6.189 Multicore Programming Primer **Mini-Quiz #2 (01/12/2007) Solutions**

1. This implementation is vulnerable to deadlock. For example, suppose we process two votes concurrently: "Candidate 1 is better than candidate 2" and "Candidate 2 is better than candidate 1". Here is one interleaving which leads to deadlock:

```
movePoints(c1, c2)                  movePoints(c2, c1)
   synchronized(c1) {
                                       synchronized(c2) {
      synchronized(c2) {
                                          synchronized(c1) {
```

2. This (correct) implementation solves the deadlock problem of (1) by acquiring the locks in a canonical order. Also, it only locks on the two candidates associated with the current vote.

3. This implementation is also correct in that it does not deadlock. However, since it locks on the (one and only) Election object, only one thread can be in the critical section at any time. So this implementation will be slower than (2) since it can't take advantage of the concurrency allowed by the problem.

4. This implementation has a race condition because the first lock is released before the second is acquired. To see that this is a problem, suppose that both candidates 1 and 2 initially have 991 points, and we need to process two votes: "Candidate 1 is better than candidate 2" and "Candidate 2 is better than candidate 1". Under either of the two possible ways in which the votes can be processed atomically, one candidate ends up with 990 points while the other ends up with 992 points. Below is one interleaving which gives both candidates 991 points. So this implementation is incorrect because the votes are not processed atomically.

```
movePoints(c1, c2)                  movePoints(c2, c1)
synchronized(c2) {
   tmp = c2.getPoints/100;
      c2.AddPoints(-tmp);
}
                                    synchronized(c1) {
                                       tmp = c1.getPoints/100;
                                          c1.AddPoints(-tmp);
                                    }

               /* Now c1 and c2 both have 982 points */

synchronized(c1) {
   c1.AddPoints(tmp);
}                                   synchronized(c2) {
                                       c2.AddPoints(tmp);
                                    }

               /* Now c1 and c2 both have 991 points */
```

5. This implementation also has a race condition; we can see that there is an unprotected region between when the candidate's point value is read and when it is used. Suppose candidates 1 and 2 initially have 1000 points, and we need to process two votes for "Candidate 1 is better than candidate 2". If the vote processing were atomic, then candidate 1 would end up with 1019 points and candidate 2 would end up with 981 points. Below is one interleaving which gives a different result. So this implementation is incorrect because the votes are not processed atomically.

```
movePoints(c1, c2)                 movePoints(c1, c2)
synchronized(c2) {
    tmp = c2.getPoints/100;
}
                                   synchronized(c2) {
                                      tmp = c2.getPoints/100;
                                   }

                 /* tmp == 10 in both threads */

synchronized(c2) {
    c2.AddPoints(-tmp);
}
                                   synchronized(c2) {
                                      c2.AddPoints(-tmp);
                                   }

         /* Now c1 has 1000 points, c2 has 980 points */

synchronized(c1) {
    c1.AddPoints(tmp);
}
                                   synchronized(c1) {
                                      c1.AddPoints(tmp);
                                   }

         /* Now c1 has 1020 points, c2 has 980 points */
```