16

A Disk Driver

This chapter describes the design of device driver routines for secondary storage devices called *disks*. At this level, the disk is viewed as a randomly accessible array of blocks. The next chapter describes how higher layers of the file system software build on this driver to provide named files and directories.

Disk drivers differ from the single-character terminal drivers covered earlier in several ways. Because disk devices are more complex than terminals, the interaction between the driver software and the device is more complicated. The disk is a *random access* device. Therefore, the driver must specify where on the disk data transfer should take place. Even the basic hardware differs. Disk hardware uses *direct-memory-access* mode (*DMA*), so it does not interrupt the CPU for every character transfer. Instead, the driver must arrange to transfer data in large blocks.

16.1 Operations Supplied By The Disk Driver

When users think of secondary storage devices like disks, they visualize programs or textual material organized into files, with files further organized into directories. It is the file system software that provides such facilities, not the hardware. At the device driver level, a disk is nothing more than a large array of data blocks that can be accessed randomly using two basic operations: copy the contents of a selected block from disk to memory, or copy the contents of memory to a selected block on disk. The data blocks, which correspond to *sectors* on the disk, are all the same size. (The disk used with PC-Xinu has 512 bytes per sector, a typical size).

Because the raw hardware provides block transfer, it makes sense to design a disk driver that *read*s and *write*s entire blocks. The question becomes how to include the notion of ‘‘block selection’’ in the existing high-level I/O operations. A high-level *seek* operation might be appropriate, because it is designed to move to a specified position in a file. However, requiring a user first to position the disk arm and then *read* or *write* to

transfer data would be clumsy and inconvenient. To keep the driver interface simple, we will design the driver always to transfer exactly one block, and to use the ''length'' argument of *read* or *write* to specify a position. For example, the call

```
read ( DISKDEV, buff, 5 )
```

requests the driver to read all of block five into memory starting at location *buff*.

As long as calls to *read* and *write* include a block position, a separate *seek* operation is unnecessary. In many systems, however, disk controller hardware does not permit simultaneous transfer from two disks even though it does allow simultaneous motion of all disk arms. Operating systems using such hardware can significantly reduce disk access times by moving one disk arm to the desired position while reading a block from another disk. Arm movement is the most costly part of disk access, so overlapping *seeks* with computation (and transfer operations on other disks) is an important optimization. To ensure that higher layers of software can perform such optimization, a *seek* operation will be included in the design even though the PC BIOS does not support it.

Thus, the driver will supply three operations to higher level routines: *read*, which copies a single block from the disk into memory; *write*, which copies data from memory onto a specified disk block; and *seek*, which conceptually positions the disk head over a specified block without transferring data. Having decided on the interface, we are ready to begin writing the driver. Before studying the three driver routines that correspond to the high-level operations, the reader should review the BIOS disk interface introduced in Chapter 2.

16.2 The List Of Pending Disk Requests

Like other device drivers, disk driver routines are partitioned into two sets, the upper-half and the lower-half, that communicate through message-passing and a shared data structure. The shared structure includes a list of pending requests. Upper-half routines enqueue a request for an operation, send a message to start the lower-half process if it is idle, and then *wait* for it to complete. The lower-half, a process created at system initialization, awakens the process waiting for the request that just completed and starts the next operation.

Before considering the upper-half and lower-half driver procedures, we need to decide on the content and format of the data area that they share. Following the terminology used with previous drivers, the shared structure is called a *control block*. The list of pending operations forms the centerpiece of this control block. In practice, the designer only adds fields to the control block structure as the driver routines are built, but we will examine all the fields now and see how the driver uses them later in the chapter.

Structure *dsblk*, found in file *disk.h*, defines the disk control block in C.

```
/* disk.h - dssync, dsdirec */

typedef unsigned int    DBADDR;         /* disk data block addresses    */
#define DBNULL          (DBADDR)0177777 /* null disk block address      */

#include <iblock.h>
#include <dir.h>

struct  dsblk   {                       /* disk driver control block    */
        struct  dreq    *dreqlst;       /* list of pending requests     */
        int     dnum;                   /* device number of this disk   */
        int     dibsem;                 /* i-block mutual exclusion sem.*/
        int     dflsem;                 /* free list  "        "      " */
        int     ddirsem;                /* directory  "        "      " */
        int     dnfiles;                /* num. of currently open files */
        struct dir *ddir;               /* address of in-core directory */
        int     dsprocnum;              /* disk server process number   */
};

extern  struct  dsblk   dstab[];

struct  dreq    {                       /* node in list of requests     */
        DBADDR  drdba;                  /* disk block address to use    */
        int     drpid;                  /* process id making request    */
        char    *drbuff;                /* buffer address for read/write*/
        int     drop;                   /* operation: READ/WRITE/SEEK   */
        int     drstat;                 /* returned status OK/SYSERR    */
        struct  dreq    *drnext;        /* ptr to next node on req. list*/
};

#define DRNULL  (struct dreq *) 0       /* null pointer in request list */
#define DIRBLK  0                       /* block used to hold directory */
#define DONQ    2                       /* status if request enqueued   */
#define DBUFSIZ 512                     /* size of disk data block      */
#define DREQSIZ sizeof(struct dreq)     /* size of disk request node    */
#define NDSECT  720                     /* no. of physical disk sectors */

#ifndef NDBUFF
#define NDBUFF  10                      /* number of disk data buffers  */
#endif
#define NDREQ   10                      /* number of disk request buf.  */
#define DREAD   0                       /* read command in dreq.drop    */
#define DWRITE  1                       /* write   "                    */
#define DSEEK   2                       /* seek    "                    */
```

```
#define DSYNC   3                       /* sync    "  (test-disk-ready) */

#define DSPRIO  200                     /* disk server process priority */

extern  int     dskrbp;                 /* disk request node buffer pool*/
extern  int     dskdbp;                 /* disk data block buffer pool  */
extern  char    *dskbcpy();             /* copy to new disk block       */

/* disk control function codes */

#define DSKSYNC 0                       /* synchronize (flush all I/O)  */
#define DSKNAME 1                       /* rename a file                */
#define DSKZAP  2                       /* remove a file                */
#define dssync(ddev)    control((ddev),DSKSYNC);

#define dsdirec(ddev)   (((struct dsblk *)(devtab[ddev].dvioblk))->ddir)
```

Field *dnum* in structure *dsblk* should look familiar because it occurs in all driver control blocks: *dnum* gives the device id for the disk. Other fields present no surprises. Field *dsprocnum* is the process id of the disk server process. Field *dreqlst* points to the list of requested operations (or equals *DRNULL* if the list is empty). The driver follows an important invariant:

> *The first request on the list is always the one that the hardware is performing; if the list is empty, the server process is idle.*

The list of pending requests, headed by field *dreqlst*, is a singly linked list where each node on the list has the form given by structure *dreq*. In a given node, field *drdba* specifies the number of the disk block that the operation will affect, and field *drop* specifies the operation to be performed. If the operation transfers data, field *drbuff* gives the memory address for the transfer. This address is assumed to be the start of a 512-byte block. The remaining field, *drstat*, is used to communicate status information from the lower-half to the upper-half. The lower-half records information about errors that occur during the operation, and the upper-half extracts the information and passes it to the caller.

16.3 Enqueuing Disk Requests

To reduce the time needed to restart a disk operation after one completes, the disk driver routines follow this rule:

> *The lower-half disk driver processes I/O requests in the order that they occur on the request list. When one completes, it is removed from the list and the next one is started.*

Thus, the responsibility for ordering requests in a sensible way falls to the upper-half routines that enqueue requests.

Why is the order of requests important? Remember that disk accesses are slow compared to the processor and memory speed. Most of the time is expended moving the disk arm; the time required is approximately proportional to the distance moved. If all accesses refer to a small locality, the time per access is lower than if the requests refer to blocks scattered over the disk. Unfortunately, the driver receives requests from all processes, so the sequence of blocks specified are not usually restricted to a small locality, even if each individual process accesses blocks sequentially. Honoring these requests in a first-in-first-out (FIFO) order usually means moving the arm back and forth across the disk frequently. To reduce the arm motion, disk drivers reorder requests in an attempt to group together requests that access blocks in a small area. (Recall from Chapter 2 that the driver can afford to spend much time organizing requests, even if it only eliminates one or two sweeps of the arm.)

Given a set of outstanding requests, the driver must decide which one to satisfy next. Ideally, it should move the disk arm as little as possible, but postponing requests for outlying blocks indefinitely would be unfair because it would always favor requests in the current locality. Furthermore, the amount of time the driver spends reordering requests should be minimized. What is the best way to satisfy the goals of efficiency and fairness? There is no ‘‘best’’ way; it depends largely on the order in which requests arrive, the mixture of processes that are executing, and the characteristics of the hardware. Fortunately, compromises are possible. With only a small amount of CPU time, the driver can produce an order of requests that takes far less time than honoring requests first-come-first-serve.

Most drivers use rule-of-thumb, or *heuristic* procedures, to help optimize disk accesses. The heuristics, which are easy to compute, usually group requests by locality. For example, we will adopt the following heuristic which orders requests by block position.

> *When adding a request for block B to the existing list of requests, schedule it to be performed between requests i and i+1 if the disk arm will pass over block B on its way from i to i+1. If no such pair i and i+1 exist, add the new request to the end of the list.*

The general idea is to force the disk arm to ‘‘sweep’’ back and forth across the surface, going from low numbered blocks to high numbered ones and back again. Inserting blocks in the existing list optimizes arm movement because it accesses blocks when the arm passes over them. Inserting outlying requests at the end of the list either extends the sweep in one direction or starts it moving in the other.

Implementing the request ordering heuristic is straightforward; as procedure *dskenq* demonstrates.

```
/* dskenq.c - dskenq */

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

/*------------------------------------------------------------------------
 *  dskenq  --  enqueue a disk request and start I/O if disk not busy
 *------------------------------------------------------------------------
 */
dskenq(drptr, dsptr)
        struct  dreq    *drptr;
        struct  dsblk   *dsptr;
{
        struct  dreq    *p, *q;         /* q follows p through requests */
        DBADDR  block;
        int     st;

        if ( (q=dsptr->dreqlst) == DRNULL ) {
                dsptr->dreqlst = drptr;
                drptr->drnext = DRNULL;
                sendn(dsptr->dsprocnum);
                return(DONQ);
        }
        block = drptr->drdba;
        for (p = q->drnext ; p != DRNULL ; q=p,p=p->drnext) {
                if (p->drdba==block && (st=dskqopt(p, q, drptr)!=SYSERR))
                        return(st);
                if ( (q->drdba <= block && block < p->drdba) ||
                     (q->drdba >= block && block > p->drdba)   ) {
                        drptr->drnext = p;
                        q->drnext = drptr;
                        return(DONQ);
                }
        }
        drptr->drnext = DRNULL;
        q->drnext = drptr;
        return(DONQ);
}
```

Dskenq first examines the request list to see if it is empty because an empty list means that the device is idle. If the list is empty, it adds the new request and sends a message to restart the lower-half server process. Otherwise, it searches the existing list.

During the search of the request list, *dskenq* keeps two pointers, *p* and *q*, because the list is singly linked. As the search proceeds, *p* and *q* always point to adjacent nodes; initially, *q* points to the first node; *p* points to the node beyond that (or is empty if the list contains one node). Ignore for the moment the first ''if'' statement in the loop, and consider only the second one. If the request specifies a disk block that lies between the blocks specified in the requests given by *p* and *q*, the new request is inserted between them in the list. Otherwise, *p* and *q* move down the list. Testing whether the new request lies between two existing requests is done with four comparisons because adjacent blocks on the list may be in either ascending or descending order. Look carefully at the comparisons – *dskenq* takes care to ensure that multiple requests for the same block are handled in FIFO order; overlooking this detail can lead to totally unexpected results.

16.4 Optimizing The Request Queue

Because disk operations take much longer than CPU operations, the driver has been constructed to minimize arm movement. In special cases, further optimization is possible. Consider the situation in which a process attempts to *read* a block for which there is a pending *write* request already in the queue. The driver can copy the data from the buffer associated with the *write* request into the buffer associated with the *read* request and allow the reading process to continue. Another special case occurs when a second *write* request arrives for a given block before an existing request has been serviced (the driver can discard the first request).

To handle these special cases, *dskenq* uses procedure *dskqopt* as shown below. Look again at the loop in *dskenq* to see how *dskqopt* is called whenever the request being inserted in the list specifies a block number identical to one already in the list. If the ''optimization'' succeeds, *dskenq* returns to its caller. Otherwise, *dskenq* continues the usual insertion algorithm.

```
/* dskqopt.c - dskqopt */

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

/*------------------------------------------------------------------------
 *  dskqopt  --  optimize requests to read/write/seek to the same block
 *------------------------------------------------------------------------
 */
dskqopt(p, q, drptr)
struct  dreq    *p, *q, *drptr;
{
        char    *to, *from;
        int     i;
        DBADDR  block;

        /* By definition, sync requests cannot be optimized.  Also,     */
        /* cannot optimize read requests if already reading.            */

        if (drptr->drop==DSYNC || (drptr->drop==DREAD && p->drop==DREAD))
                return(SYSERR);

        if (drptr->drop == DSEEK) {     /* ignore extraneous seeks      */
                freebuf(drptr);
                return(OK);
        }

        if (p->drop == DSEEK) {         /* replace existing seeks       */
                drptr->drnext = p->drnext;
                q->drnext = drptr;
                freebuf(p);
                return(OK);
        }

        if (p->drop==DWRITE && drptr->drop==DWRITE) {   /* dup write    */
                drptr->drnext = p->drnext;
                q->drnext = drptr;
                freebuf(p->drbuff);
                freebuf(p);
                return(OK);
        }

        if (drptr->drop==DREAD && p->drop==DWRITE) {    /* satisfy read */
```

```
                to = drptr->drbuff;
                from = p->drbuff;
                for (i=0 ; i<DBUFSIZ ; i++)
                        *to++ = *from++;
                return(OK);
        }

        if (drptr->drop==DWRITE && p->drop==DREAD) {    /* sat. old read*/
                block = drptr->drdba;
                from = drptr->drbuff;
                for (; p!=DRNULL && p->drdba==block ; p=p->drnext) {
                        q->drnext = p->drnext;
                        to = p->drbuff;
                        for (i=0 ; i<DBUFSIZ ; i++)
                                *to++ = *from++;
                        p->drstat = OK;
                        ready(p->drpid);
                }
                drptr->drnext = p;
                q->drnext = drptr;
                resched();
                return(OK);
        }
        return(SYSERR);
}
```

16.5 Driver Initialization

Although the driver initialization is designed after the other parts, we have chosen to examine it now, because it also accesses the hardware directly. At startup, the system calls *init* for each disk device; *init* uses the device switch table to transfer control to the corresponding driver routine *dsinit*. Procedure *dsinit* fills in the disk control block and tests the disk.

```
/* dsinit.c - dsinit */

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

#ifdef  Ndsk
struct  dsblk   dstab[Ndsk];
#endif

int     dskdbp, dskrbp;

/*------------------------------------------------------------------------
 *  dsinit  --  initialize disk drive device
 *------------------------------------------------------------------------
 */
int dsinit(devptr)
struct devsw *devptr;
{
        struct  dsblk   *dsptr;
        int     pid;
        char    cp[8];
        int     dsinter();
        int     i;
        int     err;

        i = devptr->dvminor;
        devptr->dvioblk = (char *)(dsptr = &dstab[i]);
        dsptr->dsprocnum = -1;          /* impossible process # for now */
        dsptr->dreqlst = DRNULL;
        dsptr->dnum    = devptr->dvnum;
        dsptr->dnfiles = 0;
        if ((dsptr->ddir=(struct dir *)getbuf(dskdbp))==(struct dir *)NULL)
                return(SYSERR);
        if ( (err=dread(dsptr->ddir,i,0)) != 0 ) {
                kprintf("Disk read error %02xH reading drive %d\n",err,i);
                freebuf(dsptr->ddir);
                dsptr->ddir = (struct dir *) NULL;
                return(SYSERR);
        }
        dsptr->dibsem  = screate(1);
```

```
        dsptr->dflsem  = screate(1);
        dsptr->ddirsem = screate(1);
        strcpy(cp,"*DISK *");
        cp[5] = '0'+i;
        pid = create(dsinter,INITSTK,DSPRIO,cp,2,dsptr,i);
        dsptr->dsprocnum = pid;
        ready(pid);
        return(OK);
}
```

Following the same conventions used by other drivers, *dsinit* assumes it will be called once, before other operations are attempted.

Initialization is surprisingly simple. It consists of allocating a buffer to hold block 0 and three semaphores. The file system software uses the buffer and semaphores; they are not part of the driver. To test the disk hardware, *dsinit* reads block 0 into memory using a call to the BIOS disk read routine *dread*. (The file system uses block 0 as a directory.) If this call is unsuccessful, the initialization terminates; such a condition usually indicates that there is no disk in the specified drive.

16.6 The Upper-Half Read Routine

Having defined the request list and driver initialization routines, we are ready to design the upper-half routines that implement the operations *read*, *write*, and *seek*; we begin with *read*. Basically, the upper-half input routine must build a request node, insert the request in the list of pending requests, and wait until the lower-half routine indicates that the request has been honored. The code consists of procedure *dsread*, shown below in file *dsread.c*.

```
/* dsread.c - dsread */

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

/*------------------------------------------------------------------------
 *  dsread  --  read a block from a disk device
 *------------------------------------------------------------------------
 */
dsread(devptr, buff, block)
struct devsw *devptr;
char *buff;
DBADDR block;
{
        struct  dsblk   *dsptr;
        struct  dreq    *drptr;
        int     stat;
        int     ps;

        dsptr = (struct dsblk *)devptr->dvioblk;
        if ( block<0 || block>=NDSECT
                || dsptr->dsprocnum < 0
                || (drptr=(struct dreq *) getbuf(dskrbp)) == DRNULL )
                return(SYSERR);
        disable(ps);
        drptr->drdba = block;
        drptr->drpid = currpid;
        drptr->drbuff = buff;
        drptr->drop = DREAD;
        if ( (stat=dskenq(drptr, devptr->dvioblk)) == DONQ) {
                suspend(currpid);
                stat = drptr->drstat;
        }
        freebuf(drptr);
        restore(ps);
        return(stat);
}
```

Dsread allocates a request node from the buffer pool that contains all request nodes and then fills in the desired disk block address; memory buffer address; and the operation requested, ''read.'' Once the request has been specified, *dsread* calls *dskenq* to enqueue the new request in the list of pending requests and to start the server process if necessary. After the request has been enqueued, *dsread* suspends the calling process until the operation completes and the lower-half resumes it. When resumed by the lower-half, the upper-half deallocates the request block, extracts the exit status value left by the lower-half, and returns to its caller to indicate whether an error occurred.

It may seem odd that the driver is designed to allocate storage for request nodes from a single, global buffer pool (*dskrbp*). Keeping buffers in a single pool instead of dividing them among disk devices has benefits and liabilities. For example, the single buffer pool reduces the chance that a process will be blocked waiting for a request node, but it allows activity on one disk to prevent activity on another. The exercises consider other ways in which keeping a separate set of request nodes for each disk device affect the behavior of the system.

16.7 The Upper-Half Write Routine

The upper-half output driver behaves quite differently from the upper-half input driver. Instead of waiting for a request to be honored, the output driver allocates and fills in a request node, enqueues it in the list of pending requests, and returns to its caller without waiting for the request to be honored. By doing so, it assumes that the data has been placed in a buffer, and that the caller will not change the buffer between the time it calls *write* and the time the data is written.

Returning to the caller before data has been written is a violation of the general principles for input and output laid down in Chapter 9. There, we said that the system should block processes until output has been consumed. Earlier drivers adhered to this principle either by copying data into an internal buffer or by suspending the process until the data had been written. Besides violating the general design guidelines and introducing inconsistency, arranging for the driver to return before the data has been written is dangerous, because it leaves programmers susceptible to an error that is easy to make but difficult to diagnose. After a call to *write*, only the address of the buffer has been recorded; if the program continues to change the data, changes made between the call to *write* and the time the lower-half copies the data to disk will appear in the copy on disk.

If non-blocking output is so dangerous, why design the disk driver to use it? The answer is that the current design is a compromise. Suspending the caller until the operation completes will not work because disk operations are so slow that processes using disks would spend most of their time suspended. The upper-half could avoid waiting if it copied data into a buffer itself. An earlier version of the upper-half output routine did just that – it allocated a buffer and copied the data from the caller's area into the buffer before returning. It turned out, however, that the CPU spent most of its time copying data from one buffer to another. Closer examination of the code in the next layer of the file system revealed that routines calling the output driver usually deallocated the buffer

immediately after writing data, so the driver was copying data from one system buffer to another needlessly. To avoid wasting time, the driver was changed to write directly from the specified address, leaving the responsibility of buffering to the upper layers of file system software.

Is there a better alternative to driver optimization? If users were expected to call the driver routines directly, nonblocking output routines would be unacceptable. As we will see in the next chapter, only PC-Xinu file system routines call the disk driver, so we can be careful to ensure that they buffer output before calling the driver. In a system designed to support calls from both user programs and file system routines, the problem can be solved by building two upper-level interfaces for the disk driver. One interface, called only by other system routines, would provide unbuffered output like the current driver; the other, called by user programs, would copy data into buffers before returning to the caller.

16.8 Implementation Of The Upper-Half Write Routine

Procedure *dswrite*, found in file *dswrite.c*, implements the upper-half output routine described above.

```
/* dswrite.c - dswrite */

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

/*------------------------------------------------------------------------
 *  dswrite  --  write a block (system buffer) onto a disk device
 *------------------------------------------------------------------------
 */
dswrite(devptr, buff, block)
struct devsw *devptr;
char *buff;
DBADDR block;
{
        struct  dsblk   *dsptr;
        struct  dreq    *drptr;
        int     ps;

        dsptr = (struct dsblk *)devptr->dvioblk;
        if ( block<0 || block>=NDSECT
```

```
                || dsptr->dsprocnum < 0
                || (drptr=(struct dreq *) getbuf(dskrbp)) == DRNULL )
                return(SYSERR);
        disable(ps);
        drptr->drbuff = buff;
        drptr->drdba = block;
        drptr->drpid = currpid;
        drptr->drop = DWRITE;
        if ( dskenq(drptr, devptr->dvioblk) == SYSERR ) {
                freebuf(drptr);
                restore(ps);
                return(SYSERR);
        }
        restore(ps);
        return(OK);
}
```

Like the input routine, *dswrite* is straightforward. It allocates a request node from global buffer pools *dskrbp*, fills in the request block, and calls *dskenq* to add it to the list of requests.

16.9 The Upper-Half Seek Routine

The third upper-half routine, *dsseek*, implements the *seek* operation:

```
/* dsseek.c - dsseek */

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

/*------------------------------------------------------------------------
 *  dsseek  --  schedule a request to move the disk arm
 *------------------------------------------------------------------------
 */
dsseek(devptr, block)
struct devsw *devptr;
DBADDR block;
{
        struct  dsblk   *dsptr;
        struct  dreq    *drptr;
        int     ps;

        dsptr = (struct dsblk *)devptr->dvioblk;
        if ( block<0 || block>=NDSECT
                || dsptr->dsprocnum < 0
                || (drptr=(struct dreq *) getbuf(dskrbp)) == DRNULL )
                return(SYSERR);
        disable(ps);
        drptr->drdba = block;
        drptr->drpid = currpid;
        drptr->drbuff = NULL;
        drptr->drop = DSEEK;

        /* enqueued with normal policy like other read/write requests */

        dskenq(drptr, devptr->dvioblk);
        restore(ps);
        return(OK);
}
```

Dsseek operates much like *dswrite*. It allocates a request block, fills in the operation and block address fields, and calls *dskenq* to enqueue the request in the list of pending requests. Like *dswrite*, *dsseek* returns to its caller as soon as the request has been enqueued, without waiting for the operation to complete. Observe that the BIOS software interface does not provide a *seek* operation, so *seek* requests end up being ignored in PC-Xinu.

16.10 The Lower-Half Of The Disk Driver

The lower-half of the disk driver is implemented as a process. Like the tty input and output processes described in Chapter 12, the lower-half disk driver performs an infinite loop, examining the request queue and carrying out the specified operations as long as requests remain in the queue. When the queue becomes empty, the process suspends itself by waiting for a message with a call to *receive*. Upper-half routines call *dskenq* to enqueue requests, and *dskenq* sends a message to the lower-half process if it finds the queue empty.

Procedure *dsinter* implements the lower-half.

```
/* dsinter.c - dsinter */

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

/*------------------------------------------------------------------------
 *  dsinter  --  process to handle disk requests
 *------------------------------------------------------------------------
 */
PROCESS dsinter(dsptr,dsknum)
struct  dsblk   *dsptr;
int     dsknum;
{
        int     ps;
        struct  dreq    *drptr;
        int     status;

        disable(ps);
        for(;;) {
                drptr = dsptr->dreqlst;
                if ( drptr == DRNULL ) {
                        receive();
                        continue;
                }
                dsptr->dreqlst = drptr->drnext;
                switch (drptr->drop) {
                case DREAD:
                        status = dread(drptr->drbuff,dsknum,drptr->drdba);
                        if ( status != 0 )
                        kprintf("\nRead error: cde=%02xH drv=%d blk=%d\n",
                                        status,dsknum,drptr->drdba);
                        drptr->drstat = ( status ? SYSERR : OK );
                case DSYNC:
                        if ( resume(drptr->drpid) == SYSERR )
                                panic("Disk request block pid error");
                        break;
                case DWRITE:
                        status=dwrite(drptr->drbuff,dsknum,drptr->drdba);
                        if ( status != 0 )
                        kprintf("\nWrite error: cde=%02xH drv=%d blk=%d\n",
                                        status,dsknum,drptr->drdba);
                        freebuf(drptr->drbuff);
```

```
                        /* fall through */
                case DSEEK:
                        freebuf(drptr);
                }
        }
}
```

Depending on the operation just completed, *dsinter* resumes the calling process or deallocates the request node. If the operation was a *read*, it resumes the process that requested input and returns. If the operation was a *write* or *seek*, *dsinter* deallocates the request node by calling *freebuf*; in the case of a *write* it deallocates the data buffer as well, because the process that initiated the operation has already returned from the upper-half routines. *Seek* operations are simply dequeued and disposed of.

16.11 Flushing Pending Requests

Because *dswrite* does not wait for data transfer, a process cannot know when blocks have been written to disk. However, making sure that blocks have been written may be important. For example, the system may want to ensure that activity on all disk devices completes before shutdown.

To allow programs in higher layers to check that all disk transfers have occurred, the driver includes a primitive that will block the calling process until all existing requests have been performed. Because ‘‘synchronizing’’ the disk is not a data transfer operation, we will use the high-level operation *control*. To flush pending requests, a process calls

control (device, DSKSYNC)

The driver suspends the caller until all existing requests have been satisfied on the specified device; then, the call returns.

Procedure *dscntl* implements the upper-half *control* function.

```
/* dscntl.c - dscntl */

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

/*------------------------------------------------------------------------
 *  dscntl  --  control disk driver/device
 *------------------------------------------------------------------------
 */
dscntl(devptr, func, arg1, arg2)
struct  devsw   *devptr;
int     func;
char    *arg1, *arg2;
{
        int     stat;
        int     ps;

        disable(ps);
        switch (func) {

                case DSKSYNC:
                        stat = dsksync(devptr);
                        break;
                default:
                        stat = SYSERR;
                        break;
        }
        restore(ps);
        return(stat);
}
```

Synchronization is specified with an argument given by symbolic constant *DSKSYNC*. *Dscntl* invokes procedure *dsksync* to synchronize the disk; its definition can be found in file *dsksync.c*.

```
/* dsksync.c - dsksync */

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

```
#include <mark.h>
#include <bufpool.h>

/*------------------------------------------------------------------------
 *  dsksync  --  wait for all outstanding disk requests before returning
 *------------------------------------------------------------------------
 */
dsksync(devptr)
struct  devsw   *devptr;
{
        struct  dsblk   *dsptr;
        struct  dreq    *drptr, *p, *q;
        int     stat;

        dsptr = (struct dsblk *)devptr->dvioblk;
        if ( dsptr->dsprocnum < 0 )
                return(SYSERR);
        if ( (q=dsptr->dreqlst) == DRNULL )
                return(OK);
        if ( (drptr=(struct dreq *) getbuf(dskrbp)) == DRNULL )
                return(SYSERR);
        drptr->drdba = 0;
        drptr->drpid = currpid;
        drptr->drbuff = NULL;
        drptr->drop = DSYNC;
        drptr->drnext = DRNULL;

        /* place at end of request list */

        for (p=q->drnext ; p!=DRNULL ; q=p,p=p->drnext)
                ;
        q->drnext = drptr;
        drptr->drstat = SYSERR;
        suspend(currpid);
        stat = drptr->drstat;
        freebuf(drptr);
        return(stat);
}
```

Like the other upper-half drivers, *dsksync* allocates a request node and fills it. Unlike the other drivers it does not use *dskenq* to insert the request in the list. Instead, it searches the list using pointers *p* and *q* and inserts the request when *q* points to the last node. Because the block specified is 0, the request is guaranteed to go at the end of the list (the exercises discuss one of the shortcomings of this implementation). To avoid making syn-

chronization a special case, *dsksync* requests an operation that tests the disk status. When the request reaches the front of the list, the driver passes it to the lower-half server process exactly as it does for other operations.

When the ''sync'' operation completes, the lower-half awakens the process executing *dsksync* just as it awakens the process executing *dsread* when a *read* operation completes. After being awakened, the process that suspended itself returns to *dscntl* and from there to its caller.

16.12 Summary

This chapter considers the design of a low-level disk driver that implements *read*, *write*, *seek*, and *control* operations. At this level, the disk is viewed as a large array of randomly accessible data blocks; there is no notion of named files, directories, or the index techniques used to speed searching. Reading consists of copying data from a specified block on disk into memory; writing consists of copying data from memory onto a specified disk block. Whenever an operation completes, the lower-half process takes the next pending request from the queue, and performs that operation.

The driver reduces the time required to honor requests by reordering them to minimize disk arm movement. When enqueuing a request, upper-half routines insert it between two adjacent existing requests if the arm will pass over the requested block while traveling from one block to the other. Otherwise, it adds the request to the end of the list. While this heuristic does not guarantee minimum access time for a set of requests, it works well in practice.

Because copying data for each transfer requires too much CPU time, the disk driver has been designed to accept output requests, and return to the caller without copying the data into system buffers. This strategy can be dangerous because it places all responsibility for buffering on the caller. Although unbuffered output forms an efficient foundation for file system routines, it should be rewritten before being used as a general-purpose disk interface.

FOR FURTHER STUDY

The treatment of disk scheduling heuristics in Denning [1967] compares First-Come-First-Served (FCFS) and Shortest-Seek-Time-First (SSTF) along with a heuristic similar to the one described here. Teorey [1972], Wilhelm [1976], and Hofri [1980] compare FCFS with SSTF, giving an in-depth analysis.

EXERCISES

16.1 Build a synchronous output driver that waits for I/O to complete before returning to the calling process.

16.2 The upper-half tries to minimize the amount of disk arm movement by keeping the list of requests ordered by block number. Show that the algorithm is not optimum by finding a sequence of requests for which the algorithm produces a list that requires more arm movement than necessary. What is the worst possible sequence of requests that can be presented to this algorithm?

16.3 Should requests from high-priority processes take precedence over requests from low-priority processes?

16.4 Verify that the driver honors all requests for a given block in FIFO order.

16.5 Investigate other algorithms like the ''elevator'' algorithm (mentioned in Knuth [1968]) that are used to order disk requests.

16.6 Verify that a request to ''synchronize'' will not return until all pending requests have been satisfied. Is there a bound on the time it can be delayed by new requests?

16.7 Describe a sequence of operations in which a call to *dssync* does not return even though all requests that were pending when it was inserted have completed. Redesign the synchronization routine to allow the caller to wait for a specific request.

16.8 While read failures can be recognized by the caller by inspecting the *drstat* component of the request buffer upon return, write failures cannot. Investigate ways in which the caller can be informed if a write failure occurs.

16.9 Change the lower-half driver software to be interrupt-driven by accessing the PC disk controller hardware directly.