# Section 5: Concurrency & Synchronization

## Question 1

1. What is the definition of a Race Condition?
2. From the bank account example, describe the race condition that exists when two threads simultaneously post debits on the same bank account.
3. Describe the race condition that exists when two threads simultaneously post debits on separate bank accounts.
4. What solution was presented that ensures bank accounts are debited correctly?

void postDebit(double debit) {

double balance = readBalance(accountID);

balance = balance – debit;

writeBalance(balance, accountID);

}

### Answer

1. A race condition occurs when multiple threads (or processes) read and write data items so that the final result depends on the order of execution of instructions in multiple threads. Because the order of execution of two threads is non-deterministic, the results of the processing after all threads have completed is also unknown.
2. The postDebit method will produce incorrect balances if the execution of two threads interleaves. If the first thread to start execution times out after reading the balance, and the second thread completes the operation, thread one will calculate and write an incorrect balance.
3. Trick Question! If each thread operates on separate accounts, there is no race-condition and no chance of corrupting the account balances.
4. A critical section is needed around the read / calculate / write statements.

## Question 2

Note: You will need to research this question.

1. What are the two methods in Java for creating and launching a Thread?
2. Which of these two methods is considered best i.e. a best practice?

### Answer

1. The class which is to be executed in a thread implements the interface java.lang.Runnable. This interface defines the method void run() which is the new thread’s entry point i.e. where the thread’s execution begins. The Thread is created by passing an instance of the ‘runnable’ class into java.lang.Thread’s constructor. The new thread is created by calling Thread.start(). See the class CriticalSectionBroken in the code samples for an example.
2. The second method is to subclass Thread and override the run() method. The thread is started by executing the start() method.

The Runnable approach is considered the best practice.

## Question 3

1. What does a counting semaphore, initialized to one, with a value of zero signify?
2. What is the value of a counting semaphore that has two blocked processes?

### Answer

1. That a single process owns the semaphore and there are zero processes waiting for ownership.
2. -2

## Question 4

Note: You will need to research this question.

*See the Java code in the zip archive “Sample Synchronized Code” in the supplemental materials folder in elearning.*

1. Why does the output of the sample code CriticalSectionBroken.java fail to print correctly?
2. Generally, how do we correct this problem with a critical section?

### Answer

1. The Runnable class StreamPrinter writes its message to the output stream (System.out) one character at a time. When there are multiple StreamPrinter threads executing, output of each instance is interleaved with the outputs of other instances.
2. The critical section in this code is the for-loop in the run() method that prints the message string. Each thread / runnable instance needs to execute their loop in a mutually exclusive manner (one thread at a time) to avoid interleaving with the print loops in other threads. There are several examples of how the mutual exclusive execution can be accomplished in Java. See CriticalSectionSynchronizedBlock, CriticalSectionSemaphore, & CriticalSectionMonitor for three examples that the different approaches to critical sections we studied in this section.

## Question 5

Note: You will need to research this question.

*See the Java code in the zip archive “Sample Synchronized Code” in the supplemental materials folder in elearning.*

Why does the Java program CriticalSectionFailedSynchronize.java fail to print messages correctly?

### Answer

This approach fails because the critical section created by the synchronized printMessage() method is locking on the separate instances of StreamPrinter threads. There is no mutual exclusive access to the for-loop because each thread is locking on itself and not locking a single shared (static) object that will force the mutually exclusive access to the critical section. See CriticalSectionSynchronizedBlock for a working example and notice that the synchronized block locks a static object ‘lock’.

## Question 6

In the implementation of finite bounded buffer shown in Figure 5.16…

1. What line blocks a producer thread when the buffer is full?
2. What line release a blocked producer thread when space becomes available?
3. What line blocks a consumer thread when the buffer is empty?
4. What line release a blocked consumer thread when data becomes available?

### Answer

1. If(count == N) cwait(notfull);
2. csignal(notfull);
3. If(count == 0) cwait(notempty);
4. csighal(notempty);

## Question 7

What three advantages does semaphores offer over compare and swap instructions?

### Answer

1. Semaphores cause the blocking of waiting processes / threads eliminating the busy-wait loop utilized by C&S instructions.
2. Semaphores offer the fair scheduling of blocked processes using FIFO queue of processes / treads waiting on the semaphore i.e. no starvation of waiting processes / threads.
3. With C&S, if the process in the critical section dies before setting the shared variable to zero, the waiting processes will never unblock. Because a Semaphore is managed by the OS, if the owning process dies while in a CS, the OS will release the terminated process’s ownership on any semaphores allowing waiting blocked processes to continue.

## Question 8

It appears that the book contains an error / misprint…

What is the omission / problem with the pseudo code implementation of readers / writers shown in Figure 5.22.

### Answer

In the function reader(), the second *readcount* should decrement its value i.e. readcount--.

## Question 9

What is the difference between the semaphore’s sema\_wait() operation and a monitor’s wait() operation?

### Answer

A semaphore’s sema\_wait() operation attempts to acquire the semaphore and blocks the calling thread only if the semaphore is already owned by a different thread / process. A thread that calls wait() on an ‘unowned’ semaphore will not block.

The monitor’s wait() operation always blocks the calling thread until a different thread in the system calls notify() or notifyAll().

## Question 10

In the slide “Using a Semaphore to Implement Mutual Exclusion”, why is it critical that the argument “semaphore” be declared static?

### Answer

Synchronization between multiple threads (processes) requires that each thread operate on the same instance of a shared synchronization variable (semaphore). This sharing allows changes made to the variable by a single thread in the system visible to all the threads in the system. If each tread operates on their own copy of the semaphore, there will be no communication between threads i.e. a change made to the semaphore in thread i would not be shared with the other threads in the system because each thread would have their own semaphore.