```
#
#include "../param.h"
#include "../systm.h"
#include "../filsys.h"
#include "../conf.h"
#include "../buf.h"
#include "../inode.h"
#include "../user.h"

/*
 * iinit is called once (from main)
 * very early in initialization.
 * It reads the root's super block
 * and initializes the current date
 * from the last modified date.
 *
 * panic: iinit -- cannot read the super
 * block. Usually because of an IO error.
 */
iinit()
{
        register *cp, *bp;

        (*bdevsw[rootdev.d_major].d_open)(rootdev, 1);
        bp = bread(rootdev, 1);
        cp = getblk(NODEV);
        if(u.u_error)
                panic("iinit");
        bcopy(bp->b_addr, cp->b_addr, 256);
        brelse(bp);
        mount[0].m_bufp = cp;
        mount[0].m_dev = rootdev;
        cp = cp->b_addr;
        cp->s_flock = 0;
        cp->s_ilock = 0;
        cp->s_ronly = 0;
        time[0] = cp->s_time[0];
        time[1] = cp->s_time[1];
}

/*
 * alloc will obtain the next available
 * free disk block from the free list of
 * the specified device.
 * The super block has up to 100 remembered
 * free blocks; the last of these is read to
 * obtain 100 more . . .
 *
 * no space on dev x/y -- when
 * the free list is exhausted.
 */
alloc(dev)
{
        int bno;
        register *bp, *ip, *fp;
```

mounts the root of the file system, initializes the system time

every mount a block returns

```
        fp = getfs(dev);
        while(fp->s_flock)
                sleep(&fp->s_flock, PINOD);
        do {
                if(fp->s_nfree <= 0)
                        goto nospace;
                bno = fp->s_free[--fp->s_nfree];
                if(bno == 0)
                        goto nospace;
        } while (badblock(fp, bno, dev));
        if(fp->s_nfree <= 0) {
                fp->s_flock++;
                bp = bread(dev, bno);
                ip = bp->b_addr;
                fp->s_nfree = *ip++;
                bcopy(ip, fp->s_free, 100);
                brelse(bp);
                fp->s_flock = 0;
                wakeup(&fp->s_flock);
        }
        bp = getblk(dev, bno);
        clrbuf(bp);
        fp->s_fmod = 1;
        return(bp);

nospace:
        fp->s_nfree = 0;
        prdev("no space", dev);
        u.u_error = ENOSPC;
        return(NULL);
}

/*
 * place the specified disk block
 * back on the free list of the
 * specified device.
 */
free(dev, bno)
{
        register *fp, *bp, *ip;

        fp = getfs(dev);
        fp->s_fmod = 1;
        while(fp->s_flock)
                sleep(&fp->s_flock, PINOD);
        if (badblock(fp, bno, dev))
                return;
        if(fp->s_nfree <= 0) {
                fp->s_nfree = 1;
                fp->s_free[0] = 0;
        }
        if(fp->s_nfree >= 100) {
                fp->s_flock++;
                bp = getblk(dev, bno);
                ip = bp->b_addr;
                *ip++ = fp->s_nfree;
```

chase block pointer

```
                bcopy(fp->s_free, ip, 100);
                fp->s_nfree = 0;
                bwrite(bp);
                fp->s_flock = 0;
                wakeup(&fp->s_flock);
        }
        fp->s_free[fp->s_nfree++] = bno;
        fp->s_fmod = 1;
}

/*
 * Check that a block number is in the
 * range between the I list and the size
 * of the device.
 * This is used mainly to check that a
 * garbage file system has not been mounted.
 *
 * bad block on dev x/y -- not in range
 */
badblock(afp, abn, dev)
{
        register struct filsys *fp;
        register char *bn;

        fp = afp;
        bn = abn;
        if (bn < fp->s_isize+2 || bn >= fp->s_fsize) {
                prdev("bad block", dev);
                return(1);
        }
        return(0);
}

/*
 * Allocate an unused I node
 * on the specified device.
 * Used with file creation.
 * The algorithm keeps up to
 * 100 spare I nodes in the
 * super block. When this runs out,
 * a linear search through the
 * I list is instituted to pick
 * up 100 more.
 */
ialloc(dev)
{
        register *fp, *bp, *ip;
        int i, j, k, ino;

        fp = getfs(dev);
        while(fp->s_ilock)
                sleep(&fp->s_ilock, PINOD);
loop:
        if(fp->s_ninode > 0) {
                ino = fp->s_inode[--fp->s_ninode];
                ip = iget(dev, ino);
```

```
                if (ip==NULL)
                        return(NULL);
                if(ip->i_mode == 0) {
                        for(bp = &ip->i_mode; bp < &ip->i_addr[8];)
                                *bp++ = 0;
                        fp->s_fmod = 1;
                        return(ip);
                }
                /*
                 * Inode was allocated after all.
                 * Look some more.
                 */
                iput(ip);
                goto loop;
        }
        fp->s_ilock++;
        ino = 0;
        for(i=0; i<fp->s_isize; i++) {
                bp = bread(dev, i+2);
                ip = bp->b_addr;
                for(j=0; j<256; j+=16) {
                        ino++;
                        if(ip[j] != 0)
                                continue;
                        for(k=0; k<NINODE; k++)
                        if(dev==inode[k].i_dev && ino==inode[k].i_number)
                                goto cont;
                        fp->s_inode[fp->s_ninode++] = ino;
                        if(fp->s_ninode >= 100)
                                break;
                cont:;
                }
                brelse(bp);
                if(fp->s_ninode >= 100)
                        break;
        }
        fp->s_ilock = 0;
        wakeup(&fp->s_ilock);
        if (fp->s_ninode > 0)
                goto loop;
        prdev("Out of inodes", dev);
        u.u_error = ENOSPC;
        return(NULL);
}

/*
 * Free the specified I node
 * on the specified device.
 * The algorithm stores up
 * to 100 I nodes in the super
 * block and throws away any more.
 */
ifree(dev, ino)
{
        register *fp;
```

lock ilist

```
        fp = getfs(dev);
        if(fp->s_ilock)
                return;
        if(fp->s_ninode >= 100)
                return;
        fp->s_inode[fp->s_ninode++] = ino;
        fp->s_fmod = 1;
}

/*
 * getfs maps a device number into
 * a pointer to the incore super
 * block.
 * The algorithm is a linear
 * search through the mount table.
 * A consistency check of the
 * in core free-block and i-node
 * counts.
 *
 * bad count on dev x/y -- the count
 *      check failed. At this point, all
 *      the counts are zeroed which will
 *      almost certainly lead to "no space"
 *      diagnostic
 * panic: no fs -- the device is not mounted.
 *      this "cannot happen"
 */
getfs(dev)
{
        register struct mount *p;
        register char *n1, *n2;

        for(p = &mount[0]; p < &mount[NMOUNT]; p++)
        if(p->m_bufp != NULL && p->m_dev == dev) {
                p = p->m_bufp->b_addr;
                n1 = p->s_nfree;
                n2 = p->s_ninode;
                if(n1 > 100 || n2 > 100) {
                        prdev("bad count", dev);
                        p->s_nfree = 0;
                        p->s_ninode = 0;
                }
                return(p);
        }
        panic("no fs");
}

/*
 * update is the internal name of
 * 'sync'. It goes through the disk
 * queues to initiate sandbagged IO;
 * goes through the I nodes to write
 * modified nodes; and it goes through
 * the mount table to initiate modified
 * super blocks.
 */
```

sys sync; also called from panic()

```
update()
{
        register struct inode *ip;
        register struct mount *mp;
        register *bp;

        if(updlock)
                return;
        updlock++;
        for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
                if(mp->m_bufp != NULL) {
                        ip = mp->m_bufp->b_addr;
                        if(ip->s_fmod==0 || ip->s_ilock!=0 ||
                           ip->s_flock!=0 || ip->s_ronly!=0)
                                continue;
                        bp = getblk(mp->m_dev, 1);
                        ip->s_fmod = 0;
                        ip->s_time[0] = time[0];
                        ip->s_time[1] = time[1];
                        bcopy(ip, bp->b_addr, 256);
                        bwrite(bp);
                }
        for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
                if((ip->i_flag&ILOCK) == 0) {
                        ip->i_flag =| ILOCK;
                        iupdat(ip, time);
                        prele(ip);
                }
        updlock = 0;
        bflush(NODEV);
}
```

1 — super block

prele(ip); ← unlock it

bflush(NODEV); ← updates cache

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../proc.h"

#define UMODE   0170000
#define SCHMAG  10

/*
 * clock is called straight from
 * the real time clock interrupt.
 *
 * Functions:
 *      reprime clock
 *      copy *switches to display
 *      implement callouts
 *      maintain user/system times
 *      maintain date
 *      profile
 *      tout wakeup (sys sleep)
 *      lightning bolt wakeup (every 4 sec)
 *      alarm clock signals
 *      jab the scheduler
 */
clock(dev, sp, r1, nps, r0, pc, ps)
{
        register struct callo *p1, *p2;
        register struct proc *pp;
        int a;

        /*
         * restart clock
         */

        *lks = 0115;

        /*
         * display register
         */

        display();

        /*
         * callouts
         * if none, just return
         * else update first non-zero time
         */

        if(callout[0].c_func == 0)
                goto out;
        p2 = &callout[0];
        while(p2->c_time<=0 && p2->c_func!=0)
                p2++;
        p2->c_time--;
```

```
        /*
         * if ps is high, just return
         */

        if((ps&0340) != 0)
                goto out;

        /*
         * callout
         */

        spl5();
        if(callout[0].c_time <= 0) {
                p1 = &callout[0];
                while(p1->c_func != 0 && p1->c_time <= 0) {
                        (*p1->c_func)(p1->c_arg);
                        p1++;
                }
                p2 = &callout[0];
                while(p2->c_func = p1->c_func) {
                        p2->c_time = p1->c_time;
                        p2->c_arg = p1->c_arg;
                        p1++;
                        p2++;
                }
        }

        /*
         * lightning bolt time-out
         * and time of day
         */

out:
        if((ps&UMODE) == UMODE) {
                u.u_utime++;
                if(u.u_prof[3])
                        incupc(pc, u.u_prof);
        } else
                u.u_stime++;
        pp = u.u_procp;
        if(++pp->p_cpu == 0)
                pp->p_cpu--;
        if(++lbolt >= HZ) {
                if((ps&0340) != 0)
                        return;
                lbolt -= HZ;
                if(++time[1] == 0)
                        ++time[0];
                spl1();
                if(time[1]==tout[1] && time[0]==tout[0])
                        wakeup(tout);
                if((time[1]&03) == 0) {
                        runrun++;
                        wakeup(&lbolt);
                }
                for(pp = &proc[0]; pp < &proc[NPROC]; pp++)
```

```
                if (pp->p_stat) {
                        if(pp->p_time != 127)
                                pp->p_time++;
                        if(pp->p_clktim)
                                if(--pp->p_clktim == 0)
                                        psignal(pp, SIGCLK);
                        a = (pp->p_cpu & 0377)*8/10 + pp->p_nice;
                        if(a < 0)
                                a = 0;
                        if(a > 255)
                                a = 255;
                        pp->p_cpu = a;
                        if(pp->p_pri > PUSER)
                                setpri(pp);
                }
                if(runin!=0) {
                        runin = 0;
                        wakeup(&runin);
                }
                if((ps&UMODE) == UMODE) {
                        u.u_ar0 = &r0;
                        if(issig())
                                psig();
                        setpri(u.u_procp);
                }
        }
}

/*
 * timeout is called to arrange that
 * fun(arg) is called in tim/HZ seconds.
 * An entry is sorted into the callout
 * structure. The time in each structure
 * entry is the number of HZ's more
 * than the previous entry.
 * In this way, decrementing the
 * first entry has the effect of
 * updating all entries.
 */
timeout(fun, arg, tim)
{
        register struct callo *p1, *p2;
        register t;
        int s;

        t = tim;
        s = PS->integ;
        p1 = &callout[0];
        spl7();
        while(p1->c_func != 0 && p1->c_time <= t) {
                t =- p1->c_time;
                p1++;
        }
        p1->c_time =- t;
        p2 = p1;
        while(p2->c_func != 0)
```

```
                p2++;
        while(p2 >= p1) {
                (p2+1)->c_time = p2->c_time;
                (p2+1)->c_func = p2->c_func;
                (p2+1)->c_arg = p2->c_arg;
                p2--;
        }
        p1->c_time = t;
        p1->c_func = fun;
        p1->c_arg = arg;
        PS->integ = s;
}
```

TALK ABOUT OPEN FIL

```
#
#include "../param.h"
#include "../user.h"
#include "../filsys.h"
#include "../file.h"
#include "../conf.h"
#include "../inode.h"
#include "../reg.h"

/*
 * Convert a user supplied
 * file descriptor into a pointer
 * to a file structure.
 * Only task is to check range
 * of the descriptor.
 */
getf(f)
{
        register *fp, rf;

        rf = f;
        if(rf<0 || rf>=NOFILE)
                goto bad;
        fp = u.u_ofile[rf];
        if(fp != NULL)
                return(fp);
bad:
        u.u_error = EBADF;
        return(NULL);
}

/*
 * Internal form of close.
 * Decrement reference count on
 * file structure and call closei
 * on last closef.
 * Also make sure the pipe protocol
 * does not constipate.
 */
closef(fp)
int *fp;
{
        register *rfp, *ip;

        rfp = fp;
        if(rfp->f_flag&FPIPE) {
                ip = rfp->f_inode;
                ip->i_mode =& ~(IREAD|IWRITE);
                wakeup(ip+1);
                wakeup(ip+2);
        }
        if(rfp->f_count <= 1)
                closei(rfp->f_inode, rfp->f_flag&FWRITE);
        rfp->f_count--;
}
```

```
/*
 * Decrement reference count on an
 * inode due to the removal of a
 * referencing file structure.
 * On the last closei, switchout
 * to the close entry point of special
 * device handler.
 * Note that the handler gets called
 * on every open and only on the last
 * close.
 */
closei(ip, rw)
int *ip;
{
        register *rip;
        register dev, maj;

        rip = ip;
        dev = rip->i_addr[0];
        maj = rip->i_addr[0].d_major;
        if(rip->i_count <= 1)
        switch(rip->i_mode&IFMT) {

        case IFCHR:
                (*cdevsw[maj].d_close)(dev, rw);
                break;

        case IFBLK:
                (*bdevsw[maj].d_close)(dev, rw);
        }
        iput(rip);
}

/*
 * openi called to allow handler
 * of special files to initialize and
 * validate before actual IO.
 * Called on all sorts of opens
 * and also on mount.
 */
openi(ip, rw)
int *ip;
{
        register *rip;
        register dev, maj;

        rip = ip;
        dev = rip->i_addr[0];
        maj = rip->i_addr[0].d_major;
        switch(rip->i_mode&IFMT) {

        case IFCHR:
                if(maj >= nchrdev)
                        goto bad;
                (*cdevsw[maj].d_open)(dev, rw);
                break;
```

```
        case IFBLK:
                if(maj >= nblkdev)
                        goto bad;
                (*bdevsw[maj].d_open)(dev, rw);
        }
        return;

bad:
        u.u_error = ENXIO;
}

/*
 * Check mode permission on inode pointer.
 * Mode is READ, WRITE or EXEC.
 * In the case of WRITE, the
 * read-only status of the file
 * system is checked.
 * Also in WRITE, prototype text
 * segments cannot be written.
 * The mode is shifted to select
 * the owner/group/other fields.
 * The super user is granted all
 * permissions.
 */
access(aip, mode)
int *aip;
{
        register *ip, m;

        ip = aip;
        m = mode;
        if(m == IWRITE) {
                if(getfs(ip->i_dev)->s_ronly != 0) {
                        u.u_error = EROFS;
                        return(1);
                }
                if(ip->i_flag & ITEXT) {
                        u.u_error = ETXTBSY;
                        return(1);
                }
        }
        if(u.u_uid == 0)
                return(0);
        if(u.u_uid != ip->i_uid) {
                m =>> 3;
                if(u.u_gid != ip->i_gid)
                        m =>> 3;
        }
        if((ip->i_mode&m) != 0)
                return(0);

bad:
        u.u_error = EACCES;
        return(1);
}
```

```
/*
 * Look up a pathname and test if
 * the resultant inode is owned by the
 * current user.
 * If not, try for super-user.
 * If permission is granted,
 * return inode pointer.
 */
owner()
{
        register struct inode *ip;
        extern uchar();

        if ((ip = namei(uchar, 0)) == NULL)
                return(NULL);
        if(u.u_uid == ip->i_uid)
                return(ip);
        if (suser())
                return(ip);
        iput(ip);
        return(NULL);
}

/*
 * Test if the current user is the
 * super user.
 */
suser()
{

        if(u.u_uid == 0)
                return(1);
        u.u_error = EPERM;
        return(0);
}

/*
 * Allocate a user file descriptor.
 */
ufalloc()
{
        register i;

        for (i=0; i<NOFILE; i++)
                if (u.u_ofile[i] == NULL) {
                        u.u_ar0[R0] = i;
                        return(i);
                }
        u.u_error = EMFILE;
        return(-1);
}

/*
 * Allocate a user file descriptor
 * and a file structure.
```

```
 * Initialize the descriptor
 * to point at the file structure.
 *
 * no file -- if there are no available
 *      file structures.
 */
falloc()
{
        register struct file *fp;
        register i;

        if ((i = ufalloc()) < 0)
                return(NULL);
        for (fp = &file[0]; fp < &file[NFILE]; fp++)
                if (fp->f_count==0) {
                        u.u_ofile[i] = fp;
                        fp->f_count++;
                        fp->f_offset[0] = 0;
                        fp->f_offset[1] = 0;
                        return(fp);
                }
        printf("no file\n");
        u.u_error = ENFILE;
        return(NULL);
}
```

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../inode.h"
#include "../filsys.h"
#include "../conf.h"
#include "../buf.h"

/*
 * Look up an inode by device,inumber.
 * If it is in core (in the inode structure),
 * honor the locking protocol.
 * If it is not in core, read it in from the
 * specified device.
 * If the inode is mounted on, perform
 * the indicated indirection.
 * In all cases, a pointer to a locked
 * inode structure is returned.
 *
 * printf warning: no inodes -- if the inode
 *      structure is full
 * panic: no imt -- if the mounted file
 *      system is not in the mount table.
 *      "cannot happen"
 */
iget(dev, ino)
{
        register struct inode *p;
        register *ip2;
        int *ip1;
        register struct mount *ip;

loop:
        ip = NULL;
        for(p = &inode[0]; p < &inode[NINODE]; p++) {
                if(dev==p->i_dev && ino==p->i_number) {
                        if((p->i_flag&ILOCK) != 0) {
                                p->i_flag =| IWANT;
                                sleep(p, PINOD);
                                goto loop;
                        }
                        if((p->i_flag&IMOUNT) != 0) {
                                for(ip = &mount[0]; ip < &mount[NMOUNT]; ip++)
                                if(ip->m_inodp == p) {
                                        dev = ip->m_dev;
                                        ino = ROOTINO;
                                        goto loop;
                                }
                                panic("no imt");
                        }
                        p->i_count++;
                        p->i_flag =| ILOCK;
                        return(p);
                }
                if(ip==NULL && p->i_count==0)
```

return because may have

lock it

```
                        ip = p;
        }
        if((p=ip) == NULL) {
                printf("Inode table overflow\n");
                u.u_error = ENFILE;
                return(NULL);
        }
        p->i_dev = dev;
        p->i_number = ino;
        p->i_flag = ILOCK;
        p->i_count++;
        p->i_lastr = -1;
        ip = bread(dev, ldiv(ino+31,16));
        /*
         * Check I/O errors
         */
        if (ip->b_flags&B_ERROR) {
                brelse(ip);
                iput(p);
                return(NULL);
        }
        ip1 = ip->b_addr + 32*lrem(ino+31, 16);
        ip2 = &p->i_mode;
        while(ip2 < &p->i_addr[8])
                *ip2++ = *ip1++;
        brelse(ip);
        return(p);
}

/*
 * Decrement reference count of
 * an inode structure.
 * On the last reference,
 * write the inode out and if necessary,
 * truncate and deallocate the file.
 */
iput(p)
struct inode *p;
{
        register *rp;

        rp = p;
        if(rp->i_count == 1) {
                rp->i_flag =| ILOCK;
                if(rp->i_nlink == 0) {
                        itrunc(rp);
                        rp->i_mode = 0;
                        ifree(rp->i_dev, rp->i_number);
                }
                iupdat(rp, time);
                prele(rp);
                rp->i_flag = 0;
                rp->i_number = 0;
        }
        rp->i_count--;
        prele(rp);
```

```
}

/*
 * Check accessed and update flags on
 * an inode structure.
 * If either is on, update the inode
 * with the corresponding dates
 * set to the argument tm.
 */
iupdat(p, tm)
int *p;
int *tm;
{
        register *ip1, *ip2, *rp;
        int *bp, i;

        rp = p;
        if((rp->i_flag&(IUPD|IACC)) != 0) {
                if(getfs(rp->i_dev)->s_ronly)
                        return;
                i = rp->i_number+31;
                bp = bread(rp->i_dev, ldiv(i,16));
                ip1 = bp->b_addr + 32*lrem(i, 16);
                ip2 = &rp->i_mode;
                while(ip2 < &rp->i_addr[8])
                        *ip1++ = *ip2++;
                if(rp->i_flag&IACC) {
                        *ip1++ = time[0];
                        *ip1++ = time[1];
                } else
                        ip1 =+ 2;
                if(rp->i_flag&IUPD) {
                        *ip1++ = *tm++;
                        *ip1++ = *tm;
                }
                bwrite(bp);
        }
}

/*
 * Free all the disk blocks associated
 * with the specified inode structure.
 * The blocks of the file are removed
 * in reverse order. This FILO
 * algorithm will tend to maintain
 * a contiguous free list much longer
 * than FIFO.
 */
itrunc(ip)
int *ip;
{
        register *rp, *bp, *cp;
        int *dp, *ep;

        rp = ip;
        if((rp->i_mode&(IFCHR&IFBLK)) != 0)
```

writes out inode if accessed or mod

```
                return;
        for(ip = &rp->i_addr[7]; ip >= &rp->i_addr[0]; ip--)
        if(*ip) {
                if((rp->i_mode&ILARG) != 0) {
                        bp = bread(rp->i_dev, *ip);
                        for(cp = bp->b_addr+510; cp >= bp->b_addr; cp--)
                        if(*cp) {
                                if(ip == &rp->i_addr[7]) {
                                        dp = bread(rp->i_dev, *cp);
                                        for(ep = dp->b_addr+510; ep >= dp->b_addr
                                        if(*ep)
                                                free(rp->i_dev, *ep);
                                        brelse(dp);
                                }
                                free(rp->i_dev, *cp);
                        }
                        brelse(bp);
                }
                free(rp->i_dev, *ip);
                *ip = 0;
        }
        rp->i_mode =& ~ILARG;
        rp->i_size0 = 0;
        rp->i_size1 = 0;
        rp->i_flag =| IUPD;
}

/*
 * Make a new file.
 */
maknode(mode)
{
        register *ip;

        ip = ialloc(u.u_pdir->i_dev);
        if (ip==NULL)
                return(NULL);
        ip->i_flag =| IACC|IUPD;
        ip->i_mode = mode|IALLOC;
        ip->i_nlink = 1;
        ip->i_uid = u.u_uid;
        ip->i_gid = u.u_gid;
        wdir(ip);
        return(ip);
}

/*
 * Write a directory entry with
 * parameters left as side effects
 * to a call to namei.
 */
wdir(ip)
int *ip;
{
        register char *cp1, *cp2;
```

called after calling a namei w/ a certain option

```
        u.u_dent.u_ino = ip->i_number;
        cp1 = &u.u_dent.u_name[0];
        for(cp2 = &u.u_dbuf[0]; cp2 < &u.u_dbuf[DIRSIZ];)
                *cp1++ = *cp2++;
        u.u_count = DIRSIZ+2;
        u.u_segflg = 1;
        u.u_base = &u.u_dent;
        writei(u.u_pdir);
        iput(u.u_pdir);
}
```

actua
writes
16 by
new
entr

```
#
#include "../param.h"
#include "../user.h"
#include "../systm.h"
#include "../proc.h"
#include "../text.h"
#include "../inode.h"
#include "../seg.h"

#define CLOCK1  0177546
#define CLOCK2  0172540
/*
 * Icode is the octal bootstrap
 * program executed in user mode
 * to bring up the system.
 */
int     icode[]
{
        0104413,        /* sys exec; init; initp */
        0000014,
        0000010,
        0000777,        /* br . */
        0000014,        /* initp: init; 0 */
        0000000,
        0062457,        /* init: </etc/init\0> */
        0061564,
        0064457,
        0064556,
        0000164,
};

/*
 * Initialization code.
 * Called from m40.s or m45.s as
 * soon as a stack and segmentation
 * have been established.
 * Functions:
 *      clear and free user core
 *      find which clock is configured
 *      hand craft 0th process
 *      call all initialization routines
 *      fork - process 0 to schedule
 *           - process 1 execute bootstrap
 *
 * panic: no clock -- neither clock responds
 * loop at loc 6 in user mode -- /etc/init
 *      cannot be executed.
 */
main()
{
        extern schar;
        register i, *p;

        /*
         * zero and free all of core
         */
```

```
	updlock = 0;
	i = *ka6 + USIZE;
	UISD->r[0] = 077406;
	for(;;) {
		UISA->r[0] = i;
		if(fuibyte(0) < 0)
			break;
		clearseg(i);
		maxmem++;
		mfree(coremap, 1, i);
		i++;
	}
	if(cputype == 70)
	for(i=0; i<62; i=+2) {
		UBMAP->r[i] = i<<12;
		UBMAP->r[i+1] = 0;
	}
	printf("mem = %l\n", maxmem*5/16);
	maxmem = min(maxmem, MAXMEM);
	mfree(swapmap, nswap, swplo);

	/*
	 * determine clock
	 */

	UISA->r[7] = ka6[1]; /* io segment */
	UISD->r[7] = 077406;
	lks = CLOCK1;
	if(fuiword(lks) == -1) {
		lks = CLOCK2;
		if(fuiword(lks) == -1)
			panic("no clock");
	}

	/*
	 * set up system process
	 */

	proc[0].p_addr = *ka6;
	proc[0].p_size = USIZE;
	proc[0].p_stat = SRUN;
	proc[0].p_flag =| SLOAD|SSYS;
	u.u_procp = &proc[0];

	/*
	 * set up 'known' i-nodes
	 */

	*lks = 0115;
	cinit();
	binit();
	iinit();
	rootdir = iget(rootdev, ROOTINO);
	rootdir->i_flag =& ~ILOCK;
	u.u_cdir = iget(rootdev, ROOTINO);
```

```
        u.u_cdir->i_flag =& ~ILOCK;

        /*
         * make init process
         * enter scheduling loop
         * with system process
         */

        if(newproc()) {
                expand(USIZE+1);
                estabur(0, 1, 0, 0);
                copyout(icode, 0, sizeof icode);
                /*
                 * Return goes to loc. 0 of user init
                 * code just copied out.
                 */
                return;
        }
        sched();
}

/*
 * Load the user hardware segmentation
 * registers from the software prototype.
 * The software registers must have
 * been setup prior by estabur.
 */
sureg()
{
        register *up, *rp, a;

        a = u.u_procp->p_addr;
        up = &u.u_uisa[16];
        rp = &UISA->r[16];
        if(cputype == 40) {
                up =- 8;
                rp =- 8;
        }
        while(rp > &UISA->r[0])
                *--rp = *--up + a;
        if((up=u.u_procp->p_textp) != NULL)
                a =- up->x_caddr;
        up = &u.u_uisd[16];
        rp = &UISD->r[16];
        if(cputype == 40) {
                up =- 8;
                rp =- 8;
        }
        while(rp > &UISD->r[0]) {
                *--rp = *--up;
                if((*rp & WO) == 0)
                        rp[(UISA-UISD)/2] =- a;
        }
}

/*
```

```
 * Set up software prototype segmentation
 * registers to implement the 3 pseudo
 * text,data,stack segment sizes passed
 * as arguments.
 * The argument sep specifies if the
 * text and data+stack segments are to
 * be separated.
 */
estabur(nt, nd, ns, sep)
{
        register a, *ap, *dp;

        if(sep) {
                if(cputype == 40)
                        goto err;
                if(nseg(nt) > 8 || nseg(nd)+nseg(ns) > 8)
                        goto err;
        } else
                if(nseg(nt)+nseg(nd)+nseg(ns) > 8)
                        goto err;
        if(nt+nd+ns+USIZE > maxmem)
                goto err;
        a = 0;
        ap = &u.u_uisa[0];
        dp = &u.u_uisd[0];
        while(nt >= 128) {
                *dp++ = (127<<8) | RO;
                *ap++ = a;
                a =+ 128;
                nt =- 128;
        }
        if(nt) {
                *dp++ = ((nt-1)<<8) | RO;
                *ap++ = a;
        }
        if(sep)
        while(ap < &u.u_uisa[8]) {
                *ap++ = 0;
                *dp++ = 0;
        }
        a = USIZE;
        while(nd >= 128) {
                *dp++ = (127<<8) | RW;
                *ap++ = a;
                a =+ 128;
                nd =- 128;
        }
        if(nd) {
                *dp++ = ((nd-1)<<8) | RW;
                *ap++ = a;
                a =+ nd;
        }
        while(ap < &u.u_uisa[8]) {
                *dp++ = 0;
                *ap++ = 0;
        }
```

```
        if(sep)
        while(ap < &u.u_uisa[16]) {
                *dp++ = 0;
                *ap++ = 0;
        }
        a =+ ns;
        while(ns >= 128) {
                a =- 128;
                ns =- 128;
                *--dp = (127<<8) | RW;
                *--ap = a;
        }
        if(ns) {
                *--dp = ((128-ns)<<8) | RW | ED;
                *--ap = a-128;
        }
        if(!sep) {
                ap = &u.u_uisa[0];
                dp = &u.u_uisa[8];
                while(ap < &u.u_uisa[8])
                        *dp++ = *ap++;
                ap = &u.u_uisd[0];
                dp = &u.u_uisd[8];
                while(ap < &u.u_uisd[8])
                        *dp++ = *ap++;
        }
        sureg();
        return(0);

err:
        u.u_error = ENOMEM;
        return(-1);
}

/*
 * Return the arg/128 rounded up.
 */
nseg(n)
{

        return((n+127)>>7);
}
```

```
#
/*
 * Structure of the coremap and swapmap
 * arrays. Consists of non-zero count
 * and base address of that many
 * contiguous units.
 * (The coremap unit is 64 bytes,
 * the swapmap unit is 512 bytes)
 * The addresses are increasing and
 * the list is terminated with the
 * first zero count.
 */
struct map
{
        char *m_size;
        char *m_addr;
};

/*
 * Allocate size units from the given
 * map. Return the base of the allocated
 * space.
 * Algorithm is first fit.
 */
malloc(mp, size)
struct map *mp;
{
        register int a;
        register struct map *bp;

        for (bp = mp; bp->m_size; bp++) {
                if (bp->m_size >= size) {
                        a = bp->m_addr;
                        bp->m_addr =+ size;
                        if ((bp->m_size =- size) == 0)
                                do {
                                        bp++;
                                        (bp-1)->m_addr = bp->m_addr;
                                } while ((bp-1)->m_size = bp->m_size);
                        return(a);
                }
        }
        return(0);
}

/*
 * Free the previously allocated space aa
 * of size units into the specified map.
 * Sort aa into map and combine on
 * one or both ends if possible.
 */
mfree(mp, size, aa)
struct map *mp;
{
        register struct map *bp;
        register int t;
```

```
        register int a;

        a = aa;
        for (bp = mp; bp->m_addr<=a && bp->m_size!=0; bp++);
        if (bp>mp && (bp-1)->m_addr+(bp-1)->m_size == a) {
                (bp-1)->m_size =+ size;
                if (a+size == bp->m_addr) {
                        (bp-1)->m_size =+ bp->m_size;
                        while (bp->m_size) {
                                bp++;
                                (bp-1)->m_addr = bp->m_addr;
                                (bp-1)->m_size = bp->m_size;
                        }
                }
        } else {
                if (a+size == bp->m_addr && bp->m_size) {
                        bp->m_addr =- size;
                        bp->m_size =+ size;
                } else if (size) do {
                        t = bp->m_addr;
                        bp->m_addr = a;
                        a = t;
                        t = bp->m_size;
                        bp->m_size = size;
                        bp++;
                } while (size = t);
        }
}
```

```
#
#include "../param.h"
#include "../inode.h"
#include "../user.h"
#include "../systm.h"
#include "../buf.h"

/*
 * Convert a pathname into a pointer to
 * an inode. Note that the inode is locked.
 *
 * func = function called to get next char of name
 *      &uchar if name is in user space
 *      &schar if name is in system space
 * flag = 0 if name is sought
 *      1 if name is to be created
 *      2 if name is to be deleted
 */
namei(func, flag)
int (*func)();
{
        register struct inode *dp;
        register c;
        register char *cp;
        int eo, *bp;

        /*
         * If name starts with '/' start from
         * root; otherwise start from current dir.
         */

        dp = u.u_cdir;
        if((c=(*func)()) == '/')
                dp = rootdir;
        iget(dp->i_dev, dp->i_number);
        while(c == '/')
                c = (*func)();
        if(c == '\0' && flag != 0) {
                u.u_error = ENOENT;
                goto out;
        }

cloop:
        /*
         * Here dp contains pointer
         * to last component matched.
         */

        if(u.u_error)
                goto out;
        if(c == '\0')
                return(dp);

        /*
         * If there is another component,
         * dp must be a directory and
```

(searches for inode; count; reads etc. (iput unlock) ← basic routine

```
        * must have x permission.
        */

        if((dp->i_mode&IFMT) != IFDIR) {
                u.u_error = ENOTDIR;
                goto out;
        }
        if(access(dp, IEXEC))
                goto out;

        /*
         * Gather up name into
         * users' dir buffer.
         */

        cp = &u.u_dbuf[0];
        while(c!='/' && c!='\0' && u.u_error==0) {
                if(cp < &u.u_dbuf[DIRSIZ])
                        *cp++ = c;
                c = (*func)();
        }
        while(cp < &u.u_dbuf[DIRSIZ])
                *cp++ = '\0';
        while(c == '/')
                c = (*func)();
        if(u.u_error)
                goto out;

        /*
         * Set up to search a directory.
         */

        u.u_offset[1] = 0;
        u.u_offset[0] = 0;
        u.u_segflg = 1;
        eo = 0;
        u.u_count = ldiv(dp->i_size1, DIRSIZ+2);
        bp = NULL;

eloop:

        /*
         * If at the end of the directory,
         * the search failed. Report what
         * is appropriate as per flag.
         */

        if(u.u_count == 0) {
                if(bp != NULL)
                        brelse(bp);
                if(flag==1 && c=='\0') {
                        if(access(dp, IWRITE))
                                goto out;
                        u.u_pdir = dp;
                        if(eo)
                                u.u_offset[1] = eo-DIRSIZ-2; else
```

```
                        dp->i_flag =| IUPD;
                return(NULL);
        }
        u.u_error = ENOENT;
        goto out;
}

/*
 * If offset is on a block boundary,
 * read the next directory block.
 * Release previous if it exists.
 */

if((u.u_offset[1]&0777) == 0) {
        if(bp != NULL)
                brelse(bp);
        bp = bread(dp->i_dev,
                bmap(dp, ldiv(u.u_offset[1], 512)));
}

/*
 * Note first empty directory slot
 * in eo for possible creat.
 * String compare the directory entry
 * and the current component.
 * If they do not match, go back to eloop.
 */

bcopy(bp->b_addr+(u.u_offset[1]&0777), &u.u_dent, (DIRSIZ+2)/2);
u.u_offset[1] =+ DIRSIZ+2;
u.u_count--;
if(u.u_dent.u_ino == 0) {
        if(eo == 0)
                eo = u.u_offset[1];
        goto eloop;
}
for(cp = &u.u_dbuf[0]; cp < &u.u_dbuf[DIRSIZ]; cp++)
        if(*cp != cp[u.u_dent.u_name - u.u_dbuf])
                goto eloop;

/*
 * Here a component matched in a directory.
 * If there is more pathname, go back to
 * cloop, otherwise return.
 */

if(bp != NULL)
        brelse(bp);
if(flag==2 && c=='\0') {
        if(access(dp, IWRITE))
                goto out;
        return(dp);
}
bp = dp->i_dev;
iput(dp);
dp = iget(bp, u.u_dent.u_ino);
```

converts into

eo is first blank slot

actual sear

```
        if(dp == NULL)
                return(NULL);
        goto cloop;

out:
        iput(dp);
        return(NULL);
}

/*
 * Return the next character from the
 * kernel string pointed at by dirp.
 */
schar()
{

        return(*u.u_dirp++ & 0377);
}

/*
 * Return the next character from the
 * user string pointed at by dirp.
 */
uchar()
{
        register c;

        c = fubyte(u.u_dirp++);
        if(c == -1)
                u.u_error = EFAULT;
        return(c);
}
```

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../inode.h"
#include "../file.h"
#include "../reg.h"

/*
 * Max allowable buffering per pipe.
 * This is also the max size of the
 * file created to implement the pipe.
 * If this size is bigger than 4096,
 * pipes will be implemented in LARG
 * files, which is probably not good.
 */
#define PIPSIZ  4096

/*
 * The sys-pipe entry.
 * Allocate an inode on the root device.
 * Allocate 2 file structures.
 * Put it all together with flags.
 */
pipe()
{
```

create a pipe

```
        register *ip, *rf, *wf;
        int r;

        ip = ialloc(rootdev);
        if(ip == NULL)
                return;
        rf = falloc();
        if(rf == NULL) {
                iput(ip);
                return;
        }
        r = u.u_ar0[R0];
        wf = falloc();
        if(wf == NULL) {
                rf->f_count = 0;
                u.u_ofile[r] = NULL;
                iput(ip);
                return;
        }
        u.u_ar0[R1] = u.u_ar0[R0];
        u.u_ar0[R0] = r;
        wf->f_flag = FWRITE|FPIPE;
        wf->f_inode = ip;
        rf->f_flag = FREAD|FPIPE;
        rf->f_inode = ip;
        ip->i_count = 2;
        ip->i_flag = IACC|IUPD;
        ip->i_mode = IALLOC;
}
```

```
/*
 * Read call directed to a pipe.
 */
readp(fp)
int *fp;
{
        register *rp, *ip;

        rp = fp;
        ip = rp->f_inode;

loop:
        /*
         * Very conservative locking.
         */

        plock(ip);

        /*
         * If the head (read) has caught up with
         * the tail (write), reset both to 0.
         */

        if(rp->f_offset[1] == ip->i_size1) {
                if(rp->f_offset[1] != 0) {
                        rp->f_offset[1] = 0;
                        ip->i_size1 = 0;
                        if(ip->i_mode&IWRITE) {
                                ip->i_mode =& ~IWRITE;
                                wakeup(ip+1);
                        }
                }

                /*
                 * If there are not both reader and
                 * writer active, return without
                 * satisfying read.
                 */

                prele(ip);
                if(ip->i_count < 2)
                        return;
                ip->i_mode =| IREAD;
                sleep(ip+2, PPIPE);
                goto loop;
        }

        /*
         * Read and return
         */

        u.u_offset[0] = 0;
        u.u_offset[1] = rp->f_offset[1];
        readi(ip);
        rp->f_offset[1] = u.u_offset[1];
        prele(ip);
```

lock the pipe

unlock

```
}

/*
 * Write call directed to a pipe.
 */
writep(fp)
{
        register *rp, *ip, c;

        rp = fp;
        ip = rp->f_inode;
        c = u.u_count;

loop:

        /*
         * If all done, return.
         */

        plock(ip);
        if(c == 0) {
                prele(ip);
                u.u_count = 0;
                return;
        }

        /*
         * If there are not both read and
         * write sides of the pipe active,
         * return error and signal too.
         */

        if(ip->i_count < 2) {
                prele(ip);
                u.u_error = EPIPE;
                psignal(u.u_procp, SIGPIPE);
                return;
        }

        /*
         * If the pipe is full,
         * wait for reads to deplete
         * and truncate it.
         */

        if(ip->i_size1 == PIPSIZ) {
                ip->i_mode =| IWRITE;
                prele(ip);
                sleep(ip+1, PPIPE);
                goto loop;
        }

        /*
         * Write what is possible and
         * loop back.
         */
```

lock the p'pe

```
        u.u_offset[0] = 0;
        u.u_offset[1] = ip->i_size1;
        u.u_count = min(c, PIPSIZ-u.u_offset[1]);
        c =- u.u_count;
        writei(ip);
        prele(ip);
        if(ip->i_mode&IREAD) {
                ip->i_mode =& ~IREAD;
                wakeup(ip+2);
        }
        goto loop;
}

/*
 * Lock a pipe.
 * If its already locked,
 * set the WANT bit and sleep.
 */
plock(ip)
int *ip;
{
        register *rp;

        rp = ip;
        while(rp->i_flag&ILOCK) {
                rp->i_flag =| IWANT;
                sleep(rp, PPIPE);
        }
        rp->i_flag =| ILOCK;
}

/*
 * Unlock a pipe.
 * If WANT bit is on,
 * wakeup.
 * This routine is also used
 * to unlock inodes in general.
 */
prele(ip)
int *ip;
{
        register *rp;

        rp = ip;
        rp->i_flag =& ~ILOCK;
        if(rp->i_flag&IWANT) {
                rp->i_flag =& ~IWANT;
                wakeup(rp);
        }
}
```

unlock

```
#
#include "../param.h"
#include "../seg.h"
#include "../buf.h"
#include "../conf.h"
#include "../systm.h"

/*
 * Address and structure of the
 * KL-11 console device registers.
 */
struct
{
        int     rsr;
        int     rbr;
        int     xsr;
        int     xbr;
};
char    *msgbufp msgbuf;        /* Next saved printf character */

/*
 * In case console is off,
 * panicstr contains argument to last
 * call to panic.
 */

char    *panicstr;

/*
 * Scaled down version of C Library printf.
 * Only %s %l %d (==%l) %o are recognized.
 * Used to print diagnostic information
 * directly on console tty.
 * Since it is not interrupt driven,
 * all system activities are pretty much
 * suspended.
 * Printf should not be used for chit-chat.
 */
printf(fmt,x1,x2,x3,x4,x5,x6,x7,x8,x9,xa,xb,xc)
char fmt[];
{
        register char *s;
        register *adx, c;

        adx = &x1;
loop:
        while((c = *fmt++) != '%') {
                if(c == '\0')
                        return;
                putchar(c);
        }
        c = *fmt++;
        if(c == 'd' || c == 'l' || c == 'o')
                printn(*adx, c=='o'? 8: 10);
        if(c == 's') {
                s = *adx;
```

```
                while(c = *s++)
                        putchar(c);
        }
        adx++;
        goto loop;
}

/*
 * Print an unsigned integer in base b.
 */
printn(n, b)
{
        register a;

        if(a = ldiv(n, b))
                printn(a, b);
        putchar(lrem(n, b) + '0');
}

/*
 * Print a character on console.
 * Attempts to save and restore device
 * status.
 * If the switches are 0, all
 * printing is inhibited.
 *
 * Whether or not printing is inhibited,
 * the last MSGBUFS characters
 * are saved in msgbuf for inspection later.
 */
putchar(c)
{
        register rc, s, timo;

        rc = c;
        if (rc!='\0' && rc!='\r' && rc!=0177) {
                *msgbufp++ = rc;
                if (msgbufp >= &msgbuf[MSGBUFS])
                        msgbufp = msgbuf;
        }
        if(SW->integ == 0)
                return;
        timo = 30000;
        /*
         * Try waiting for the console tty to come ready,
         * otherwise give up after a reasonable time.
         */
        while((KL->xsr&0200)==0 && --timo!=0)
                ;
        if(rc == 0)
                return;
        s = KL->xsr;
        KL->xsr = 0;
        KL->xbr = rc;
        if(rc == '\n') {
                putchar('\r');
```

```
                putchar(0177);
                putchar(0177);
        }
        putchar(0);
        KL->xsr = s;
}

/*
 * Panic is called on unresolvable
 * fatal errors.
 * It syncs, prints "panic: mesg" and
 * then loops.
 */
panic(s)
char *s;
{
        panicstr = s;
        update();
        printf("panic: %s\n", s);
        for(;;)
                idle();
}

/*
 * prdev prints a warning message of the
 * form "mesg on dev x/y".
 * x and y are the major and minor parts of
 * the device argument.
 */
prdev(str, dev)
{

        printf("%s on dev %l/%l\n", str, dev.d_major, dev.d_minor);
}

/*
 * deverr prints a diagnostic from
 * a device driver.
 * It prints the device, block number,
 * and an octal word (usually some error
 * status register) passed as argument.
 */
deverror(bp, o1, o2)
int *bp;
{
        register *rbp;

        rbp = bp;
        prdev("err", rbp->b_dev);
        printf("bn%l er%o %o\n", rbp->b_blkno, o1, o2);
}
```

```
#
#include "../param.h"
#include "../inode.h"
#include "../user.h"
#include "../buf.h"
#include "../conf.h"
#include "../systm.h"

/*
 * Read the file corresponding to
 * the inode pointed at by the argument.
 * The actual read arguments are found
 * in the variables:
 *      u_base          core address for destination
 *      u_offset        byte offset in file
 *      u_count         number of bytes to read
 *      u_segflg        read to kernel/user
 */
readi(aip)
struct inode *aip;
{
        int *bp;
        int lbn, bn, on;
        register dn, n;
        register struct inode *ip;

        ip = aip;
        if(u.u_count == 0)
                return;
        ip->i_flag =| IACC;
        if((ip->i_mode&IFMT) == IFCHR) {
                (*cdevsw[ip->i_addr[0].d_major].d_read)(ip->i_addr[0]);
                return;
        }

        do {
                lbn = bn = lshift(u.u_offset, -9);
                on = u.u_offset[1] & 0777;
                n = min(512-on, u.u_count);
                if((ip->i_mode&IFMT) != IFBLK) {
                        dn = dpcmp(ip->i_size0&0377, ip->i_size1,
                                u.u_offset[0], u.u_offset[1]);
                        if(dn <= 0)
                                return;
                        n = min(n, dn);
                        if ((bn = bmap(ip, lbn)) == 0)
                                return;
                        dn = ip->i_dev;
                } else {
                        dn = ip->i_addr[0];
                        rablock = bn+1;
                }
                if (ip->i_lastr+1 == lbn)
                        bp = breada(dn, bn, rablock);
                else
                        bp = bread(dn, bn);
```

```
                ip->i_lastr = lbn;
                iomove(bp, on, n, B_READ);
                brelse(bp);
        } while(u.u_error==0 && u.u_count!=0);
}

/*
 * Write the file corresponding to
 * the inode pointed at by the argument.
 * The actual write arguments are found
 * in the variables:
 *      u_base          core address for source
 *      u_offset        byte offset in file
 *      u_count         number of bytes to write
 *      u_segflg        write to kernel/user
 */
writei(aip)
struct inode *aip;
{
        int *bp;
        int n, on;
        register dn, bn;
        register struct inode *ip;

        ip = aip;
        ip->i_flag =| IACC|IUPD;
        if((ip->i_mode&IFMT) == IFCHR) {
                (*cdevsw[ip->i_addr[0].d_major].d_write)(ip->i_addr[0]);
                return;
        }
        if (u.u_count == 0)
                return;

        do {
                bn = lshift(u.u_offset, -9);
                on = u.u_offset[1] & 0777;
                n = min(512-on, u.u_count);
                if((ip->i_mode&IFMT) != IFBLK) {
                        if ((bn = bmap(ip, bn)) == 0)
                                return;
                        dn = ip->i_dev;
                } else
                        dn = ip->i_addr[0];
                if(n == 512)
                        bp = getblk(dn, bn); else
                        bp = bread(dn, bn);
                iomove(bp, on, n, B_WRITE);
                if(u.u_error != 0)
                        brelse(bp); else
                if ((u.u_offset[1]&0777)==0)
                        bawrite(bp); else
                        bdwrite(bp);
                if(dpcmp(ip->i_size0&0377, ip->i_size1,
                  u.u_offset[0], u.u_offset[1]) < 0 &&
                  (ip->i_mode&(IFBLK&IFCHR)) == 0) {
                        ip->i_size0 = u.u_offset[0];
```

if writing who else read

```
                        ip->i_size1 = u.u_offset[1];
                }
                ip->i_flag =| IUPD;
        } while(u.u_error==0 && u.u_count!=0);
}

/*
 * Return the logical maximum
 * of the 2 arguments.
 */
max(a, b)
char *a, *b;
{

        if(a > b)
                return(a);
        return(b);
}

/*
 * Return the logical minimum
 * of the 2 arguments.
 */
min(a, b)
char *a, *b;
{

        if(a < b)
                return(a);
        return(b);
}

/*
 * Move 'an' bytes at byte location
 * &bp->b_addr[o] to/from (flag) the
 * user/kernel (u.segflg) area starting at u.base.
 * Update all the arguments by the number
 * of bytes moved.
 *
 * There are 2 algorithms,
 * if source address, dest address and count
 * are all even in a user copy,
 * then the machine language copyin/copyout
 * is called.
 * If not, its done byte-by-byte with
 * cpass and passc.
 */
iomove(bp, o, an, flag)
struct buf *bp;
{
        register char *cp;
        register int n, t;

        n = an;
        cp = bp->b_addr + o;
        if(u.u_segflg==0 && ((n | cp | u.u_base)&01)==0) {
```

unsigned

```
                if (flag==B_WRITE)
                        cp = copyin(u.u_base, cp, n);
                else
                        cp = copyout(cp, u.u_base, n);
                if (cp) {
                        u.u_error = EFAULT;
                        return;
                }
                u.u_base =+ n;
                dpadd(u.u_offset, n);
                u.u_count =- n;
                return;
        }
        if (flag==B_WRITE) {
                while(n--) {
                        if ((t = cpass()) < 0)
                                return;
                        *cp++ = t;
                }
        } else
                while (n--)
                        if(passc(*cp++) < 0)
                                return;
}
```

word orien
(if everyth

byte oriente
(something i

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../proc.h"
#include "../inode.h"
#include "../reg.h"

/*
 * Priority for tracing
 */
#define IPCPRI  0

/*
 * Structure to access an array of integers.
 */
struct
{
        int     inta[];
};

/*
 * Tracing variables.
 * Used to pass trace command from
 * parent to child being traced.
 * This data base cannot be
 * shared and is locked
 * per user.
 */
struct
{
        int     ip_lock;
        int     ip_req;
        int     ip_addr;
        int     ip_data;
} ipc;

/*
 * Send the specified signal to
 * all processes with 'pgrp' as
 * process group.
 * Called by tty.c for quits and
 * interrupts.
 */
signal(apgrp, sig)
{
        register struct proc *p;
        int pgrp;

        if ((pgrp = apgrp)==0)
                return;
        for(p = &proc[0]; p < &proc[NPROC]; p++)
                if(p->p_pgrp == pgrp)
                        psignal(p, sig);
}
```

```
/*
 * Send the specified signal to
 * the specified process.
 */
psignal(p, sig)
int *p;
char *sig;
{
        register *rp;

        if(sig >= NSIG)
                return;
        rp = p;
        if(rp->p_sig != SIGKIL)
                rp->p_sig = sig;
        if(rp->p_pri > PUSER)
                rp->p_pri = PUSER;
        if(rp->p_stat == SSLEEP && rp->p_pri>0)
                setrun(rp);
}

/*
 * Returns true if the current
 * process has a signal to process.
 * This is asked at least once
 * each time a process enters the
 * system.
 * A signal does not do anything
 * directly to a process; it sets
 * a flag that asks the process to
 * do something to itself.
 */
issig()
{
        register n;
        register struct proc *p;

        p = u.u_procp;
        if(n = p->p_sig) {
                if (p->p_flag&STRC) {
                        stop();
                        if ((n = p->p_sig) == 0)
                                return(0);
                }
                if((u.u_signal[n]&1) == 0)
                        return(n);
        }
        return(0);
}

/*
 * Enter the tracing STOP state.
 * In this state, the parent is
 * informed and the process is able to
 * receive commands from the parent.
 */
```

```
stop()
{
        register struct proc *pp, *cp;

loop:
        cp = u.u_procp;
        if(cp->p_ppid != 1)
        for (pp = &proc[0]; pp < &proc[NPROC]; pp++)
                if (pp->p_pid == cp->p_ppid) {
                        wakeup(pp);
                        cp->p_stat = SSTOP;
                        swtch();
                        if ((cp->p_flag&STRC)==0 || procxmt())
                                return;
                        goto loop;
                }
        exit();
}

/*
 * Perform the action specified by
 * the current signal.
 * The usual sequence is:
 *      if(issig())
 *              psig();
 */
psig()
{
        register n, p;
        register *rp;

        rp = u.u_procp;
        n = rp->p_sig;
        rp->p_sig = 0;
        if((p=u.u_signal[n]) != 0) {
                u.u_error = 0;
                if(n != SIGINS && n != SIGTRC)
                        u.u_signal[n] = 0;
                n = u.u_ar0[R6] - 4;
                grow(n);
                suword(n+2, u.u_ar0[RPS]);
                suword(n, u.u_ar0[R7]);
                u.u_ar0[R6] = n;
                u.u_ar0[RPS] =& ~TBIT;
                u.u_ar0[R7] = p;
                return;
        }
        switch(n) {

        case SIGQIT:
        case SIGINS:
        case SIGTRC:
        case SIGIOT:
        case SIGEMT:
        case SIGFPT:
        case SIGBUS:
```

```
        case SIGSEG:
        case SIGSYS:
                u.u_arg[0] = n;
                if(core())
                        n =+ 0200;
        }
        u.u_arg[0] = (u.u_ar0[R0]<<8) | n;
        exit();
}

/*
 * Create a core image on the file "core"
 * If you are looking for protection glitches,
 * there are probably a wealth of them here
 * when this occurs to a suid command.
 *
 * It writes USIZE block of the
 * user.h area followed by the entire
 * data+stack segments.
 */
core()
{
        register s, *ip;
        extern schar;

        u.u_error = 0;
        u.u_dirp = "core";
        ip = namei(&schar, 1);
        if(ip == NULL) {
                if(u.u_error)
                        return(0);
                ip = maknode(0666);
                if (ip==NULL)
                        return(0);
        }
        if(!access(ip, IWRITE) &&
           (ip->i_mode&IFMT) == 0 &&
           u.u_uid == u.u_ruid) {
                itrunc(ip);
                u.u_offset[0] = 0;
                u.u_offset[1] = 0;
                u.u_base = &u;
                u.u_count = USIZE*64;
                u.u_segflg = 1;
                writei(ip);
                s = u.u_procp->p_size - USIZE;
                estabur(0, s, 0, 0);
                u.u_base = 0;
                u.u_count = s*64;
                u.u_segflg = 0;
                writei(ip);
        }
        iput(ip);
        return(u.u_error==0);
}
```

```
/*
 * grow the stack to include the SP
 * true return if successful.
 */

grow(sp)
char *sp;
{
        register a, si, i;

        if(sp >= -u.u_ssize*64)
                return(0);
        si = ldiv(-sp, 64) - u.u_ssize + SINCR;
        if(si <= 0)
                return(0);
        if(estabur(u.u_tsize, u.u_dsize, u.u_ssize+si, u.u_sep))
                return(0);
        expand(u.u_procp->p_size+si);
        a = u.u_procp->p_addr + u.u_procp->p_size;
        for(i=u.u_ssize; i; i--) {
                a--;
                copyseg(a-si, a);
        }
        for(i=si; i; i--)
                clearseg(--a);
        u.u_ssize =+ si;
        return(1);
}

/*
 * sys-trace system call.
 */
ptrace()
{
        register struct proc *p;

        if (u.u_arg[2] <= 0) {
                u.u_procp->p_flag =| STRC;
                return;
        }
        for (p=proc; p < &proc[NPROC]; p++)
                if (p->p_stat==SSTOP
                 && p->p_pid==u.u_arg[0]
                 && p->p_ppid==u.u_procp->p_pid)
                        goto found;
        u.u_error = ESRCH;
        return;

    found:
        while (ipc.ip_lock)
                sleep(&ipc, IPCPRI);
        ipc.ip_lock = p->p_pid;
        ipc.ip_data = u.u_ar0[R0];
        ipc.ip_addr = u.u_arg[1] & ~01;
        ipc.ip_req = u.u_arg[2];
        p->p_flag =& ~SWTED;
```

```
        setrun(p);
        while (ipc.ip_req > 0)
                sleep(&ipc, IPCPRI);
        u.u_ar0[R0] = ipc.ip_data;
        if (ipc.ip_req < 0)
                u.u_error = EIO;
        ipc.ip_lock = 0;
        wakeup(&ipc);
}

/*
 * Code that the child process
 * executes to implement the command
 * of the parent process in tracing.
 */
procxmt()
{
        register int i;
        register int *p;

        if (ipc.ip_lock != u.u_procp->p_pid)
                return(0);
        i = ipc.ip_req;
        ipc.ip_req = 0;
        wakeup(&ipc);
        switch (i) {

        /* read user I */
        case 1:
                if (fuibyte(ipc.ip_addr) == -1)
                        goto error;
                ipc.ip_data = fuiword(ipc.ip_addr);
                break;

        /* read user D */
        case 2:
                if (fubyte(ipc.ip_addr) == -1)
                        goto error;
                ipc.ip_data = fuword(ipc.ip_addr);
                break;

        /* read u */
        case 3:
                i = ipc.ip_addr;
                if (i<0 || i >= (USIZE<<6))
                        goto error;
                ipc.ip_data = u.inta[i>>1];
                break;

        /* write user I (for now, always an error) */
        case 4:
                if (suiword(ipc.ip_addr, 0) < 0)
                        goto error;
                suiword(ipc.ip_addr, ipc.ip_data);
                break;
```

```
        /* write user D */
        case 5:
                if (suword(ipc.ip_addr, 0) < 0)
                        goto error;
                suword(ipc.ip_addr, ipc.ip_data);
                break;

        /* write u */
        case 6:
                p = &u.inta[ipc.ip_addr>>1];
                if (p >= u.u_fsav && p < &u.u_fsav[25])
                        goto ok;
                for (i=0; i<9; i++)
                        if (p == &u.u_ar0[regloc[i]])
                                goto ok;
                goto error;
        ok:
                if (p == &u.u_ar0[RPS]) {
                        ipc.ip_data =| 0170000; /* assure user space */
                        ipc.ip_data =& ~0340;   /* priority 0 */
                }
                *p = ipc.ip_data;
                break;

        /* set signal and continue */
        case 7:
                u.u_procp->p_sig = ipc.ip_data;
                return(1);

        /* force exit */
        case 8:
                exit();

        default:
        error:
                ipc.ip_req = -1;
        }
        return(0);
}
```

```
#
#include "../param.h"
#include "../user.h"
#include "../proc.h"
#include "../text.h"
#include "../systm.h"
#include "../file.h"
#include "../inode.h"
#include "../buf.h"

/*
 * Give up the processor till a wakeup occurs
 * on chan, at which time the process
 * enters the scheduling queue at priority pri.
 * The most important effect of pri is that when
 * pri<=0 a signal cannot disturb the sleep;
 * if pri>0 signals will be processed.
 * Callers of this routine must be prepared for
 * premature return, and check that the reason for
 * sleeping has gone away.
 */
sleep(chan, pri)
{
        register *rp, s;

        s = PS->integ;
        rp = u.u_procp;
        spl6();
        rp->p_stat = SSLEEP;
        rp->p_wchan = chan;
        rp->p_pri = pri;
        spl0();
        if(pri > 0) {
                if(issig())
                        goto psig;
                if(runin != 0) {
                        runin = 0;
                        wakeup(&runin);
                }
                swtch();
                if(issig())
                        goto psig;
        } else
                swtch();
        PS->integ = s;
        return;

        /*
         * If priority was low (>0) and
         * there has been a signal,
         * execute non-local goto to
         * the qsav location.
         * (see trap1/trap.c)
         */
psig:
        rp->p_stat = SRUN;
```

```
        aretu(u.u_qsav);
}

/*
 * Wake up all processes sleeping on chan.
 */
wakeup(chan)
{
        register struct proc *p;
        register c, i;

        c = chan;
        p = &proc[0];
        i = NPROC;
        do {
                if(p->p_wchan == c) {
                        setrun(p);
                }
                p++;
        } while(--i);
}

/*
 * Set the process running;
 * arrange for it to be swapped in if necessary.
 */
setrun(p)
{
        register struct proc *rp;

        rp = p;
        rp->p_wchan = 0;
        rp->p_stat = SRUN;
        if(rp->p_pri < curpri)
                runrun++;
        if(runout != 0 && (rp->p_flag&SLOAD) == 0) {
                runout = 0;
                wakeup(&runout);
        }
}

/*
 * Set user priority.
 * The rescheduling flag (runrun)
 * is set if the priority is higher
 * than the currently running process.
 */
setpri(up)
{
        register *pp, p;

        pp = up;
        p = (pp->p_cpu & 0377)/16;
        p =+ PUSER + pp->p_nice;
        if(p > 127)
                p = 127;
```

```
        if(p > curpri)
                runrun++;
        pp->p_pri = p;
}

/*
 * The main loop of the scheduling (swapping)
 * process.
 * The basic idea is:
 *  see if anyone wants to be swapped in;
 *  swap out processes until there is room;
 *  swap him in;
 *  repeat.
 * Although it is not remarkably evident, the basic
 * synchronization here is on the runin flag, which is
 * slept on and is set once per second by the clock routine.
 * Core shuffling therefore takes place once per second.
 *
 * panic: swap error -- IO error while swapping.
 *      this is the one panic that should be
 *      handled in a less drastic way. Its
 *      very hard.
 */
sched()
{
        struct proc *p1;
        register struct proc *rp;
        register a, n;
        int *p2, fspass, s;

        /*
         * find user to swap in
         * of users ready, select one out longest
         */

        fspass = 0;
        goto loop;

sloop:
        runin++;
        sleep(&runin, PSWP);

loop:
        spl6();
        n = -1;
        for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)==0 &&
            rp->p_time > n) {
                p1 = rp;
                n = rp->p_time;
        }
        if(n == -1) {
                runout++;
                sleep(&runout, PSWP);
                goto loop;
        }
```

```
        /*
         * see if there is core for that process
         */

findsp:
        spl0();
        rp = p1;
        a = rp->p_size;
        if((rp=rp->p_textp) != NULL)
                if(rp->x_ccount == 0)
                        a =+ rp->x_size;
        s = a;
        if((a=malloc(coremap, a)) != NULL)
                goto found2;

        /*
         * none found,
         * look around for easy core
         */

        spl6();
        for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
            ((rp->p_stat==SSLEEP && rp->p_pri>=0) || rp->p_stat==SSTOP)
                && (fspass || s <= rp->p_size))
                goto found1;

        /*
         * no easy core,
         * if this process is deserving,
         * look around for
         * oldest process in core
         */

        if(n < 3)
                goto sloop;
        n = -1;
        for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
           (rp->p_stat==SRUN || rp->p_stat==SSLEEP) &&
           rp->p_time > n && (fspass || (s<=rp->p_size))) {
                p2 = rp;
                n = rp->p_time;
        }
        if(n < 2)
                if (fspass) {
                        fspass = 0;
                        goto sloop;
                } else {
                        fspass++;
                        goto findsp;
                }
        rp = p2;

        /*
```

```
		 * swap user out
		 */

found1:
		spl0();
		rp->p_flag =& ~SLOAD;
		xswap(rp, 1, 0);
		goto findsp;

		/*
		 * swap user in
		 */

found2:
		if((rp=p1->p_textp) != NULL) {
			if(rp->x_ccount == 0) {
				if(swap(rp->x_daddr, a, rp->x_size, B_READ))
					goto swaper;
				rp->x_caddr = a;
				a =+ rp->x_size;
			}
			rp->x_ccount++;
		}
		rp = p1;
		if(swap(rp->p_addr, a, rp->p_size, B_READ))
			goto swaper;
		mfree(swapmap, (rp->p_size+7)/8, rp->p_addr);
		rp->p_addr = a;
		rp->p_flag =| SLOAD;
		rp->p_time = 0;
		goto loop;

swaper:
		panic("swap error");
}

/*
 * This routine is called to reschedule the CPU.
 * if the calling process is not in RUN state,
 * arrangements for it to restart must have
 * been made elsewhere, usually by calling via sleep.
 */
swtch()
{
		static struct proc *p;
		register i, n;
		register struct proc *rp;

		if(p == NULL)
			p = &proc[0];
		/*
		 * Remember stack of caller
		 */
		savu(u.u_rsav);
		/*
		 * Switch to scheduler's stack
```

```
         */
        retu(proc[0].p_addr);

loop:
        runrun = 0;
        rp = p;
        p = NULL;
        n = 128;
        /*
         * Search for highest-priority runnable process
         */
        i = NPROC;
        do {
                rp++;
                if(rp >= &proc[NPROC])
                        rp = &proc[0];
                if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)!=0) {
                        if(rp->p_pri < n) {
                                p = rp;
                                n = rp->p_pri;
                        }
                }
        } while(--i);
        /*
         * If no process is runnable, idle.
         */
        if(p == NULL) {
                p = rp;
                idle();
                goto loop;
        }
        rp = p;
        curpri = n;
        /*
         * Switch to stack of the new process and set up
         * his segmentation registers.
         */
        retu(rp->p_addr);
        sureg();
        /*
         * If the new process paused because it was
         * swapped out, set the stack level to the last call
         * to savu(u_ssav).  This means that the return
         * which is executed immediately after the call to aretu
         * actually returns from the last routine which did
         * the savu.
         *
         * You are not expected to understand this.
         */
        if(rp->p_flag&SSWAP) {
                rp->p_flag =& ~SSWAP;
                aretu(u.u_ssav);
        }
        /*
         * The value returned here has many subtle implications.
         * See the newproc comments.
```

```
         */
        return(1);
}

/*
 * Create a new process-- the internal version of
 * sys fork.
 * It returns 1 in the new process.
 * How this happens is rather hard to understand.
 * The essential fact is that the new process is created
 * in such a way that appears to have started executing
 * in the same call to newproc as the parent;
 * but in fact the code that runs is that of swtch.
 * The subtle implication of the returned value of swtch
 * (see above) is that this is the value that newproc's
 * caller in the new process sees.
 */
newproc()
{
        int a1, a2;
        struct proc *p, *up;
        register struct proc *rpp;
        register *rip, n;

        p = NULL;
        /*
         * First, just locate a slot for a process
         * and copy the useful info from this process into it.
         * The panic "cannot happen" because fork has already
         * checked for the existence of a slot.
         */
retry:
        mpid++;
        if(mpid < 0) {
                mpid = 0;
                goto retry;
        }
        for(rpp = &proc[0]; rpp < &proc[NPROC]; rpp++) {
                if(rpp->p_stat == NULL && p==NULL)
                        p = rpp;
                if (rpp->p_pid==mpid)
                        goto retry;
        }
        if ((rpp = p)==NULL)
                panic("no procs");

        /*
         * make proc entry for new proc
         */

        rip = u.u_procp;
        up = rip;
        rpp->p_stat = SRUN;
        rpp->p_flag = SLOAD;
        rpp->p_uid = rip->p_uid;
        rpp->p_pgrp = rip->p_pgrp;
```

```
        rpp->p_nice = rip->p_nice;
        rpp->p_textp = rip->p_textp;
        rpp->p_pid = mpid;
        rpp->p_ppid = rip->p_pid;
        rpp->p_time = 0;
        rpp->p_cpu = 0;

        /*
         * make duplicate entries
         * where needed
         */

        for(rip = &u.u_ofile[0]; rip < &u.u_ofile[NOFILE];)
                if((rpp = *rip++) != NULL)
                        rpp->f_count++;
        if((rpp=up->p_textp) != NULL) {
                rpp->x_count++;
                rpp->x_ccount++;
        }
        u.u_cdir->i_count++;
        /*
         * Partially simulate the environment
         * of the new process so that when it is actually
         * created (by copying) it will look right.
         */
        savu(u.u_rsav);
        rpp = p;
        u.u_procp = rpp;
        rip = up;
        n = rip->p_size;
        a1 = rip->p_addr;
        rpp->p_size = n;
        a2 = malloc(coremap, n);
        /*
         * If there is not enough core for the
         * new process, swap out the current process to generate the
         * copy.
         */
        if(a2 == NULL) {
                rip->p_stat = SIDL;
                rpp->p_addr = a1;
                savu(u.u_ssav);
                xswap(rpp, 0, 0);
                rpp->p_flag |= SSWAP;
                rip->p_stat = SRUN;
        } else {
        /*
         * There is core, so just copy.
         */
                rpp->p_addr = a2;
                while(n--)
                        copyseg(a1++, a2++);
        }
        u.u_procp = rip;
        return(0);
}
```

```
/*
 * Change the size of the data+stack regions of the process.
 * If the size is shrinking, it's easy-- just release the extra core.
 * If it's growing, and there is core, just allocate it
 * and copy the image, taking care to reset registers to account
 * for the fact that the system's stack has moved.
 * If there is no core, arrange for the process to be swapped
 * out after adjusting the size requirement-- when it comes
 * in, enough core will be allocated.
 * Because of the ssave and SSWAP flags, control will
 * resume after the swap in swtch, which executes the return
 * from this stack level.
 *
 * After the expansion, the caller will take care of copying
 * the user's stack towards or away from the data area.
 */
expand(newsize)
{
        int i, n;
        register *p, a1, a2;

        p = u.u_procp;
        n = p->p_size;
        p->p_size = newsize;
        a1 = p->p_addr;
        if(n >= newsize) {
                mfree(coremap, n-newsize, a1+newsize);
                return;
        }
        savu(u.u_rsav);
        a2 = malloc(coremap, newsize);
        if(a2 == NULL) {
                savu(u.u_ssav);
                xswap(p, 1, n);
                p->p_flag =| SSWAP;
                swtch();
                /* no return */
        }
        p->p_addr = a2;
        for(i=0; i<n; i++)
                copyseg(a1+i, a2++);
        mfree(coremap, n, a1);
        retu(p->p_addr);
        sureg();
}
```

```
#
#include "../param.h"
#include "../conf.h"
#include "../inode.h"
#include "../user.h"
#include "../buf.h"
#include "../systm.h"

/*
 * Bmap defines the structure of file system storage
 * by returning the physical block number on a device given the
 * inode and the logical block number in a file.
 * When convenient, it also leaves the physical
 * block number of the next block of the file in rablock
 * for use in read-ahead.
 */
bmap(ip, bn)
struct inode *ip;
int bn;
{
        register *bp, *bap, nb;
        int *nbp, d, i;

        d = ip->i_dev;
        if(bn & ~077777) {
                u.u_error = EFBIG;
                return(0);
        }

        if((ip->i_mode&ILARG) == 0) {

                /*
                 * small file algorithm
                 */

                if((bn & ~7) != 0) {

                        /*
                         * convert small to large
                         */

                        if ((bp = alloc(d)) == NULL)
                                return(NULL);
                        bap = bp->b_addr;
                        for(i=0; i<8; i++) {
                                *bap++ = ip->i_addr[i];
                                ip->i_addr[i] = 0;
                        }
                        ip->i_addr[0] = bp->b_blkno;
                        bdwrite(bp);
                        ip->i_mode =| ILARG;
                        goto large;
                }
                nb = ip->i_addr[bn];
                if(nb == 0 && (bp = alloc(d)) != NULL) {
                        bdwrite(bp);
```

```
                        nb = bp->b_blkno;
                        ip->i_addr[bn] = nb;
                        ip->i_flag =| IUPD;
                }
                rablock = 0;
                if (bn<7)
                        rablock = ip->i_addr[bn+1];
                return(nb);
        }

        /*
         * large file algorithm
         */

    large:
        i = bn>>8;
        if(bn & 0174000)
                i = 7;
        if((nb=ip->i_addr[i]) == 0) {
                ip->i_flag =| IUPD;
                if ((bp = alloc(d)) == NULL)
                        return(NULL);
                ip->i_addr[i] = bp->b_blkno;
        } else
                bp = bread(d, nb);
        bap = bp->b_addr;

        /*
         * "huge" fetch of double indirect block
         */

        if(i == 7) {
                i = ((bn>>8) & 0377) - 7;
                if((nb=bap[i]) == 0) {
                        if((nbp = alloc(d)) == NULL) {
                                brelse(bp);
                                return(NULL);
                        }
                        bap[i] = nbp->b_blkno;
                        bdwrite(bp);
                } else {
                        brelse(bp);
                        nbp = bread(d, nb);
                }
                bp = nbp;
                bap = bp->b_addr;
        }

        /*
         * normal indirect fetch
         */

        i = bn & 0377;
        if((nb=bap[i]) == 0 && (nbp = alloc(d)) != NULL) {
                nb = nbp->b_blkno;
                bap[i] = nb;
```

```
                bdwrite(nbp);
                bdwrite(bp);
        } else
                brelse(bp);
        rablock = 0;
        if(i < 255)
                rablock = bap[i+1];
        return(nb);
}

/*
 * Pass back  c  to the user at his location u_base;
 * update u_base, u_count, and u_offset.  Return -1
 * on the last character of the user's read.
 * u_base is in the user address space unless u_segflg is set.
 */
passc(c)
char c;
{

        if(u.u_segflg)
                *u.u_base = c; else
                if(subyte(u.u_base, c) < 0) {
                        u.u_error = EFAULT;
                        return(-1);
                }
        u.u_count--;
        if(++u.u_offset[1] == 0)
                u.u_offset[0]++;
        u.u_base++;
        return(u.u_count == 0? -1: 0);
}

/*
 * Pick up and return the next character from the user's
 * write call at location u_base;
 * update u_base, u_count, and u_offset.  Return -1
 * when u_count is exhausted.  u_base is in the user's
 * address space unless u_segflg is set.
 */
cpass()
{
        register c;

        if(u.u_count == 0)
                return(-1);
        if(u.u_segflg)
                c = *u.u_base; else
                if((c=fubyte(u.u_base)) < 0) {
                        u.u_error = EFAULT;
                        return(-1);
                }
        u.u_count--;
        if(++u.u_offset[1] == 0)
                u.u_offset[0]++;
        u.u_base++;
```

```
        return(c&0377);
}

/*
 * Routine which sets a user error; placed in
 * illegal entries in the bdevsw and cdevsw tables.
 */
nodev()
{

        u.u_error = ENODEV;
}

/*
 * Null routine; placed in insignificant entries
 * in the bdevsw and cdevsw tables.
 */
nulldev()
{
}

/*
 * copy count words from from to to.
 */
bcopy(from, to, count)
int *from, *to;
{
        register *a, *b, c;

        a = from;
        b = to;
        c = count;
        do
                *b++ = *a++;
        while(--c);
}
```

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../proc.h"
#include "../buf.h"
#include "../reg.h"
#include "../inode.h"

/*
 * exec system call.
 * Because of the fact that an I/O buffer is used
 * to store the caller's arguments during exec,
 * and more buffers are needed to read in the text file,
 * deadly embraces waiting for free buffers are possible.
 * Therefore the number of processes simultaneously
 * running in exec has to be limited to NEXEC.
 */
#define EXPRI   -1

exec()
{
        int ap, na, nc, *bp;
        int ts, ds, sep;
        register c, *ip;
        register char *cp;
        extern uchar;

        /*
         * pick up file names
         * and check various modes
         * for execute permission
         */

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        while(execnt >= NEXEC)
                sleep(&execnt, EXPRI);
        execnt++;
        bp = getblk(NODEV);
        if(access(ip, IEXEC))
                goto bad;
        if((ip->i_mode & IFMT) != 0 ||
           (ip->i_mode & (IEXEC|(IEXEC>>3)|(IEXEC>>6))) == 0) {
                u.u_error = EACCES;
                goto bad;
        }

        /*
         * pack up arguments into
         * allocated disk buffer
         */

        cp = bp->b_addr;
        na = 0;
```

steal a block used to co
from old core image

check

```
        nc = 0;
        while(ap = fuword(u.u_arg[1])) {
                na++;
                if(ap == -1)
                        goto bad;
                u.u_arg[1] =+ 2;
                for(;;) {
                        c = fubyte(ap++);
                        if(c == -1)
                                goto bad;
                        *cp++ = c;
                        nc++;
                        if(nc > 510) {
                                u.u_error = E2BIG;
                                goto bad;
                        }
                        if(c == 0)
                                break;
                }
        }
        if((nc&1) != 0) {
                *cp++ = 0;
                nc++;
        }

        /*
         * read in first 8 bytes
         * of file for segment
         * sizes:
         * w0 = 407/410/411 (410 implies RO text) (411 implies sep ID)
         * w1 = text size
         * w2 = data size
         * w3 = bss size
         */

        u.u_base = &u.u_arg[0];
        u.u_count = 8;
        u.u_offset[1] = 0;
        u.u_offset[0] = 0;
        u.u_segflg = 1;
        readi(ip);
        u.u_segflg = 0;
        if(u.u_error)
                goto bad;
        sep = 0;
        if(u.u_arg[0] == 0407) {
                u.u_arg[2] =+ u.u_arg[1];
                u.u_arg[1] = 0;
        } else
        if(u.u_arg[0] == 0411)
                sep++; else
        if(u.u_arg[0] != 0410) {
                u.u_error = ENOEXEC;
                goto bad;
        }
        if(u.u_arg[1]!=0 && (ip->i_flag&ITEXT)==0 && ip->i_count!=1) {
```

not executable

```
                u.u_error = ETXTBSY;
                goto bad;
        }

        /*
         * find text and data sizes
         * try them out for possible
         * exceed of max sizes
         */

        ts = ((u.u_arg[1]+63)>>6) & 01777;
        ds = ((u.u_arg[2]+u.u_arg[3]+63)>>6) & 01777;
        if(estabur(ts, ds, SSIZE, sep))
                goto bad;

        /*
         * allocate and clear core
         * at this point, committed
         * to the new image
         */

        u.u_prof[3] = 0;
        xfree();
        expand(USIZE);
        xalloc(ip);
        c = USIZE+ds+SSIZE;
        expand(c);
        while(--c >= USIZE)
                clearseg(u.u_procp->p_addr+c);

        /*
         * read in data segment
         */

        estabur(0, ds, 0, 0);
        u.u_base = 0;
        u.u_offset[1] = 020+u.u_arg[1];
        u.u_count = u.u_arg[2];
        readi(ip);

        /*
         * initialize stack segment
         */

        u.u_tsize = ts;
        u.u_dsize = ds;
        u.u_ssize = SSIZE;
        u.u_sep = sep;
        estabur(u.u_tsize, u.u_dsize, u.u_ssize, u.u_sep);
        cp = bp->b_addr;
        ap = -nc - na*2 - 4;
        u.u_ar0[R6] = ap;
        suword(ap, na);
        c = -nc;
        while(na--) {
                suword(ap=+2, c);
```

release old text seg
trunc to zero length
get new text seg
make it non zero length
} clear text seg

copy arguments to new image

```
                do
                        subyte(c++, *cp);
                while(*cp++);
        }
        suword(ap+2, -1);

        /*
         * set SUID/SGID protections, if no tracing
         */

        if ((u.u_procp->p_flag&STRC)==0) {
                if(ip->i_mode&ISUID)
                        if(u.u_uid != 0) {
                                u.u_uid = ip->i_uid;
                                u.u_procp->p_uid = ip->i_uid;
                        }
                if(ip->i_mode&ISGID)
                        u.u_gid = ip->i_gid;
        }

        /*
         * clear sigs, regs and return
         */

        c = ip;
        for(ip = &u.u_signal[0]; ip < &u.u_signal[NSIG]; ip++)
                if((*ip & 1) == 0)
                        *ip = 0;
        for(cp = &regloc[0]; cp < &regloc[6];)
                u.u_ar0[*cp++] = 0;
        u.u_ar0[R7] = 0;
        for(ip = &u.u_fsav[0]; ip < &u.u_fsav[25];)
                *ip++ = 0;
        ip = c;

bad:
        iput(ip);
        brelse(bp);
        if(execnt >= NEXEC)
                wakeup(&execnt);
        execnt--;
}

/*
 * exit system call:
 * pass back caller's r0
 */
rexit()
{

        u.u_arg[0] = u.u_ar0[R0] << 8;
        exit();
}

/*
 * Release resources.
```

clear pc

real exit

```
 * Save u. area for parent to look at.
 * Enter zombie state.
 * Wake up parent and init processes,
 * and dispose of children.
 */
exit()
{
        register int *q, a;
        register struct proc *p;

        p = u.u_procp;
        p->p_flag =& ~STRC;
        p->p_clktim = 0;
        for(q = &u.u_signal[0]; q < &u.u_signal[NSIG];)
                *q++ = 1;
        for(q = &u.u_ofile[0]; q < &u.u_ofile[NOFILE]; q++)
                if(a = *q) {
                        *q = NULL;
                        closef(a);
                }
        iput(u.u_cdir);
        xfree();
        a = malloc(swapmap, 1);
        if(a == NULL)
                panic("out of swap");
        p = getblk(swapdev, a);
        bcopy(&u, p->b_addr, 256);
        bwrite(p);
        q = u.u_procp;
        mfree(coremap, q->p_size, q->p_addr);
        q->p_addr = a;
        q->p_stat = SZOMB;

loop:
        for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(q->p_ppid == p->p_pid) {
                wakeup(&proc[1]);
                wakeup(p);
                for(p = &proc[0]; p < &proc[NPROC]; p++)
                if(q->p_pid == p->p_ppid) {
                        p->p_ppid  = 1;
                        if (p->p_stat == SSTOP)
                                setrun(p);
                }
                swtch();
                /* no return */
        }
        q->p_ppid = 1;
        goto loop;
}

/*
 * Wait system call.
 * Search for a terminated (zombie) child,
 * finally lay it to rest, and collect its status.
 * Look also for stopped (traced) children,
```

ignore sig

close open

save u. area

free core

find pare

```
 * and pass back status from them.
 */
wait()
{
        register f, *bp;
        register struct proc *p;

        f = 0;

loop:
        for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(p->p_ppid == u.u_procp->p_pid) {
                f++;
                if(p->p_stat == SZOMB) {
                        u.u_ar0[R0] = p->p_pid;
                        bp = bread(swapdev, f=p->p_addr);
                        mfree(swapmap, 1, f);
                        p->p_stat = NULL;
                        p->p_pid = 0;
                        p->p_ppid = 0;
                        p->p_sig = 0;
                        p->p_pgrp = 0;
                        p->p_flag = 0;
                        p = bp->b_addr;
                        u.u_cstime[0] =+ p->u_cstime[0];
                        dpadd(u.u_cstime, p->u_cstime[1]);
                        dpadd(u.u_cstime, p->u_stime);
                        u.u_cutime[0] =+ p->u_cutime[0];
                        dpadd(u.u_cutime, p->u_cutime[1]);
                        dpadd(u.u_cutime, p->u_utime);
                        u.u_ar0[R1] = p->u_arg[0];
                        brelse(bp);
                        return;
                }
                if(p->p_stat == SSTOP) {
                        if((p->p_flag&SWTED) == 0) {
                                p->p_flag =| SWTED;
                                u.u_ar0[R0] = p->p_pid;
                                u.u_ar0[R1] = (p->p_sig<<8) | 0177;
                                return;
                        }
                        p->p_flag =& ~(STRCISWTED);
                        setrun(p);
                }
        }
        if(f) {
                sleep(u.u_procp, PWAIT);
                goto loop;
        }
        u.u_error = ECHILD;
}

/*
 * fork system call.
 */
fork()
```

get time

trace

```
{
        register struct proc *p1, *p2, *p3;

        p1 = u.u_procp;
        p3 = &proc[NPROC-2];
        if (p1->p_uid==0)
                p3 = &proc[NPROC];
        for(p2 = &proc[0]; p2 < p3; p2++)
                if(p2->p_stat == NULL)
                        goto found;
        u.u_error = EAGAIN;
        goto out;

found:
        if(newproc()) {
                u.u_ar0[R0] = p1->p_pid;
                u.u_cstime[0] = 0;
                u.u_cstime[1] = 0;
                u.u_stime = 0;
                u.u_cutime[0] = 0;
                u.u_cutime[1] = 0;
                u.u_utime = 0;
                return;
        }
        u.u_ar0[R0] = p2->p_pid;

out:
        u.u_ar0[R7] =+ 2;
}

/*
 * break system call.
 *  -- bad planning: "break" is a dirty word in C.
 */
sbreak()
{
        register a, n, d;
        int i;

        /*
         * set n to new data size
         * set d to new-old
         * set n to new total size
         */

        n = (((u.u_arg[0]+63)>>6) & 01777);
        if(!u.u_sep)
                n =- nseg(u.u_tsize) * 128;
        if(n < 0)
                n = 0;
        d = n - u.u_dsize;
        n =+ USIZE+u.u_ssize;
        if(estabur(u.u_tsize, u.u_dsize+d, u.u_ssize, u.u_sep))
                return;
        u.u_dsize =+ d;
        if(d > 0)
```

```
                goto bigger;
        a = u.u_procp->p_addr + n - u.u_ssize;
        i = n;
        n = u.u_ssize;
        while(n--) {
                copyseg(a-d, a);
                a++;
        }
        expand(i);
        return;

bigger:
        expand(n);
        a = u.u_procp->p_addr + n;
        n = u.u_ssize;
        while(n--) {
                a--;
                copyseg(a-d, a);
        }
        while(d--)
                clearseg(--a);
}
```

) copy stack down

) copy stack up

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../reg.h"
#include "../file.h"
#include "../inode.h"

/*
 * read system call
 */
read()
{
        rdwr(FREAD);
}

/*
 * write system call
 */
write()
{
        rdwr(FWRITE);
}

/*
 * common code for read and write calls:
 * check permissions, set base, count, and offset,
 * and switch out to readi, writei, or pipe code.
 */
rdwr(mode)
{
        register *fp, m;

        m = mode;
        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        if((fp->f_flag&m) == 0) {
                u.u_error = EBADF;
                return;
        }
        u.u_base = u.u_arg[0];
        u.u_count = u.u_arg[1];
        u.u_segflg = 0;
        if(fp->f_flag&FPIPE) {
                if(m==FREAD)
                        readp(fp); else
                        writep(fp);
        } else {
                u.u_offset[1] = fp->f_offset[1];
                u.u_offset[0] = fp->f_offset[0];
                if(m==FREAD)
                        readi(fp->f_inode); else
                        writei(fp->f_inode);
                dpadd(fp->f_offset, u.u_arg[1]-u.u_count);
        }
```

pipe

inode

```
        u.u_ar0[R0] = u.u_arg[1]-u.u_count;
}

/*
 * open system call
 */
open()
{
        register *ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        u.u_arg[1]++;
        open1(ip, u.u_arg[1], 0);
}

/*
 * creat system call
 */
creat()
{
        register *ip;
        extern uchar;

        ip = namei(&uchar, 1);
        if(ip == NULL) {
                if(u.u_error)
                        return;
                ip = maknode(u.u_arg[1]&07777&(~ISVTX));
                if (ip==NULL)
                        return;
                open1(ip, FWRITE, 2);
        } else
                open1(ip, FWRITE, 1);
}

/*
 * common code for open and creat.
 * Check permissions, allocate an open file structure,
 * and call the device open routine if any.
 */
open1(ip, mode, trf)
int *ip;
{
        register struct file *fp;
        register *rip, m;
        int i;

        rip = ip;
        m = mode;
        if(trf != 2) {
                if(m&FREAD)
                        access(rip, IREAD);
                if(m&FWRITE) {
```

```
                        access(rip, IWRITE);
                        if((rip->i_mode&IFMT) == IFDIR)
                                u.u_error = EISDIR;
                }
        }
        if(u.u_error)
                goto out;
        if(trf)
                itrunc(rip);
        prele(rip);
        if ((fp = falloc()) == NULL)
                goto out;
        fp->f_flag = m&(FREAD|FWRITE);
        fp->f_inode = rip;
        i = u.u_ar0[R0];
        openi(rip, m&FWRITE);
        if(u.u_error == 0)
                return;
        u.u_ofile[i] = NULL;
        fp->f_count--;

out:
        iput(rip);
}

/*
 * close system call
 */
close()
{
        register *fp;

        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        u.u_ofile[u.u_ar0[R0]] = NULL;
        closef(fp);
}

/*
 * seek system call
 */
seek()
{
        int n[2];
        register *fp, t;

        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        if(fp->f_flag&FPIPE) {
                u.u_error = ESPIPE;
                return;
        }
        t = u.u_arg[1];
        if(t > 2) {
```

open if special file

fairly complex, mainly because of double prec

```
                n[1] = u.u_arg[0]<<9;
                n[0] = u.u_arg[0]>>7;
                if(t == 3)
                        n[0] =& 0777;
        } else {
                n[1] = u.u_arg[0];
                n[0] = 0;
                if(t!=0 && n[1]<0)
                        n[0] = -1;
        }
        switch(t) {

        case 1:
        case 4:
                n[0] =+ fp->f_offset[0];
                dpadd(n, fp->f_offset[1]);
                break;

        default:
                n[0] =+ fp->f_inode->i_size0&0377;
                dpadd(n, fp->f_inode->i_size1);

        case 0:
        case 3:
                ;
        }
        fp->f_offset[1] = n[1];
        fp->f_offset[0] = n[0];
}

/*
 * link system call
 */
link()
{
        register *ip, *xp;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if(ip->i_nlink == 255) {
                u.u_error = EMLINK;
                goto out;
        }
        if((ip->i_mode&IFMT)==IFDIR && !suser())
                goto out;
        /*
         * unlock to avoid possibly hanging the namei
         */
        ip->i_flag =& ~ILOCK;
        u.u_dirp = u.u_arg[1];
        xp = namei(&uchar, 1);
        if(xp != NULL) {
                u.u_error = EEXIST;
                iput(xp);
```

```
        }
        if(u.u_error)
                goto out;
        if(u.u_pdir->i_dev != ip->i_dev) {
                iput(u.u_pdir);
                u.u_error = EXDEV;
                goto out;
        }
        wdir(ip);
        ip->i_nlink++;
        ip->i_flag =| IUPD;

out:
        iput(ip);
}

/*
 * mknod system call
 */
mknod()
{
        register *ip;
        extern uchar;

        if(suser()) {
                ip = namei(&uchar, 1);
                if(ip != NULL) {
                        u.u_error = EEXIST;
                        goto out;
                }
        }
        if(u.u_error)
                return;
        ip = maknode(u.u_arg[1]);
        if (ip==NULL)
                return;
        ip->i_addr[0] = u.u_arg[2];

out:
        iput(ip);
}

/*
 * sleep system call
 * not to be confused with the sleep internal routine.
 */
sslep()
{
        char *d[2];

        spl7();
        d[0] = time[0];
        d[1] = time[1];
        dpadd(d, u.u_ar0[R0]);

        while(dpcmp(d[0], d[1], time[0], time[1]) > 0) {
```

exactly like create, but doesn throw away special bits

(not very
several pr
to sleep

```
                if(dpcmp(tout[0], tout[1], time[0], time[1]) <= 0 ||
                   dpcmp(tout[0], tout[1], d[0], d[1]) > 0) {
                        tout[0] = d[0];
                        tout[1] = d[1];
                }
                sleep(tout, PSLEP);
        }
        spl0();
}

/*
 * access system call
 */
saccess()
{
        extern uchar;
        register svuid, svgid;
        register *ip;

        svuid = u.u_uid;
        svgid = u.u_gid;
        u.u_uid = u.u_ruid;
        u.u_gid = u.u_rgid;
        ip = namei(&uchar, 0);
        if (ip != NULL) {
                if (u.u_arg[1]&(IREAD>>6))
                        access(ip, IREAD);
                if (u.u_arg[1]&(IWRITE>>6))
                        access(ip, IWRITE);
                if (u.u_arg[1]&(IEXEC>>6))
                        access(ip, IEXEC);
                iput(ip);
        }
        u.u_uid = svuid;
        u.u_gid = svgid;
}
```

"does a given user have the appropriate permission"

e.g., mail

```
#
#include "../param.h"
#include "../systm.h"
#include "../reg.h"
#include "../buf.h"
#include "../filsys.h"
#include "../user.h"
#include "../inode.h"
#include "../file.h"
#include "../conf.h"

/*
 * the fstat system call.
 */
fstat()
{
        register *fp;

        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        stat1(fp->f_inode, u.u_arg[0]);
}

/*
 * the stat system call.
 */
stat()
{
        register ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        stat1(ip, u.u_arg[1]);
        iput(ip);
}

/*
 * The basic routine for fstat and stat:
 * get the inode and pass appropriate parts back.
 */
stat1(ip, ub)
int *ip;
{
        register i, *bp, *cp;

        iupdat(ip, time);
        bp = bread(ip->i_dev, ldiv(ip->i_number+31, 16));
        cp = bp->b_addr + 32*lrem(ip->i_number+31, 16) + 24;
        ip = &(ip->i_dev);
        for(i=0; i<14; i++) {
                suword(ub, *ip++);
                ub =+ 2;
        }
```

```
        for(i=0; i<4; i++) {
                suword(ub, *cp++);
                ub =+ 2;
        }
        brelse(bp);
}

/*
 * the dup system call.
 */
dup()
{
        register i, *fp;

        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        if ((i = ufalloc()) < 0)
                return;
        u.u_ofile[i] = fp;
        fp->f_count++;
}

/*
 * the mount system call.
 */
smount()
{
        int d;
        register *ip;
        register struct mount *mp, *smp;
        extern uchar;

        d = getmdev();
        if(u.u_error)
                return;
        u.u_dirp = u.u_arg[1];
        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if(ip->i_count!=1 || (ip->i_mode&(IFBLK&IFCHR))!=0)
                goto out;
        smp = NULL;
        for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++) {
                if(mp->m_bufp != NULL) {
                        if(d == mp->m_dev)
                                goto out;
                } else
                if(smp == NULL)
                        smp = mp;
        }
        if(smp == NULL)
                goto out;
        (*bdevsw[d.d_major].d_open)(d, !u.u_arg[2]);
        if(u.u_error)
                goto out;
```

```
        mp = bread(d, 1);
        if(u.u_error) {
                brelse(mp);
                goto out1;
        }
        smp->m_inodp = ip;
        smp->m_dev = d;
        smp->m_bufp = getblk(NODEV);
        bcopy(mp->b_addr, smp->m_bufp->b_addr, 256);
        smp = smp->m_bufp->b_addr;
        smp->s_ilock = 0;
        smp->s_flock = 0;
        smp->s_ronly = u.u_arg[2] & 1;
        brelse(mp);
        ip->i_flag =| IMOUNT;
        prele(ip);
        return;

out:
        u.u_error = EBUSY;
out1:
        iput(ip);
}

/*
 * the umount system call.
 */
sumount()
{
        int d;
        register struct inode *ip;
        register struct mount *mp;

        update();
        d = getmdev();
        if(u.u_error)
                return;
        for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
                if(mp->m_bufp!=NULL && d==mp->m_dev)
                        goto found;
        u.u_error = EINVAL;
        return;

found:
        for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
                if(ip->i_number!=0 && d==ip->i_dev) {
                        u.u_error = EBUSY;
                        return;
                }
        (*bdevsw[d.d_major].d_close)(d, 0);
        ip = mp->m_inodp;
        ip->i_flag =& ~IMOUNT;
        iput(ip);
        ip = mp->m_bufp;
        mp->m_bufp = NULL;
        brelse(ip);
```

read super block

say that it's mounted

```
}

/*
 * Common code for mount and umount.
 * Check that the user's argument is a reasonable
 * thing on which to mount, and return the device number if so.
 */
getmdev()
{
        register d, *ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if((ip->i_mode&IFMT) != IFBLK)
                u.u_error = ENOTBLK;
        d = ip->i_addr[0];
        if(ip->i_addr[0].d_major >= nblkdev)
                u.u_error = ENXIO;
        iput(ip);
        return(d);
}
```

```
#
/*
 * Everything in this file is a routine implementing a system call.
 */

#include "../param.h"
#include "../user.h"
#include "../reg.h"
#include "../inode.h"
#include "../systm.h"
#include "../proc.h"

getswit()
{

        u.u_ar0[R0] = SW->integ;
}

gtime()
{

        u.u_ar0[R0] = time[0];
        u.u_ar0[R1] = time[1];
}

stime()
{

        if(suser()) {
                time[0] = u.u_ar0[R0];
                time[1] = u.u_ar0[R1];
                wakeup(tout);
        }
}

setuid()
{
        register uid;

        uid = u.u_ar0[R0].lobyte;
        if(u.u_ruid == uid.lobyte || suser()) {
                u.u_uid = uid;
                u.u_procp->p_uid = uid;
                u.u_ruid = uid;
        }
}

getuid()
{

        u.u_ar0[R0].lobyte = u.u_ruid;
        u.u_ar0[R0].hibyte = u.u_uid;
}

setgid()
{
```

TRASH ROUTINES

```
        register gid;

        gid = u.u_ar0[R0].lobyte;
        if(u.u_rgid == gid.lobyte || suser()) {
                u.u_gid = gid;
                u.u_rgid = gid;
        }
}

getgid()
{

        u.u_ar0[R0].lobyte = u.u_rgid;
        u.u_ar0[R0].hibyte = u.u_gid;
}

getpid()
{
        u.u_ar0[R0] = u.u_procp->p_pid;
}

sync()
{

        update();
}

nice()
{
        register n;

        n = u.u_ar0[R0];
        if(n > 20)
                n = 20;
        if(n < 0 && !suser())
                n = 0;
        u.u_procp->p_nice = n;
}

/*
 * Unlink system call.
 * panic: unlink -- "cannot happen"
 */
unlink()
{
        register *ip, *pp;
        extern uchar;

        pp = namei(&uchar, 2);
        if(pp == NULL)
                return;
        prele(pp);
        ip = iget(pp->i_dev, u.u_dent.u_ino);
        if(ip == NULL)
                goto out1;
        if((ip->i_mode&IFMT)==IFDIR && !suser())
```

```
                goto out;
        u.u_offset[1] =- DIRSIZ+2;
        u.u_base = &u.u_dent;
        u.u_count = DIRSIZ+2;
        u.u_dent.u_ino = 0;
        writei(pp);
        if (ip->i_nlink-- == 0)
                ip->i_nlink = 0;
        ip->i_flag =| IUPD;

out:
        iput(ip);
out1:
        iput(pp);
}

chdir()
{
        register *ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if((ip->i_mode&IFMT) != IFDIR) {
                u.u_error = ENOTDIR;
        bad:
                iput(ip);
                return;
        }
        if(access(ip, IEXEC))
                goto bad;
        iput(u.u_cdir);
        u.u_cdir = ip;
        prele(ip);
}

chmod()
{
        register *ip;

        if ((ip = owner()) == NULL)
                return;
        ip->i_mode =& ~07777;
        if (u.u_uid)
                u.u_arg[1] =& ~ISVTX;
        ip->i_mode =| u.u_arg[1]&07777;
        ip->i_flag =| IUPD;
        iput(ip);
}

chown()
{
        register *ip;

        if (!suser() || (ip = owner()) == NULL)
```

```
                return;
        ip->i_uid = u.u_arg[1].lobyte;
        ip->i_gid = u.u_arg[1].hibyte;
        ip->i_flag =| IUPD;
        iput(ip);
}

/*
 * Change modified date of file:
 * time to r0-r1; sys smdate; file
 * This call has been withdrawn because it messes up
 * incremental dumps (pseudo-old files aren't dumped).
 * It works though and you can uncomment it if you like.

smdate()
{
        register struct inode *ip;
        register int *tp;
        int tbuf[2];

        if ((ip = owner()) == NULL)
                return;
        ip->i_flag =| IUPD;
        tp = &tbuf[2];
        *--tp = u.u_ar0[R1];
        *--tp = u.u_ar0[R0];
        iupdat(ip, tp);
        ip->i_flag =& ~IUPD;
        iput(ip);
}
*/

ssig()
{
        register a;

        a = u.u_arg[0];
        if(a<=0 || a>=NSIG || a==SIGKIL) {
                u.u_error = EINVAL;
                return;
        }
        u.u_ar0[R0] = u.u_signal[a];
        u.u_signal[a] = u.u_arg[1];
        if(u.u_procp->p_sig == a)
                u.u_procp->p_sig = 0;
}

kill()
{
        register struct proc *p, *q;
        register a;
        int f;

        f = 0;
        a = u.u_ar0[R0];
        q = u.u_procp;
```

```
        for(p = &proc[0]; p < &proc[NPROC]; p++) {
                if(p == q)
                        continue;
                if(a != 0 && p->p_pid != a)
                        continue;
                if(a == 0 && (p->p_pgrp != q->p_pgrp || p <= &proc[1]))
                        continue;
                if(u.u_uid != 0 && u.u_uid != p->p_uid)
                        continue;
                f++;
                psignal(p, u.u_arg[0]);
        }
        if(f == 0)
                u.u_error = ESRCH;
}

times()
{
        register *p;

        for(p = &u.u_utime; p  < &u.u_utime+6;) {
                suword(u.u_arg[0], *p++);
                u.u_arg[0] =+ 2;
        }
}

profil()
{

        u.u_prof[0] = u.u_arg[0] & ~1;  /* base of sample buf */
        u.u_prof[1] = u.u_arg[1];       /* size of same */
        u.u_prof[2] = u.u_arg[2];       /* pc offset */
        u.u_prof[3] = (u.u_arg[3]>>1) & 077777; /* pc scale */
}

/*
 * alarm clock signal
 */
alarm()
{
        register c, *p;

        p = u.u_procp;
        c = p->p_clktim;
        p->p_clktim = u.u_ar0[R0];
        u.u_ar0[R0] = c;
}

/*
 * indefinite wait.
 * no one should wakeup(&u)
 */
pause()
{

        for(;;)
```

```
                sleep(&u, PSLEP);
}
```

```
#
/*
 * This table is the switch used to transfer
 * to the appropriate routine for processing a system call.
 * Each row contains the number of arguments expected
 * and a pointer to the routine.
 */
int     sysent[]
{
        0, &nullsys,                    /*  0 = indir */
        0, &rexit,                      /*  1 = exit */
        0, &fork,                       /*  2 = fork */
        2, &read,                       /*  3 = read */
        2, &write,                      /*  4 = write */
        2, &open,                       /*  5 = open */
        0, &close,                      /*  6 = close */
        0, &wait,                       /*  7 = wait */
        2, &creat,                      /*  8 = creat */
        2, &link,                       /*  9 = link */
        1, &unlink,                     /* 10 = unlink */
        2, &exec,                       /* 11 = exec */
        1, &chdir,                      /* 12 = chdir */
        0, &gtime,                      /* 13 = time */
        3, &mknod,                      /* 14 = mknod */
        2, &chmod,                      /* 15 = chmod */
        2, &chown,                      /* 16 = chown */
        1, &sbreak,                     /* 17 = break */
        2, &stat,                       /* 18 = stat */
        2, &seek,                       /* 19 = seek */
        0, &getpid,                     /* 20 = getpid */
        3, &smount,                     /* 21 = mount */
        1, &sumount,                    /* 22 = umount */
        0, &setuid,                     /* 23 = setuid */
        0, &getuid,                     /* 24 = getuid */
        0, &stime,                      /* 25 = stime */
        3, &ptrace,                     /* 26 = ptrace */
        0, &alarm,                      /* 27 = alarm */
        1, &fstat,                      /* 28 = fstat */
        0, &pause,                      /* 29 = pause */
        1, &nullsys,                    /* 30 = smdate; inoperative */
        1, &stty,                       /* 31 = stty */
        1, &gtty,                       /* 32 = gtty */
        2, &saccess,                    /* 33 = access */
        0, &nice,                       /* 34 = nice */
        0, &sslep,                      /* 35 = sleep */
        0, &sync,                       /* 36 = sync */
        1, &kill,                       /* 37 = kill */
        0, &getswit,                    /* 38 = switch */
        0, &nosys,                      /* 39 = x */
        0, &nosys,                      /* 40 = x */
        0, &dup,                        /* 41 = dup */
        0, &pipe,                       /* 42 = pipe */
        1, &times,                      /* 43 = times */
        4, &profil,                     /* 44 = prof */
        0, &nosys,                      /* 45 = tiu */
        0, &setgid,                     /* 46 = setgid */
```

```
        0, &getgid,                     /* 47 = getgid */
        2, &ssig,                       /* 48 = sig */
        0, &nosys,                      /* 49 = x */
        0, &nosys,                      /* 50 = x */
        0, &nosys,                      /* 51 = x */
        0, &nosys,                      /* 52 = x */
        0, &nosys,                      /* 53 = x */
        0, &nosys,                      /* 54 = x */
        0, &nosys,                      /* 55 = x */
        0, &nosys,                      /* 56 = x */
        0, &nosys,                      /* 57 = x */
        0, &nosys,                      /* 58 = x */
        0, &nosys,                      /* 59 = x */
        0, &nosys,                      /* 60 = x */
        0, &nosys,                      /* 61 = x */
        0, &nosys,                      /* 62 = x */
        0, &nosys                       /* 63 = x */
};
```

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../proc.h"
#include "../text.h"
#include "../inode.h"

/*
 * Swap out process p.
 * The ff flag causes its core to be freed--
 * it may be off when called to create an image for a
 * child process in newproc.
 * Os is the old size of the data area of the process,
 * and is supplied during core expansion swaps.
 *
 * panic: out of swap space
 * panic: swap error -- IO error
 */
xswap(p, ff, os)
int *p;
{
        register *rp, a;

        rp = p;
        if(os == 0)
                os = rp->p_size;
        a = malloc(swapmap, (rp->p_size+7)/8);
        if(a == NULL)
                panic("out of swap space");
        xccdec(rp->p_textp);
        rp->p_flag =| SLOCK;
        if(swap(a, rp->p_addr, os, 0))
                panic("swap error");
        if(ff)
                mfree(coremap, os, rp->p_addr);
        rp->p_addr = a;
        rp->p_flag =& ~(SLOADISLOCK);
        rp->p_time = 0;
        if(runout) {
                runout = 0;
                wakeup(&runout);
        }
}

/*
 * relinquish use of the shared text segment
 * of a process.
 */
xfree()
{
        register *xp, *ip;

        if((xp=u.u_procp->p_textp) != NULL) {
                u.u_procp->p_textp = NULL;
                xccdec(xp);
```

called each tim
a given st

↖ core count

```
                if(--xp->x_count == 0) {
                        ip = xp->x_iptr;
                        if((ip->i_mode&ISVTX) == 0) {
                                xp->x_iptr = NULL;
                                mfree(swapmap, (xp->x_size+7)/8, xp->x_daddr);
                                ip->i_flag =& ~ITEXT;
                                iput(ip);
                        }
                }
        }
}

/*
 * Attach to a shared text segment.
 * If there is no shared text, just return.
 * If there is, hook up to it:
 * if it is not currently being used, it has to be read
 * in from the inode (ip) and established in the swap space.
 * If it is being used, but is not currently in core,
 * a swap has to be done to get it back.
 * The full coroutine glory has to be invoked--
 * see slp.c-- because if the calling process
 * is misplaced in core the text image might not fit.
 * Quite possibly the code after "out:" could check to
 * see if the text does fit and simply swap it in.
 *
 * panic: out of swap space
 */
xalloc(ip)
int *ip;
{
        register struct text *xp;
        register *rp, ts;

        if(u.u_arg[1] == 0)
                return;
        rp = NULL;
        for(xp = &text[0]; xp < &text[NTEXT]; xp++)
                if(xp->x_iptr == NULL) {
                        if(rp == NULL)
                                rp = xp;
                } else
                        if(xp->x_iptr == ip) {
                                xp->x_count++;
                                u.u_procp->p_textp = xp;
                                goto out;
                        }
        if((xp=rp) == NULL)
                panic("out of text");
        xp->x_count = 1;
        xp->x_ccount = 0;
        xp->x_iptr = ip;
        ts = ((u.u_arg[1]+63)>>6) & 01777;
        xp->x_size = ts;
        if((xp->x_daddr = malloc(swapmap, (ts+7)/8)) == NULL)
                panic("out of swap space");
```

```
        expand(USIZE+ts);
        estabur(0, ts, 0, 0);
        u.u_count = u.u_arg[1];
        u.u_offset[1] = 020;
        u.u_base = 0;
        readi(ip);
        rp = u.u_procp;
        rp->p_flag =| SLOCK;
        swap(xp->x_daddr, rp->p_addr+USIZE, ts, 0);
        rp->p_flag =& ~SLOCK;
        rp->p_textp = xp;
        rp = ip;
        rp->i_flag =| ITEXT;
        rp->i_count++;
        expand(USIZE);

out:
        if(xp->x_ccount == 0) {
                savu(u.u_rsav);
                savu(u.u_ssav);
                xswap(u.u_procp, 1, 0);
                u.u_procp->p_flag =| SSWAP;
                swtch();
                /* no return */
        }
        xp->x_ccount++;
}

/*
 * Decrement the in-core usage count of a shared text segment.
 * When it drops to zero, free the core space.
 */
xccdec(xp)
int *xp;
{
        register *rp;

        if((rp=xp)!=NULL && rp->x_ccount!=0)
                if(--rp->x_ccount == 0)
                        mfree(coremap, rp->x_size, rp->x_caddr);
}
```

scheduler

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../proc.h"
#include "../reg.h"
#include "../seg.h"

#define EBIT    1               /* user error bit in PS: C-bit */
#define UMODE   0170000         /* user-mode bits in PS word */
#define SETD    0170011         /* SETD instruction */
#define SYS     0104400         /* sys (trap) instruction */
#define USER    020             /* user-mode flag added to dev */

/*
 * structure of the system entry table (sysent.c)
 */
struct sysent   {
        int     count;          /* argument count */
        int     (*call)();      /* name of handler */
} sysent[64];

/*
 * Offsets of the user's registers relative to
 * the saved r0. See reg.h
 */
char    regloc[9]
{
        R0, R1, R2, R3, R4, R5, R6, R7, RPS
};

/*
 * Called from 140.s or 145.s when a processor trap occurs.
 * The arguments are the words saved on the system stack
 * by the hardware and software during the trap processing.
 * Their order is dictated by the hardware and the details
 * of C's calling sequence. They are peculiar in that
 * this call is not 'by value' and changed user registers
 * get copied back on return.
 * dev is the kind of trap that occurred.
 */
trap(dev, sp, r1, nps, r0, pc, ps)
{
        register i, a;
        register struct sysent *callp;

        savfp();
        if ((ps&UMODE) == UMODE)
                dev =| USER;
        u.u_ar0 = &r0;
        switch(dev) {

        /*
         * Trap not expected.
         * Usually a kernel mode bus error.
         * The numbers printed are used to
```

save the floating pt state (hard as hell)

```
         * find the hardware PS/PC as follows.
         * (all numbers in octal 18 bits)
         *      address_of_saved_ps =
         *              (ka6*0100) + aps - 0140000;
         *      address_of_saved_pc =
         *              address_of_saved_ps - 2;
         */
        default:
                printf("ka6 = %o\n", *ka6);
                printf("aps = %o\n", &ps);
                printf("trap type %o\n", dev);
                panic("trap");

        case 0+USER: /* bus error */
                i = SIGBUS;
                break;

        /*
         * If illegal instructions are not
         * being caught and the offending instruction
         * is a SETD, the trap is ignored.
         * This is because C produces a SETD at
         * the beginning of every program which
         * will trap on CPUs without 11/45 FPU.
         */
        case 1+USER: /* illegal instruction */
                if(fuiword(pc-2) == SETD && u.u_signal[SIGINS] == 0)
                        goto out;
                i = SIGINS;
                break;

        case 2+USER: /* bpt or trace */
                i = SIGTRC;
                break;

        case 3+USER: /* iot */
                i = SIGIOT;
                break;

        case 5+USER: /* emt */
                i = SIGEMT;
                break;

        case 6+USER: /* sys call */
                u.u_error = 0;
                ps =& ~EBIT;
                callp = &sysent[fuiword(pc-2)&077];
                if (callp == sysent) { /* indirect */
                        a = fuiword(pc);
                        pc =+ 2;
                        i = fuword(a);
                        if ((i & ~077) != SYS)
                                i = 077;        /* illegal */
                        callp = &sysent[i&077];
                        for(i=0; i<callp->count; i++)
                                u.u_arg[i] = fuword(a =+ 2);
```

```
        } else {
                for(i=0; i<callp->count; i++) {
                        u.u_arg[i] = fuiword(pc);
                        pc =+ 2;
                }
        }
        u.u_dirp = u.u_arg[0];
        trap1(callp->call);
        if(u.u_intflg)
                u.u_error = EINTR;
        if(u.u_error < 100) {
                if(u.u_error) {
                        ps =| EBIT;
                        r0 = u.u_error;
                }
                goto out;
        }
        i = SIGSYS;
        break;

/*
 * Since the floating exception is an
 * imprecise trap, a user generated
 * trap may actually come from kernel
 * mode. In this case, a signal is sent
 * to the current process to be picked
 * up later.
 */
case 8: /* floating exception */
        psignal(u.u_procp, SIGFPT);
        return;

case 8+USER:
        i = SIGFPT;
        break;

/*
 * If the user SP is below the stack segment,
 * grow the stack automatically.
 * This relies on the ability of the hardware
 * to restart a half executed instruction.
 * On the 11/40 this is not the case and
 * the routine backup/140.s may fail.
 * The classic example is on the instruction
 *      cmp     -(sp),-(sp)
 */
case 9+USER: /* segmentation exception */
        a = sp;
        if(backup(u.u_ar0) == 0)
        if(grow(a))
                goto out;
        i = SIGSEG;
        break;
}
psignal(u.u_procp, i);
```

do the sys call

int out of system ca
it was completed

in the system

not in system

'grow()' grows the stack

send a signal to the process

```
out:
        if(issig())
                psig();
        setpri(u.u_procp);
}

/*
 * Call the system-entry routine f (out of the
 * sysent table). This is a subroutine for trap, and
 * not in-line, because if a signal occurs
 * during processing, an (abnormal) return is simulated from
 * the last caller to savu(qsav); if this took place
 * inside of trap, it wouldn't have a chance to clean up.
 *
 * If this occurs, the return takes place without
 * clearing u_intflg; if it's still set, trap
 * marks an error which means that a system
 * call (like read on a typewriter) got interrupted
 * by a signal.
 */
trap1(f)
int (*f)();
{

        u.u_intflg = 1;
        savu(u.u_qsav);
        (*f)();
        u.u_intflg = 0;
}

/*
 * nonexistent system call-- set fatal error code.
 */
nosys()
{
        u.u_error = 100;
}

/*
 * Ignored system call
 */
nullsys()
{
}
```

"if there is a signal, process the

sets user priority to 100 + ε

tells you if you interrupted it or if it returned normally out of trap1

savu — sto

excep

retu — la

loo

```
#
/*
 */

#include "../param.h"
#include "../user.h"
#include "../buf.h"
#include "../conf.h"
#include "../systm.h"
#include "../proc.h"
#include "../seg.h"

/*
 * This is the set of buffers proper, whose heads
 * were declared in buf.h.  There can exist buffer
 * headers not pointing here that are used purely
 * as arguments to the I/O routines to describe
 * I/O to be done-- e.g. swbuf, just below, for
 * swapping.
 */
char    buffers[NBUF][514];
struct  buf     swbuf;

/*
 * Declarations of the tables for the magtape devices;
 * see bdwrite.
 */
int     tmtab;
int     httab;

/*
 * The following several routines allocate and free
 * buffers with various side effects.  In general the
 * arguments to an allocate routine are a device and
 * a block number, and the value is a pointer to
 * to the buffer header; the buffer is marked "busy"
 * so that no on else can touch it.  If the block was
 * already in core, no I/O need be done; if it is
 * already busy, the process waits until it becomes free.
 * The following routines allocate a buffer:
 *      getblk
 *      bread
 *      breada
 * Eventually the buffer must be released, possibly with the
 * side effect of writing it out, by using one of
 *      bwrite
 *      bdwrite
 *      bawrite
 *      brelse
 */

/*
 * Read in (if necessary) the block and return a buffer pointer.
 */
bread(dev, blkno)
{
```

```
        register struct buf *rbp;

        rbp = getblk(dev, blkno);
        if (rbp->b_flags&B_DONE)
                return(rbp);
        rbp->b_flags =| B_READ;
        rbp->b_wcount = -256;
        (*bdevsw[dev.d_major].d_strategy)(rbp);
        iowait(rbp);
        return(rbp);
}

/*
 * Read in the block, like bread, but also start I/O on the
 * read-ahead block (which is not allocated to the caller)
 */
breada(adev, blkno, rablkno)
{
        register struct buf *rbp, *rabp;
        register int dev;

        dev = adev;
        rbp = 0;
        if (!incore(dev, blkno)) {
                rbp = getblk(dev, blkno);
                if ((rbp->b_flags&B_DONE) == 0) {
                        rbp->b_flags =| B_READ;
                        rbp->b_wcount = -256;
                        (*bdevsw[adev.d_major].d_strategy)(rbp);
                }
        }
        if (rablkno && !incore(dev, rablkno)) {
                rabp = getblk(dev, rablkno);
                if (rabp->b_flags & B_DONE)
                        brelse(rabp);
                else {
                        rabp->b_flags =| B_READ|B_ASYNC;
                        rabp->b_wcount = -256;
                        (*bdevsw[adev.d_major].d_strategy)(rabp);
                }
        }
        if (rbp==0)
                return(bread(dev, blkno));
        iowait(rbp);
        return(rbp);
}

/*
 * Write the buffer, waiting for completion.
 * Then release the buffer.
 */
bwrite(bp)
struct buf *bp;
{
        register struct buf *rbp;
        register flag;
```

```
        rbp = bp;
        flag = rbp->b_flags;
        rbp->b_flags =& ~(B_READ | B_DONE | B_ERROR | B_DELWRI);
        rbp->b_wcount = -256;
        (*bdevsw[rbp->b_dev.d_major].d_strategy)(rbp);
        if ((flag&B_ASYNC) == 0) {
                iowait(rbp);
                brelse(rbp);
        } else if ((flag&B_DELWRI)==0)
                geterror(rbp);
}

/*
 * Release the buffer, marking it so that if it is grabbed
 * for another purpose it will be written out before being
 * given up (e.g. when writing a partial block where it is
 * assumed that another write for the same block will soon follow).
 * This can't be done for magtape, since writes must be done
 * in the same order as requested.
 */
bdwrite(bp)
struct buf *bp;
{
        register struct buf *rbp;
        register struct devtab *dp;

        rbp = bp;
        dp = bdevsw[rbp->b_dev.d_major].d_tab;
        if (dp == &tmtab || dp == &httab)
                bawrite(rbp);
        else {
                rbp->b_flags =| B_DELWRI | B_DONE;
                brelse(rbp);
        }
}

/*
 * Release the buffer, start I/O on it, but don't wait for completion.
 */
bawrite(bp)
struct buf *bp;
{
        register struct buf *rbp;

        rbp = bp;
        rbp->b_flags =| B_ASYNC;
        bwrite(rbp);
}

/*
 * release the buffer, with no I/O implied.
 */
brelse(bp)
struct buf *bp;
{
```

```
        register struct buf *rbp, **backp;
        register int sps;

        rbp = bp;
        if (rbp->b_flags&B_WANTED)
                wakeup(rbp);
        if (bfreelist.b_flags&B_WANTED) {
                bfreelist.b_flags =& ~B_WANTED;
                wakeup(&bfreelist);
        }
        if (rbp->b_flags&B_ERROR)
                rbp->b_dev.d_minor = -1;  /* no assoc. on error */
        backp = &bfreelist.av_back;
        sps = PS->integ;
        spl6();
        rbp->b_flags =& ~(B_WANTED|B_BUSY|B_ASYNC);
        (*backp)->av_forw = rbp;
        rbp->av_back = *backp;
        *backp = rbp;
        rbp->av_forw = &bfreelist;
        PS->integ = sps;
}

/*
 * See if the block is associated with some buffer
 * (mainly to avoid getting hung up on a wait in breada)
 */
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);
}

/*
 * Assign a buffer for the given block.  If the appropriate
 * block is already associated, return it; otherwise search
 * for the oldest non-busy buffer and reassign it.
 * When a 512-byte area is wanted for some random reason
 * (e.g. during exec, for the user arglist) getblk can be called
 * with device NODEV to avoid unwanted associativity.
 */
getblk(dev, blkno)
{
        register struct buf *bp;
        register struct devtab *dp;
        extern lbolt;

        if(dev.d_major >= nblkdev)
```

```
                panic("blkdev");

    loop:
        if (dev < 0)
                dp = &bfreelist;
        else {
                dp = bdevsw[dev.d_major].d_tab;
                if(dp == NULL)
                        panic("devtab");
                for (bp=dp->b_forw; bp != dp; bp = bp->b_forw) {
                        if (bp->b_blkno!=blkno || bp->b_dev!=dev)
                                continue;
                        spl6();
                        if (bp->b_flags&B_BUSY) {
                                bp->b_flags =| B_WANTED;
                                sleep(bp, PRIBIO);
                                spl0();
                                goto loop;
                        }
                        spl0();
                        notavail(bp);
                        return(bp);
                }
        }
        spl6();
        if (bfreelist.av_forw == &bfreelist) {
                bfreelist.b_flags =| B_WANTED;
                sleep(&bfreelist, PRIBIO);
                spl0();
                goto loop;
        }
        spl0();
        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;
        dp->b_forw->b_back = bp;
        dp->b_forw = bp;
        bp->b_dev = dev;
        bp->b_blkno = blkno;
        return(bp);
}

/*
 * Wait for I/O completion on the buffer; return errors
 * to the user.
 */
iowait(bp)
struct buf *bp;
```

```
{
        register struct buf *rbp;

        rbp = bp;
        spl6();
        while ((rbp->b_flags&B_DONE)==0)
                sleep(rbp, PRIBIO);
        spl0();
        geterror(rbp);
}

/*
 * Unlink a buffer from the available list and mark it busy.
 * (internal interface)
 */
notavail(bp)
struct buf *bp;
{
        register struct buf *rbp;
        register int sps;

        rbp = bp;
        sps = PS->integ;
        spl6();
        rbp->av_back->av_forw = rbp->av_forw;
        rbp->av_forw->av_back = rbp->av_back;
        rbp->b_flags =| B_BUSY;
        PS->integ = sps;
}

/*
 * Mark I/O complete on a buffer, release it if I/O is asynchronous,
 * and wake up anyone waiting for it.
 */
iodone(bp)
struct buf *bp;
{
        register struct buf *rbp;

        rbp = bp;
        if(rbp->b_flags&B_MAP)
                mapfree(rbp);
        rbp->b_flags =| B_DONE;
        if (rbp->b_flags&B_ASYNC)
                brelse(rbp);
        else {
                rbp->b_flags =& ~B_WANTED;
                wakeup(rbp);
        }
}

/*
 * Zero the core associated with a buffer.
 */
clrbuf(bp)
int *bp;
```

```
{
        register *p;
        register c;

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

/*
 * Initialize the buffer I/O system by freeing
 * all buffers and setting all device buffer lists to empty.
 */
binit()
{
        register struct buf *bp;
        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;
}

/*
 * Device start routine for disks
 * and other devices that have the register
 * layout of the older DEC controllers (RF, RK, RP, TM)
 */
#define IENABLE 0100
#define WCOM    02
#define RCOM    04
#define GO      01
```

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

        dp = devloc;
        rbp = bp;
        *dp = devblk;                   /* block address */
        *--dp = rbp->b_addr;            /* buffer address */
        *--dp = rbp->b_wcount;          /* word count */
        com = (hbcom<<8) | IENABLE | GO |
                ((rbp->b_xmem & 03) << 4);
        if (rbp->b_flags&B_READ)        /* command + x-mem */
                com =| RCOM;
        else
                com =| WCOM;
        *--dp = com;
}

/*
 * startup routine for RH controllers.
 */
#define RHWCOM  060
#define RHRCOM  070

rhstart(bp, devloc, devblk, abae)
struct buf *bp;
int *devloc, *abae;
{
        register int *dp;
        register struct buf *rbp;
        register int com;

        dp = devloc;
        rbp = bp;
        if(cputype == 70)
                *abae = rbp->b_xmem;
        *dp = devblk;                   /* block address */
        *--dp = rbp->b_addr;            /* buffer address */
        *--dp = rbp->b_wcount;          /* word count */
        com = IENABLE | GO |
                ((rbp->b_xmem & 03) << 8);
        if (rbp->b_flags&B_READ)        /* command + x-mem */
                com =| RHRCOM; else
                com =| RHWCOM;
        *--dp = com;
}

/*
 * 11/70 routine to allocate the
 * UNIBUS map and initialize for
 * a unibus device.
 * The code here and in
```

```
 * rhstart assumes that an rh on an 11/70
 * is an rh70 and contains 22 bit addressing.
 */
int     maplock;
mapalloc(abp)
struct buf *abp;
{
        register i, a;
        register struct buf *bp;

        if(cputype != 70)
                return;
        spl6();
        while(maplock&B_BUSY) {
                maplock =| B_WANTED;
                sleep(&maplock, PSWP);
        }
        maplock =| B_BUSY;
        spl0();
        bp = abp;
        bp->b_flags =| B_MAP;
        a = bp->b_xmem;
        for(i=16; i<32; i=+2)
                UBMAP->r[i+1] = a;
        for(a++; i<48; i=+2)
                UBMAP->r[i+1] = a;
        bp->b_xmem = 1;
}

mapfree(bp)
struct buf *bp;
{

        bp->b_flags =& ~B_MAP;
        if(maplock&B_WANTED)
                wakeup(&maplock);
        maplock = 0;
}

/*
 * swap I/O
 */
swap(blkno, coreaddr, count, rdflg)
{
        register int *fp;

        fp = &swbuf.b_flags;
        spl6();
        while (*fp&B_BUSY) {
                *fp =| B_WANTED;
                sleep(fp, PSWP);
        }
        *fp = B_BUSY | B_PHYS | rdflg;
        swbuf.b_dev = swapdev;
        swbuf.b_wcount = - (count<<5);  /* 32 w/block */
        swbuf.b_blkno = blkno;
```

non-interrupt code

interrupt code

```
        swbuf.b_addr = coreaddr<<6;     /* 64 b/block */
        swbuf.b_xmem = (coreaddr>>10) & 077;
        (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
        spl6();
        while((*fp&B_DONE)==0)
                sleep(fp, PSWP);
        if (*fp&B_WANTED)
                wakeup(fp);
        spl0();
        *fp =& ~(B_BUSY|B_WANTED);
        return(*fp&B_ERROR);
}

/*
 * make sure all write-behind blocks
 * on dev (or NODEV for all)
 * are flushed out.
 * (from umount and update)
 */
bflush(dev)
{
        register struct buf *bp;

loop:
        spl6();
        for (bp = bfreelist.av_forw; bp != &bfreelist; bp = bp->av_forw) {
                if (bp->b_flags&B_DELWRI && (dev == NODEV||dev==bp->b_dev)) {
                        bp->b_flags =| B_ASYNC;
                        notavail(bp);
                        bwrite(bp);
                        goto loop;
                }
        }
        spl0();
}

/*
 * Raw I/O. The arguments are
 *      The strategy routine for the device
 *      A buffer, which will always be a special buffer
 *        header owned exclusively by the device for this purpose
 *      The device number
 *      Read/write flag
 * Essentially all the work is computing physical addresses and
 * validating them.
 */
physio(strat, abp, dev, rw)
struct buf *abp;
int (*strat)();
{
        register struct buf *bp;
        register char *base;
        register int nb;
        int ts;

        bp = abp;
```

essentially "sys sync"

```
        base = u.u_base;
        /*
         * Check odd base, odd count, and address wraparound
         */
        if (base&01 || u.u_count&01 || base>=base+u.u_count)
                goto bad;
        ts = (u.u_tsize+127) & ~0177;
        if (u.u_sep)
                ts = 0;
        nb = (base>>6) & 01777;
        /*
         * Check overlap with text. (ts and nb now
         * in 64-byte clicks)
         */
        if (nb < ts)
                goto bad;
        /*
         * Check that transfer is either entirely in the
         * data or in the stack: that is, either
         * the end is in the data or the start is in the stack
         * (remember wraparound was already checked).
         */
        if ((((base+u.u_count)>>6)&01777) >= ts+u.u_dsize
            && nb < 1024-u.u_ssize)
                goto bad;
        spl6();
        while (bp->b_flags&B_BUSY) {
                bp->b_flags |= B_WANTED;
                sleep(bp, PRIBIO);
        }
        bp->b_flags = B_BUSY | B_PHYS | rw;
        bp->b_dev = dev;
        /*
         * Compute physical address by simulating
         * the segmentation hardware.
         */
        bp->b_addr = base&077;
        base = (u.u_sep? UDSA: UISA)->r[nb>>7] + (nb&0177);
        bp->b_addr +=+ base<<6;
        bp->b_xmem = (base>>10) & 077;
        bp->b_blkno = lshift(u.u_offset, -9);
        bp->b_wcount = -((u.u_count>>1) & 077777);
        bp->b_error = 0;
        u.u_procp->p_flag |= SLOCK;
        (*strat)(bp);
        spl6();
        while ((bp->b_flags&B_DONE) == 0)
                sleep(bp, PRIBIO);
        u.u_procp->p_flag &= ~SLOCK;
        if (bp->b_flags&B_WANTED)
                wakeup(bp);
        spl0();
        bp->b_flags &= ~(B_BUSY|B_WANTED);
        u.u_count = (-bp->b_resid)<<1;
        geterror(bp);
        return;
```

```
    bad:
        u.u_error = EFAULT;
}

/*
 * Pick up the device's error number and pass it to the user;
 * if there is an error but the number is 0 set a generalized
 * code.  Actually the latter is always true because devices
 * don't yet return specific errors.
 */
geterror(abp)
struct buf *abp;
{
        register struct buf *bp;

        bp = abp;
        if (bp->b_flags&B_ERROR)
                if ((u.u_error = bp->b_error)==0)
                        u.u_error = EIO;
}
```

```
#
/*
 */

/*
 *   KL/DL-11 driver
 */
#include "../param.h"
#include "../conf.h"
#include "../user.h"
#include "../tty.h"

/* base address */
#define KLADDR  0177560 /* console */
#define KLBASE  0176500 /* kl and dl11-a */
#define DLBASE  0175610 /* dl-e */
#define NKL11   1
#define NDL11   0
#define DSRDY   02
#define RDRENB  01

struct  tty kl11[NKL11+NDL11];

struct klregs {
        int klrcsr;
        int klrbuf;
        int kltcsr;
        int kltbuf;
};

klopen(dev, flag)
{
        register char *addr;
        register struct tty *tp;

        if(dev.d_minor >= NKL11+NDL11) {
                u.u_error = ENXIO;
                return;
        }
        tp = &kl11[dev.d_minor];
        /*
         * set up minor 0 to address KLADDR
         * set up minor 1 thru NKL11-1 to address from KLBASE
         * set up minor NKL11 on to address from DLBASE
         */
        addr = KLADDR + 8*dev.d_minor;
        if(dev.d_minor)
                addr =+ KLBASE-KLADDR-8;
        if(dev.d_minor >= NKL11)
                addr =+ DLBASE-KLBASE-8*NKL11+8;
        tp->t_addr = addr;
        if ((tp->t_state&ISOPEN) == 0) {
                tp->t_state = ISOPEN|CARR_ON;
                tp->t_flags = XTABS|LCASE|ECHO|CRMOD|ANYP;
                tp->t_erase = CERASE;
                tp->t_kill = CKILL;
```

number of KL11's

number of DL11's

```
        }
        addr->klrcsr =| IENABLE|DSRDY|RDRENB;
        addr->kltcsr =| IENABLE;
        ttyopen(dev, tp);
}

klclose(dev)
{
        register struct tty *tp;

        tp = &kl11[dev.d_minor];
        wflushtty(tp);
        tp->t_state = 0;
}

klread(dev)
{
        ttread(&kl11[dev.d_minor]);
}

klwrite(dev)
{
        ttwrite(&kl11[dev.d_minor]);
}

klxint(dev)     transmitter interrupt
{
        register struct tty *tp;

        tp = &kl11[dev.d_minor];
        ttstart(tp);
        if (tp->t_outq.c_cc == 0 || tp->t_outq.c_cc == TTLOWAT)
                wakeup(&tp->t_outq);
}

klrint(dev)     reciever interrupt
{
        register int c, *addr;
        register struct tty *tp;

        tp = &kl11[dev.d_minor];
        addr = tp->t_addr;
        c = addr->klrbuf;
        addr->klrcsr =| RDRENB;
        ttyinput(c, tp);
}

klsgtty(dev, v)
int *v;
{
        register struct tty *tp;

        tp = &kl11[dev.d_minor];
        ttystty(tp, v);
}
```

```
#
/*
 */

/*
 * RF disk driver
 */

#include "../param.h"
#include "../buf.h"
#include "../conf.h"
#include "../user.h"

struct {
        int     rfcs;
        int     rfwc;
        int     rfba;
        int     rfda;
        int     rfdae;
};

struct  devtab  rftab;
struct  buf     rrfbuf;

#define NRFBLK  1024
#define RFADDR  0177460

#define GO      01
#define RCOM    02
#define WCOM    04
#define CTLCLR  0400
#define IENABLE 0100

rfstrategy(abp)
struct buf *abp;
{
        register struct buf *bp;

        bp = abp;
        if(bp->b_flags&B_PHYS)
                mapalloc(bp);
        if (bp->b_blkno >= NRFBLK*(bp->b_dev.d_minor+1)) {
                bp->b_flags =| B_ERROR;
                iodone(bp);
                return;
        }
        bp->av_forw = 0;
        spl5();
        if (rftab.d_actf==0)
                rftab.d_actf = bp;
        else
                rftab.d_actl->av_forw = bp;
        rftab.d_actl = bp;
        if (rftab.d_active==0)
                rfstart();
        spl0();
```

block interface

allocate map if physical I/O

test for in

sort into list

start if not already active

```
}

rfstart()
{
        register struct buf *bp;

        if ((bp = rftab.d_actf) == 0)
                return;
        rftab.d_active++;
        RFADDR->rfdae = bp->b_blkno.hibyte;
        devstart(bp, &RFADDR->rfda, bp->b_blkno<<8, 0);
}

rfintr()
{
        register struct buf *bp;

        if (rftab.d_active == 0)
                return;
        bp = rftab.d_actf;
        rftab.d_active = 0;
        if (RFADDR->rfcs < 0) {          /* error bit */
                deverror(bp, RFADDR->rfcs, RFADDR->rfdae);
                RFADDR->rfcs = CTLCLR;
                if (++rftab.d_errcnt <= 10) {
                        rfstart();
                        return;
                }
                bp->b_flags |= B_ERROR;
        }
        rftab.d_errcnt = 0;
        rftab.d_actf = bp->av_forw;
        iodone(bp);
        rfstart();
}

rfread(dev)
{

        physio(rfstrategy, &rrfbuf, dev, B_READ);
}

rfwrite(dev)
{

        physio(rfstrategy, &rrfbuf, dev, B_WRITE);
}
```

chan

for

```
#
/*
 */

/*
 * general TTY subroutines
 */
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../tty.h"
#include "../proc.h"
#include "../inode.h"
#include "../file.h"
#include "../reg.h"
#include "../conf.h"

/*
 * Input mapping table-- if an entry is non-zero, when the
 * corresponding character is typed preceded by "\" the escape
 * sequence is replaced by the table value.  Mostly used for
 * upper-case only terminals.
 */
char    maptab[]
{
        000,000,000,000,004,000,000,000,
        000,000,000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        000,'|',000,'#',000,000,000,'`',
        '{','}',000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        '@',000,000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        000,000,000,000,000,000,'~',000,
        000,'A','B','C','D','E','F','G',
        'H','I','J','K','L','M','N','O',
        'P','Q','R','S','T','U','V','W',
        'X','Y','Z',000,000,000,000,000,
};

/*
 * The actual structure of a clist block manipulated by
 * getc and putc (mch.s)
 */
struct cblock {
        struct cblock *c_next;
        char info[6];
};

/* The character lists-- space for 6*NCLIST characters */
struct cblock cfree[NCLIST];
/* List head for unused character blocks. */
struct cblock *cfreelist;
```

```
/*
 * structure of device registers for KL, DL, and DC
 * interfaces-- more particularly, those for which the
 * SSTART bit is off and can be treated by general routines
 * (that is, not DH).
 */
struct {
        int ttrcsr;
        int ttrbuf;
        int tttcsr;
        int tttbuf;
};

/*
 * routine called on first teletype open.
 * establishes a process group for distribution
 * of quits and interrupts from the tty.
 */
ttyopen(dev, atp)
struct tty *atp;
{
        register struct proc *pp;
        register struct tty *tp;

        pp = u.u_procp;
        tp = atp;
        if(pp->p_pgrp == 0) {
                pp->p_pgrp = pp->p_pid;
                u.u_ttyp = tp;
                u.u_ttyd = dev;
                tp->t_pgrp = pp->p_pid;
        }
        tp->t_state =& ~WOPEN;
        tp->t_state =| ISOPEN;
}

/*
 * The routine implementing the gtty system call.
 * Just call lower level routine and pass back values.
 */
gtty()
{
        int v[3];
        register *up, *vp;

        vp = v;
        sgtty(vp);
        if (u.u_error)
                return;
        up = u.u_arg[0];
        suword(up, *vp++);
        suword(++up, *vp++);
        suword(++up, *vp++);
}
```

```
/*
 * The routine implementing the stty system call.
 * Read in values and call lower level.
 */
stty()
{
        register int *up;

        up = u.u_arg[0];
        u.u_arg[0] = fuword(up);
        u.u_arg[1] = fuword(++up);
        u.u_arg[2] = fuword(++up);
        sgtty(0);
}

/*
 * Stuff common to stty and gtty.
 * Check legality and switch out to individual
 * device routine.
 * v  is 0 for stty; the parameters are taken from u.u_arg[].
 * c  is non-zero for gtty and is the place in which the device
 * routines place their information.
 */
sgtty(v)
int *v;
{
        register struct file *fp;
        register struct inode *ip;

        if ((fp = getf(u.u_ar0[R0])) == NULL)
                return;
        ip = fp->f_inode;
        if ((ip->i_mode&IFMT) != IFCHR) {
                u.u_error = ENOTTY;
                return;
        }
        (*cdevsw[ip->i_addr[0].d_major].d_sgtty)(ip->i_addr[0], v);
}

/*
 * Wait for output to drain, then flush input waiting.
 */
wflushtty(atp)
struct tty *atp;
{
        register struct tty *tp;

        tp = atp;
        spl5();
        while (tp->t_outq.c_cc) {
                tp->t_state =| ASLEEP;
                sleep(&tp->t_outq, TTOPRI);
        }
        flushtty(tp);
        spl0();
}
```

```
/*
 * Initialize clist by freeing all character blocks, then count
 * number of character devices. (Once-only routine)
 */
cinit()
{
        register int ccp;
        register struct cblock *cp;
        register struct cdevsw *cdp;

        ccp = cfree;
        for (cp=(ccp+07)&~07; cp <= &cfree[NCLIST-1]; cp++) {
                cp->c_next = cfreelist;
                cfreelist = cp;
        }
        ccp = 0;
        for(cdp = cdevsw; cdp->d_open; cdp++)
                ccp++;
        nchrdev = ccp;
}

/*
 * flush all TTY queues
 */
flushtty(atp)
struct tty *atp;
{
        register struct tty *tp;
        register int sps;

        tp = atp;
        while (getc(&tp->t_canq) >= 0);
        while (getc(&tp->t_outq) >= 0);
        wakeup(&tp->t_rawq);
        wakeup(&tp->t_outq);
        sps = PS->integ;
        spl5();
        while (getc(&tp->t_rawq) >= 0);
        tp->t_delct = 0;
        PS->integ = sps;
}

/*
 * transfer raw input list to canonical list,
 * doing erase-kill processing and handling escapes.
 * It waits until a full line has been typed in cooked mode,
 * or until any character has been typed in raw mode.
 */
canon(atp)
struct tty *atp;
{
        register char *bp;
        char *bp1;
        register struct tty *tp;
        register int c;
```

```
        tp = atp;
        spl5();
        while (tp->t_delct==0) {
                if ((tp->t_state&CARR_ON)==0)
                        return(0);
                sleep(&tp->t_rawq, TTIPRI);
        }
        spl0();
loop:
        bp = &canonb[2];
        while ((c=getc(&tp->t_rawq)) >= 0) {
                if (c==0377) {
                        tp->t_delct--;
                        break;
                }
                if ((tp->t_flags&RAW)==0) {
                        if (bp[-1]!='\\') {
                                if (c==tp->t_erase) {
                                        if (bp > &canonb[2])
                                                bp--;
                                        continue;
                                }
                                if (c==tp->t_kill)
                                        goto loop;
                                if (c==CEOT)
                                        continue;
                        } else
                        if (maptab[c] && (maptab[c]==c || (tp->t_flags&LCASE))) {
                                if (bp[-2] != '\\')
                                        c = maptab[c];
                                bp--;
                        }
                }
                *bp++ = c;
                if (bp>=canonb+CANBSIZ)
                        break;
        }
        bp1 = bp;
        bp = &canonb[2];
        c = &tp->t_canq;
        while (bp<bp1)
                putc(*bp++, c);
        return(1);
}

/*
 * Place a character on raw TTY input queue, putting in delimiters
 * and waking up top half as needed.
 * Also echo if required.
 * The arguments are the character and the appropriate
 * tty structure.
 */
ttyinput(ac, atp)
struct tty *atp;
{
```

interrupt code

```
        register int t_flags, c;
        register struct tty *tp;

        tp = atp;
        c = ac&0177;
        t_flags = tp->t_flags;
        if (c=='\r' && t_flags&CRMOD)
                c = '\n';
        if ((t_flags&RAW)==0) {
                if (c==CQUIT || c==CINTR) {
                        signal(tp->t_pgrp, c==CINTR? SIGINT:SIGQIT);
                        flushtty(tp);
                        return;
                }
                if (c==CSTOP) {
                        if (tp->t_state&XMTSTOP) {
                                tp->t_state =& ~XMTSTOP;
                                ttstart(tp);
                        }
                        else
                                tp->t_state =| XMTSTOP;
                        return;
                }
        }
        if (tp->t_rawq.c_cc>=TTYHOG) {
                flushtty(tp);
                return;
        }
        if (t_flags&LCASE && c>='A' && c<='Z')
                c =+ 'a'-'A';
        putc(c, &tp->t_rawq);
        if (t_flags&RAW || c=='\n' || c==004) {
                wakeup(&tp->t_rawq);
                if (putc(0377, &tp->t_rawq)==0)
                        tp->t_delct++;
        }
        if (t_flags&ECHO) {
                tp->t_state =& ~XMTSTOP;
                ttyoutput(c, tp);
                ttstart(tp);
        }
}

/*
 * put character on TTY output queue, adding delays,
 * expanding tabs, and handling the CR/NL bit.
 * It is called both from the top half for output, and from
 * interrupt level for echoing.
 * The arguments are the character and the tty structure.
 */
ttyoutput(ac, tp)
struct tty *tp;
{
        register int c;
        register struct tty *rtp;
        register char *colp;
```

max readahead allowed

map UPPER → lower

```
        int ctype;

        rtp = tp;
        /*
         * output all 8 bits if no parity is specified.
         */
        if (!(rtp->t_flags&ANYP)) {
                putc(ac, &rtp->t_outq);
                return;
        }
        c = ac&0177;
        /*
         * Ignore EOT in normal mode to avoid hanging up
         * certain terminals.
         */
        if (c==004 && (rtp->t_flags&RAW)==0)
                return;
        /*
         * Turn tabs to spaces as required
         */
        if (c=='\t' && rtp->t_flags&XTABS) {
                do
                        ttyoutput(' ', rtp);
                while (rtp->t_col&07);
                return;
        }
        /*
         * for upper-case-only terminals,
         * generate escapes.
         */
        if (rtp->t_flags&LCASE) {
                colp = "({)}!|^~'`";
                while(*colp++)
                        if(c == *colp++) {
                                ttyoutput('\\', rtp);
                                c = colp[-2];
                                break;
                        }
                if ('a'<=c && c<='z')
                        c =+ 'A' - 'a';
        }
        /*
         * turn <nl> to <cr><lf> if desired.
         */
        if (c=='\n' && rtp->t_flags&CRMOD)
                ttyoutput('\r', rtp);
        if (putc(c, &rtp->t_outq))
                return;
        /*
         * Calculate delays.
         * The numbers here represent clock ticks
         * and are not necessarily optimal for all terminals.
         * The delays are indicated by characters above 0200,
         * thus (unfortunately) restricting the transmission
         * path to 7 bits.
         */
```

```
        colp = &rtp->t_col;
        ctype = partab[c];
        c = 0;
        switch (ctype&077) {

        /* ordinary */
        case 0:
                (*colp)++;

        /* non-printing */
        case 1:
                break;

        /* backspace */
        case 2:
                if (*colp)
                        (*colp)--;
                break;

        /* newline */
        case 3:
                ctype = (rtp->t_flags >> 8) & 03;
                if(ctype == 1) { /* tty 37 */
                        if (*colp)
                                c = max((*colp>>4) + 3, 6);
                } else
                if(ctype == 2) { /* vt05 */
                        c = 6;
                }
                *colp = 0;
                break;

        /* tab */
        case 4:
                ctype = (rtp->t_flags >> 10) & 03;
                if(ctype == 1) { /* tty 37 */
                        c = 1 - (*colp | ~07);
                        if(c < 5)
                                c = 0;
                }
                *colp =| 07;
                (*colp)++;
                break;

        /* vertical motion */
        case 5:
                if(rtp->t_flags & VTDELAY) /* tty 37 */
                        c = 0177;
                break;

        /* carriage return */
        case 6:
                ctype = (rtp->t_flags >> 12) & 03;
                if(ctype == 1) { /* tn 300 */
                        c = 5;
                } else
```

```
                if(ctype == 2) { /* ti 700 */
                        c = 10;
                }
                *colp = 0;
        }
        if(c)
                putc(c|0200, &rtp->t_outq);
}

/*
 * Restart typewriter output following a delay
 * timeout.
 * The name of the routine is passed to the timeout
 * subroutine and it is called during a clock interrupt.
 */
ttrstrt(atp)
{
        register struct tty *tp;

        tp = atp;
        tp->t_state =& ~TIMEOUT;
        ttstart(tp);
}

/*
 * Start output on the typewriter. It is used from the top half
 * after some characters have been put on the output queue,
 * from the interrupt routine to transmit the next
 * character, and after a timeout has finished.
 * If the SSTART bit is off for the tty the work is done here,
 * using the protocol of the single-line interfaces (KL, DL, DC);
 * otherwise the address word of the tty structure is
 * taken to be the name of the device-dependent startup routine.
 */
ttstart(atp)
struct tty *atp;
{
        register int *addr, c;
        register struct tty *tp;
        struct { int (*func)(); };

        tp = atp;
        addr = tp->t_addr;
        if (tp->t_state&SSTART) {
                (*addr.func)(tp);
                return;
        }
        if ((addr->tttcsr&DONE)==0 || tp->t_state&(TIMEOUT|XMTSTOP))
                return;
        if ((c=getc(&tp->t_outq)) >= 0) {
                if (!(tp->t_flags&ANYP))
                        addr->tttbuf = c;
                else
                        if (c<=0177)
                                addr->tttbuf = c | (partab[c]&0200);
                        else {
```

```
                                timeout(ttrstrt, tp, c&0177);
                                tp->t_state =| TIMEOUT;
                        }
        }
}

/*
 * Called from device's read routine after it has
 * calculated the tty-structure given as argument.
 * The pc is backed up for the duration of this call.
 * In case of a caught interrupt, an RTI will re-execute.
 */
ttread(atp)
struct tty *atp;
{
        register struct tty *tp;

        tp = atp;
        if ((tp->t_state&CARR_ON)==0)
                return;
        if (tp->t_canq.c_cc || canon(tp))
                while (tp->t_canq.c_cc && passc(getc(&tp->t_canq))>=0);
}

/*
 * Called from the device's write routine after it has
 * calculated the tty-structure given as argument.
 */
ttwrite(atp)
struct tty *atp;
{
        register struct tty *tp;
        register int c;

        tp = atp;
        if ((tp->t_state&CARR_ON)==0)
                return;
        while ((c=cpass())>=0) {
                spl5();
                while (tp->t_outq.c_cc > TTHIWAT) {
                        ttstart(tp);
                        tp->t_state =| ASLEEP;
                        sleep(&tp->t_outq, TTOPRI);
                }
                spl0();
                ttyoutput(c, tp);
        }
        ttstart(tp);
}

/*
 * Common code for gtty and stty functions on typewriters.
 * If v is non-zero then gtty is being done and information is
 * passed back therein;
 * if it is zero stty is being done and the input information is in the
 * u_arg array.
```

```
 */
ttystty(atp, av)
int *atp, *av;
{
        register  *tp, *v;

        tp = atp;
        if(v = av) {
                *v++ = tp->t_speeds;
                v->lobyte = tp->t_erase;
                v->hibyte = tp->t_kill;
                v[1] = tp->t_flags;
                return(1);
        }
        wflushtty(tp);
        v = u.u_arg;
        tp->t_speeds = *v++;
        tp->t_erase = v->lobyte;
        tp->t_kill = v->hibyte;
        tp->t_flags = v[1];
        return(0);
}
```

```
/ machine language assist
/ for 11/40

/ non-UNIX instructions
mfpi    = 6500^tst
mtpi    = 6600^tst
wait    = 1
rtt     = 6
reset   = 5

.globl  trap, call
.globl  _trap
trap:
        mov     PS,-4(sp)
        tst     nofault
        bne     1f
        mov     SSR0,ssr
        mov     SSR2,ssr+4
        mov     $1,SSR0
        jsr     r0,call1; _trap
        / no return
1:
        mov     $1,SSR0
        mov     nofault,(sp)
        rtt

.globl  _runrun, _swtch
call1:
        tst     -(sp)
        bic     $340,PS
        br      1f

call:
        mov     PS,-(sp)
1:
        mov     r1,-(sp)
        mfpi    sp
        mov     4(sp),-(sp)
        bic     $!37,(sp)
        bit     $30000,PS
        beq     1f
        jsr     pc,*(r0)+
2:
        bis     $340,PS
        tstb    _runrun
        beq     2f
        bic     $340,PS
        jsr     pc,_swtch
        br      2b
2:
        tst     (sp)+
        mtpi    sp
        br      2f
1:
        bis     $30000,PS
        jsr     pc,*(r0)+
```

```
        cmp     (sp)+,(sp)+
2:
        mov     (sp)+,r1
        tst     (sp)+
        mov     (sp)+,r0
        rtt

.globl  _savfp, _display
_savfp:
_display:
        rts     pc

.globl  _incupc
_incupc:
        mov     r2,-(sp)
        mov     6(sp),r2        / base of prof with base,leng,off,scale
        mov     4(sp),r0        / pc
        sub     4(r2),r0        / offset
        clc
        ror     r0
        mul     6(r2),r0        / scale
        ashc    $-14.,r0
        inc     r1
        bic     $1,r1
        cmp     r1,2(r2)        / length
        bhis    1f
        add     (r2),r1         / base
        mov     nofault,-(sp)
        mov     $2f,nofault
        mfpi    (r1)
        inc     (sp)
        mtpi    (r1)
        br      3f
2:
        clr     6(r2)
3:
        mov     (sp)+,nofault
1:
        mov     (sp)+,r2
        rts     pc

/ Character list get/put

.globl  _getc, _putc
.globl  _cfreelist

_getc:
        mov     2(sp),r1
        mov     PS,-(sp)
        mov     r2,-(sp)
        bis     $340,PS
        bic     $100,PS         / spl 5
        mov     2(r1),r2        / first ptr
        beq     9f              / empty
        movb    (r2)+,r0        / character
        bic     $!377,r0
```

```
        mov     r2,2(r1)
        dec     (r1)+           / count
        bne     1f
        clr     (r1)+
        clr     (r1)+           / last block
        br      2f
1:
        bit     $7,r2
        bne     3f
        mov     -10(r2),(r1)    / next block
        add     $2,(r1)
2:
        dec     r2
        bic     $7,r2
        mov     _cfreelist,(r2)
        mov     r2,_cfreelist
3:
        mov     (sp)+,r2
        mov     (sp)+,PS
        rts     pc
9:
        clr     4(r1)
        mov     $-1,r0
        mov     (sp)+,r2
        mov     (sp)+,PS
        rts     pc

_putc:
        mov     2(sp),r0
        mov     4(sp),r1
        mov     PS,-(sp)
        mov     r2,-(sp)
        mov     r3,-(sp)
        bis     $340,PS
        bic     $100,PS         / spl 5
        mov     4(r1),r2        / last ptr
        bne     1f
        mov     _cfreelist,r2
        beq     9f
        mov     (r2),_cfreelist
        clr     (r2)+
        mov     r2,2(r1)        / first ptr
        br      2f
1:
        bit     $7,r2
        bne     2f
        mov     _cfreelist,r3
        beq     9f
        mov     (r3),_cfreelist
        mov     r3,-10(r2)
        mov     r3,r2
        clr     (r2)+
2:
        movb    r0,(r2)+
        mov     r2,4(r1)
        inc     (r1)            / count
```

```
        clr     r0
        mov     (sp)+,r3
        mov     (sp)+,r2
        mov     (sp)+,PS
        rts     pc
9:
        mov     pc,r0
        mov     (sp)+,r3
        mov     (sp)+,r2
        mov     (sp)+,PS
        rts     pc

.globl  _backup
.globl  _regloc
_backup:
        mov     2(sp),ssr+2
        mov     r2,-(sp)
        jsr     pc,backup
        mov     r2,ssr+2
        mov     (sp)+,r2
        movb    jflg,r0
        bne     2f
        mov     2(sp),r0
        movb    ssr+2,r1
        jsr     pc,1f
        movb    ssr+3,r1
        jsr     pc,1f
        movb    _regloc+7,r1
        asl     r1
        add     r0,r1
        mov     ssr+4,(r1)
        clr     r0
2:
        rts     pc
1:
        mov     r1,-(sp)
        asr     (sp)
        asr     (sp)
        asr     (sp)
        bic     $!7,r1
        movb    _regloc(r1),r1
        asl     r1
        add     r0,r1
        sub     (sp)+,(r1)
        rts     pc

/ hard part
/ simulate the ssr2 register missing on 11/40

backup:
        clr     r2              / backup register ssr1
        mov     $1,bflg         / clrs jflg
        mov     ssr+4,r0
        jsr     pc,fetch
        mov     r0,r1
        ash     $-11.,r0
```

```
        bic     $!36,r0
        jmp     *0f(r0)
0:              t00; t01; t02; t03; t04; t05; t06; t07
                t10; t11; t12; t13; t14; t15; t16; t17

t00:
        clrb    bflg

t10:
        mov     r1,r0
        swab    r0
        bic     $!16,r0
        jmp     *0f(r0)
0:              u0; u1; u2; u3; u4; u5; u6; u7

u6:     / single op, m[tf]pi, sxt, illegal
        bit     $400,r1
        beq     u5              / all but m[tf], sxt
        bit     $200,r1
        beq     1f              / mfpi
        bit     $100,r1
        bne     u5              / sxt

/ simulate mtpi with double (sp)+,dd
        bic     $4000,r1        / turn instr into (sp)+
        br      t01

/ simulate mfpi with double ss,-(sp)
1:
        ash     $6,r1
        bis     $46,r1          / -(sp)
        br      t01

u4:     / jsr
        mov     r1,r0
        jsr     pc,setreg       / assume no fault
        bis     $173000,r2      / -2 from sp
        rts     pc

t07:    / EIS
        clrb    bflg

u0:     / jmp, swab
u5:     / single op
        mov     r1,r0
        br      setreg

t01:    / mov
t02:    / cmp
t03:    / bit
t04:    / bic
t05:    / bis
t06:    / add
t16:    / sub
        clrb    bflg
```

```
t11:    / movb
t12:    / cmpb
t13:    / bitb
t14:    / bicb
t15:    / bisb
        mov     r1,r0
        ash     $-6,r0
        jsr     pc,setreg
        swab    r2
        mov     r1,r0
        jsr     pc,setreg

/ if delta(dest) is zero,
/ no need to fetch source

        bit     $370,r2
        beq     1f

/ if mode(source) is R,
/ no fault is possible

        bit     $7000,r1
        beq     1f

/ if reg(source) is reg(dest),
/ too bad.

        mov     r2,-(sp)
        bic     $174370,(sp)
        cmpb    1(sp),(sp)+
        beq     t17

/ start source cycle
/ pick up value of reg

        mov     r1,r0
        ash     $-6,r0
        bic     $!7,r0
        movb    _regloc(r0),r0
        asl     r0
        add     ssr+2,r0
        mov     (r0),r0

/ if reg has been incremented,
/ must decrement it before fetch

        bit     $174000,r2
        ble     2f
        dec     r0
        bit     $10000,r2
        beq     2f
        dec     r0
2:

/ if mode is 6,7 fetch and add X(R) to R
```

```
        bit     $4000,r1
        beq     2f
        bit     $2000,r1
        beq     2f
        mov     r0,-(sp)
        mov     ssr+4,r0
        add     $2,r0
        jsr     pc,fetch
        add     (sp)+,r0
2:

/ fetch operand
/ if mode is 3,5,7 fetch *

        jsr     pc,fetch
        bit     $1000,r1
        beq     1f
        bit     $6000,r1
        bne     fetch
1:
        rts     pc

t17:    / illegal
u1:     / br
u2:     / br
u3:     / br
u7:     / illegal
        incb    jflg
        rts     pc

setreg:
        mov     r0,-(sp)
        bic     $!7,r0
        bis     r0,r2
        mov     (sp)+,r0
        ash     $-3,r0
        bic     $!7,r0
        movb    0f(r0),r0
        tstb    bflg
        beq     1f
        bit     $2,r2
        beq     2f
        bit     $4,r2
        beq     2f
1:
        cmp     r0,$20
        beq     2f
        cmp     r0,$-20
        beq     2f
        asl     r0
2:
        bisb    r0,r2
        rts     pc

0:      .byte   0,0,10,20,-10,-20,0,0
```

```
fetch:
        bic     $1,r0
        mov     nofault,-(sp)
        mov     $1f,nofault
        mfpi    (r0)
        mov     (sp)+,r0
        mov     (sp)+,nofault
        rts     pc

1:
        mov     (sp)+,nofault
        clrb    r2                      / clear out dest on fault
        mov     $-1,r0
        rts     pc

.bss
bflg:   .=.+1
jflg:   .=.+1
.text

.globl  _fubyte, _subyte
.globl  _fuibyte, _suibyte
.globl  _fuword, _suword
.globl  _fuiword, _suiword
_fuibyte:
_fubyte:
        mov     2(sp),r1
        bic     $1,r1
        jsr     pc,gword
        cmp     r1,2(sp)
        beq     1f
        swab    r0
1:
        bic     $!377,r0
        rts     pc

_suibyte:
_subyte:
        mov     2(sp),r1
        bic     $1,r1
        jsr     pc,gword
        mov     r0,-(sp)
        cmp     r1,4(sp)
        beq     1f
        movb    6(sp),1(sp)
        br      2f
1:
        movb    6(sp),(sp)
2:
        mov     (sp)+,r0
        jsr     pc,pword
        clr     r0
        rts     pc

_fuiword:
_fuword:
```

```
        mov     2(sp),r1
fuword:
        jsr     pc,gword
        rts     pc

gword:
        mov     PS,-(sp)
        bis     $340,PS
        mov     nofault,-(sp)
        mov     $err,nofault
        mfpi    (r1)
        mov     (sp)+,r0
        br      1f

_suiword:
_suword:
        mov     2(sp),r1
        mov     4(sp),r0
suword:
        jsr     pc,pword
        rts     pc

pword:
        mov     PS,-(sp)
        bis     $340,PS
        mov     nofault,-(sp)
        mov     $err,nofault
        mov     r0,-(sp)
        mtpi    (r1)
1:
        mov     (sp)+,nofault
        mov     (sp)+,PS
        rts     pc

err:
        mov     (sp)+,nofault
        mov     (sp)+,PS
        tst     (sp)+
        mov     $-1,r0
        rts     pc

.globl  _copyin, _copyout
_copyin:
        jsr     pc,copsu
1:
        mfpi    (r0)+
        mov     (sp)+,(r1)+
        sob     r2,1b
        br      2f

_copyout:
        jsr     pc,copsu
1:
        mov     (r0)+,-(sp)
        mtpi    (r1)+
        sob     r2,1b
```

```
2:
        mov     (sp)+,nofault
        mov     (sp)+,r2
        clr     r0
        rts     pc

copsu:
        mov     (sp)+,r0
        mov     r2,-(sp)
        mov     nofault,-(sp)
        mov     r0,-(sp)
        mov     10(sp),r0
        mov     12(sp),r1
        mov     14(sp),r2
        asr     r2
        mov     $1f,nofault
        rts     pc

1:
        mov     (sp)+,nofault
        mov     (sp)+,r2
        mov     $-1,r0
        rts     pc

.globl  _idle
_idle:
        mov     PS,-(sp)
        bic     $340,PS
        wait
        mov     (sp)+,PS
        rts     pc

.globl  _savu, _retu, _aretu
_savu:
        bis     $340,PS
        mov     (sp)+,r1
        mov     (sp),r0
        mov     sp,(r0)+
        mov     r5,(r0)+
        bic     $340,PS
        jmp     (r1)

_aretu:
        bis     $340,PS
        mov     (sp)+,r1
        mov     (sp),r0
        br      1f

_retu:
        bis     $340,PS
        mov     (sp)+,r1
        mov     (sp),KISA6
        mov     $_u,r0
1:
        mov     (r0)+,sp
        mov     (r0)+,r5
```

```
        bic     $340,PS
        jmp     (r1)

.globl  _spl0, _spl1, _spl4, _spl5, _spl6, _spl7
_spl0:
        bic     $340,PS
        rts     pc

_spl1:
        bis     $40,PS
        bic     $300,PS
        rts     pc

_spl4:
_spl5:
        bis     $340,PS
        bic     $100,PS
        rts     pc

_spl6:
        bis     $340,PS
        bic     $40,PS
        rts     pc

_spl7:
        bis     $340,PS
        rts     pc

.globl  _copyseg
_copyseg:
        mov     PS,-(sp)
        mov     UISA0,-(sp)
        mov     UISA1,-(sp)
        mov     $30340,PS
        mov     10(sp),UISA0
        mov     12(sp),UISA1
        mov     UISD0,-(sp)
        mov     UISD1,-(sp)
        mov     $6,UISD0
        mov     $6,UISD1
        mov     r2,-(sp)
        clr     r0
        mov     $8192.,r1
        mov     $32.,r2
1:
        mfpi    (r0)+
        mtpi    (r1)+
        sob     r2,1b
        mov     (sp)+,r2
        mov     (sp)+,UISD1
        mov     (sp)+,UISD0
        mov     (sp)+,UISA1
        mov     (sp)+,UISA0
        mov     (sp)+,PS
        rts     pc
```

```
.globl  _clearseg
_clearseg:
        mov     PS,-(sp)
        mov     UISA0,-(sp)
        mov     $30340,PS
        mov     6(sp),UISA0
        mov     UISD0,-(sp)
        mov     $6,UISD0
        clr     r0
        mov     $32.,r1
1:
        clr     -(sp)
        mtpi    (r0)+
        sob     r1,1b
        mov     (sp)+,UISD0
        mov     (sp)+,UISA0
        mov     (sp)+,PS
        rts     pc

.globl  _dpadd
_dpadd:
        mov     2(sp),r0
        add     4(sp),2(r0)
        adc     (r0)
        rts     pc

.globl  _dpcmp
_dpcmp:
        mov     2(sp),r0
        mov     4(sp),r1
        sub     6(sp),r0
        sub     8(sp),r1
        sbc     r0
        bge     1f
        cmp     r0,$-1
        bne     2f
        cmp     r1,$-512.
        bhi     3f
2:
        mov     $-512.,r0
        rts     pc
1:
        bne     2f
        cmp     r1,$512.
        blo     3f
2:
        mov     $512.,r1
3:
        mov     r1,r0
        rts     pc

.globl  dump
dump:
        bit     $1,SSR0
        bne     dump
```

```
/ save regs r0,r1,r2,r3,r4,r5,r6,KIA6
/ starting at abs location 4

        mov     r0,4
        mov     $6,r0
        mov     r1,(r0)+
        mov     r2,(r0)+
        mov     r3,(r0)+
        mov     r4,(r0)+
        mov     r5,(r0)+
        mov     sp,(r0)+
        mov     KISA6,(r0)+

/ dump all of core (ie to first mt error)
/ onto mag tape. (9 track or 7 track 'binary')

        mov     $MTC,r0
        mov     $60004,(r0)+
        clr     2(r0)
1:
        mov     $-512.,(r0)
        inc     -(r0)
2:
        tstb    (r0)
        bge     2b
        tst     (r0)+
        bge     1b
        reset

/ end of file and loop

        mov     $60007,-(r0)
        br      .

.globl  boots
boots:  012737
        100001
        164000
        000777

.globl  bootldr
bootldr:
        mov     $157744,r1
        mov     $bldrcd,r2
        mov     $14.,r3
1:      mov     (r2)+,(r1)+
        dec     r3
        bne     1b
        jmp     157744
bldrcd:
        016701
        000026
        012702
        000352
        005211
        105711
```

```
        100376
        116162
        000002
        157400
        005267
        177756
        000765
        177560

.globl  start, _end, _edata, _main
start:
        bit     $1,SSR0
        bne     start                   / loop if restart
        reset

/ initialize systems segments

        mov     $KISA0,r0
        mov     $KISD0,r1
        mov     $200,r4
        clr     r2
        mov     $6,r3
1:
        mov     r2,(r0)+
        mov     $77406,(r1)+            / 4k rw
        add     r4,r2
        sob     r3,1b

/ initialize user segment

        mov     $_end+63.,r2
        ash     $-6,r2
        bic     $!1777,r2
        mov     r2,(r0)+                / ksr6 = sysu
        mov     $usize-1\<8|6,(r1)+

/ initialize io segment
/ set up counts on supervisor segments

        mov     $IO,(r0)+
        mov     $77406,(r1)+            / rw 4k

/ get a sp and start segmentation

        mov     $_u+[usize*64.],sp
        inc     SSR0

/ clear bss

        mov     $_edata,r0
1:
        clr     (r0)+
        cmp     r0,$_end
        blo     1b

/ clear user block
```

```
        mov     $_u,r0
1:
        clr     (r0)+
        cmp     r0,$_u+[usize*64.]
        blo     1b

/ set up previous mode and call main
/ on return, enter user mode at 0R

        mov     $30000,PS
        jsr     pc,_main
        mov     $170000,-(sp)
        clr     -(sp)
        rtt

.globl  _ldiv
_ldiv:
        clr     r0
        mov     2(sp),r1
        div     4(sp),r0
        rts     pc

.globl  _lrem
_lrem:
        clr     r0
        mov     2(sp),r1
        div     4(sp),r0
        mov     r1,r0
        rts     pc

.globl  _lshift
_lshift:
        mov     2(sp),r1
        mov     (r1)+,r0
        mov     (r1),r1
        ashc    4(sp),r0
        mov     r1,r0
        rts     pc

.globl  csv
csv:
        mov     r5,r0
        mov     sp,r5
        mov     r4,-(sp)
        mov     r3,-(sp)
        mov     r2,-(sp)
        jsr     pc,(r0)

.globl cret
cret:
        mov     r5,r1
        mov     -(r1),r4
        mov     -(r1),r3
        mov     -(r1),r2
        mov     r5,sp
```

```
        mov     (sp)+,r5
        rts     pc

.globl  _u
_u      = 140000
usize   = 16.

PS      = 177776
SSR0    = 177572
SSR2    = 177576
KISA0   = 172340
KISA6   = 172354
KISD0   = 172300
MTC     = 172522
UISA0   = 177640
UISA1   = 177642
UISD0   = 177600
UISD1   = 177602
IO      = 7600

.data
.globl  _ka6, _cputype
_ka6:   KISA6
_cputype:40.

.bss
.globl  nofault, ssr, badtrap
nofault:.=.+2
ssr:    .=.+6
badtrap:.=.+2
```

```
/*
 */

int     (*bdevsw[])()
{
        &nulldev,       &nulldev,       &dvstrategy,    &dvtab,
        &tmopen,        &tmclose,       &tmstrategy,    &tmtab,
        0
};

int     (*cdevsw[])()
{
        &klopen,    &klclose,   &klread,    &klwrite,   &klsgtty,
        &nulldev,   &nulldev,   &mmread,    &mmwrite,   &nodev,
        &nulldev,   &nulldev,   &dvread,    &dvwrite,   &nodev,
        &tmopen,    &tmclose,   &tmread,    &tmwrite,   &nodev,
        &dhopen,    &dhclose,   &dhread,    &dhwrite,   &dhsgtty,
        &lpopen,    &lpclose,   &nodev,     &lpwrite,   &nodev,
        &dropen,    &drclose,   &drread,    &drwrite,   &nodev,
        &pcopen,    &pcclose,   &pcread,    &pcwrite,   &nodev,
        0
};

int     rootdev ((0<<8)|64);
int     swapdev ((0<<8)|72);
int     swplo   19680;
int     nswap   1920;
```

diva disk
mag tape
console TTY
/dev/mem
diva
mag
line printe
papertape
major
minor
size of swap area

```
/ low core

br4 = 200
br5 = 240
br6 = 300
br7 = 340

. = 0^.
        br      1f
        4

/ trap vectors
        trap; br7+0.            / bus error
        trap; br7+1.            / illegal instruction
        trap; br7+2.            / bpt-trace trap
        trap; br7+3.            / iot trap
        trap; br7+4.            / power fail
        trap; br7+5.            / emulator trap
        trap; br7+6.            / system entry

. = 40^.
.globl  start, dump
1:      jmp     start
        jmp     dump

. = 50^.
.globl  boots, bootldr
        jmp     boots
        jmp     bootldr

. = 60^.
        klin; br4
        klou; br4

. = 70^.
        pcin; br4
        pcout; br4

. = 100^.
        kwlp; br6
        kwlp; br6

. = 204^.
        lpin; br4

. = 224^.
        tmio; br5

. = 240^.
        trap; br7+7.            / programmed interrupt
        trap; br7+8.            / floating point
        trap; br7+9.            / segmentation violation

. = 254^.
        dvio; br5
```

```
/ floating vectors
. = 300^.
        drou; br5+0.
        drin; br5+0.

. = 320^.
        dhin; br5+0.
        dhou; br5+0.

. = 330^.
        dmintr; br4+0.

////////////////////////////////////////////////////////////
/                   interface code to C
////////////////////////////////////////////////////////////

.globl  call, trap

.globl  _klrint
klin:   jsr     r0,call; _klrint
.globl  _klxint
klou:   jsr     r0,call; _klxint

.globl  _clock
kwlp:   jsr     r0,call; _clock

.globl  _tmintr
tmio:   jsr     r0,call; _tmintr

.globl  _dhrint
dhin:   jsr     r0,call; _dhrint

.globl  _dhxint
dhou:   jsr     r0,call; _dhxint

.globl  _dvintr
dvio:   jsr     r0,call; _dvintr

.globl  _lpint
lpin:   jsr     r0,call; _lpint

.globl  _drrint
drin:   jsr     r0,call; _drrint
.globl  _drwint
drou:   jsr     r0,call; _drwint

.globl  _dmint
dmintr: jsr     r0,call; _dmint

.globl  _pcrint, _pcpint
pcin:   jsr     r0,call; _pcrint
pcout:  jsr     r0,call; _pcpint
```

```
/*
 * Each buffer in the pool is usually doubly linked into 2 lists:
 * the device with which it is currently associated (always)
 * and also on a list of blocks available for allocation
 * for other use (usually).
 * The latter list is kept in last-used order, and the two
 * lists are doubly linked to make it easy to remove
 * a buffer from one list when it was found by
 * looking through the other.
 * A buffer is on the available list, and is liable
 * to be reassigned to another disk block, if and only
 * if it is not marked BUSY.  When a buffer is busy, the
 * available-list pointers can be used for other purposes.
 * Most drivers use the forward ptr as a link in their I/O
 * active queue.
 * A buffer header contains all the information required
 * to perform I/O.
 * Most of the routines which manipulate these things
 * are in bio.c.
 */
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];

/*
 * Each block device has a devtab, which contains private state stuff
 * and 2 list heads: the b_forw/b_back list, which is doubly linked
 * and has all the buffers currently associated with that major
 * device; and the d_actf/d_actl list, which is private to the
 * device but in fact is always used for the head and tail
 * of the I/O queue for the device.
 * Various routines in bio.c look at b_forw/b_back
 * (notice they are the same as in the buf structure)
 * but the rest is private to each device driver.
 */
struct devtab
{
        char    d_active;               /* busy flag */
        char    d_errcnt;               /* error count (for recovery) */
        struct  buf *b_forw;            /* first buffer for this dev */
        struct  buf *b_back;            /* last buffer for this dev */
        struct  buf *d_actf;            /* head of I/O queue */
        struct  buf *d_actl;            /* tail of I/O queue */
};
```

```
/*
 * This is the head of the queue of available
 * buffers-- all unused except for the 2 list heads.
 */
struct  buf bfreelist;

/*
 * These flags are kept in b_flags.
 */
#define B_WRITE 0       /* non-read pseudo-flag */
#define B_READ  01      /* read when I/O occurs */
#define B_DONE  02      /* transaction finished */
#define B_ERROR 04      /* transaction aborted */
#define B_BUSY  010     /* not on av_forw/back list */
#define B_PHYS  020     /* Physical IO potentially using UNIBUS map */
#define B_MAP   040     /* This block has the UNIBUS map allocated */
#define B_WANTED 0100   /* issue wakeup when BUSY goes off */
#define B_RELOC 0200    /* no longer used */
#define B_ASYNC 0400    /* don't wait for I/O completion */
#define B_DELWRI 01000  /* don't write till block leaves available list */
```

```
/*
 * Used to dissect integer device code
 * into major (driver designation) and
 * minor (driver parameter) parts.
 */
struct
{
        char    d_minor;
        char    d_major;
};

/*
 * Declaration of block device
 * switch. Each entry (row) is
 * the only link between the
 * main unix code and the driver.
 * The initialization of the
 * device switches is in the
 * file conf.c.
 */
struct  bdevsw
{
        int     (*d_open)();
        int     (*d_close)();
        int     (*d_strategy)();
        int     *d_tab;
} bdevsw[];

/*
 * Nblkdev is the number of entries
 * (rows) in the block switch. It is
 * set in binit/bio.c by making
 * a pass over the switch.
 * Used in bounds checking on major
 * device numbers.
 */
int     nblkdev;

/*
 * Character device switch.
 */
struct  cdevsw
{
        int     (*d_open)();
        int     (*d_close)();
        int     (*d_read)();
        int     (*d_write)();
        int     (*d_sgtty)();
} cdevsw[];

/*
 * Number of character switch entries.
 * Set by cinit/tty.c
 */
int     nchrdev;
```

used (usually) by mount & umount. Also used if you open (e.g.) /dev/rk0

I/O entry point

Sorts address into device I/O queue

(devtab — buf.h)

ptr to head of I/O chain

includes /dev/rrk0, etc.

can also be used to rewind tape, etc.

```
/*
 * One file structure is allocated
 * for each open/creat/pipe call.
 * Main use is to hold the read/write
 * pointer associated with each open
 * file.
 */
struct  file
{
        char    f_flag;
        char    f_count;        /* reference count */
        int     f_inode;        /* pointer to inode structure */
        char    *f_offset[2];   /* read/write character pointer */
} file[NFILE];

/* flags */
#define FREAD   01
#define FWRITE  02
#define FPIPE   04
```

```
/*
 * Definition of the unix super block.
 * The root super block is allocated and
 * read in iinit/alloc.c. Subsequently
 * a super block is allocated and read
 * with each mount (smount/sys3.c) and
 * released with unmount (sumount/sys3.c).
 * A disk block is ripped off for storage.
 * See alloc.c for general alloc/free
 * routines for free list and I list.
 */
struct  filsys
{
        int     s_isize;        /* size in blocks of I list */
        int     s_fsize;        /* size in blocks of entire volume */
        int     s_nfree;        /* number of in core free blocks (0-100) */
        int     s_free[100];    /* in core free blocks */
        int     s_ninode;       /* number of in core I nodes (0-100) */
        int     s_inode[100];   /* in core free I nodes */
        char    s_flock;        /* lock during free list manipulation */
        char    s_ilock;        /* lock during I list manipulation */
        char    s_fmod;         /* super block modified flag */
        char    s_ronly;        /* mounted read-only flag */
        int     s_time[2];      /* current date of last update */
        int     pad[50];
};
```

11/6
5/6

```
/*
 * Inode structure as it appears on
 * the disk. Not used by the system,
 * but by things like check, df, dump.
 */
struct  inode
{
        int     i_mode;
        char    i_nlink;
        char    i_uid;
        char    i_gid;
        char    i_size0;
        char    *i_size1;
        int     i_addr[8];
        int     i_atime[2];
        int     i_mtime[2];
};

/* modes */
#define IALLOC  0100000
#define IFMT    060000
#define         IFDIR   040000
#define         IFCHR   020000
#define         IFBLK   060000
#define ILARG   010000
#define ISUID   04000
#define ISGID   02000
#define ISVTX   01000
#define IREAD   0400
#define IWRITE  0200
#define IEXEC   0100
```

```
/*
 * The I node is the focus of all
 * file activity in unix. There is a unique
 * inode allocated for each active file,
 * each current directory, each mounted-on
 * file, text file, and the root. An inode is 'named'
 * by its dev/inumber pair. (iget/iget.c)
 * Data, from mode on, is read in
 * from permanent inode on volume.
 */
struct  inode
{
        char    i_flag;
        char    i_count;        /* reference count */
        int     i_dev;          /* device where inode resides */
        int     i_number;       /* i number, 1-to-1 with device address */
        int     i_mode;
        char    i_nlink;        /* directory entries */
        char    i_uid;          /* owner */
        char    i_gid;          /* group of owner */
        char    i_size0;        /* most significant of size */
        char    *i_size1;       /* least sig */
        int     i_addr[8];      /* device addresses constituting file */
        int     i_lastr;        /* last logical block read (for read-ahead) */
} inode[NINODE];

/* flags */
#define ILOCK   01              /* inode is locked */
#define IUPD    02              /* inode has been modified */
#define IACC    04              /* inode access time to be updated */
#define IMOUNT  010             /* inode is mounted on */
#define IWANT   020             /* some process waiting on lock */
#define ITEXT   040             /* inode is pure text prototype */

/* modes */
#define IALLOC  0100000         /* file is used */
#define IFMT    060000          /* type of file */
#define         IFDIR   040000  /* directory */
#define         IFCHR   020000  /* character special */
#define         IFBLK   060000  /* block special, 0 is regular */
#define ILARG   010000          /* large addressing algorithm */
#define ISUID   04000           /* set user id on execution */
#define ISGID   02000           /* set group id on execution */
#define ISVTX   01000           /* save swapped text even after use */
#define IREAD   0400            /* read, write, execute permissions */
#define IWRITE  0200
#define IEXEC   0100
```

```
/*
 * tunable variables
 */

#define NBUF    19              /* size of buffer cache */
#define NINODE  150             /* number of in core inodes */
#define NFILE   100             /* number of in core file structures */
#define NMOUNT  9               /* number of mountable file systems */
#define NEXEC   2               /* number of simultaneous exec's */
#define MAXMEM  (64*32)         /* max core per process - first # is Kw */
#define SSIZE   20              /* initial stack size (*64 bytes) */
#define SINCR   20              /* increment of stack (*64 bytes) */
#define NOFILE  15              /* max open files per process */
#define CANBSIZ 256             /* max size of typewriter line */
#define CMAPSIZ 100             /* size of core allocation area */
#define SMAPSIZ 100             /* size of swap allocation area */
#define NCALL   20              /* max simultaneous time callouts */
#define NPROC   50              /* max number of processes */
#define NTEXT   40              /* max number of pure texts */
#define NCLIST  100             /* max total clist size */
#define HZ      60              /* Ticks/second of the clock */
#define MSGBUFS 128             /* Characters saved from error messages */

/*
 * priorities
 * probably should not be
 * altered too much
 */

#define PSWP    -100
#define PINOD   -90
#define PRIBIO  -50
#define PPIPE   1
#define PWAIT   40
#define PSLEP   90
#define PUSER   100

/*
 * signals
 * dont change
 */

#define NSIG    20
#define         SIGHUP  1       /* hangup */
#define         SIGINT  2       /* interrupt (rubout) */
#define         SIGQIT  3       /* quit (FS) */
#define         SIGINS  4       /* illegal instruction */
#define         SIGTRC  5       /* trace or breakpoint */
#define         SIGIOT  6       /* iot */
#define         SIGEMT  7       /* emt */
#define         SIGFPT  8       /* floating exception */
#define         SIGKIL  9       /* kill */
#define         SIGBUS  10      /* bus error */
#define         SIGSEG  11      /* segmentation violation */
#define         SIGSYS  12      /* sys */
#define         SIGPIPE 13      /* end of pipe */
```

```
#define         SIGCLK  14      /* alarm clock */

/*
 * fundamental constants
 * cannot be changed
 */

#define USIZE   16              /* size of user block (*64) */
#define NULL    0
#define NODEV   (-1)
#define ROOTINO 1               /* i number of all roots */
#define DIRSIZ  14              /* max characters per directory */

/*
 * structure to access an
 * integer in bytes
 */
struct
{
        char    lobyte;
        char    hibyte;
};

/*
 * structure to access an integer
 */
struct
{
        int     integ;
};

/*
 * Certain processor registers
 */
#define PS      0177776
#define KL      0177560
#define SW      0177570
```

```
/*
 * One structure allocated per active
 * process. It contains all data needed
 * about the process while the
 * process may be swapped out.
 * Other per process data (user.h)
 * is swapped with the process.
 */
struct  proc
{
        char    p_stat;
        char    p_flag;
        char    p_pri;          /* priority, negative is high */
        char    p_sig;          /* signal number sent to this process */
        char    p_uid;          /* user id, used to direct tty signals */
        char    p_time;         /* resident time for scheduling */
        char    p_cpu;          /* cpu usage for scheduling */
        char    p_nice;         /* nice for cpu usage */
        int     p_pgrp;         /* name of process group leader */
        int     p_pid;          /* unique process id */
        int     p_ppid;         /* process id of parent */
        int     p_addr;         /* address of swappable image */
        int     p_size;         /* size of swappable image (*64 bytes) */
        int     p_wchan;        /* event process is awaiting */
        int     *p_textp;       /* pointer to text structure */
        int     p_clktim;       /* time to alarm clock signal */
} proc[NPROC];

/* stat codes */
#define SSLEEP  1               /* awaiting an event */
#define SWAIT   2               /* (abandoned state) */
#define SRUN    3               /* running */
#define SIDL    4               /* intermediate state in process creation */
#define SZOMB   5               /* intermediate state in process termination */
#define SSTOP   6               /* process being traced */

/* flag codes */
#define SLOAD   01              /* in core */
#define SSYS    02              /* scheduling process */
#define SLOCK   04              /* process cannot be swapped */
#define SSWAP   010             /* process is being swapped out */
#define STRC    020             /* process is being traced */
#define SWTED   040             /* another tracing flag */
```

```
struct  passwd
{
        char    *pw_name;
        char    *pw_passwd;
        int     pw_uid;
        int     pw_gid;
        int     pw_quota;
        char    *pw_comment;
        char    *pw_gecos;
        char    *pw_dir;
        char    *pw_shell;
};
```

```
/*
 * Location of the users' stored
 * registers relative to R0.
 * Usage is u.u_ar0[XX].
 */
#define R0      (0)
#define R1      (-2)
#define R2      (-9)
#define R3      (-8)
#define R4      (-7)
#define R5      (-6)
#define R6      (-3)
#define R7      (1)
#define RPS     (2)

#define TBIT    020             /* PS trace bit */
```

```
/*
 * KT-11 addresses and bits.
 */

#define UISD    0177600         /* first user I-space descriptor register */
#define UISA    0177640         /* first user I-space address register */
#define UDSA    0177660         /* first user D-space address register */
#define RO      02              /* access abilities */
#define WO      04
#define RW      06
#define ED      010             /* extend direction */

/*
 * structure used to address
 * a sequence of integers.
 */
struct
{
        int     r[];
};
int     *ka6;           /* 11/40 KISA6; 11/45 KDSA6 */

/*
 * address to access 11/70 UNIBUS map
 */
#define UBMAP   0170200
```

I space

D space

```
/*
 * Random set of variables
 * used by more than one
 * routine.
 */
char    canonb[CANBSIZ];        /* buffer for erase and kill (#@) */
int     coremap[CMAPSIZ];       /* space for core allocation */
int     swapmap[SMAPSIZ];       /* space for swap allocation */
int     *rootdir;               /* pointer to inode of root directory */
int     cputype;                /* type of cpu =40, 45, or 70 */
int     execnt;                 /* number of processes in exec */
int     lbolt;                  /* time of day in 60th not in time */
int     time[2];                /* time in sec from 1970 */
int     tout[2];                /* time of day of next sleep */
/*
 * The callout structure is for
 * a routine arranging
 * to be called by the clock interrupt
 * (clock.c) with a specified argument,
 * in a specified amount of time.
 * Used, for example, to time tab
 * delays on teletypes.
 */
struct  callo
{
        int     c_time;         /* incremental time */
        int     c_arg;          /* argument to routine */
        int     (*c_func)();    /* routine */
} callout[NCALL];
/*
 * Mount structure.
 * One allocated on every mount.
 * Used to find the super block.
 */
struct  mount
{
        int     m_dev;          /* device mounted */
        int     *m_bufp;        /* pointer to superblock */
        int     *m_inodp;       /* pointer to mounted on inode */
} mount[NMOUNT];
int     mpid;                   /* generic for unique process id's */
char    runin;                  /* scheduling flag */
char    runout;                 /* scheduling flag */
char    runrun;                 /* scheduling flag */
char    curpri;                 /* more scheduling */
int     maxmem;                 /* actual max memory per process */
int     *lks;                   /* pointer to clock device */
int     rootdev;                /* dev of root see conf.c */
int     swapdev;                /* dev of swap see conf.c */
int     swplo;                  /* block number of swap space */
int     nswap;                  /* size of swap space */
int     updlock;                /* lock for sync */
int     rablock;                /* block to be read ahead */
char    regloc[];               /* locs. of saved user registers (trap.c) */
char    msgbuf[MSGBUFS];        /* saved "printf" characters */
```

```
/*
 * Text structure.
 * One allocated per pure
 * procedure on swap device.
 * Manipulated by text.c
 */
struct text
{
        int     x_daddr;        /* disk address of segment */
        int     x_caddr;        /* core address, if loaded */
        int     x_size;         /* size (*64) */
        int     *x_iptr;        /* inode of prototype */
        char    x_count;        /* reference count */
        char    x_ccount;       /* number of loaded references */
} text[NTEXT];
```

swap address

if sti
arou
cou

```
/*
 * A clist structure is the head
 * of a linked list queue of characters.
 * The characters are stored in 4-word
 * blocks containing a link and 6 characters.
 * The routines getc and putc (m45.s or m40.s)
 * manipulate these structures.
 */
struct clist
{
        int     c_cc;           /* character count */
        int     c_cf;           /* pointer to first block */
        int     c_cl;           /* pointer to last block */
};

/*
 * A tty structure is needed for
 * each UNIX character device that
 * is used for normal terminal IO.
 * The routines in tty.c handle the
 * common code associated with
 * these structures.
 * The definition and device dependent
 * code is in each driver. (kl.c dc.c dh.c)
 */
struct tty
{
        struct  clist t_rawq;   /* input chars right off device */
        struct  clist t_canq;   /* input chars after erase and kill */
        struct  clist t_outq;   /* output list to device */
        int     t_flags;        /* mode, settable by stty call */
        int     *t_addr;        /* device address (register or startup fcn) */
        char    t_delct;        /* number of delimiters in raw q */
        char    t_col;          /* printing column of device */
        char    t_erase;        /* erase character */
        char    t_kill;         /* kill character */
        char    t_state;        /* internal state, not visible externally */
        char    t_char;         /* character temporary */
        int     t_speeds;       /* output+input line speed */ <-- for gt
        int     t_pgrp;         /* process group name */
};

char partab[];                  /* ASCII table: parity, character class */

#define TTIPRI  10
#define TTOPRI  20

#define CERASE  '#'             /* default special characters */
#define CEOT    004
#define CKILL   '@'
#define CQUIT   034             /* FS, cntl shift L */
#define CINTR   0177            /* DEL */
#define CSTOP   033             /* Stop output character */

/* limits */
#define TTHIWAT 50
```

```
#define TTLOWAT 30
#define TTYHOG  256

/* modes */
#define HUPCL   01
#define XTABS   02
#define LCASE   04
#define ECHO    010
#define CRMOD   020
#define RAW     040
#define ODDP    0100
#define EVENP   0200
#define ANYP    0300
#define NLDELAY 001400
#define TBDELAY 006000
#define CRDELAY 030000
#define VTDELAY 040000

/* Hardware bits */
#define DONE    0200
#define IENABLE 0100

/* Internal state bits */
#define TIMEOUT 01              /* Delay timeout in progress */
#define WOPEN   02              /* Waiting for open to complete */
#define ISOPEN  04              /* Device is open */
#define SSTART  010             /* Has special start routine at addr */
#define CARR_ON 020             /* Software copy of carrier-present */
#define BUSY    040             /* Output in progress */
#define ASLEEP  0100            /* Wakeup when output done */
#define XMTSTOP 0200            /* Output stopped */
```

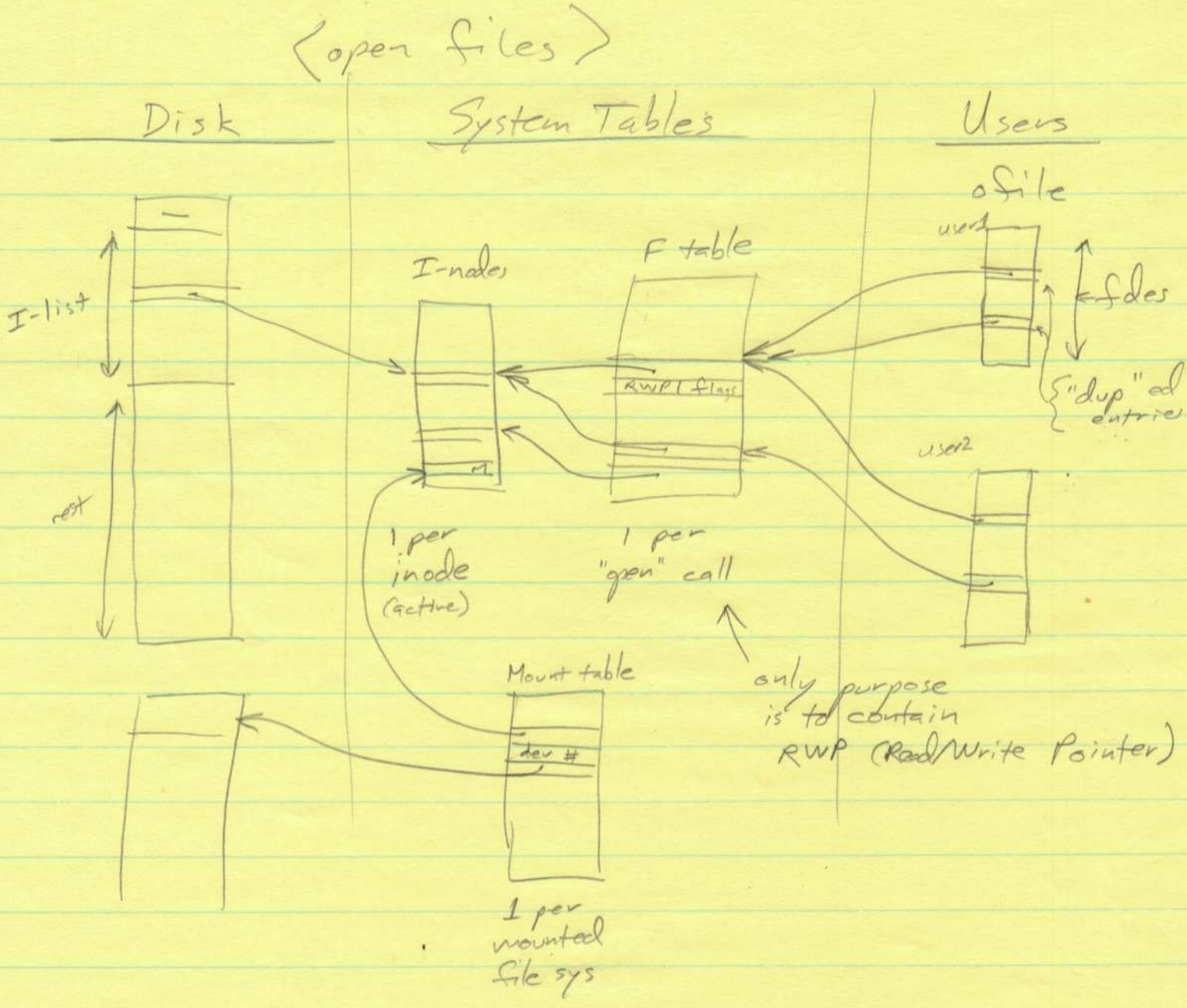
right out of manual

```
/*
 * The user structure.
 * One allocated per process.
 * Contains all per process data
 * that doesn't need to be referenced
 * while the process is swapped.
 * The user block is USIZE*64 bytes
 * long; resides at virtual kernel
 * loc 140000; contains the system
 * stack per user; is cross referenced
 * with the proc structure for the
 * same process.
 */
struct user
{
        int     u_rsav[2];              /* save r5,r6 when exchanging stacks */
        int     u_fsav[25];             /* save fp registers */
                                        /* rsav and fsav must be first in structur
        char    u_segflg;               /* flag for IO; user or kernel space */
        char    u_error;                /* return error code */
        char    u_uid;                  /* effective user id */
        char    u_gid;                  /* effective group id */
        char    u_ruid;                 /* real user id */
        char    u_rgid;                 /* real group id */
        int     u_procp;                /* pointer to proc structure */
        char    *u_base;                /* base address for IO */
        char    *u_count;               /* bytes remaining for IO */
        char    *u_offset[2];           /* offset in file for IO */
        int     *u_cdir;                /* pointer to inode of current directory *
        char    u_dbuf[DIRSIZ];         /* current pathname component */
        char    *u_dirp;                /* current pointer to inode */
        struct  {                       /* current directory entry */
                int     u_ino;
                char    u_name[DIRSIZ];
        } u_dent;
        int     *u_pdir;                /* inode of parent directory of dirp */
        int     u_uisa[16];             /* prototype of segmentation addresses */
        int     u_uisd[16];             /* prototype of segmentation descriptors *
        int     u_ofile[NOFILE];        /* pointers to file structures of open fil
        int     u_arg[5];               /* arguments to current system call */
        int     u_tsize;                /* text size (*64) */
        int     u_dsize;                /* data size (*64) */
        int     u_ssize;                /* stack size (*64) */
        int     u_sep;                  /* flag for I and D separation */
        int     u_qsav[2];              /* label variable for quits and interrupts
        int     u_ssav[2];              /* label variable for swapping */
        int     u_signal[NSIG];         /* disposition of signals */
        int     u_utime;                /* this process user time */
        int     u_stime;                /* this process system time */
        int     u_cutime[2];            /* sum of childs' utimes */
        int     u_cstime[2];            /* sum of childs' stimes */
        int     *u_ar0;                 /* address of users saved R0 */
        int     u_prof[4];              /* profile arguments */
        char    u_intflg;               /* catch intr from sys */
        int     u_ttyp;                 /* controlling tty pointer */
        int     u_ttyd;                 /* controlling tty dev */
```

```
er structure.
located per process.
ns all per process data
oesn't need to be referenced
the process is swapped.
er block is USIZE*64 bytes
resides at virtual kernel
0000; contains the system
per user; is cross referenced
he proc structure for the
rocess.

er

nt      u_rsav[2];              /* save r5,r6 when exchanging stacks */
nt      u_fsav[25];             /* save fp registers */
                                /* rsav and fsav must be first in structure */
har     u_segflg;               /* flag for IO; user or kernel space */
har     u_error;                /* return error code */
har     u_uid;                  /* effective user id */
har     u_gid;                  /* effective group id */
har     u_ruid;                 /* real user id */
har     u_rgid;                 /* real group id */
nt      u_procp;                /* pointer to proc structure */
har     *u_base;                /* base address for IO */
har     *u_count;               /* bytes remaining for IO */
har     *u_offset[2];           /* offset in file for IO */
nt      *u_cdir;                /* pointer to inode of current directory */
har     u_dbuf[DIRSIZ];         /* current pathname component */
har     *u_dirp;                /* current pointer to inode */
truct   {                       /* current directory entry */
        int     u_ino;
        char    u_name[DIRSIZ];
 u_dent;
nt      *u_pdir;                /* inode of parent directory of dirp */
nt      u_uisa[16];             /* prototype of segmentation addresses */
nt      u_uisd[16];             /* prototype of segmentation descriptors */
nt      u_ofile[NOFILE];        /* pointers to file structures of open files */
nt      u_arg[5];               /* arguments to current system call */
nt      u_tsize;                /* text size (*64) */
nt      u_dsize;                /* data size (*64) */
nt      u_ssize;                /* stack size (*64) */
nt      u_sep;                  /* flag for I and D separation */
nt      u_qsav[2];              /* label variable for quits and interrupts */
nt      u_ssav[2];              /* label variable for swapping */
nt      u_signal[NSIG];         /* disposition of signals */
nt      u_utime;                /* this process user time */
nt      u_stime;                /* this process system time */
nt      u_cutime[2];            /* sum of childs' utimes */
nt      u_cstime[2];            /* sum of childs' stimes */
nt      *u_ar0;                 /* address of users saved R0 */
nt      u_prof[4];              /* profile arguments */
har     u_intflg;               /* catch intr from sys */
nt      u_ttyp;                 /* controlling tty pointer */
nt      u_ttyd;                 /* controlling tty dev */
```

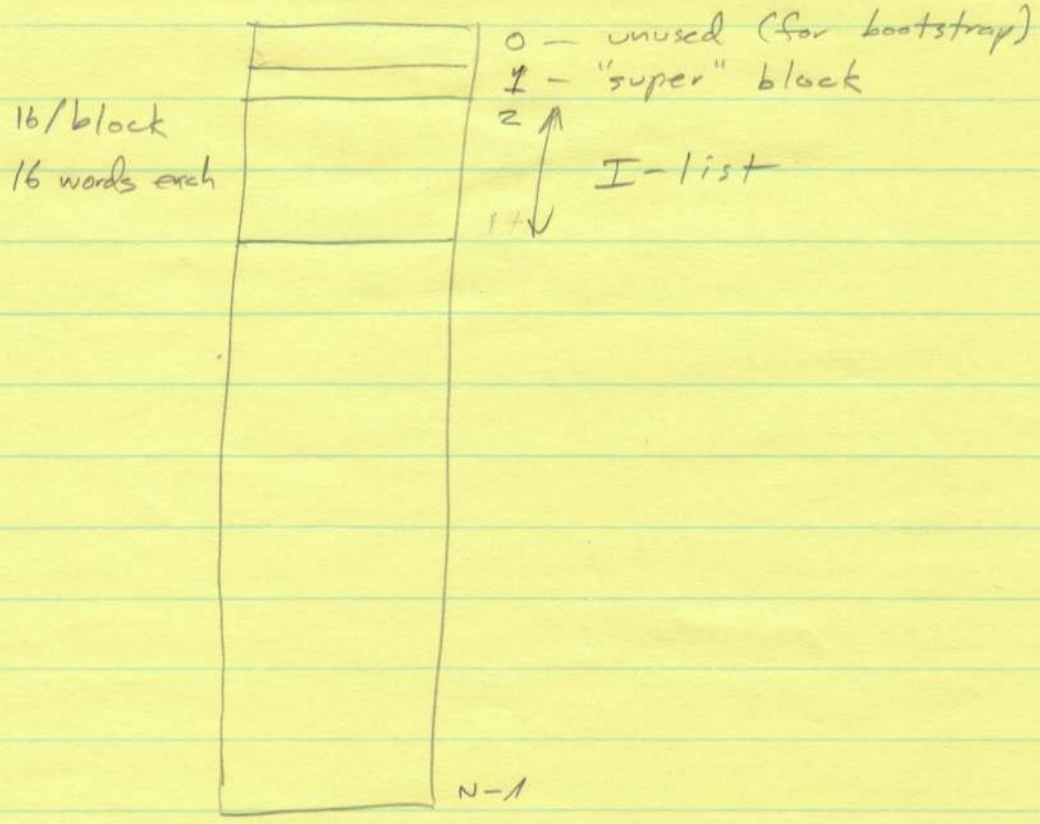
3 poin

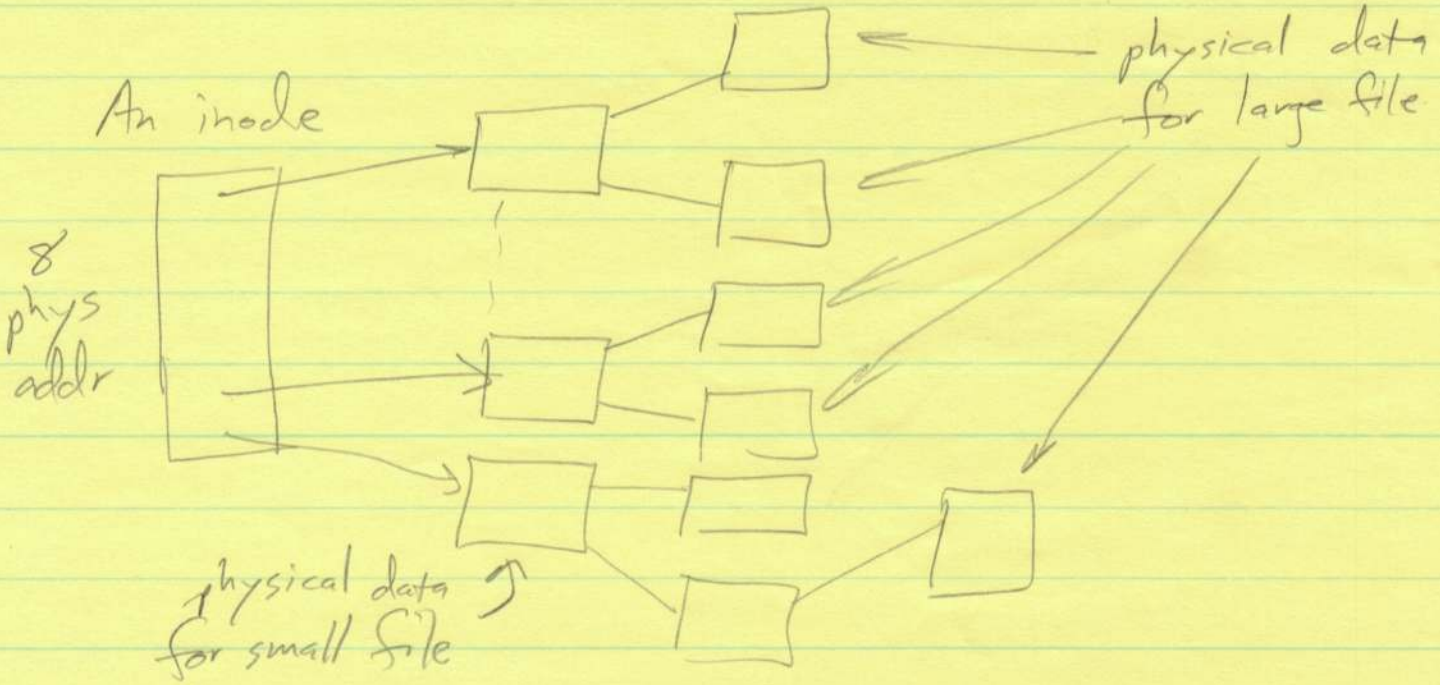
```
                                        /* kernel stack per user
                                         * extends from u + USIZE*64
                                         * backward not to reach here
                                         */
} u;

/* u.u_error codes */
#define EFAULT  106
#define EPERM   1
#define ENOENT  2
#define ESRCH   3
#define EINTR   4
#define EIO     5
#define ENXIO   6
#define E2BIG   7
#define ENOEXEC 8
#define EBADF   9
#define ECHILD  10
#define EAGAIN  11
#define ENOMEM  12
#define EACCES  13
#define ENOTBLK 15
#define EBUSY   16
#define EEXIST  17
#define EXDEV   18
#define ENODEV  19
#define ENOTDIR 20
#define EISDIR  21
#define EINVAL  22
#define ENFILE  23
#define EMFILE  24
#define ENOTTY  25
#define ETXTBSY 26
#define EFBIG   27
#define ENOSPC  28
#define ESPIPE  29
#define EROFS   30
#define EMLINK  31
#define EPIPE   32
```



If the mount flag in the Inode table is set, the mount table is searched

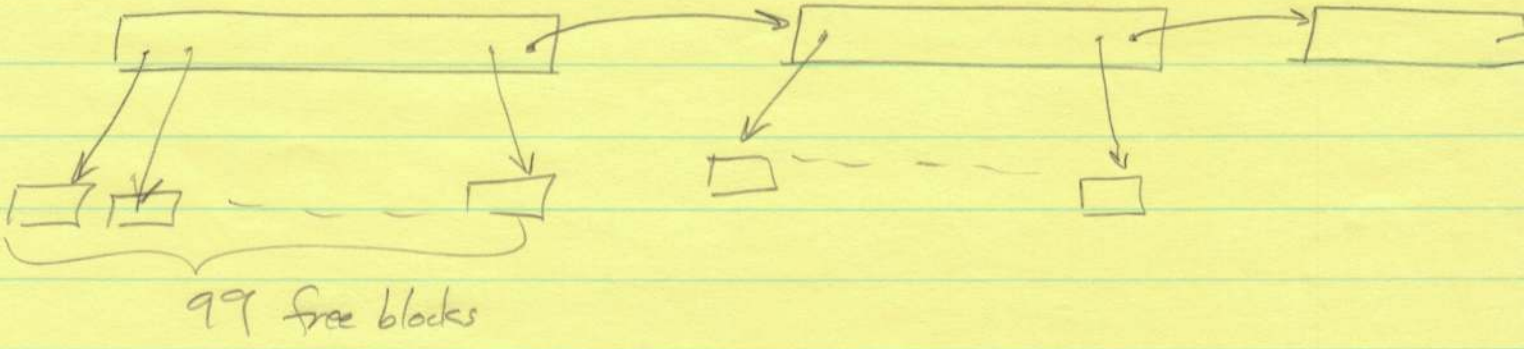
Disk | consists of a psuedo-disk





inode 1 on any file system is the root.

Free list is a linked list



File block cache is allocated LRU