Operating Systems

Xinu - Process Management

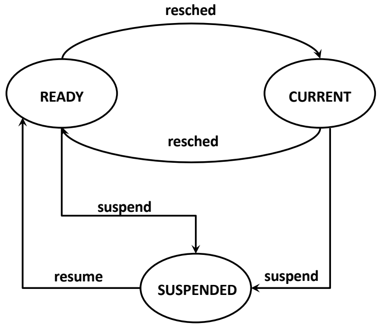
1

Suspending and Resuming

- Sometimes OS functions temporarily stop a process (suspend)
  - Waiting for one of a set of conditions to become true before resuming
  - Cannot be in *ready* state – new state: *suspended*
- Operations:
  - suspend (make ineligible to use the CPU)
  - resume (make eligible to run again)
- Augmented state transition diagram

2

Process State Transition Diagram

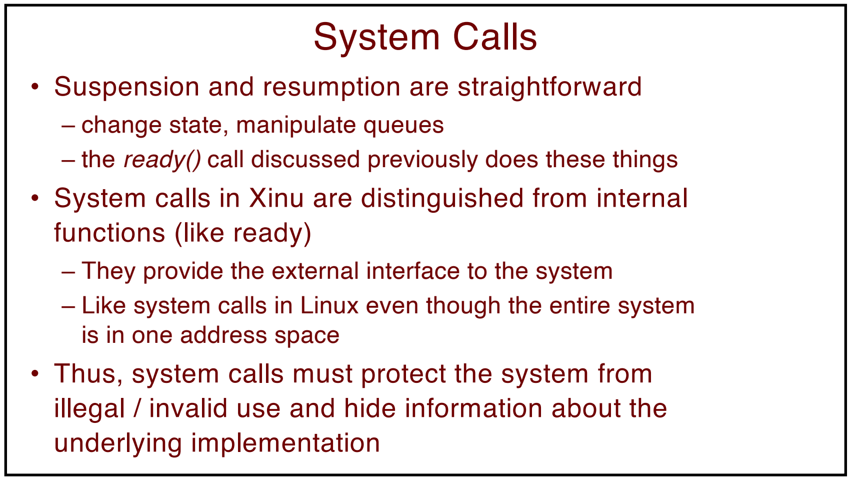


3

Suspending and Implementation

- One process can suspend another process, or a process may suspend itself
  - The function takes a PID as an argument: *suspend(PID);*
- The function *getpid()* returns the PID of the calling process
  - Despite the fact that this value is stored in the global variable *currpid*
- The principle of information hiding states that implementation details should be hidden unless necessary
  - The implementation can change without affecting programs
  - Outweighs efficiency concerns (in general)

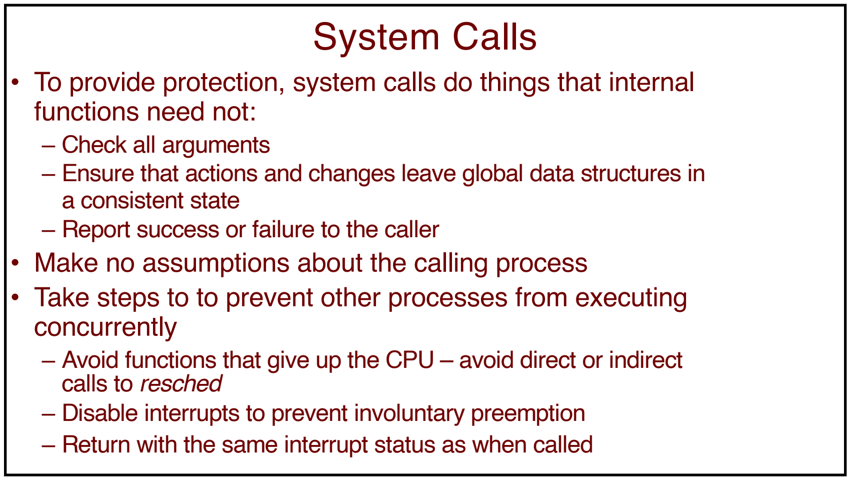
4



System Calls

- Suspension and resumption are straightforward
  - change state, manipulate queues
  - the *ready()* call discussed previously does these things
- System calls in Xinu are distinguished from internal functions (like ready)
  - They provide the external interface to the system
  - Like system calls in Linux even though the entire system is in one address space
- Thus, system calls must protect the system from illegal / invalid use and hide information about the underlying implementation

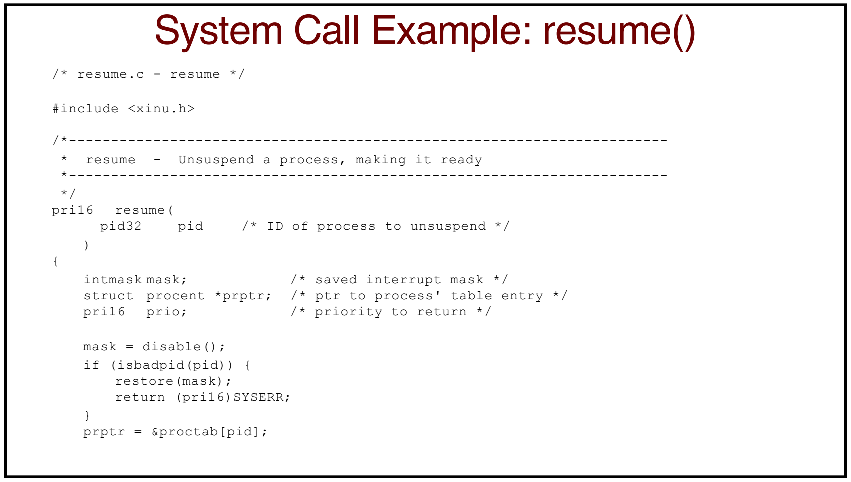
5



System Calls

- To provide protection, system calls do things that internal functions need not:
  - Check all arguments
  - Ensure that actions and changes leave global data structures in a consistent state
  - Report success or failure to the caller
- Make no assumptions about the calling process
- Take steps to to prevent other processes from executing concurrently
  - Avoid functions that give up the CPU – avoid direct or indirect calls to *resched*
  - Disable interrupts to prevent involuntary preemption
  - Return with the same interrupt status as when called

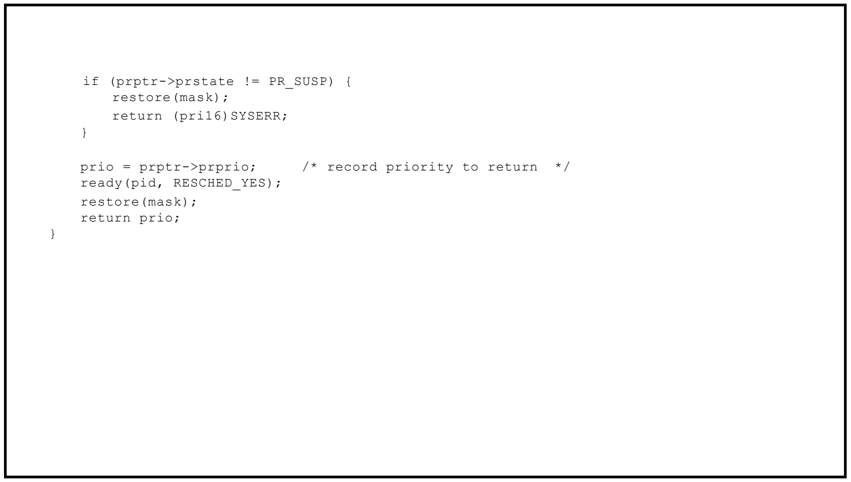
6



System Call Example: resume()

```
/* resume.c - resume */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  resume  -  Unsuspend a process, making it ready
 *------------------------------------------------------------------------
 */
pri16   resume(
      pid32     pid     /* ID of process to unsuspend */
    )
{
    intmask mask;             /* saved interrupt mask */
    struct  procent *prptr;   /* ptr to process' table entry */
    pri16   prio;             /* priority to return */

    mask = disable();
    if (isbadpid(pid)) {
        restore(mask);
        return (pri16)SYSERR;
    }
    prptr = &proctab[pid];
```

7



```
    if (prptr->prstate != PR_SUSP) {
        restore(mask);
        return (pri16)SYSERR;
    }

    prio = prptr->prprio;      /* record priority to return  */
    ready(pid, RESCHED_YES);
    restore(mask);
    return prio;
}
```

8

Interrupt state

- Disable interrupts to prevent involuntary preemption
- Return with the same interrupt status as when called
- Rather than just enabling interrupts before returning, save and restore interrupt status
- `disable()` turns off interrupts and records the prior state (mask)
  - Which may have been disabled already
- `restore()` takes a prior state and configures the processor with it
- Calling *disable* works called with interrupts enabled or disabled
- *An operating system function always returns to its caller with the same interrupt status as when it was called.*

System Call Skeleton

```
syscall function_name ( args )     {
    intmask mask;        /* interrupt mask */
    mask = disable ( ) ; /* disable interrupts at start of function */
    if ( args are incorrect ) {
        restore (mask) ; /* restore interrupts before error return */
        return (SYSERR) ;
    }
    . . . other processing . . .
    if ( an error occurs )    {
        restore (mask) ; /* restore interrupts before error return */
        return (SYSERR) ;
    }
    . . . more processing . . .
    restore (mask) ;     /* restore interrupts before normal return */
    return ( appropriate value ) ;
```

Implementation and Suspend

- Some system calls simply return the constant OK, while others return a relevant value
  - SYSERR is returned otherwise
- Suspend can be called on a process that is ready
  - easy: remove from ready list, change state to suspended (PR_SUSP)
- Applying to the current process is slightly more involved
  - it must result in calling *resched*

Suspend

```
/* suspend.c - suspend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  suspend  -  Suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
syscall suspend(
      pid32    pid    /* ID of process to suspend   */
    )
{
    intmask mask;                   /* saved interrupt mask */
    struct  procent *prptr;         /* ptr to process' table entry */
    pri16   prio;                   /* priority to return */

    mask = disable();
    if (isbadpid(pid) || (pid == NULLPROC)) {
        restore(mask);
        return SYSERR;
    }
/* Only suspend a process that is current or ready */

    prptr = &proctab[pid];
    if ((prptr->prstate != PR_CURR) && (prptr->prstate != PR_READY)) {
        restore(mask);
        return SYSERR;
    }
    if (prptr->prstate == PR_READY) {
        getitem(pid);            /* remove a ready process */
                                  /* from the ready list    */
        prptr->prstate = PR_SUSP;
    } else {
        prptr->prstate = PR_SUSP;   /* mark the current process */
        resched();                  /* suspended and reschedule */
    }
    prio = prptr->prprio;
    restore(mask);
    return prio;
}
```

Priority used to communicate information

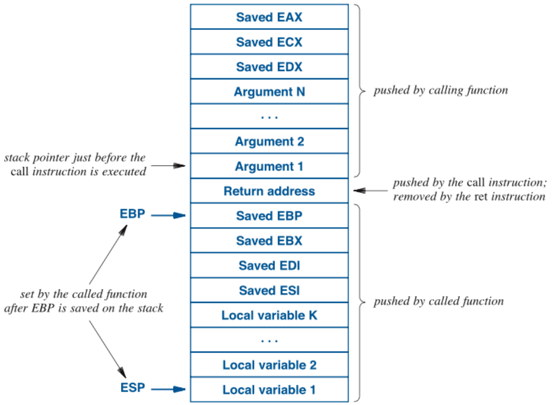
```
newprio = suspend ( getpaid ( ) ) ;
if ( newprio == 25 ) {
    ... event 1 occurred ...
} else {
    ... event 2 occurred ...
}
```

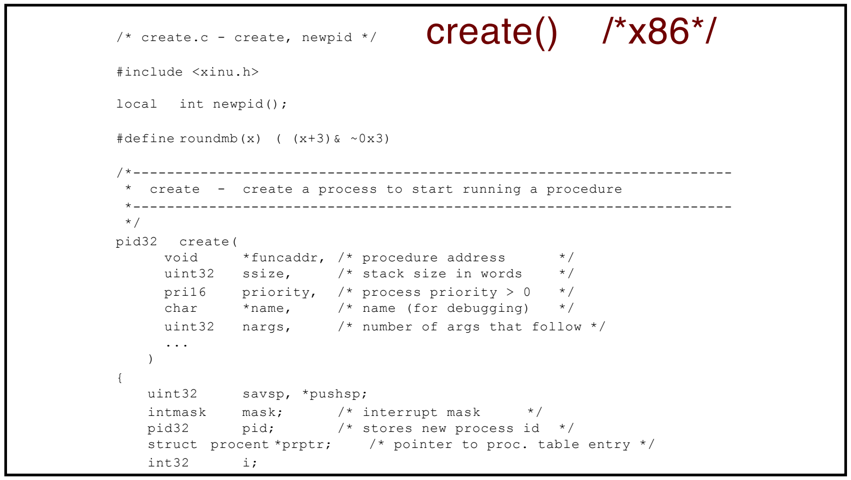
- Suspend returns the priority that the process had when suspend returns

Process Creation

- The create() call configures the stack as if the function had been called from elsewhere in the code
- Configures the stack according to the compiler and architecture calling conventions
- Then it creates entries on the stack as if *ctxsw* had been called

x86 Stack Layout

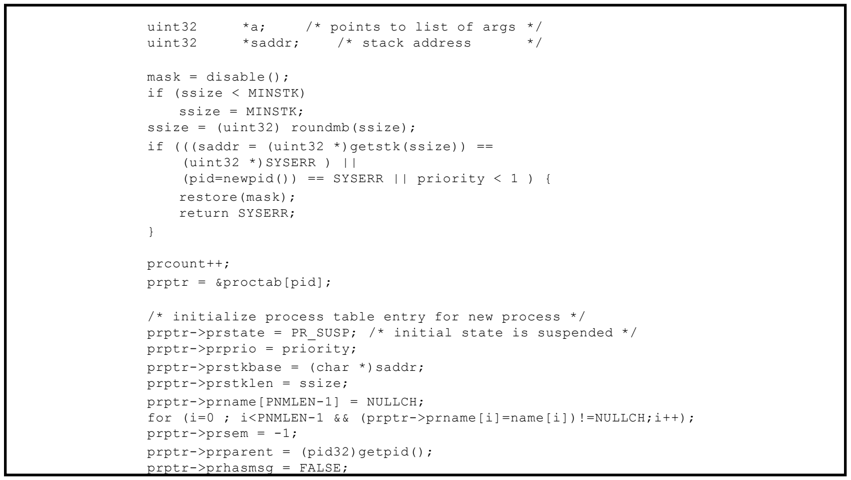

Saved EAX
Saved ECX
Saved EDX
Argument N
. . .
Argument 2
Argument 1
Return address
Saved EBP
Saved EBX
Saved EDI
Saved ESI
Local variable K
. . .
Local variable 2
Local variable 1
pushed by calling function
stack pointer just before the call instruction is executed
pushed by the call instruction; removed by the ret instruction
EBP
set by the called function after EBP is saved on the stack
ESP
pushed by called function



create()   /*x86*/

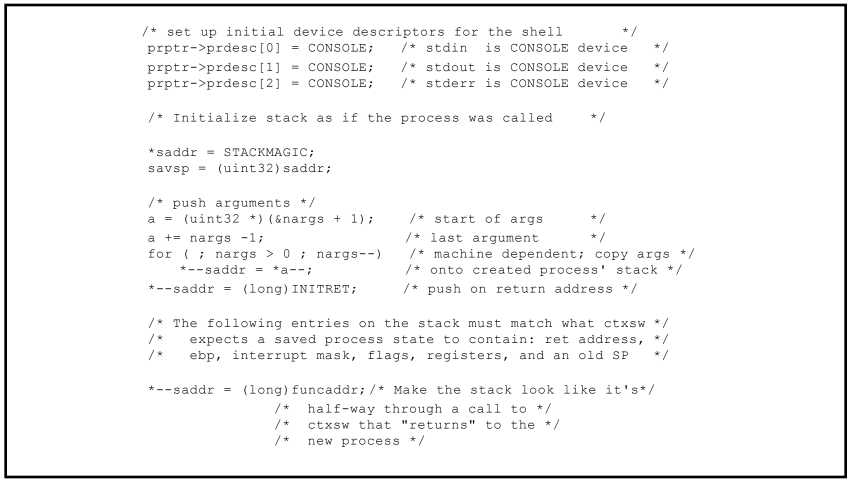
```
/* create.c - create, newpid */

#include <xinu.h>

local   int newpid();

#define roundmb(x)  ( (x+3)& ~0x3)

/*------------------------------------------------------------------------
 *  create  -  create a process to start running a procedure
 *------------------------------------------------------------------------
 */
pid32   create(
      void      *funcaddr, /* procedure address      */
      uint32    ssize,     /* stack size in words    */
      pri16     priority,  /* process priority > 0   */
      char      *name,     /* name (for debugging)   */
      uint32    nargs,     /* number of args that follow */
      ...
    )
{
    uint32      savsp, *pushsp;
    intmask     mask;      /* interrupt mask      */
    pid32       pid;       /* stores new process id   */
    struct  procent *prptr;    /* pointer to proc. table entry */
    int32       i;
```

17



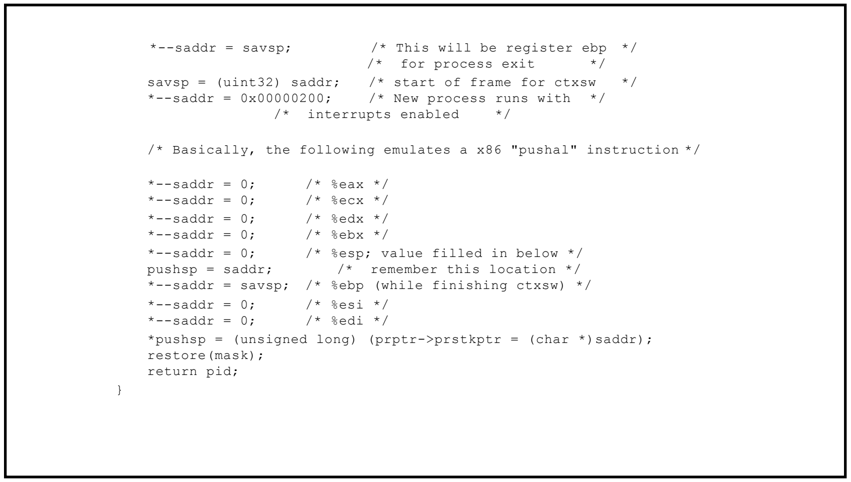
```
uint32      *a;     /* points to list of args */
uint32      *saddr;    /* stack address       */

mask = disable();
if (ssize < MINSTK)
    ssize = MINSTK;
ssize = (uint32) roundmb(ssize);
if (((saddr = (uint32 *)getstk(ssize)) ==
     (uint32 *)SYSERR ) ||
     (pid=newpid()) == SYSERR || priority < 1 ) {
    restore(mask);
    return SYSERR;
}

prcount++;
prptr = &proctab[pid];

/* initialize process table entry for new process */
prptr->prstate = PR_SUSP; /* initial state is suspended */
prptr->prprio = priority;
prptr->prstkbase = (char *)saddr;
prptr->prstklen = ssize;
prptr->prname[PNMLEN-1] = NULLCH;
for (i=0 ; i<PNMLEN-1 && (prptr->prname[i]=name[i])!=NULLCH;i++);
prptr->prsem = -1;
prptr->prparent = (pid32)getpid();
prptr->prhasmsg = FALSE;
```

18



```
/* set up initial device descriptors for the shell       */
 prptr->prdesc[0] = CONSOLE;   /* stdin  is CONSOLE device   */
 prptr->prdesc[1] = CONSOLE;   /* stdout is CONSOLE device   */
 prptr->prdesc[2] = CONSOLE;   /* stderr is CONSOLE device   */

 /* Initialize stack as if the process was called    */

 *saddr = STACKMAGIC;
 savsp = (uint32)saddr;

 /* push arguments */
 a = (uint32 *)(&nargs + 1);    /* start of args      */
 a += nargs -1;                /* last argument      */
 for ( ; nargs > 0 ; nargs--)   /* machine dependent; copy args */
     *--saddr = *a--;          /* onto created process' stack */
 *--saddr = (long)INITRET;     /* push on return address */

 /* The following entries on the stack must match what ctxsw */
 /*   expects a saved process state to contain: ret address, */
 /*   ebp, interrupt mask, flags, registers, and an old SP   */

 *--saddr = (long)funcaddr;/* Make the stack look like it's*/
                /*  half-way through a call to */
                /*  ctxsw that "returns" to the */
                /*  new process */
```

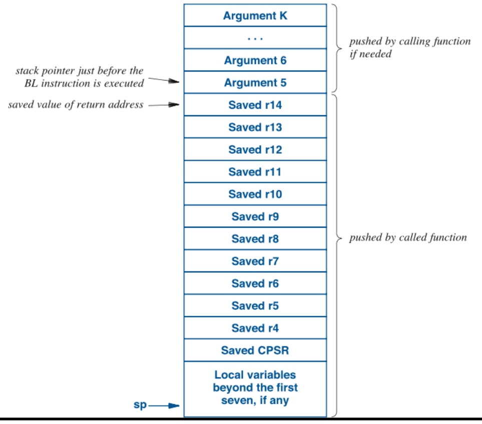
19



```
     *--saddr = savsp;         /* This will be register ebp  */
                              /*  for process exit       */
    savsp = (uint32) saddr;   /* start of frame for ctxsw   */
    *--saddr = 0x00000200;    /* New process runs with  */
                /*  interrupts enabled    */

    /* Basically, the following emulates a x86 "pushal" instruction*/

    *--saddr = 0;      /* %eax */
    *--saddr = 0;      /* %ecx */
    *--saddr = 0;      /* %edx */
    *--saddr = 0;      /* %ebx */
    *--saddr = 0;      /* %esp; value filled in below */
    pushsp = saddr;        /*  remember this location */
    *--saddr = savsp;  /* %ebp (while finishing ctxsw) */
    *--saddr = 0;      /* %esi */
    *--saddr = 0;      /* %edi */
    *pushsp = (unsigned long) (prptr->prstkptr = (char *)saddr);
    restore(mask);
    return pid;
}
```

20

ARM Stack



Argument K

. . .

Argument 6

Argument 5

Saved r14

Saved r13

Saved r12

Saved r11

Saved r10

Saved r9

Saved r8

Saved r7

Saved r6

Saved r5

Saved r4

Saved CPSR

Local variables beyond the first seven, if any

pushed by calling function if needed

pushed by called function

stack pointer just before the BL instruction is executed

saved value of return address

sp

21

ARM version

```
/* push arguments */
a = (uint32 *)(&nargs + 1);     /* start of args                 */
a += nargs -1;                  /* last argument                 */
for ( ; nargs > 4 ; nargs--)    /* machine dependent; copy args */
        *--saddr = *a--;        /* onto created process' stack  */
*--saddr = (long)procaddr;
for(i = 11; i >= 4; i--)
        *--saddr = 0;
for(i = 4; i > 0; i--) {
        if(i <= nargs)
                *--saddr = *a--;
        else
                *--saddr = 0;
}
*--saddr = (long)INITRET;       /* push on return address       */
*--saddr = (long)0x00000053;    /* CPSR, A, F bits set,         */
                                /* Supervisor mode              */
prptr->prstkptr = (char *)saddr;
restore(mask);
return pid;
```

22

newpid()

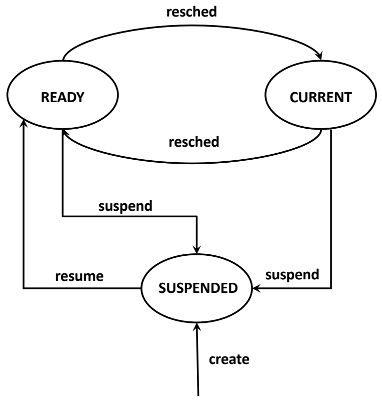
```
/*------------------------------------------------------------------------
 *  newpid  -  Obtain a new (free) process ID
 *------------------------------------------------------------------------
 */
local   pid32   newpid(void)
{
    uint32  i;                  /* iterate through all processes*/
    static  pid32 nextpid = 1;  /* position in table to try or */
                                /*  one beyond end of table   */

    /* check all NPROC slots */

    for (i = 0; i < NPROC; i++) {
        nextpid %= NPROC;  /* wrap around to beginning */
        if (proctab[nextpid].prstate == PR_FREE) {
            return nextpid++;
        } else {
            nextpid++;
        }
    }
    return (pid32) SYSERR;
}
```

23

userret()

```
/* userret.c - userret */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  userret  -  Called when a process returns from the top-level function
 *------------------------------------------------------------------------
 */
void    userret(void)
{
    kill(getpid());             /* force process exit */
}
```

24



kill()

```
/* kill.c - kill */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  kill  -  Kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
syscall kill(
      pid32     pid     /* ID of process to kill  */
    )
{
    intmask mask;            /* saved interrupt mask */
    struct  procent *prptr; /* ptr to process' table entry */
    int32   i;               /* index into descriptors    */

    mask = disable();
    if (isbadpid(pid) || (pid == NULLPROC)
        || ((prptr = &proctab[pid])->prstate) == PR_FREE) {
        restore(mask);
        return SYSERR;
    }
```

```
    if (--prcount <= 1) {     /* last user process completes */
        xdone();
    }

    send(prptr->prparent, pid);
    for (i=0; i<3; i++) {
        close(prptr->prdesc[i]);
    }
    freestk(prptr->prstkbase, prptr->prstklen);

    switch (prptr->prstate) {
    case PR_CURR:
        prptr->prstate = PR_FREE; /* suicide */
        resched();

    case PR_SLEEP:
    case PR_RECTIM:
        unsleep(pid);
        prptr->prstate = PR_FREE;
        break;

    case PR_WAIT:
        semtab[prptr->prsem].scount++;
        /* fall through */
```

```
    case PR_READY:
        getitem(pid);      /* remove from queue */
        /* fall through */

    default:
        prptr->prstate = PR_FREE;
    }

    restore(mask);
    return OK;
}
```

xdone()

```
/* xdone.c - xdone */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  xdone  -  Print system completion message as last thread exits
 *------------------------------------------------------------------------
 */
void	xdone(void)
{
	kprintf("\r\n\r\nAll user processes have completed.\r\n\r\n");
	/* gpioLEDOff(GPIO_LED_CISCOWHT); turn off LED "run" light */
	halt();			/* halt the processor	*/
}
```

getprio()

```
/* getprio.c - getprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  getprio  -  Return the scheduling priority of a process
 *------------------------------------------------------------------------
 */
syscall	getprio(
	  pid32		pid		/* process ID		*/
	)
{
	intmask	mask;			/* saved interrupt mask	*/
	uint32	prio;			/* priority to return	*/

	mask = disable();
	if (isbadpid(pid)) {
		restore(mask);
		return SYSERR;
	}
	prio = proctab[pid].prprio;
	restore(mask);
	return prio;
}
```

chprio()

```
/* chprio.c - chprio */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  chprio  -  Change the scheduling priority of a process
 *------------------------------------------------------------------------
 */
pri16	chprio(
	  pid32		pid,		/* ID of process to change*/
	  pri16		newprio		/* new priority		*/
	)
{
	intmask	mask;			/* saved interrupt mask	*/
	struct	procent *prptr;		/* ptr to process' table entry */
	pri16	oldprio;		/* priority to return	*/

	mask = disable();
	if (isbadpid(pid)) {
		restore(mask);
		return (pri16) SYSERR;
	}
	prptr = &proctab[pid];
	oldprio = prptr->prprio;
	prptr->prprio = newprio;
	restore(mask);
	return oldprio;
```