrent access to a file.

FOR FURTHER STUDY

Literature on file systems abounds. Knuth [1973] describes several data structures used for indexes. The search method described here is an *indexed sequential file* where the offset in the file is the key; it is a simplification of the UNIX file system described by Ritchie and Thompson [1974]. Broader descriptions of file system alternatives can be found in Calingaert [1982], Habermann [1976], and Peterson and Silberschatz [1983]. Some of these authors distinguish between the terms *file system*, using it to refer to the lowest layer of file manipulation software, and *directory system*, using it to refer to the mapping of names onto files.

EXERCISES

17.1 The number of index blocks is important because having too many blocks wastes space that could be used for data, while having too few makes it impossible to allocate all space to files. Given that there are 29 data block pointers in an i-block and that 8 i-blocks fill a disk block, how many index blocks might be needed for a disk of n total blocks if the directory can hold k files?

17.2 Write a routine to make an empty PC-Xinu file system on a formatted disk having 720 disk blocks (sectors). You will need to arrange for the directory (one block, 28 directory entries), the index area, and the data area. Use the previous exercise to determine the number of disk blocks needed for the index area.

17.3 Write a routine to print the directory of a PC-Xinu file system disk.

17.4 Redesign routines *lfread* and *lfwrite* to perform high-speed copies like the *tty read* and *write* routines.

17.5 Consider what happens when two processes open the same file and begin writing on it. Rewrite the code to prevent the problem.

17.6 Free data blocks are chained together on a singly-linked list. Redesign the software to use an index for them (i.e., link them into a giant ''file'' made up of free blocks).

17.7 What are the maximum number of disk accesses necessary to allocate/free a data block under the current scheme, and the scheme suggested in the previous problem? The average number of accesses?

17.8 The size of index i-blocks is as important as the number of them. Does the distribution of file sizes on your local computer system suggest how the i-block size was chosen?

17.9 Rewrite the code that handles process termination so it closes all files associated with the terminating process.

17.10 Change the system to separate the file switch table from the device switch table. Make the directory contain a field for each name, telling whether that name refers to a conventional file or a device.

17.11 Discuss the advantages and disadvantages of adopting the convention that all entries in the device switch table above position *NDEVS* are files.

17.12 Copying data among buffers can be expensive. Modify the disk driver and file system procedures to use *int* pointers in place of *char* pointers when copying data, and measure the resulting speed-up.

17.13 *Lfclose* modifies the directory block but does not do so in a critical section under control of the directory semaphore *dirsem*. Modify *lfclose* to correct this.

17.14 When opening a nonempty file, *dsopen* reads the first block of the file into the data buffer. Under what circumstances can this be considered inefficient? Modify *dsopen* so that it will not read the first block.

17.15 If a file is to be read (written) sequentially (*i.e.*, no *seek* operations), the file system can be reading (writing) the next (previous) data block into another buffer, while a process is manipulating data in the current block. Such a scheme is called *double buffering*. Modify the file system design to allow for double buffering.

17.16 When is double buffering disadvantageous?

17.17 Design the file system to switch to or from double buffering automatically based on whether a file is being accessed sequentially. Can the file system reliably and efficiently determine whether a file is being accessed sequentially?