Lion's Chapters 16 & 17

By Pavel Babenko and Michael Buchoff

The RK Disk Driver

Introduction

- RK disk storage consists of
  - disk controller RK11-D
  - a number of RK disk drives, up to eight for each disk controller
  - Removable disk cartridges
- This disk storage is most used in PDP11 systems

Disk format

- Surfaces: 2
- Tracks/surface: 200
- Sectors/track: 12
- Bytes/Sector: 512
- Total of 2.4M Capacity

RK11 Hardware controller

- Contains total of 7 hardware registers

Disk address	in
Word count	in
Bus address	in
Control status	in/out
Drive status	out
Error	out
Data buffer	in/out

Hardware registers

- Disk address (in) – address of block on disk to read/write
- Word count (in) – number of 2-bytes to read/write
- Bus address (in) – memory location for data used in read/write operations

Hardware registers (cont.)

- Control status register
  - (in) Operation flags
    - The type of operation: read/write/reset
    - Generate interrupt upon completion or not
    - Operation start bit
  - (out) Status information
    - Ready flag
    - Error flag

Hardware registers (cont.)

There are two more registers used by UNIX:

- Drive status register holds information on drive condition after operation
- Error status register holds drive error code

In case of error, UNIX prints error status and drive status

PDP11 memory model

- In PDP11 hardware registers use the same address space as memory
- For example, memory may be located at addresses 0-010000 and hardware registers located at 012000
- Hardware registers for RK11-D controller are located at memory address base of 0177400 (octadecimal)

UNIX mounted device concept

- After removable cartridge is plugged in, UNIX operator has to ‘mount’ this drive with mount command.
- After this, drive gets device number and device parameter records. Files and directories on drive are linked to some UNIX filesystem subdirectory.
- Drive has to be dismounted after use.

RK driver software model

- Each UNIX block device has a queue of pending IO operations. Each operation is defined by one IO buffer
- Each block device has associated devtab structure

```
struct devtab
{
  char d_active;
  struct buf *d_actf;
  struct buf *d_actl;
  …
}
```

d_active indicates if device is currently busy

d_actf and d_actl point to begin and end of pending IO queue

Overview of IO buffer structure

- IO buffer contains
  - Device name
  - Memory address to read/write data
  - Number of bytes to read/write
  - Number of block on device to access
  - Operation flags: operation type (read/write), whether operation asynchronous or not etc.
  - Pointer to the next buffer in queue

IO operation start: rkstrategy()

- Function rkstrategy(buf *bp) starts IO operation
- It adds buffer bp to IO queue
- Then it checks if device is now busy
- If it is busy, it does nothing
- If it is not, it starts device operation with rkstart() function

```
rkstrategy(abp)
struct buf *abp;
{
  register struct buf * bp;
  bp = abp;
  ...

  /* add buffer bp to IO queue */
  if (rktab.d_actf == 0)
    rktab.d_actf = bp;
  else
    rktab.d_actl->av_forw = bp;
  rktab.d_actl = bp;

  /* start device operation */
  if (rktab.d_active == 0)
    rkstart();
}
```

rkstart() and devstart() functions

- rkstart() checks if there are IO operations in queue
- If yes, rkstart() sets busy flag and calls devstart()
- devstart() is responsible for executing one IO operation.
  - It takes next buffer from IO queue.
  - It loads controller registers with data from buffer.
  - It sets interrupt flag of status register.
  - It sets GO flag of status register =>
  - Hardware begins executing this IO operation

```
#define RKADDR 0177400

struct {
  int rkds;
  int rker;
  int rkcs;
  int rkwc;
  int rkba;
  int rkda;
}

rkstart()  {
  register struct buf * bp;

  /* if queue is empty then return */
  if ((bp = rktab.d_actf) == 0)
    return;

  /* set busy flag and call devstart() */
  rktab.d_active ++;
  devstart(bp, &RKADDR->rkda, rkaddr(bp), 0);
}

/* rkaddr Is an auxiliary function that, given linear block number on disk, returns coded
   sectornumber/track number information in device format */
```

```
devstart(bp, devloc, devblk, hbcom)
struct buf *bp;
int *devloc;
{
  register int *dp;
  register struct buf *rbp;
  register int com;

  dp = devloc;                  /* contains the upper, rkda port number */
  rbp = bp;                     /* rbp now points to the next buffer */
  *dp = devblk;                 /* track/sector number sent to device */
  *--dp = rbp->b_addr;          /* memory address sent to device */
  *--dp = rbp->b_wcount;        /* number of bytes to transfer sent to device */

  com = (hbcom << 8) | IENABLE | GO | ((rbp->b_xmem & 03) << 4);
  if (rbp->bflags & B_READ)
    com =| RCOM;
  else
    com =| WCOM;
  *--dp = com;                  /* status register is set, operation is started! */
}
```

IO operation in progress

- Data is moved by RK controller directly from/to specified memory address.
- Process that requested this IO operation has option to wait for completion in either synchronous or asynchronous mode
- If it had chosen synchronous mode, it was put to sleep by high-level file system driver
- Mode is specified by ‘flag’ field of IO buffer

Operation completed

- UNIX function devstart() always requests interrupt to occur upon completion of IO operation
- RK hardware uses interrupt vector 220 from interrupt vector table
- RK interrupts are handled by function rkintr()

rkintr() function

- clears busy flag from device
- checks for IO errors
- If error happened, it executes the same IO operation for up to 10 times
- Otherwise, it removes used buffer from IO queue and calls iodone() function on removed buffer
- Then, it AGAIN calls rkstart() to process all other IO requests left in IO queue

```
rkintr()
{
    register struct buf *bp;

   if (rktab.d_active == 0)          /* if operation is not started, exit */
     return;

   bp = rktab.d_actf;                /* bp points to executed buffer */
   rktab.d_active = 0;               /* clear busy flag */

   if (RKADDR->rkcs < 0) {           /* if error bit is set */
     ...
     RKADDR->rkcs = RESET | GO;
     ...
     if (++rktab.d_errcnt <= 10) { /* try to repeat faulty operation up to 10 times */
       rkstart();
       return;
     }
     bp->b_flags =| B_ERROR; /* if still experiencing an error, give it up */
   }
   rktab.d_errcnt = 0;               /* clear error repeat count flag */

   rktab.d_actf = bp->av_forw;   /* remove this IO buffer from queue */
   iodone(bp);                   /* do some postprocessing on removed buffer */
   rkstart();                    /* start operation again, for the next buffer */
}
```

iodone() function

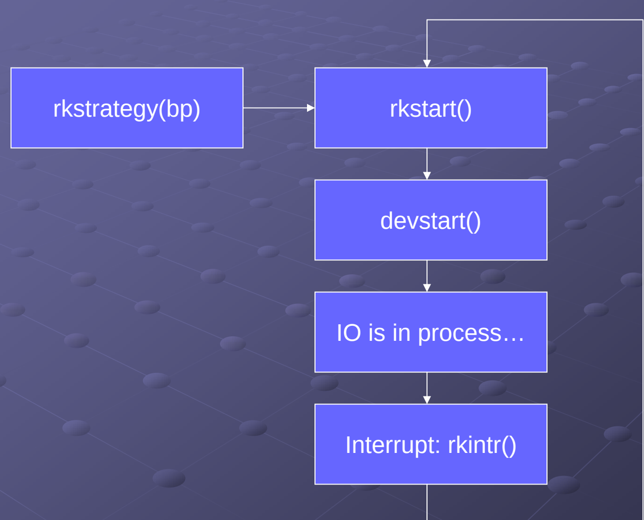
- If operation was asynchronous, it releases used buffer by adding it to unused buffers list
- If operation was synchronous, it awakes the process that requested the operation
- If not, the process is responsible for releasing the buffer

```
iodone(bp)
struct buf *bp;
{
  register struct buf *rbp;

  rbp = bp;                    /* rbp is a buffer which operation is finished */
  ...
  rbp->b_flags =| B_DONE;      /* mark operation as done */

  if (rbp->b_flags & B_ASYNC)
    brelse(rbp);               /* in asynchronous mode, release buffer */
  else {
    rbp->b_flags =& ~B_WANTED;
    wakeup(rbp);               /* in synchronous mode, wakeup a process */
  }
}
```

Operation flowchart



Chapter 17

IO buffers

Chapter 17 - Buffers

- buf structure
- Buffer functions
  - clrbuf
  - incore
  - getblk
  - Init
  - bread, bwrite, bflush

What is a buffer?

A buffer is an area of memory used for storing messages

How are buffers useful?

Example: We wish to write 5 bytes to a disk.

Method 1:

Method 2 (using buffers):

Major drawback of buffers

- Heavy memory requirements
- (Although a few buffers no big deal, a few hundred is)

Programming newbie – 1 buffer for every possible use

Experienced UNIX programmers – Have a pool of buffers ready for arbitrary use

(av-list)

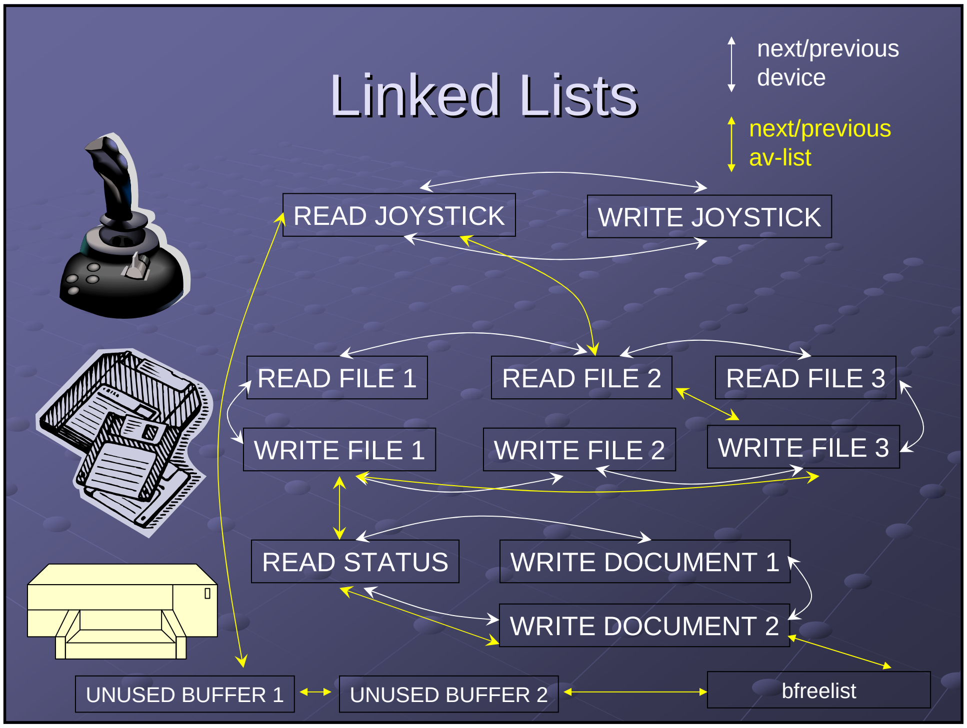
How are do we implement a buffer?

- Technically, we need nothing more than a chunk of data and its length.

- We will use chunks of 514 bytes

4720 char buffers[NBUF][514]

(Where NBUF = 15)

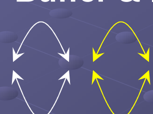


The complicated part

- Each buffer will have a header

struct buf
{

Buffer & length



Double-linked
lists

Flags (reading, writing, etc.)
Which device and where on device
Error information

} buf[NBUF];

The complicated part

- Each buffer will have a header

```
struct buf
{
        int b_flags;             /* see defines below */
        struct buf *b_forw;      /* headed by devtab of b_dev */
        struct buf *b_back;      /* " */
        struct buf *av_forw;     /* position on free list, */
        struct buf *av_back;     /*        if not BUSY */
        int b_dev;               /* major+minor device name */
        int b_wcount;            /* transfer count (usu. words) */
        char *b_addr;            /* low order core address */
        char *b_xmem;            /* high order core address */
        char *b_blkno;           /* block # on device */
        char *b_error;           /* returned after I/O */
        char *b_resid;           /* words not transferred after
                                          error */
} buf[NBUF];
```

clrbuf(struct buf *bp)

- Clears out the first 512 bytes (256 words) of the buffer

clrbuf(struct buf *bp)

```
clrbuf (bp)
int *bp;
{
        register *p;
        register c;

        p = bp->b_addr;
        c = 256;
        do
                *p++ = 0;
        while (--c);
}
```

```
struct buf
{
…
    char *b_addr;
…
}
```

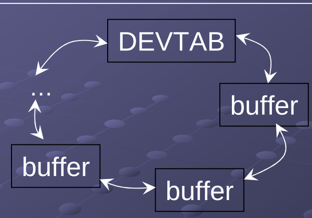
incore(adev, char *blkno)

- Searches for a buffer with a matching device number of *adev* and a block of *blkno*.
- Returns the buffer if it finds a match
- Else, returns 0

incore(adev, char *blkno)

```
incore(adev, blkno)
{
    register int dev;
    register struct buf *bp;
    register struct devtab *dp;

    dev = adev;
    dp = bdevsw[adev.d_major].d_tab;
    for (bp = dp->b_forw;
         bp != dp;
         bp = bp->b_forw)
            if (bp->b_blkno==blkno && bp->b_dev == dev)
                    return(bp);
    return(0);
}
```

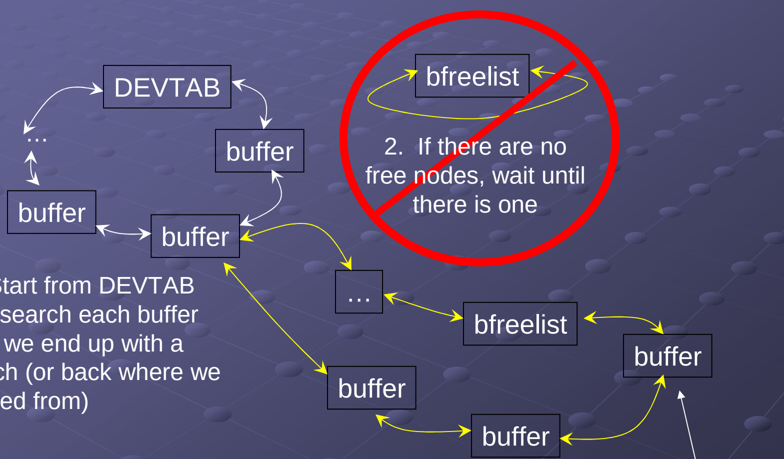


Start from DEVTAB and search each buffer until we end up with a match (or back where we started from)

getblk(dev, char *blkno)

- If there is a match, return it.
- Otherwise, search for the oldest non-busy block and allocate it.

getblk(dev, char *blkno)



1. Start from DEVTAB and search each buffer until we end up with a match (or back where we started from)

3. Use the next free buffer. (this one)

getblk(dev, char *blkno)

```
getblk(dev, blkno)
{
    register struct buf *bp;
    register struct devtab *dp;
    extern lbolt;

    if(dev.d_major >= nblkdev)
        dp = &bfreelist;

     loop:
    if (dev < 0)
        dp = &bfreelist;
    else {
        dp = bdevsw[dev.d_major].d_tab;
        if(dp == NULL)
            panic("devtab");
        for (bp = dp->forw; bp != dp; bp = bp->b_forw) {
            if (bp->b_blkno!=blkno || bp->b_dev!=dev)
                continue;
            sp16();
            if (bp->b_flags&B_BUSY) {
                bp->b_flags =| B_WANTED;
                sleep(bp, PRIBIO);
                sp10();
                goto loop;
            }
            sp10();
            notavail(bp);
            return(bp);
        }
    }
    sp16();
    if (bfreelist.av_forw == &bfreelist) {
        bfreelist.b_flags =| B_WANTED
        sleep(&bfreelist, PRIBIO);
        sp10();
        goto loop
    }
    sp10();
    notavail(bp = bfreelist.av_forw);
    if (bp->b_flags & B_DELWRI) {
        bp->b_flags =| B_ASYNC
        bwrite(bp);
        goto loop;
    }
    bp->b_flags = B_BUSY | B_RELOC;
    bp->b_back->b_forw = bp->b_forw;
    bp->b_forw->b_back = bp->b_back;
    bp->b_forw = dp->b_forw;
    bp->b_back = dp->forw;
    dp->b_forw->b_back = dp;
    dp->b_forw = bp;
    bp->b_dev = dev;
    bp->b_blkno = blkno;
    return(bp);
}
```

getblk(dev, char *blkno)

```
getblk(dev, blkno)
{
…
    loop:
…
        dp = bdevsw[dev.d_major].d_tab;
…
        for (bp = dp->forw; bp != dp; bp = bp->b_forw) {
            if (bp->b_blkno!=blkno || bp->b_dev!=dev)
                continue;
…
            if (bp->b_flags&B_BUSY) {
                bp->b_flags =| B_WANTED;
                sleep(bp, PRIBIO);
…
                goto loop;
            }
…
            notavail(bp);
            return(bp);
        }
…
```

Code will return here every time something changes

Get the devtab from the device

Iterate through the circularly linked list

If there is no match, check the next one

If the match we found is busy, mark it as wanted and sleep until something has happened to it. Then try again.

Mark it as ours and return

getblk(dev, char *blkno)

```
    if (bfreelist.av_forw == &bfreelist) {
        bfreelist.b_flags =| B_WANTED
        sleep(&bfreelist, PRIBIO);
…
        goto loop
    }
…
    notavail(bp = bfreelist.av_forw);
    if (bp->b_flags & B_DELWRI) {
        bp->b_flags =| B_ASYNC
        bwrite(bp);
        goto loop;
    }
    bp->b_flags = B_BUSY | B_RELOC;
    bp->b_back->b_forw = bp->b_forw;
    bp->b_forw->b_back = bp->b_back;
    bp->b_forw = dp->b_forw;
    bp->b_back = dp->forw;
    dp->b_forw->b_back = dp;
    dp->b_forw = bp;
    bp->b_dev = dev;
    bp->b_blkno = blkno;
    return(bp);
}
```

If there are no free elements, tell the OS we want one and sleep until it becomes available. Then, try again.

Mark the free element as ours.

If we need to write out, go ahead and try again.

Mark this flag as busy and unused.

Remove this from the av-list and insert it into the device linked list

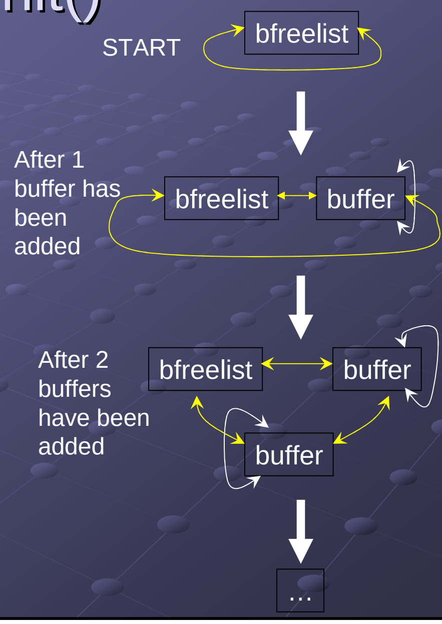
Set the device information

Return the new buffer

binit()

```
binit()
{
        register struct buf *dp;
        register struct devtab *dp;
        register int i;
        struct bdevsw *bdp;

        bfreelist.b_forw = bfreelist.b_back =
          bfreelist.av_forw = bfreelist.av_back = &bfreelist;
        for (i = 0; i<NBUF; i++) {
                bp = &buf[i];
                bp->b_dev = -1;
                bp->b_addr = buffers[i];
                bp->b_back = &bfreelist;
                bp->b_forw = bfreelist.b_forw;
                bfreelist.b_forw->b_back = bp;
                bfreelist.b_forw = bp;
                bp->b_flags = B_BUSY;
                brelse(bp);
        }
        i = 0;
        for (bdp = bdevsw; bdp->d_open; bdp++) {
                dp = bdp->d_tab;
                if(dp) {
                                dp->b_forw = dp;
                                dp->b_back = dp;
                }
                i++;
        }
        nblkdev = i;
}
```



Other Functions

- bread (buffer read, not the food) – Pass it a device number and an address in that device, it will read it and return the buffer
- bwrite (buffer write) – Pass it a buffer and it writes it out to the device
- bflush (buffer flush) – Pass it a device and it writes out all the buffers

Conclusions

- RK disk driver, devtab structure
  - rkstrategy, rkstart, devstart, rkintr, iodone
- buf structure
- Buffer functions
  - clrbuf
  - incore
  - getblk
  - Init
  - bread, bwrite, bflush

Works Cited

http://www.dictionary.com

http://www.snopes.com/common/computer/disk.gif

http://minnie.tuhs.org/UnixTree/V6/usr/sys/dmr/bio.c.html

Lions' Commentary on UNIX 6th Edition (With Source Code)


Questions?