CO327: Operating Systems
Assignment 4
E/15/271

1. **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 These two functions are passed the amount that is to be deposited or withdrawn from the bank account balance.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.**

   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.

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

   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 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 spin lock 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.

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

   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.

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

   Binary Semaphore can take only two values: 0 and 1.
   Therefore, it can be used to check the status of the lock, like if 0 would mean lock is unavailable and 1 would mean lock is available. Hence before entering the critical section it
   can check the availability and provide mutual exclusion.
   n processes share a semaphore – mutex which is initialized to 1.
   Each process can be organized as shown in the following;

```
do
{
wait(mutex);
/* critical section */
signal(mutex);
/* remainder section */
}
while (true);
```

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

   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. **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.**

   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 monopolize the processor so the other process may never execute so there will be starvation.

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

   Depends on how interrupts are implemented, but regardless of how, it is a poor poor choice of techniques.

   Case 1 -- interrupts are disabled for ONE processor only -- result is that threads running on other processors could ignore the synchronization primitive and access the shared data.

   Case 2 -- interrupts are disabled for ALL processors -- this means task dispatching, handling I/O completion, etc. is also disabled on ALL processors, so threads running on all CPUs can grind to a halt.

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.**

   Linux has an approach that a procedure can't hold a spinlock while endeavoring to gain a semaphore.
   Clarify why this arrangement is set up. You can't hold a turn lock while you procure a semaphore, since you may need to rest while hanging tight for the semaphore, and you can't rest while holding a turn lock. It will have a powerful effect.

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.**

   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.

10. **How does the signal() operation associated with monitors differ from the corresponding operation defined for semaphores?**

    High-level synchronization and interprocess communication such as monitors incorporate condition variables. A condition variable is a data structure that can only appear within a monitor. Those variables are global for all procedures within the monitor and may have its value manipulated by three operations which you might know: wait( ), signal( ), and queue( ).
    Since you are asking specifically about signal( ), in Monitors this operation resumes exactly one other process if any process is currently suspended due to a wait( ) operation on the condition variable. Ofcourse, if no thread is waiting, then the signal is not saved (and will have no effect)

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.
    • A thread may be put to sleep while holding the lock

    The type of mutex lock is also called a spinlock because the process “spins” while waiting for the lock to become available. 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. In certain circumstances on multicore systems, spinlocks are in fact the preferable choice for locking. If a lock is to be held for a **short duration**, one thread can “spin” on one processing core while another thread performs its critical section on another core. On modern multicore computing systems, spinlocks are widely used in many operating systems.