CHAPTER 6

Synchronization

Chapter 6 is concerned with the topic of process synchronization among concurrently executing processes. Concurrency is generally very hard for students to deal with correctly, and so we have tried to introduce it and its problems with the classic process coordination problems: mutual exclusion, bounded-buffer, readers/writers, and so on. An understanding of these problems and their solutions is part of current operating-system theory and development.

We first use semaphores and monitors to introduce synchronization techniques. Next, Java synchronization is introduced to further demonstrate a language-based synchronization technique. We conclude with a discussion of how contemporary operating systems provide features for process synchronization and thread safety.

6.1 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.

6.2 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.

6.3 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 Here is a pseudocode for implementing this:

```
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();
    }
}
```

6.5 The pseudocode is as follows:

```
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();
    }
}
```

6.6

```
monitor resources {
    int available_resources;
    condition resources_avail;

    int decrease_count(int count) {
       while (available_resources < count)
       resources_avail.wait();
       available_resources -= count; }

    int increase_count(int count) {
       available_resources += count;
       resources_avail.signal();
    }
```

6.7 A counting sempahore or condition variable works fine. The sempahore 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.

6.8 The system clock is updated at every clock interrupt. If interrupts were disabled—particularly for a long period of time—it is possible 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 quantums. This effect can be minimized by disabling clock interrupts for only very short periods.

6.9 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.

6.10 Solaris, Linux, and Windows 2000 use spinlocks as a synchronization mechanism only on multiprocessor systems. Spinlocks are not appropriate for single-processor systems because the condition that would break a process out of the spinlock could 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 thereby modify the program state in order to release the first process from the spinlock.

6.11 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, if the two operations are not performed atomically, then it is possible that both operations might proceed to decrement the semaphore value, thereby violating mutual exclusion.

6.12 Here is the pseudocode for implementing the operations:

```
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;
}
```

6.13 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.

6.14 A schedule refers to the execution sequence of the operations for one or more transactions. A serial schedule is the situation where each transaction of a schedule is performed atomically. If a schedule consists of two different transactions where consecutive operations from the different transactions access the same data and at least one of the operations is a write, then we have what is known as a *conflict*. If a schedule can be transformed into a serial schedule by a series of swaps on nonconflicting operations, we say that such a schedule is conflict serializable.

The two-phase locking protocol ensures conflict serializabilty because exclusive locks (which are used for write operations) must be acquired serially, without releasing any locks during the acquire (growing) phase. Other transactions that wish to acquire the same locks must wait for the first transaction to begin releasing locks. By requiring that all locks must first be acquired before releasing any locks, we are ensuring that potential conflicts are avoided.

6.15 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.

6.16 Volatile storage refers to main and cache memory and is very fast. However, volatile storage cannot survive system crashes or powering down the system. Nonvolatile storage survives system crashes and powered-down systems. Disks and tapes are examples of nonvolatile storage. Recently, USB devices using erasable program read-only memory (EPROM) have appeared providing nonvolatile storage. Stable storage refers to storage that technically can *never* be lost as there are redundant backup copies of the data (usually on disk).

6.17 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.

6.18 Here is the pseudocode:

```
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();
    }
}
```

6.19 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.

6.20

a. Identify the data involved in the race condition: The variable available_resources.

b. Identify the location (or locations) in the code where the race condition occurs: The code that decrements available_resources and the code that increments available_resources are the statements that could be involved in race conditions.

c. Using a semaphore, fix the race condition: Use a semaphore to represent the available_resources variable and replace increment and decrement operations by semaphore increment and semaphore decrement operations.

6.21 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 thereby modify the program state in order to release the first process from the spinlock.

6.22 Please refer to the supporting Web site for source code solution.

6.23

```
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);
}
```

6.24 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.

6.25 If the transactions that were issued after the rolled-back transaction had accessed variables that were updated by the rolled-back transaction, then these transactions would have to be rolled back as well. If they have not performed such operations (that is, there is no overlap with the rolled-back transaction in terms of the variables accessed), then these operations are free to commit when appropriate.

6.26 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.

6.27 The solution to 6.5 (Note: The text refers to "preceding exercise," it should refer to 6.5 instead.) 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 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 *Busy waiting* means that a process is waiting for a condition to be satisfied in a tight loop without relinquishing the processor. Alternatively, a process could wait by relinquishing the processor, and block on a condition and wait to be awakened at some appropriate time in the future. Busy waiting can be avoided but incurs the overhead associated with putting a process to sleep and having to wake it up when the appropriate program state is reached.

6.29 A semaphore can be implemented using the following monitor code:

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

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

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

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.

6.30 If the transaction needs to be aborted, then the values of the updated data values need to be rolled back to the old values. This requires the old values of the data entries to be logged before the updates are performed.

6.31 A checkpoint log record indicates that a log record and its modified data has been written to stable storage and that the transaction need not to be redone in case of a system crash. Obviously, the more often checkpoints are performed, the less likely it is that redundant updates will have to be performed during the recovery process.

a. System performance when no failure occurs—If no failures occur, the system must incur the cost of performing checkpoints that are essentially unnecessary. In this situation, performing checkpoints less often will lead to better system performance.

b. The time it takes to recover from a system crash—The existence of a checkpoint record means that an operation will not have to be redone during system recovery. In this situation, the more often checkpoints were performed, the faster the recovery time is from a system crash.

c. The time it takes to recover from a disk crash—The existence of a checkpoint record means that an operation will not have to be redone during system recovery. In this situation, the more often checkpoints were performed, the faster the recovery time is from a disk crash.

6.32

```
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;
    }
}
```

6.33 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.

6.34 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.

6.35 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.