CHAPTER 6

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, discussed in Chapter 4. Concurrent access to shared data may result in data inconsistency, however. In this chapter, we discuss various mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.

Exercises

6.1 Race conditions are possible in many computer systems. Consider a banking system with two methods: `deposit(amount)` and `withdraw( amount)`. These two methods are passed the amount that is to be deposited or withdrawn from a bank account. Assume that a husband and wife share a bank account and that concurrently the husband calls the `withdraw()` method 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 the balance in the account is 250.00 and 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 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 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.

6.2 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;
```

The structure of process P_i (`i` == 0 or 1) is shown in Figure 6.21; 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 their critical section, it can set their `flag` variable to true and enter their 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 two processes wish to enter their respective 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 therefore 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 set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.

6.3 The first known correct software solution to the critical-section problem for *n* processes with a lower bound on waiting of *n* − 1 turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

All the elements of `flag` are initially `idle`; the initial value of `turn` is immaterial (between 0 and `n`-1). The structure of process P_i is shown in Figure 6.22. Prove that the algorithm satisfies all three requirements for the critical-section problem.

Answer:

This algorithm satisfies the three conditions. Before we show that the three conditions are satisfied, we give a brief explanation of what the algorithm does to ensure mutual exclusion. When a process *i* requires access to critical section, it first sets its `flag` variable to `want_in` to indicate its desire. It then performs the following steps: (1) It ensures that all processes whose index lies between `turn` and *i* are idle. (2) If so, it updates its `flag` to in_cs and checks whether there is already some other process that has updated its `flag` to in cs. (3) If not and if it is this process's turn to enter the critical section or if the process indicated by the `turn` variable is idle, it enters the critical section. Given the above description, we can reason about how the algorithm satisfies the requirements in the following manner:

a. Mutual exclusion is ensured:Notice that a process enters the critical section only if the following requirements is satisfied: no other process has its `flag` variable set to in_cs. Since the process sets its own `flag` variable set to in_cs before checking the status of other processes, we are guaranteed that no two processes will enter the critical section simultaneously.

b. Progress requirement is satisfied: Consider the situation where multiple processes simultaneously set their `flag` variables to in_cs and then check whether there is any other process has the `flag` variable set to in cs. When this happens, all processes realize that there are competing processes, enter the next iteration of the outer *while*(1) loop and reset their `flag` variables to want_in. Now the only process that will set its `turn` variable to in_cs is the process whose index is closest to `turn`. It is however possible that new processes whose index values are even closer to `turn` might decide to enter the critical section at this point and therefore might be able to simultaneously set its `flag` to in_cs. These processes would then realize there are competing processes and might restart the process of entering the critical section. However, at each iteration, the index values of processes that set their `flag` variables to in_cs become closer to `turn` and eventually we reach the following condition: only one process (say `k`) sets its `flag` to in_cs and no other process whose index lies between `turn` and *k* has set its `flag` to in_cs. This process then gets to enter the critical section.

c. Bounded-waiting requirement is met: The bounded waiting requirement is satisfied by the fact that when a process *k* desires to enter the critical section, its `flag` is no longer set to idle. Therefore, any process whose index does not lie between `turn` and *k* cannot enter the critical section. In the meantime, all processes whose index falls between `turn` and *k* and desire to enter the critical section would indeed enter the critical section (due to the fact that the system always makes progress) and the turn value monotonically becomes closer to *k*. Eventually, either `turn` becomes *k* or there are no processes whose index values lie between `turn` and *k*, and therefore process *k* gets to enter the critical section.

6.4 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

Answer:

If a user-level program is given the ability to disable interrupts, then it can disable the timer interrupt and prevent context switching from taking place, thereby allowing it to use the processor without letting other processes execute.

6.5 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

Answer:

Interrupts are not sufficient in multiprocessor systems since disabling interrupts only prevents other processes from executing on the processor in which interrupts were disabled; there are no limitations on what processes could be executing on other processors and therefore the process disabling interrupts cannot guarantee mutually exclusive access to program state.

6.6 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

Answer:

Because acquiring a semaphore may put the process to sleep while it is waiting for the semaphore to become available. Spinlocks are to only be held for short durations and a process that is sleeping may hold the spinlock for too long a period.

6.7 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

Answer:

There are many answers to this question. Some kernel data structures include a process id (pid) management system, kernel process table, and scheduling queues. With a pid management system, it is possible two processes may be created at the same time and there is a race condition assigning each process a unique pid. The same type of race condition can occur in the kernel process table: two processes are created at the same time and there is a race assigning them a location in the kernel process table. With scheduling queues, it is possible one process has been waiting for IO which is now available. Another process is being context-switched out. These two processes are being moved to the Runnable queue at the same time. Hence there is a race condition in the Runnable queue.

6.8 Describe how the `compare_and_swap()` instruction can be used to provide mutual exclusion that satisfies the bounded-waiting requirement.

Answer:

Please see Figure 6.1

6.9 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

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

```
do {waiting[i] = TRUE; key = TRUE;
      while (waiting[i] && key) key = Swap(&lock, &key);

      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;
```

```
      n/* remainder section */
      while (TRUE);
}
```

Figure 6.1 Program for Exercise 6.8.

where `(available == 0)` indicates the lock is available; a value of 1 indicates the lock is unavailable. Using this `struct`, illustrate how the following functions may be implemented using the `test_and_set()` and `compare_and_swap()` instructions.

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

Be sure to include any initialization that may be necessary.

Answer:

Please see Figure 6.2

6.10 The implementation of mutex locks provided in Section 6.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed into a waiting queue until the lock became available.

Answer:

This would be very similar to the changes made in the description of the semaphore. Associated with each mutex lock would be a queue of waiting processes. When a process determines the lock is unavailable, they are placed into the queue. When a process releases the lock, it removes and awakens the first process from the list of waiting processes.

6.11 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.
- The thread may be put to sleep while holding the lock.

```
// 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 5.2 Program for Exercise 6.9.

Answer:

- Spinlock
- Mutex lock
- Mutex lock

6.12 Assume a context switch takes *T* time. Ssuggest an upper bound (in terms of *T*) for holding a spin lock and that if the spin lock is held for any longer duration, a mutex lock (where waiting threads are put to sleep) is a better alternative.

Answer:

The spinlock should be held for < *2xT*. Any longer than this duration it would be faster to put the thread to sleep (requiring one context switch) and then subsequently awaken it (requiring a second context switch.)

6.13 A multithreaded web server wishes to keep track of the number of requests it services (known as **hits**.) Consider the following two 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();
```

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 possibly putting a process to sleep (and thus requiring a context switch) if the lock is unavailable. (Awakening the process will similarly require another subsequent 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 and therefore the second approach is more efficient.

6.14 Consider the code example for allocating and releasing processes shown in Figure 6.23.

a. Identify the race condition(s).
b. Assume you have a mutex lock named `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s).
c. Could we replace the integer variable
   `int number_of_processes = 0`
   with the atomic integer
   `atomic_t number_of_processes = 0`
   to prevent the race condition(s)?

Answer:

a. There is a race condition on the variable `number_of_processes`.

b. A call to `acquire()` must be placed upon entering each function and a call to `release()` immediately before exiting each function.
c. No, it would not help. The reason is because the race occurs in the `allocate_process()` function where `number_of_processes` is first tested in the if statement, yet is updated afterwards, based upon the value of the test. it is possible that `number_of_processes = 254` at the time of the test, yet because of the race condition, is set to 255 by another thread before it is incremented yet again.

6.15 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. Explain 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.

6.16 Windows Vista provides a new lightweight synchronization tool called a **slim reader–writer lock**. Whereas most implementations of reader–writer locks favor either readers or writers, or perhaps order waiting threads using a FIFO policy, slim reader–writer locks favor neither readers nor writers and do not order waiting threads in a FIFO queue. Explain the benefits of providing such a synchronization tool.
Answer:
Simplicity. IF RW locks provide fairness or favor readers or writers, there is more overhead to the lock. By providing such a simple synchronization mechanism, access to the lock is fast. Usage of this lock may be most appropriate for situations where reader–locks are needed, but quickly acquiring and releasing the lock is similarly important.

6.17 Show how to implement the `wait()` and `signal()` semaphore operations in multiprocessor environments using the `test_and_set()` instruction. The solution should exhibit minimal busy waiting.
Answer:
Here is the pseudocode for implementing the operations:
Please see Figure 6.3

6.18 Exercise 4.26 requires the parent thread to wait for the child thread to finish its execution before printing out the computed values. If we let the parent thread access the Fibonacci numbers as soon as they have been computed by the child thread—rather than waiting for the child thread to terminate—what changes would be necessary to the solution for this exercise? Implement your modified solution.
Answer:
A counting sempahore or condition variable works fine. The semaphore would be initialized to zero, and the parent would call the `wait()` function. When completed, the child would invoke `signal()`, thereby notifying the parent. If a condition variable is used, the parent thread will invoke `wait()` and the child will call `signal()` when completed. In both instances, the idea is that the parent thread waits for the child for notification that its data is available.

```
int guard = 0;
int semaphore_value = 0;

wait()
{
```

```
      while (TestAndSet(&guard) == 1);
      if (semaphore_value == 0) {
            atomically add process to a queue of processes
            waiting for the semaphore and set guard to 0;
      }else {
            semaphore_value--;
            guard = 0;
      }
}

signal()
{
      while (TestAndSet(&guard) == 1);
      if (semaphore_value == 0 &&
                  there is a process on the wait queue)
            wake up the first process in the queue
            of waiting processes
      else
            semaphore_value++;
      guard = 0;
}
```

Figure 6.3 Program for Exercise 6.17.

6.19 Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement the same types of synchronization problems.

Answer:

A semaphore can be implemented using the following monitor code:

Please see Figure 6.4

A monitor could be implemented using a semaphore in the following manner. Each condition variable is represented by a queue of threads waiting for the condition. Each thread has a semaphore associated with its queue entry. When a thread performs a wait operation, it creates a new semaphore (initialized to zero), appends the semaphore to the queue associated with the condition variable, and performs a blocking semaphore decrement operation on the newly created semaphore. When a thread performs a signal on a condition variable, the first process in the queue is awakened by performing an increment on the corresponding semaphore.

```
monitor semaphore {
      int value = 0;
      condition c;

      semaphore_increment() {
            value++;
            c.signal();
      }

      semaphore_decrement() {
            while (value == 0)
            c.wait();
            value--;
      }
}
```

Figure 6.4 Program for Exercise 6.19.

6.20 Design an algorithm for a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.

Answer:

Please see Figure 6.5

6.21 The strict mutual exclusion within a monitor makes the bounded-buffer monitor of Exercise 6.20 mainly suitable for small portions.

a. Explain why this is true.
b. Design a new scheme that is suitable for larger portions.

Answer:

The solution to the bounded buffer problem given above copies the produced value into the monitor's local buffer and copies it back from the monitor's local buffer to the consumer. These copy operations could be expensive if one were using large extents of memory for each buffer region. The increased cost of copy operation means that the monitor is held for a longer period of time while a process is in the produce or consume operation. This decreases the overall throughput of the system. This problem could be alleviated by storing pointers to buffer regions within the monitor instead of storing the buffer regions themselves. Consequently, one could modify the code given above to simply copy the pointer to the buffer region into and out of the monitor's state. This operation should be relatively inexpensive and therefore the period of time that the monitor is being held will be much shorter, thereby increasing the throughput of the monitor.

```
monitor bounded_buffer {
      int items[MAX_ITEMS];
      int numItems = 0;
      condition full, empty;

      void produce(int v) {
            while (numItems == MAX_ITEMS) full.wait();
            items[numItems++] = v;
            empty.signal();
      }

      int consume() {
            int retVal;
            while (numItems == 0) empty.wait();
            retVal = items[--numItems];
            full.signal();
            return retVal;
      }
}
```

Figure 6.5 Program for Exercise 6.20.

6.22 Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.

Answer:

Throughput in the readers-writers problem is increased by favoring multiple readers as opposed to allowing a single writer to exclusively access the shared values. On the other hand, favoring readers could result in starvation for writers. The starvation in the readers-writers problem could be avoided by keeping timestamps associated with waiting processes. When a writer is finished with its task, it would wake up the process that

has been waiting for the longest duration. When a reader arrives and notices that another reader is accessing the database, then it would enter the critical section only if there are no waiting writers. These restrictions would guarantee fairness.

6.23 How does the `signal()` operation associated with monitors differ 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, on the other hand, 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.

```
monitor printers {
      int num_avail = 3;
      int waiting_processes[MAX_PROCS];
      int num_waiting;
      condition c;

      void request_printer(int proc_number) {
            if (num_avail > 0) {
            num_avail--;
            return;
            }
            Waiting_processes[num_waiting] = proc_number;
            Num_waiting++;
            sort(waiting_processes);
            while (num_avail == 0 &&
                  waiting_processes[0] != proc_number)
            c.wait();
            waiting_processes[0] =
                  waiting_processes[num_waiting-1];
            num_waiting--;
            sort(waiting_processes);
            num_avail--;
      }

      void release_printer() {
            num_avail++;
            c.broadcast();
      }
}
```

Figure 6.6 Program for Exercise 6.25.

6.24 Suppose the `signal()` statement can appear only as the last statement in a monitor procedure. Suggest how the implementation described in Section 6.8 can be simplified in this situation.

Answer:

If the signal operation were the last statement, then the lock could be transferred from the signalling process to the process that is the recipient of the signal. Otherwise, the signalling process would have to explicitly

release the lock and the recipient of the signal would have to compete with all other processes to obtain the lock to make progress.

6.25 Consider a system consisting of processes P_1, P_2, ..., P_n, each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.

Answer:

The pseudocode is presented in Figure 6.6

6.26 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several

```
monitor file_access {
      int curr_sum = 0;
      int n;
      condition c;

      void access_file(int my_num) {
            while (curr_sum + my_num >= n)
            c.wait();
            curr_sum += my_num;
      }

      void finish_access(int my_num) {
            curr_sum -= my_num;
            c.broadcast();
      }
}
```

Figure 6.7 Program for Exercise 6.26.

processes, subject to the following constraint: The sum of all unique numbers associated with all the processes currently accessing the file must be less than *n*. Write a monitor to coordinate access to the file.

Answer:

The pseudocode is presented in Figure 6.7.

6.27 When a signal is performed on a condition inside a monitor, the signaling process can either continue its execution or transfer control to the process that is signaled. How would the solution to the preceding exercise differ with these two different ways of performing signaling?

Answer:

The solution to the previous exercise is correct under both situations. However, it could suffer from the problem that a process might be awakened only to find that it is still not possible for it to make forward progress either because there was not sufficient slack to begin with when a process was awakened or if an intervening process gets control, obtains the monitor and starts accessing the file. Also, note that the broadcast operation wakes up all of the waiting processes. If the signal also transfers control and the monitor from the current thread to the target, then one could check whether the target would indeed be able to make forward progress and perform the signal only if it were possible. Then the “while” loop for the waiting thread could be replaced by an “if” condition since it is guaranteed that the condition will be satisfied when the process is woken up.

6.28 Suppose we replace the `wait()` and `signal()` operations of monitors with a single construct `await(B)`, where `B` is a general Boolean expression that causes the process executing it to wait until `B` becomes `true`.

a. Write a monitor using this scheme to implement the readers–writers problem.
b. Explain why, in general, this construct cannot be implemented efficiently.
c. What restrictions need to be put on the await statement so that it can be implemented efficiently? (Hint: Restrict the generality of `B`; see Kessels [1977].)

Answer:

a. The readers–writers problem could be modified with the following more generate `await` statements:
A reader can perform "`await(active_writers == 0 && waiting_writers == 0)`" to check that there are no active writers and there are no waiting writers before it enters the critical section. The writer can perform a "`await(active_writers == 0 && active_readers == 0)`" check to ensure mutually exclusive access.
b. The system would have to check which one of the waiting threads have to be awakened by checking which one of their waiting conditions are satisfied after a signal. This requires considerable complexity and might require some interaction with the compiler to evaluate the conditions at different points in time. One could restrict the Boolean condition to be a disjunction of conjunctions with each component being a simple check (equality or inequality with respect to a static value) on a program variable. In that case, the Boolean condition could be communicated to the run-time system, which could perform the check every time it needs to determine which thread to be awakened.
c. Please see Kessels [1977].

6.29 Design an algorithm for a monitor that implements an *alarm clock* that enables a calling program to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function `tick()` in your monitor at regular intervals.

Answer:

A pseudocode for implementing this is presented in Figure 6.8

```
monitor alarm {
      condition c;

      void delay(int ticks) {
            int begin_time = read_clock();
            while (read_clock() < begin_time + ticks)
            c.wait();
      }

      void tick() {
            c.broadcast();
      }
}
```

Figure 6.8 Program for Exercise 6.29.