7

Coordination Of Concurrent Processes

The future belongs to him who knows how to wait.

— Russian Proverb

7.1 Introduction

Previous chapters introduce pieces of a process manager, including scheduling, context switching, and functions that create and terminate processes. This chapter continues the exploration of process management by discussing functions that a set of processes can use to coordinate and synchronize their actions. The chapter explains the motivation for such primitives and their implementation. The chapter also considers coordination of multiple processors, such as those on a multicore chip.

The next chapter extends our discussion of a process manager by describing a low-level message passing mechanism. Later chapters show how synchronization functions are used to perform I/O.

7.2 The Need For Synchronization

Because they execute concurrently, processes need to cooperate when sharing global resources. In particular, an operating system designer must ensure that only one process attempts to change a given variable at any time. For example, consider the process table. When a new process is created, a slot in the table must be allocated and values inserted. If two processes each attempt to create a new process, the system must guarantee that only one of them can execute *create* at a given time, or errors can result.

The previous chapter illustrates one approach system functions can take to guarantee that no other process interferes with them: a function disables interrupts and avoids using any functions that call *resched*. Indeed, system calls such as *suspend*, *resume*, *create*, and *kill* each use the approach.

Why not use the same solution whenever a process needs to guarantee non-interference? The answer is that disabling interrupts has an undesirable global effect on all parts of the system: it stops all activity except for one process, and limits what the process can do. In particular, no I/O can occur while interrupts are disabled. We will learn later that disabling interrupts too long can cause problems (e.g., if packets continue to arrive over a network while interrupts are disabled, the network interface will start to discard them). Therefore, we need a general-purpose coordination mechanism that permits arbitrary subsets of the processes to coordinate the use of individual data items without disabling device interrupts for long periods of time, without interfering with processes outside the subset, and without limiting what the running process can do. For example, it should be possible for one process to prohibit changes to a large data structure long enough to format and print the data, without stopping processes that do not need to access the data structure. The mechanism should be transparent: a programmer should be able to understand the consequences of process coordination. Thus, further synchronization mechanisms are needed that:

- Allow a subset of processes to contend for access to a resource
- Provide a policy that guarantees fair access

The first item ensures that coordination is local: instead of disabling all interrupts, only those processes contending for a given resource will block waiting for access. Other parts of the system can continue to operate unaffected. The second item ensures that if *K* processes all attempt to access a given resource, each of the *K* will eventually receive access (i.e., no process is *starved*).

Chapter 2 introduces the fundamental mechanism that solves the problem: *counting semaphores*. The chapter also provides examples that show how processes use semaphores to coordinate. As Chapter 2 indicates, semaphores provide an elegant solution for two problems:

- Mutual exclusion
- Producer–consumer interaction

Mutual exclusion. The term *mutual exclusion* is used to describe a situation where a set of processes needs to guarantee that only one of them operates at a given time. Mutual exclusion includes access to shared data, but can also include access to an arbitrary shared resource, such as an I/O device.

Producer–consumer interaction. We use the term *producer–consumer interaction* to refer to a situation where processes exchange data items. In the simplest form, one process acts as a *producer* by generating a sequence of data items, and another process acts as a *consumer* by accepting the data items. In more complex forms, one or more processes can act as producers and one or more processes can act as consumers. The key to coordinating the interaction is that each item produced must be received by exactly one consumer (i.e., no items are lost and no items are duplicated).

Both forms of process coordination arise throughout an operating system. For example, consider a set of applications that are producing messages to be displayed on the console. The console device software must coordinate processes to ensure that characters do not arrive faster than the hardware can display them. Outgoing characters can be placed in a buffer in memory. Once the buffer fills, the producer must be blocked until space becomes available. Similarly, if the buffer becomes empty, the device stops sending characters. The key idea is that a producer must be blocked when the consumer is not ready to receive data, and a consumer must be blocked when a producer is not ready to send data.

7.3 A Conceptual View Of Counting Semaphores

A counting semaphore mechanism that solves both problems described above has a surprisingly elegant implementation. Conceptually, a semaphore, *s*, consists of an integer count and a set of blocked processes. Once a semaphore has been created, processes use two functions, *wait* and *signal*, to operate on the semaphore. A process calls *wait*(*s*) to decrement the count of semaphore *s*, and *signal*(*s*) to increment the count. If the semaphore count becomes negative when a process executes *wait*(*s*), the process is temporarily blocked and placed in the semaphore's set of blocked processes. From the point of view of the process, the call to *wait* does not return for a while. A blocked process becomes ready to run again when another process calls *signal* to increment the semaphore count. That is, if any processes are blocked waiting for a semaphore when *signal* is called, one of the blocked processes will be made ready and allowed to execute. Of course, a programmer must use semaphores with caution: if no process ever signals the semaphore, the blocked processes will wait forever.

7.4 Avoidance Of Busy Waiting

What should a process do while waiting on a semaphore? It might seem that after it decrements the semaphore count, a process could repeatedly test the count until the value becomes positive. On a single processor system, however, such *busy waiting* is unacceptable because other processes will be deprived of the processor. If no other process receives processor service, no process can call *signal* to terminate the wait. Therefore, operating systems avoid busy waiting. Instead, semaphore implementations follow an important principle:

While a process waits on a semaphore, the process does not execute instructions.

7.5 Semaphore Policy And Process Selection

To implement semaphores without busy waiting, an operating system associates a process list with each semaphore. Only the current process can choose to wait on a semaphore. When a process waits on semaphore *s*, the system decrements the count associated with *s*. If the count becomes negative, the process must be blocked. To block a process, the system places the process on the list associated with the semaphore, changes the state so the process is no longer current, and calls *resched* to allow other processes to run.

Later, when *signal* is called on semaphore *s*, the semaphore count is incremented. In addition, the *signal* examines the process list associated with *s*. If the list is not empty (i.e., at least one process is waiting on the semaphore), *signal* extracts a process from the list and moves the process back to the ready list.

A question arises: if multiple processes are waiting, which one should *signal* select? Several policies have been used:

- Highest scheduling priority
- First-come-first-served (longest waiting time)
- Random

Although it may seem reasonable, selecting the highest priority waiting process violates the principle of fairness. To see why, consider a set of low-priority and high-priority processes that are using a mutual exclusion semaphore. Suppose each process repeatedly waits on the semaphore, uses the resource, and signals the semaphore. If the semaphore system always selects a high-priority process and the scheduling policy always gives the processor to high-priority processes, the low-priority processes can be blocked forever while high-priority processes continue to gain access.

To avoid unfairness, many implementations choose a first-come-first-served policy: if processes are waiting, the system always chooses the process that has been waiting the longest. The implementation of a first-come-first-served policy is both elegant and efficient: the system creates a FIFO queue for each semaphore, and uses the queue to store processes that are waiting. When it needs to block a process, *wait* inserts the process at the tail; when it needs to unblock a process, *signal* extracts a process from the head.

A first-come-first-served policy can lead to a *priority inversion* in the sense that a high-priority process can be blocked on a semaphore while a low-priority process executes. In addition, it can lead to a synchronization problem discussed in an exercise. One alternative consists of choosing among waiting processes at *random*. The chief disadvantage of random selection lies in computational overhead (e.g., random number generation).

After considering the advantages and disadvantages of various schemes, we chose a first-come-first-served policy for Xinu:

> *Xinu semaphore process selection policy: if one or more processes are waiting for semaphore* s *when a* signal *operation occurs for* s, *the process that has been waiting the longest becomes ready.*

7.6 The Waiting State

In what state should a process be placed while it is waiting for a semaphore? Because it is neither using the processor nor eligible to run, the process is neither *current* nor *ready*. The *suspended* state, introduced in the previous chapter cannot be used because functions *suspend* and *resume*, which move processes in and out of the suspended state, have no connection with semaphores. More important, processes waiting for semaphores appear on a list, but suspended processes do not — *kill* must distinguish the two cases when terminating a process. Because existing states do not adequately encompass processes waiting on a semaphore, a new state must be invented. We call the new state *waiting*, and use symbolic constant *PR_WAIT* in the code. Figure 7.1 shows the expanded state transition diagram.

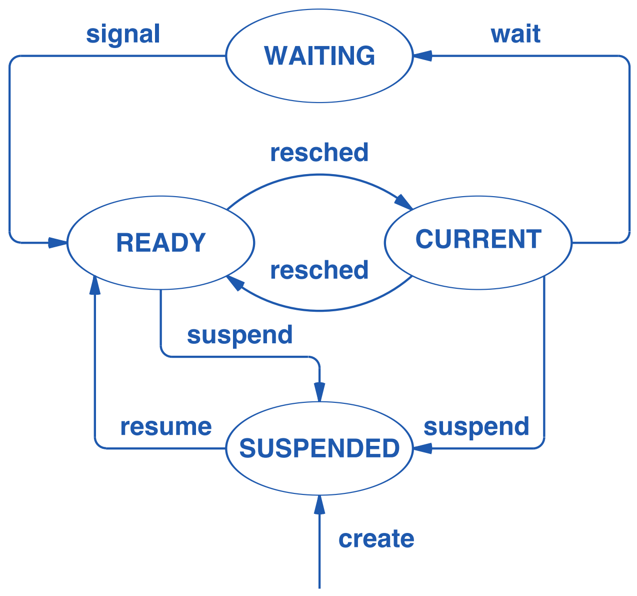

Figure 7.1 State transitions including the *waiting* state.

7.7 Semaphore Data Structures

The example system stores semaphore information in a global semaphore table, *semtab*. Each entry in *semtab* corresponds to one semaphore. An entry contains an integer count for the semaphore, and the ID of a queue that can be used to hold waiting processes. The definition of an entry is given by structure *sentry*. File *semaphore.h* contains the details.

```
/* semaphore.h - isbadsem */

#ifndef NSEM
#define NSEM            120     /* Number of semaphores, if not defined */
#endif

/* Semaphore state definitions */

#define S_FREE  0               /* Semaphore table entry is available   */
#define S_USED  1               /* Semaphore table entry is in use      */

/* Semaphore table entry */
struct  sentry  {
        byte    sstate;         /* Whether entry is S_FREE or S_USED    */
        int32   scount;         /* Count for the semaphore              */
        qid16   squeue;         /* Queue of processes that are waiting  */
                                /*      on the semaphore                */
};

extern  struct  sentry semtab[];

#define isbadsem(s)     ((int32)(s) < 0 || (s) >= NSEM)
```

In structure *sentry*, field *scount* contains the current integer count of the semaphore. The list of processes waiting for a semaphore resides in the queue structure, and field *squeue* gives the index of the head of the list for a given semaphore. The state field, *sstate*, tells whether the entry is currently used (i.e., allocated) or free (currently unallocated).

Throughout the system, semaphores are identified by an integer ID. As with other identification values, semaphore IDs are assigned to make lookup efficient: the semaphore table is an array, and each ID is an index in the array. To summarize:

A semaphore is identified by its index in the global semaphore table, semtab.

7.8 The Wait System Call

Recall that the two primary operations on a semaphore are *wait* and *signal*. *Wait* decrements the count of a semaphore. If the count remains nonnegative, *wait* returns to the caller immediately. In essence, a process executing *wait* on a semaphore with a nonpositive count voluntarily gives up control of the processor. That is, *wait* enqueues the calling process on the list for the semaphore, changes the process state to *PR_WAIT*, and calls *resched* to switch to a ready process. Also recall that our policy maintains the list of processes as a FIFO queue, which means a new process is inserted at the tail of a list. File *wait.c* contains the code.

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  wait  -  Cause current process to wait on a semaphore
 *------------------------------------------------------------------------
 */
syscall wait(
          sid32         sem             /* Semaphore on which to wait  */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  procent *prptr;         /* Ptr to process' table entry  */
        struct  sentry *semptr;         /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }

        if (--(semptr->scount) < 0) {           /* If caller must block */
                prptr = &proctab[currpid];
                prptr->prstate = PR_WAIT;       /* Set state to waiting */
                prptr->prsem = sem;             /* Record semaphore ID  */
                enqueue(currpid,semptr->squeue);/* Enqueue on semaphore */
                resched();                      /*   and reschedule     */
        }
        restore(mask);
        return OK;
}
```

Once enqueued on a semaphore list, a process remains in the waiting state (i.e., not eligible to execute) until the process reaches the head of the queue and some other process signals the semaphore. When the call to *signal* moves a waiting process back to the ready list, the process becomes eligible to use the processor, and eventually resumes execution. From the point of view of the waiting process, its last act consisted of a call to *ctxsw*. When the process restarts, the call to *ctxsw* returns to *resched*, the call to *resched* returns to *wait*, and the call to *wait* returns to the location from which it was called.

7.9 The Signal System Call

Function *signal* takes a semaphore ID as an argument, increments the count of the specified semaphore, and makes the first process ready, if any are waiting. 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 always enqueue the calling process, the reason is both easy to understand and easy to implement. *Wait* and *signal* maintain the following invariant regarding the count of a semaphore:

> *Semaphore 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.*

In essence, a count of positive *N* means that *wait* can be called *N* more times before any process blocks. Because *wait* and *signal* each change the semaphore count, they must each adjust the queue length to reestablish the invariant. When it decrements the count, *wait* examines the result, and adds the current process to the queue if the new count is negative. Because it increments the count, *signal* examines the queue and removes a process from the queue if the queue is nonempty.

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  signal  -  Signal a semaphore, releasing a process if one is waiting
 *------------------------------------------------------------------------
 */
```

```
syscall signal(
          sid32         sem             /* ID of semaphore to signal    */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  sentry *semptr;         /* Ptr to sempahore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }
        semptr= &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        if ((semptr->scount++) < 0) {   /* Release a waiting process */
                ready(dequeue(semptr->squeue));
        }
        restore(mask);
        return OK;
}
```

7.10 Static And Dynamic Semaphore Allocation

An operating system designer must choose between two approaches for semaphore allocation:

- Static allocation: a programmer defines a fixed set of semaphores at compile time; the set does not change as the system runs.
- Dynamic allocation: the system includes functions that allow semaphores to be created on demand and deallocated when they are no longer needed.

The advantage of static allocation lies in saving space and reducing processing overhead — the system only contains memory for the needed semaphores, and the system does not require functions to allocate or deallocate semaphores. Thus, the smallest embedded systems use static allocation.

The chief advantage of dynamic allocation arises from the ability to accommodate new uses at runtime. For example, a dynamic allocation scheme allows a user to launch an application that allocates a semaphore, terminate the application, and then launch another application. Thus, larger embedded systems and most large operating systems provide dynamic allocation of resources, including semaphores. The next sections show that dynamic allocation does not introduce much additional code.

7.11 Example Implementation Of Dynamic Semaphores

Xinu provides a limited form of dynamic allocation: processes can create semaphores dynamically, and a given process can create multiple semaphores, provided the total number of semaphores allocated simultaneously does not exceed a predefined maximum. Furthermore, to minimize the allocation overhead, the system preallocates a list in the queue structure for each semaphore when the operating system boots. Thus, only a small amount of work needs be done when a process creates a semaphore.

Two system calls, *semcreate* and *semdelete*, handle dynamic semaphore allocation and deallocation. *Semcreate* takes an initial semaphore count as an argument, allocates a semaphore, assigns the semaphore the specified count, and returns the semaphore ID. To preserve the semaphore invariant, the initial count must be nonnegative. Therefore, *semcreate* begins by testing its argument. If the argument is valid, *semcreate* searches the semaphore table, *semtab*, for an unused entry and initializes the count. To search the table, *semcreate* calls function *newsem*, which iterates through all *NSEM* entries of the table. If no free entry is found, *newsem* returns *SYSERR*. Otherwise, *newsem* changes the state of the entry to *S_USED*, and returns the table index as the ID.

Once a table entry has been allocated, *semcreate* only needs to initialize the count and return the index of the semaphore to its caller; the head and tail of a queue used to store waiting processes have been allocated when the operating system boots. File *semcreate.c* contains the code for function *newsem* as well as function *semcreate*. Note the use of a static index variable *nextsem* to optimize searching (i.e., allow a search to start where the last search left off).

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

#include <xinu.h>

local   sid32   newsem(void);

/*------------------------------------------------------------------------
 *  semcreate  -  Create a new semaphore and return the ID to the caller
 *------------------------------------------------------------------------
 */
sid32   semcreate(
          int32         count           /* Initial semaphore count      */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        sid32   sem;                    /* Semaphore ID to return       */

        mask = disable();

        if (count < 0 || ((sem=newsem())==SYSERR)) {
                restore(mask);
                return SYSERR;
        }
        semtab[sem].scount = count;     /* Initialize table entry       */

        restore(mask);
        return sem;
}

/*------------------------------------------------------------------------
 *  newsem  -  Allocate an unused semaphore and return its index
 *------------------------------------------------------------------------
 */
local   sid32   newsem(void)
{
        static  sid32   nextsem = 0;    /* Next semaphore index to try  */
        sid32   sem;                    /* Semaphore ID to return       */
        int32   i;                      /* Iterate through # entries    */

        for (i=0 ; i<NSEM ; i++) {
                sem = nextsem++;
                if (nextsem >= NSEM)
                        nextsem = 0;
                if (semtab[sem].sstate == S_FREE) {
                        semtab[sem].sstate = S_USED;
                        return sem;
                }
        }
        return SYSERR;
}
```

7.12 Semaphore Deletion

Function *semdelete* reverses the actions of *semcreate*. *Semdelete* takes the ID of a semaphore as an argument and releases the semaphore table entry for subsequent use. Deallocating a semaphore requires three steps. First, *semdelete* verifies that the argument specifies a valid semaphore ID and that the corresponding entry in the semaphore table is currently in use. Second, *semdelete* sets the state of the entry to *S_FREE* to indicate that the table entry can be reused. Finally, *semdelete* iterates through the set of processes that are waiting on the semaphore and makes each process ready. File *semdelete.c* contains the code.

```
/* semdelete.c - semdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 * semdelete  -  Delete a semaphore by releasing its table entry
 *------------------------------------------------------------------------
 */
syscall semdelete(
          sid32         sem             /* ID of semaphore to delete    */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  sentry *semptr;         /* Ptr to semaphore table entry */

        mask = disable();
        if (isbadsem(sem)) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        if (semptr->sstate == S_FREE) {
                restore(mask);
                return SYSERR;
        }
        semptr->sstate = S_FREE;

        resched_cntl(DEFER_START);
        while (semptr->scount++ < 0) {  /* Free all waiting processes   */
                ready(getfirst(semptr->squeue));
        }
        resched_cntl(DEFER_STOP);
        restore(mask);
        return OK;
}
```

If processes remain enqueued on a semaphore when the semaphore is deallocated, an operating system must handle each of the processes. In the example implementation, *semdelete* places each waiting process back on the ready list, allowing the process to resume execution as if the semaphore had been signaled. The example only represents one strategy, and other strategies are possible. For example, some operating systems consider it an error to attempt to deallocate a semaphore on which processes are waiting. The exercises suggest exploring alternatives.

Note that the code to make processes ready uses deferred rescheduling. That is, *semdelete* calls *resched_cntl* to start deferral before making processes ready, and only calls *resched_cntl* to end the deferral period after all waiting processes have been moved to the ready list. The second call will invoke *resched* to reestablish the scheduling invariant.

7.13 Semaphore Reset

It is sometimes convenient to reset the count of a semaphore without incurring the overhead of deleting an existing semaphore and creating a new one. The system call *semreset*, shown in file *semreset.c* below, resets the count of a semaphore.

```
/* semreset.c - semreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  semreset  -  Reset a semaphore's count and release waiting processes
 *------------------------------------------------------------------------
 */
syscall semreset(
          sid32         sem,            /* ID of semaphore to reset     */
          int32         count           /* New count (must be >= 0)     */
        )
{
        intmask mask;                   /* Saved interrupt mask         */
        struct  sentry *semptr;         /* Ptr to semaphore table entry */
        qid16   semqueue;               /* Semaphore's process queue ID */
        pid32   pid;                    /* ID of a waiting process      */

        mask = disable();

        if (count < 0 || isbadsem(sem) || semtab[sem].sstate==S_FREE) {
                restore(mask);
                return SYSERR;
        }

        semptr = &semtab[sem];
        semqueue = semptr->squeue;      /* Free any waiting processes */
        resched_cntl(DEFER_START);
        while ((pid=getfirst(semqueue)) != EMPTY)
                ready(pid);
        semptr->scount = count;         /* Reset count as specified */
        resched_cntl(DEFER_STOP);
        restore(mask);
        return OK;
}
```

Semreset must preserve the semaphore invariant. Rather than build a general-purpose solution that allows a caller to specify an arbitrary semaphore count, our implementation takes a simplified approach by requiring the new count to be nonnegative. As a result, once the semaphore count has been changed, the queue of waiting processes will be empty. As with *semdelete*, *semreset* must be sure that no processes are already waiting on the semaphore. Thus, after checking its arguments and verifying that the semaphore exists, *semreset* iterates through the list of waiting processes, removing each from the semaphore queue and making the process ready to execute. As expected, *semreset* uses *resched_cntl* to defer rescheduling during the time processes are placed on the ready list.

7.14 Coordination Across Parallel Processors (Multicore)

The semaphore system described above works well on a computer that has a single processor core. However, many modern processor chips include multiple cores. One core is usually dedicated to run operating system functions, and other cores are used to execute user applications. On such systems, using semaphores supplied by the operating system to coordinate processes can be inefficient. To see why, consider what happens when an application running on core 2 needs exclusive access to a specific memory location. The application process calls *wait*, which must pass the request to the operating system on core 1. Communication among cores often involves raising an interrupt. Furthermore, while it runs an operating system function, core 1 disables interrupts, which defeats one of the reasons to use semaphores.

Some multiprocessor systems supply hardware primitives, known as *spin locks*, that allow multiple processors to contend for mutually exclusive access. The hardware defines a set of *K* spin locks (*K* might be less than 1024). Conceptually, each of the spin locks is a single bit, and the spin locks are initialized to zero. The instruction set includes a special instruction, called a *test-and-set*, that a core can use to coordinate. A test-and-set performs two operations atomically: it sets a spin lock to 1 and returns the value of the spin lock before the operation. The hardware guarantees atomicity, which means that if two or more processors attempt to set a given spin lock simultaneously, one of them will receive 0 as the previous value and the others will receive 1. Once it finishes using the locked item, the core that obtained the lock resets the value to 0, allowing another core to obtain the lock.

To see how spin locks work, suppose two cores need exclusive access to a shared data item and are using spin lock 5. When a core wants to obtain mutually exclusive access, the core executes a loop:†

†Because it uses hardware instructions, test-and-set code is usually written in assembly language; it is shown in pseudo code for clarity.

```
while (test_and_set(5)) {
        ;
}
```

The loop repeatedly uses the test-and-set instruction to set spin lock 5. If the lock was set before the instruction executed, the instruction will return 1, and the loop will continue. If the lock was not set before the instruction executed, the hardware will return 0, and the loop terminates. If multiple processors are all trying to set spin lock 5 at the same time, the hardware guarantees that only one will be granted access. Thus, test_and_set is analogous to *wait*.

Once a core finishes using the shared data, the core executes an instruction that clears the spin lock:

```
clear(5);
```

On multicore machines, vendors include various instructions that can be used as a spin lock. For example, in addition to *test-and-set*, Intel multicore processors provide an atomic *compare-and-swap* instruction. If multiple cores attempt to execute a compare-and-swap instruction at the same time, one of them will succeed and the others will all find that the comparison fails. A programmer can use such instructions to build the equivalent of a spin lock.

It may seem that a spin lock is wasteful because a processor merely blocks (i.e., busy waits) in a loop until access is granted. However, if the probability of two processors contending for access is low, a spin lock mechanism is much more efficient than a system call (e.g., waiting on a semaphore). Therefore, a programmer must be careful in choosing when to use spin locks and when to use system calls.

7.15 Perspective

The counting semaphore abstraction is significant for two reasons. First, it provides a powerful mechanism that can be used to control both mutual exclusion and producer–consumer synchronization, the two primary process coordination paradigms. Second, the implementation is surprisingly compact and extremely efficient. To appreciate the small size, reconsider functions *wait* and *signal*. If the code to test arguments and returns results is removed, only a few lines of code remain. As we examine the implementation of other abstractions, the point will become more significant: despite their importance, only a trivial amount of code is needed to implement counting semaphores.

7.16 Summary

Instead of disabling interrupts, which stops all activities other than the current process, operating systems offer synchronization primitives that allow subsets of processes to coordinate without affecting other processes. A fundamental coordination mechanism, known as a counting semaphore, allows processes to coordinate without using busy-waiting. Each semaphore consists of an integer count plus a queue of processes. The semaphore adheres to an invariant that specifies a count of negative *N* means the queue contains *N* processes.

The two fundamental primitives, *signal* and *wait*, permit a caller to increment or decrement the semaphore count. If a call to *wait* makes the semaphore count negative, the calling process is placed in the *waiting* state and the processor passes to another process. In essence, a process that waits for a semaphore voluntarily enqueues itself on the list of processes waiting for the semaphore, and calls *resched* to allow other processes to execute.

Either static or dynamic allocation can be used with semaphores. The example code includes functions *semcreate* and *semdelete* to permit dynamic allocation. If a semaphore is deallocated while processes are waiting, the processes must be handled. The example code makes the processes ready as if the semaphore had been signaled.

Multiprocessors can use a mutual exclusion mechanism known as spin locks. Although spin locks seem inefficient because they require a processor to repeatedly test for access, they can be more efficient than an arrangement where one processor interrupts another to place a system call.