4

Scheduling and Context Switching

An operating system achieves the illusion of concurrent processing by rapidly switching one processor among several computations. Because the speed of the computation is extremely fast compared to that of a human, the effect is impressive – many activities appear to proceed simultaneously.

Context switching lies at the heart of the process juggling act. It consists of stopping the current computation, saving enough information so it may be restarted later, and restarting another process. What makes such a change difficult is that the CPU cannot be stopped at all – it must continue to execute the code that switches to a new process.

This chapter describes the basic context switching mechanism, showing exactly how a process saves its state information, chooses another process to run from among those that are ready, and relinquishes control to that process. It describes the data structure that holds information about processes while they are not executing and shows how the context switch uses that data structure. For the present, we ignore the questions of when or why processes choose to switch context. Later chapters address these issues, showing how higher layers of the system use the context switch built here.

4.1 The Process Table

The system keeps all information about processes in a data structure called the *process table*. There is one entry in the process table for each process. Because exactly one process is running at any time, one of those entries corresponds to an active process – its saved state information is out of date. All other process table entries contain information about processes that have been stopped temporarily. To switch context the operating system saves information about the currently running process in its process table entry and

restores information from the process table entry corresponding to the process it is about to execute.

Exactly what information must be saved in the process table? The system must save any values that will be destroyed when the new process runs. For example, in PC-Xinu each process has its own separate stack memory, so a copy of the stack need not be saved. (The new process may change the machine registers when it executes, but these registers will be saved on the stack for that process.) In addition to data that must be reloaded when it resumes a process, the system also keeps information in the process table that it uses to control processes and account for their resources. These details will become clear as we see how the process table is used.

The PC-Xinu process table, *proctab*, is an array with entries for up to *NPROC* processes. It is declared in file *proc.h*, below. Each entry in *proctab* is a structure named *pentry* that defines the information kept for each process.

```
/* proc.h - isbadpid */

/* process table declarations and defined constants                    */

#ifndef NPROC                           /* set the number of processes  */
#define NPROC           30              /*  allowed if not already done */
#endif

/* process state constants */

#define PRCURR          '\01'           /* process is currently running */
#define PRFREE          '\02'           /* process slot is free         */
#define PRREADY         '\03'           /* process is on ready queue    */
#define PRRECV          '\04'           /* process waiting for message  */
#define PRSLEEP         '\05'           /* process is sleeping          */
#define PRSUSP          '\06'           /* process is suspended         */
#define PRWAIT          '\07'           /* process is on semaphore queue*/

/* miscellaneous process definitions */

#define PNMLEN          9               /* length of process "name"     */
#define NULLPROC        0               /* id of the null process; it   */
                                        /*  is always eligible to run   */

#define isbadpid(x)     (x<=0 || x>=NPROC)

/* process table entry */

struct  pentry  {
        char    pstate;                 /* process state: PRCURR, etc.  */
```

```
        int     pprio;                  /* process priority             */
        int     psem;                   /* semaphore if process waiting */
        int     pmsg;                   /* message sent to this process */
        int     phasmsg;                /* nonzero iff pmsg is valid    */
        char    *pregs;                 /* saved regs. (SP)             */
        char    *pbase;                 /* base of run time stack       */
        word    plen;                   /* stack length                 */
        char    pname[PNMLEN+1];        /* process name                 */
        int     pargs;                  /* initial number of arguments  */
        int     (*paddr)();             /* initial code address         */
};

extern  struct  pentry proctab[];
extern  int     numproc;                /* currently active processes   */
extern  int     nextproc;               /* search point for free slot   */
extern  int     currpid;                /* currently executing process  */
```

Throughout PC-Xinu, each process is identified by an integer. The following rule gives the relationship between those integers and the process table:

> *Processes are referenced by their process id, which is the index of the saved state information in proctab.*

Only the *pstack* field of the PC-Xinu process table entry contains information needed to restart the process; other fields contain information for bookkeeping and error checking. For example, the fields *pbase*, *plen*, *pargs*, and *pname* contain the address of the process stack, the length of the stack, the number of arguments passed to the process when it was created, and a character string identifying the process. Some of these values are used to free memory when a process completes; others are merely for debugging.

4.2 Process States

The system uses the *pstate* field of the process table to help it keep track of what the process is doing and, consequently, the validity and semantics of operations performed on it. The system designer must evolve this set of process states as the initial design proceeds. The set should be well-defined before implementation begins because many of the routines that manipulate processes base their actions on the process' state, requiring the programmer to carefully consider each case.

In PC-Xinu, the following six states are used: *current*, *ready*, *receiving*, *sleeping*, *suspended*, and *waiting*. File *proc.h* contains symbolic constants for each of these that are used throughout the code: *PRCURR*, *PRREADY*, *PRRECV*, *PRSLEEP*, *PRSUSP*, and *PRWAIT*. In addition to the above values, the *pstate* field contains *PRFREE* when no process is using that process table entry. Later, we will explore each state in detail, see-

ing why they arose and how a process moves between them. Only the current and ready states concern us at this time.

4.3 Selecting A Ready Process

Almost every system needs *ready* and *current* process states. Processes are classified *ready* when they are eligible for CPU service but are not currently executing; the single process receiving CPU service is classified as *current*. Switching context consists of two things: selecting a process from among those that are ready (or current), and giving control of the CPU to the selected process. Software that implements the policy used to select a process from among those that are ready to run is called a *scheduler*. In PC-Xinu, procedure *resched* makes that selection according to the following well-known scheduling policy:

> *At any time, the highest priority process eligible for CPU service is executing. Among processes with equal priority, scheduling is round-robin.*

Round-robin means that processes are selected one after another so that all members of the set have an opportunity to execute before any member has a second opportunity. Priorities, kept in the *pprio* field of the process table entry, are nothing more than positive integers that give the user some control over how processes are selected for CPU service. (More complex systems adjust priorities from time-to-time, based on observed behavior of the process.)

To make the selection of a new process faster, all ready processes appear in a list ordered by priority, such that the highest priority process is immediately accessible. *Resched* uses the queue mechanisms described in Chapter 3 to examine and update that list. It uses process priorities as keys and keeps the list ordered by key, so highest priority processes are found at the tail. Global variables *rdyhead* and *rdytail* point to the head and tail of the ready list in the *q* structure. Whether the current process should also be kept on the ready list is determined largely by the details of each implementation, but the entire system must be designed to obey the same rule. In PC-Xinu,

> *The current process does not appear on the ready list, but its process id is always given by the global integer variable currpid.*

Consider what happens to the currently executing process during a context switch. Often, the currently executing process remains eligible to use the CPU even though it must temporarily pass control to another process. In such situations, the context switch must change the current process' state to *PRREADY* and move it onto the ready list, so it will be considered for CPU service again later.

How does the rescheduler, *resched*, decide whether to move the current process onto the ready list? It does not receive an explicit parameter telling the disposition of the current process. Instead, the system routines cooperate to save the current process in the following way: if the currently executing process will not remain eligible to use the CPU, system routines assign to the current process' *pstate* field the desired next state before calling *resched*. Whenever *resched* prepares to switch context, it checks *pstate* for the current process and makes it ready only if the state still indicates *PRCURR*.

In some situations it is necessary to suspend rescheduling temporarily while critical system activities are taking place. Suspension of rescheduling makes it possible for one PC-Xinu process to have exclusive use of the CPU even when interrupts are enabled. The procedure *sys_pcxget* returns a nonzero value if rescheduling is permitted and returns zero otherwise. If the current process calls *resched* when rescheduling is not permitted, the procedure returns immediately. Since *any* return from *resched* must leave the process in the current state, it is an error if a process enters *resched* when rescheduling is suspended and the process is not the current process. Consequently, the system *panic* procedure is called to halt PC-Xinu if this error occurs. Suspension of rescheduling is described in more detail in Chapter 9.

In addition to moving the current process to the ready list, *resched* completes every detail of scheduling and context switching except saving and restoring machine registers and switching stacks (which cannot be done directly in a high-level language like C). It selects a new process to run, changes the process table entry for the new process, removes the new process from the ready list, marks it current, and updates *currpid*. It also resets the preemption counter, something we will consider later. Finally, it calls *ctxsw* to save the current registers, switch stacks, and restore the registers for the new process. The code is shown below.

```
/* resched.c - resched */

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

/*------------------------------------------------------------------------
 *  resched  --  reschedule processor to highest priority ready process
 *
 * Notes:       Upon entry, currpid gives current process id.
 *              Proctab[currpid].pstate gives correct NEXT state for
 *                      current process if other than PRCURR.
 *------------------------------------------------------------------------
 */
int     resched()
{
        register struct pentry  *optr;  /* pointer to old process entry */
        register struct pentry  *nptr;  /* pointer to new process entry */

        optr = &proctab[currpid];
        if ( optr->pstate == PRCURR ) {
                /* no switch needed if current prio. higher than next   */
                /* or if rescheduling is disabled ( pcxflag == 0 )      */
                if ( sys_pcxget() == 0 ||  lastkey(rdytail) < optr->pprio )
                        return;
                /* force context switch */
                optr->pstate = PRREADY;
                insert(currpid,rdyhead,optr->pprio);
        } else if ( sys_pcxget() == 0 ) {
                kprintf("pid=%d state=%d name=%s",
                        currpid,optr->pstate,optr->pname);
                panic("Reschedule impossible in this state");
        }

        /* remove highest priority process at end of ready list */

        nptr = &proctab[ (currpid = getlast(rdytail)) ];
        nptr->pstate = PRCURR;          /* mark it currently running    */
        preempt = QUANTUM;              /* reset preemption counter     */
        ctxsw(&optr->pregs,&nptr->pregs);

        /* The OLD process returns here when resumed. */
        return;
}
```

Resched uses procedure *ctxsw* to save the process state and change stacks because registers and the stack pointer cannot be manipulated in a high-level language. The code for *ctxsw* is, of course, machine dependent. When *ctxsw* switches processes the FLAGS register must be saved, since it contains the interrupt state of the process. The other registers that must be saved are BP, SI and DI, since C procedures assume that these will not change across procedure calls. *Ctxsw* saves these registers on the stack in exactly the same way as the assembly language programs in Chapter 2. Obviously, the stack pointer must be changed only after these registers have been preserved because as soon as the stack pointer changes, the CPU will be using the stack of the new process, wherever that happens to be. From this point on, the processor is working with the stack of the new process, and all subsequent operations refer to this new stack. The DI, SI, FLAGS and BP registers for the new process are loaded from its stack. Upon return from *ctxsw*, the instruction pointer is set to the code for the new process, and the machine resumes executing instructions associated with the new process.

The parameters passed to *ctxsw* are pointers to the *pregs* fields of the current and new process table entries. The pointer to the *pregs* field of the new process is used to obtain the stack environment of the new process, while that of the current process is used to store the saved stack environment of the current process.

```
; ctxsw.asm - _ctxsw

        include dos.asm

        dseg
; null data segment
        endds

        pseg

        public  _ctxsw

;---------------------------------------------------------------------------
; _ctxsw  --  context switch
;---------------------------------------------------------------------------
; void ctxsw(opp,npp)
; char *opp, *npp;
;---------------------------------------------------------------------
; Stack contents upon entry to ctxsw:
;       SP+4 => address of new context stack save area
;       SP+2 => address of old context stack save area
;       SP   => return address
; The addresses of the old and new context stack save areas are
; relative to the DS segment register, which must be set properly
; to access the save/restore locations.
;
; The saved state consists of the current BP, SI and DI registers,
; and the FLAGS register
;---------------------------------------------------------------------
_ctxsw  proc    near
        push    bp
        mov     bp,sp           ; frame pointer
        pushf                   ; flags save interrupt condition
        cli                     ; disable interrupts just to be sure
        push    si
        push    di              ; preserve registers
        mov     bx,[bp+4]       ; old stack save address
        mov     [bx],sp
        mov     bx,[bp+6]       ; new stack save address
        mov     sp,[bx]
        pop     di
        pop     si
        popf                    ; restore interrupt state
        pop     bp
```

```
        ret
_ctxsw  endp

        endps

        end
```

The code in *ctxsw* reveals how to resolve the dilemma caused by trying to save registers while a process is still using them. Think of an executing process that has called *resched*, which in turn has called *ctxsw*. Instead of trying to save registers explicitly as the process executes, *ctxsw* captures the value of the stack pointer precisely when the registers (including the instruction pointer and FLAGS) are already on the stack as a result of the code in *ctxsw*. This freezes the stack of the process as if it were in the midst of executing a normal procedure. Then *ctxsw* restores the stack pointer to that of another frozen process; *ctxsw* restores the registers and returns normally to resume execution of the other process.

It is interesting to note that all processes call *resched* to perform context switching, and *resched* calls *ctxsw*, so all suspended processes will resume at the same place – just after the call to *ctxsw* in *resched*. Each process has its own stack of procedure calls, however, so the return from *resched* will take them in various directions. Note also that if the two pointers passed to *ctxsw* are equal – for example, if a process could perform a context switch to itself – then *ctxsw* will simply return to the caller with no change.

Building all procedures to return to their caller is a key ingredient in keeping the system design clean. It would be impossible unless the scheduler returned to its caller. So, both *resched* and *ctxsw* have been designed to behave just like any other procedures – they eventually return. Of course, there may be considerable delay before a call to *resched* returns because the CPU may execute other processes arbitrarily long before restarting the calling process (depending on the process priorities).

4.4 The Null Process

Resched only switches the processor among the current and ready processes; it does not create new processes. It assumes that at least one process is available and does not bother to verify whether the ready list is empty. There is a strong consequence:

> *Resched can only switch context from one process to another, so at least one process must always remain ready to run.*

To insure that a ready process always exists, PC-Xinu creates an extra process, called the *null process*, when it initializes the system. The null process has process id zero and priority zero; its code, which will be shown in Chapter 13, consists of an infinite loop. Because user processes must have a priority greater than zero, the scheduler switches to the null process only when no user process remains ready to run.

4.5 Making A Process Ready

When *resched* needed to move the current process onto the ready list, it manipulated the list directly. Making a process eligible for CPU service occurs so frequently that we have invented a procedure to do just that. It is named *ready*:

```
/* ready.c - ready */

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

/*------------------------------------------------------------------------
 *  ready  --  make a process eligible for CPU service
 *------------------------------------------------------------------------
 */
int     ready (pid)
        int     pid;                    /* id of process to make ready  */
{
        register struct pentry  *pptr;

        if (isbadpid(pid))
                return(SYSERR);
        pptr = &proctab[pid];
        pptr->pstate = PRREADY;
        insert(pid,rdyhead,pptr->pprio);
        return(OK);
}
```

Usually, a procedure which calls *ready* needs to call *resched* after placing the process on the ready list to ensure that the CPU is executing the highest priority ready process. When the caller needs to move several processes from some other list onto the ready list, rescheduling after each call to *ready* can be time-consuming. The answer is to suspend temporarily the calls to *resched* and call *ready* several times. Then, after all processes have been moved and the list manipulation is complete, a call to *resched* reinstates the policy by assuring that the highest priority ready process is currently executing. We will see an example of delayed rescheduling in Chapter 6.

4.6 Summary

Scheduling and context switching are closely related activities that make concurrent execution possible. Scheduling consists of choosing a process from among those that are eligible for execution. Context switching consists of stopping one process and starting a new one. To keep track of the processes, the system uses a global data structure called the process table. Whenever the scheduler temporarily suspends a process, it saves all pertinent information about that process in its process table entry, along with a value that indicates the process state. This chapter considered procedures *ready*, *resched*, and *ctxsw* that performed transitions between the *current* and *ready* states.

FOR FURTHER STUDY

Many scheduling algorithms have been devised and analyzed. Coffman and Denning [1973] contains a formal treatment of the subject. Less formal are Bull and Packham [1971] and Bunt [1976]. Lampson [1968] discusses scheduling based on priorities.

Other books, by Habermann [1976], Calingaert [1982], and Bic and Shaw [1988] emphasize the consequences of various scheduling schemes.

EXERCISES

4.1 Write a test program to show that a call to *ctxsw* which passes the same pointer in both parameters simply returns to the caller.

4.2 Identify fields in the process table used only for error checking.

4.3 Investigate another processor (e.g., the Motorola 68000 or the PDP-11 series), and determine what information needs to be saved during a context switch.

4.4 Write *ctxsw* for another processor, trying to minimize the number of instructions.

4.5 Because each process has its own process table entry, information like register values could be saved in the process table entry rather than the process stack during a context switch. Does this reduce or increase the amount of assembly code required? What are the advantages of each approach?

4.6 It would be possible for *ctxsw* to reach into the run-time stack to obtain enough information to have the process resume in the routine that called *resched* instead of in *resched*. What are the advantages and disadvantages of doing so?

4.7 The C compiler used to build PC-Xinu does not require registers AX, BX, CX, DX and ES to be saved and restored across procedure calls. Suggest a way to do context switching which saves these registers as well. Give reasons for adopting such an approach. Redesign *ctxsw* to save these registers as well as BP, the FLAGS, SI, and DI in the process table entry.

4.8 Rewrite *resched* to have an explicit parameter giving the disposition of the currently executing process. Does it require more time?

4.9 Describe the relationship between coroutines and context switching in this chapter.

4.10 Implement stack checking by placing an uncommon value at the base of each process stack. Have *resched* check both the current process stack size and the value at the base of the process stack before the process resumes. If either indicate stack overflow, print an error message and halt.