CS350 hw7 zheming sun
mikesun@bu.edu

1.
a.
Yes, the entry keep checking the flags when the active process is processing. In the first while loop, index is initialized with current turn, if the flags[turn] is processing(not in IDLE mode), the while loop keep checking(waiting) until the current process finishes and set turn to other process. Then the while loop start again, as long as the index updated not to i. Since the value of index and turn is always incremented, then finally index will be equal to process i and the waiting process will exit this busy waiting section. But before exit, it is stuck here as long as all previous waiting process done their tasks.

b.
yes, when the previous process is done using critical section and it is going to assign the turn to next process. there is a while loop at the botom of the code. It keep looking for the process that is in waiting status. Say we start from process 2, it checks 3,4,5,1 in order. If the status of process 3, 4 and 5 are idle and process 1 is wating. then index will be assigned to 1 and so be turn assigned to 1.

c. p2 because p2 starts at the first.

d. p4, p0.

e.
yes. After p2 finished, p2 set turn be 0 since the assignment is in order. And p0 starts using the critical section, then finishes while p1 is waiting. When p0 set the turn to 1. p1 exits the entry while loop and now p0 set back to idle state(which let p1 pass the second wait-other-active-process while loop) and finish the whole process once. RIGHT HERE, imagine, p0 is really slow to start the next request, and p1 runs like flash, finishes using crtical section, and start assigning turn. Since now, all other processes are in idle state, except p1 is in active. The while loop assign turn to non-idle process, so either waiting or active could get the next turn. Then p1 could repeat the process, as many times as before p0 set it flag to waiting.

2.

a.
Initialize semaphore[i] to 0
process i will signal itself upon arrival and then in the for loop, it waits and signals the other processes in pair.
process i can be executed only when all other processes arrive. The arrival (say j) will only signal one waiting process(say i) and let i continue. i will signal j again, the next process waiting for j will get continued. j's arrival triggers a series of signaling because there are N processes in the system, so there must be N-1 processes waiting for j except itself. Processes arrive after j, will not need to wait at all because j's value is 1 because it signals itself to 1 and the rest of wait and signal calls are in pair. So j's value is always 1 after arrived till the end. Clearly, no process will proceed without arrival of all other processes due to the for loop makes each process waits

for rest. And there is no deadlock since all new arrivals signal itself at the first and the rest calls is in wait-signal fashion, so after one waiting process is signaled, it will signal the others. This code works for rendezvous purpose.

b.
the initial value for semaphore[i] is all 0's.
That code can exactly work once bacause after the first run, the values of all semaphores are 1's because in the code there are 1 more signal call than wait calls and it sets all semaphores value to be 1 at the end. Therefore, if the processes run repeatedly(say in the second run), with semaphores' initial values being 1 instead of 0, the first arrival process won't need to wait for anyone! It is signal itself at the first, and no wait calls can stop(block) it because all semaphore values are 1's. There is no rendezvous.

c.

```
for(n=1; n<=N-1; n++) {
        signal  (semaphore[i]);
}
for(j=0; j<N; j++) {
        wait(semaphore[j]);
}
```

d. (codes are attached)

3. (codes are attached)

4.
a.

```
        private static Semaphore biSema[] = new Semaphore[5];
        static {
                for(int i=0; i<5; i++) {
                        biSema[i] = new Semaphore(0);
                }
        }
        private static Semaphore mutex = new Semaphore(1);
        private static volatile int[] rankList = new int[5];
        private static volatile int counter =0;
        private int id;
```

Explaination:
semaphore biSema[]: is the list of binary semaphore for each process
semaphore mutex: used to increment/decrement the counter in entry and exit
int[] rankList: used for process to check the highest rank
int counter: to count the number of processes in the system.

int id: the id of each process

b. code is attached

c. code is attached

d. code is attached

e. Yes, there will definitely be starvation for low priority processes. It will be cut in the way all the time. Yes there is a bound number of out-of order use of the semaphore for c and d. And it will be N times bounded(the number of process).