5

More Process Management

Chapter 4 discussed context switching and showed how processes move between the ready and current states. This chapter shows how new processes come into existence in the first place and how they eventually exit. It also introduces a new process state, *suspended*, and explores routines that move processes among the *current*, *ready*, and *suspended* states.

5.1 Process Suspension And Resumption

Having a way to temporarily stop a process from executing and then to restart it proves to be quite useful. We will say that a stopped process has been placed in a state of ‘‘suspended animation.’’ Suspended animation can be used, for example, when a process wants to wait for one of several restart conditions without knowing which will occur first.

The first step in implementing suspended animation consists of defining operations that will be used to suspend processes. In this case, the choice is obvious because only two are needed: *suspend*, to stop a process, and *resume*, to restart it.

Because suspended processes are not eligible to use the CPU, a new process state is needed to distinguish them from *ready* and *current* processes. We will call the new state *suspended*. Figure 5.1 summarizes the actions that *suspend* and *resume* perform, showing how processes move among the *ready*, *current*, and *suspended* states.

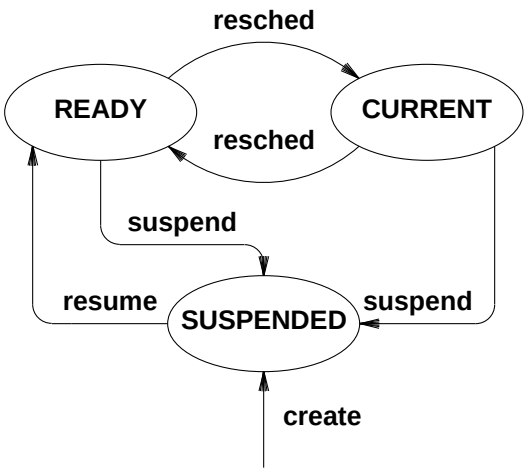

Figure 5.1 Transitions among the current, ready, and suspended states

The details of *suspend* and *resume* are obvious. *Suspend* needs an argument that specifies the process to be suspended. It must verify that the process to be suspended is either *ready* or *current*. *Resume* also needs an argument that specifies the process to be restarted; it must verify that the specified process is in the *suspended* state.

Process resumption is straightforward. *Resume* only needs to move the process back to the ready list and change its state. Suspension is not much more complex. Suspending a ready process involves changing the state recorded in its process table entry and removing the process from the ready list, so *resched* will not switch to it. The currently executing procedure can suspend itself by passing *suspend* its own process id:

```
suspend( getpid( ) );
```

Suspending the current process involves moving it to the suspended state and then rescheduling to allow another process to execute.

5.1.1 Implementation Of Resume

Procedure *resume* moves a suspended process back to the ready state where it becomes eligible for processor service. The code is contained in file *resume.c*.

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  resume  --  unsuspend a process, making it ready; return the priority
 *------------------------------------------------------------------------
 */

SYSCALL resume(pid)
int pid;
{
        int     ps;                     /* saved processor status       */
        struct  pentry  *pptr;          /* pointer to proc. tab. entry  */
        int     prio;                   /* priority to return           */

        disable(ps);
        if (isbadpid(pid) || (pptr = &proctab[pid])->pstate != PRSUSP) {
                restore(ps);
                return(SYSERR);
        }
        prio = pptr->pprio;
        ready(pid);
        resched();
        restore(ps);
        return(prio);
}
```

Although *resume* calls *ready* to link the process into the ready list, it performs important chores that *ready* does not. It checks to be sure that its argument specifies a valid process and that the referenced process is suspended. It also disables interrupts before calling *ready*. These actions make *resume* callable from a user's program.

Sometimes it is useful for the process calling *resume* to know the priority of the process it restarts, so *resume* returns that priority as its value. Care must be taken to capture the priority before the call to *ready* because the resumed process may start running when *ready* calls *resched* (before the process executing *resume* completes). The delay that occurs between the call to *ready* and the next instruction can be arbitrarily long because an arbitrary number of processes can execute before the calling process is rescheduled. To be sure that the returned priority reflects the resumed process' priority at the time of resumption, *resume* makes a copy in local variable *prio* before calling *ready* and returns the value in *prio*.

5.1.2 The Return Values SYSERR And OK

File *resume.c* includes file *kernel.h* along with *conf.h* and *proc.h*. *Kernel.h*, shown later in this chapter, defines several constants used throughout PC-Xinu, including the integers *SYSERR* and *OK*. We have already encountered SYSERR in Chapter 2 and OK in Chapter 3. By convention, PC-Xinu procedures always return SYSERR to indicate that their arguments were unacceptable or that something else prevented successful completion of the operation they tried to perform. Similarly, routines like *ready* that do not use the returned value to carry information back to the caller return the integer OK to indicate successful completion.

5.2 System Calls

The precautions that *resume* takes to verify that an operation is legal make it a general purpose routine that can be invoked by any process at any time. It is our first example of what are generally referred to as *system calls*. System calls stand between a naive user's program and the rest of the operating system, so they must protect the internal system from illegal use. As the examples in Chapter 1 illustrate, systems calls do more than merely protect the system. To the programmer, system calls define the exterior of the operating system by providing an interface through which the user accesses all system services.

When a process executes a system call like *resume* and the procedures that *resume* calls, it changes the process table and other system data structures like the *q* structure. There is only one process table in the system, shared by all processes in the system. How can a process be sure that no other process will interfere by trying to change the process table at the same time? For one thing, it must not call *resched* because rescheduling could mean a context switch to another process that needs to change the system tables. We will see that rescheduling can also result when a device interrupts because interrupt routines sometimes call *resched* as well. To prevent interrupts, *resume* invokes the procedure *disable* to disable processor interrupts. *Disable* records the current interrupt state (actually the FLAGS register contents), disables processor interrupts, and returns the recorded interrupt state. Just before leaving *resume*, the process calls the procedure *restore* to reset the interrupt state to its original value. *Resume* cannot merely enable interrupts before returning to its caller, because the caller may have been executing with *interrupts* disabled. Therefore, *resume* restores the interrupt state to its original value before returning.

5.2.1 Disable and Restore

Actually, *disable* is expanded by the C preprocessor into a call to an assembly language procedure *sys_disabl*, which records the current value of the FLAGS register, disables interrupts, and returns the recorded FLAGS register value. Accessing the FLAGS register and disabling interrupts are low-level operations that must be carried out in assembly

language. Similarly, *restore* is in reality an assembly language procedure *sys_restor*, which deposits its argument into the FLAGS register. *Disable* and *restore* are usually used in pairs, bracketing critical sections of code which must not be interrupted by other system activities.

The code for *sys_disabl* and *sys_restor* is in the file *eidi.asm*. This file also contains other low-level utility functions including *sys_enabl* for enabling interrupts, *sys_wait* for suspending the processor waiting for an interrupt, and *sys_hlt* for returning control to the host operating system.

```
; eidi.asm - _sys_disabl, _sys_enabl, _sys_restor, _sys_wait, _sys_hlt

        include dos.asm         ; segment macros

        dseg
; null data segment
        endds

        pseg

        public  _sys_disabl,_sys_restor,_sys_enabl,_sys_wait,_sys_hlt

;---------------------------------------------------------------------------
; _sys_disabl  --  return interrupt status and disable interrupts
;---------------------------------------------------------------------------
; int sys_disabl()
_sys_disabl     proc    near
        pushf                   ; put flag word on the stack
        cli                     ; disable interrupts!
        pop     ax              ; deposit flag word in return register
        ret
_sys_disabl     endp

;---------------------------------------------------------------------------
; _sys_restor  --  restore interrupt status
;---------------------------------------------------------------------------
; void sys_restor(ps)
; int ps;
_sys_restor     proc    near
        push    bp
        mov     bp,sp           ; C calling convenion
        push    [bp+4]
        popf                    ; restore flag word
        pop     bp
        ret
_sys_restor     endp
```

```
;---------------------------------------------------------------------------
; _sys_enabl  --  enable interrupts unconditionally
;---------------------------------------------------------------------------
; void sys_enabl()
_sys_enabl      proc    near
        sti                     ; enable interrupts
        ret
_sys_enabl      endp

;---------------------------------------------------------------------------
; _sys_wait  --  wait for interrupt
;---------------------------------------------------------------------------
; void sys_wait()
_sys_wait       proc    near
        pushf
        sti                     ; interrupts must be enabled here
        hlt
        popf
        ret
_sys_wait       endp

;---------------------------------------------------------------------------
; _sys_hlt  --  halt the current program and return to host
;---------------------------------------------------------------------------
; void sys_hlt()
_sys_hlt        proc    near
        mov     ah,4ch          ; terminate function
        xor     al,al           ; OK return code
        int     21h             ; MS-DOS function call
        ret
_sys_hlt        endp

        endps

        end
```

5.2.2 Implementation of Suspend

Suspending a process is not much more complex than resuming a suspended one. File *suspend.c* contains the code. *Suspend* first checks to see that argument *pid* specifies a valid process that is currently executing or ready. If the process to be suspended is in the ready state, it must be removed from the ready list and moved to the suspended state.

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  suspend  --  suspend a process, placing it in hibernation
 *------------------------------------------------------------------------
 */
SYSCALL suspend(pid)
        int     pid;                    /* id of process to suspend     */
{
        struct  pentry  *pptr;          /* pointer to proc. tab. entry  */
        int     ps;                     /* saved processor status       */
        int     prio;                   /* priority returned            */

        disable(ps);
        if (isbadpid(pid) || pid==NULLPROC ||
         ((pptr= &proctab[pid])->pstate!=PRCURR&&pptr->pstate!=PRREADY)) {
                restore(ps);
                return(SYSERR);
        }
        if (pptr->pstate == PRREADY) {
                dequeue(pid);
                pptr->pstate = PRSUSP;
        } else {
                pptr->pstate = PRSUSP;
                resched();
        }
        prio = pptr->pprio;
        restore(ps);
        return(prio);
}
```

5.2.3 Suspending The Current Process

Although suspending the currently executing process may seem odd, it turns out to be quite useful, and the code brings up two interesting points. First, the currently executing process will stop executing, at least temporarily, so it must arrange to switch to another process. To do so, it merely marks its process state suspended (*PRSUSP*) and calls *resched*. If you remember, *resched* looks at the process state to determine the disposition of the current process. In this case, it will switch context without moving the current process back onto the ready list. Second, *suspend*, like *resume*, returns the priori-

ty of the suspended process to its caller. However, *suspend* records the process priority after it has suspended the process. When a process suspends another one, the code makes perfect sense because nothing can change the priority while *suspend* executes with interrupts disabled. But when a process suspends itself, it calls *resched*, allowing other processes to execute. They may change the process' priority. When *suspend* eventually resumes executing, it reports the priority at the time of resumption. The exercises consider the motivation for this arrangement.

You may have noticed that *suspend* and *resume* do not maintain a separate linked list of suspended processes as *ready* did. The reason is simple. Ready processes are kept on an ordered list only to speed the search for the highest priority process during rescheduling. Because the system never searches through suspended processes looking for one to resume, the set of suspended processes need not be kept on a list.

5.3 Process Termination

Suspend freezes processes, but leaves them in the system so they can be resumed later. Another system call, *kill* stops a process immediately and removes it from the system completely. Once a process has been removed, it cannot be restarted because *kill* eradicates its entire record, freeing the process table entry.

The actions taken by *kill* depend on the process' state. Before writing the code, the designer needs to have some notion of the possible states and what it would mean to terminate a process in that state. For example, processes that are ready, sleeping, or waiting are all kept on linked lists in the *q* structure, so *kill* must dequeue them. If the process is waiting for a semaphore, *kill* must adjust the semaphore count as well. Not all these cases will make complete sense until you know more about the process states.

The code for *kill* appears in file *kill.c*, below. Consider how it operates for a process in the *ready* state. *Kill* checks its argument, *pid*, to ensure that it corresponds to a valid, active process by verifying that it is in the correct range, and that the process table entry is not free. It then decrements *numproc*, the global variable that records the number of active user processes. Next, *kill* calls procedure *freestk* to free memory that the process used for a stack. *Freestk* unlinks the process from the ready list with procedure *dequeue* and frees the process table entry by assigning its state field *PRFREE*. Because the process no longer appears on the ready list, it will never regain control of the CPU.

Now consider what happens when *kill* needs to terminate the currently executing process. As before, it validates its argument and decrements the count of active processes. If the current process happens to be the last user process, decrementing *numproc* makes it zero, so *kill* calls procedure *xdone*, shown below. After *kill* marks the current process' state free, it calls *resched* to pass control to another ready process.

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <sem.h>
#include <mem.h>

/*------------------------------------------------------------------------
 *  kill  --  kill a process and remove it from the system
 *------------------------------------------------------------------------
 */
SYSCALL kill(pid)
        int     pid;                    /* process to kill             */
{
        struct  pentry  *pptr;          /* points to proc. table for pid*/
        int     ps;                     /* saved processor status      */

        disable(ps);
        if (isbadpid(pid) || (pptr = &proctab[pid])->pstate==PRFREE) {
                restore(ps);
                return(SYSERR);
        }
        if (--numproc == 0)
                xdone();
        freestk(pptr->pbase, pptr->plen);
        switch (pptr->pstate) {

        case PRCURR:    pptr->pstate = PRFREE;  /* suicide */
                        resched();

        case PRWAIT:    semaph[pptr->psem].semcnt++;
                                                /* fall through */
        case PRSLEEP:
        case PRREADY:   dequeue(pid);
                                                /* fall through */
        default:        pptr->pstate = PRFREE;
        }
        restore(ps);
        return(OK);
}
```

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <io.h>
#include <disk.h>
#include <tty.h>

/*------------------------------------------------------------------------
 *  xdone  --  print system termination message and terminate PC-Xinu
 *------------------------------------------------------------------------
 */
int xdone()
{
        int     kprintf();
        int     ps;
        int     i;

#ifdef Ndsk
        for ( i=0; i<Ndsk; i++ )
                control(dstab[i].dnum,DSKSYNC); /* sync the disks       */
#endif
        sleep(1);                       /* let tty output settle        */
        disable(ps);
        maprestore();                   /* restore interrupt vectors    */
        restore(ps);
        kprintf("\n\n-- system halt --\n\n");
        if (numproc==0)
                kprintf("All user processes have completed\n");
        else
                kprintf("PC-Xinu terminated with %d process%s active\n",
                        numproc, ((numproc==1) ? "" : "es"));
        kprintf("Returning . . .\n\n");
        halt();                         /* return to caller             */
}
```

5.4 Kernel Declarations

The file *kernel.h* defines macros including *disable* and *restore* mentioned above, as well as a few other variables and symbolic constants used throughout PC-Xinu. Although not all the names that appear in it make sense yet, they will by the end of the chapter.

```
/* kernel.h - isodd,disable,restore,enable,pause,halt,xdisable,xrestore */

/* Symbolic constants used throughout Xinu */

typedef char            Bool;           /* Boolean type                 */
typedef unsigned int    word;           /* word type                    */

#define FALSE           0               /* Boolean constants            */
#define TRUE            1
#define NULL            (char *)0       /* Null pointer for linked lists*/
#define SYSCALL         int             /* System call declaration      */
#define LOCAL           static          /* Local procedure declaration  */
#define INTPROC         int             /* Interrupt procedure          */
#define PROCESS         int             /* Process declaration          */
#define WORD            word            /* 16-bit word                  */
#define MININT          0100000         /* minimum integer (-32768)     */
#define MAXINT          0077777         /* maximum integer (+32767)     */
#define MINSTK          256             /* minimum process stack size   */
#define NULLSTK         256             /* process 0 stack size         */
#define OK               1              /* returned when system call ok */
#define SYSERR          -1              /* returned when sys. call fails*/

/* initialization constants */

#define INITARGC        2               /* initial process argc         */
#define INITSTK         512             /* initial process stack        */
#define INITPRIO        20              /* initial process priority     */
#define INITNAME        "xmain"         /* initial process name         */
#define INITRET         userret         /* processes return address     */
#define INITREG         0               /* initial register contents    */
#define QUANTUM         1               /* clock ticks until preemption */

/* misc. utility functions */

#define isodd(x)        (01&(int)(x))
#define disable(x)      (x)=sys_disabl() /* save interrupt status       */
#define restore(x)      sys_restor(x)   /* restore interrupt status     */
#define enable()        sys_enabl()     /* enable interrupts            */
#define pause()         sys_wait()      /* machine "wait for interr."   */
#define halt()          sys_hlt()       /* halt PC-Xinu                 */
#define xdisable(x)     (x)=sys_xdisabl() /* save int & dosflag status  */
#define xrestore(x)     sys_xrestor(x)  /* restore int & dosflag status */

/* system-specific functions and variables */
```

```
extern  int     sys_disabl();           /* return flags & disable ints  */
extern  void    sys_restor();           /* restore the flag register    */
extern  void    sys_enabl();            /* enable interrupts            */
extern  void    sys_wait();             /* wait for an interrupt        */
extern  void    sys_hlt();              /* halt the processor           */
extern  int     sys_xdisabl();          /* Return interrupts to MS-DOS  */
extern  void    sys_restor();           /* Interrupts back to Xinu      */

/* process management variables */

extern  int     rdyhead, rdytail;
extern  int     preempt;
```

5.5 Process Creation

The system call *create* creates a new, independent process. The idea is to lay down an exact image of the process as if it had been stopped while running, so *ctxsw* can switch to it. *Create* finds a free (unused) slot in the process table, allocates space for the new process' stack, and fills in the process table entry.

A look at the code in file *create.c* explains most of the details. Procedure *newpid* searches the process table for a free process id, returning *SYSERR* if none exists. *Create* uses procedure *roundew* to round the specified stack size to the next largest even word and calls *getmem* to allocate space for the stack (Chapter 8 discusses both of these memory management routines).

We refer to the initial process stack as a *pseudo-call* because *create* carefully pushes values on it to simulate a procedure call. In C, the pseudo-call consists of arguments and a return address. When started, the new process begins executing the code for the designated procedure, obeying the normal calling conventions for accessing arguments and allocating local variables. In short, it behaves exactly as if it had been called from another procedure.

How does the designer choose a return address value to use in the pseudo call? Fortunately, there is a guideline for what happens when a process returns from its initial procedure: it should exit. *Create* makes the return address in the pseudo call the address of procedure *userret*. If the process does attempt to return from the initial procedure, control passes to *userret*. Procedure *userret* terminates the calling process with *kill*.

Create also fills in the process table entry. Knowing that *ctxsw* switches to processes by picking up register contents from its stack pointed to by the *pregs* field, *create* fills in values on the stack and sets the *pregs* entry to point to the ''top of stack.'' The values for the SI, DI, and BP registers are immaterial, but the value for the FLAGS register is important. Because the process should begin execution with interrupts enabled, the FLAGS register should have its interrupt bit set. *Create* makes the state of the newly created process *PRSUSP*, leaving it suspended, but otherwise ready to run. Final-

ly, *create* returns the process id of the newly created process. The id must be passed to *resume* to start the new process executing.

Many of the process initialization details depend on the C run-time environment – there is simply no way to start a process without facing such details. For example, *create* pushes arguments onto the process stack so the first argument is near the top of the stack. The code that pushes arguments is difficult to understand because *create* copies those arguments directly from its own run-time stack onto the stack that it has allocated for the new process. To do so, it finds the address of the arguments on its own stack and moves through the list using pointer arithmetic. This is clearly a machine (and compiler) dependent trick.

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>
#include <mem.h>

#define INITF   0x0200  /* initial flag register - set interrupt flag,  */
                        /* clear direction and trap flags               */

extern  int     INITRET();      /* location to return upon termination  */

/*------------------------------------------------------------------------
 *  create  --  create a process to start running a procedure
 *------------------------------------------------------------------------
 */
SYSCALL create(procaddr,ssize,priority,namep,nargs,args)
int (*procaddr)();              /* procedure address                    */
word ssize;                     /* stack size in words                  */
short priority;                 /* process priority > 0                 */
char *namep;                    /* name (for debugging)                 */
int nargs;                      /* number of args that follow           */
int args;                       /* arguments (treated like an array)    */
{
        int     pid;                    /* stores new process id        */
        struct  pentry  *pptr;          /* pointer to proc. table entry */
        int     i;                      /* loop variable                */
        int     *a;                     /* points to list of args       */
        char    *saddr;                 /* start of stack address       */
        int     *sp;                    /* stack pointer                */
        int     ps;                     /* saved processor status       */

        disable(ps);
        ssize = roundew(ssize);
        if ( ssize < MINSTK || priority < 1 ||
                (pid=newpid()) == SYSERR ||
                ((saddr=getstk(ssize)) == NULL ) ) {
                restore(ps);
                return(SYSERR);
        }
        numproc++;
        pptr = &proctab[pid];
        pptr->pstate = PRSUSP;
        for (i=0 ; i<PNMLEN ; i++)
```

```
                pptr->pname[i] = (*namep ? *namep++ : ' ');
        pptr->pname[PNMLEN]='\0';
        pptr->pprio = priority;
        pptr->phasmsg = 0;              /* no message                   */
        pptr->pbase = saddr;
        pptr->plen = ssize;
        sp = (int *) (saddr+ssize);     /* simulate stack pointer       */
        sp -= 4;                        /* a little elbow room          */
        pptr->pargs = nargs;
        a = (&args) + nargs;            /* point past last argument     */
        for ( ; nargs > 0 ; nargs--)    /* machine dependent; copy args */
                *(--sp) = *(--a);       /*  onto created process' stack */
        *(--sp) = (int)INITRET;         /* push on return address       */
        *(--sp) = (int)procaddr;        /* simulate a context switch    */
        --sp ;                          /* 1 word for bp                */
        *(--sp) = INITF;                /* FLAGS value                  */
        sp -= 2;                        /* 2 words for si and di        */
        pptr->pregs = (char *)sp;       /* save for context switch      */
        pptr->paddr = procaddr;
        restore(ps);
        return(pid);
}

/*------------------------------------------------------------------------
 *  newpid  --  obtain a new (free) process id
 *------------------------------------------------------------------------
 */
LOCAL   newpid()
{
        int     pid;                    /* process id to return         */
        int     i;

        for (i=0 ; i<NPROC ; i++) {     /* check all NPROC slots        */
                if ( (pid=nextproc--) <= 0)
                        nextproc = NPROC-1;
                if (proctab[pid].pstate == PRFREE)
                        return(pid);
        }
        return(SYSERR);
}
```

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

#include <conf.h>
#include <kernel.h>

/*------------------------------------------------------------------------
 *  userret  --  entered when a process exits by return
 *------------------------------------------------------------------------
 */
userret()
{
        int     pid;

        kill( pid=getpid() );
        kprintf("Fatal system error - unable to kill process %d",pid);
}
```

5.6 Utility Procedures

Three additional system calls help manage processes: *getpid*, *getprio*, and *chprio*. *Getpid* allows a process to obtain its process id. *Userret* shows one reason a procedure may need to know the id of the process executing it. *Getprio* allows a process to obtain a process' scheduling priority. Another useful system call, *chprio*, allows a process to change a process' priority. The implementation of all three routines is exceedingly simple:

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  getprio  --  return the scheduling priority of a given process
 *------------------------------------------------------------------------
 */
SYSCALL getprio(pid)
        int     pid;
{
        struct  pentry  *pptr;
        int     ps;

        disable(ps);
```

```
        if (isbadpid(pid) || (pptr = &proctab[pid])->pstate == PRFREE) {
                restore(ps);
                return(SYSERR);
        }
        restore(ps);
        return(pptr->pprio);
}
```

After checking its argument, *getprio* extracts the scheduling priority for the specified process from the process table entry and returns the priority to the caller.

```
/* getpid.c - getpid */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  getpid  --  get the process id of currently executing process
 *------------------------------------------------------------------------
 */
SYSCALL getpid()
{
        return(currpid);
}
```

It may seem that procedure *getpid* is useless because it returns the value of variable *currpid*, a value that the process could obtain directly with less overhead. Why not have processes access *currpid*? If PC-Xinu is transported to a machine in which user processes cannot access the address space occupied by the system, it may not be possible for a user process to obtain the value of *currpid* directly.

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  chprio  --  change the scheduling priority of a process
 *------------------------------------------------------------------------
 */
SYSCALL chprio(pid,newprio)
        int     pid;
        int     newprio;                /* newprio > 0                  */
{
        int     oldprio;
        struct  pentry  *pptr;
        int     ps;

        disable(ps);
        if (isbadpid(pid) || newprio<=0 ||
            (pptr = &proctab[pid])->pstate == PRFREE) {
                restore(ps);
                return(SYSERR);
        }
        oldprio = pptr->pprio;
        pptr->pprio = newprio;
        restore(ps);
        return(oldprio);
}
```

This implementation of *chprio* seems to do exactly what is needed. It checks to be sure the specified process exists before changing the priority field in its process table entry. As the exercises point out, however, it contains a serious flaw.

5.7 Summary

This chapter has expanded the ideas of process management by discussing how to add another layer of software on top of the scheduler and context switch. The new layer includes routines to *suspend* and *resume* execution, as well as routines that *create* processes and *kill* them. Finally, we looked at three utility procedures that obtain a process' identifier (*getpid*), obtain a process' scheduling priority (*getprio*), or change a process' priority (*chprio*). Despite its brevity, the code built thus far forms the basis of a process manager. With proper initialization and a few support routines, it will multiplex the CPU among multiple computations.

The following chapters discuss the design of synchronization and interprocess communication, components that are built on top of the existing layers. Chapter 8 will discuss the only low-level layer that we have ignored so far, the memory manager. Following that, we will continue the pattern of building layers, one on top of the other.

FOR FURTHER STUDY

Primitives for process creation and management vary widely among systems. Calingaert [1982] includes a good survey of process creation techniques. Knuth [1968] gives the details of coroutine activation; Ritchie and Thompson [1974] describe the *fork* primitive used to create processes in the UNIX system.

EXERCISES

5.1 Processes can tell which of several events triggered their resumption if their priority is set to a unique value before each call to *resume*. Use this method to create a process that suspends itself and determines which of two other processes resumes it first.

5.2 Why does *create* build a pseudo-call that returns to *userret* at process exit instead of one that calls *kill* directly?

5.3 Modify *create* to call *kill* directly, pushing the process id onto the stack as an argument to *kill*.

5.4 One alternative to pushing the new process' parameters directly onto its stack is to have the caller construct an array of argument pointers, and have *create* copy pointers from the list onto the new process' stack. Show how to implement this approach, being careful to take into account a variable number of parameters.

5.5 Global variable *numproc* tells the number of active user processes. Considering the code in *kill*, can you tell whether the count in *numproc* should include the null process or not?

5.6 Find out how the *trap* or *emt* instructions can be used to pass control to a system call on the PDP-11. What are the disadvantages of this mechanism? Hints: think of the memory needed and the number of possible system calls.

5.7 *Create* leaves the new process suspended instead of running. Why?

5.8 *Ctxsw* is always called with interrupts disabled, and when it eventually returns to its caller, interrupts will continue to be disabled. Can you save time in *ctxsw* by omitting the *pushf* and *popf* instructions?

5.9 Procedure *resume* saves the resumed process' priority in a local variable before calling *ready*. Show that if it referenced *pptr->prio* after the call to *ready*, *resume* could return a priority value that the resumed process never had (not even after resumption).

5.10 In procedure *newpid*, the variable *nextproc* is a global integer that tells the next process table slot to check for a free one. Starting the search from where it left off eliminates looking past the used slots again and again. Speculate on whether the technique is worthwhile.

5.11 *Getpid* simply returns the value of *currpid* to the caller. Discuss reasons for hiding the operating system internals with system calls. Find out about address space mapping on a machine more complex than an 8088, and consider why it might be necessary to have a system call instead of allowing user processes to access *currpid* directly.

5.12 Procedure *chprio* contains a serious design flaw. Find it, describe its consequences, and repair it.