Process Synchronization

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. The former case is achieved through the use of threads. Concurrent access to shared data may result in data inconsistency, however. Here, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

5.1 Background

Processes can execute concurrently or in parallel. We have already seen process scheduling and how the CPU scheduler switches rapidly between processes to provide concurrent execution. One process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process. Also, parallel execution, in which two instruction streams (representing different processes) execute simultaneously on separate processing cores. Here, we explain how concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.

Let's consider an example of how this can happen. Consider producer – consumer problem with bounded buffer. Our original solution allowed at most BUFFER SIZE − 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency. One possibility is to add an integer variable counter, initialized to 0. counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer. The code for the producer process can be modified as follows:

```
while (true) {
/* produce an item in next_produced */
while (counter == BUFFER_SIZE)
; /* do nothing */
buffer[in] = next_produced;
in = (in + 1) % BUFFER_SIZE;
counter++;
}
```

The code for the consumer process can be modified as follows:

```
while (true) {
while (counter == 0)
; /* do nothing */
Next_consumed = buffer[out];
out = (out + 1) % BUFFER_SIZE;
counter--;
/* consume the item in next consumed */
}
```

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes concurrently execute the statements "counter++" and "counter--". Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6! The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately. We might arrive at the incorrect state "counter == 4", indicating that four buffers are full, when, in fact, five buffers are full.

We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter concurrently. A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**. To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

Situations such as the one just described occur frequently in OS as different parts of the system manipulate resources. Furthermore, the growing importance of multicore systems has brought an increased emphasis on developing multithreaded applications. In such applications, several threads which are quite possibly sharing data — are running in parallel on different processing cores. Clearly, we want any changes that result from such activities not to interfere with one another. Because of the importance of this issue, **process synchronization** and **coordination** among cooperating processes are introduced.

5.2 The Critical-Section Problem

Consider a system consisting of n processes $\{ P_0, P_1, ..., P_{n-1}\}$. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its

critical section, no other process is allowed to execute in its critical section. That is, no two processes are executing in their critical sections at the same time. The ***critical-section problem*** is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**. The general structure of a typical process P_i is shown in Figure 5.1.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

Figure 5.1 General structure of a typical process P_i.

A solution to the critical-section problem must satisfy the following three requirements:

Mutual exclusion. If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

Progress. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely.

Bounded waiting. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

At a given point in time, many kernel-mode processes may be active in the operating system. As a result, the code implementing an operating system (***kernel code***) is subject to several possible race conditions. Consider a kernel data structure that maintains a list of all open files in the system. This list must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition. Other kernel data

structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.

Two general approaches are used to handle critical sections in operating systems: **preemptive kernels** and **nonpreemptive kernels**. A preemptive kernel allows a process to be preempted while it is running in kernel mode. A nonpreemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

A nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time. We cannot say the same about preemptive kernels, so they must be carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

A preemptive kernel may be more responsive, since there is less risk that a kernel-mode process will run for an arbitrarily long period before relinquishing the processor to waiting processes. Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

5.3 Peterson's Solution

A classic software-based solution to the critical-section problem is **Peterson's solution**. Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered P_0 and P_1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, j equals $1 - i$.

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);

        critical section

    flag[i] = false;

        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.

Peterson's solution requires the two processes to share two data items:

```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section. If turn == i, then process P_i is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[i] is true, this value indicates that P_i is ready to enter its critical section.

To enter the critical section, process P_i first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately. The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.
2. The progress requirement is satisfied.
3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either flag[j] == false or turn == i. If both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that P_0 and P_1 could not have successfully executed their while statements at about the same time, since the value of turn can be either 0 or 1 but cannot be both. Hence, one of the processes — say, P_j — must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this

condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If P_j is not ready to enter the critical section, then flag[j] == false, and P_i can enter its critical section. If P_j has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then P_i will enter the critical section. If turn == j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section. If P_j resets flag[j] to true, it must also set turn to i. Thus, since P_i does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

5.4 Synchronization Hardware

Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures. Here, we explore several more solutions to the critical-section problem using techniques ranging from hardware to software-based APIs available to both kernel developers and application programmers. All these solutions are based on the premise of **locking** — that is, protecting critical regions through the use of locks. The designs of such locks can be quite sophisticated.

We start by presenting some simple hardware instructions that are available on many systems and how they can be used effectively in solving the critical-section problem. The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. In this way, we could be sure that the current sequence of instructions would be allowed to execute in order without pre-emption. No other instructions would be run, so no unexpected modifications could be made to the shared variable. This is often the approach taken by nonpreemptive kernels. Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words **atomically** — that is, as one uninterruptible unit. We can

use these special instructions to solve the critical-section problem in a relatively simple manner. We abstract the main concepts behind these types of instructions by describing the test_and_set() and compare_and_swap() instructions.

The test_and_set() instruction can be defined as shown in Figure 5.3. The important characteristic of this instruction is that it is executed atomically. Thus, if two test and set() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order. If the machine supports the test and set() instruction, then we can implement mutual exclusion by declaring a boolean variable lock, initialized to false. The structure of process P_i is shown in Figure 5.4.

```
boolean test_and_set(boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

Figure 5.3 The definition of the test_and_set() instruction.

```
do {
while (test_and_set(&lock))
; /* do nothing */
/* critical section */
lock = false;
/* remainder section */
} while (true);
```

Figure 5.4 Mutual-exclusion implementation with test_and_set().

The compare_and_swap() instruction, in contrast to the test_and_set() instruction, operates on three operands; it is defined in Figure 5.5. The operand value is set to new value only if the expression (*value == exected) is true. Regardless, compare and swap() always returns the original value of the variable value. Like the test and set() instruction, compare and swap() is executed atomically. Mutual exclusion can be provided as follows: a global variable (lock) is declared and is initialized to 0. The first process that invokes compare and swap() will set lock to 1. It will then enter its critical section, because the original value of lock was equal to the expected value of 0. Subsequent calls to compare and swap() will not succeed, because lock now is not equal to the expected value of 0. When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section. The structure of process P_i is shown in Figure 5.6.

```
int compare_and_swap(int *value, int expected, int new_value) {
int temp = *value;
if (*value == expected)
*value = new_value;
return temp;
}
```

Figure 5.5 The definition of the compare and swap() instruction.

```
do {
while (compare_and_swap(&lock, 0, 1) != 0)
; /* do nothing */
        /* critical section */
    lock = 0;
        /* remainder section */
} while (true);
```

Figure 5.6 Mutual-exclusion implementation with the compare_and_swap() instruction.

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. In Figure 5.7, we present another algorithm using the test_and_set() instruction that satisfies all the critical-section requirements. The common data structures are

```
do {
                waiting[i] = true;
                key = true;
                while (waiting[i] && key)
                    key = test_and_set(&lock);
                    waiting[i] = false;
                    /* critical section */
                j = (i + 1) % n;
                while ((j != i) && !waiting[j])
                    j = (j + 1) % n;
                if (j == i)
                    lock = false;
                else
                    waiting[j] = false;

/* remainder section */
            } while (true);
```

Figure 5.7 Bounded-waiting mutual exclusion with test_and_set().

boolean waiting[n];
boolean lock;

These data structures are initialized to false. To prove that the mutual-exclusion requirement is met, we note that process P_i can enter its critical section only if either waiting[i] == false or key == false. The value of key can become false only if the test and set() is executed. The first process to execute the test_and_set() will find key == false; all others must wait. The variable waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual-exclusion requirement.

To prove that the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed.

To prove that the bounded-waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering ($i + 1, i + 2, ..., n - 1, 0, ..., i - 1$). It designates the first process in this ordering (waiting[j] == true) as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

5.5 Mutex Locks

Operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the **mutex lock**. We use the mutex lock to protect critical regions and thus prevent race conditions. That is, a process must acquire the lock before entering a critical section; it releases the lock when it exits the critical section. The acquire()function acquires the lock, and the release() function releases the lock, as illustrated in Figure 5.8.

A mutex lock has a boolean variable available whose value indicates if the lock is available or not. If the lock is available, a call to acquire() succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released. The definition of acquire() is as follows:

```
acquire() {
      while (!available)
            /* busy wait */
available = false;;     }
   do {
         acquire lock
```

```
        critical section
    release lock
        remainder section
} while (true);
```

Figure 5.8 Solution to the critical-section problem using mutex locks.

The definition of release() is as follows:

```
release() {
    available = true;  }
```

Calls to either acquire() or release() must be performed atomically. Thus, mutex locks are often implemented using one of the hardware mechanisms described in the previous topic.

The main disadvantage of the implementation given here is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the call to acquire(). In fact, this type of mutex lock is also called a **spinlock** because the process "spins" while waiting for the lock to become available. This continual looping is a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively.

Spinlocks do have an **advantage**, in that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful. They are often employed on multiprocessor systems where one thread can "spin" on one processor while another thread performs its critical section on another processor.

5.6 Semaphores

Mutex locks, are considered the simplest of synchronization tools. Here, we examine a more robust tool that can behave similarly to a mutex lock but can also provide more sophisticated ways for processes to synchronize their activities.

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal(). The wait() operation was originally termed P ("to test"); signal() was originally called V ("to increment"). The definition of wait() is as follows:

```
wait(S) {
    while (S <= 0)
        // busy wait
    S--;  }
```

The definition of signal() is as follows:

```
signal(S) {
S++; }
```

All modifications to the integer value of the semaphore in the wait() and signal() operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of wait(S), the testing of the integer value of S ($S \leq 0$), as well as its possible modification (S--), must be executed without interruption. Let's see how semaphores can be used.

5.6.1 Semaphore Usage

Operating systems often distinguish between counting and binary semaphores. The value of a **counting semaphore** can range over an unrestricted domain. The value of a **binary semaphore** can range only between 0 and 1. Thus, binary semaphores behave similarly to mutex locks. On systems that do not provide mutex locks, binary semaphores can be used for providing mutual exclusion.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a signal() operation (incrementing the count). When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1 and P_2 with a statement S_2. Suppose we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore synch, initialized to 0. In process P_1, we insert the statements

```
S1;
signal(synch);
```

In process P_2, we insert the statements

```
wait(synch);
S2;
```

Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after statement S_1 has been executed.

5.6.2 Semaphore Implementation

The definitions of the wait() and signal() semaphore operations suffers from busy waiting problem. To overcome the need for busy waiting, we can modify the definition of the wait() and signal() operations as follows: When a process executes the wait() operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal() operation. The process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. To implement semaphores under this definition, we define a semaphore as follows:

```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

Each semaphore has an integer value and a list of processes list. When a process must wait on a semaphore, it is added to the list of processes. A signal() operation removes one process from the list of waiting processes and awakens that process. Now, the wait() semaphore operation can be defined as

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
                add this process to S->list;
                block(); } }
```

and the signal() semaphore operation can be defined as

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
```

remove a process *P* from S->list;
wakeup(P); *} }*

The block() operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

In this implementation, semaphore values may be negative, whereas semaphore values are never negative under the classical definition of semaphores with busy waiting. If a semaphore value is negative, its magnitude is the number of processes waiting on that semaphore. This fact results from switching the order of the decrement and the test in the implementation of the wait() operation.

The list of waiting processes can be easily implemented by a link field in each process control block (PCB). Each semaphore contains an integer value and a pointer to a list of PCBs. One way to add and remove processes from the list so as to ensure bounded waiting is to use a FIFO queue, where the semaphore contains both head and tail pointers to the queue.

It is critical that semaphore operations be executed atomically. We must guarantee that no two processes can execute wait()_and_signal() operations on the same semaphore at the same time. This is a critical-section problem; and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the wait()_and_signal() operations are executing. This scheme works in a single-processor environment because, once interrupts are inhibited, instructions from different processes cannot be interleaved. Only the currently running process executes until interrupts are re-enabled and the scheduler can regain control.

In a multiprocessor environment, interrupts must be disabled on every processor. Otherwise, instructions from different processes (running on different processors) may be interleaved in some arbitrary way. Disabling interrupts on every processor can be a difficult task and can seriously diminish performance. Therefore, SMP systems must provide alternative locking techniques — such as compare_and_swap() or spinlocks — to ensure that wait() and signal() are performed atomically.

We have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait() and signal() operations,

and these sections are short. Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time. An entirely different situation exists with application programs whose critical sections may be long or may almost always be occupied. In such cases, busy waiting is extremely inefficient.

5.6.3 Deadlocks and Starvation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be **deadlocked**.

To illustrate this, consider a system consisting of two processes, P_0 and P_1, each accessing two semaphores, S and Q, set to the value 1:

P_0	P_1
wait(S);	wait(Q);
wait(Q);	wait(S);
.	.
.	.
signal(S);	signal(Q);
signal(Q);	signal(S);

Suppose that P_0 executes wait(S) and then P_1 executes wait(Q). When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S). Since these signal() operations cannot be executed, P_0 and P_1 are deadlocked.

We say that a set of processes is in a deadlocked state when every process in the set is waiting for an event that can be caused only by another process in the set. The events with which we are mainly concerned here are resource acquisition and release. Another problem related to deadlocks is **indefinite blocking** or **starvation**, a situation in which processes wait indefinitely within the semaphore. Indefinite blocking may occur if we remove processes from the list associated with a semaphore in LIFO (last-in, first-out) order.

5.6.4 Priority Inversion

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process.

Since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority.

As an example, assume we have three processes — *L*, *M*, and *H* — whose priorities follow the order $L < M < H$. Assume that process *H* requires resource *R*, which is currently being accessed by process *L*. Ordinarily, process *H* would wait for *L* to finish using resource *R*. However, now suppose that process *M* becomes runnable, thereby preempting process *L*. Indirectly, a process with a lower priority — process *M*— has affected how long process *H* must wait for *L* to relinquish resource *R*.

This problem is known as **priority inversion**. It occurs only in systems with more than two priorities, so one solution is to have only two priorities. However, That is insufficient for most general-purpose operating systems. Typically these systems solve the problem by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol would allow process *L* to temporarily inherit the priority of process *H*, thereby preventing process *M* from preempting its execution. When process had finished using resource *R*, it would relinquish its inherited priority from *H* and assume its original priority. Because resource *R* would now be available, process *H* — not *M*— would run next.

5.7 Classical Problems of Synchronization using semaphores

5.7.1 The Bounded-Buffer Problem

The ***bounded-buffer problem*** is commonly used to illustrate the power of synchronization primitives. In our problem, the **producer and consumer** processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number

of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

```
do {
    . . .
   /* produce an item in next produced */
   wait(empty);
   wait(mutex);
    . . .
   /* add next produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
} while (true);
```

Figure 5.9 The structure of the producer process.

The code for the producer process is shown in Figure 5.9, and the code for the consumer process is shown in Figure 5.10.

```
do {
   wait(full);
   wait(mutex);
    . . .
/* remove an item from buffer to next_consumed */
   . . .
   signal(mutex);
   signal(empty);
    . . .
   /* consume the item in next_consumed */
   . . .
} while (true);
```

Figure 5.10 The structure of the consumer process.

5.7.2 Readers – Writers problem

Suppose that a database is to be shared among several concurrent processes. Some of these processes may want only to read the database, whereas others may want to update the database. We refer to the former as readers and to the latter as writers. If two readers access the shared data simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared database while writing to the database. This synchronization problem is referred to as the readers– writers problem. The readers– writers problem has several variations, all involving priorities. The simplest one, referred to as the first readers– writers problem, requires that no reader be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers – writers problem requires that, once a writer is ready, that writer perform its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. Now, we present a solution to the first readers– writers problem. In the solution to the first readers– writers problem, the reader processes share the following data structures:

```
semaphore rw_mutex = 1;
semaphore mutex =  1;
int  read _count  = 0;
```

The semaphores mutex and rw_mutex are initialized to 1; read count is initialized to 0. The semaphore rw_mutex is common to both reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable read count is updated. The read count variable keeps track of how many processes are currently reading the object. The semaphore rw_mutex functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

```
do {
wait(rw_mutex);
. . .
/* writing is performed  */
. . .
signal(rw mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

```
do { wait(mutex); read count++;
if (read count == 1) wait(rw mutex);
signal(mutex);
```

```
. . .
/* reading is performed */    ---
wait(mutex); read count--;
if (read count == 0) signal(rw  mutex);
signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

The code for a writer process is shown in Figure 5.11; the code for a reader process is shown in Figure 5.12. If a writer is in the critical section and n readers are waiting, then one reader is queued on rw_mutex, and n − 1 readers are queued on mutex. Also, when a writer executes signal(rw_mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

The readers– writers problem and its solutions have been generalized to provide reader – writer locks on some systems. Acquiring a reader– writer lock requires specifying the mode of the lock: either read or write access. When a process wishes only to read shared data, it requests the reader– writer lock in read mode. A process wishing to modify the shared data must request the lock in write mode. Multiple processes are permitted to concurrently acquire a reader– writer lock in read mode, but only one process may acquire the lock for writing, as exclusive access is required for writers.

Reader– writer locks are most useful in the following situations:

- In applications where it is easy to identify which processes only read shared data and which processes only write shared data.
- In applications that have more readers than writers. This is because reader–writer locks generally require more overhead to establish than semaphores or mutual-exclusion locks. The increased concurrency of allowing multiple readers compensates for the overhead involved in setting up the reader–writer lock.

5.7.3 Dining Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 5.13). When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up

only one chopstick at a time. She cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again. The dining-philosophers problem is considered a classic synchronization problem because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner.

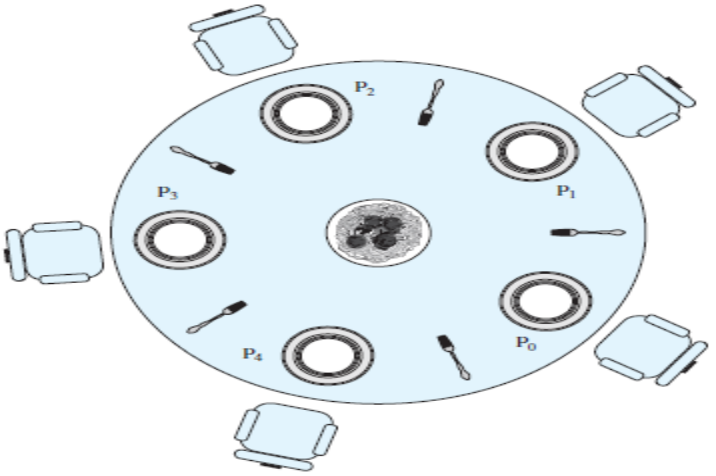

Figure 6.11 Dining Arrangement for Philosophers

One simple solution is to represent each chopstick with a semaphore. A philosopher tries to grab a chopstick by executing a wait() operation on that semaphore. She releases her chopsticks by executing the signal() operation on the appropriate semaphores. Thus, the shared data are

semaphore chopstick[5];

where all the elements of chopstick are initialized to 1. The structure of philosopher i is shown in Figure 5.14.

```
do {
wait(chopstick[i]);
wait(chopstick[(i+1) % 5]);
. . .
/* eat for a while */
. . .
signal(chopstick[i]);
signal(chopstick[(i+1)  %  5]);
. . .
/* think for a while */
. . .
} while (true);
```

Figure 5.14 The structure of philosopher i.

Although this solution guarantees that no two neighbors are eating simultaneously, it must be rejected because it could create a deadlock. Suppose that all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Several possible remedies to the deadlock problem are replaced by:

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this, she must pick them up in a critical section).
- Use an asymmetric solution— that is, an odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even-numbered philosopher picks up her right chopstick and then her left chopstick.

5.8 Monitors

Although semaphores provide a convenient and effective mechanism for process synchronization, using them incorrectly can result in timing errors that are difficult to detect, since these errors happen only if particular execution sequences take place and these sequences do not always occur. To illustrate how, we review the semaphore solution to the critical-section problem. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously. Next, we examine the various difficulties that may result. These difficulties will arise even if a ***single*** process is not well behaved. This situation may be caused by an honest programming error or an uncooperative programmer.

- Suppose that a process interchanges the order in which the wait() and signal() operations on the semaphore mutex are executed, resulting in the following execution:

```
signal(mutex);
...
critical section
...
wait(mutex);
```

In this situation, several processes may be executing in their critical sections simultaneously, violating the mutual-exclusion requirement. This error may be

discovered only if several processes are simultaneously active in their critical sections.

- Suppose that a process replaces signal(mutex) with wait(mutex). That is, it executes

```
wait(mutex);
    ...
  critical section
    ...
wait(mutex);
```

In this case, a deadlock will occur.

- Suppose that a process omits the wait(mutex), or the signal(mutex), or both. In this case, either mutual exclusion is violated or a deadlock will occur.

These examples illustrate that various types of errors can be generated easily when programmers use semaphores incorrectly to solve the critical-section problem. To deal with such errors, researchers have developed high-level language constructs. One fundamental high-level synchronization construct is the **monitor** type.

5.8.1 Monitor Usage

An **abstract data type** encapsulates data with a set of functions to operate on that data that are independent of any specific implementation of the ADT. A ***monitor type*** is an ADT that includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor. The monitor type also declares the variables whose values define the state of an instance of that type, along with the bodies of functions that operate on those variables. The syntax of a monitor type is shown in Figure 5.15. The representation of a monitor type cannot be used directly by the various processes. Thus, a function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can be accessed by only the local functions.

```
monitor monitor name
{
    /* shared variable declarations */
    function P1 ( . . . ) {
```

```
    . . .    }
function P2 ( . . . ) {
    . . .    }
        .
        .
function Pn ( . . . ) {
    . . .    }
initialization code ( . . . ) {
  . . . } }
```

Figure 5.15 Syntax of a monitor.

The monitor construct ensures that only one process at a time is active within the monitor. The programmer does not need to code this synchronization constraint explicitly (Figure 5.16).

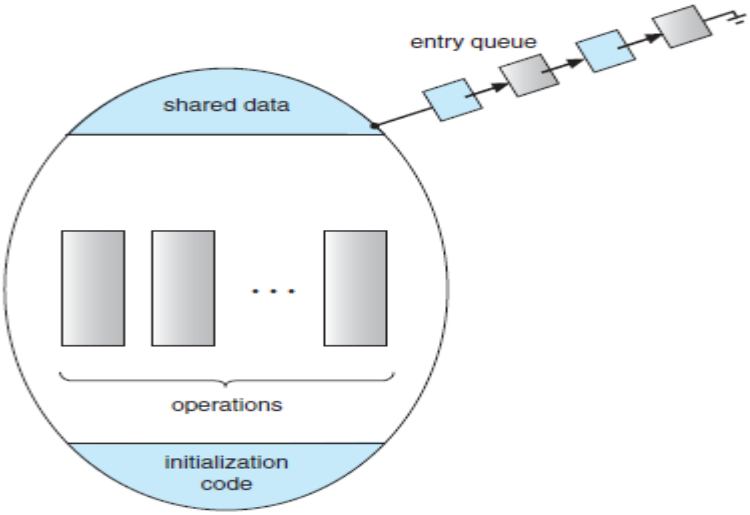

Figure 5.16 Schematic view of a monitor.

However, the monitor construct, is not sufficiently powerful for modeling some synchronization schemes. For this purpose, we need to define additional synchronization mechanisms. These mechanisms are provided by the condition construct. A programmer who needs to write a tailor-made synchronization scheme can define one or more variables of type *condition*: condition x, y;

The only operations that can be invoked on a condition variable are wait() and signal(). The operation
x.wait(); - means that the process invoking this operation is suspended until another process invokes x.signal();
The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal() operation has no effect; that is, the state of x is the same as if the operation had never been executed (Figure 5.17).

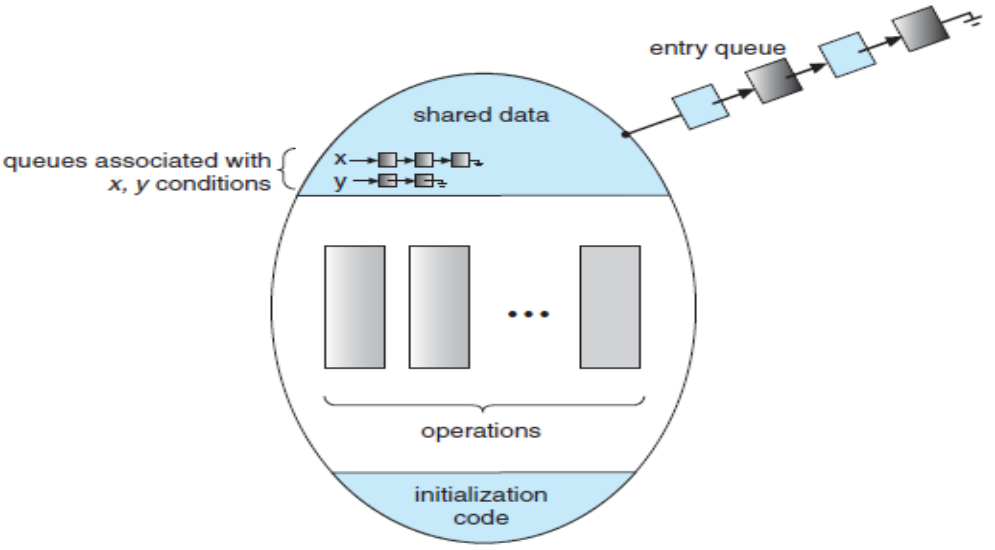

Figure 5.17 Monitor with condition variables.

Now suppose that, when the x.signal() operation is invoked by a process *P*, there exists a suspended process *Q* associated with condition x. Clearly, if the suspended process *Q* is allowed to resume its execution, the signaling process *P* must wait. Otherwise, both *P* and *Q* would be active simultaneously within the monitor. Conceptually both processes can continue with their execution. Two possibilities exist:

Signal and wait. *P* either waits until *Q* leaves the monitor or waits for another condition.
Signal and continue. *Q* either waits until *P* leaves the monitor or waits for another condition.

There are reasonable arguments in favor of adopting either option. On the one hand, since *P* was already executing in the monitor, the ***signal-and-continue*** method seems more reasonable. On the other, if we allow thread *P* to continue, then by the time *Q* is resumed, the logical condition for which *Q* was waiting may no longer hold. When thread *P* executes the signal operation, it immediately leaves the monitor. Hence, *Q* is immediately resumed.

5.8.2 Dining-Philosophers Solution Using Monitors

This solution imposes the restriction that a philosopher may pick up her chopsticks only if both of them are available. To code this solution, we need to distinguish among three states in which we may find a philosopher. For this purpose, we introduce the following data structure:

enum {THINKING, HUNGRY, EATING} state[5];

Philosopher *i* can set the variable state[i] = EATING only if her two neighbors are not eating: (state[(i+4) % 5] != EATING) and (state[(i+1) % 5] != EATING). We also need to declare

condition self[5];

This allows philosopher *i* to delay herself when she is hungry but is unable to obtain the chopsticks she needs.

```
monitor DiningPhilosophers
{
   enum {THINKING, HUNGRY, EATING} state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Figure 5.18 A monitor solution to the dining-philosopher problem.

The distribution of the chopsticks is controlled by the monitor DiningPhilosophers, whose definition is shown in Figure 5.18. Each philosopher, before starting to eat, must invoke the operation pickup(). This act may result in the suspension of the philosopher process. After the successful completion of the operation, the philosopher may eat. Following this, the philosopher invokes the putdown() operation. Thus, philosopher *i* must invoke the operations pickup() and putdown() in the following sequence:

DiningPhilosophers.pickup(i);

...

eat

...

DiningPhilosophers.putdown(i);

It is easy to show that this solution ensures that no two neighbors are eating simultaneously and that no deadlocks will occur. However, it is possible for a philosopher to starve to death.

5.8.3 Implementing a Monitor Using Semaphores

For each monitor, a semaphore mutex (initialized to 1) is provided. A process must execute wait(mutex) before entering the monitor and must execute signal(mutex) after leaving the monitor. Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0. The signaling processes can use next to suspend themselves. An integer variable next_count is also provided to count the number of processes suspended on next. Thus, each external function F is replaced by

```
wait(mutex);
    ...
  body of F
    ...
if (next_count > 0)
signal(next);
else
  signal(mutex);
```

Mutual exclusion within a monitor is ensured.

For each condition x, we introduce a semaphore x_sem and an integer variable x_count, both initialized to 0. The operation x_wait() can now be implemented as

```
  x_count++;
if (next_count > 0)
  signal(next);
else
  signal(mutex);
wait(x_sem);
  x count--;
```

The operation x.signal() can be implemented as

```
if (x_count > 0) {
next_count++;
signal(x_sem);
wait(next);
next_count--;    }
```

This implementation is applicable to the definitions of monitors.

5.8.4 Resuming Processes within a Monitor

If several processes are suspended on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes should be resumed next? One simple solution is to use a FCFS ordering, so that the process that has been waiting the longest is resumed first. In many circumstances, however, such a simple scheduling scheme is not adequate. For this purpose, the **conditional-wait** construct can be used. This construct has the form

x.wait(c);

where c is an integer expression that is evaluated when the wait() operation is executed. The value of c, which is called a **priority number**, is then stored with the name of the process that is suspended. When x.signal() is executed, the process with the smallest priority number is resumed next.

To illustrate this new mechanism, consider the ResourceAllocator monitor shown in Figure 5.19, which controls the allocation of a single resource among competing processes.

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }
    void release() {
        busy = false;
        x.signal();
    }
    initialization code() {
busy = false;
    } }
```

Figure 5.19 A monitor to allocate a single resource.

Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource. The monitor allocates the resource to the process that has the shortest time-allocation request. A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
    ...
  access the resource;
    ...
R.release();
```

where R is an instance of type ResourceAllocator.

Unfortunately, the monitor concept cannot guarantee that the preceding access sequence will be observed. In particular, the following problems can occur:

- A process might access a resource without first gaining access permission to the resource.
- A process might never release a resource once it has been granted access to the resource.
- A process might attempt to release a resource that it never requested.
- A process might request the same resource twice (without first releasing the resource).

The same difficulties are encountered with the use of semaphores, and these difficulties are similar in nature to those that encouraged us to develop the monitor constructs in the first place. Previously, we had to worry about the correct use of semaphores. Now, we have to worry about the correct use of higher-level programmer-defined operations, with which the compiler can no longer assist us.

One possible solution to the current problem is to include the resource-access operations within the ResourceAllocator monitor. However, using this solution will mean that scheduling is done according to the built-in monitor-scheduling algorithm rather than the one we have coded.

To ensure that the processes observe the appropriate sequences, we must inspect all the programs that make use of the ResourceAllocator monitor and its managed resource. We must check two conditions to establish the correctness of this system.

First, user processes must always make their calls on the monitor in a correct sequence. **Second**, we must be sure that an uncooperative process does not simply ignore the mutual-exclusion gateway provided by the monitor and try to access the shared resource directly, without using the access protocols. Only if these two conditions can be ensured can we guarantee that no time-dependent errors will occur and that the scheduling algorithm will not be defeated. Although this inspection may be possible for a small, static system, it is not reasonable for a large system or a dynamic system.