1.

a. Mutual Exclusion is not satisfied.

P0	**P1**
0: while (true) {	
	0: while (true) {
1: flag[0] = true;	
	1: flag[1] = true;
2: while (flag[1]) {	
	2: while (flag[0]) {
3: flag[0] = false;	
	3: flag[1] = false;
4: while (flag[1]) { //fail because flag[1] = false	
	4: while (flag[0]) { //fail because flag[0] = false
7: flag[0] = true;	
	7: flag[1] = true;
8: }	
	8: }
9: critical_section	
	9: critical_section
…	…

b. Progress is satisfied.

In this algorithm, the only part of code where a process cannot advance before getting to its critical section is the while loop from line 4 to 6. If both processes are stuck in this loop forever, progress is not guaranteed. However, this is not possible because before the loop, at line 3, the flag of any process is set to false. This allows the while loop of the other process to fail, which allows that other process to advance to its critical section. In other words, if 1 or even both processes get into the while loop from line 4 to 6, there is no possible way both flags are true at that same time and create the above scenario.

After 1 process exits its critical section, in line 10 it sets its flag to false so the other process can end the while loop and enter its critical section.

c. Bounded Waiting is not satisfied. 1 process can keep executing its statements without letting the scheduler switch to the other process.

P0	**P1**
0: while (true) {	
1: flag[0] = true;	
	0: while (true) {
2: while (flag[1]) { //fail because flag[1] = false	
	1: flag[1] = true;
9: critical_section	
10: flag[0] = false;	
11: remainder_section	
12: }	
0: while (true) {	
1: flag[0] = true;	
	2: while (flag[0]) {
	3: flag[1] = false;
2: while (flag[1]) { //fail because flag[1] = false	
9: critical_section	
10: flag[0] = false;	
11: remainder_section	
12: }	
… **P0** keeps executing forever	… **P1** is not scheduled to execute anymore since there is no algorithm to prevent the scheduler from executing **P0** forever

2.

a. The problem in this code example comes from the global variable number_of_processes.
1st scenario: The race condition can happen when 2 fork() are called at the same time and number_of_processes at that time is MAX_PROCS - 1 (i.e., number_of_processes = 1534). In this situation, just 1 more process is allowed to be created. Say both fork() enter code "if (number_of_processes == MAX_PROC)" at the same time, since the global variable now is

1534, the condition fails in both fork() and both of them move to the "else" branch. Next, they freely allocate new resource and create 2 more processes without any lock. Now, num_of_processes is incremented by 2 and becomes 1536 > MAX_PROCS, creating the race condition.

2nd scenario: When 1 fork() and 1 exit() are called at the same time, number_of_processes got ++ and - -, so afterward it keeps the same value as before. But if in the assembly level, the scheduler creates interleaved execution by putting - - operation in the middle of ++, and say the - - happens after the ++, value of num_of_processes will be updated wrongly (decremented by 1 at the end), creating the race condition.

b.
```
#define MAX_PROCS 1535
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
   int new_pid;
   mutex.acquire() ;
   if (number_of_processes == MAX_PROCS) {
      mutex.release() ;
      return -1;
   }
   else {/* allocate process resources and assign the PID to new_pid */
      ++number_of_processes;
      mutex.release() ;
      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
   /* release process resources */
   mutex.acquire() ;
   --number_of_processes;
   mutex.release() ;
}
```

c. The solution mentioned in part c will not solve the problem. Atomic variables and operations can help prevent race condition in the 2nd scenario above, but not the 1st scenario. Mutex can, but atomic operations cannot prevent 2 fork() accessing number_of_processes at the same time and modifying it out of valid range.

3.

```
s1 = 0; s2 = 0; s3 = 0; s4 = 0; s5= 0; s6 = 0;
p1: body; V(s1); V(s1); V(s1);
p2: P(s1); body; V(s2);
p3: P(s1); body; V(s3); V(s3);
p4: P(s1); body; V(s4);
p5: P(s3); body; V(s5);
p6: P(s3); body; V(s6);
p7: P(s2); P(s4); P(s5); P(s6); body;
```

4.

```
void produce(int v)
{
   if (numItems == MAX_ITEMS)
      full.wait() ;
   items[numItems] = v ;
   numItems++ ;
   empty.signal() ;
}

int consume()
{
   if (numItems == 0)
      empty.wait() ;
   numItems- - ;
   full.signal() ;
   return items[numItems];
}
```