Synchronization Tools

CHAPTER 6

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.