3. a) If the turn variable is set to itself, then the processes are no longer guaranteed mutual exclusion and fairness. Consider Process 0 waiting at *while(wantCS[1] && turn == 1).* Because Process 1 sets turn to 1 each time it calls requestCS, the wait for Process 0 will always remain true, thus never allowing Process 0 to reach CS, making the algorithm unfair. Another case is when Process 0 makes the request to the CS and successfully enters. While Process 0 is in the CS, Process 1 makes the request to the CS. When Process 1 sets the turn variable to 1 and reaches *while(wantCS[0] && turn == 0)* while Process 0 is still in the CS, Process 1 will see that turn != 0 and decide to enter the CS. Thus, the modification does not uphold mutual exclusion either.

b) If the turn variable is set before *wantCS* variable, then mutual exclusion is not guaranteed. When the *wantCS* variable is set after the turn variable, there is a possibility of the other process reading the *wantCS* variable before it is properly set. For example, consider Process 1 calling *requestCS* and setting turn to 0 while Process 0 also makes a call to *requestCS* a little bit later, causing turn to be set back to 1, but gets to *while(wantCS[j] && turn == j)* just a little bit faster. In this case, Process 0 reads *wantCS[1]* as false and enters CS despite the turn being set to 1. As a result, Process 1 can enter the CS while Process 0 is in it due to the turn being set to 1. Having *wantCS* set before turn prevents another process from incorrectly reading *wantCS* before the turn is handed over to the other process.

4. Let P1 and P2 be two processes. In the trivial cases, if neither of the processes want to enter the critical section then nothing occurs and if only one process wants to enter the critical section, then that process will be able to enter the critical section. Consider the non-trivial case when P1 is in the critical section and P2 wants to enter the critical section. In this case, once P1 exits the critical section, it will set it's *wantCS* flag to false, which will break P2 out of its while loop and P2 will enter the critical section. The same argument applies if P2 is in the critical section, and P1 wants to enter the critical section. If both P1 and P2 want to enter the critical section at the same time, the process that sets the *turn* variable second will allow the other process to enter the critical section. Then, once the process in the critical section finishes executing, the other process can enter the critical section, avoiding any possibility of a deadlock situation. Thus, in all cases, we have that a process is able to eventually enter the critical section upon requesting access to enter the critical section, meaning that Peterson's algorithm is starvation-free.