10

Real-Time Clock Management

A *clock* is a hardware device that emits pulses, usually square waves, at regular intervals with high precision. Besides the central system clock that controls the rate at which the CPU executes instructions, computer systems may have a *real-time clock* and a *time-of-day clock*, the two being related, but not identical.

Like a digital wristwatch, a time-of-day clock is a chronometer. It consists of an accurate clock that pulses an integral number of times per second and a counter to tally the pulses. Programs read the counter to determine the current time and date, and privileged programs write into the counter to set the time. Resetting is rarely needed because time-of-day clocks continue counting correctly as long as they receive power, independent of whether the CPU is heavily loaded or halted. (Stories abound about the confusion introduced by computer operators who set the time-of-day clock inaccurately after a power failure.)

10.1 The Real-Time Clock Mechanism

Unlike the time-of-day clock, a real-time clock does not tally pulses or keep track of the date. Instead, it pulses regularly a number of times each second, signaling the CPU each time a pulse occurs, by posting an interrupt. Thus, one distinction between the two clocks is based on whether the clock controls the CPU or the CPU controls the clock:

> *The CPU reads the time-of-day clock whenever it wants to obtain the current date and time; the real-time clock forces the CPU to process an interrupt each time it pulses.*

The two clocks are further distinguished by whether they count pulses. The real-time clock does not contain a counter, and it does not accumulate interrupts. Responsibility for counting interrupts falls on the system:

> *If the CPU takes too long to service a real-time clock interrupt, or if it operates with interrupts disabled for more than one clock cycle, it will miss the interrupt.*

Obviously, systems must be designed to service clock interrupts quickly. The hardware helps by giving highest priority to clock interrupts. Some clock hardware buffer a few interrupts allowing the CPU to delay more than one cycle. Even so, some slow processors cannot afford to call a C procedure on each clock interrupt, or they would spend most of their time handling clock interrupts.

10.2 PC Real-Time Clock Interrupts

The typical PC processor clock operates at 4.77272MHz. Circuitry on the processor board divides the rate by 4, producing a 1.19318MHz signal for the 8253 timer chip. The timer chip, in turn, divides the signal by 65536 to produce a rate of 18.20648Hz, the rate of the real time clock interrupt. The clock interrupts through vector *CLK_INT* (type 08H) and is dispatched through the *intmap* table described in Chapter 9.

Certain systems have a faster real-time clock – that is, a more frequent interrupt rate. Such systems cannot afford to call a C procedure on each clock interrupt, or they would spend most of their time handling clock interrupts. How can such a system avoid spending all its time processing real-time clock interrupts? The answer is that the clock rate must be adjusted to match the system. Slowing the clock is a significant optimization because it permits the designer to build rich functionality by sacrificing some precision.

Ideally, the hardware clock should be slowed, but this is usually inconvenient or impossible. Instead, a special-purpose clock interrupt handler can be designed to simulate a slower-rate clock. The easiest way to simulate a slow clock consists of dividing the clock rate. For example, the real-time clock on the Digital Equipment Corporation LSI 11/2 – the original processor Xinu was designed for – generates 60 pulses per second, a rate that would require much time to handle. To simulate a slower clock, the clock handler can be programmed to ignore five clock interrupts before processing one, effectively dividing the rate by 6. In practice, the clock handler must accept all clock interrupts, but it merely decrements a counter and returns quickly on five out of six. As a result, the main body of the LSI 11/2 clock interrupt handling code is executed only 10 times per second.

The rate at which the main clock interrupt handling is performed is affectionately known as the *tick rate*, and we say a *tick* occurs each time this main clock interrupt handler is called. In PC-Xinu, the main clock interrupt code is performed on each clock interrupt, so its tick rate is the same as the real-time clock interrupt rate – approximately

18.2Hz. While this rate is slow enough to avoid a significant interrupt-handling overhead, the rate is an unfortunate choice since it is not an integral number of ticks per second. This rate makes it difficult to handle activities which require a resolution measured in seconds or multiples of seconds.

10.3 The Use Of A Real-Time Clock

Operating systems use real-time clocks internally to limit the amount of time a process can execute, as well as externally, to provide user programs with services like timed delays. Usually, the system maintains a list of ''events,'' ordered by the time at which they should occur. Whenever the real-time clock interrupts, it examines the event list and initiates any events for which the delay has expired.

Our design allows two kinds of events to be scheduled for the future. The first is *preemption*: when granting CPU service to a process, the system schedules a preemption event to prevent the process from running forever. The second is a *timed delay*: when a process requests a timed delay, it removes the process from the current state and schedules a *wakeup* event to restart it at the correct time.

The system uses preemption to guarantee that equal priority processes receive service round-robin (specified by the scheduling policy in Chapter 4). Recall that whenever *resched* switches context, it resets the variable *preempt* to *QUANTUM* (*QUANTUM* is a symbolic constant defined in file *kernel.h*). The clock interrupt dispatcher decrements *preempt* on each clock tick, calling *resched* when it reaches zero. The currently executing process is always the highest priority process that is eligible for CPU service, but others of equal priority may be waiting on the ready list. If they are, *resched* places the current process on the ready list behind other processes with equal priority, and it and switches to the first process on the list. Thus, if *k* equal-priority processes all need CPU service, all *k* execute for, at most, *QUANTUM* clock ticks before any of them receive more service.

The value of symbolic constant *QUANTUM* gives the *granularity of preemption*; it can be changed before the system is compiled. Setting *QUANTUM* small, say 2 or 3, makes the granularity small, by rescheduling every few tenths of a second. Small granularity tends to keep all equal priority processes proceeding at approximately the same pace. But a small granularity introduces much overhead because it forces the clock interrupt routine to call *resched* often. Setting *QUANTUM* to a large value, say 100, reduces the overhead of context switching, but makes the granularity of switching large. The potential disadvantage of large granularity is that a process may execute many seconds before switching to another of equal priority.

As it turns out, processes seldom use the CPU long enough to warrant preemption. A process voluntarily calls *resched* by executing system routines like *wait* or doing input and output (I/O). Because input and output are slow compared to processing, processes spend most of their time waiting for devices. However, the system could never regain control from a process that executed an infinite loop without a preemptive capability, so it is important to include it in any system that supports multiprogramming.

Systems also use the real-time clock to honor requests for timed delays. For example, when the currently executing process requests a delay, PC-Xinu moves it to a list of ''sleeping'' processes, arranging to have it awakened after the appropriate number of clock ticks. On each clock tick, the clock interrupt routine checks sleeping processes and moves those that have been delayed the specified time to the ready list. Routines to handle such delays will be considered next.

10.4 Delta List Processing

Because it cannot afford to search through arbitrarily long lists of sleeping processes to find those that should awaken on each clock tick, the system keeps sleeping processes in a data structure called a *delta list*. Like other lists of processes, the sleeping process delta list resides in the *q* structure. Variable *clockq* contains the *q* index of its head. On each clock tick, the clock interrupt dispatcher examines the first process in *clockq* and calls the high-level interrupt routine *wakeup* to awaken processes if their time delay has expired.

Unlike other lists in the *q* structure, *the delta list* is neither ordered by increasing key, nor FIFO. Instead, keys record successive deltas (differences) in delay:

> *Processes on clockq are ordered by the time at which they will awaken; each key tells the number of clock ticks that the process must delay beyond the preceding one on the list.*

The first process on the list is the one with least delay, and its key gives the remaining delay in clock ticks until it must awaken. The delta organization permits the clock interrupt routine to decrement the first key on each clock tick without scanning the list because the remaining delays are relative to it. For example, if four processes need to delay 17, 27, 28, and 32 ticks, then their keys on the delta list contain 17, 10, 1, and 4. Given only the delta list, partial sums of keys give the total delay before processes awaken. The total delay before the first process awakens is 17, the total for the second is 17+10, the total for the third is 17+10+1, and the total for the last is 17+10+1+4.

Routines to manipulate delta lists are easy to design, but the details can be tricky; close scrutiny of the code is worthwhile. Procedure *insertd*, shown below, inserts a process *pid* in *clockq*, given its delay in parameter *key*. As with priority queues, the *qkey* field of each node on the list records the key value for that node. In the code, variable *next* scans the list searching for the place to insert the new process.

Keys in the delta list specify delays relative to their predecessor; they cannot be compared directly to the initial value of *key*, which specifies a delay relative to the current time. To keep the delays comparable, *insertd* subtracts the relative delays from *key* as the search proceeds, maintaining the following invariant:

At any time during the search, both key and q[next].qkey specify a delay relative to the time at which the predecessor of the ‘‘next’’ awakens.

Insertd inserts the new process at the point where its relative delay is less than the relative delay of those left on the list. Note that *insertd* does not have to explicitly check for the end of the list, because the key value in the tail forces an insertion. After linking process *pid* into the list, *insertd* subtracts the extra delay that it introduces from the delay of the next process.

```
/* insertd.c - insertd */

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

/*------------------------------------------------------------------------
 *  insertd  --  insert process pid in delta list "head", given its key
 *------------------------------------------------------------------------
 */
INTPROC insertd(pid, head, key)
        int     pid;
        int     head;
        int     key;
{
        int     next;                   /* runs through list            */
        int     prev;                   /* follows next through list    */

        for(prev=head,next=q[head].qnext ;
            q[next].qkey < key ; prev=next,next=q[next].qnext)
                key -= q[next].qkey;
        q[pid].qnext = next;
        q[pid].qprev = prev;
        q[pid].qkey  = key;
        q[prev].qnext = pid;
        q[next].qprev = pid;
        if (next < NPROC)
                q[next].qkey -= key;
        return(OK);
}
```

10.5 Putting A Process To Sleep

User programs do not usually access the real-time clock queue directly; they call system routines that provide delays. System call *sleept(n)* delays the calling process *n* ticks. It does so by inserting the process into the delta list of sleeping processes.

When a process is moved to the list of sleeping processes, it is no longer *ready* or *current*. In what state should it be placed? Sleeping processes differ from processes that are suspended, waiting to receive messages, or waiting for semaphores, so none of these states suffices. It is time to add a new process state to the design; we will call it *sleeping* and denote it with symbolic constant *PRSLEEP*. The new diagram of process state transitions is shown in Figure 10.1.

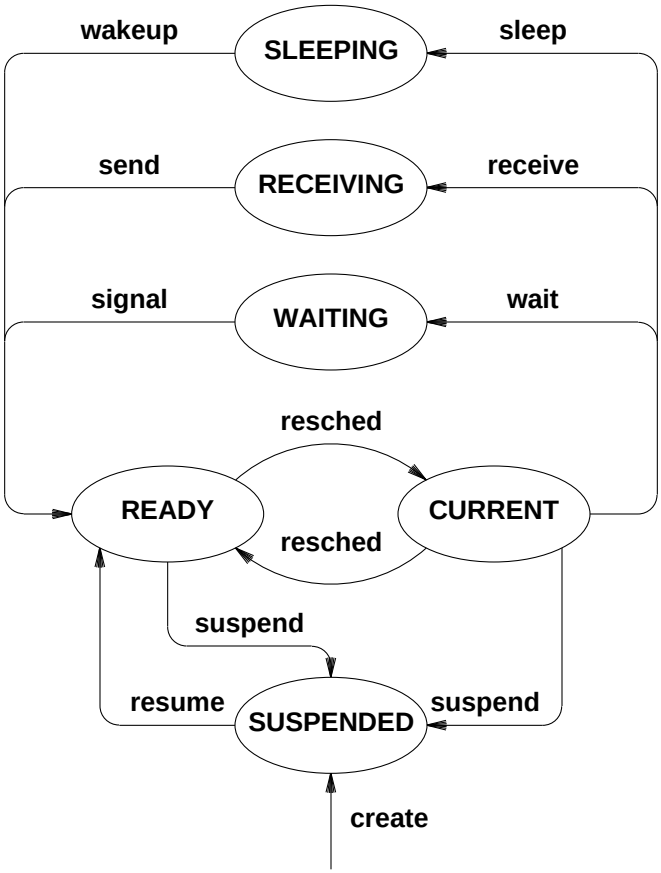

Figure 10.1 The process state transition diagram for the 'sleep' state

The implementation of *sleept* is straightforward. As shown below, it uses *insertd* to move the current process to the sleeping process list, changes its state to *sleeping*, and then calls *resched* to allow other processes to execute.

```
/* sleept.c - sleept */

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

/*------------------------------------------------------------------------
 *  sleept  --  delay the caller for a time specified in system ticks
 *------------------------------------------------------------------------
 */
SYSCALL sleept(n)
int n;
{
        int     ps;

        if ( n<0 )
                return(SYSERR);
        if (n == 0)
                return(OK);
        disable(ps);
        insertd(currpid,clockq,n);
        slnempty = TRUE;
        sltop = & q[q[clockq].qnext].qkey;
        proctab[currpid].pstate = PRSLEEP;
        resched();
        restore(ps);
        return(OK);
}
```

Sleept references three external variables defined in file *sleep.h* (below): *clockq*, *sltop*, and *slnempty*. *Clockq* contains the *q* index of the list of sleeping processes.

```
/* sleep.h */

extern  int     clockq;         /* q index of sleeping process list     */
extern  int     *sltop;         /* address of first key on clockq       */
extern  int     slnempty;       /* 1 iff clockq is nonempty             */
extern  long    tod;            /* time of day (ticks since startup)    */
extern  int     defclk;         /* >0 iff clock interrupts are deferred */
extern  int     clkdiff;        /* number of clock ticks deferred       */

#define TICSN   19663           /* no. of ticks per 1080 seconds        */
#define TICSD   1080
```

Even though the tick rate for PC-Xinu is relatively slow, processing clock interrupts is expensive. Variables *sltop* and *slnempty* help optimize interrupt processing by making it easy to determine whether sleeping processes should be awakened. *Slnempty* tells whether the list *clockq* is currently nonempty. If any processes remain on *clockq*, *sltop* gives the address of the key in the first one. The interrupt routine completely skips the code for sleeping processes when *slnempty* is zero, and it uses *sltop* to quickly locate the delta-key of the first sleeping process when *slnempty* is nonzero.

10.6 Delays Measured In Seconds

The size of an integer, 16 bits, limits the delay allowed by *sleept* to $2^{15} - 1$ ticks, which is about 1800 seconds, or 30 minutes. System call *sleep* provides a way for processes to delay up to 9 hours because its argument specifies a delay measured in seconds rather than ticks. *Sleep* uses *sleept* repeatedly to schedule shorter delays until the total delay time has elapsed.

```
/* sleep.c - sleep */

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

/*------------------------------------------------------------------------
 *  sleep  --  delay the calling process n seconds
 *------------------------------------------------------------------------
 */
SYSCALL sleep(n)
int n;
{
        int     ps;
```

```
        if ( n<0 )
                return(SYSERR);
        if (n == 0) {
                disable(ps);
                resched();
                restore(ps);
                return(OK);
        }
        while (n >= TICSD) {
                sleept(TICSN);
                n -= TICSD;
        }
        if (n > 0)
                sleept( (int)( ((long)n*(long)TICSN) / (long)TICSD ) );
        return(OK);
}
```

The numbers *TICSN* and *TICSD* (having values 19663 and 1080 respectively) defined in sleep.h deserve special comment. It turns out that the quotient *TICSN*/*TICSD* very closely approximates the tick rate, with an error of about 25 parts per billion. Put another way, *TICSN* ticks equals *TICSD* seconds. When scheduling a timed delay in seconds, delays exceeding *TICSD* seconds can be scheduled as (possibly repeated) delays of *TICSN* ticks using *sleept*. When the remaining delay in seconds is less than *TICSD*, it is used to determine how many ticks should be scheduled as a fraction of *TICSN* to achieve the required delay in seconds. For example, if *n* is the number of seconds to delay, the quotient

*n*TICSN/TICSD*

will be the number of ticks to schedule for *sleept*. Observe that the multiplication *n*TICSN* should be done before dividing by *TICSD* to avoid integer round-off errors. In view of the magnitudes of these numbers, this product must be carried out using long integer arithmetic.

Notice that a request for a delay of one second will mean scheduling a delay of exactly 18 ticks and will result in an actual delay of approximately 0.988659 seconds, giving an error of about 1.1341 percent. These errors can accumulate when such delays are scheduled repeatedly over long periods of time. Also observe that *sleep* is designed to cause an immediate reschedule if the sleep time is zero. Since *resched* cannot be called with interrupts enabled, this feature provides a simple way for a process to reschedule itself.

10.7 Clock Interrupt Processing

The *intcom* interrupt dispatcher calls the clock interrupt service routine *clkint* on each clock tick. Because it has been called from an interrupt dispatcher, *clkint* assumes that interrupts have been disabled upon entry; the keyword *INTPROC* reminds the reader of this assumption. The parameter to *clkint* is a dummy parameter passed automatically by *intcom*. *Clkint* first increments the global time-of-day value *tod*, which counts the number of ticks since system startup. If deferred clock processing is in effect, *clkdiff* is incremented and *clkint* returns immediately.

If the list of sleeping processes is not empty, *clkint* decrements the first key on *clockq*, and if the delay has reached zero, *wakeup* is called to wake up processes whose delay time has elapsed. Note how *slnempty* and *sltop* eliminate the computation of subscripts at interrupt time. Finally, the preemption counter is decremented which will result in rescheduling the current process if its time slice has expired.

```
/* clkint.c - clkint */

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

/*------------------------------------------------------------------------
 *  clkint  --  clock service routine
 *  called at every clock tick and when starting the deferred clock
 *------------------------------------------------------------------------
 */
INTPROC clkint(mdevno)
int mdevno;                             /* minor device number          */
{
        int     i;

        tod++;
        if (defclk) {
                clkdiff++;
                return;
        }
        if (slnempty)
                if ( (--*sltop) <= 0 )
                        wakeup();
        if ( (--preempt) <= 0 )
                resched();
}
```

10.8 Awakening Sleeping Processes

Wakeup makes ready all processes whose delay time has elapsed. The count of the first key on *clockq* takes into account any ticks that have accumulated since the last call to *wakeup*. *Wakeup* removes and makes ready the first process from *clockq* whose delay time has elapsed, and it propagates any deficiency in accumulated ticks to the next item in the waiting list. Finally *wakeup* resets *sltop* and *slnempty* to reflect the new queue status before calling *resched*, because rescheduling may allow another process to run (and the clock to interrupt).

```
/* wakeup.c - wakeup */

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

/*------------------------------------------------------------------------
 *  wakeup  --  called by clock interrupt dispatcher to awaken processes
 *------------------------------------------------------------------------
 */
wakeup()
{
        register int    makeup;         /* makeup for lost time         */
        register int    k;              /* key value                    */

        makeup = 0;
        while ( nonempty(clockq) && (k=firstkey(clockq)) <= makeup ) {
                makeup -= k;
                ready(getfirst(clockq));
        }
        if ( slnempty = nonempty(clockq) ) {
                sltop = &firstkey(clockq);
                *sltop -= makeup;
        }
        resched();
}
```

10.9 Deferred Clock Processing

The clock dispatcher includes an additional feature that complicates it: *deferred clock processing*. In essence, deferred clock mode allows the system to accumulate clock ticks without initiating events. The difference between ignoring clock interrupts and deferring them is that the clock handler can schedule events that ‘‘should have occurred’’ when it leaves deferred mode and returns to normal mode. If the clock only remains deferred for a few ticks at a time, the system will appear to operate normally.

The motivation for deferred clock processing is that the operating system keeps interrupts disabled while it switches context. Disabling interrupts poses no problems when input comes from a keyboard because humans type slowly compared to the speed at which computers consume data. But for some computer-to-computer communication, data is sent and received at a high speed. If a context switch happens when receiving a block of characters, some of the characters may be lost.

To solve this problem, the I/O system needs to prohibit context switches for short periods of time even though interrupts remain enabled. Ideally, the system should be able to ‘‘make up for lost time’’ when context switching is reenabled again. Even though it is impossible to prevent context switching without changing the system behavior, the idea is to find a solution that has minimal impact without corrupting the basic design. Deferred clock processing consists of postponing, but not ignoring, context switches. During a deferred clock period, context switching is suspended by deferring clock interrupts. When the deferral ends, normal processing resumes.

10.9.1 Procedures For Changing To And From Deferred Mode

A process can place the clock in *deferred mode* by calling *stopclk* and return the clock to *real-time* mode by calling *strtclk*. Any number of processes can request that the clock be deferred – it remains deferred until they have all called *strtclk*. *Stopclk* counts deferral requests by incrementing *defclk*, and *strtclk* counts restart requests by decrementing it. As long as *defclk* remains positive, the interrupt handler counts clock ticks in *clkdiff* without processing them.

Strtclk "makes up for lost time" when *defclk* reaches zero again, by catching up on all events that should have occurred while the clock remained deferred. To do so, *strtclk* updates the preemption counter and subtracts the accumulated ticks from the delay of sleeping processes, waking up processes whose sleep times have elapsed.

The code for both *strtclk* and *stopclk* is contained in file *ssclock.c*, shown below.

```
/* ssclock.c - stopclk, strtclk */

#include <conf.h>
#include <kernel.h>
#include <q.h>
#include <sleep.h>
```

```
/*------------------------------------------------------------------------
 *  stopclk  --  put the clock in defer mode
 *------------------------------------------------------------------------
 */
stopclk()
{
        int     ps;

        disable(ps);
        defclk++;
        restore(ps);
}

/*------------------------------------------------------------------------
 *  strtclk  --  take the clock out of defer mode
 *------------------------------------------------------------------------
 */
strtclk()
{
        int     ps;
        int makeup;
        int next;

        disable(ps);
        if ( defclk<=0 || --defclk>0 ) {
                restore(ps);
                return;
        }
        makeup = clkdiff;
        clkdiff = 0;
        if (slnempty)
                if ( (*sltop -= makeup) <= 0 )
                        wakeup();
        if ( (preempt -= makeup) <= 0 )
                resched();
        restore(ps);
}
```

10.10 Clock Initialization

Procedure *clkinit*, shown below, performs the necessary initialization.

```
/* clkinit.c - clkinit */

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

/*------------------------------------------------------------------------
 *  clkinit  --  initialize the clock and sleep queue (called at startup)
 *------------------------------------------------------------------------
 */
clkinit()
{
        tod = 0L;                       /* initial time of day          */
        preempt = QUANTUM;              /* initial time quantum         */
        slnempty = FALSE;               /* initially, no process asleep */
        clkdiff = 0;                    /* zero deferred ticks          */
        defclk = 0;                     /* clock is not deferred        */
        clockq = newqueue();            /* allocate clock queue in q    */
}
```

10.11 Summary

A real-time clock interrupts the CPU at regular intervals. Because the clock interrupts frequently, the interrupt routine must be designed to operate efficiently. The real-time clock manager uses clock interrupts to schedule events in the future and then to initiate the events at the appropriate time. A preemption event, scheduled every time the system switches context, forces a call to the scheduler after *QUANTUM* clock ticks pass. Preemption guarantees that no process uses the CPU forever. Wakeup events, scheduled when user processes request timed delays, cause the running process to enqueue itself on a list of sleeping processes and pass control to another process. The interrupt handler awakens sleeping processes when their delay expires by moving them back to the ready list.

FOR FURTHER STUDY

Because timing and clock processing rely on the hardware available, most books describe how the operating system uses timed delays without giving much detail about the clock routines. Examples of the use of clocks in virtual memory management, process management (e.g., time slicing), and distributed systems can be found in Calingaert [1982], Peterson and Silberschatz [1983], and Habermann [1976].

EXERCISES

10.1 The PC does not have a hardware time-of-day clock. Show how to use the variable *tod* to simulate a time-of-day clock.

10.2 Program the 8253 timer chip to interrupt at different tick rates. Measure the difference it makes in the execution of CPU intensive programs.

10.3 Conduct an experiment to determine whether the system ever misses clock interrupts. Before beginning, estimate the number of instructions executed during an interrupt when a sleeping process awakens, and the time it will take to execute them.

10.4 Trace the series of calls starting with a clock interrupt that awakens two sleeping processes, one of which has higher priority than the currently executing process.

10.5 What goes wrong if *QUANTUM* is set to 1? Hint: consider switching back to a process that was suspended by *resched* while processing an interrupt.

10.6 Speculate about the usefulness of deferring clock interrupts. Compare its effect when the deferral lasts only a few clock ticks to cases where it lasts many seconds.

10.7 Does *sleep*(3) guarantee a minimum delay of 3 seconds, an exact delay of 3 seconds, or a maximum delay of 3 seconds?

10.8 Identify a problem in the way that *kill* removes processes from the queue of sleeping processes. Rewrite *kill* to correct the problem.

10.9 What might happen if *wakeup* called *wait*?

10.10 Systems that charge processes for CPU time face the following problem: when an interrupt occurs, it is most convenient to let the current process execute the interrupt routine even if the interrupt has nothing to do with the current process. Investigate how such systems charge the cost of executing interrupt routines like *wakeup* to the processes that are affected.

10.11 Rewrite *sleep* to avoid the overhead of calling *sleept*.

10.12 Design an experiment to see if preemption ever causes the system to reschedule. Be careful: awakening a sleeping process to test a variable or using normal output will affect the results of the experiment by forcing calls to *resched*.

10.13 Some machines have programmable *interval timer* hardware. The interval timer is set by specifying a delay; it interrupts when the delay has completed. Redesign the clock routines assuming that you have three independent interval timers available.