COP 5611: Operating Systems Design Principles

Presentation by:

Fahd Rafi

Saad Ali

Software Interrupts
Chapter 13

Software Interrupts

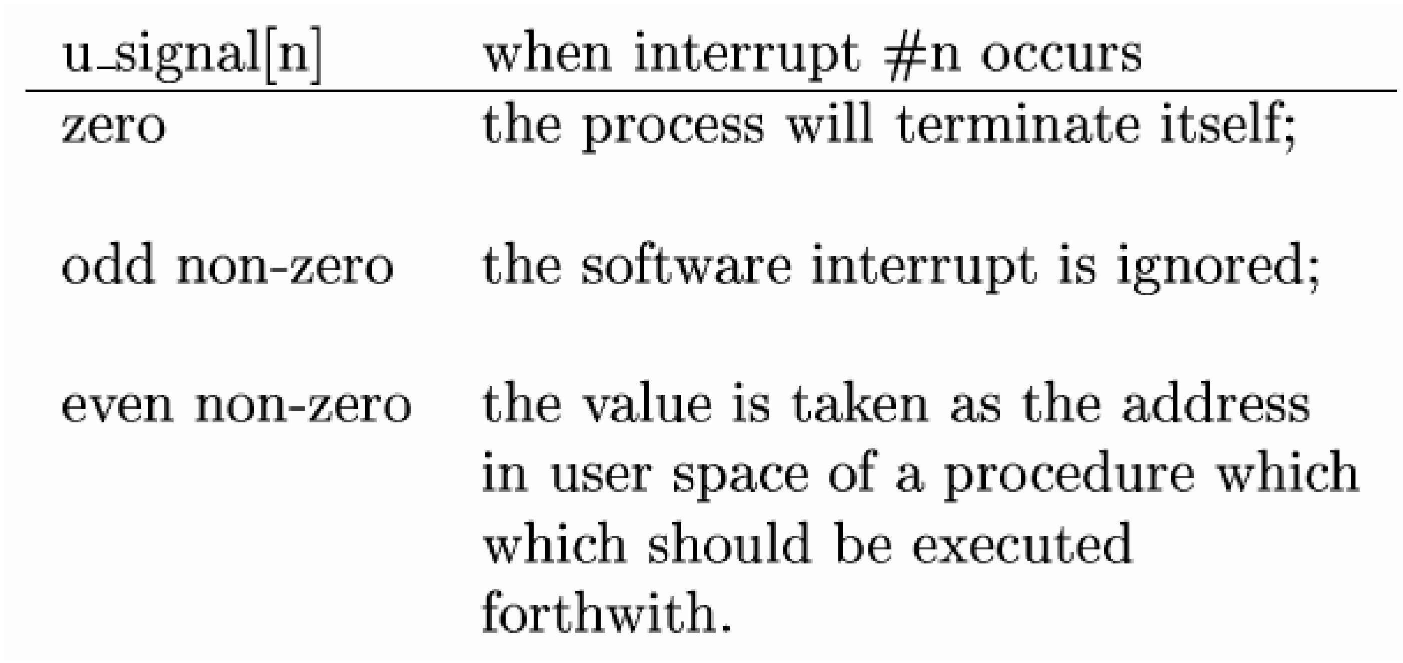
- Method to interrupt user mode operation by other processes or due to error
- Software Interrupt – Signal
- 20 software interrupts in UNIX
- 0 is no interrupt
- u.u_signal[n] specifies action on interrupt n

u.u_signal[n]

- Structure u lies in user.h
- It is the per process data area which is swapped out along with process
- Always contains data for the currently running process

u.u_signal[n]

- Operation to be performed on signal



u_signal[n]	when interrupt #n occurs
zero	the process will terminate itself;
odd non-zero	the software interrupt is ignored;
even non-zero	the value is taken as the address in user space of a procedure which which should be executed forthwith.

SIGKIL

- n=9
- Distinguished from other interrupts and process always terminates on SIGKIL
- Supposed to remain ‘0’ until the end of process

User Setup

- User can set up the action to be taken for any signal using the ‘signal()’ system call.

  signal(2,1) // sets u_signal[2]=1;

  (meaning it will be ignored due to odd number)

- u_signal[SIGKIL] cannot be modified

Causing Interrupt

- Set “p_sig” in process “proc” entry to interrupt number;
  - For example: p->p_sig=SIGINT;
- Since only one p_sig is provided, only one and most recent signal can be maintained.

Handling Interrupt

- The interrupt is always handled when the target process becomes active
  - Interrupts must wait till process becomes active
- If user-mode action is to be performed, the user mode stack is used

Tracing

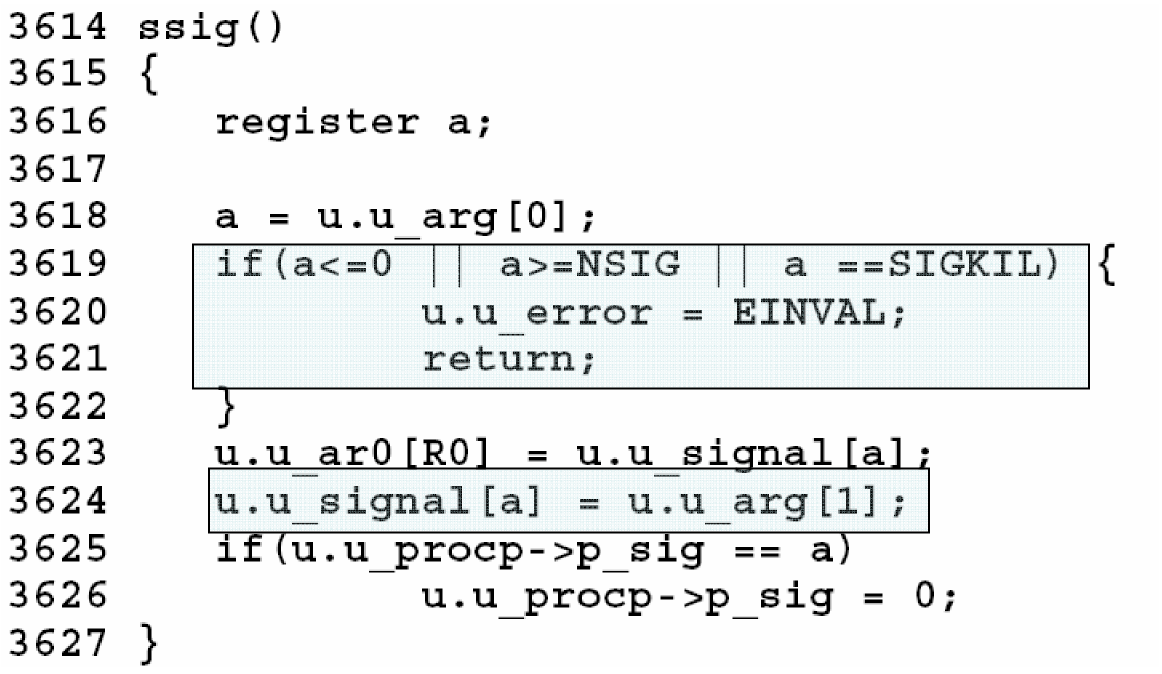
- Tracing is implemented using software interrupts.
  - SIGTRC
- Parent can monitor the progress of a child process

Implementation

- Specify signal action:
  - ssig() – Specify action for signal
- Send signal:
  - kill() – Send signal to some process
- Other functions:
  - psignal() – Send signal to a process
  - signal() – Send signal to all processes from a terminal
  - issig() – To check if there is an outstanding interrupt
  - psig() – To implement action when issig returns true
  - core() – When core dump is indicated for a terminating process
  - grow() – To grow stack size when needed
  - exit() – Terminates the currently active process
  - ptrace() – Implements ptrace system call
  - stop() – To stop a process for debugging
  - procxmt() – Child carries out certain operations for parent when stopped

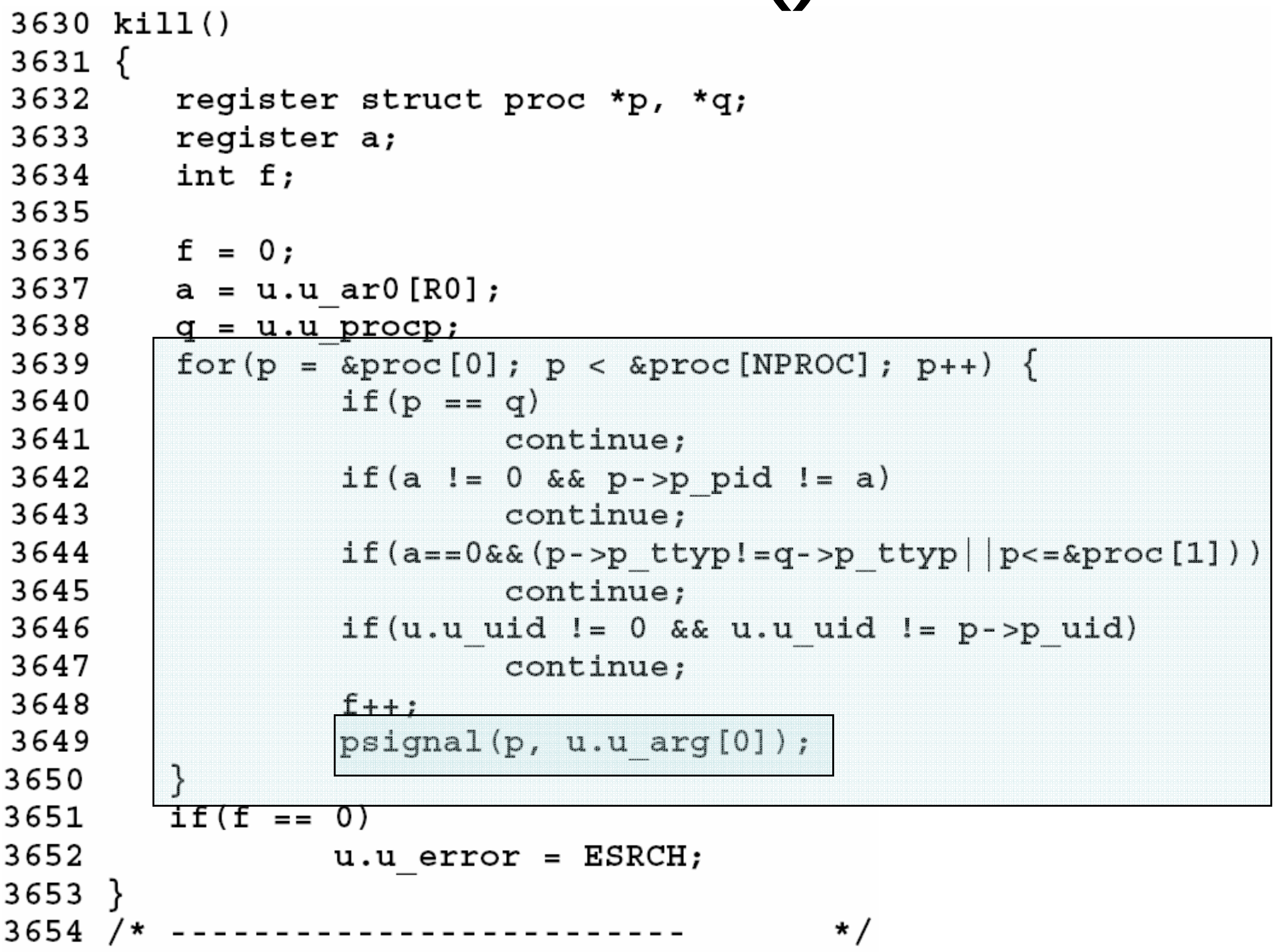
Code

ssig()



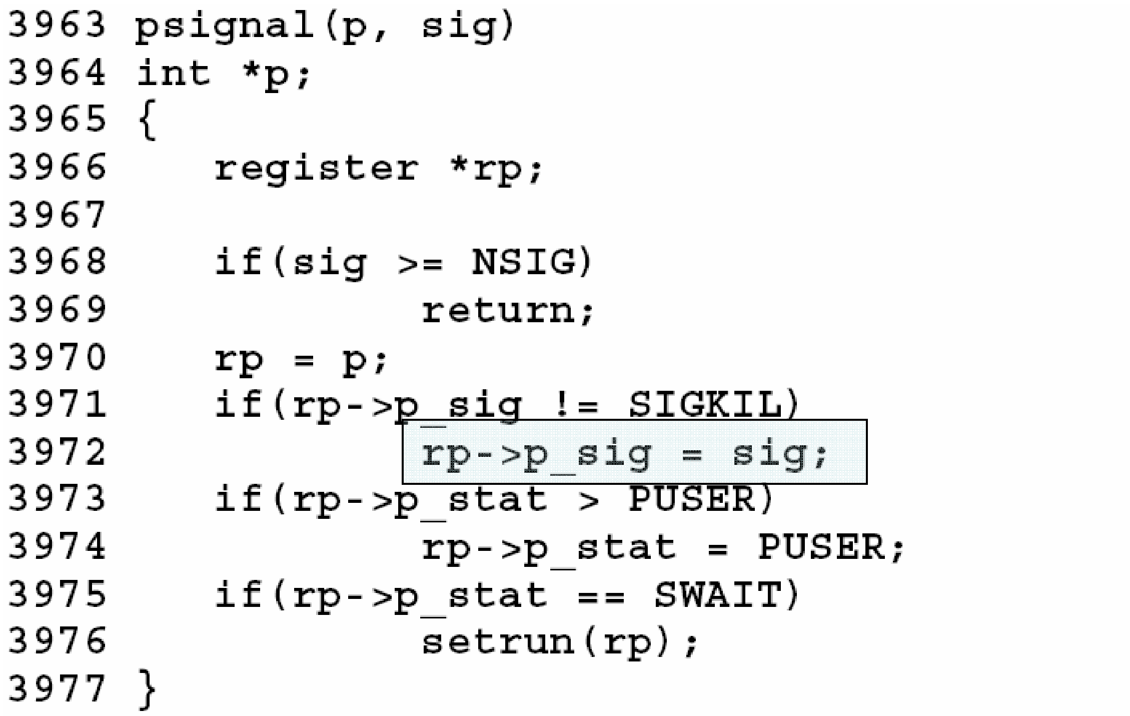
```
ssig()
{
   register a;

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

kill()



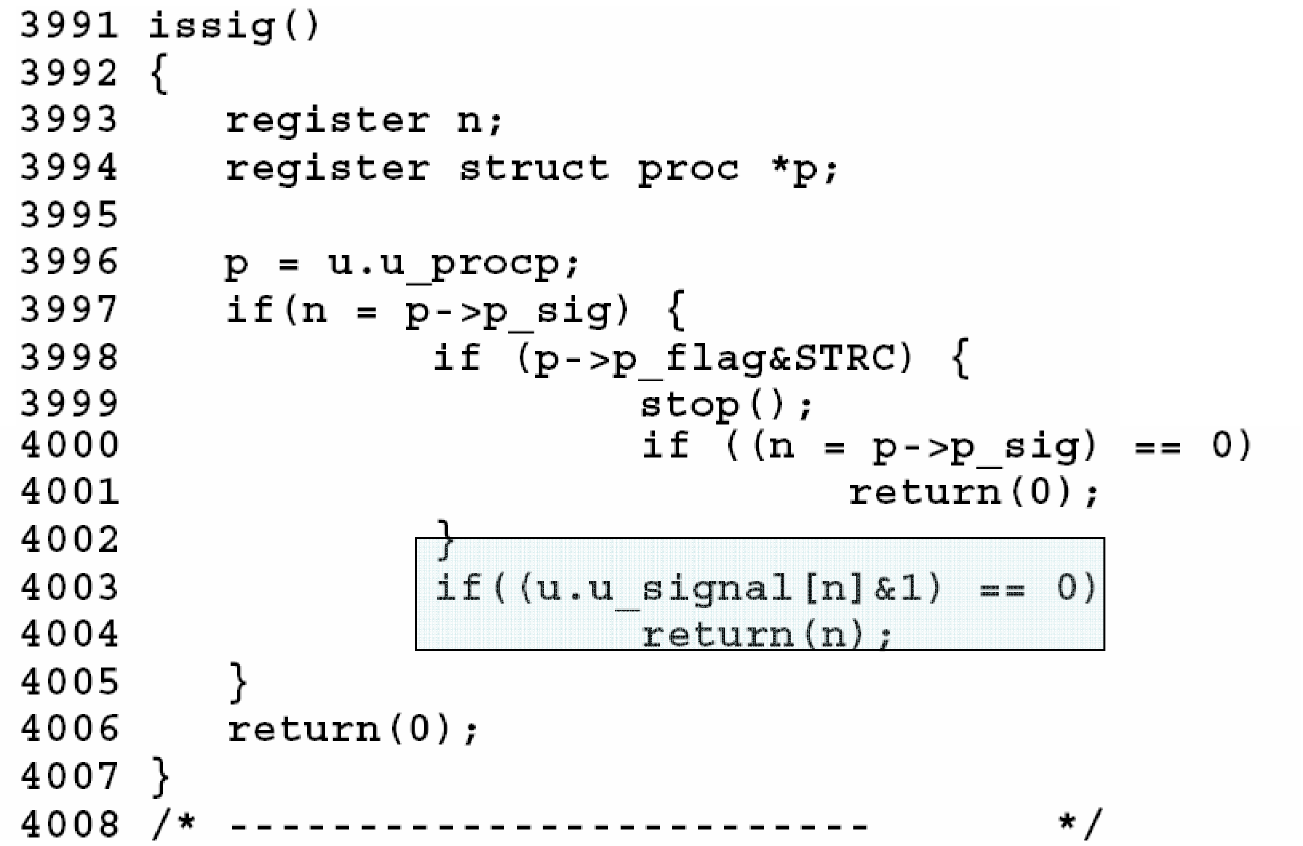
```
kill()
{
        register struct proc *p, *q;
        register a;
        int f;

        f = 0;
        a = u.u_ar0[R0];
        q = u.u_procp;
        for(p = &proc[0]; p < &proc[NPROC]; p++) {
                if(p == q)
                        continue;
                if(a != 0 && p->p_pid != a)
                        continue;
                if(a==0&&(p->p_ttyp!=q->p_ttyp||p<=&proc[1]))
                        continue;
                if(u.u_uid != 0 && u.u_uid != p->p_uid)
                        continue;
                f++;
                psignal(p, u.u_arg[0]);
        }
        if(f == 0)
                u.u_error = ESRCH;
}
/* -------------------------       */
```

psignal()



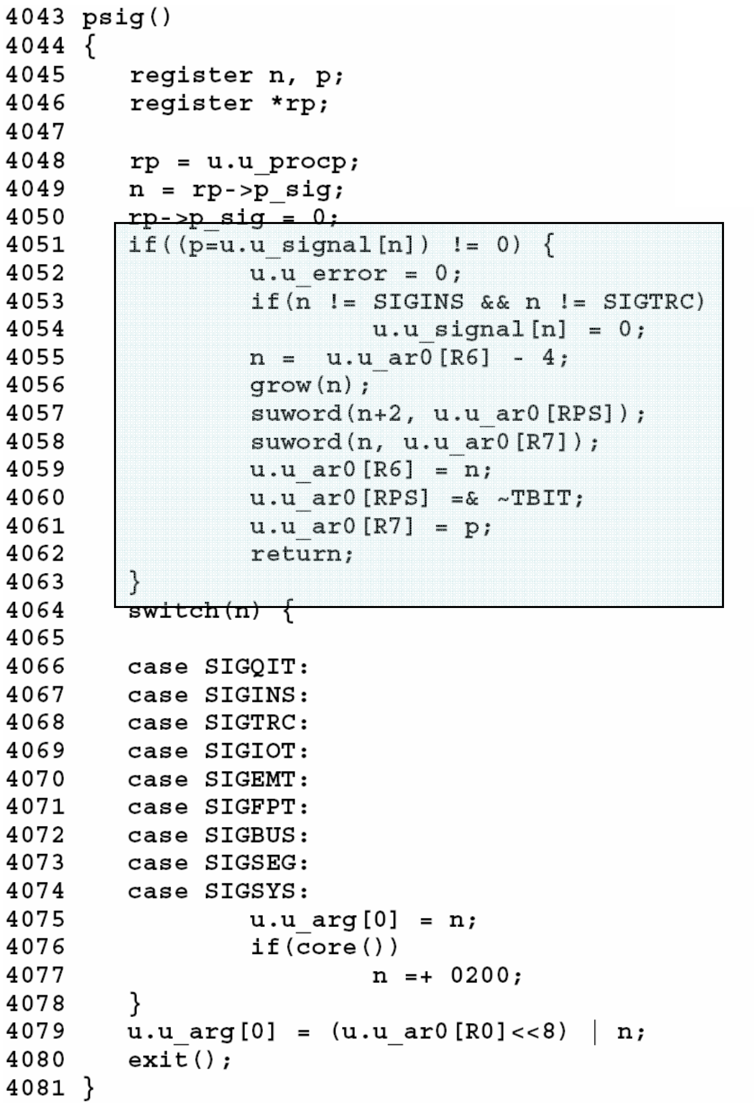
```
psignal(p, sig)
int *p;
{
	register *rp;

	if(sig >= NSIG)
		return;
	rp = p;
	if(rp->p_sig != SIGKIL)
		rp->p_sig = sig;
	if(rp->p_stat > PUSER)
		rp->p_stat = PUSER;
	if(rp->p_stat == SWAIT)
		setrun(rp);
}
```

issig()



```
issig()
{
        register n;
        register struct proc *p;

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

psig ()



```
psig()
{
    register n, p;
    register *rp;

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

    case SIGQIT:
    case SIGINS:
    case SIGTRC:
    case SIGIOT:
    case SIGEMT:
    case SIGFPT:
    case SIGBUS:
    case SIGSEG:
    case SIGSYS:
            u.u_arg[0] = n;
            if(core())
                    n =+ 0200;
    }
    u.u_arg[0] = (u.u_ar0[R0]<<8) | n;
    exit();
}
```

Pipes – Chapter 21

“Pipe.c”

Pipes

- Used for creating Pipes
  - Pipe is a FIFO character list
  - One group of processes write other read
  - Intercommunication

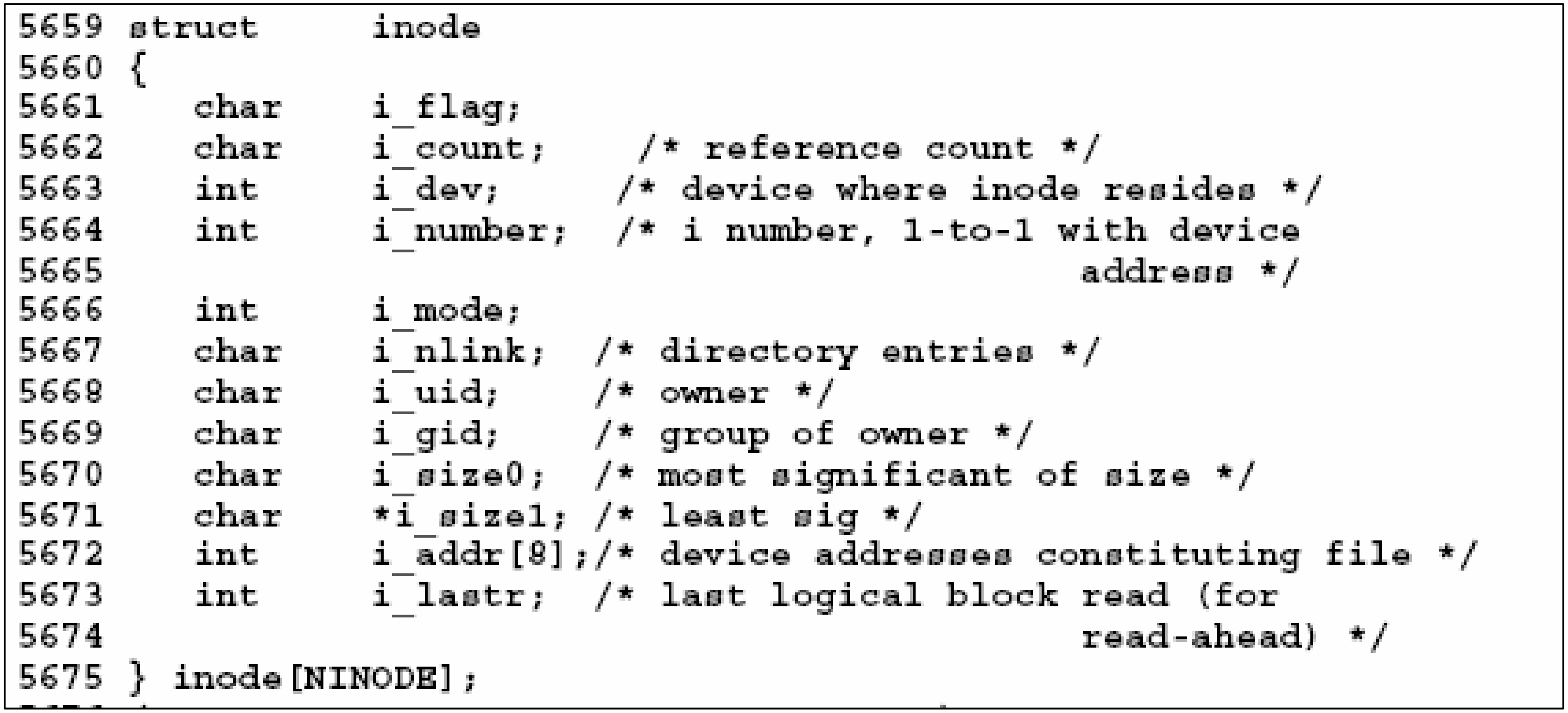
Pipe.c

- Global Variable
  - PIPSIZ (4096)

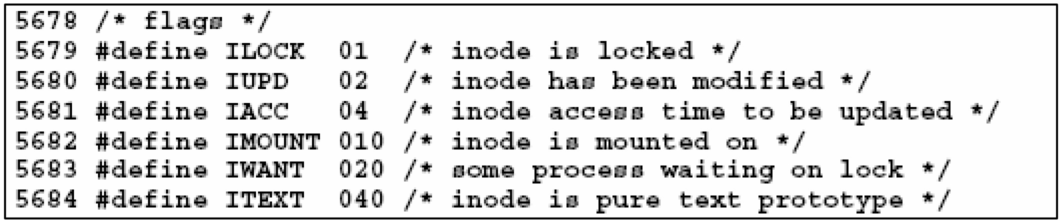
- Functions
  - pipe()
  - readp()
  - writep()
  - plock()
  - prele()

Structures

INODE – Focus of all file activities – Unique inode for each file



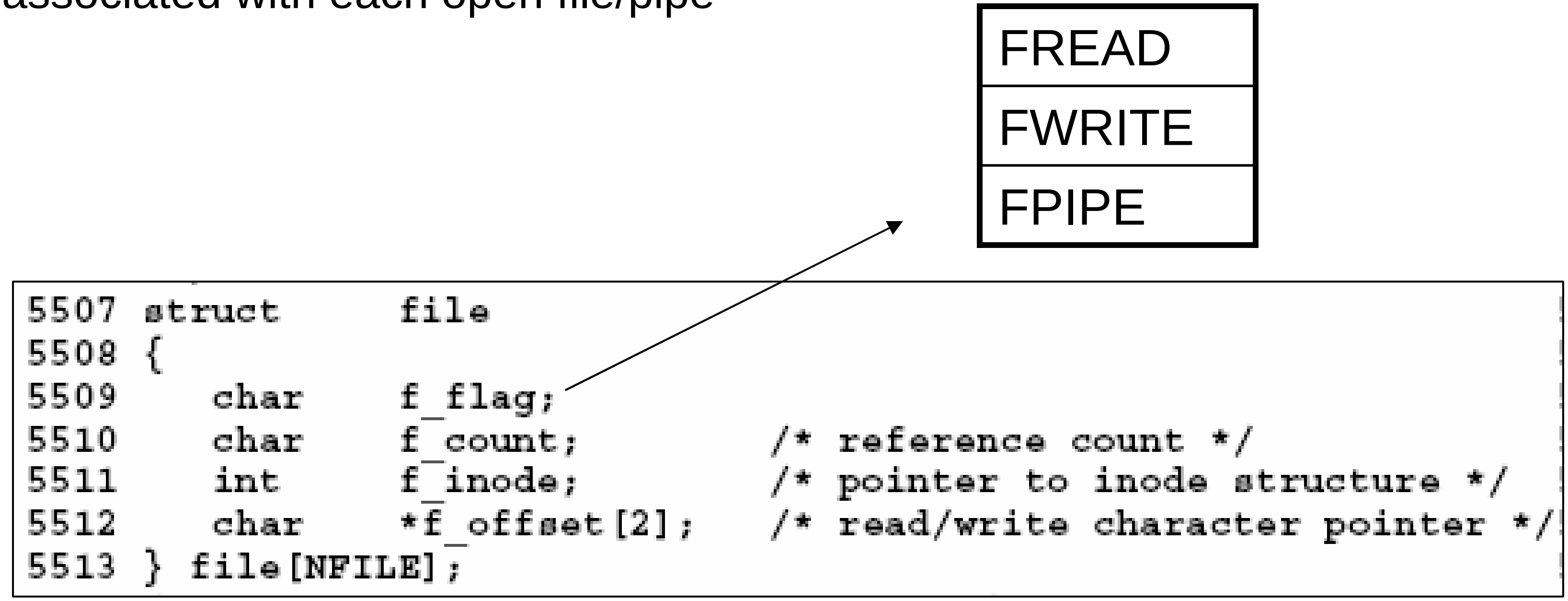
```
struct     inode
{
   char    i_flag;
   char    i_count;    /* reference count */
   int     i_dev;     /* device where inode resides */
   int     i_number;  /* i number, 1-to-1 with device
                                              address */
   int     i_mode;
   char    i_nlink;  /* directory entries */
   char    i_uid;    /* owner */
   char    i_gid;    /* group of owner */
   char    i_size0;  /* most significant of size */
   char    *i_size1; /* least sig */
   int     i_addr[8];/* device addresses constituting file */
   int     i_lastr;  /* last logical block read (for
                                              read-ahead) */
} inode[NINODE];
```



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

Structure .. File

- One file structure is allocated for each pipe call. It holds read write pointers associated with each open file/pipe



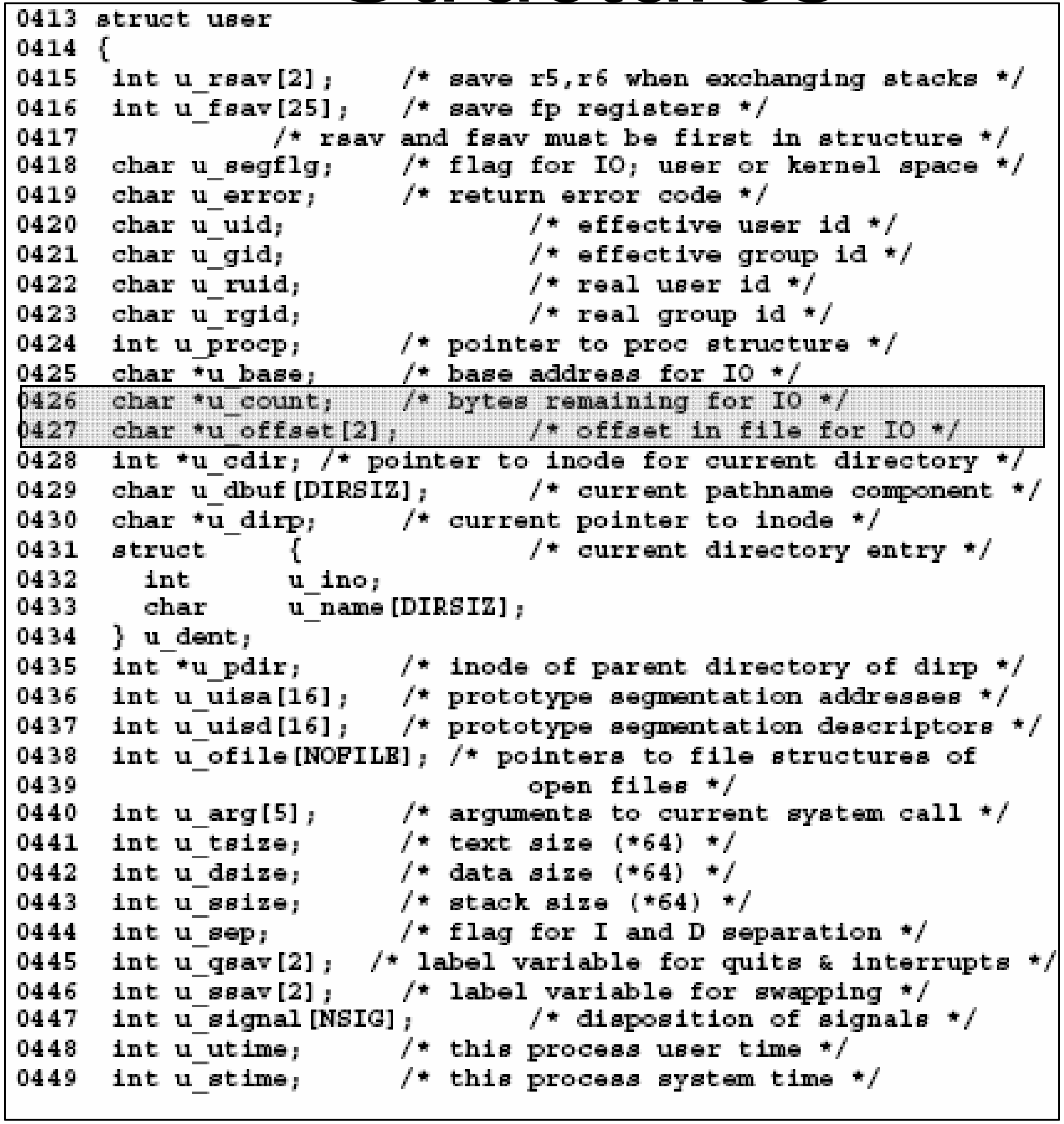
FREAD

FWRITE

FPIPE

```
struct      file
{
    char    f_flag;
    char    f_count;        /* reference count */
    int     f_inode;        /* pointer to inode structure */
    char    *f_offset[2];   /* read/write character pointer */
} file[NFILE];
```

Structures

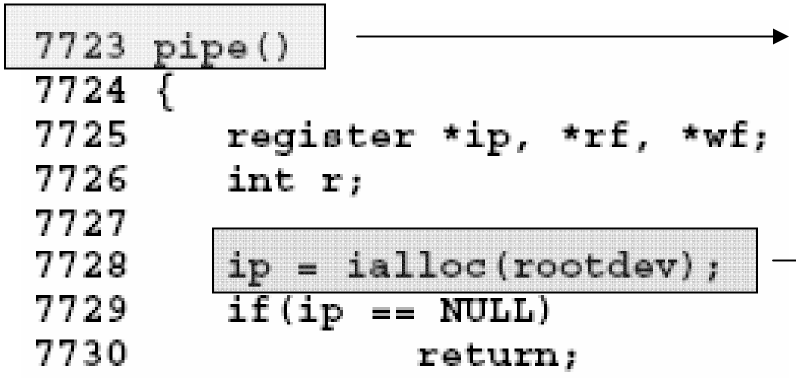


```
struct user
{
 int u_rsav[2];     /* save r5,r6 when exchanging stacks */
 int u_fsav[25];    /* save fp registers */
            /* rsav and fsav must be first in structure */
 char u_segflg;     /* flag for IO; user or kernel space */
 char u_error;      /* return error code */
 char u_uid;   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

Pipe System Call

- Allocate an inode for the root device
- Allocate a file table entry
- Remember file table entry in ‘r’ and allocate another file table entry
- Return user file identification in R0 and R1
- Complete the entries in ‘file’ and ‘inode’ structure.

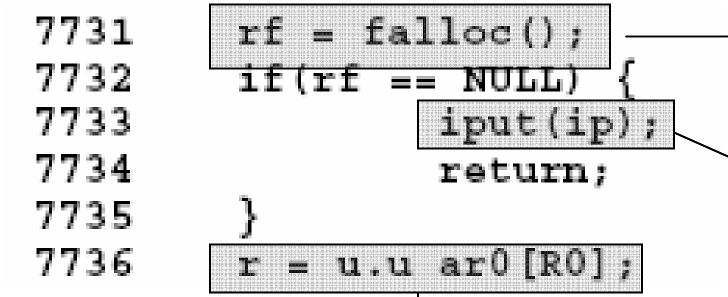
Pipe - Code



```
pipe()
{
    register *ip, *rf, *wf;
    int r;

    ip = ialloc(rootdev);
    if(ip == NULL)
            return;
```

System Call

Allocate an unused ‘inode’ on ‘rootdev’

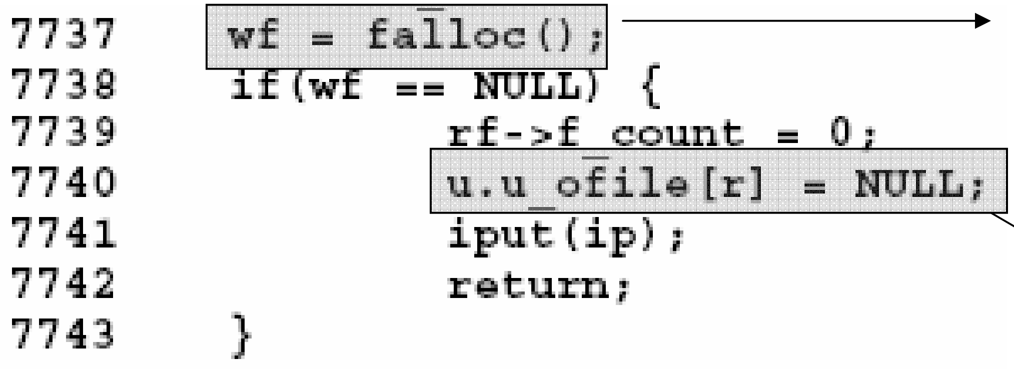


```
    rf = falloc();
    if(rf == NULL) {
            iput(ip);
            return;
    }
    r = u.u ar0[R0];
```

Allocates a user file descriptor and file structure. File descriptor is copied into user register R0

Decrements the reference count of an ‘inode’ structure

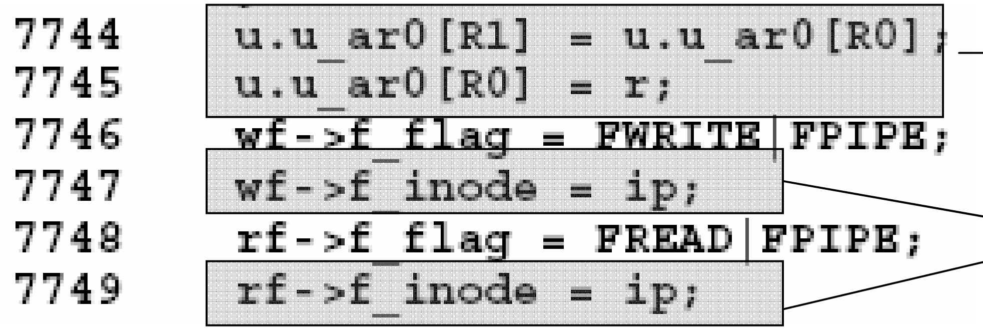
Save the new file descriptor



```
    wf = falloc();
    if(wf == NULL) {
            rf->f_count = 0;
            u.u_ofile[r] = NULL;
            iput(ip);
            return;
    }
```

Allocates a user file descriptor and file structure. File descriptor is again copied into user register R0

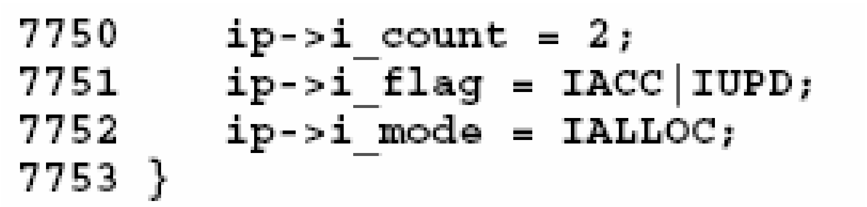
Set pointer to file structure of read open file to NULL



```
    u.u_ar0[R1] = u.u_ar0[R0];
    u.u_ar0[R0] = r;
    wf->f_flag = FWRITE|FPIPE;
    wf->f_inode = ip;
    rf->f_flag = FREAD|FPIPE;
    rf->f_inode = ip;
```

Register R1 = Write File Descriptor
Register R0 = Read File Descriptor

Make inode pointer of both structures equal to same inode

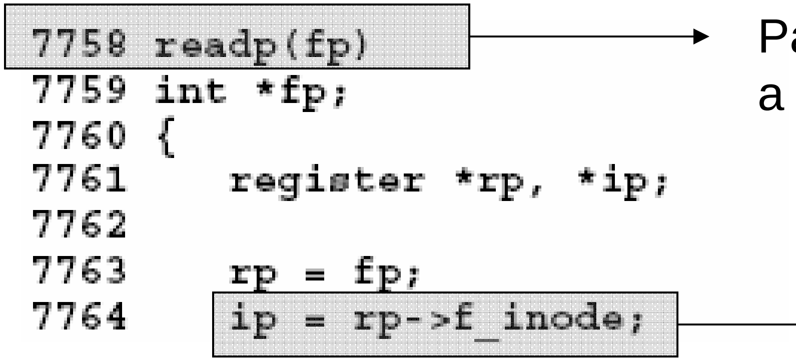


```
    ip->i_count = 2;
    ip->i_flag = IACC|IUPD;
    ip->i_mode = IALLOC;
}
```

Function – readp

- Two offsets are required:
    - For read
    - For write (write offset = filesize)
- Pass a file pointer to readp → Extract inode pointer from the file structure
- Lock the pipe
- Check if both reader and writer side of pipe is active: If not error
- Read and unlock the pipe

Readp - Code

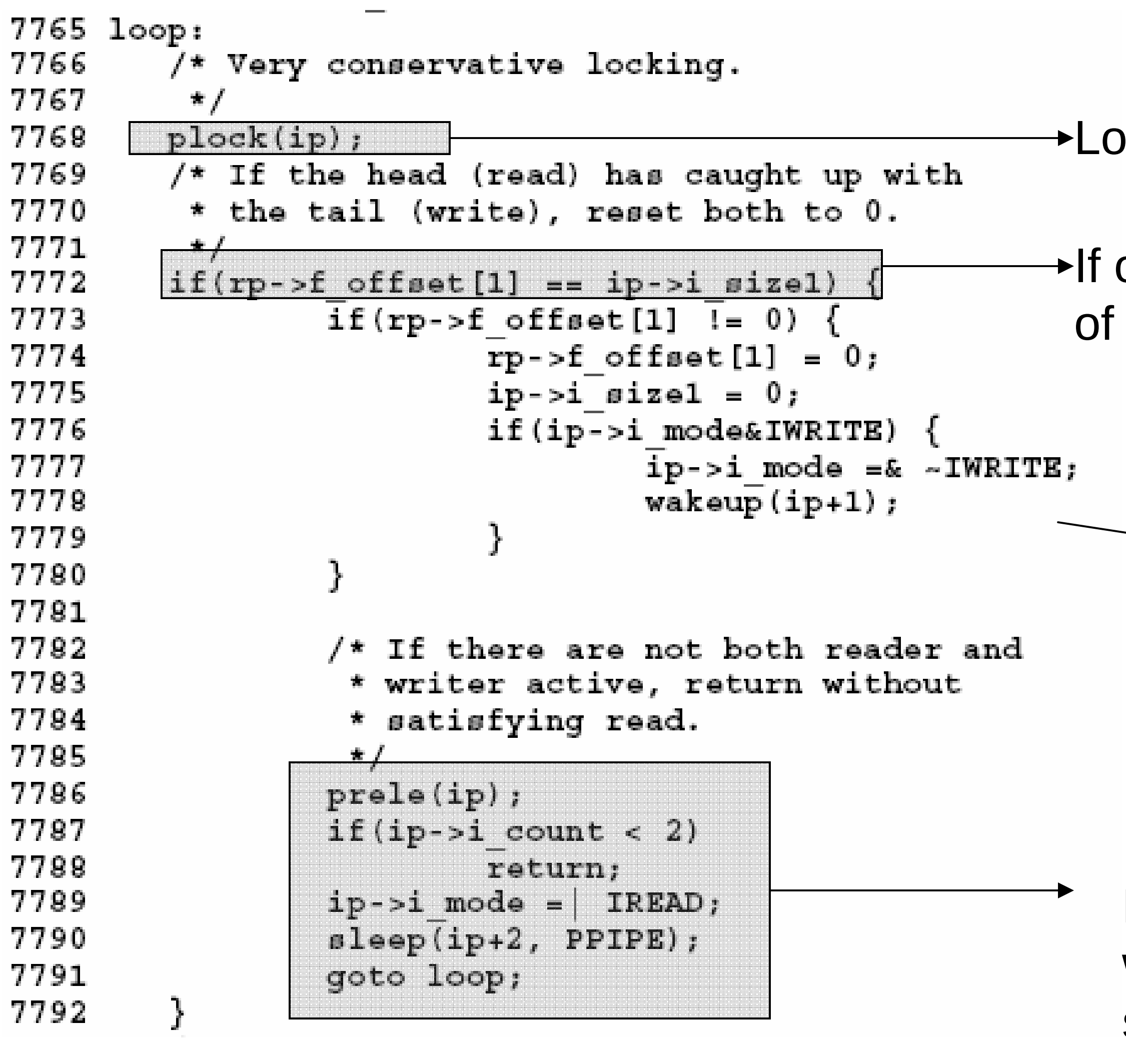


```
readp(fp)
int *fp;
{
    register *rp, *ip;

    rp = fp;
    ip = rp->f_inode;
```

Pass a pointer of file structure from which has a pointer to inode of the pipe

Extract inode pointer

Readp – Code .. Cont’d



```
loop:
   /* Very conservative locking.
    */
   plock(ip);
   /* If the head (read) has caught up with
    * the tail (write), reset both to 0.
    */
   if(rp->f_offset[1] == ip->i_size1) {
           if(rp->f_offset[1] != 0) {
                   rp->f_offset[1] = 0;
                   ip->i_size1 = 0;
                   if(ip->i_mode&IWRITE) {
                           ip->i_mode =& ~IWRITE;
                           wakeup(ip+1);
                   }
           }

           /* If there are not both reader and
            * writer active, return without
            * satisfying read.
            */
           prele(ip);
           if(ip->i_count < 2)
                   return;
           ip->i_mode =| IREAD;
           sleep(ip+2, PPIPE);
           goto loop;
   }
```

Lock the inode

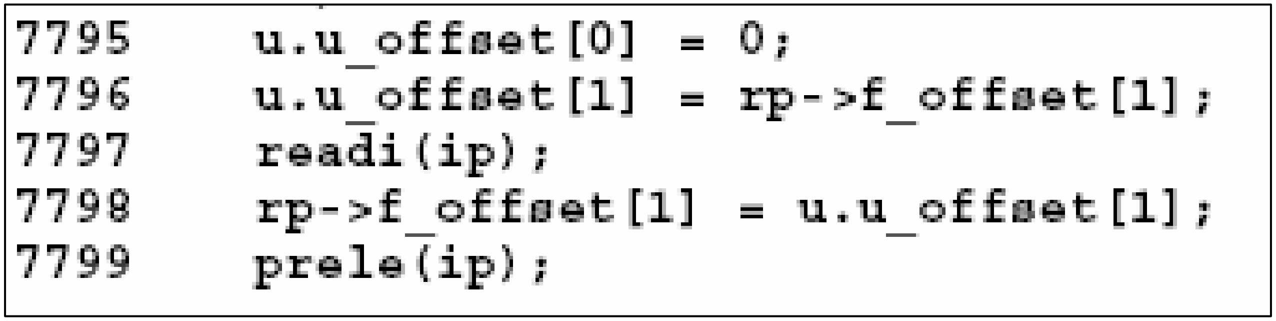
If offset becomes equal to size of the inode than reset

Wake up blocked writer

Raise the flag that I want to read and go to sleep

Readp – Code .. Cont’d

If every thing is fine than read and return:

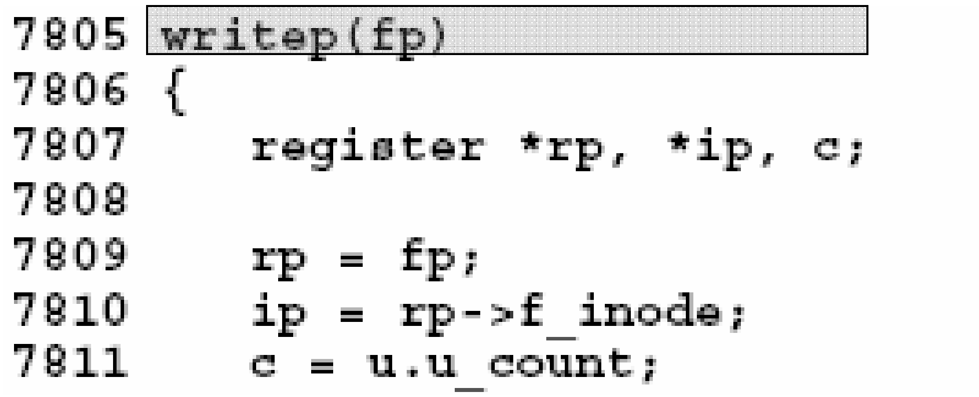


```
u.u_offset[0] = 0;
u.u_offset[1] = rp->f_offset[1];
readi(ip);
rp->f_offset[1] = u.u_offset[1];
prele(ip);
```

Function – writep()

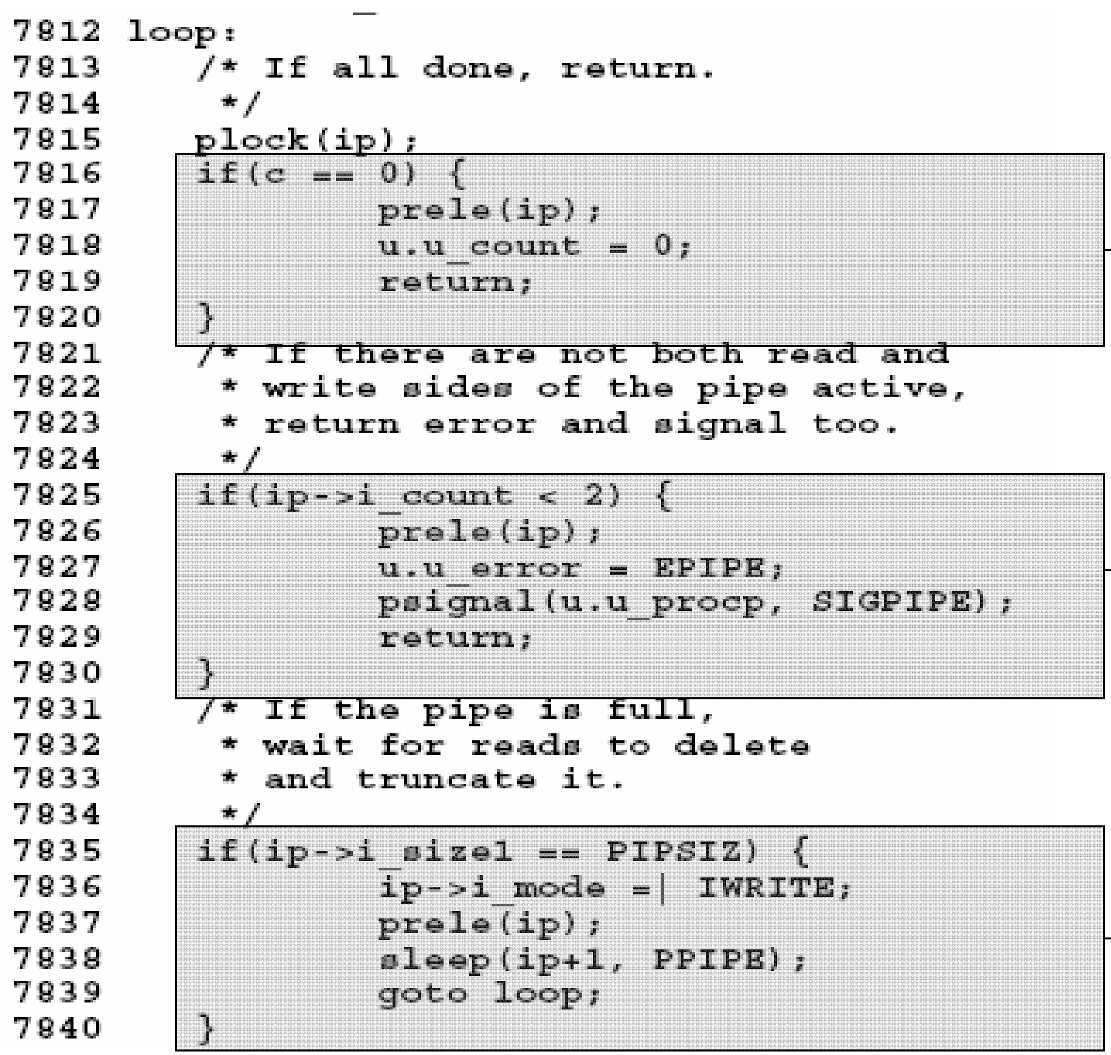
- Lock the pipe
- Check if both reader and writer side of pipe is active: If not error
- If pipe is full wait for reader to consume characters
- Write desired number of bytes

Writep - Code



```
writep(fp)
{
    register *rp, *ip, c;

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

Writep - Code



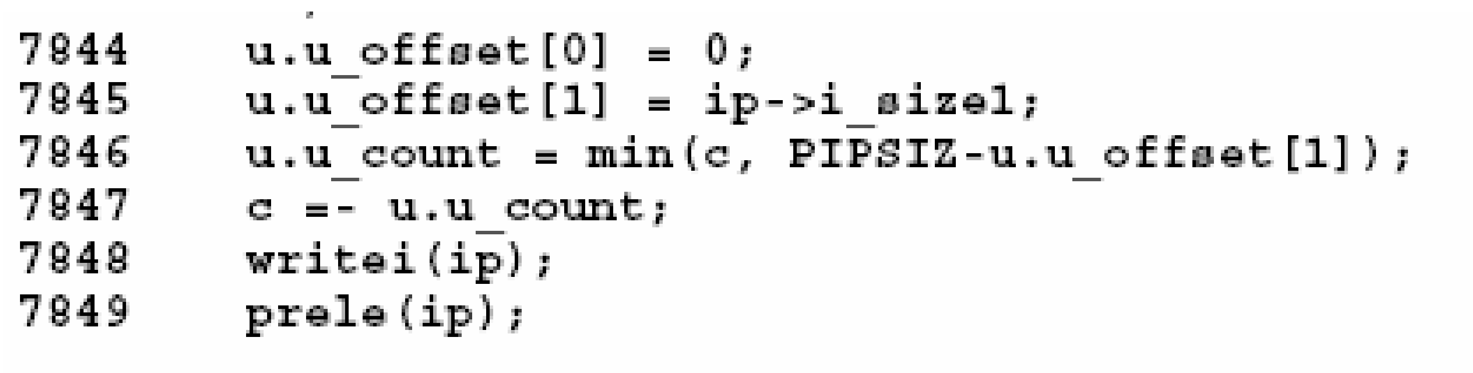
```
loop:
    /* If all done, return.
     */
    plock(ip);
    if(c == 0) {
            prele(ip);
            u.u_count = 0;
            return;
    }
    /* If there are not both read and
     * write sides of the pipe active,
     * return error and signal too.
     */
    if(ip->i_count < 2) {
            prele(ip);
            u.u_error = EPIPE;
            psignal(u.u_procp, SIGPIPE);
            return;
    }
    /* If the pipe is full,
     * wait for reads to delete
     * and truncate it.
     */
    if(ip->i_size1 == PIPSIZ) {
            ip->i_mode =| IWRITE;
            prele(ip);
            sleep(ip+1, PPIPE);
            goto loop;
    }
```

No more bytes to write - return

Receive the signal that there are no more readers

Size reaches default size – no more writes can be done

Writep - Code

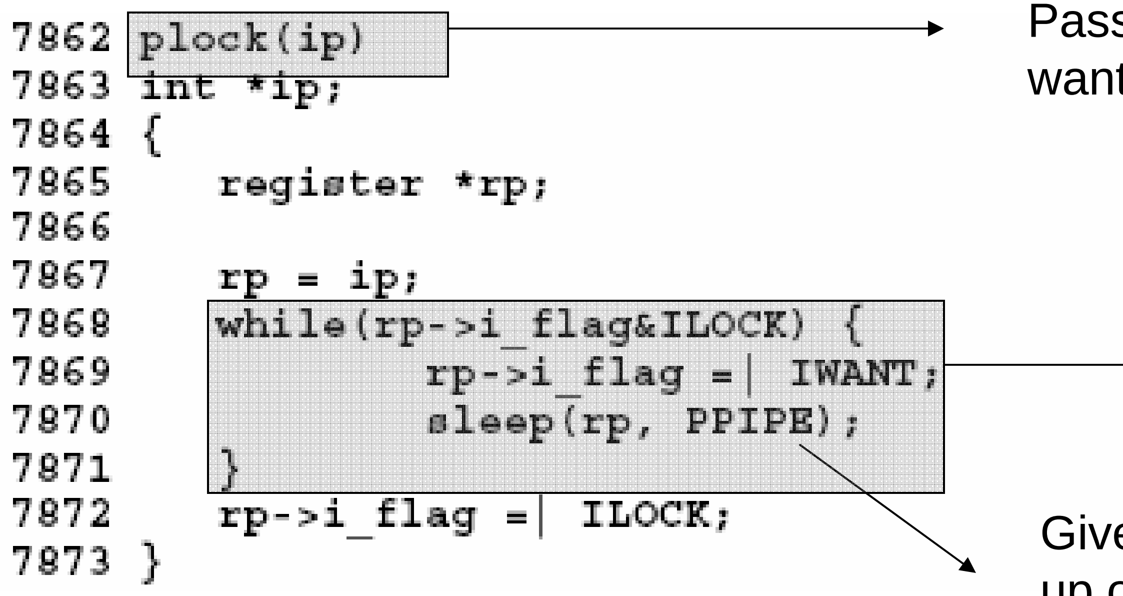


```
u.u_offset[0] = 0;
u.u_offset[1] = ip->i_size1;
u.u_count = min(c, PIPSIZ-u.u_offset[1]);
c =- u.u_count;
writei(ip);
prele(ip);
```

Function – plock()

- Locks a pipe before writing or reading
- If already locked:
  - Set the want bit
  - Sleep
- Otherwise:
  - Set the lock flag

Plock - Code

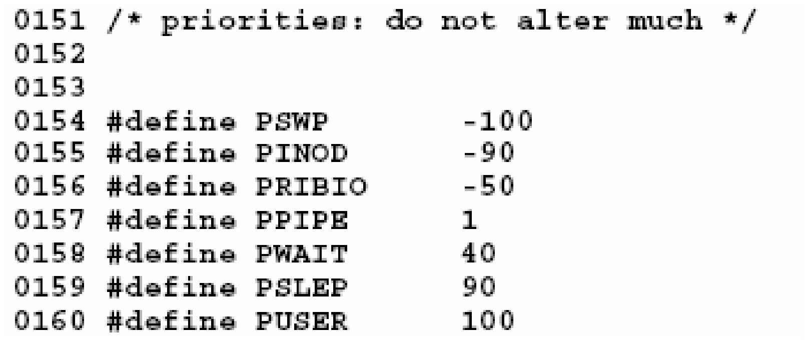


```
plock(ip)
int *ip;
{
    register *rp;

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

Pass pointer of inode that we want to lock

Set the IWANT bit

Give up the processor till a wake up occurs on ip, at which the process enters the scheduling queue at priority PIPE.



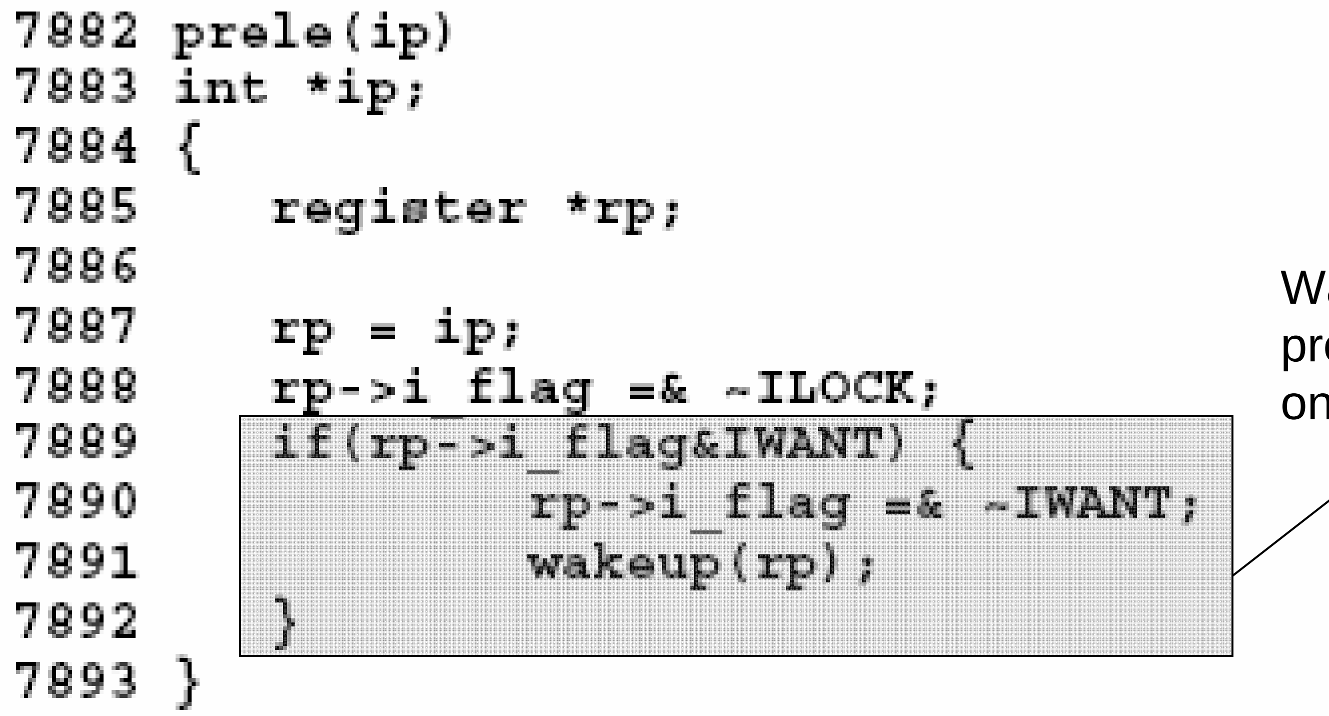
```
/* priorities: do not alter much */


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

Function – prele()

- Unlocks the pipe after writing or reading

- If WANT bit is on:
    - Wakeup

Prele - Code



```
prele(ip)
int *ip;
{
	register *rp;

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

Wake up all processes waiting on this inode

End