**Part 1**

* 1. NO. The process’ frag is still true and others are waiting in the while loop.
  2. YES. Because the process in CS will never unlock, others can’t enter CS and will starve.
  3. YES. Starvation will violate bounded waiting properties.

1. Figure 2
   1. Mutual exclusion : NO. Both p0 and p1 can enter CS. Initially flag[0] and flag[1] are both false. Each sees the other’s flag is false, change their flags to true and enter CS simultaneously.
   2. Deadlock freedom : YES. If algorithm doesn’t provide mutual exclusion, deadlock cannot happen.
   3. Starvation freedom : NO. p0 can re-enter CS before p1 checks flag[0]’s value in while statement. Then, p1 never enter CS.

Figure 3

1. Mutual exclusion : YES. Because of shared variable “turn”, only one can enter CS.
2. Deadlock freedom :
3. Starvation freedom :
4. The solution satisfies neither safety (deadlock free) nor liveness (starvation free).

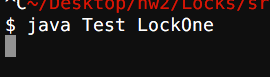
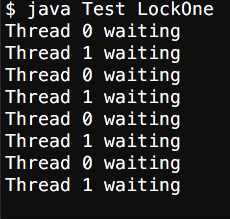
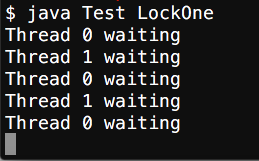
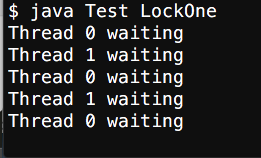
In line #1, If both “a” and “b” are 1, it causes deadlock.

Let’s assume that a = 1 and first thread is in the critical section. When the thread exists from the critical section, then a = 0. Other thread (second thread) looking for entering the critical section (line #7, b = 1), sees a is 0, then passes while loop (line #8), and b is now 0. It will check “a” (line #10). However, if the first thread enters the critical section between line #9 and #10, the second thread finds a = 1 and should wait. It causes starvation of the second thread.

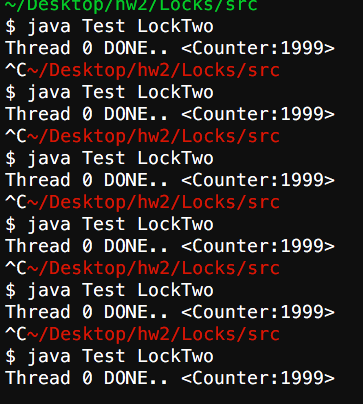
* 1. Even though two threads start simultaneously, in line #7, one of the threads should wait for x to be 0. After one thread exits CS, then x will be 0 and, the thread that is waiting in line #9 will enter CS.  
     Because of line #7, it satisfies mutual exclusion.
  2. NO. It does not satisfy starvation-freedom. A thread is waiting for y to be 0 in line #3. Finally, the thread confirms y is 0 and goes to start. Before the thread check whether y is 0 or not (line #2), other thread can change y to 1 (line #6). Then, the thread will wait for y to be 0 again. This cycles can cause starvation.
  3. YES. Let’s assume that three threads want to enter CS at the same time. Last thread that changed x to its thread-id will go through the if-statement and finally enter CS even though other two threads are waiting. The thread will exit CS and set x, y to 0. Then other threads can proceed. This is two thread case.
  4. YES. Only one thread can enter CS because of line #7. If x is not thread-id, the thread should wait.

1. Let’s assume that token A is on Node 12(Node 2 in subgraph 1) and tokens B and C on Nodes 21 and 22 (Nodes 1 and 2 in subgraph 2) respectively. Token A wants to dominate the others, so it will scan and decides to move to Node 00 (Node 0 in subgraph 0). Before token A moves to Node00, token B can move to Node 11 to dominates the others. Token A have to moves to Node 00 after token B moved to Node 11 even though it breaks total order because token A already have decided to move to Node 00. In this example, the sequential time-stamp system T^3 does not work for three threads.

**Part 2**

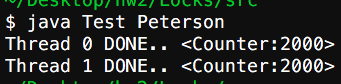
1. NULL
2. LockOne:  
   
   1. mutual-exclusion : YES
   2. fairness : NO
   3. deadlock-freedom : NO
   4. starvation-freedom : NO

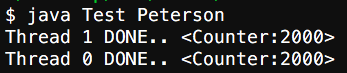
LockTwo :



1. mutual-exclusion : YES
2. fairness : NO
3. deadlock-freedom : YES
4. starvation-freedom : NO

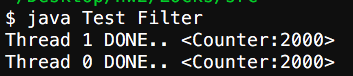
Peterson :

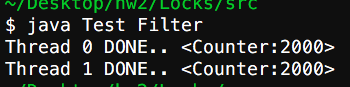




1. mutual-exclusion : YES
2. fairness : YES
3. deadlock-freedom : YES
4. starvation-freedom : YES

Filter :





1. mutual-exclusion : YES
2. fairness : NO
3. deadlock-freedom : YES
4. starvation-freedom : YES