3. TACC UserIDs
Brendan Ngo: bngo
Neil Charles: ncharles

4a. The second condition in the while loop will fail and allow two processes into the critical section. Ex. Process 0 sets wantCS[0] = true, then sets turn = 0, then waits for (wantCS[1] && turn = 1) to be false, which happens right away since turn = 0. So Process 0 is in the critical section. While process 0 is still on the critical section, Process 1 starts. Process 1 sets wanCS[1] = true, then sets turn = 1, then waits for (wantCS[0] && turn = 0) to be false, which happens right away since turn = 1. Thus Process 1 is in the critical section with Process 0 and mutual exclusion is lost.

4b. Setting the turn variable first allows two processes to enter the critical section. Without loss of generality let's assume that P0 sets turn = 1 first. Then P1 goes in and sets turn = 0, wantCS[1] = true, and checks the condition on the while loop where P0 does not wantCS yet so P1 enters the critical section. Now P0 continues where it left off and sets wantCS[0] = true, checks the while loop conditions and sees that P1 wants CS since P1 is in the CS already but that the turn is set to 0 which causes the loop to fail and P0 also enters the critical section, so mutual exclusion is lost.

5.

```
Class PetersonAlgorithm implements Lock{
        boolean wantCS[ ] = { false, false };
        int turn[ ] = {0,0};
        public voidrequestCS(int i){
                Int j = 1-i;
                wantCS[ i ]=true;
                turn[ i ] = turn[ j ] + 1;
                while(wantCS[ j ] && [(turn[ j ] < turn[ i ]) || (turn[ j ] == turn[ i ] && i == 1)] ;
                }
        public void releaseCS(inti){
                wantCS [ i ] =false;
                turn[ i ] = 0;
                }
}
```

We will prove that mutual exclusion is satisfied by the method of contradiction. Suppose, if possible, both processes P0 and P1 are in the critical section for some execution. Each of the processes Pi must have set the turn variable to 1 more than the other's turn variable or both read the others' turn variable at the same time (they would read both turn variables as 0) and incremented their turn variable at the same time so their turn variables have the value 1. Without loss of generality let's assume that P1 set the turn variable last so turn[1] is 2 if turn[1]

incremented after turn[0] or 1 if both processes incremented at the same time. This means that turn[1] was greater than turn[0] when P1 checked the condition on the while loop or both turn[0]==turn[1] if they read each others' values as 0 and incremented their turn values to 1 at the same time. If turn[1] is greater then that or condition evaluates to true. If turn[0]==turn[1] then since i = 1 is P1, the or condition evaluates to true. And since P0 is already in CS, wantCS[0] is true. This leads to a contradiction that P1 is in CS at the same time as P0. Because if turn[0] and turn[1] are the same then the or condition evaluates to false for P0 because i = 0 is P0 and it enters the CS whereas P1 is waiting at the while loop.

To check the progress property, we check if both processes are forever checking the entry protocol in the while loop then we get:
(wantCS[1] and [(turn[1] < turn[0]) or (turn[1]==turn[0] and process == 1)]) and
(wantCS[0] and [(turn[0] < turn[1]) or (turn[0]==turn[1] and process == 1)])
Without loss of generality in the P0's while loop the first part of the or condition could be true and the second part false and vise-versa in the P1's or condition. But both processes can't be 1 so this is false.

We will prove freedom from starvation with a direct proof. Without loss of generality lets assume that P0 is in the critical section and P1 is waiting at the while loop. When P0 goes into releaseCS it makes wantCS[0] false which will let P1 go into CS. Since we have proved mutex, we know that both processes cannot go into CS at the same time.

6. In the absences of choosing variables, the bakery algorithm fails because it loses mutual exclusion and allows two threads into the critical section. Consider the bakery algorithm without choosing variables. Process 0 first requestCS and finishes the first for loop so number[0] = 0. Before Process 0 increments number[0], Process 1 enters requestCS and finishes choosing a number so number[1] = 0 also. Process one increments itself so number[1] = 1 and then enters the second for loop. Process 1 enters the critical section because supposedly no other process is requesting the CS since every number in numbers is 0 (numbers[0] = 0 still). Now Process 0 resumes and assigns numbers[0] = 1. Process 0 then enters the second for loop and makes it through into the critical section because numbers[0] == numbers[1] and 0 < 1 so Process 0 has priority. Thus both Process 0 and Process 1 are in the critical section and the bakery algorithm fails to uphold mutual exclusion without choosing variables.