Synchronization Examples

Practice Exercises

7.1 Explain why Windows and Linux implement multiple locking mechanisms. Describe the circumstances under which they use spinlocks, mutex locks, semaphores, and condition variables. In each case, explain why the mechanism is needed.

Answer:
These operating systems provide different locking mechanisms depending on the application developers' needs. Spinlocks are useful for multiprocessor systems where a thread can run in a busy loop (for a short period of time) rather than incurring the overhead of being put in a sleep queue. Mutexes are useful for locking resources. Solaris 2 uses adaptive mutexes, meaning that the mutex is implemented with a spinlock on multiprocessor machines. Semaphores and condition variables are more appropriate tools for synchronization when a resource must be held for a long period of time, since spinning is inefficient for a long duration.

7.2 Windows provides a lightweight synchronization tool called **slim reader–writer** locks. 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, nor are waiting threads ordered in a FIFO queue. Explain the benefits of providing such a synchronization tool.

Answer:
Simplicity. If reader–writer locks provide fairness or favor readers or writers, they involve more overhead. Providing such a simple synchronization mechanism makes access to the lock fast. Use of this lock may be most appropriate for situations where reader–writer locks are needed, but quickly acquiring and releasing them is similarly important.

7.3 Describe what changes would be necessary to the producer and consumer processes in Figure 7.1 and Figure 7.2 so that a mutex lock could be used instead of a binary semaphore.

Answer:
The calls to `wait(mutex)` and `signal(mutex)` need to be replaced so that they are now calls to the API for a mutex lock, such as `acquire(mutex)` and `release()` mutex.

7.4 Describe how deadlock is possible with the dining-philosophers problem.

Answer:
If all philosophers simultaneously pick up their left forks, when they turn to pick up their right forks they will realize they are unavailable, and will block while waiting for it to become available. This blocking while waiting for a resource to become available is a deadlocked situation.

7.5 Explain the difference between signaled and non-signaled states with Windows dispatcher objects.

Answer:
An object that is in the signaled state is available, and a thread will not block when it tries to acquire it. When the lock is acquired, it is in the non-signaled state. When the lock is released, it transitions back to the signaled state.

7.6 Assume `val` is an atomic integer in a Linux system. What is the value of `val` after the following operations have been completed?

```
atomic_set(&val,10);
atomic_sub(8,&val);
atomic_inc(&val);
atomic_inc(&val);
atomic_add(6,&val);
atomic_sub(3,&val);
```

Answer:
The final value of `val` is 10 - 8 + 1 + 1 + 6 - 3 = 7

Exercises

7.7 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 (PID) management system, kernel process table, and scheduling queues. With a PID management system, two processes may be created at the same time and there is a race condition in assigning each process a unique PID. The same type of race condition can occur in the kernel process table: two processes may be created at the same time, and there is a race assigning them a location in the kernel process table. With scheduling queues, one process may have been waiting for I/O that is now available. At the same time, 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.

7.8 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 be held only for short durations, and a process that is sleeping may hold the spinlock for too long.

7.9 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 were no waiting writers. These restrictions would guarantee fairness.

7.10 Explain the difference between software and hardware transactional memory.

Answer:
Software transactional memory is provided entirely in software by having the compiler inject the necessary instrumentation into transaction blocks within the code, with no support from the hardware. Alternatively, hardware transactional memory does not require software support, and instead uses cache hierarchies and cache coherency protocols to provide transactional memory.