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

<table>
<thead>
<tr>
<th>u_signal[n]</th>
<th>when interrupt #n occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>zero</td>
<td>the process will terminate itself;</td>
</tr>
<tr>
<td>odd non-zero</td>
<td>the software interrupt is ignored;</td>
</tr>
<tr>
<td>even non-zero</td>
<td>the value is taken as the address in user space of a procedure which should be executed</td>
</tr>
<tr>
<td></td>
<td>forthwith.</td>
</tr>
</tbody>
</table>
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.
  
  ```c
  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()

3614 ssig()
3615 {
3616     register a;
3617
3618     a = u.u_arg[0];
3619     if(a<=0 || a>=NSIG || a ==SIGKIL) {
3620         u.u_error = EINVAL;
3621         return;
3622     }
3623     u.u_ar0[RO] = u.u_signal[a];
3624     u.u_signal[a] = u.u_arg[1];
3625     if(u.u_procp->p_sig == a)
3626         u.u_procp->p_sig = 0;
3627 }
```c
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<char>&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()

3963 psignal(p, sig)
3964 int *p;
3965 {
3966     register *rp;
3967
3968     if(sig >= NSIG)
3969         return;
3970     rp = p;
3971     if(rp->p_sig != SIGKIL)
3972         rp->p_sig = sig;
3973     if(rp->p_stat > PUSER)
3974         rp->p_stat = PUSER;
3975     if(rp->p_stat == SWAIT)
3976         setrun(rp);
issig()

3991 issig()
3992 {
3993     register n;
3994     register struct proc *p;
3995
3996     p = u.u_procp;
3997     if(n = p->p_sig) {
3998         if (p->p_flag&STRC) {
3999             stop();
4000             if ((n = p->p_sig) == 0)
4001                 return(0);
4002         }
4003     }
4004     if((u.u_signal[n]&1) == 0)
4005         return(n);
4006     return(0);
4007 }
4008 /*--------------------------*/
psig()

```c
4043  psig()
4044  {
4045      register n, p;
4046      register *rp;
4047
4048      rp = u.u_procp;
4049      n = rp->p_sig;
4050      rp->p_sig = 0;
4051      if((p=u.u_signal[n]) != 0) {
4052          u.u_error = 0;
4053          if(n != SIGINS && n != SIGTRC)
4054              u.u_signal[n] = 0;
4055          n = u.u_ar0[R6] - 4;
4056          grow(n);
4057          suword(n+2, u.u_ar0[RPS]);
4058          suword(n, u.u_ar0[R7]);
4059          u.u_ar0[R6] = n;
4060          u.u_ar0[RPS] = &-TBIT;
4061          u.u_ar0[R7] = p;
4062          return;
4063      }
4064      switch(n) {
4065
4066          case SIGQUIT:
4067          case SIGINS:
4068          case SIGTRC:
4069          case SIGIOT:
4070          case SIGEMT:
4071          case SIGFPT:
4072          case SIGBUS:
4073          case SIGSEGP:
4074          case SIGSYS:
4075              u.u_arg[0] = n;
4076              if(core())
4077                  n =+ 0200;
4078          }
4079      u.u_arg[0] = (u.u_ar0[R0]<<8) | n;
4080      exit();
4081  }
```
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

```c
5659 struct inode
5660 {
5661    char i_flag;
5662    char i_count;  /* reference count */
5663    int i_dev;     /* device where inode resides */
5664    int i_number; /* i number, 1-to-1 with device address */
5665    int i_mode;
5666    char i_inlink; /* directory entries */
5667    char i_uid;    /* owner */
5668    char i_gid;    /* group of owner */
5669    char i_size0;  /* most significant of size */
5670    char *i_size1; /* least sig */
5671    int i_addr[8]; /* device addresses constituting file */
5672    int i_lastr;   /* last logical block read (for read-ahead) */
5673    } inode[NINODE];
5674 */ flags */
5675 #define ILOCK 01 /* inode is locked */
5676 #define IUPD 02 /* inode has been modified */
5677 #define IAACC 04 /* inode access time to be updated */
5678 #define IMOUNT 010 /* inode is mounted on */
5679 #define IWANT 020 /* some process waiting on lock */
5680 #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

```
5507 struct file
5508 {
5509   char f_flag;
5510   char f_count;    /* reference count */
5511   int  f_inode;    /* pointer to inode structure */
5512   char *f_offset[2]; /* read/write character pointer */
5513 } file[NFILE];
```
Structures

0413 struct user
0414 {
0415  int r sav[2]; /* save r5,r6 when exchanging stacks */
0416  int f sav[25]; /* save fp registers */
0417  /* r sav and fsav must be first in structure */
0418  char segflg; /* flag for IO; user or kernel space */
0419  char u_error; /* return error code */
0420  char uid;     /* effective user id */
0421  char gid;    /* effective group id */
0422  char r uid;     /* real user id */
0423  char r gid;     /* real group id */
0424  int proc;    /* pointer to proc structure */
0425  char *b ase; /* base address for IO */
0426  char *count; /* bytes remaining for IO */
0427  char offset[2]; /* offset in file for IO */
0428  int *cdir; /* pointer to inode for current directory */
0429  chardbuf[DIRSZ]; /* current pathname component */
0430  char *dirc; /* current pointer to inode */
0431  struct {
0432    int u_ino;
0433    char u_name[DIRSZ];
0434  } dent;
0435  int *p dir; /* inode of parent directory of dirp */
0436  int ulsa[16]; /* prototype segmentation addresses */
0437  int uisd[16]; /* prototype segmentation descriptors */
0438  int ofile[NFILE]; /* pointers to file structures of
0439    open files */
0440  int arg[5]; /* arguments to current system call */
0441  int tsize; /* text size (*64) */
0442  int dsize; /* data size (*64) */
0443  int ssize; /* stack size (*64) */
0444  int s sep; /* flag for I and D separation */
0445  int gsav[2]; /* label variable for quits & interrupts */
0446  int ssav[2]; /* label variable for swapping */
0447  int signal[NSIG]; /* disposition of signals */
0448  int utime; /* this process user time */
0449  int 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

```c
7723    pipe()
7724    {
7725      register *ip, *rf, *wf;
7726      int r;
7727
7728      ip = ialloc(rootdev);
7729      if(ip == NULL)
7730        return;

7731    rf = falloc();
7732    if(rf == NULL) {
7733      iput(ip);
7734        return(ip);
7735    }
7736    r = u.u.ar0[R0];
```

- **System Call**
  - Allocate an unused ‘inode’ on ‘rootdev’
  - 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
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

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

Make inode pointer of both structures equal to same inode
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

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

Extract inode pointer
Readp – Code .. Cont’d

```c
7765 loop:
7766    /* Very conservative locking.
7767    */
7768    lock(ip);
7769    /* If the head (read) has caught up with
7770    * the tail (write), reset both to 0.
7771    */
7772    if(rp->f_offset[1] == ip->i_size)
7773       if(rp->f_offset[1] != 0) {
7774           rp->f_offset[1] = 0;
7775           ip->i_size = 0;
7776           if(ip->i_mode & IWRITE) {
7777               ip->i_mode &= ~IWRITE;
7778               wakeup(ip+1);
7779            }
7780        }
7781    /* If there are not both reader and
7782    * writer active, return without
7783    * satisfying read.
7784    */
7785    psem(ip);
7786    if(ip->i_count < 2)
7787       return;
7788    ip->i_mode |= IREAD;
7789    sleep(ip+2, PPIPE);
7790    goto loop;
7791  }
```

- 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
If every thing is fine than read and return:

7795 \quad u.u_{\text{offset}}[0] = 0;
7796 \quad u.u_{\text{offset}}[1] = rp->f_{\text{offset}}[1];
7797 \quad \text{readi}(ip);
7798 \quad rp->f_{\text{offset}}[1] = u.u_{\text{offset}}[1];
7799 \quad \text{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

```c
writep(fp)
{
    register *rp, *ip, c;
    rp = fp;
    ip = rp->f_inode;
    c = u.u_count;
```
Receive the signal that there are no more readers

Size reaches default size – no more writes can be done

No more bytes to write - return
7844    u.u_offset[0] = 0;
7845    u.u_offset[1] = ip->i_size;
7846    u.u_count = min(c, PIPSIZ-u.u_offset[1]);
7847    c -= u.u_count;
7848    writei(ip);
7849    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

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.

```
7862  plock(ip)
7863  int *ip;
7864  {
7865      register *rp;
7866
7867      rp = ip;
7868      while(rp->i_flag&ILOCK) {
7869          rp->i_flag |= IWANT;
7870          sleep(rp, PPIPE);
7871      }
7872      rp->i_flag |= ILOCK;
7873  }
```

/* priorities: do not alter much */
0152
0153
0154  #define PSWP    -100
0155  #define PINOD   -90
0156  #define PRIBIO  -50
0157  #define PPIPE   1
0158  #define PWAIT   40
0159  #define PSLEP   90
0160  #define PUSER   100
Function – prele()

- Unlocks the pipe after writing or reading

- If WANT bit is on:
  - Wakeup
prele(ip)
int *ip;
{
    register *rp;
    rp = ip;
    rp->i_flag &= ~ILOCK;
    if(rp->i_flag&IWANT) {
        rp->i_flag &= ~IWANT;
        wakeup(rp);
    }
}
End