6

Process Coordination

Independent processes use coordination primitives to synchronize their actions and to cooperate in sharing resources. Chapter 1 introduced *counting semaphores*, the basic process coordination mechanism in PC-Xinu and gave examples of their use. In one example, two processes coordinated to guarantee that the ‘‘consumer’’ received every value emitted by the ‘‘producer.’’ In another example, a set of processes used a semaphore to obtain exclusive access to a data structure they shared.

Semaphores are perhaps the easiest process coordination primitives to understand and implement. Conceptually, each semaphore, *s*, consists of an integer count. Processes call *wait*(*s*) to decrement its count and *signal*(*s*) to increment it. If the semaphore count becomes negative when a process executes *wait*(*s*), that process is delayed. From the point of view of the process, the call to *wait* just does not return for a while. A delayed process becomes ready to run again each time *signal* is called. If no process ever calls *signal*, the delayed process continues to wait forever.

Exactly how does procedure *wait* delay its caller? It is important to remember that even though processes run concurrently, we are discussing a system that usually executes on a single processor. A process cannot execute instructions while waiting without depriving other processes of CPU service. For example, waiting that involves testing a memory location in a tight loop is dangerous – if the CPU spends all its time executing the ‘‘waiting’’ process, no other process can ever call *signal* to terminate the wait. Even in systems with many processors, so-called *busy waiting* techniques may interfere with processing because each processor contends with others while using the memory or bus systems to fetch instructions or data. To minimize system overhead, the coordination primitives in PC-Xinu follow this principle:

> *Waiting processes do not execute instructions; when all user processes are waiting, the system does not execute code.*

Whether a system executes code when all user processes are waiting depends somewhat on the machine architecture. Like most machines, the 8088 includes a *HALT* instruction to halt the CPU while all processes wait. For the time being we will defer the problem of halting the CPU when all processes wait and consider the simpler case of how to avoid busy waiting when at least one process remains ready to run. Remember that PC-Xinu always has a ready process – the null process.

6.1 Low-Level Coordination Techniques

The previous chapter contained examples of process coordination techniques in the code for routines like *ready* and *resume*. When a process executing one of these routines needs to modify a shared data structure like the process table, it must be sure that no other process attempts concurrent access. Coordination between these low-level system routines involves disabling interrupts and being careful not to call *resched*. Why not use this solution again? Disabling interrupts has an undesirable global effect on the system: it stops all but one process and limits what that process can do. We need general purpose coordination primitives so that arbitrary subsets of the processes can coordinate without stopping other processes, without disabling device interrupts for long periods of time, and without limiting what running processes can do. For example, it should be possible for one process to prohibit changes to a large data structure long enough to print it, without stopping those processes that do not need to access it.

6.2 Implementation Of High-Level Coordination Primitives

The PC-Xinu implementation of counting semaphores avoids ''busy waiting'' by denying CPU service to waiting processes. When a process needs to wait for some semaphore, the system places it on a list of processes associated with that semaphore. Naturally, each semaphore must have its own, independent list of waiting processes. To delay the current process, *wait*(*s*) enqueues it on the list for *s* and calls *resched* allowing other processes to run. *Signal*(*s*) checks the list associated with *s* whenever it is called. Provided the list of waiting processes is nonempty, *signal* restarts a process by moving it back to the ready list.

In what state should a process be placed while it is waiting for a semaphore? It is clearly not *current* or *ready* because it is neither using the CPU nor eligible to use it. The *suspended* state, introduced in Chapter 5, will not suffice either, because it is used by procedures like *suspend* and *resume* that have no connection with semaphores. More importantly, processes waiting for semaphores appear on a list, but suspended processes do not – *kill* needs to distinguish the two cases. Whenever the existing process states cannot adequately indicate how operations should be carried out, the designer can invent a new state. In this case we will call the new state ''waiting,'' and refer to it in the code with the symbolic constant *PRWAIT*. Figure 6.1 shows the expanded state transitions.

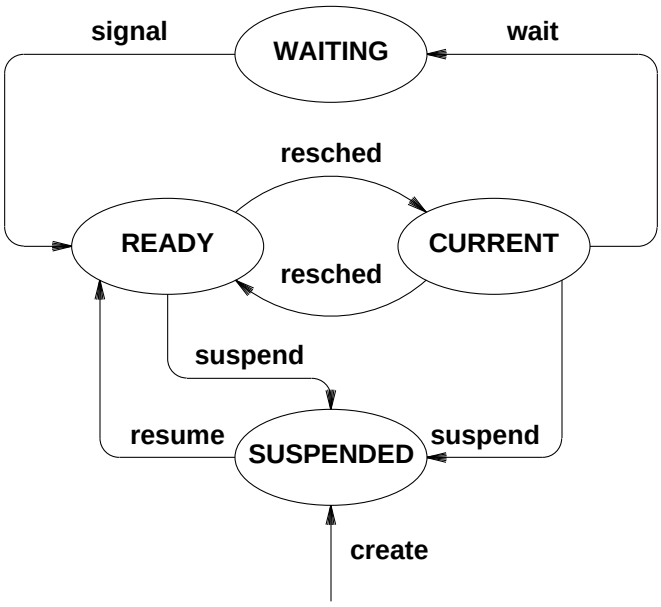

Figure 6.1 Transitions for the 'waiting' state

6.2.1 Semaphore Data Structures

In PC-Xinu, semaphore information is kept in the global semaphore table, *semaph*. Each entry in *semaph* contains the integer count and list of processes corresponding to one semaphore; its definition is given in C by structure *sentry*. File *sem.h* contains the details:

```
/* sem.h - isbadsem */

#ifndef NSEM
#define NSEM            45      /* number of semaphores, if not defined */
#endif

#define SFREE   '\01'           /* this semaphore is free               */
#define SUSED   '\02'           /* this semaphore is used               */

struct  sentry  {               /* semaphore table entry                */
        char    sstate;         /* the state SFREE or SUSED             */
        short   semcnt;         /* count for this semaphore             */
        short   sqhead;         /* q index of head of list              */
        short   sqtail;         /* q index of tail of list              */
};

extern  struct  sentry  semaph[];
extern  int     nextsem;

#define isbadsem(s)     (s<0 || s>=NSEM)
```

In structure *sentry*, field *semcnt* contains the current integer value of the semaphore. The list of processes waiting for a semaphore resides in the *q* structure; *sentry* fields *sqhead* and *sqtail* only give the index of the head and tail. The state field, *sstate* tells whether each semaphore entry is currently free (unallocated) or in use.

Throughout the system, semaphores are identified by an integer. As with processes, the semaphore identifiers are meaningful values, chosen to connect the semaphore and its table entry:

> *Semaphores are identified by their index in the global semaphore table, semaph.*

System calls *wait* and *signal* implement the basic semaphore operations. *Wait*(*s*) decrements the count of semaphore *s*. If the count remains nonnegative, *wait* returns to the caller immediately. Otherwise, it enqueues the calling process on the list for *s*, changes the process state to *PRWAIT*, and calls *resched* to switch to a ready process. The list is maintained as a FIFO queue with insertions at the tail and deletions at the head. In essence, a process executing *wait* on a semaphore with a nonpositive count voluntarily gives up control of the CPU after *wait* records its id on the list of waiting processes.

Once enqueued on a semaphore list, a process remains there (and hence, not executing) until it reaches the head of the queue and some other process signals the semaphore. When the call to *signal* moves the waiting process back to the ready list, it becomes eligi-

ble to use the CPU and eventually resumes execution. From the point of view of the waiting process, its last act consisted of calling *ctxsw*. The call to *ctxsw* returns to *resched*, the call to *resched* returns to *wait*, and the call to *wait* returns to wherever it was called.

```
/* wait.c - wait */

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

/*------------------------------------------------------------------------
 *  wait  --  make current process wait on a semaphore
 *------------------------------------------------------------------------
 */
SYSCALL wait(sem)
        int     sem;
{
        int     ps;
        register struct sentry  *sptr;
        register struct pentry  *pptr;

        disable(ps);
        if (isbadsem(sem) || (sptr = &semaph[sem])->sstate == SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if ( --sptr->semcnt < 0 ) {
                (pptr = &proctab[currpid])->pstate = PRWAIT;
                pptr->psem = sem;
                enqueue(currpid,sptr->sqtail);
                resched();
        }
        restore(ps);
        return(OK);
}
```

The code for *signal* is straightforward. It increments the semaphore count and makes the first waiting process ready to run.

```
/* signal.c - signal */

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

/*------------------------------------------------------------------------
 *  signal  --  signal a semaphore, releasing one waiting process
 *------------------------------------------------------------------------
 */
SYSCALL signal(sem)
register int    sem;
{
        register struct sentry  *sptr;
        int     ps;

        disable(ps);
        if (isbadsem(sem) || (sptr = &semaph[sem])->sstate == SFREE) {
                restore(ps);
                return(SYSERR);
        }
        if ( sptr->semcnt++ < 0 ) {
                ready(getfirst(sptr->sqhead));
                resched();
        }
        restore(ps);
        return(OK);
}
```

Although it may seem difficult to understand why *signal* makes a process ready even though the semaphore count remains negative, or why *wait* does not enqueue the process every time, the reason is both easy to understand and easy to implement. *Wait* and *signal* keep the following condition invariant:

> *A nonnegative semaphore count means that the queue is empty; a semaphore count of negative n means that the queue contains n waiting processes.*

Both *wait* and *signal* change the semaphore count, so they adjust the queue length, if necessary, to reestablish the invariant. Because *wait* decrements the count, it adds the current process to the queue if the new count is negative. Because *signal* increments the count, it removes a process from the queue if the queue is nonempty.

6.3 Semaphore Creation and Deletion

The need for semaphores may come and go as execution progresses, so it would be foolish to allocate each semaphore a specific purpose. Rather than fix the use of semaphores at compile time, Xinu allows processes to request a semaphore, use it, and then release it. Processes can create an arbitrary number of semaphores in arbitrary order, as long as the number allocated simultaneously does not exceed the maximum table size. To help minimize the cost of creating semaphores, the system preallocates head and tail nodes in the *q* structure for each semaphore list at system initialization time. Thus, only a small amount of work needs to be done at semaphore creation time.

System calls *screate* and *sdelete* allocate and release semaphores. *Screate*, shown below, takes the initial semaphore count as an argument and returns the semaphore id of a semaphore with that count. The method is simple: search for an unused entry in the semaphore table *semaph* and initialize it. *Screate* uses procedure *newsem* to search for a free entry. *Screate* then initializes the count and returns the index of the semaphore just allocated.

```
/* screate.c - screate, newsem */

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

/*------------------------------------------------------------------------
 *  screate  --  create and initialize a semaphore, returning its id
 *------------------------------------------------------------------------
 */
SYSCALL screate(count)
        int     count;                  /* initial count (>=0)          */
{
        int     ps;
        int     sem;

        disable(ps);
        if ( count<0 || (sem=newsem())==SYSERR ) {
                restore(ps);
                return(SYSERR);
        }
        semaph[sem].semcnt = count;
        /* sqhead and sqtail were initialized at system startup */
        restore(ps);
        return(sem);
}

/*------------------------------------------------------------------------
 *  newsem  --  allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
LOCAL   newsem()
{
        int     sem;
        int     i;

        for (i=0 ; i<NSEM ; i++) {
                sem=nextsem--;
                if (nextsem < 0)
                        nextsem = NSEM-1;
                if (semaph[sem].sstate==SFREE) {
                        semaph[sem].sstate = SUSED;
```

```
                        return(sem);
                }
        }
        return(SYSERR);
}
```

Sdelete reverses the actions of *screate*. It takes the index of a semaphore as an argument, and releases the semaphore table entry for use again.

```
/* sdelete.c - sdelete */

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

/*------------------------------------------------------------------------
 *  sdelete  --  delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
SYSCALL sdelete(sem)
        int     sem;
{
        int     ps;
        int     pid;
        struct  sentry  *sptr;          /* address of sem to free       */

        disable(ps);
        if ( isbadsem(sem) || semaph[sem].sstate==SFREE ) {
                restore(ps);
                return(SYSERR);
        }
        sptr = &semaph[sem];
        sptr->sstate = SFREE;
        if ( nonempty(sptr->sqhead) ) { /* free waiting processes       */
                while( (pid=getfirst(sptr->sqhead)) != EMPTY )
                        ready(pid);
                resched();
        }
        restore(ps);
        return(OK);
}
```

If processes remain enqueued when *sdelete* tries to delete a semaphore, it must dispose of them. When faced with an active semaphore, *sdelete* places waiting processes back on the ready list, allowing each to resume execution as if the semaphore had been signaled. This is only one possible disposition of the waiting processes; the exercises suggest some other alternatives.

6.4 Returning The Semaphore Count

It is useful to be able to retrieve the count of a semaphore. For example, knowing that a semaphore count is positive may indicate that there are units available of a particular resource. The count of a semaphore, contained in the *semcnt* component of the semaphore structure, is returned by the *scount* routine shown here:

```
/* scount.c - scount */

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

/*------------------------------------------------------------------------
 *  scount  --  return a semaphore count
 *------------------------------------------------------------------------
 */
SYSCALL scount(sem)
int sem;
{
        extern  struct  sentry  semaph[];
        int     ps;
        int     ct;

        disable(ps);
        if (isbadsem(sem) || semaph[sem].sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        ct = semaph[sem].semcnt;
        restore(ps);
        return(ct);
}
```

6.5 Other Semaphore Utilities

Two additional utilities, *signaln* and *sreset*, are used in the input/output software and are also available to user programs. *Signaln* is equivalent to calling *signal* a number of times, while *sreset* is the same as deleting a semaphore and then creating it with a new initial *count*. These routines are designed to be more efficient than the combination of routines they substitute for.

```
/* signaln.c - signaln */

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

/*------------------------------------------------------------------------
 *  signaln  --  signal a semaphore n times
 *------------------------------------------------------------------------
 */
SYSCALL signaln(sem,count)
        int     sem;
        int     count;
{
        struct  sentry  *sptr;
        int     ps;

        disable(ps);
        if (isbadsem(sem) || semaph[sem].sstate==SFREE || count<=0) {
                restore(ps);
                return(SYSERR);
        }
        sptr = &semaph[sem];
        for (; count > 0  ; count--)
                if ((sptr->semcnt++) < 0)
                        ready(getfirst(sptr->sqhead));
        resched();
        restore(ps);
        return(OK);
}
```

```
/* sreset.c - sreset */

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

/*------------------------------------------------------------------------
 *  sreset  --  reset the count and queue of a semaphore
 *------------------------------------------------------------------------
 */
SYSCALL sreset(sem,count)
        int     sem;
        int     count;
{
        struct  sentry  *sptr;
        int     ps;
        int     pid;
        int     slist;

        disable(ps);
        if (isbadsem(sem) || count<0 || semaph[sem].sstate==SFREE) {
                restore(ps);
                return(SYSERR);
        }
        sptr = &semaph[sem];
        slist = sptr->sqhead;
        while ((pid=getfirst(slist)) != EMPTY)
                ready(pid);
        sptr->semcnt = count;
        resched();
        restore(ps);
        return(OK);
}
```

6.6 Summary

This chapter discussed process synchronization primitives known as counting semaphores. In addition to showing how to build procedures that create and delete semaphores, it showed how the primitive semaphore operations *wait* and *signal* cooperate to suspend processes and resume them, such that waiting processes do not use any CPU time. In essence, a process that needs to wait for a semaphore voluntarily enqueues itself

on the list of processes waiting for the semaphore and calls the scheduler to allow other processes to execute. Eventually, when another process signals the semaphore, it moves the waiting process back to the ready list, so it can regain control of the CPU.

FOR FURTHER STUDY

Dijkstra [1965] introduced semaphores and showed how to use them for synchronization. Initially, semaphores had only binary values, and the operations were known as *P* (wait) and *V* (signal). These are summarized in the appendix of Dijkstra [1968]. Although binary semaphores are sufficient to provide basic synchronization and mutual exclusion, the addition of counts makes them much more convenient to use. Patil [1971] and Kosaraju [1973] consider whether binary semaphores can solve all synchronization problems.

Brinch Hansen [1970, 1972] showed how to synchronize by exchanging messages. Another process synchronization tool, the monitor, is described by Hoare [1974]. Its beginnings can be seen in the ''secretary'' of Dijkstra [1971] and in Brinch Hansen [1973]. Although high-level primitives like monitors make it easier to express the intended process interaction, they can be implemented with semaphores.

EXERCISES

6.1 Deleting a semaphore while processes remain enqueued waiting for it might cause abnormal system behavior. Give a scenario where this might occur. Rewrite *sdelete* to refuse to delete a busy semaphore.

6.2 Another way to handle the deletion of an active resource is to defer the deletion. Rewrite *sdelete*, *wait*, and *signal* to place deleted semaphores in a state such that *signal* releases the semaphore table entry when it removes the last waiting process from the queue. What unexpected effects might deferring deletion have?

6.3 Instead of placing responsibility for deletion of active semaphores on the process calling *sdelete*, consider allowing waiting processes to handle the problem. Modify *wait* so it returns an integer *DELETED* in case the semaphore was deleted while the calling process was waiting. (Choose a value for *DELETED* that will not interfere with *SYSERR* or *OK*.) How can *wait* know when to return *DELETED*? Checking the state of the semaphore is insufficient because a higher priority process may reuse the table entry before all the deleted processes resume execution and examine *sstate*. Hint: deposit the return value in the process table.

6.4 Instead of allocating a central semaphore table, arrange to have each process allocate space for semaphore entries as needed, and use the address of an entry as the semaphore id. Compare this method to that of a centralized table. What are the advantages? Disadvantages?

6.5 *Wait*, *signal*, *screate*, and *sdelete* coordinate among themselves for use of the semaphore table. How much easier would it be to code them if semaphores were used?

6.6 Assuming procedure calls are expensive, *sdelete* could check the priority of processes as it added them to the ready list and not bother calling *resched* if none had higher priority than the current process. Speculate about the wisdom of adding this optimization.

6.7 Languages meant specifically for writing concurrent programs often have coordination and synchronization imbedded in the language constructs directly. For example, it might be possible to declare procedures in groups such that the compiler automatically inserts code to prohibit more than one process from executing in any group. Find an example of a language designed for concurrent programming, and compare process coordination with the semaphores in PC-Xinu. What types of mistakes can a programmer make when required to manipulate semaphores directly?

6.8 Because it is much more likely that an incorrect expression will evaluate to 0 or 1, *newsem* begins allocating semaphores from the high end of the table to reduce the chance of inadvertently waiting on the wrong semaphore. If all entries are allocated, the problem persists. Suggest better ways of identifying semaphores.

6.9 Draw a call graph of all procedures in Chapters 1 through 6, showing which procedures each procedure calls. Can the layered structure be deduced from the graph?

6.10 The *scount* routine returns *SYSERR* in case of error. But *SYSERR* is a legal semaphore count value, so the caller cannot reliably tell whether the return value is a semaphore count or an error indicator. Redesign *scount* to eliminate this problem.