17 A Disk Driver

Introduction

- At the level of device driver routines, a disk is viewed as a **randomly accessible array of blocks** (here, of size 512 bytes).
- The interaction between driver software and the disk device itself is more complicated than for a SLU – therefore, an **extra controller** (the disk-controller) is needed with its own format of controller request records.
- A disk is a random access device, therefore, a disk must **position the disk arm** as well as **transfer data**.
- Disk hardware uses **direct-memory-access-mode (DMA)**, allowing for the **direct transfer** to/from memory blocks **without help from the CPU**. So it does not interrupt the CPU for the transfer of every single character. Hence the CPU can continue operation while a block transfer takes place.

17.1 Operations supplied by Disk Drivers

- At the device driver level, a disk is nothing more than a large array of data blocks that can be accessed **randomly** using three basic operations:
  - select a block: `seek`
  - copy the contents of the selected block from disk to memory: `read`
  - copy the contents of memory to the selected block on disk: `write`
- Format of an instruction: `read(DISKDEV, buff, blocknr)`

  This means read diskblock with number `blocknr` into memory starting at address `buff` from the disk identified by `DISKDEV`.
- As long as calls to read and write include a block position, a **separate seek operation** is unnecessary. So why is operation `seek` needed? **To optimize disk access**!
- Why is optimization necessary? Because the relation between the times taken by an execution of a memory instruction and moving a disk arm is

$$\mathbf{1} : \mathbf{300.000}!$$

Operations supplied by Disk Drivers

- 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 through 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 computations (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.

Recapitulation of §2.2: Disk Storage Organization

- Disk devices are **slow and awkward**, compared with main memory operations at speeds measured in **nanoseconds**!
- A disc access is on the order of **300,000 times slower** than a memory access. To put this in perspective, imagine that you need to retrieve a pencil before you continue some task. Suppose that it takes 10 seconds to walk to your desk (high-speed memory) and retrieve one. A retrieval operation that took 300,000 times as long would last roughly **34.7 days**. After a few such retrieval operations, you would quickly learn to save pencils in your desk drawer to avoid the trip. Operating systems use an analogous technique – they often contain **complex algorithms** that save **copies of blocks** to avoid unnecessary disk access.

Recapitulation of §2.2: Disk Storage Organization

- So **why use disks** at all? Because main memory is **volatile**, i.e., data "evaporates" when power is off, whereas disks ar **nonvolatile**, because **their data persists** when the system stops and power is off.
- Most significant **difference** with respect to **SLU**: Unlike SLU hardware which interrupts CPU for **each single character** received or transmitted, the disk interface transfers **large blocks** of data directly to or from memory, interrupting **only** when transfer has completed.
- This technique is called **Direct Memory Access (DMA)** and **dramatically increases throughput** while the CPU continues to execute instructions.

Recapitulation of §2.2.1: Disk Interface and Controller Hardware

A disk consists of:

- a **disk drive** – i.e., platters (spindle) which rotate at high speed, a physiscal arm and read/write head(s)
- an **electronic controller** (Xebec) – it contains **microprocessor(s)**, controls positioning of the disk arm and transfer of data, it also senses **error** occuring, it can operate on **multiple** disk drives, but it's not able to transfer data to/from more than one diskplatter **simultaneously**,
- a **host interface** (DTC) – it **connects** controller to system bus, i.e., it passes requests for starting an I/O operation **from processor to controller**, **interrupts the CPU** when controller signals that an operation has completed, and **passes data** between controller and main memory during data transfer.

Recapitulation of §2.2.2: The pieces of a Disk Hardware

- The disk hardware has no notion of **file** or built-in **directory**. The hardware considers each disk device as an **array ob 512-byte blocks** (called **sectors**), addressed $0, 1, \ldots, n$ (with $n = 1000 \ldots 30.000$ anno 1984!).
- Hardware reads data by copying a sector on the disk to a block in memory, and writes data by copying a block from memory onto a sector of the disk.
- Using the disk hardware consists of **passing requests to the host interface**, which carries them out and reports the results. Part of this information constitues a **request for the controller**. The other part consists of a **block to which data is to be transferred**.

Recapitulation of §2.2.3: Interface and Controller Request Formats

- The disk host interface starts at address 0177460_8 (device-`csr`-address) **beyond real memory** and has a 6 word format, including 4 essentially used words (on the 11/2).
- Xebec controller **request record** arrangement:

Byte 0	command
Byte 1	disk unit (0)
Byte 2	high byte of block-number
Byte 3	low byte of block-number
Byte 4	block count (1)
Byte 5	control (0)

- The disk driver discussed in the chapter was built for an *International Technologies, Inc., 5018H Winchester Disk Drive*, a *Data Technology Corporation DTC-11-1 host interface* and a *Xebec Systems Inc. S1410 Winchester Disk Controller*. Despite being manufactured by three vendors, these pieces of equipment work well together.

Recapitulation of §2.2.3: Interface and Controller Request Formats

`CCSR`	Completion Status Register		
`CSR`	Control & Status Register	←	see below
`DAR`	DMA Address Register	←	Buffer address for read/write
`CAR`	Control Address Request Register	←	Adress in memory of request record for the disk controller
`XDAR`	extended `DAR`, not used		
`XCAR`	extended `CAR`, not used		

The interface `CSR` register contains several bits that **report errors** and **control interrupts**:

15	14	13	12 ... 8	7	6	5 ... 2	1	0
E		P		D	I		F	G
error bit		parity error		operation completed	interrupts enabled		reset controler	go-bit

- If bit 6 (I) is set, interface will interrupt whenever an operation is completed.
- If go-bit 0 (G) is set, then I/O operation is started.

Summary of §17

1. **Low-level** disk drivers implement 4 operations (`read`, `write`, `seek` and `control`). **Upper-half** routines merely **enqueue requests for services**.

   Whenever an data-transfer-operation completes, a lower-half routines takes the **next pending request from queue**, starts up hardware performing that operation, and wakes up the process waiting for **previous operation to end**.

2. Driver reduces time to honor requests by recording them to **minimize 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. While this heuristic does not **guarantee** minimum access time for a set of requests, it **works well in practice**.

3. Because **copying data** for each transfer takes too much time, the driver accepts output requests, and returns **asynchronously** to caller – **without having coped the data** in the output buffer into the system buffers – **a dangerous strategy**, unusable for a general-purpose disk interface. However the routines developed in this chapter are used **safely**, without causing errors, inside the system procedures for **handling files** in the next chapter.

17.2 Controller Requests and Interface Register Descriptions

The Xebec Disc Controller layout is described by structure `xbdcb` on pages 284/285, corresponding to the figure of the request record layout just seen.

```
/* xebec.h */
/* Xebec S1410 5.25 inch Winchester Disk Controller */

struct      xbdcb {                /* Xebec disk controller regs.     */
      char  xop;                   /* command class/opcode            */
      char  xunit;                 /* unit/high-order address bits    */
      char  xmaddr;                /* middle-order address bits       */
      char  xladdr;                /* low-order address bits          */
      char  xcount;                /* block count                     */
      char  xcntl;                 /* control field                   */
      char  xres1;                 /* reserved field #1               */
      char  xres2;                 /* reserved field #2               */
};

/* Xebec controller operation codes */
#define     XOTDR       0          /* Test Drive Ready              */
#define     XORCAL      1          /* Recalibrate                   */
#define     XONONO      2          /* Reserved (is a no-no)         */
#define     XORSS       3          /* Request Sense Status          */
#define     XOFMTD      4          /* Format Drive                  */
#define     XOCTF       5          /* Check Track Format            */
#define     XOFMTT      6          /* Format Track                  */
#define     XOFMTBT     7          /* Format Bad Track              */
#define     XOREAD      8          /* Read                          */
#define     XONO2       9          /* Reserved (is also a no-no)    */
#define     XOWRITE    10          /* Write                         */
#define     XOSEEK     11          /* Seek                          */
#define     XOINIT     12          /* Initialize Drive Character.*/
#define     XOREBEL    13          /* Read ECC Burst Error Length*/
#define     XORAMD     14          /* Perform RAM Diagnostic Test*/
#define     XRETRY     0           /* retry control byte            */
```

Controller Requests and Interface Register Descriptions

- Notice the many operations listed in `xebec.h`! For us, only `XOREAD`, `XOWRITE` and `XOSEEK` are interesting.
- The DTC host interface parallels the figure of the **host interface device register layout**:

```
/* dtc.h */

/* Data Technology Corp. DTC-11-1 disk controller host interface */

struct  dtc     {                       /* controller interface regs.   */
        int     dt_ccsr;                /* command completion status    */
        int     dt_csr;                 /* control and status register  */
        char    *dt_dar;                /* data address register        */
        struct  xbdcb   *dt_car;        /* command address register     */
        int     dt_xdar;                /* extension of dar (not used)  */
        int     dt_xcar;                /* extension of car (not used)  */
};

/* bits in the dtc csr register */
#define DTGO    0000001                 /* "go" bit - start interface   */
#define DTRESET 0000002                 /* "force reset" bit            */
#define DTINTR  0000100                 /* enable interface interrupt   */
#define DTDONE  0000200                 /* command done                 */
#define DTERROR 0100000                 /* some error occurred          */
```

- Notice that the bits in figure 2.13 correspond to operations:

`DTGO`	–	set bit nr. 1	`DTRESET`	–	set bit nr. 2
`DTINTR`	–	set bit nr. 6	`DTDONE`	–	set bit nr. 7
`DTERROR`	–	set bit nr. 15			

17.3 The List of Pending Disk Requests

- Disk driver routines are partitioned into two sets: Upper- and Lower-half routines, communication through a **shared data structure**, the **disc control block**, which includes a **list of pending requests**.
- Upper-half routines **enqueue requests** for operations, **start** the device when it is idle, and then **wait** for the operation to complete.
- Lower-half routines are interrupt handlers, which are called when an **operation completes**, then they **start the next operation** and awaken the process waiting for the request just completed.

The List of Pending Disk Requests

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

#include <xebec.h>                      /* disk controller constants    */
#include <dtc.h>                        /* disk interface constants     */

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

struct  dsblk   {                       /* disk driver control block    */
        struct  dtc     *dcsr;          /* disk interface csr address   */
        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 */
        char    *ddir;                  /* address of in-core directory */
        struct  xbdcb   ddcb;           /* holds command that interface */
                                        /*   sends to disk controller   */
};
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*/
        char    drop;                   /* operation: READ/WRITE/SEEK   */
        int     drstat;                 /* returned status OK/SYSERR    */
        struct  dreq    *drnext;        /* ptr to next node on req. list*/
};
```

The List of Pending Disk Requests

`disk.h` continued:

```
#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    */
#ifndef NDBUFF
#define NDBUFF  10                      /* number of disk data buffers  */
#endif
#define NDREQ   10                      /* number of disk request buf.  */
#define DREAD   XOREAD                  /* read command in dreq.drop    */
#define DWRITE  XOWRITE                 /* write   "                    */
#define DSEEK   XOSEEK                  /* seek    "                    */
#define DSYNC   XOTDR                   /* sync    "  (test-disk-ready) */

extern  int     dskrbp;                 /* disk request node buffer pool*/
extern  int     dskdbp;                 /* disk data block buffer pool  */

/* disk control function codes */

#define DSKSYNC 0                       /* synchronize (flush all I/O)  */
#define dssync(ddev)    control((ddev),DSKSYNC);

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

The List of Pending Disk Requests

- For disks, the required information for transfer between upper- and lower-level routines resides in **control blocks**.
- The disk driver control block consists of an array `dstab[]`, consisting of structures `dsblk`, containing, among others:
  - an address of `CSR` (0177460_8 – taken from device switch table),
  - a list of pending requests `*dreqlist`,
  - a device number of the disk, `dnum`,
  - three `semaphores` for implementing mutual exclusion because of the possibility for concurrent access by processes to disk, caused by file operations,
  - `dnfiles` – the numer of currently open files, and
  - a 6 byte command, that the interface sends to the controller of the disk, structures `xbdcb` (for Xebec disc controller block) `ddcb`.

The List of Pending Disk Requests

- The format of each entry in the list of requests is of structure `dreq`, consisting of fields for:
  - diskblock address to use, `drdba`,
  - identifier of requesting process (because of need for notification), `drpid`,
  - buffer address for read/write, `*drbuff`,
  - operation to-be-performed, `drop`, and
  - returned status, `drstat` (`OK/SYSERR/DONQ`).
- There are two constants: `DIRBLK` (block used to hold directory) and `DONQ` (status when request is enqueued).
- The disk request-node buffer pool is accessed by `dskrbp`, the disk data block buffer pool by `dskdbp`.

The List of Pending Disk Requests

- The disk driver maintains the following invariant:

  The first request on the pending request list is always the one the hardware is performing; if the list is empty, the hardware is idle.

- The list of pending request, i.e., `dreqlst`, is a **singly linked list** consisting of structures `dreq`.

- Field `drstat` **communicates** status information from lower-half to upper-half: The lower-half records information about **errors** occuring during the operation; the upper-half extracts that information and **passes it on to the caller**.

17.4 Enqueuing disk requests

- The disk driver processes I/O requests in the order in which they occur on the **request list**. When one completes, it is **removed from the list** and the **next one** is started.
- When procedure `dskenq` adds a request for block B to the existing list of requests, it schedules 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 $(i, i+1)$ exists, `dskenq` adds this new request to the end of list.

Enqueuing disk requests

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

#include <conf.h> <kernel.h> <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;
                dskstrt(dsptr);
                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);
}
```

17.5 Optimizing the Request Queue

- In case of a **read operation** to be enqueued for which there is a **write for the same block** in the list of pending requests, the read can be **instantaneously performed**, by copying the write request buffer directly into the read request buffer, and allow the reading process to continue.
- Optimizations are also possible for double or more **writes to same block**: the first one can be discarded! This is done by procedure `dskqopt`.

Optimizing the Request Queue

```
/* 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);
        }
```

Optimizing the Request Queue

dskqopt.c continued:

```
        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, RESCHNO);
                }
                drptr->drnext = p;
                q->drnext = drptr;
                resched();
                return(OK);
        }
        return(SYSERR);
}
```

17.6 Starting a Disk Operation

- Procedure `dskstrt` **starts** a disk operation by building a **controller request structure** in the `ddcb`-field of the disc control block, feeding that request to the controller through the interface.
- Recall that the first request on that line is the one **hardware should perform**. Once finished – as indicated by an interrupt – the **second one** is started, and the process waiting for termination of that first request is notified.

Starting a Disk Operation

```
/* dskstrt.c - dskstrt */

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

/*------------------------------------------------------------------------
 *  dskstrt  --  start an I/O operation on a disk device
 *------------------------------------------------------------------------
 */
dskstrt(dsptr)
        struct  dsblk   *dsptr;
{
        struct  xbdcb   *xptr;
        struct  dtc     *dtptr;
        struct  dreq    *drptr;

        /* build command for controller */

        drptr = dsptr->dreqlst;
        xptr = & dsptr->ddcb;
        xptr->xop    = (char) drptr->drop;              /* opcode       */
        xptr->xunit  = (char) 0;                        /* top addr bits*/
        xptr->xmaddr = (char) ((drptr->drdba>>8)&0377); /* mid addr bits*/
        xptr->xladdr = (char) (drptr->drdba & 0377);    /* low addr bits*/
        xptr->xcount = (char) 1;                        /* num of blocks*/
        xptr->xcntl  = (char) XRETRY;                   /* retry code   */

        /* feed command to controller through interface */

        dtptr = dsptr->dcsr;
        dtptr->dt_dar = drptr->drbuff;
        dtptr->dt_car = xptr;
        dtptr->dt_xdar = dtptr->dt_xcar = 0;
        dtptr->dt_csr = DTINTR | DTGO;
}
```

17.7 Driver Initialization

- Procedure `dsinit`
  - fills in the **disk control block**,
  - fills in the corresponding **interrupt vectors**, and the corresponding entries into the **interrupt dispatch table** by calling

    `iosetvec(devptr->dvnum, dsptr, dsptr)`,

  - sets up a read command for the **disk directory** to be read, and
  - then **starts** the interface up.
- Because the system executes the initialization routine **before interrupts are enabled**, `dsinit` must **poll** to determine when the read operation completes!

Driver Initialization

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

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

#ifdef  Ndsk
struct  dsblk   dstab[Ndsk];
#endif
int     dskdbp, dskrbp;

/*------------------------------------------------------------------------
 *  dsinit  --  initialize disk drive device
 *------------------------------------------------------------------------
 */
dsinit(devptr)
        struct  devsw   *devptr;
{
        struct  dsblk   *dsptr;
        struct  dtc     *dtptr;
        int     status;
        char    ps;

        disable(ps);
        devptr->dvioblk = dsptr = &dstab[ devptr->dvminor ];
        dsptr->dcsr    = devptr->dvcsr;
        dsptr->dreqlst = DRNULL;
        dsptr->dnum    = devptr->dvnum;
        dsptr->dibsem  = screate(1);
        dsptr->dflsem  = screate(1);
        dsptr->ddirsem = screate(1);
        dsptr->dnfiles = 0;
        dsptr->ddir    = getbuf(dskdbp);
        iosetvec(devptr->dvnum, dsptr, dsptr);
```

Driver Initialization

- `dvioblk` should contain address or control block
- `dcsr` should contain address of host interface
- `dreqlst` is `DRNULL`!
- `dnum` should contain device number of *this* disk
- `dibsem` contains semaphore dealing with access to the **index block list** in the file system
- `dflsem` contains semaphore dealing with access to the **free list of data blocks** in the file system
- `ddirsem` contains semaphore dealing with **directory access** in the file system
- `dnfiles` is the number of open files, initially 0
- `ddir` is the address for **incore memory** for the data directory: `getbuf(dskdbp)` with `dskdbp` being the disk data block buffer pool
- `iosetvec(..., ..., ...)` initializes the interrupt dispatch table and interrupt vectors

Driver Initialization

`dsinit.c` continued:

- Next the Xebec controller interface structure for a **read operation** is initialized. The host interface is initialized and the interface is started setting the **go**-bit.
- The procedure goes in a while loop, checking wheterh the operation is not yet completed by polling the `DTDONE`-bit, until it is 1.

```
        /* read directory block: setup read command then start interface */

        dsptr->ddcb.xop    = (char) XOREAD;
        dsptr->ddcb.xunit  = (char) 0;
        dsptr->ddcb.xcntl  = (char) XRETRY;
        dsptr->ddcb.xmaddr = (char) ((DIRBLK>>8)&(0377));
        dsptr->ddcb.xladdr = (char) (DIRBLK&0377);
        dsptr->ddcb.xcount = (char) 1;
        dtptr = dsptr->dcsr;
        dtptr->dt_dar   = dsptr->ddir;
        dtptr->dt_car   = &dsptr->ddcb;
        dtptr->dt_xdar  = dtptr->dt_xcar = NULL;
        dtptr->dt_csr   = DTGO;
        while ( ( (status=dtptr->dt_csr) & DTDONE) == 0)
                ;
        if (status & DTERROR)
                panic("Disk error");
        restore(ps);
        return(OK);
}
```

17.8 Upper-Half Read Routine

The upper-half input routine:

- builds a **request node**,
- inserts that node in **pending request list**,
- waits until lower-half routine signals **completion** of that request,
- then **starts up** the suspended process making that request, and
- deallocates the used request block to free memory.

Upper-Half Read Routine

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

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

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

        disable(ps);
        drptr = (struct dreq *) getbuf(dskrbp);
        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);
}
```

17.9 Upper-Half Output Routine

- The upper-half output routine is **asynchronous**!
- I.e., the operation **does not wait until output is completed** because of considerations of **speed**.

! **Danger: One shouldn't overwrite the to be outputted buffer until the operation is completed!**

- Indeed, this is done in XINU: **Only** XINU **file system routines** call disk driver, taking care that output is buffered **before** calling the driver.

17.10 Implementation of the Upper-Half Output Routine

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

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

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

        disable(ps);
        drptr = (struct dreq *) getbuf(dskrbp);
        drptr->drbuff = buff;
        drptr->drdba = block;
        drptr->drpid = currpid;
        drptr->drop = DWRITE;
        dskenq(drptr, devptr->dvioblk);
        restore(ps);
        return(OK);
}
```

17.11 The Upper-Half Seek Routine

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

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

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

        disable(ps);
        drptr = (struct dreq *) getbuf(dskrbp);
        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);
}
```

17.12 Lower-Half Disk Driver Routine

- The procedure `dsinter` is called whenever device interface interrupts to signal **completion of an operation**.
- Then `dsinter` **removes the first request** from the head of pending request list (the operation that just finished) and **starts up** the next operation if one exists.
- `dsinter` also disposes of the request node just removed.
- If just-satisfied request was input request, `dsinter` awakens/unsuspends the reading process (allowing it to return to its caller). Otherwise, it deallocates request node itself because the operation was `WRITE` (in this case it also deallocates data buffer) or `SEEK`.

Lower-Half Disk Driver Routine

```
/* dsinter.c - dsinter */
#include <conf.h> <kernel.h> <disk.h>

/*------------------------------------------------------------------------
 *  dsinter  --  process disk interrupt (DTC interface; XEBEC controller)
 *------------------------------------------------------------------------
 */
INTPROC dsinter(dsptr)
        struct  dsblk   *dsptr;
{
        struct  dtc     *dtptr;
        struct  dreq    *drptr;

        dtptr = dsptr->dcsr;
        drptr = dsptr->dreqlst;
        if (drptr == DRNULL) {
                panic("Disk interrupt when disk not busy");
                return;
        }
        if (dtptr->dt_csr & DTERROR)
                drptr->drstat = SYSERR;
        else
                drptr->drstat = OK;
        if ( (dsptr->dreqlst=drptr->drnext) != DRNULL)
                dskstrt(dsptr);
        switch (drptr->drop) {

                case DREAD:
                case DSYNC:
                        ready(drptr->drpid, RESCHYES);
                        return;
                case DWRITE:
                        freebuf(drptr->drbuff);
                        /* fall through */
                case DSEEK:
                        freebuf(drptr);
        }
}
```

17.13 Flushing pending Requests

- Because write operations are treated **asynchronously**, at **system shutdown** all **pending requests** must be finished before system can be shutdown, indeed.
- This is done by calling the high-level routine `CONTROL(device, DSKSYNC)`. This operation **suspends the caller** until all existing requests have been satisfied. This is implemented by `dskcntl(..., ...)`, which calls `dsksync(devptr)`.
- All `dskcntl` does is to put request node with operation `DSYNC` at the **end of the request list**; suspends `currpid` and waits till it is started up again (by `dsinter`).
- Then it checks the status of the job just completed, frees the node and returns `status`.

Flushing pending Requests

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

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

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

        disable(ps);
        switch (func) {

                case DSKSYNC:
                        stat = dsksync(devptr);
                        break;

                default:
                        stat = SYSERR;
                        break;
        }
        restore(ps);
        return(stat);
}
```

Flushing pending Requests

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

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

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

        if ( (q=(devptr->dvioblk)->dreqlst) == DRNULL )
                return(OK);
        drptr = (struct dreq *) getbuf(dskrbp);
        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);
}
```