```
/*      @(#)subr.c      2.7     */

#
#include "sys/param.h"
#include "sys/conf.h"
#include "sys/inode.h"
#include "sys/inodex.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/systm.h"
#include "sys/tty.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.
 */
daddr_t
bmap(ip, bn, rwflg)
struct inode *ip;
daddr_t bn;
{
        register struct buf *bp;
        register daddr_t nb;
        register unsigned i;
        int *nbp, d;

        d = ip->i_dev;
        if(bn & ~03777) {
                u.u_error = EFBIG;
                return((daddr_t)0);
        }
        rablock = 0;

        if((ip->i_mode&IFMT)!=IFLRG && (ip->i_mode&IFMT)!=IFLDR) {

                /*
                 * small file algorithm
                 */

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

                        /*
                         * convert small to large
                         */

                        if (rwflg==B_READ || (bp = alloc(d))==NULL)
                                return((daddr_t)-1);
                        copyio(paddr(bp), ip->i_un.i_addr, 16, U_WKD);
                        for(i=0; i<8; i++)
```

```
                        ip->i_un.i_addr[i] = 0;
                ip->i_un.i_addr[0] = bp->b_blkno;
                bdwrite(bp);
                switch(ip->i_mode&IFMT) {
                case IFREG:
                        ip->i_mode = (ip->i_mode&(~IFMT)) | IFLRG;
                        break;

                case IFDIR:
                        ip->i_mode = (ip->i_mode&(~IFMT)) | IFLDR;
                        break;

                default:
                        panic("bmap");
                }
                ip->i_flag =| IUPD;
                goto large;
            }
            nb = ip->i_un.i_addr[bn];
            if (nb == 0) {
                if (rwflg==B_READ || (bp = alloc(d))==NULL)
                        return((daddr_t)-1);
                bdwrite(bp);
                nb = bp->b_blkno;
                ip->i_un.i_addr[bn] = nb;
                ip->i_flag =| IUPD;
            }
            if (bn<7)
                rablock = ip->i_un.i_addr[bn+1];
            return(nb);
        }

        /*
         * large file algorithm
         */

    large:
        i = bn>>8;
        if((nb=ip->i_un.i_addr[i]) == 0) {
            if (rwflg==B_READ || (bp = alloc(d)) == NULL)
                return((daddr_t)-1);
            ip->i_un.i_addr[i] = bp->b_blkno;
            ip->i_flag =| IUPD;
        } else
            bp = bread(d, nb);

        /*
         * normal indirect fetch
         */

        i = bn & 0377;
        if ((nb = xget(paddr(bp) + 2*i)) == 0) {
            if (rwflg==B_READ || (nbp = alloc(d)) == NULL) {
                brelse(bp);
                return((daddr_t)-1);
            }
```

```
                nb = nbp->b_blkno;
                xput(paddr(bp) + 2*i, nb);
                bdwrite(nbp);
                bdwrite(bp);
        } else
                brelse(bp);
        if(i < 255)
                rablock = xget(paddr(bp) + 2*(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;
{
        register id;

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

        if(u.u_count == 0)
                return(-1);
        if((id = u.u_segflg) == 1)
                c = *u.u_base;
        else
                if((c = id==0?fubyte(u.u_base):fuibyte(u.u_base)) < 0) {
                        u.u_error = EFAULT;
                        return(-1);
                }
        u.u_count--;
```

```
        u.u_offset++;
        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()
{
        return(0);      /* make stty happy*/
}

/*
 * copy count bytes from from to to.
 */
bcopy(from, to, count)
register int *from, *to;
register unsigned count;
{

        if (((int)from | (int)to | count) & 01) {
                do
                        *((char *)to)++ = *((char *)from)++;
                while(--count);
        } else {
                count /= 2;
                do
                        *to++ = *from++;
                while(--count);
        }
}

/*
 * Integer (2-byte) get/put
 * using clists
 */
getw(p)
register struct clist *p;
{
        register int s;

        if (p->c_cc <= 1)
                return(-1);
        s = getc(p);
```

```
        return(s | (getc(p)<<8));
}

putw(c, p)
register struct clist *p;
{
        register int s;
        extern cfreelist;
#ifdef NXCLIST
        extern xcfreelist;
#endif

        s = spl6();
#ifdef NXCLIST
        if (cfreelist==NULL || xcfreelist==NULL)
#else
        if (cfreelist==NULL)
#endif
        {
                splx(s);
                return(-1);
        }
        putc(c, p);
        putc(c>>8, p);
        splx(s);
        return(0);
}
```

```
/*      @(#)sys1.c      2.18    */

#
#include "sys/param.h"
#include "sys/systm.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/reg.h"
#include "sys/inode.h"
#include "sys/inodex.h"
#include "sys/seg.h"
#include "sys/sysmes.h"
#include "sys/sysmesx.h"
#include "sys/vtmn.h"
#include "sys/maus.h"
#include "sys/version.h"
#include "sys/elog.h"


struct execa {
        char *fname;
        char **argp;
        char **envp;
};

#define NCABLK  ((NCARGS+BSIZE-1)/BSIZE)

#ifdef SPROF
#include "sys/sprof.h"
#include "sys/sprofx.h"

extern  sprofoff();
#endif



/*
 * exec system call.
 */
exec(ip)
struct inode *ip;
{
        ((struct execa *)u.u_arg)->envp = NULL;
        exece(ip);
}

exece()
{
        register unsigned nc;
        register char *cp;
```

```
        register struct buf *bp;
        register struct execa *uap;
        int na, ne, ucp, ap, c;
        unsigned bno;
        struct inode *ip;
#ifdef VTMON
        char *nptr;
        char nbuf[30];
#endif

        /*
         * it is illegal for locked processes to exec
         */
        if (u.u_lock != 0)
        {
                u.u_error = ELOCK;
                return;
        }
        if((ip = gethead()) == NULL)
                return;
        bp = 0;
        na = nc = ne = 0;
        uap = (struct execa *)u.u_arg;
        /* collect arglist */
        if ((bno = malloc(swapmap,NCABLK)) == 0)
                panic(ooswap);
        if (uap->argp) for (;;) {
                ap = NULL;
                if (uap->argp) {
                        ap = fuword((caddr_t)uap->argp);
                        uap->argp++;
                }
                if (ap==NULL && uap->envp) {
                        uap->argp = NULL;
                        if ((ap = fuword((caddr_t)uap->envp)) == NULL)
                                break;
                        uap->envp++;
                        ne++;
                }
                if (ap==NULL)
                        break;
                na++;
                if(ap == -1)
                        u.u_error = EFAULT;
                do {
                        if (nc >= NCARGS-1)
                                u.u_error = E2BIG;
                        if ((c = fubyte((caddr_t)ap++)) < 0)
                                u.u_error = EFAULT;
                        if (u.u_error)
                                goto bad;
                        if ((nc&BMASK) == 0) {
                                if (bp)
                                        bdwrite(bp);
                                bp = getblk(swapdev, bno+(nc>>BSHIFT));
                                cp = NULL;
```

```
                        }
                        nc++;
                        xputc(paddr(bp)+(unsigned)cp, c);
                        cp++;
                } while (c>0);
        }
        if (bp)
                bdwrite(bp);
        bp = 0;
        nc = (nc + NBPW-1) & ~(NBPW-1);
        getxfile(ip, nc+2*na);
        if (u.u_error) {
                psignal(u.u_procp, SIGKIL);
                goto bad;
        }
        if (u.u_exdata.ux_mag == 0405)
                goto bad;
        /* copy back arglist */
#ifdef VTMON
        nptr = nbuf;
#endif
        ucp = -nc - NBPW;
        ap = ucp - na*NBPW - 3*NBPW;
        u.u_ar0[R6] = ap;
        suword((caddr_t)ap, na-ne);
        nc = 0;
        for (;;) {
                ap += NBPW;
                if (na==ne) {
                        ap += NBPW;
                }
                if (--na < 0)
                        break;
                suword((caddr_t)ap, ucp);
                do {
                        if ((nc&BMASK) == 0) {
                                if (bp)
                                        brelse(bp);
                                bp = bread(swapdev, bno+(nc)>>BSHIFT));
                                bp->b_flags |= B_AGE;
                                bp->b_flags &= ~B_DELWRI;
                                cp = NULL;
                        }
                    subyte((caddr_t)ucp++,(c=xgetc(paddr(bp)+(unsigned)cp)));
                        cp++;
#ifdef VTMON
                        if (nptr < nbuf+sizeof(nbuf))
                                *nptr++ = c;
#endif
                        nc++;
                } while(c&0377);
        }
        VTNEWPROC(u.u_procp, PR_NAME, nbuf, nptr-nbuf);
        setregs();
        if (bp)
                brelse(bp);
```

```
        iput(ip);
        mfree(swapmap, NCABLK, bno);
        meas.m_exec++;
        return;
bad:
        if (bp)
                brelse(bp);
        iput(ip);
        for (nc = 0; nc < NCABLK; nc++) {
                bp = getblk(swapdev, bno+nc);
                bp->b_flags |= B_AGE;
                bp->b_flags &= ~B_DELWRI;
                brelse(bp);
        }
        mfree(swapmap, NCABLK, bno);
}

gethead()
{
        register struct inode *ip;
        register unsigned ds, ts;
        register sep;
        int savver, savmbit;
        extern int uchar();

        if ((ip = namei(uchar, 0)) == NULL)
                return(NULL);
        savmbit = u.u_mbitm;
        savver = u.u_exdata.ux_uver;
        if (access(ip, IEXEC) ||
           ((ip->i_mode & IFMT) != IFREG && (ip->i_mode & IFMT) != IFLRG) ||
           (ip->i_mode & (IEXEC|(IEXEC>>3)|(IEXEC>>6))) == 0) {
                u.u_error = EACCES;
                goto bad;
        }
        /*
         * read in first few bytes of file for segment sizes
         * ux_mag = 407/410/411/405
         *  407 is plain executable
         *  410 is RO text
         *  411 is separated ID
         *  405 is overlaid text
         */
        u.u_base = (caddr_t)&u.u_exdata;
        u.u_count = sizeof(u.u_exdata);
        u.u_offset = 0;
        u.u_segflg = 1;
        readi(ip);
        u.u_segflg = 0;
        if (u.u_error)
                goto bad;
        if (u.u_count!=0)
                u.u_exdata.ux_mag = 0;
        sep = 0;
        if (u.u_exdata.ux_mag == 0407) {
                ds = btoc((long)u.u_exdata.ux_tsize +
```

```
                        (long)u.u_exdata.ux_dsize +
                        (long)u.u_exdata.ux_bsize);
                ts = 0;
                u.u_exdata.ux_dsize += u.u_exdata.ux_tsize;
                u.u_exdata.ux_tsize = 0;
        } else {
                ts = btoc(u.u_exdata.ux_tsize);
                ds = btoc(u.u_exdata.ux_dsize+u.u_exdata.ux_bsize);
                if ((ip->i_flag&ITEXT)==0 && ip->i_count!=1)
                        u.u_error = ETXTBSY;
                if (u.u_exdata.ux_mag == 0411)
                        sep++;
                else if (u.u_exdata.ux_mag == 0405) {
                        if (u.u_sep==0 && ctos(ts) != ctos(u.u_tsize))
                                u.u_error = ENOMEM;
                        goto bad;
                }
                else if (u.u_exdata.ux_mag != 0410) {
                        u.u_error = ENOEXEC;
                        goto bad;
                }
        }
        switch(u.u_exdata.ux_uver) {    /*check version and define default */
                case UV_DEF:
                        u.u_exdata.ux_uver = UV_CBR2;
                        break;
                case UV_CBR2:
                case UV_CBR3:
                        break;
                default:
                        u.u_error = EINVAL;
                        goto bad;
        }
        u.u_mbitm = 0;
        checkur(ts,ds,SSIZE+btoc(NCARGS-1),sep);
bad:
        if(u.u_error) {
                u.u_mbitm = savmbit;
                u.u_exdata.ux_uver = savver;
                iput(ip);
                ip = NULL;
        }
        return(ip);
}

/*
 * Read in and set up memory for executed file.
 */
getxfile(ip, nargc)
register struct inode *ip;
{
        register unsigned ds;
        register i;

        xfree();
        u.u_tsize = btoc(u.u_exdata.ux_tsize);
```

```
        if (u.u_exdata.ux_mag == 0405) {
                xalloc(ip);
                u.u_ar0[PC] = u.u_exdata.ux_entloc & ~01;
        } else {
                u.u_prof[3] = 0;
                u.u_ssize = SSIZE + btoc(nargc);
                u.u_dsize = btoc(u.u_exdata.ux_dsize + u.u_exdata.ux_bsize);
                i = USIZE+u.u_dsize+u.u_ssize;
                expand(i);
                ds = USIZE+((u.u_exdata.ux_dsize>>6)&01777);
                while(--i >= ds)
                        clearseg(u.u_procp->p_addr+i);
                xalloc(ip);
                /* read in data segment */
                estabur((unsigned)0, u.u_dsize, (unsigned)0, 0, RO);
                u.u_base = 0;
                u.u_offset = sizeof(u.u_exdata)+u.u_exdata.ux_tsize;
                u.u_count = u.u_exdata.ux_dsize;
                readi(ip);
                if (u.u_count!=0)
                        u.u_error = EFAULT;
                /*
                 * 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;
                } else
                        psignal(u.u_procp, SIGTRC);
                u.u_sep = (u.u_exdata.ux_mag == 0411)?1:0;
        }
        u.u_dev = ip->i_dev;
        u.u_inode = ip->i_number;
        estabur(u.u_tsize, u.u_dsize, u.u_ssize,u.u_sep,RO);
}

/*
 * Clear registers on exec
 */
setregs()
{
        register int *rp;
        register char *cp;
        register i;

        for(rp = &u.u_signal[0]; rp < &u.u_signal[NSIG]; rp++)
                if((*rp & 1) == 0)
                        *rp = 0;
        for(cp = &regloc[0]; cp < &regloc[6];)
                u.u_ar0[*cp++] = 0;
        u.u_ar0[PC] = u.u_exdata.ux_entloc & ~01;
```

```
        for(rp = &u.u_fsav[0]; rp < &u.u_fsav[25];)
                *rp++ = 0;
        for(i=0; i<NOFILE; i++) {
                if ((u.u_pofile[i]&EXCLOSE) && u.u_ofile[i] != NULL) {
                        closef(u.u_ofile[i]);
                        u.u_ofile[i] = NULL;
                }
        }
        /*
         * Remember file name for accounting.
         */
        u.u_acflag &= ~1;
        for (i=0; i<DIRSIZ; i++)
                u.u_comm[i] = u.u_dbuf[i];
}

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

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

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

        p = u.u_procp;
        p->p_flag =& ~STRC;
        p->p_clktim = 0;
        for(a = &u.u_signal[0]; a < &u.u_signal[NSIG];)
                *a++ = 1;
        for(a = &u.u_ofile[0]; a < &u.u_ofile[NOFILE]; a++)
                if (*a) {
                        closef(*a);
                        *a = NULL;
                }
        if(u.u_msgqhdr != NULL)
                msgflush();
        semafree();
#ifdef SPROF
        if(u.u_procp->p_pid == sysprof.pid)
                sprofoff();
#endif
```

```
#ifdef PTLOCK
        if(punlock())
        {
                if(runout !=0)
                {
                        runout=0;
                        wakeup(&runout);
                }
        }
#endif
        plock(u.u_cdir);
        iput(u.u_cdir);
        plock(u.u_rdir);
        iput(u.u_rdir);
        xfree();
        acct();

        a = malloc(swapmap, 8);
        q = getblk(swapdev, a);
        copyio(q->b_paddr, (caddr_t)&u, BSIZE, U_WKD);
        bwrite(q);

        mfree(coremap, p->p_size, p->p_addr);
        p->p_stat = SZOMB;
        p->xp_xstat = u.u_arg[0];
        p->p_addr = a;
        p->p_flag =& ~SLOAD;
        for(q = &proc[1]; q < procend; q++) {
                if(q->p_ppid == p->p_pid) {
                        q->p_ppid = 1;
                        VTPROCENT(q,PR_PPD);
                        if(q->p_stat == SZOMB)
                                psignal(&proc[1], SIGCLD);
                        if (q->p_stat==SSTOP)
                                setrun(q);
                } else if(p->p_ppid == q->p_pid)
                        psignal(q, SIGCLD);
        }
        swtch();
        /* no deposit, no return */
}

/*
 * Wait system call.
 * Search for a terminated (zombie) child,
 * finally lay it to rest, and collect its status.
 * Look also for stopped (traced) children,
 * and pass back status from them.
 */
wait()
{
        register f;
        register struct proc *p;

loop:
        f = 0;
```

```
        for(p = &proc[0]; p < procend; p++)
        if(p->p_ppid == u.u_procp->p_pid) {
                f++;
                if(p->p_stat == SZOMB) {
                        freeproc(p, 1);
                        return;
                }
                if(p->p_stat == SSTOP) {
                        if((p->p_flag&SWTED) == 0) {
                                p->p_flag =| SWTED;
                                VTPROCENT(p,PR_FLAG);
                                u.u_ar0[R0] = p->p_pid;
                                u.u_ar0[R1] = (fsig(p)<<8) | 0177;
                                return;
                        }
                        continue;
                }
        }
        if(f) {
                sleep(u.u_procp, PWAIT);
                goto loop;
        }
        u.u_error = ECHILD;
}

/*
 * Dispose of zombie children by removing them
 * from the process table.
 */

freeproc(p, flg)
        register struct proc *p;
{
        register *bp, f;

        if(flg) {
                u.u_ar0[R0] = p->p_pid;
                u.u_ar0[R1] = p->xp_xstat;
        }
        bp = abread(swapdev, f=p->p_addr);
        mfree(swapmap, 8, 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->p_wchan = 0;
        VTPROCENT(p,PR_CLEAR);
        p = (caddr_t)bp->b_paddr;
        u.u_cstime =+ p->u_cstime + p->u_stime;
        u.u_cutime =+ p->u_cutime + p->u_utime;
        u.u_cgbcnt =+ p->u_cgbcnt + p->u_gbcnt;
        u.u_cdread =+ p->u_cdread + p->u_dread;
        u.u_cdwrit =+ p->u_cdwrit + p->u_dwrit;
        brelse(bp);
```

```
}

/*
 * fork system call.
 */
fork()
{
        register struct proc *p1, *p2;
        register a;

        /*
         * Make sure there's enough swap space for max
         * core image, thus reducing chances of running out
         */
        if ((a = malloc(swapmap, ctod(MAXMEM))) == 0) {
                u.u_error = ENOMEM;
                goto out;
        }
        mfree(swapmap, ctod(MAXMEM), a);
        p1 = u.u_procp;
        for(p2 = &proc[0]; p2 < &proc[NPROC]; p2++)
                if(p2->p_stat == NULL)
                        goto found;
        u.u_error = EAGAIN;
        logovfl(E_PROCO);
        meas.m_povfl++;
        goto out;

found:
        if(newproc()) {
                u.u_ar0[R0] = p1->p_pid;
                u.u_start = time;
                u.u_cstime = 0;
                u.u_stime = 0;
                u.u_cutime = 0;
                u.u_utime = 0;
                u.u_cgbcnt = 0; u.u_gbcnt = 0;
                u.u_cdread = 0; u.u_dread = 0;
                u.u_cdwrit = 0; u.u_dwrit = 0;
                u.u_msgqhdr = NULL;
                u.u_acflag =| 1;
                u.u_lock = 0;
                return(0);
        }
        u.u_ar0[R0] = p2->p_pid;

out:
        u.u_ar0[R7] =+ 2;
        return(1);
}

/*
 * 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 =- ctos(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, RO))
                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);
}
```

```
*       @(#)sys2.c      2.8.1.1 */

#include "sys/param.h"
#include "sys/systm.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/reg.h"
#include "sys/file.h"
#include "sys/filex.h"
#include "sys/inode.h"
#include "sys/inodex.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/vtmn.h"
#include "sys/version.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, *ip, m;
        int fmt;

        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 {
                ip = fp->f_inode;
                if (fp->f_flag&FMP)
                        u.u_offset = 0;
                else
                        u.u_offset = fp->f_un.f_offset;
                fmt = ip->i_mode&IFMT;
                fmt = (fmt==IFREG || fmt==IFDIR || fmt==IFLRG || fmt==IFLDR);
                if(fmt)
                        plock(ip);
                if(m==FREAD)
                        readi(ip); else
                        writei(ip);
                if(fmt)
                        prele(ip);
                if ((fp->f_flag&FMP) == 0)
                        fp->f_un.f_offset += u.u_arg[1]-u.u_count;
        }
        u.u_ar0[R0] = u.u_arg[1]-u.u_count;
}

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

        if(u.u_arg[0] == -1) {  /* enable autoclose file descriptors */
                return;
        }
        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);
                        i = rip->i_mode&IFMT;
                        if(i==IFDIR || i==IFLDR)
                                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()
{
        register struct file *fp;
        register struct a {
                off_t off;
                int sbase;
                } *uap;
        int t;
        long n;
        int vflag;

        vflag = argadj(UV_CBR2,1);
        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        if(fp->f_flag&(FPIPE|FMP|FNPIPE)) {
                u.u_error = ESPIPE;
                return;
        }
        uap = (struct a *)u.u_arg;

        if(vflag) {
                t = u.u_arg[1];         /* CBR2: seek(int,int,int)      */
                n = (long) u.u_arg[0];

                if (t==3) {
                        n <<= 9;
                        n.hiword &= 0777;
                        t = 0;
                        }
                else if (t>2) {
                        n <<= 9;
                        t -= 3;
                        }
                else if (t==0)
                        n &= 01777771;
                uap->off = n;
                uap->sbase = t;
                }                               /* CBR2 end */
        if(uap->sbase==1)
                uap->off += fp->f_un.f_offset;
        else if (uap->sbase==2) {
                uap->off.hiword += fp->f_inode->i_size0&0377;
                uap->off += fp->f_inode->i_size1;
```

```
                }
        else if (uap->sbase != 0) {
                u.u_error = EINVAL;
                psignal(u.u_procp, SIGSYS);
                return;
                }
        fp->f_un.f_offset = uap->off;
        if (vflag == 0) {
                u.u_ar0[R0] = uap->off.hiword;
                u.u_ar0[R1] = uap->off.loword;
        }
}


/*
 * Tell -- discover offset of file R/W pointer
 */
tell()
{
        register struct file *fp;

        if(!VERSION(UV_CBR2)) {
                u.u_error = ENOENT;
                return;
        }
        if (fp = getf(u.u_ar0[R0])) {
                u.u_ar0[R0] = fp->f_un.f_offset.hiword;
                u.u_ar0[R1] = fp->f_un.f_offset.loword;
        }
}

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

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if(ip->i_nlink >= 127) {
                u.u_error = EMLINK;
                goto out;
        }
        /*
         * Unlock to avoid possibly hanging the namei.
         * Sadly, this means races. (Suppose someone
         * deletes the file in the meantime?)
         * Nor can it be locked again later
         * because then there will be deadly
         * embraces.
         */
        prele(ip);
```

```
        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;
        fmt = ip->i_mode&IFMT;
        if((fmt==IFDIR || fmt==IFLDR) && u.u_dbuf[0]!='.' && !suser()) {
                iput(u.u_pdir);
                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_un.i_rdev = u.u_arg[2];

out:
        iput(ip);
}

/*
 * sleep system call
 * not to be confused with the sleep internal routine.
 */
```

```
/******
sslep()
{
        register struct proc *pp;

        pp = u.u_procp;
        spl7();
        pp->p_flag =| SSLEP;
        VTPROCENT(pp,PR_FLAG);
        if(pp->p_clktim = u.u_ar0[R0])
        while(pp->p_clktim)
                sleep(pp, 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;
}
```

```
/*      @(#)sys3.c      2.10.1.1        */

#include "sys/param.h"
#include "sys/systm.h"
#include "sys/reg.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/filsys.h"
#include "sys/user.h"
#include "sys/usrx.h"
#include "sys/inode.h"
#include "sys/inodex.h"
#include "sys/file.h"
#include "sys/filex.h"
#include "sys/conf.h"
#include "sys/confx.h"
#include "sys/stat.h"
#include "sys/ioctl.h"
#include "sys/version.h"

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

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

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

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

/*
 * The basic routine for fstat and stat:
 * get the inode and pass appropriate parts back.
 */

struct  stat6
```

```
{
        int     oi_dev;
        int     oi_number;
        int     oi_mode;
        char    oi_nlink;
        char    oi_uid;
        char    oi_gid;
        char    oi_size0;
        unsigned        oi_size1;
        int     oi_addr[8];
        long    oi_atime;
        long    oi_mtime;
};

stat1(ip, ub, fp)
register struct inode *ip;
caddr_t ub;
struct file *fp;
{
        register *rst, *rip;
        int *bp;
        int i;
        struct stat6 ods;

        if (ip->i_flag & (IACC|IUPD|ICHG))
                iupdat(ip,&time);
/*
 *      First copy Version 6 file system data
 */

        rst = (int *) &ods;
        rip = &(ip->i_dev);
        for (i=0; i<14; i++)
                *rst++ = *rip++;
/*
 *      Next copy dates from the disk
 */
        bp = bread(ip->i_dev, (int) (((long) ip->i_number + 31L) / 16));
                /* magic number 24 = &0->di_atime */
        copyio(paddr(bp) + 32*(int) (((long)ip->i_number + 31L) % 16) + 24,
                rst, 2*sizeof(time_t), U_RKD);
        brelse(bp);

/*
 *      Now fix up pipes and named pipes
 */
        if(fp != NULL) {
                if (fp->f_flag & FNPIPE) {
                    fp = (*cdevsw[major(ip->i_un.i_rdev)].d_ioctl)
                        (ip->i_un.i_rdev, fp->f_flag & FWRITE ? GETWFP:GETRFP);
                    ip = fp->f_inode;
                }
                if (fp->f_flag & FPIPE) {
                    ods.oi_size1 = fp->f_flag & FREAD ?
                        ip->i_size1 - fp->f_un.f_offset.loword :
                        max (0, PIPSIZ - ip->i_size1);
```

```
                }
        }

        if (VERSION(UV_CBR3)) {
                struct stat ds;

                _cstat(&ods,&ds);
                if(copyout((caddr_t) &ds, ub, sizeof(ds))<0)
                        u.u_error = EFAULT;
        }
        else
                if (copyout((caddr_t)&ods, ub, sizeof(ods))<0)
                        u.u_error = EFAULT;
}


_cstat(bp6, bp7)
register struct stat6 *bp6;
register struct stat *bp7;
{
        union {
                long l;
                int i[2];
        } u;

        bp7->st_dev = bp6->oi_dev;
        bp7->st_ino = bp6->oi_number;
        bp7->st_mode = (bp6->oi_mode & ~IFMT);
        bp7->st_rdev = bp6->oi_addr[0];
        switch(bp6->oi_mode & IFMT) {
        case IFREG:
        case IFLRG:
                bp7->st_mode |= S_IFREG;
                bp7->st_rdev = 0;
                break;
        case IFDIR:
        case IFLDR:
                bp7->st_mode |= S_IFDIR;
                bp7->st_rdev = 0;
                break;
        case IFCHR:
                bp7->st_mode |= S_IFCHR;
                break;
        case IFBLK:
                bp7->st_mode |= S_IFBLK;
                break;
        case IFMPC:
                bp7->st_mode |= S_IFMPC;
                break;
        case IFMPB:
                bp7->st_mode |= S_IFMPB;
        }
        bp7->st_nlink = bp6->oi_nlink;
        bp7->st_uid = bp6->oi_uid & 0377;
        bp7->st_gid = bp6->oi_gid & 0377;
        u.i[0] = bp6->oi_size0;
```

```
        u.i[1] = bp6->oi_size1;
        bp7->st_size = u.l;
        bp7->st_atime = bp6->oi_atime;
        bp7->st_mtime = bp6->oi_mtime;
        bp7->st_ctime = bp6->oi_mtime;  /* creation time */
}

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

        fp = getf(u.u_ar0[R0].lobyte);
        if(fp == NULL)
                return;
        i = u.u_ar0[R0].hibyte & 0377;
        if (i > NOFILE || (i = ufalloc(i)) < 0)
                return;
        u.u_ofile[i] = fp;
        fp->f_count++;
}

/*
 * the file control system call.
 */
fcntl()
{
        register struct file *fp;
        register i;

        fp = getf(u.u_ar0[R0]);
        if (fp == NULL)
                return;
        switch(u.u_arg[0]) {
        case 0:
                i = u.u_arg[1];
                if (i < 0 || i > NOFILE || (i = ufalloc(i)) < 0)
                        return;
                u.u_ofile[i] = fp;
                fp->f_count++;
                break;

        case 1:
                u.u_ar0[R0] = u.u_pofile[u.u_ar0[R0]];
                break;

        case 2:
                u.u_pofile[u.u_ar0[R0]] = u.u_arg[1]&03;
                break;

        default:
                u.u_error = EINVAL;
        }
}
```

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

        d = getmdev();
        if(u.u_error)
                return;
        u.u_dirp = u.u_arg[1];
        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        fmt = ip->i_mode&IFMT;
        if(ip->i_count!=1 || fmt==IFBLK || fmt==IFCHR || fmt==IFMPC
           || fmt==IFMPB)
                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 out1;
        mp = bread(d, 1);
        if(u.u_error) {
            /* Should really only call close here if noone else has it open. */
                (*bdevsw[d.d_major].d_close) (d, 0);
                brelse(mp);
                goto out1;
        }
        smp->m_inodp = ip;
        smp->m_dev = d;
        if((smp->m_bufp = malloc(swapmap, 1)) == NULL)
                panic(ooswap);
        smp->m_bufp = agetblk(swapdev, smp->m_bufp);
        copyio(mp->b_paddr, (caddr_t)smp->m_bufp->b_paddr, BSIZE, U_RKD);
        brelse(mp);
        mp = (caddr_t)smp->m_bufp->b_paddr;
        mp->s_ilock = 0;
```

```
        mp->s_flock = 0;
        if(mp->s_ronly =| u.u_arg[2] & 1)
                smp->m_flags = M_RONLY|M_INCOR;
        else
                smp->m_flags = M_INCOR;
        suincnt++;
        ip->i_flag =| IMOUNT;
        prele(ip);
        if (suincnt >= NBUF-NFBUF)
                update();
        return;

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

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

        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:
        xflsh(d);
        for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
                if(ip->i_number!=0 && d==ip->i_dev) {
                        u.u_error = EBUSY;
                        return;
                }
        if(mp->m_flags&M_INCOR) {
                blkno = mp->m_bufp->b_blkno;
                brelse(mp->m_bufp);
                mp->m_bufp = blkno;
                suincnt--;
        }
        mfree(swapmap, 1, mp->m_bufp);
        (*bdevsw[d.d_major].d_close)(d, 0);
        ip = mp->m_inodp;
        ip->i_flag =& ~IMOUNT;
        plock(ip);
```

```
        iput(ip);
        mp->m_bufp = NULL;
}

/*
 * 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 *ip;
        int d;
        extern uchar;

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

```
/*      @(#)sys4.c      2.18.1.2        */

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

#include "sys/param.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/reg.h"
#include "sys/inode.h"
#include "sys/inodex.h"
#include "sys/systm.h"
#ihclude "sys/proc.h"
#include "sys/procx.h"
#include "sys/timeb.h"
#include "sys/vtmn.h"
#include "sys/version.h"

short   timezone = TIMEZONE;
short   dstflag = DSTFLAG;

gtime()
{
        u.u_ar0[R0] = time.hiword;
        u.u_ar0[R1] = time.loword;
}

/*
 * New time entry-- return TOD with milliseconds, timezone,
 * DST flag
 */
ftime()
{
        register struct a {
                struct  timeb   *tp;
        } *uap;
        struct timeb t;
        register unsigned ms;

        uap = (struct a *)u.u_arg;
        spl7();
        t.time = time;
        ms = lbolt;
        spl0();
        while (ms >= HZ) {
                ms -= HZ;
                t.time++;
        }
        t.millitm = (1000*ms)/HZ;
        t.timezone = timezone;
        t.dstflag = dstflag;
        if (copyout((caddr_t)&t, (caddr_t)uap->tp, sizeof(t)) < 0)
                u.u_error = EFAULT;
```

```
}

stime()
{
        long    ntime;

        if(suser()) {
                ntime.hiword = u.u_ar0[R0];
                ntime.loword = u.u_ar0[R1];
                logtchg(ntime);
                time = ntime;
        }
}

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()
{
        if(VERSION(UV_CBR2)) {
                u.u_ar0[R0].hibyte = u.u_uid;
                u.u_ar0[R0].lobyte = u.u_ruid;
        } else {
                u.u_ar0[R1] = u.u_uid & 0377;
                u.u_ar0[R0] = u.u_ruid & 0377;
        }
}

setgid()
{
        register gid;

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

getgid()
{
        if(VERSION(UV_CBR2)) {
                u.u_ar0[R0].hibyte = u.u_gid;
                u.u_ar0[R0].lobyte = u.u_rgid;
        } else {
                u.u_ar0[R1] = u.u_gid & 0377;
                u.u_ar0[R0] = u.u_rgid & 0377;
```

```
        }
}

getpid()                /* get own and parent process id */
{

        u.u_ar0[R0] = u.u_procp->p_pid;
        u.u_ar0[R1] = u.u_procp->p_ppid;
}

sync()
{

        update();
}

nice()
{
        register n;

        n = u.u_ar0[R0];
        if(n > 20)
                n = 20;
        if(n < -1 && !suser())
                n = 0;
        u.u_ar0[R0] = u.u_procp->p_nice;
        u.u_procp->p_nice = n;
}

/*
 * Unlink system call.
 * Hard to avoid races here, especially
 * in unlinking directories.
 */
unlink()
{
        register *ip, *pp;
        extern uchar;

        pp = namei(&uchar, 2);
        if(pp == NULL)
                return;
        /*
         * Check for unlink(".")
         * to avoid hanging on the iget
         */
        if (pp->i_number != u.u_dent.u_ino)
                ip = iget(pp->i_dev, u.u_dent.u_ino);
        else {
                ip = pp;
                ip->i_count++;
        }
        if(ip == NULL)
                goto out1;
        if(((ip->i_mode&IFMT)==IFDIR || (ip->i_mode&IFMT)==IFLDR)
           && u.u_dbuf[0]!='.' && !suser())
```

```
                goto out;
        /*
         * Don't unlink a mounted file.
         */
        if (ip->i_dev != pp->i_dev) {
                u.u_error = EBUSY;
                goto out;
        }
        if (ip->i_flag&ITEXT && ip->i_nlink==1) {
                u.u_error = ETXTBSY;
                goto out;
        }
        u.u_offset.loword -= DIRSIZ+2;
        u.u_base = &u.u_dent;
        u.u_count = DIRSIZ+2;
        u.u_dent.u_ino = 0;
        writei(pp);
        ip->i_nlink--;
        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 && (ip->i_mode&IFMT)!=IFLDR) {
                u.u_error = ENOTDIR;
        bad:
                iput(ip);
                return;
        }
        if(access(ip, IEXEC))
                goto bad;
        prele(ip);
        plock(u.u_cdir);
        iput(u.u_cdir);
        u.u_cdir = ip;
}

chmod()
{
        register *ip;

        if ((ip = owner()) == NULL)
                return;
        if (u.u_uid) {
                if (u.u_gid != ip->i_gid  && u.u_arg[1] & ISGID) {
```

```
                        u.u_error = EPERM;
                        goto bad;
                }
                u.u_arg[1] =& ~ISVTX;
        }
        ip->i_mode =& ~07777;
        ip->i_mode =| u.u_arg[1]&07777;
        ip->i_flag =| IUPD;
bad:
        iput(ip);
}

chown()
{
        register *ip, vflag;

        vflag = argadj(UV_CBR2,1);
        if ((ip = owner()) == NULL)
                return;
        ip->i_uid = u.u_arg[1].lobyte;
        if(u.u_uid != 0)
                ip->i_mode =& ~ISUID;
        ip->i_flag =| IUPD;
        if(vflag == 0)
                schgrp(ip,u.u_arg[2].lobyte);
        iput(ip);
}

/* CBR2 Vestigial Change Group */
chgrp()
{
        register *ip;

        if ((ip = owner()) == NULL)
                return;
        schgrp(ip,u.u_arg[1].lobyte);
        ip->i_flag =| IUPD;
        iput(ip);
}

schgrp(ip,a)
register *ip;
char a;
{
        ip->i_gid = a;
        if(u.u_uid != 0)
                ip->i_mode &= ~ISGID;
}

/*
 * Change modified date of file:
 * time to r0-r1; sys utime; file
 */
utime()
{
        register struct inode *ip;
```

```
        register vflag;
        int tbuf[2];

        vflag = argadj(UV_CBR2,1);
        if ((ip = owner()) == NULL)
                return;
        if (vflag) {    /* CBR2 -- time in R0, R1       */
                tbuf[0] = u.u_ar0[R0];
                tbuf[1] = u.u_ar0[R1];
        } else {                /* CBR3 -- time in structure*/
                if(copyin((caddr_t)(u.u_arg[1]+4), (caddr_t) tbuf, 4)) {
                        u.u_error = EFAULT;  /*Note: first argument ignored */
                        goto out;
                        }
                }
        ip->i_flag |= IUPD;
        iupdat(ip, tbuf);
out:
        iput(ip);
}

ssig()
{
        register a;
        register struct proc *p;

        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];
        u.u_procp->p_sig =& ~(1L<<(a-1));
        VTPROCENT(u.u_procp,PR_SIG);
        if(a == SIGCLD) {
                a = u.u_procp->p_pid;
                for(p = &proc[0]; p < &proc[NPROC]; p++)
                        if(a == p->p_ppid && p->p_stat == SZOMB)
                                psignal(u.u_procp, SIGCLD);
        }
}

kill()
{
        register struct proc *p;
        register dest, f;
        int mypgrp, signo;
        int oneflg;

        f = 0;
        dest = u.u_ar0[R0];
        mypgrp = u.u_procp->p_pgrp;
        signo = u.u_arg[0];
        oneflg = dest > 0 && signo > 0;
        if (oneflg)
```

```
                p = &proc[0];
        else
                p = &proc[2];
        for(; p < procend; p++) {
                if(p->p_stat == NULL)
                        continue;
                if(oneflg && p->p_pid != dest)
                        continue;
                if(dest == 0 && p->p_pgrp != mypgrp)
                        continue;
                if(signo < 0) {
                        if (dest == 0 || dest == -1) {
                                if (p->p_pid == u.u_procp->p_pid)
                                        continue;
                        } else if(dest != p->p_pgrp)
                                continue;
                }
                if(u.u_uid != 0 && u.u_uid != p->p_uid)
                        if (oneflg) {
                                u.u_error = EPERM;
                                return;
                        } else
                                continue;
                f++;
                psignal(p, signo >= 0 ? signo : -signo);
                if (oneflg)
                        break;
        }
        if(f == 0)
                u.u_error = ESRCH;
}

times()
{
        register *p;

        for(p = &u.u_utime; p  < &u.u_utime+4;) {
                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;
        spl7();
        c = p->p_clktim;
        p->p_clktim = u.u_ar0[R0];
        spl0();
        VTPROCENT(p,PR_CLOCK);
        VTPROCENT(p,PR_FLAG);
        u.u_ar0[R0] = c;
}

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

        pp = u.u_procp;
        spl7();
        while(pp->p_clktim)
                sleep(&u, PSLEP);
        spl0();
}

chroot()
{
        register *oldroot, *ip;
        extern uchar;

        if(VERSION(UV_CBR2)) {
                chgrp();                /* CBR2 chgrp sys entry */
                return;
        }
        if (!suser())
                return;
        oldroot = u.u_rdir;
        u.u_rdir = rootdir;
        ip = namei(&uchar, 0);
        if (ip == NULL)
                goto bad;
        if((ip->i_mode&IFMT)!=IFDIR && (ip->i_mode&IFMT)!=IFLDR) {
                u.u_error = ENOTDIR;
                iput(ip);
        bad:
                u.u_rdir = oldroot;
                return;
        }
        prele(ip);
        plock(oldroot);
        iput(oldroot);
        u.u_rdir = ip;
}
```

```
#define PERM 0777       /* permission part of inode mode */
umask()
{
        register svmask;

        svmask = u.u_mask;
        u.u_mask = u.u_arg[0] & PERM;
        u.u_ar0[R0] = svmask;
}

ulimit()
{
        if(argadj(UV_CBR2,2))
                u.u_ar0[R0] = SW->integ;        /* CBR2: getswitch      */
        else {
                u.u_ar0[R0] = 0;
                u.u_ar0[R1] = 2048;
                }
}
```

```
/*      @(#)sys5.c      2.9.1.1 */

#
/*
 *              semaphore operations
 *              setpgrp sys call
 */

#include "sys/param.h"
#include "sys/systm.h"
#include "sys/reg.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/file.h"
#include "sys/proc.h"
#include "sys/vtmn.h"
#include "sys/version.h"

#define PEVT    1

#define POST    0
#define BLOCK   1
#define P       2
#define V       3
#define TEST    4
#define SETSEM  5
#define LOCK    6
#define UNLOCK  7
#define TLOCK   8
#define RDSEM   9
#define NOAULK  10

#define SEMNOAULK       01
#define SEMUSED         02

#ifdef SEMATEST
int     sematest        78;
#endif
int  semi[NEVT] ;

event()
{
        register int s;
        register int op;
        register int sav;

        op = u.u_arg[0];
        if(op == NOAULK) {
                u.u_semflg |= SEMNOAULK;
                return;
        }
        s = u.u_arg[1];
        if (s<0 || s>=NEVT) {
                u.u_error = EINVAL;
                return;
```

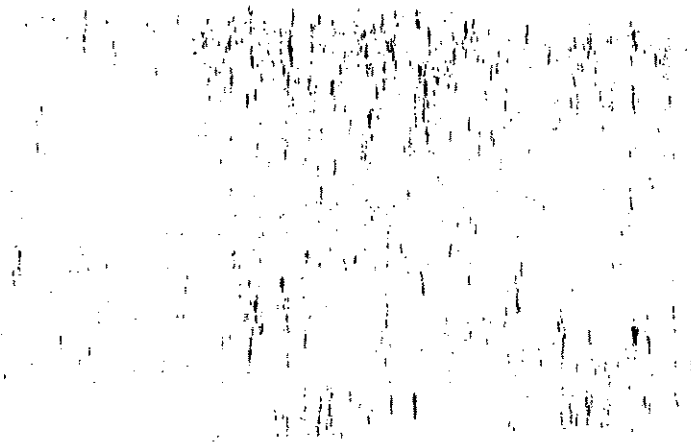
```
        }
#ifdef SEMATEST
        if (s == sematest) {
                printf("semaop %d(%d); %s; uid=%d; pid=%d\n", op, sematest,
                        u.u_comm, u.u_ruid, u.u_procp->p_pid);
        }
#endif

        switch (op) {
                case P:
                case V:
                case TEST:
                case POST:
                case BLOCK:
                        if (semi[s] < 0) {
                                u.u_error = EFUNC;
                                return;
                        }
                        break;

                case LOCK:
                case TLOCK:
                        if (semi[s] > 0) {
                                u.u_error = EFUNC;
                                return;
                        }
                        break;

                case UNLOCK:
                        if(semi[s] != -u.u_procp->p_pid) {
                                u.u_error = EFUNC;
                                return;
                        }
                        break;
        }

        switch (op) {
                case V:
                case POST:
                        u.u_ar0[R0] = semi[s];
                        if (++semi[s] < 0)
                                semi[s] = 1;
                        wakeup(&semi[s]);
                        return;
                case P:
                        while (semi[s]==0) sleep(&semi[s],PEVT);
                        u.u_ar0[R0] = semi[s];
                        semi[s]--;
                        return;
                case BLOCK:
                        u.u_semav = semi[s];
                        do {
                                sleep(&semi[s],PEVT);
                        } while(u.u_semav == semi[s]);
                        u.u_ar0[R0] = u.u_semav;
                        return;
```



```
                case TEST:
                        u.u_ar0[R0] = semi[s];
                        if (semi[s] != 0) semi[s]--;
                        return;
                case SETSEM:
                        sav = semi[s];
                        semi[s] = u.u_ar0[R0];
                        u.u_ar0[R0] = sav;
                        wakeup(&semi[s]);
                        return;
                case LOCK:
                        while((u.u_ar0[R0] = semi[s]) != 0)
                                sleep(&semi[s], PEVT);
                lck:
                        semi[s] = -u.u_procp->p_pid;
                        u.u_semflg |= SEMUSED;
                        return;
                case TLOCK:
                        if((u.u_ar0[R0] = semi[s]) != 0)
                                return;
                        goto lck;
                case UNLOCK:
                        u.u_ar0[R0] = semi[s];
                        semi[s] = 0;
                        wakeup(&semi[s]);
                        return;
                case RDSEM:
                        u.u_ar0[R0] = semi[s];
                        return;
        }
        u.u_error=ENOENT;
}

semafree()
{
        register i, pid, flgs;

        flgs = u.u_semflg;
        u.u_semflg = 0;
        if ((flgs&SEMUSED)==0 || (flgs&SEMNOAULK)!=0)
                return;
        pid = -u.u_procp->p_pid;
        for(i=0; i<NEVT; i++)
                if (semi[i] == pid) {
                        semi[i] = 0;
                        wakeup(&semi[i]);
                }
}

setpgrp()
{
        register oldgrp;
        register newgrp;

        oldgrp = u.u_procp->p_pgrp;
        if(newgrp = u.u_ar0[R0]) {
```

```
                if (newgrp == -1)
                        u.u_procp->p_pgrp = 0 ;
                else
                        u.u_procp->p_pgrp = newgrp;
                VTPROCENT(u.u_procp,PR_GROUP);
        }
        u.u_ar0[R0] = oldgrp;
}

/* Adjust arguments by narg if a.out version matches spec */
/*  and system call is direct                             */

argadj(version,narg)
register version, narg;
{
        register flag;
        if((flag = VERSION(version)) && (u.u_indflg == 0))
                u.u_ar0[R7] -= (narg << 1);
        return(flag);
}
```

```
/*      @(#)syscb.c     2.11    */

#include "sys/param.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/text.h"
#include "sys/textx.h"
#include "sys/seg.h"
#include "sys/systm.h"
#include "sys/syserr.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/reg.h"
#include "sys/lock.h"
#include "sys/sprof.h"
#include "sys/sprofx.h"
#include "sys/version.h"


#define CB_GETU         0
#define CB_CSW          1
#define CB_REBOOT       2
#define CB_LOCK         3
#define CB_SPROF        4

int     rebunit;        /* device and unit spec for DEC YC ROM */
char    rebstr[14];     /* file name to reboot terminated by newline */


syscb()
{

        switch(u.u_ar0[R1]) {

        case CB_GETU:
                {
                        register cnt, offset;

                        offset = u.u_ar0[R0];
                        cnt = u.u_arg[1];

                        if(cnt+offset > USIZE*64)
                                u.u_ar0[R0] = 0;
                        else if(copyout(&u.lobyte+offset,u.u_arg[0],cnt) == -1)
                                u.u_error = EFAULT;
                        else
                                u.u_ar0[R0] = cnt;
                }
                return;

        case CB_CSW:
                {
                        u.u_ar0[R0] = SW->integ;
                        return;
                }
```

```
        case CB_REBOOT:
                {
                        register char *p;
                        register int c;

                        if(!suser())
                                return;
                        rebunit = u.u_ar0[R0];
                        p = rebstr;
                        while((c = uchar()) != -1) {
                                *p++ = c;
                                if(c == '\n')
                                        reboot();       /* no return */
                                if(p == &rebstr[sizeof(rebstr)])
                                        break;
                        }
                }
                return;

#ifdef PTLOCK
        case CB_LOCK:
                lock();
                return;
#endif

#ifdef SPROF
        case CB_SPROF:
                sprofil();
                return;
#endif

        default:
                u.u_error = EINVAL;
                return;
        }
}

#ifdef PTLOCK
lock()
{
        if(!suser())
                return;
        switch(u.u_arg[0])
        {
                case TXTLOCK:
                        if((u.u_lock&(PROCLOCK|TXTLOCK))  || textlock() == 0)
                                goto bad;
                        break;
                case PROCLOCK:
                        if(u.u_lock&(PROCLOCK|TXTLOCK))
                                goto bad;
                        textlock();
                        proclock();
                        break;
                case PUNLOCK:
```

```
                        if(punlock() == 0)
                                goto bad;
                        break;
                case TUNLOCK:
                        if((u.u_lock&PROCLOCK) || tunlock() == 0)
                                goto bad;
                        break;
        }
        if(runout)
        {
                runout = 0;
                wakeup(&runout);
        }
        sleep(&runlock,-1);
        return;
bad:
        u.u_error = ELOCK;
}
textlock()
{
        struct text *xp;

        if((xp=u.u_procp->p_textp) == NULL )
                return(0);
        u.u_lock =| TXTLOCK;
        if(xp->x_lcount++ == 0)
        {
        /* bump runlock to notify sched to shuffle core next pass */
                runlock++;
                xp->x_ccount++;
        }
        return(1);
}

tunlock()
{
        struct text *xp;

        if((xp=u.u_procp->p_textp) == NULL || (u.u_lock&TXTLOCK)==0)
                return(0);
        u.u_lock =& ~TXTLOCK;
        if( --(xp->x_lcount) == 0)
        {
                xccdec(xp);
        /* bump runlock to notify sched to shuffle core next pass */
                runlock++;
        }
        return(1);
}

proclock()
{
        u.u_procp->p_flag =| SSYS;
        u.u_lock =| PROCLOCK;
        runlock++;
}
```

```
punlock()
{
        if((u.u_lock&(PROCLOCK|TXTLOCK)) == 0)
                return(0);
        u.u_procp->p_flag =& ~SSYS;
        u.u_lock =& ~PROCLOCK;
        tunlock();
        runlock++;
        return(1);
}
#endif

#ifdef SPROF
#define BLKMSK  077
#define PLF     0077400

struct sysprof sysprof = {NULL, NULL, 0,0,0,0,0};

sprofil ()

/*
 *      in R0:  address of counters area (SPCNT struct)
 *      arg[0]: number of system routines (r option)
 *              number of intervals (i option)
 *      arg[1]: lowpc for i option (word, not byte address)
 *      arg[2]: interval size for i option (number of words)
 *              0 for r option
 *
 *      Notice that arg[2] is the flag that determines if the
 *      r or i option is being used.
 *
 *      R0==0 implies that system profiling is to be turned off
 */

{
        register unsigned int b;
        register struct sysprof *tp;
        int     blockno;
        int     szSPCNT;

        tp = &sysprof;

        if(b=u.u_ar0[R0]){
                /*
                 *      kernel D-space register 5 will be
                 *      used to access the SPCNT structure
                 *      in the clock routine.
                 *      it should point to the beginning of the
                 *      block where the SPCNT struct starts
                 */
                if (tp->pid) {
                        u.u_error = EBUSY;
                        return;
                }
```

```
                tp->base = 0120000 | (b & BLKMSK);
                tp->numcnts = u.u_arg[0];
                tp->lowpc = u.u_arg[1];
                tp->intsize = u.u_arg[2];

                /*      see if table too big */
                szSPCNT = sizeof(tp->base->b_urhits) +
                        sizeof(tp->base->b_syhits) +
                        sizeof(tp->base->b_idhits) + tp->numcnts *
                        (tp->intsize ? sizeof(spcnt_t) : sizeof(struct NHIT));
                if ((szSPCNT > 8192 - (b & BLKMSK)) ||
                        (tp->numcnts == 0) ||
                        ((caddr_t)tp->base & 01) ) {
                        u.u_error = EINVAL;
                        return;
                }

                blockno = (b>>6) & 0177; /* block number in (user) page */

                /*      get relative number of last block in SPCNT */
                szSPCNT = (((b & BLKMSK) + szSPCNT) >> 6) & 0177;

                b = (b>>13) & 07;       /* get (user) page number */
                /*
                 *      point to block in page where SPCNT
                 *      starts
                 */
                u.u_procp->p_flag |= SLOCK;     /* lock the data in memory */
                tp->newpg.par = UDSA->r[b] + blockno;
                tp->newpg.pdr = (UDSD->r[b] & ~PLF) | (szSPCNT << 8);
                tp->pid = u.u_procp->p_pid;

#ifdef IPROFCLK

                KW11K->kw11kb = -150;   /* interrupt every 66.6th of a sec */
                KW11K->kw11ks = 0507;   /* start independent prof k clock */

#else
#ifdef IPROFCLB
                *TCU100 = TCURATE;      /* interrupt a little over 60 times/sec
*   */
                                        /* start indep battery TCU-100 clk */
#endif
#endif

        } else {
                sprofoff();
        }
        u.u_ar0[R0] = 0;
}


sprofoff()
{
        /*
         *      clock will stop profiling when pid
```

```
         *      is set to zero
         */

#ifdef IPROFCLK
        KW11K->kw11ks = 0;      /* turn off profiling clock */
#else
#ifdef IPROFCLB
        *TCU100 = 0;            /* turn off prof battery clock */
#endif
#endif

        sysprof.pid = 0;
        u.u_procp->p_flag =& ~SLOCK;
}
#endif
```

```
/*      @(#)sysent.c    2.12    */

#
#include "sys/param.h"
/*
 * 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.
 */

#include "sys/maus.h"

extern nullsys(), nosys();
extern rexit(), fork(), read(), write(), open(), close(), wait();
extern creat(), link(), unlink(), exec(), chdir(), gtime(), mknod(), chmod();
extern exece();
extern chown(), sbreak(), stat(), seek(), getpid(), smount(), sumount();
extern setuid(), getuid(), stime(), ptrace(), alarm(), fstat(), pause();
extern utime(), stty(), gtty(), saccess(), nice(), sync(), kill();
extern ulimit(), setpgrp(), tell(), dup(), pipe(), times(), profil();
extern setgid(), getgid(), ssig(), messag(), sysacct();
extern syscb(), chroot(), umask(), event();
extern utssys();
extern fcntl();
extern ftime();
extern ioctl();

extern mpxchan();

#ifdef MAKEMAUS
extern maus();
#endif

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 */
        3, &chown,                      /* 16 = chown */
        1, &sbreak,                     /* 17 = break */
        2, &stat,                       /* 18 = stat */
```

```
        3, &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 */
        2, &utime,                      /* 30 = utime */
        1, &stty,                       /* 31 = stty */
        1, &gtty,                       /* 32 = gtty */
        2, &saccess,                    /* 33 = access */
        0, &nice,                       /* 34 = nice */
        1, &ftime,                      /* 35 = ftime; formerly sleep */
        0, &sync,                       /* 36 = sync */
        1, &kill,                       /* 37 = kill */
        2, &ulimit,                     /* 38 = ulimit */
        0, &setpgrp,                    /* 39 = setpgrp */
        0, &tell,                       /* 40 = tell */
        0, &dup,                        /* 41 = dup */
        0, &pipe,                       /* 42 = pipe */
        1, &times,                      /* 43 = times */
        4, &profil,                     /* 44 = prof */
        3, &syscb,                      /* 45 = syscb (tiu on research) */
                                        /* (getu,getcsw,reboot,lock,sprofil */
        0, &setgid,                     /* 46 = setgid */
        0, &getgid,                     /* 47 = getgid */
        2, &ssig,                       /* 48 = sig */
        4, &message,                    /* 49 = message */
        0, &nosys,                      /* 50 = x (formerly serrlog) */
        1, &sysacct,                    /* 51 = turn acct off/on */
        0, &nosys,                      /* 52 = x */
        0, &nosys,                      /* 53 = smux */
        3, &ioctl,                      /* 54 = ioctl */
        0, &nosys,                      /* 55 = x */
        2, &mpxchan,                    /* 56 = create mpx comm channel */
        1, &utssys,                     /* 57 = utssys */
#ifdef MAUS
        1, &maus,                       /* 58 = set up MAUS segment reg */
#else
        0, &nosys,                      /* 58 = maus    */
#endif
        3, &exece,                      /* 59 = exece */
        1, &umask,                      /* 60 = umask */
        2, &chroot,                     /* 61 = chroot */
        2, &fcntl,                      /* 62 = fcntl   */
        2, &xvent                       /* 63 = eventc */
};

/*
 * Dummy entry for illegal system calls
 */
```

```
int badent[] {
        0, &nosys
};
```

```
/*      @(#)sysent.tu.c 2.1     */

/*      Based on sysent.c       2.11    */

#
#include "sys/param.h"
/*
 * 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.
 */


extern nullsys(), nosys();
extern rexit(), fork(), read(), write(), open(), close(), wait();
extern creat(), link(), unlink(), exec(), chdir(), gtime(), mknod(), chmod();
extern exece();
extern chown(), sbreak(), stat(), seek(), getpid(), smount(), sumount();
extern setuid(), getuid(), stime(), ptrace(), alarm(), fstat(), pause();
extern utime(), stty(), gtty(), saccess(), nice(), sync(), kill();
extern ulimit(), setpgrp(), tell(), dup(), pipe(), times(), profil();
extern setgid(), getgid(), ssig(), messag(), sysacct();
extern syscb(), chroot(), umask(), event();
extern fcntl();
extern ftime();
extern ioctl();


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 */
        3, chown,                       /* 16 = chown */
        1, sbreak,                      /* 17 = break */
        2, stat,                        /* 18 = stat */
        3, 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 */
        2, utime,                       /* 30 = utime */
        1, stty,                        /* 31 = stty */
        1, gtty,                        /* 32 = gtty */
        2, saccess,                     /* 33 = access */
        0, nice,                        /* 34 = nice */
        1, ftime,                       /* 35 = ftime; formerly sleep */
        0, sync,                        /* 36 = sync */
        1, kill,                        /* 37 = kill */
        2, ulimit,                      /* 38 = ulimit */
        0, setpgrp,                     /* 39 = setpgrp */
        0, tell,                        /* 40 = tell */
        0, dup,                         /* 41 = dup */
        0, pipe,                        /* 42 = pipe */
        1, times,                       /* 43 = times */
        4, profil,                      /* 44 = prof */
        3, syscb,                       /* 45 = syscb (tiu on research) */
                                        /* (getu,getcsw,reboot,lock,sprofil */
        0, setgid,                      /* 46 = setgid */
        0, getgid,                      /* 47 = getgid */
        2, ssig,                        /* 48 = sig */
        0, nosys,                       /* 49 = message */
        0, nosys,                       /* 50 = x (formerly serrlog) */
        0, nosys,                       /* 51 = turn acct off/on */
        0, nosys,                       /* 52 = x */
        0, nosys,                       /* 53 = smux */
        3, ioctl,                       /* 54 = ioctl */
        0, nosys,                       /* 55 = x */
        0, nosys,                       /* 56 = creat mpx comm channel */
        0, nosys,                       /* 57 = x (was 3call) */
        0, nosys,                       /* 58 = mau:     */
        3, exece,                       /* 59 = exece */
        1, umask,                       /* 60 = umask */
        2, chroot,                      /* 61 = chroot */
        2, fcntl,                       /* 62 = fcntl    */
        2, event                        /* 63 = event */
};

/*
 * Dummy entry for illegal system calls
 */

int badent[] = {
        0, nosys
};
```

```
/*      @(#)sysent.util.        2.1     */

/*      Based on sysent.c       2.11    */

#
#include "sys/param.h"
/*
 * 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.
 */


extern nullsys(), nosys();
extern rexit(), fork(), read(), write(), open(), close(), wait();
extern creat(), link(), unlink(), exec(), chdir(), gtime(), mknod(), chmod();
extern exece();
extern chown(), sbreak(), stat(), seek(), getpid(), smount(), sumount();
extern setuid(), getuid(), stime(), ptrace(), alarm(), fstat(), pause();
extern utime(), stty(), gtty(), saccess(), nice(), sync(), kill();
extern ulimit(), setpgrp(), tell(), dup(), pipe(), times(), profil();
extern setgid(), getgid(), ssig(), messag(), sysacct();
extern syscb(), chroot(), umask(), event();
extern fcntl();
extern ftime();
extern ioctl();


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 */
        3, chown,                       /* 16 = chown */
        1, sbreak,                      /* 17 = break */
        2, stat,                        /* 18 = stat */
        3, 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 */
        2, utime,                       /* 30 = utime */
        1, stty,                        /* 31 = stty */
        1, gtty,                        /* 32 = gtty */
        2, saccess,                     /* 33 = access */
        0, nice,                        /* 34 = nice */
        1, ftime,                       /* 35 = ftime; formerly sleep */
        0, sync,                        /* 36 = sync */
        1, kill,                        /* 37 = kill */
        2, ulimit,                      /* 38 = ulimit */
        0, setpgrp,                     /* 39 = setpgrp */
        0, tell,                        /* 40 = tell */
        0, dup,                         /* 41 = dup */
        0, pipe,                        /* 42 = pipe */
        1, times,                       /* 43 = times */
        4, profil,                      /* 44 = prof */
        3, syscb,                       /* 45 = syscb (tiu on research) */
                                        /* (getu,getcsw,reboot,lock,sprofil */
        0, setgid,                      /* 46 = setgid */
        0, getgid,                      /* 47 = getgid */
        2, ssig,                        /* 48 = sig */
        0, nosys,                       /* 49 = message */
        0, nosys,                       /* 50 = x (formerly serrlog) */
        0, nosys,                       /* 51 = turn acct off/on */
        0, nosys,                       /* 52 = x */
        0, nosys,                       /* 53 = smux */
        3, ioctl,                       /* 54 = ioctl */
        0, nosys,                       /* 55 = x */
        0, nosys,                       /* 56 = creat mpx comm channel */
        0, nosys,                       /* 57 = x (was scall) */
        0, nosys,                       /* 58 = mass    */
        3, exece,                       /* 59 = exece */
        1, umask,                       /* 60 = umask */
        2, chroot,                      /* 61 = chroot */
        2, fcntl,                       /* 62 = fcntl   */
        2, event                        /* 63 = event */
};

/*
 * Dummy entry for illegal system calls
 */

int badent[] = {
        0, nosys
};
```

```
/*      @(#)text.c      2.7     */

#
#include "sys/param.h"
#include "sys/systm.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/proc.h"
#include "sys/text.h"
#include "sys/textx.h"
#include "sys/inode.h"
#include "sys/inodex.h"
#include "sys/buf.h"
#include "sys/bufx.h"
#include "sys/seg.h"
#include "sys/sysmes.h"
#include "sys/sysmesx.h"
#include "sys/vtmm.h"
#include "sys/elog.h"

char *ooswap "out of swap space";

/*
 * 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
 */
xswap(p, ff, os)
int *p;
{
        register *rp, a;

        rp = p;
        if(os == 0)
                os = rp->p_size;
        a = malloc(swapmap, ctod(rp->p_size));
        if(a == NULL)
                panic(ooswap);
        rp->p_flag =| SLOCK;
        VTPROCHIT(rp,PR_FLAG);
        xccdec(rp->p_textp);
        swap(a, rp->p_addr, os, B_WRITE);
        if(ff)
                mfree(coremap, os, rp->p_addr);
        rp->p_addr = a;
        rp->p_flag =& ~(SLOAD|SLOCK);
        rp->p_time = 0;
        VTPROCHIT(rp,PR_SWP);
        if(runout) {
                runout = 0;
```

```
                wakeup(&runout);
        }
}

/*
 * flush out any inactive saved text for device dev.
 * NODEV => all inactive saved text.
 */
xflsh(dev)
register int dev;
{
        register struct inode *ip;
        register struct text *xp;
        int count;

        count = 0;
        for(xp=text; xp<&text[NTEXT]; xp++) {
                if((ip = xp->x_iptr) == NULL || xp->x_count != 0)
                        continue;
                if(dev != NODEV && dev != ip->i_dev)
                        continue;
                xp->x_iptr = NULL;
                mfree(swapmap, ctod(xp->x_size), xp->x_daddr);
                ip->i_flag =& ~ITEXT;
                if(ip->i_flag&ILOCK)
                        ip->i_count--;
                else
                        iput(ip);
                count++;
        }
        return(count);
}

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

        if((xp=u.u_procp->p_textp) == NULL)
                return;
        xlock(xp);
        xp->x_flag =& ~XLOCK;
        u.u_procp->p_textp = NULL;
        ip = xp->x_iptr;
        if(--xp->x_count==0 && (ip->i_mode&ISVTX)==0) {
                xp->x_iptr = NULL;
                mfree(swapmap, ctod(xp->x_size), xp->x_daddr);
                mfree(coremap, xp->x_size, xp->x_caddr);
                ip->i_flag =& ~ITEXT;
                if (ip->i_flag&ILOCK)
                        ip->i_count--;
                else
```

```
                        iput(ip);
        } else
                xccdec(xp);
}

/*
 * 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); the written bit is set to force it
 * to be written out as appropriate.
 * If it is being used, but is not currently in core,
 * a swap has to be done to get it back.
 */
xalloc(ip)
int *ip;
{
        register struct text *xp;
        register *rp, ts;

        if(u.u_exdata.ux_tsize == 0)
                return;
        rp = NULL;
loop:
        for (xp = &text[0]; xp < &text[NTEXT]; xp++) {
                if(xp->x_iptr == NULL) {
                        if(rp == NULL)
                                rp = xp;
                        continue;
                }
                if(xp->x_iptr == ip) {
                        xlock(xp);
                        xp->x_count++;
                        u.u_procp->p_textp = xp;
                        if (xp->x_ccount == 0)
                                xexpand(xp);
                        else
                                xp->x_ccount++;
                        xunlock(xp);
                        return;
                }
        }
        if((xp=rp) == NULL) {
                if(xflsh(NODEV))
                        goto loop;
                meas.m_tovfl++;
                logovfl(E_TEXTO);
                printf("out of text");
                psignal(u.u_procp, SIGKIL);
                return;
        }
        xp->x_flag = XLOAD|XLOCK;
        xp->x_count = 1;
        xp->x_ccount = 0;
        xp->x_lcount = 0;
```

```
        xp->x_iptr = xp = ip;
        rp->i_flag =| ITEXT;
        rp->i_count++;
        ts = ((u.u_exdata.ux_tsize+63)>>6) & 01777;
        xp->x_size = ts;
        if((xp->x_daddr = malloc(swapmap, ctod(ts))) == NULL)
                panic(ooswap);
        u.u_procp->p_textp = xp;
        xexpand(xp);
        estabur(ts, 0, 0, 0, RW);
        u.u_count = u.u_exdata.ux_tsize;
        u.u_offset = 020;
        u.u_base = 0;
        u.u_segflg = 2;
        u.u_procp->p_flag =| SLOCK;
        VTPROCENT(u.u_procp,PR_FLAG);
        readi(rp);
        u.u_procp->p_flag =& ~SLOCK;
        VTPROCENT(u.u_procp,PR_FLAG);
        u.u_segflg = 0;
        xp->x_flag = XWRIT;
}

/*
 * Assure core for text segment
 * Text must be locked to keep someone else from
 * freeing it in the meantime.
 * x_ccount must be 0.
 */
xexpand(axp)
struct text *axp;
{
        register struct text *xp;

        xp = axp;
        if ((xp->x_caddr = malloc(coremap, xp->x_size)) != NULL) {
                if ((xp->x_flag&XLOAD)==0)
                        swap(xp->x_daddr, xp->x_caddr, xp->x_size, B_READ);
                xp->x_ccount++;
                xunlock(xp);
                return;
        }
        savu(u.u_rsav);
        savu(u.u_ssav);
        xswap(u.u_procp, 1, 0);
        xunlock(xp);
        u.u_procp->p_flag =| SSWAP;
        qswtch();
        /* no return */
}

/*
 * Lock and unlock a text segment from swapping
 */
xlock(axp)
{
```

```
        register struct text *xp;

        xp = axp;
        while(xp->x_flag&XLOCK) {
                xp->x_flag =| XWANT;
                sleep(xp, PSWP);
        }
        xp->x_flag =| XLOCK;
}

xunlock(axp)
{
        register struct text *xp;

        xp = axp;
        if (xp->x_flag&XWANT)
                wakeup(xp);
        xp->x_flag =& ~(XLOCK|XWANT);
}

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

        if ((xp = axp)==NULL || xp->x_ccount==0)
                return;
        xlock(xp);
        if (--xp->x_ccount==0) {
                if (xp->x_flag&XWRIT) {
                        xp->x_flag =& ~XWRIT;
                        swap(xp->x_daddr,xp->x_caddr,xp->x_size,B_WRITE);
                }
                mfree(coremap, xp->x_size, xp->x_caddr);
        }
        xunlock(xp);
}
xswapin(axp)
struct text *axp;
{
        register struct text *xp;
        register x;

        xp = axp;
        xlock(xp);
        if (xp->x_ccount==0) {
                if ((x = malloc(coremap, xp->x_size)) == NULL) {
                        xunlock(xp);
                        return(0);
                }
                xp->x_caddr = x;
                if ((xp->x_flag&XLOAD)==0)
```

```
                        swap(xp->x_daddr,x,xp->x_size,B_READ);
        }
        xp->x_ccount++;
        xunlock(xp);
}
```

```
/*      @(#)trap.c      2.6     */

#
#include "sys/param.h"
#include "sys/systm.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/proc.h"
#include "sys/procx.h"
#include "sys/reg.h"
#include "sys/seg.h"
#include "sys/vtmn.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 */
#define MEMORY  0177740         /* 11/70 "memory" subsystem */
#define NSYSENT 64              /* number of syscall entries */

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

/*
 * Dummy entry for illegal system calls
 */

struct sysent badent[1];

/*
 * 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
         * 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;
                if((i=fuiword(pc-2)&0377) < NSYSENT)
                        callp = &sysent[i];
                else
                        callp = badent;
                if (u.u_indflg = (callp == sysent)) { /* indirect */
                        a = fuiword(pc);
                        pc =+ 2;
                        i = fuword(a);
                        if((i&(~0377)) != SYS || (i =& 0377) >= NSYSENT)
                                callp = badent;                 /* illegal */
                        else
                                callp = &sysent[i];
                        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;

		/*
		 * The code here is a half-hearted
		 * attempt to do something with all
		 * of the 11/70 parity registers.
		 * In fact, there is little that
		 * can be done.
		 */
		case 10:
		case 10+USER:
			printf("parity\n");
			if(cputype == 70) {
				logparity(MEMORY);
				for(i=0; i<4; i++)
					printf("%o ", MEMORY->r[i]);
				printf("\n");
				MEMORY->r[2] = -1;
				if(dev & USER) {
					i = SIGBUS;
					break;
				}
			}
			panic("parity");

		}
		psignal(u.u_procp, i);

out:
		curpri = setpri(u.u_procp);
		if(curpri > nxtpri)
			qswtch();
		if(issig())
			psig();
}

/*
 * 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 proces[illegible], an (abnormal) return is simulated from
 * the last callee 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;
}
/*
 * Catch stray interrupts by using trace feature
 */
stray(junk, sp, r1, addr, r0, pc, ps)
{
        addr -= 2;
        logstray(addr);
        printf("stray interrupt at %o\n", addr);
}

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

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

```
/*      @(#)utssys.c    2.1     */

#include "sys/param.h"
#include "sys/user.h"
#include "sys/userx.h"
#include "sys/reg.h"
#include "sys/utsname.h"

utssys()
{

        switch(u.u_arg[0]) {
                case 0:
                        if (copyout(&utsname, u.u_ar0[R0], sizeof(utsname)))
                                u.u_error = EFAULT;
                        break;

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