Zack Campbell - zcc254
Kazuhiro Aoyama - ka25635

Homework #1

3. TACC Username: zcampbel (Zack), ka25635 (Kazuhiro)

4. A. If each thread sets the turn variable to itself instead of the other person, the threads can deadlock. Suppose the first thread successfully set turn to itself. Then that thread will enter critical section because it isn't waiting for the other thread. At the same time, suppose a second thread set turn to itself. Then the second thread can also enter the critical section because it doesn't have to wait for the other thread to go. Therefore, both threads are in critical section simultaneously which violates the principle of mutual exclusion.

B. If the turn variable was set before the wantCS variable, the algorithm would also be incorrect. For example, if thread A sets the turn variable to thread B and then thread B sets the turn variable to thread A before thread A can execute any more instructions, both threads think it's the other thread's turn. Then if thread B completes the wantCS instruction and thread A completes the wantCS instruction after that, then thread A will be stuck in the while loop thinking it is thread B's turn while thread B will be stuck in the while loop thinking it is thread A's turn. This results in deadlock between the two threads.

5. Suppose thread A enters the requestCS block at the same time as thread B enters their requestCS block. Then suppose thread A and thread B simultaneously set their wantCS variables to true. One of the threads successfully sets the turn to the opposite thread when the instruction is called simultaneously (suppose thread A was successful in this endeavor). As a result, thread A is stuck spinning on the while loop because thread B wants the critical section and the turn was set to thread B. Thread B then enters the critical section because thread A wants the critical section but it is not thread A's turn. Thread B then is the only thread in the critical section, completes its work while the other thread waits, and then sets it's wantCS variable to false. This exit condition releases thread A from its while loop and allows thread A to enter the critical section. Suppose thread B attempts to request the critical section again while thread A is in the critical section. Then thread B will set it's wantCS variable to true and attempt to set the turn to thread A. It will succeed in setting the turn to thread A because thread A is not attempting to set it to thread B. Thread B will then be stuck in the while loop while thread A remains in the critical section because thread A's wantCS variable is still true and it successfully set the turn to thread A. Once thread A finishes in the critical section and completes the exit protocol of setting it's wantCS variable to false, thread B will again be able to access the critical section. Because each thread is only spinning on a single thread and the protocol for exiting the critical section is setting the wantCS variable to false, the threads are always able to get through the critical section once the other thread is finished, indicating that Peterson's algorithm is starvation free.

6. class PetersonAlgorithm implements Lock {

```
  boolean wantCS[] = {false, false};
  Int turn0;
  int turn1;
  public void requestCS(int i) {
     wantCS[i] = true;
     If (i == 0) {
        Turn0 = 1;
        Int j = 1 - turn1;
        while (wantCS[j] && (turn0 == j));
     } Else {
        Turn1 = 0;
        Int j = 1 - turn0;
        While (wantCS[j] && (turn1 == j));
     }
  }
  public void releaseCS(int i) {
     wantCS[i] = false;
  }
}
```