Problem Set 5

(6 Points) Cohort Exercise 1:

Fix FirstBlood.java using AtomicInteger, assuming the post-condition is that the sum is the number of additions.

Solution: FirstFixWithAtomicInteger.java

(6 Points) Cohort Exercise 2:

Assuming that the correctness requirement is that "saving + cash = 5000" is an invariant, fix the following class: LockStaticVariables.java. Hint: there are different ways.

Solution: LockStaticVariablesFixed.java

(6 Points) Cohort Exercise 4:

CachedFactorizer.java provides a service to factorize integers. For efficiency, it stores the last input and its factors so that in case the new input is the same as the last (a.k.a. a hit), the saved factors are returned. It also counts the number of hits and maintains a hit ratio. Fix the class so that it becomes thread-safe. Narrow the scope of the synchronized block, without compromising thread-safety.

Solution: CachedFactorizerThreadSafe.java

(6 Points) Cohort Exercise 5:

Fixed Buffer.java so that it is thread-safe and efficient

Solution: BufferFixed.java

(14 Points) Homework Question 1:

In this exercise we will attempt to synchronize a block of code. Within that block of code we will get the lock on an object, so that other threads cannot modify it while the block of code is executing. We will create three threads that will all attempt to manipulate the same object. Each thread will output a single letter 100 times, and then increment that letter by one (e.g. from A to B). The object we will be using is StringBuffer (i.e. a thread-safe mutable sequence of characters) object. The final output should have 100 As, 100 Bs, and 100 Cs all in unbroken lines.

1. Create a class and extend the Thread class.
2. Override the run() method of Thread. This is where the synchronized block of code will go.
3. For our three thread objects to share the same object, we will need to create a constructor that accepts a StringBuffer object in the argument.
4. The synchronized block of code will obtain a lock on the StringBuffer object from step 3.

5. Within the block, output the StringBuffer 100 times and then increment the letter in the StringBuffer.
6. Finally, in the main() method, create a single StringBuffer object using the letter A, then create three instances of our class and start all three of them.

Discuss why we need to use StringBuffer instead of String in this case.

(14 Points) Homework Question 2:

Implement a simple counter in Java which increments its value each second. Use sleep(1000) to delay each print. The counter stops whenever the user inserts the value 0 from console. Note that the program cannot make any assumption on when the user will press the key.

Solution: SleepCounter.java

(14 Points) Homework Question 3:

If you call wait() and notify() on an object, it must be inside of a block synchronized on the same object. What happens if not? Why is it necessary to own the lock on an object before calling wait() and notify() on it?

(14 Points) Homework Question 4:

As a rising star in a bank's IT department, you have been given the job of creating a new bank account class called SynchronizedAccount. This class must have methods to support the following operations: deposite, withdraw, and check balance. Each method should print a status message to the screen on completion. Also, the method for withdraw should return false and do nothing if there are insufficient funds. Because the latest system is multi-threaded, these methods must be designed so that the bookkeeping is consistent even if many threads are accessing a single account. Additionally, to maximize concurrency, SynchronizedAccount should be synchronized differently for read and write accesses. Any number of threads should simultaneously be able to check the balance on an account, but only one thread can deposit or withdraw at a time.

Solution: SynchronizedAccount.java

(14 Points) Homework Question 5:

Write a simulation program for the fruit market. The farmer will be able to produce different types of fruits (apple, orange, grape, and watermelon), and put them in the market to sell. The market has limited capacity and farmers have to stand in a queue if the capacity is exceeded to sell their fruits. Consumers can come to the market any time and purchase their desired fruits; and if the fruits they want to buy runs out, they are willing to wait until the supply of that kind is

ready. (Hint: implementing this market will encounter the producer and consumer problem, and it needs multiple buffers for different kinds of fruits).

Solution: FrurtMarket.java