Sample Solution for Homework 8

Problem 1 AMP, p. 449: Exercise 216 (6 Points)

1. To see why it is necessary to check whether the object is locked consider the following two transactions.

```
A: atomic {                    B: atomic {
  x = x + y;                     y = y + x;
}                              }
```

   Suppose that initially, both `x` and `y` are 1, the global version clock is 0, and the stamps of both `x` and `y` are 0. Now `A` and `B` execute concurrently, each initializing their read stamps to the current value of the global version clock, which is 0. Before the two transactions commit, `A` has `x` in its write set and `y` in its read set. The value of the virtual copy of `x` in `A` is 2. For `B`, the situation is similar with the roles of `x` and `y` interchanged.

   Now both transactions start their commit concurrently. First, both `A` and `B` atomically increment the global version clock and store the new value of the clock in their own write stamps. Suppose that `A`'s write stamp is 1 and `B`'s write stamp is 2. Next, both transactions lock the objects in their write sets, i.e., both `x` and `y` are now locked.

   If the transactions would now check whether the objects in their read sets are locked, they would both abort. Instead, each transaction continues to check that the stamp of each object in their read sets are not greater than the transaction's read stamp. Since all stamps are 0, these checks succeed. Next, both transactions commit, changing the values of `x` and `y` to 2, and updating their stamps to `1` and `2`, respectively.

   Note that any serialized execution of the transactions `A` and `B` yields either `x==2` and `y===3`, or `x==3` and `y==2`. Hence, the above example shows that this implementation of the transactions commit handler would not guarantee serializability of committed transactions.

2. Yes. If an object is first read and then written with a transaction, it is both in the read set and the write set of that transaction. Therefore, the additional check on line 70 of Fig. 18.31 is needed.

3. Consider the same example as above. We start from the same initial state with both transaction executing concurrently until their commit handlers are called. Now, suppose that `A` starts executing its commit handler first, locking `x` and setting its write stamp to 1. Next, it checks whether `y`'s stamp is consistent with `A`'s read stamp and whether `y` is unlocked. Both checks succeed. Now, suppose that `B` starts its commit handler, setting its write stamp to 2, locking `y` and checking whether `x`'s stamp is consistent with `B`'s read stamp. This check succeeds. Next, `A` completes its commit, which updates `x` to 2 and unlocks `x`. Next, `B` continues, sees that `x` is unlocked and completes its own commit, setting `y` to 2. Again, the resulting state cannot be obtained by any serial execution of `A` and `B`.

Problem 2 AMP, p. 449: Exercise 218 (6 Points)

```
class AtomicArray[T](length: Int) {

  val array = new Array[Ref[T]](length)

  def compareAndSet(i: Int, expect: T, update: T): Boolean =
    // No need to check index bounds as Scala's Arrays are already checked.
    array(i).single.compareAndSet(expect, update)

  def get(i: Int): T =
    array(i).single()

  def set(i: Int, newValue: T): Unit =
    array(i).single() = newValue
}
```

Problem 3 Transactional Stack

See `EliminationStack.scala` for the source code of the solution to this exercise.