**Synchronization Hardware**

We have just described one software-based solution to the critical-section problem. However, as mentioned, software-based solutions such as Peterson’s are not guaranteed to work on modern computer architectures. In the following discussions, 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. As we shall see, the designs of such locks can be quite sophisticated.

We start by presenting some simple hardware instructions that are available on many systems and showing how they can be used effectively in solving the critical-section problem. Hardware features can make any programming task easier and improve system efficiency.

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 preemption. 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.

boolean test and set(boolean \*target) {

boolean rv = \*target;

\*target = true;

return rv;

}

Figure 1.72 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 1.73 Mutual-exclusion implementation with test and set().

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. Also consider the effect on a system’s clock if the clock is kept updated by interrupts.

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. Rather than discussing one specific instruction for one specific machine, 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 1.72. 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 Pi is shown in Figure 1.73.

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

int compare and swap(int \*value, int expected, int new value) {

int temp = \*value;

if (\*value == expected)

\*value = new value;

return temp;

}

Figure 1.74 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 1.75 Mutual-exclusion implementation with the compare and swap() instruction.

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 Pi is shown in Figure 1.75.

Although these algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement. In Figure 1.76, we present another algorithm using the test and set() instruction that satisfies all the critical-section requirements. The common data structures are

boolean waiting[n];

boolean lock;

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 1.76 Bounded-waiting mutual exclusion with test and set().

These data structures are initialized to false. To prove that the mutualexclusion requirement is met, we note that process Pi 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 that is in the entry section (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.

Details describing the implementation of the atomic test and set() and compare and swap() instructions are discussed more fully in books on computer architecture.

**Mutex Locks**

The hardware-based solutions to the critical-section problem presented are complicated as well as generally inaccessible to application programmers. Instead, operating-systems designers build software tools to solve the critical-section problem. The simplest of these tools is the mutex lock. (In fact, the term mutex is short for mutual exclusion.) 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 1.77.

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;;

}

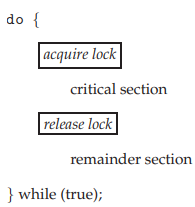


Figure 1.77 Solution to the critical-section problem using mutex locks.

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.

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. (We see the same issue with the code examples illustrating the test and set() instruction and the compare and swap() instruction.) This continual looping is clearly 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, however, 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.

**Semaphores**

Mutex locks, as we mentioned earlier, are generally considered the simplest of synchronization tools. In this section, 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 (from the Dutch proberen, “to test”); signal() was originally called V (from verhogen, “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 ≤ 0), as well as its possible modification (S--), must be executed without interruption. We shall see how these operations can be implemented in Section 5.6.2. First, let’s see how semaphores can be used.

**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. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead 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: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0. In process P1, we insert the statements

S1;

signal(synch);

In process P2, we insert the statements

wait(synch);

S2;

Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal(synch), which is after statement S1 has been executed.

**Semaphore Implementation**

Recall that the implementation of mutex locks suffers from busy waiting. The definitions of the wait() and signal() semaphore operations just described present the same 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. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

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.

Note that 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. In general, however, the list can use any queueing strategy. Correct usage of semaphores does not depend on a particular queueing strategy for the semaphore lists.

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 reenabled 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 furthermore 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.

It is important to admit that 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 (if properly coded, they should be no more than about ten instructions). 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 (minutes or even hours) or may almost always be occupied. In such cases, busy waiting is extremely inefficient.