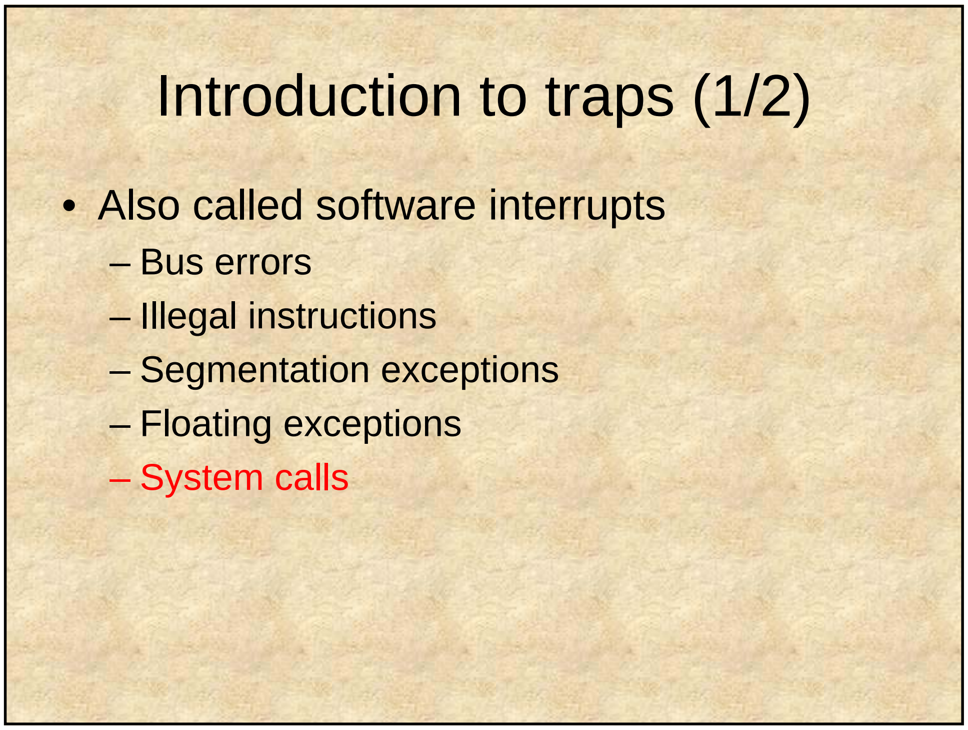
Lion's Commentary on UNIX Chapter 12

COP 5611: Operating Systems

Presented by: Dahai Guo
Hua Zhang

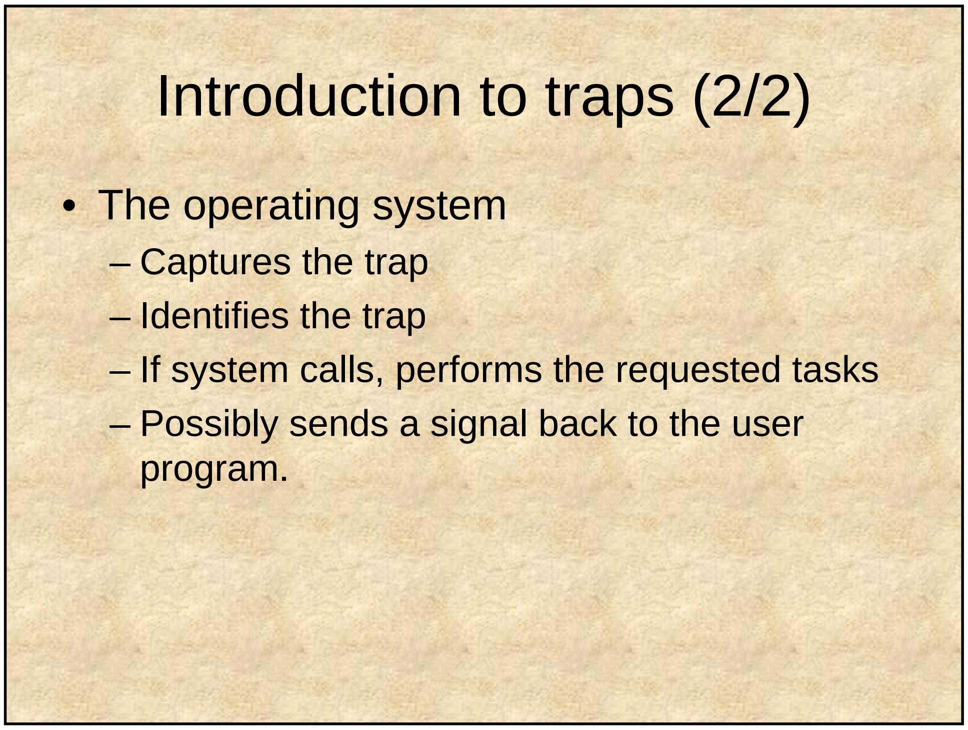
Outline

- Introduction to traps
- Traps in UNIX
- The ***trap*** function
- System calls
- "exec" system call
- Conclusions



Introduction to traps (1/2)

- Also called software interrupts
  - Bus errors
  - Illegal instructions
  - Segmentation exceptions
  - Floating exceptions
  - System calls



Introduction to traps (2/2)

- The operating system
  - Captures the trap
  - Identifies the trap
  - If system calls, performs the requested tasks
  - Possibly sends a signal back to the user program.

Traps in UNIX (1/4)

- UNIX divides traps into three classes, depending on the prior processor mode and the source of the trap
  - Kernel mode
  - User mode, not due to a “trap” instruction
  - User mode, due to a “trap” instruction
    - i.e. System calls

Traps in UNIX (2/4)

- Kernel mode traps
  - Unexpected
  - Usually caused by a kernel mode bus error
  - Examples:
    - Reading past EOF, reading a closed file, bad file pointers
    - Referencing a non-existent bus device

Traps in UNIX (3/4)

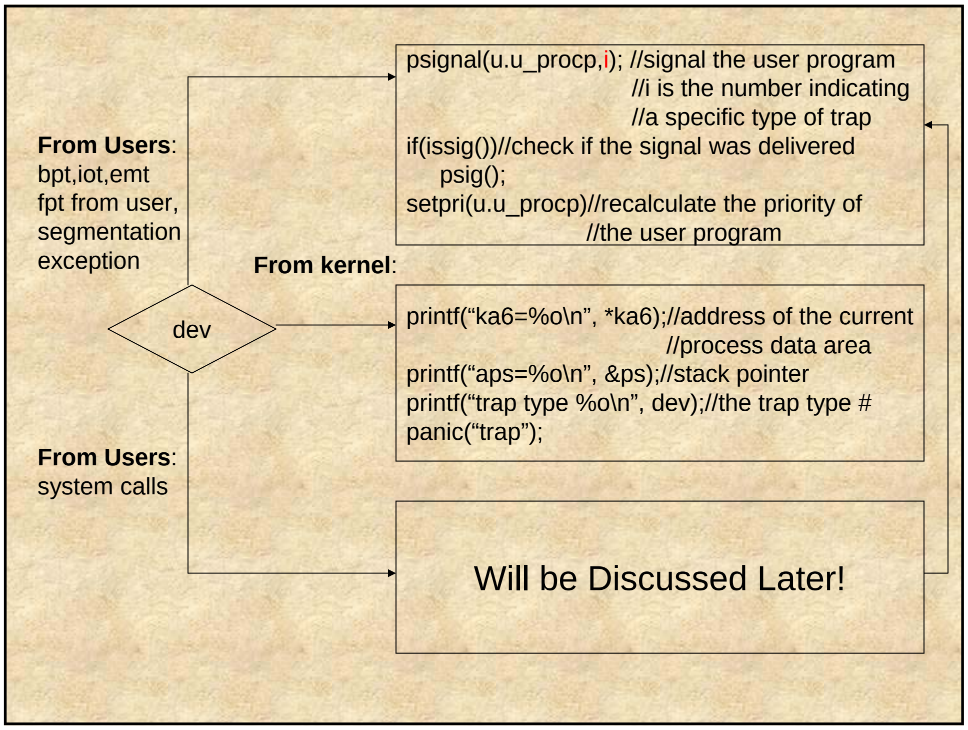
- User mode traps, not due to a “trap” instruction
  - Unexpected
  - Regarded as errors for which the operating system do not provide any handling, but “core dump”
  - Examples: I/O Traps, Trace/BPT Traps, Floating Exceptions, Segmentation Faults, Illegal Instructions

Traps in UNIX (4/4)

- User mode traps, due to a “trap” instruction
  - Expected system calls
  - User mode programs use “trap” instructions as part of the “system call” mechanism to call upon the os for assistance.
  - Examples: exec, open, close, time, etc.

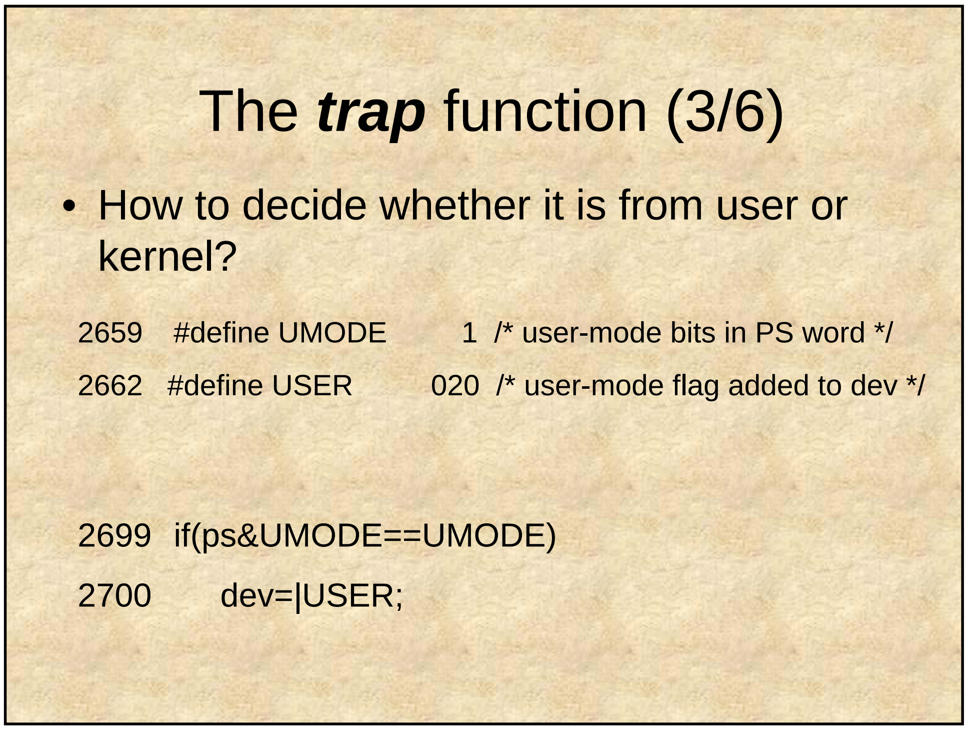
The ***trap*** function (1/6)

- trap(dev,sp,r1,nps,r0,pc,ps)
  - dev: the kind of trap that occurred
  - sp: stack pointer
  - nps: new process status
  - r1 and r0: two registers
  - pc: program counter
  - ps: process status
- The change to any of these parameters will be reflected to the caller.



```
psignal(u.u_procp,i); //signal the user program
                       //i is the number indicating
                       //a specific type of trap
if(issig())//check if the signal was delivered
    psig();
setpri(u.u_procp)//recalculate the priority of
                  //the user program
```

```
printf("ka6=%o\n", *ka6);//address of the current
                          //process data area
printf("aps=%o\n", &ps);//stack pointer
printf("trap type %o\n", dev);//the trap type #
panic("trap");
```

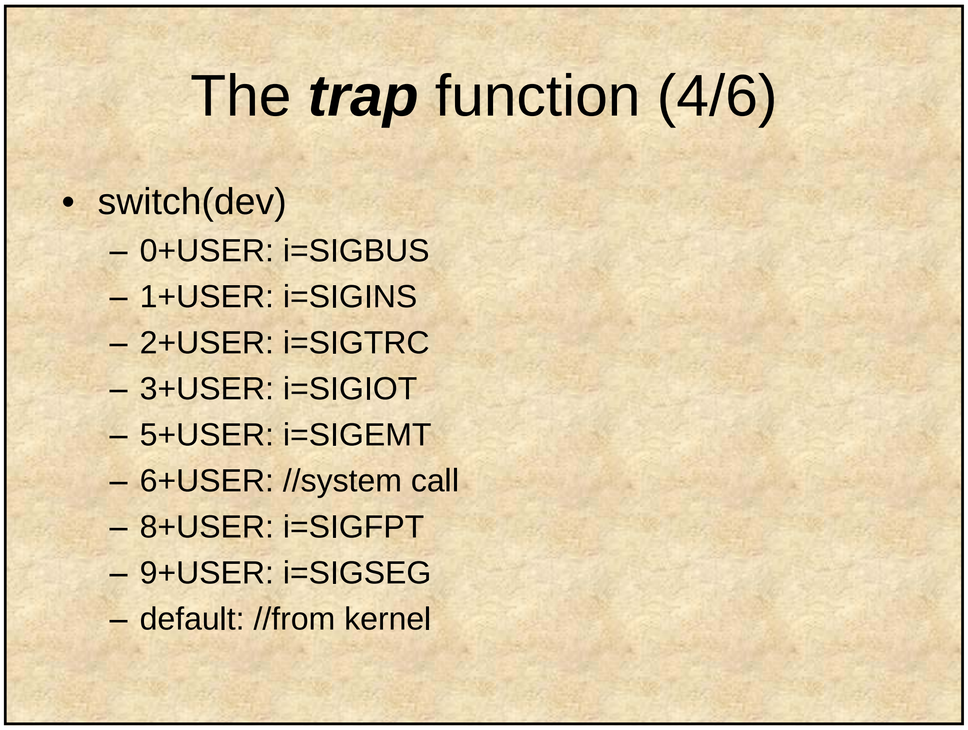


The *trap* function (3/6)

- How to decide whether it is from user or kernel?

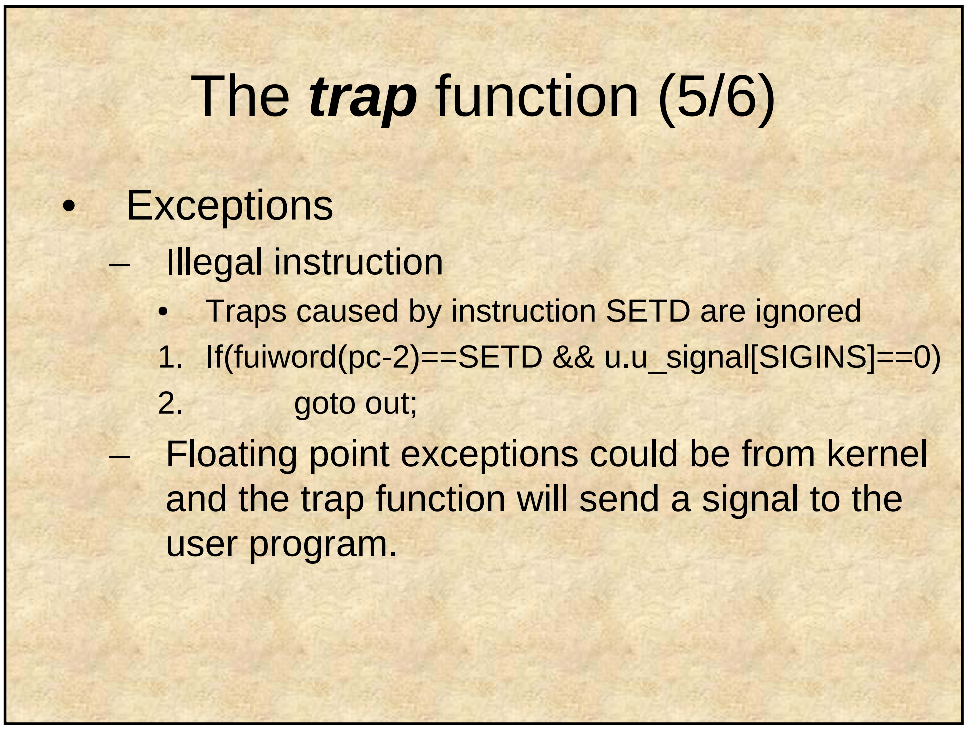
```
2659    #define UMODE        1  /* user-mode bits in PS word */
2662   #define USER        020  /* user-mode flag added to dev */


2699   if(ps&UMODE==UMODE)
2700        dev=|USER;
```



The *trap* function (4/6)

- switch(dev)
  - 0+USER: i=SIGBUS
  - 1+USER: i=SIGINS
  - 2+USER: i=SIGTRC
  - 3+USER: i=SIGIOT
  - 5+USER: i=SIGEMT
  - 6+USER: //system call
  - 8+USER: i=SIGFPT
  - 9+USER: i=SIGSEG
  - default: //from kernel

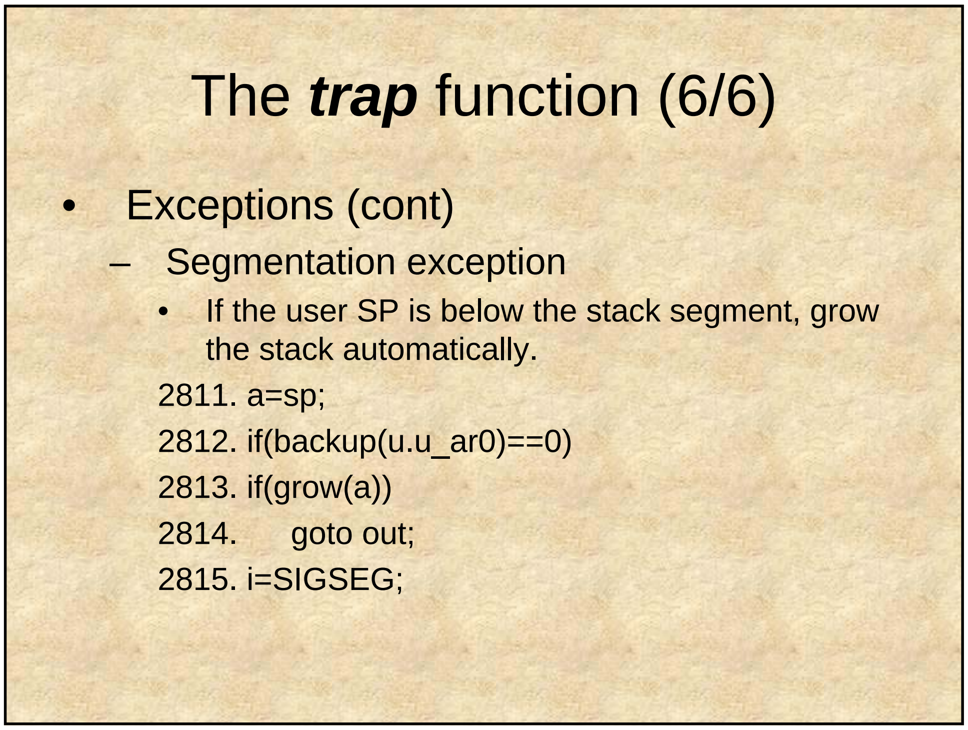


The *trap* function (5/6)

- Exceptions
  - Illegal instruction
    - Traps caused by instruction SETD are ignored

```
1. If(fuiword(pc-2)==SETD && u.u_signal[SIGINS]==0)
2.         goto out;
```

  - Floating point exceptions could be from kernel and the trap function will send a signal to the user program.

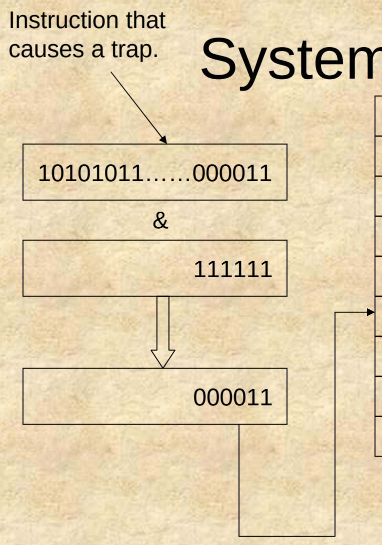


The *trap* function (6/6)

- Exceptions (cont)
  - Segmentation exception
    - If the user SP is below the stack segment, grow the stack automatically.

```
2811. a=sp;
2812. if(backup(u.u_ar0)==0)
2813. if(grow(a))
2814.     goto out;
2815. i=SIGSEG;
```

System Calls (1/11)



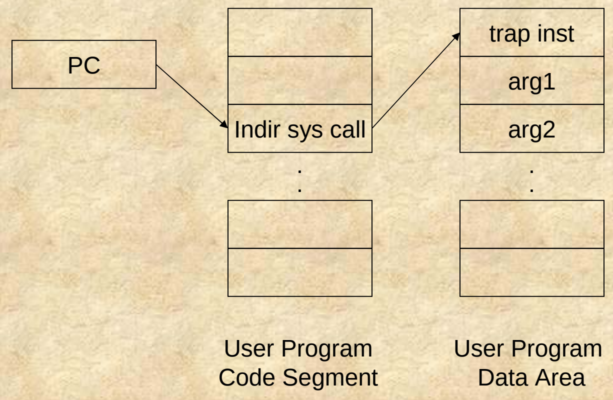
sysent		
index	arg count	System function
0	0	nullsys (indirect)
1	0	exit
2	0	fork
3	2	read
4	2	write
5	2	open
6	0	close
. . .	. . .	. . .
62	0	nosys
63	0	nosys

System Calls (2/11)

```
2754. callp=&sysent[fuiword(pc-2)&077];
2755. if(callp==sysent){
                //indirect system call
2764. }else{
                //direct system call
2769. }
```

System Calls (3/11)

Indirect System Call

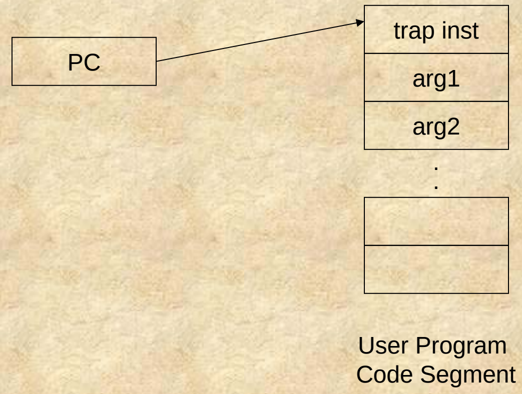


System Calls (4/11)

```
2755. if(callp==sysent)
2756. {//indirect system call
2757.    a=fuiword(pc);        //through pc, finds the addr of the
                               //instruction that causes the trap
2758.    pc+=2;                //increments pc
2759.    i=fuword(a);          //through a, finds the system call
2760.    if((i& ~077)!=SYS) //see if this is a system trap
2761.          i=077;          //if yes, i is nosys which
2762.                          //is fetal to the user
2762.    callp=&sysent[i&077];        //identifies the real system
                                      //call
2763.    for(i=0;i<callp->count;i++)  //fetches all the parameter
2764.         u.u_arg[i]=fuword(a+=2);
2765. }
```

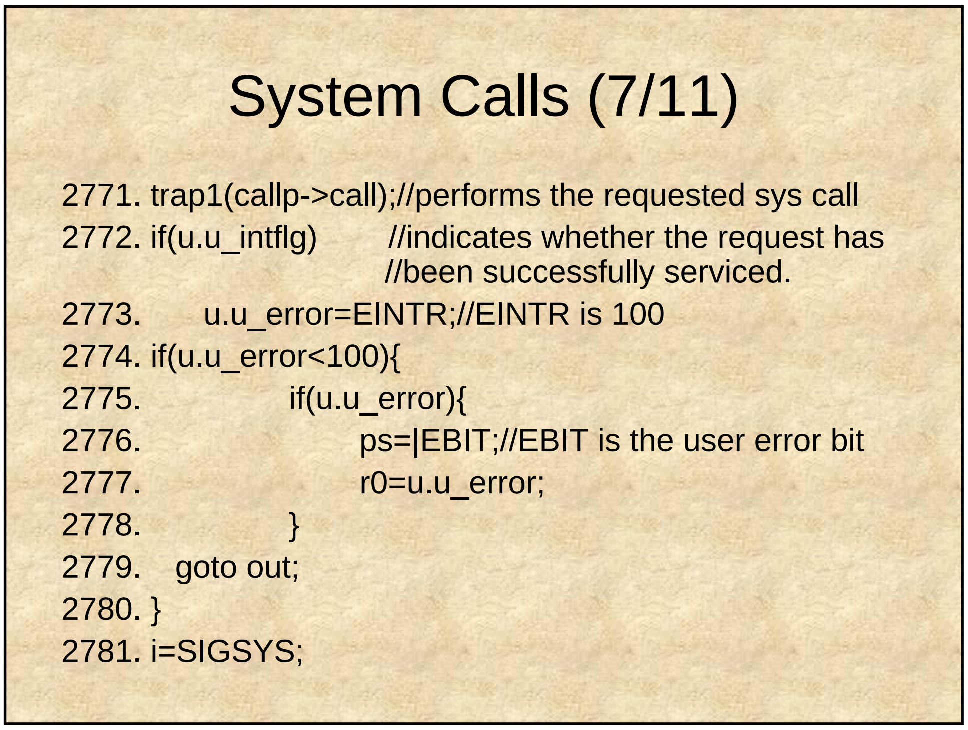
System Calls (5/11)

Direct System Call



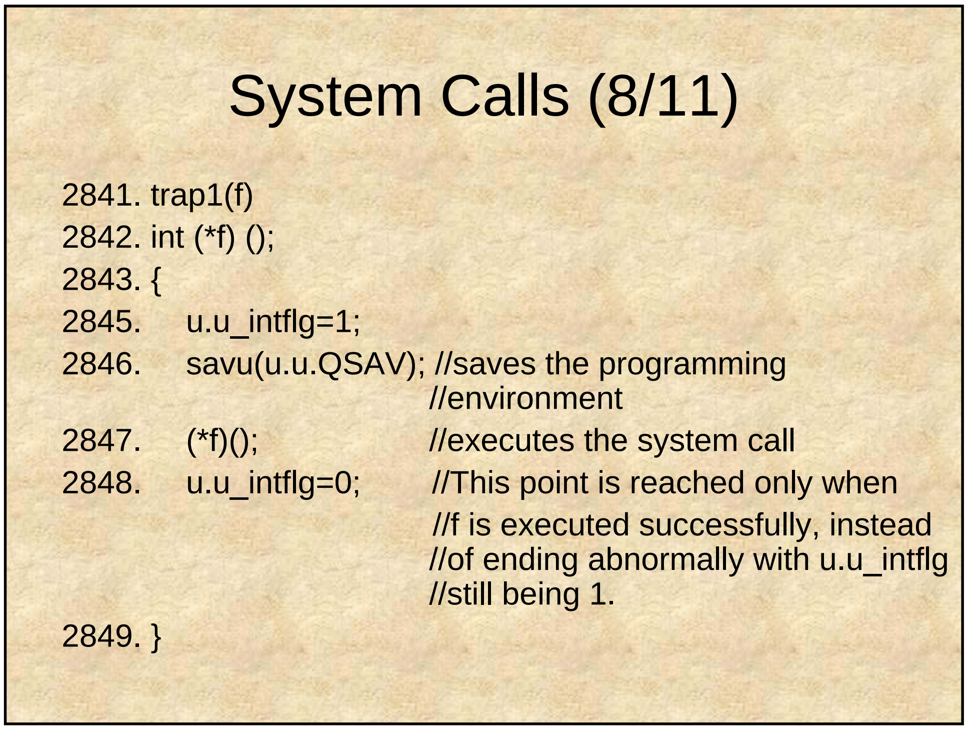
System Calls (6/11)

```
2755. if(callp==sysent){
              //indirect system call
2764. }else{//direct system call
2765.     for(i=0;i<callp->count;i++){
2766.          u.u_arg[i]=fuiword(pc);
2767.          pc+=2;
2768.     }
2769. }
```



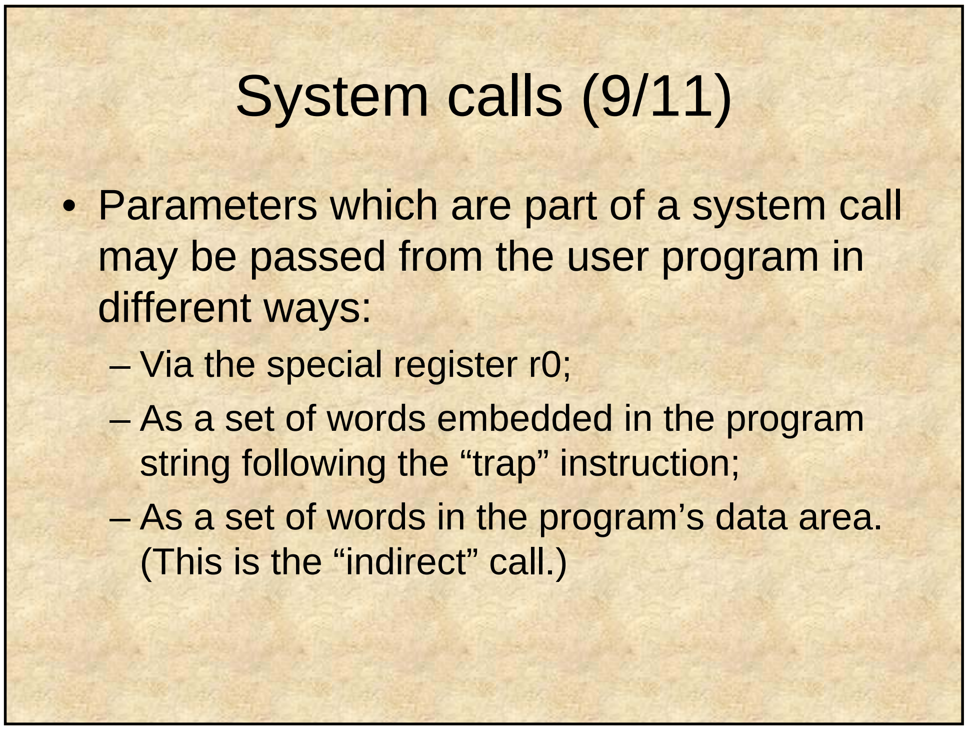
System Calls (7/11)

```
2771. trap1(callp->call);//performs the requested sys call
2772. if(u.u_intflg)       //indicates whether the request has
                           //been successfully serviced.
2773.     u.u_error=EINTR;//EINTR is 100
2774. if(u.u_error<100){
2775.           if(u.u_error){
2776.                 ps=|EBIT;//EBIT is the user error bit
2777.                 r0=u.u_error;
2778.           }
2779.   goto out;
2780. }
2781. i=SIGSYS;
```



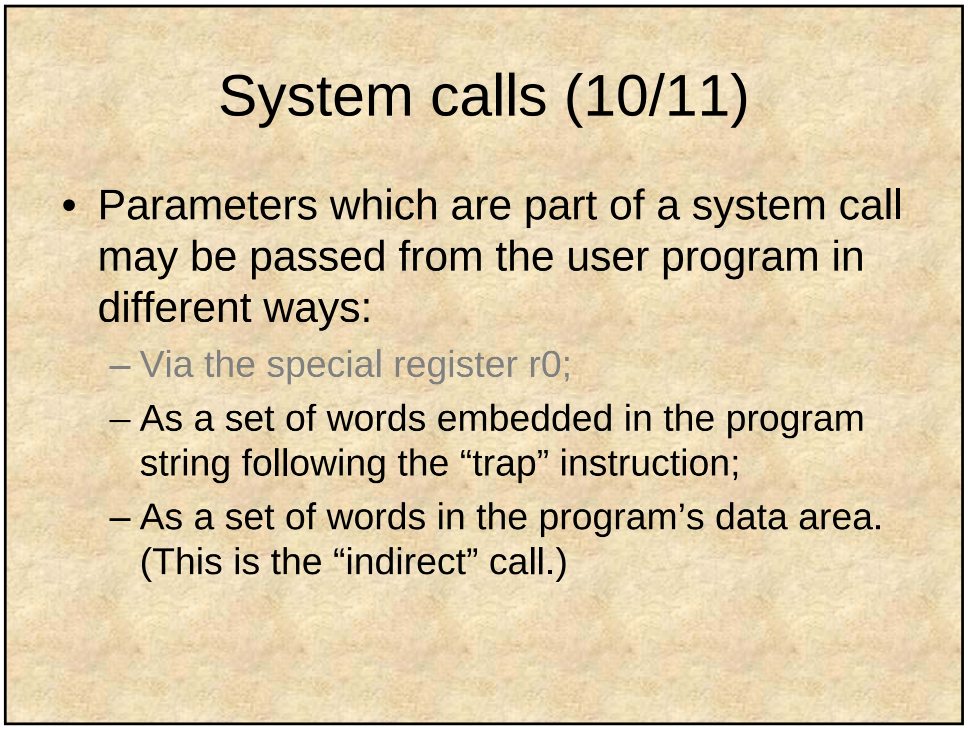
System Calls (8/11)

```
2841. trap1(f)
2842. int (*f) ();
2843. {
2845.     u.u_intflg=1;
2846.     savu(u.u.QSAV); //saves the programming
                      //environment
2847.     (*f)();         //executes the system call
2848.     u.u_intflg=0;   //This point is reached only when
                      //f is executed successfully, instead
                      //of ending abnormally with u.u_intflg
                      //still being 1.
2849. }
```



System calls (9/11)

- Parameters which are part of a system call may be passed from the user program in different ways:
  - Via the special register r0;
  - As a set of words embedded in the program string following the “trap” instruction;
  - As a set of words in the program’s data area. (This is the “indirect” call.)



System calls (10/11)

- Parameters which are part of a system call may be passed from the user program in different ways:
  - Via the special register r0;
  - As a set of words embedded in the program string following the “trap” instruction;
  - As a set of words in the program’s data area. (This is the “indirect” call.)

System calls (11/11)

- In the program, we only see the last two ways in terms of how parameters are passed when system calls occur.

exec: SYNOPSIS

- The 11th system call
- SYNOPSIS

  sys exec; name; args

  name: <...\0>

  ...

  – args: arg1; arg2; ...; 0

  – arg1: <...\0>

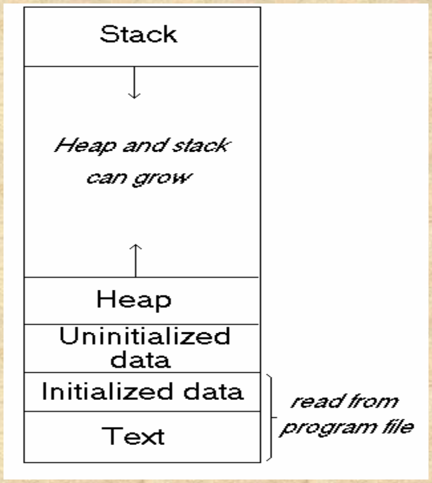
  – ...

exec: DESCRIPTION

- DESCRIPTION
  - exec overlays the calling process with the named file, then transfers to the beginning of the core image of the file.
  - The first argument to exec is a pointer to the name of the file to be executed. The second is the address of a list of pointers to arguments to be passed to the file. Conventionally, the first argument is the name of the file. Each pointer addresses a string terminated by a null byte.

exec: process address space

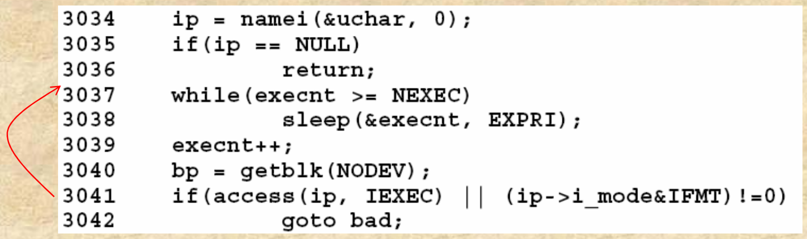
- **The text segment**
  - instructions
- **The initialized data segment**
  - initialized static variables.
- **The unitialized data segment**
  - unitialized static variables
- **The stack**
  - dynamic data, like arguments, return address, local variables
- **The heap**
  - dynamic allocated memory, like malloc() in c



exec: variables

- ip: reference to the *inode of the program file*
- *c: register for memory copy*
- bp: pointer to a temporary buffer
- cp: pointer to the address of bp
- ap: pointer to a argument
- na: number of arguments
- nc: number of bytes of all the arguments

exec: initializing

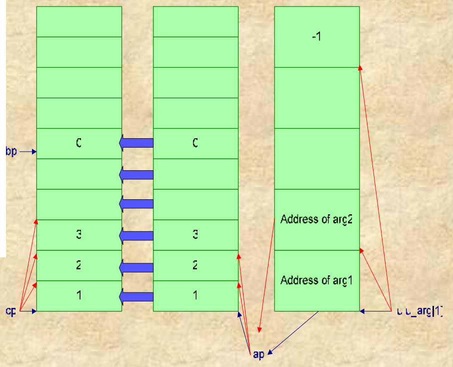


```
	ip = namei(&uchar, 0);
	if(ip == NULL)
		return;
	while(execnt >= NEXEC)
		sleep(&execnt, EXPRI);
	execnt++;
	bp = getblk(NODEV);
	if(access(ip, IEXEC) || (ip->i_mode&IFMT)!=0)
		goto bad;
```

- Convert the first argument into an “inode” reference
- Limit the number of processes running simultaneously to avoid deadly waiting
- Allocate temporary buffer to read in arguments
- Check whether the file is executable
  - Might be more efficient

exec: copy aruguments

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

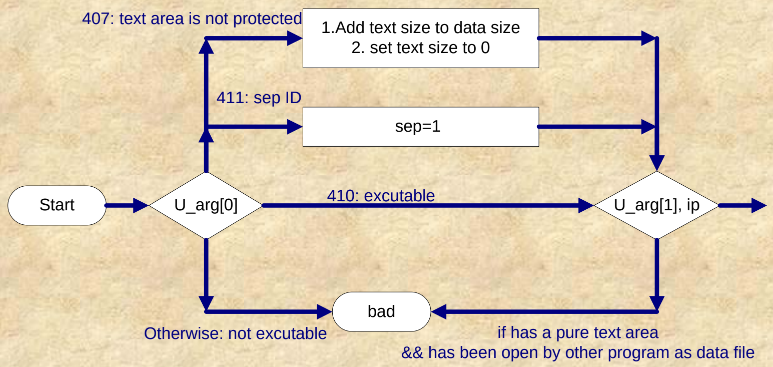


exec: read header

```
u.u_base = &u.u_arg[0];
u.u_count = 8;
u.u_offset[1] = 0;
u.u_offset[0] = 0;
u.u_segflg = 1;
readi(ip);
u.u_segflg = 0;
if(u.u_error)
        goto bad;
```

- Read 8 bytes into u_arg[0] through u_arg[3]
  - u_arg[0]: 407/410/411 (410 -> RO text) (411 -> sep ID)
  - u_arg[1]: text size
  - u_arg[2]: data size
  - u_arg[3]: bss size
- u_segflg: 1 kernel space, 0 user space

exec: process header

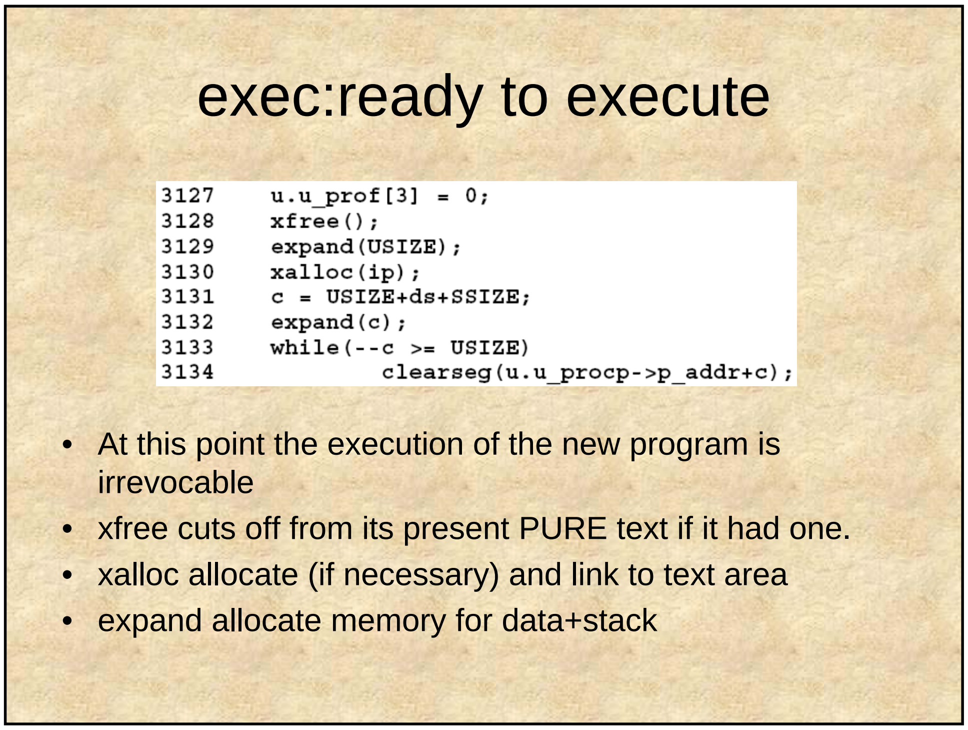


exec: memory check

```
    ts = ((u.u_arg[1]+63)>>6) & 01777;
    ds = ((u.u_arg[2]+u.u_arg[3]+63)>>6) & 0177;
    if(estabur(ts, ds, SSIZE, sep))
            goto bad;

```

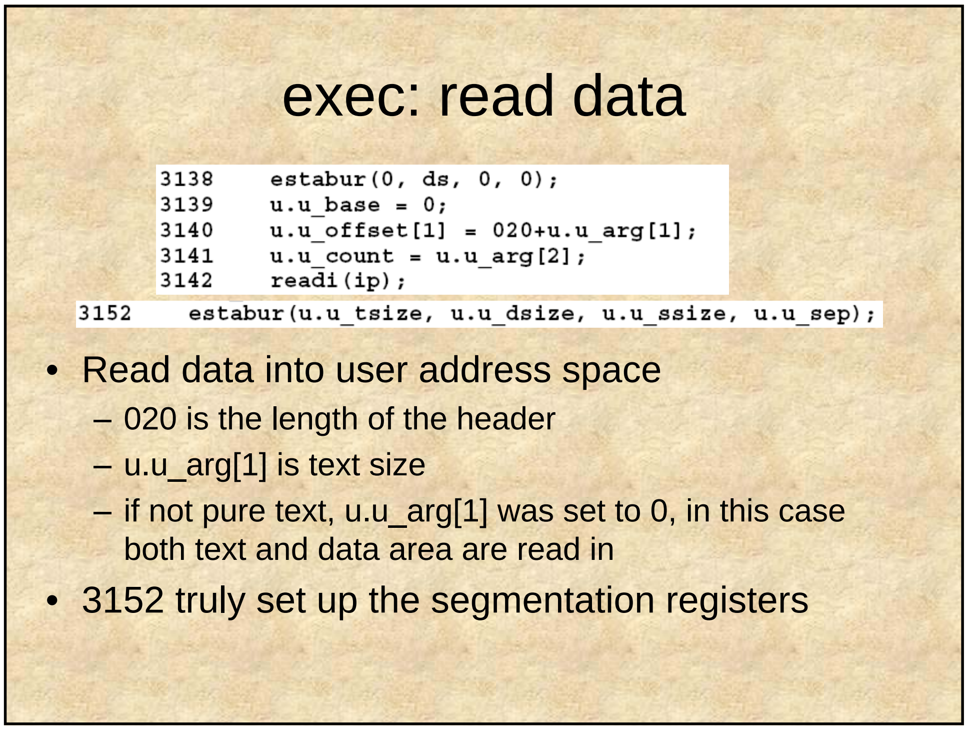
- Check whether text and data size exceed max sizes
- estabur()
  - Line 1650
  - Set up software prototype segmentation registers to implement the 3 pseudo text, data, stack segment sizes passed as arguments
  - The argument sep specifies if the text and data+stack segments are to be separated



exec:ready to execute

```
3127     u.u_prof[3] = 0;
3128     xfree();
3129     expand(USIZE);
3130     xalloc(ip);
3131     c = USIZE+ds+SSIZE;
3132     expand(c);
3133     while(--c >= USIZE)
3134             clearseg(u.u_procp->p_addr+c);
```

- At this point the execution of the new program is irrevocable
- xfree cuts off from its present PURE text if it had one.
- xalloc allocate (if necessary) and link to text area
- expand allocate memory for data+stack



exec: read data

```
3138     estabur(0, ds, 0, 0);
3139     u.u_base = 0;
3140     u.u_offset[1] = 020+u.u_arg[1];
3141     u.u_count = u.u_arg[2];
3142     readi(ip);
3152     estabur(u.u_tsize, u.u_dsize, u.u_ssize, u.u_sep);
```

- Read data into user address space
  - 020 is the length of the header
  - u.u_arg[1] is text size
  - if not pure text, u.u_arg[1] was set to 0, in this case both text and data area are read in
- 3152 truly set up the segmentation registers

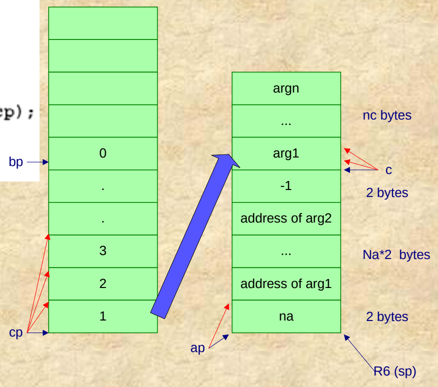
exec: copy arguments

```
cp = bp->b_addr;
ap = -nc - na*2 - 4;
u.u_ar0[R6] = ap;
suword(ap, na);
c = -nc;
while(na--) {
        suword(ap=+2, c);
        do
                subyte(c++, *cp);
        while(*cp++);
}
suword(ap+2, -1);
```

- Copy information from bp to user space
- R6 is the stack pointer, ap is unsigned integer, so not negative
- suword and subyte are functions to write a word or a byte into user address space

exec: copy arguments

```
cp = bp->b_addr;
ap = -nc - na*2 - 4;
u.u_ar0[R6] = ap;
suword(ap, na);
c = -nc;
while(na--) {
        suword(ap=+2, c);
        do
                subyte(c++, *cp);
        while(*cp++);
}
suword(ap+2, -1);
```



exec: set SUID/SGID

```
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;
}
```

- Set SUID/SGID protections, if not tracing
- SUID/SGID are used in UNIX for a user to run a program as the program is run by its owner
  - eg., allow a normal user to run a script which need root privilege

exec: clearing

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

- Clear sigs, regs, and return
  - R7 is “pc”, the instruction counter. R7 is set to 0 so that when returns the next instruction will be executed is the instruction in user space at address 0
  - Remember that address 0 is the text area

exec: exit

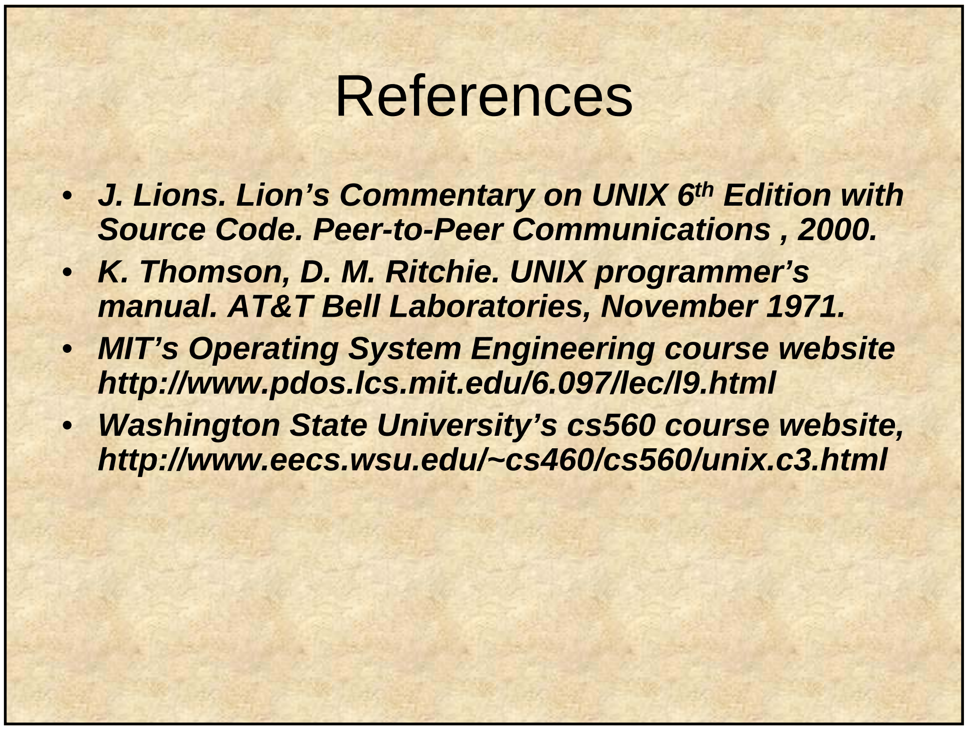
```
3193 bad:
3194     iput(ip);
3195     brelse(bp);
3196     if(execnt >= NEXEC)
3197             wakeup(&execnt);
3198     execnt--;
```

```
3037     while(execnt >= NEXEC)
3038             sleep(&execnt, EXPRI);
```

- Any jump to here would be an error in u
- Release the inode pointer
- Release the buffer
- Wake up anyone waiting at line 3038

Summary

- Introduction of traps
- Traps in UNIX
- The ***trap*** function
- exec() system call



References

- ***J. Lions. Lion’s Commentary on UNIX 6th Edition with Source Code. Peer-to-Peer Communications , 2000.***
- ***K. Thomson, D. M. Ritchie. UNIX programmer’s manual. AT&T Bell Laboratories, November 1971.***
- ***MIT’s Operating System Engineering course website http://www.pdos.lcs.mit.edu/6.097/lec/l9.html***
- ***Washington State University’s cs560 course website, http://www.eecs.wsu.edu/~cs460/cs560/unix.c3.html***