CHAPTER 6

Synchronization Tools

Practice Exercises

6.1 In Section 6.4, we mentioned that disabling interrupts frequently can affect the system's clock. Explain why this can occur and how such effects can be minimized.

Answer:
The system clock is updated at every clock interrupt. If interrupts were disabled—particularly for a long period of time—the system clock could easily lose the correct time. The system clock is also used for scheduling purposes. For example, the time quantum for a process is expressed as a number of clock ticks. At every clock interrupt, the scheduler determines if the time quantum for the currently running process has expired. If clock interrupts were disabled, the scheduler could not accurately assign time quanta. This effect can be minimized by disabling clock interrupts for only very short periods.

6.2 What is the meaning of the term ***busy waiting***? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.

Answer:
Busy waiting means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. One strategy to avoid busy waiting temporarily puts the waiting process to sleep and awakens it when the appropriate program state is reached, but this solution incurs the overhead associated with putting the process to sleep and later waking it up.

6.3 Explain why spinlocks are not appropriate for single-processor systems yet are often used in multiprocessor systems.

Answer:
Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock can be obtained only by executing a different process. If the process is not relinquishing the processor, other processes do not get the opportunity to set the

program condition required for the first process to make progress. In a multiprocessor system, other processes execute on other processors and therefore can modify the program state in order to release the first process from the spinlock.

6.4 Show that, if the `wait()` and `signal()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

Answer:
A `wait()` operation atomically decrements the value associated with a semaphore. If two `wait()` operations are executed on a semaphore when its value is 1 and the operations are not performed atomically, then both operations might decrement the semaphore value, thereby violating mutual exclusion.

6.5 Illustrate how a binary semaphore can be used to implement mutual exclusion among n processes.

Answer:
The n processes share a semaphore, `mutex`, initialized to 1. Each process P_i is organized as follows:

```
do {
   wait(mutex);

      /* critical section */

   signal(mutex);

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

6.6 Race conditions are possible in many computer systems. Consider a banking system that maintains an account balance with two functions: `deposit(amount)` and `withdraw(amount)`. These two functions are passed the `amount` that is to be deposited or withdrawn from the bank account balance. Assume that a husband and wife share a bank account. Concurrently, the husband calls the `withdraw()` function, and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

Answer:
Assume that the balance in the account is \$250.00 and that the husband calls `withdraw($50)` and the wife calls `deposit($100)`. Obviously, the correct value should be \$300.00. Since these two transactions will be serialized, the local value of the balance for the husband becomes \$200.00, but before he can commit the transaction, the `deposit(100)` operation takes place and updates the shared value of the balance to \$300.00. We then switch back to the husband, and the value of the shared balance is set to \$200.00—obviously an incorrect value.

```
push(item) {
   acquire();
   if (top < SIZE) {
      stack[top] = item;
      top++;
   }
   else
      ERROR
   release();
}

pop() {
   acquire();
   if (!is_empty()) {
      top--;
      item = stack[top];
      release();
      return item;
   }
   else
      ERROR
   release();
}

is_empty() {
   if (top == 0)
      return true;
   else
      return false;
}
```

Figure 6.15 Array-based stack for Exercise 6.7.

Exercises

6.7 The pseudocode of Figure 6.15 illustrates the basic `push()` and `pop()` operations of an array-based stack. Assuming that this algorithm could be used in a concurrent environment, answer the following questions:

a. What data have a race condition?

b. How could the race condition be fixed?

Answer:

a. There is a race condition on the variable `top`. Assume that the `push()` and `pop()` functions are called concurrently. Before the

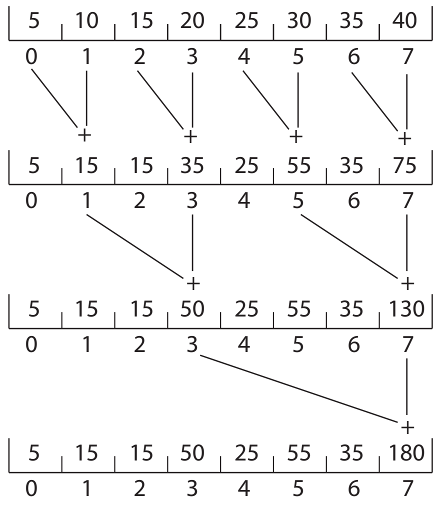

Figure 6.16 Summing an array as a series of partial sums for Exercise 6.8.

`stack[top] = item` statement can be performed, the `top--` statement is performed in `pop()`. The item being pushed will then replace what currently exists at location `stack[top]`.

b. The race condition could be fixed with mutex locks.

6.8 The following program example can be used to sum the array `values` of size *N* elements in parallel on a system containing *N* computing cores (there is a separate processor for each array element):

```
for j = 1 to log_2(N) {
   for k = 1 to N {
      if ((k + 1) % pow(2,j) == 0) {
         values[k] += values[k - pow(2,(j-1))]
      }
   }
}
```

This has the effect of summing the elements in the array as a series of partial sums, as shown in Figure 6.16. After the code has executed, the sum of all elements in the array is stored in the last array location. Are there any race conditions in the above code example? If so, identify where they occur and illustrate with an example. If not, demonstrate why this algorithm is free from race conditions.

Answer:
Yes, there is a race condition on the line

```
val[k] = val[k - pow(2,(j-1))] + val[k]
```

If the execution of the threads becomes interleaved, it is possible that one thread will have completed its assigned partial summation before the other thread has completed. As an example, 50 contains the sum of the partial sums 15 and 35 (which are stored at array locations 1 and 3, respectively). If 15 has not yet been calculated, the partial sum 50 will instead be $10 + 35 = 45$.

6.9 One approach for using `compare_and_swap()` for implementing a spinlock is as follows:

```
void lock_spinlock(int *lock) {
  while (compare_and_swap(lock, 0, 1) != 0)
    ; /* spin */
}
```

A suggested alternative approach is to use the "compare and compare-and-swap" idiom, which checks the status of the lock before invoking the `compare_and_swap()` operation. (The rationale behind this approach is to invoke `compare_and_swap()`only if the lock is currently available.) This strategy is shown below:

```
void lock_spinlock(int *lock) {
{
  while (true) {
    if (*lock == 0) {
      /* lock appears to be available */

      if (!compare_and_swap(lock, 0, 1))
        break;
    }
  }
}
```

Does this "compare and compare-and-swap" idiom work appropriately for implementing spinlocks? If so, explain. If not, illustrate how the integrity of the lock is compromised.

Answer:

This idiom works appropriately. Assume that Thread 1 is in the function and the value of lock equals 0. A context switch occurs to Thread 2, which also enters the function. Thread 2 still sees lock as having value 0 and enters the `compare_and_swap()` function, sets lock to 1, and returns 0, which breaks Thread 2 out of the while loop. When control switches back to Thread 1 (where Thread 1 still thinks lock is equal to 0), it too invokes `compare_and_swap()`, which now returns 1 (the value of lock), and it continues to spin.

6.10 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

```
while (true) {
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j)
            ; /* do nothing */
         flag[i] = true;
      }
   }

      /* critical section */

   turn = j;
   flag[i] = false;

      /* remainder section */
}
```

Figure 6.17 The structure of process P_i in Dekker's algorithm.

The structure of process P_i (`i` == 0 or 1) is shown in Figure 6.18. The other process is P_j (`j` == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

Answer:

This algorithm satisfies the three conditions of mutual exclusion. (1) Mutual exclusion is ensured through the use of the `flag` and `turn` variables. If both processes set their `flag` to `true`, only one will succeed—namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of `turn`. (2) Progress is provided, again, through the `flag` and `turn` variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access its critical section, it can set its `flag` variable to `true` and enter the critical section. It sets `turn` to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting `turn` to the other process upon exiting. (3) Bounded waiting is preserved through the use of the TTturn variable. Assume that two processes wish to enter their critical sections. They both set their value of `flag` to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserved, it would be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Dekker's algorithm has a process to set the value of `turn` to the other process, thereby ensuring that the other process will enter its critical section next.

6.11 Consider how to implement a mutex lock using the `compare_and_swap()` instruction. Assume that the following structure defining the mutex lock is available:

```
typedef struct {
     int available;
} lock;
```

The value `(available == 0)` indicates that the lock is available, and a value of 1 indicates that the lock is unavailable. Using this `struct`, illustrate how the following functions can be implemented using the `compare_and_swap()` instruction:

- `void acquire(lock *mutex)`
- `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

Answer:
Please see Figure 6.101.

6.12 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism—a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

- The lock is to be held for a short duration.
- The lock is to be held for a long duration.
- A thread may be put to sleep while holding the lock.

Answer:

- Spinlock
- Mutex lock
- Mutex lock

6.13 A multithreaded web server wishes to keep track of the number of requests it services (known as ***hits***). Consider the two following strategies to prevent a race condition on the variable `hits`. The first strategy is to use a basic mutex lock when updating `hits`:

```
int hits;
mutex_lock hit_lock;

hit_lock.acquire();
hits++;
hit_lock.release();
```

```
// initialization
mutex->available = 0;

// acquire using compare_and_swap()
void acquire(lock *mutex) {
   while (compare_and_swap(&mutex->available, 0, 1) != 0)
      ;

   return;
}


// acquire using test_and_set()
void acquire(lock *mutex) {
   while (test_and_set(&mutex->available) != 0)
      ;

   return;
}


void release(lock *mutex) {
   mutex->available = 0;

   return;
}
```

Figure 6.101 Solution for Exercise 6.11.

A second strategy is to use an atomic integer:

```
atomic_t hits;
atomic_inc(&hits);
```

Explain which of these two strategies is more efficient.

Answer:
The use of locks is overkill in this situation. Locking generally requires a system call and may require putting a process to sleep (thus requiring a context switch) if the lock is unavailable. (Awakening the process will require another context switch.) On the other hand, the atomic integer provides an atomic update of the `hits` variable and ensures no race condition on `hits`. This can be accomplished with no kernel intervention. Therefore, the second approach is more efficient.

6.14 Servers can be designed to limit the number of open connections. For example, a server may wish to have only *N* socket connections at any point in time. As soon as *N* connections are made, the server will not accept another incoming connection until an existing connection is released. Illustrate how semaphores can be used by a server to limit the number of concurrent connections.

Answer:
A semaphore is initialized to the number of allowable open socket connections. When a connection is accepted, the `acquire()` method is called; when a connection is released, the `release()` method is called. If the system reaches the number of allowable socket connections, subsequent calls to `acquire()` will block until an existing connection is terminated and the release method is invoked.

6.15 Describe how the `signal()` operation associated with monitors differs from the corresponding operation defined for semaphores.

Answer:
The `signal()` operation associated with monitors is not persistent in the following sense: if a signal is performed and if there are no waiting threads, then the signal is simply ignored, and the system does not remember that the signal took place. If a subsequent wait operation is performed, then the corresponding thread simply blocks. In semaphores, every signal results in a corresponding increment of the semaphore value even if there are no waiting threads. A future wait operation would immediately succeed because of the earlier increment.