1. (a) Multiple processes are updating the variable number_of_process, so there will be a race condition. In a process, this situation can be possible: in if, the number_of_process is checked that it is smaller than 255, but later, some other processes increment the value to be 255. However, the original process goes into the else statement and increments the variable, so it is greater than 255. It doesn't work correctly.

   (b)

   ```
   int allocate_process(){
    acquire();
   int new_pid;
   …
   else{
    return new_pid;
   }
    release();
   }
   Void release process(){
    acquire();
   --number_of_processes;
   release();
   }
   ```

   (c) It cannot work. The purpose of this program is to limit the total number

of processes to be fewer than 255. Even though we use atomic variable, there still can be some processes updating the variable between an if and else statement of a process. The variable in else statement should have exactly the same value as the variable in if statement to make sure that the total number of processes doesn't exceed 255.

2.

(a)Mutual Exclusion satisfied. When P0 enters its critical section, flag[0] = true; flag[1] = false; Because flag[0] = true, p1 is either looping in line 2 or in line 4. It cannot enter into critical section. Vice versa, when p1 is in critical section, p0 is looping either in line 2 or line 4 since flag[1] = true.

(b)Progress is not satisfied. Progress should be satisfied without time dependent. If it works in the following way, none of the two processes can reach critical section forever.

| P0: | P1: |
|---|---|
| 1: flag[0] = true; | |
| | 1: flag[1] = true;<br>2: while(flag[0] = true)<br>Goes into the loop |
| 2: while(flag[1])<br>Goes into the while loop | |
| 3: flag[0] = false; | |
| | 3: flag[1] = false; |

| | |
|---|---|
| 4: while(flag[1]);<br>Jump out of the loop | |
| | 4: while(flag[0]);<br>Jump out of the loop |
| 7: flag[0] = true | |
| | 7: flag[1] = true |
| Goes back to line 2, and repeat<br>From line 2 to line 7 continuously | |
| | Goes back to line 2 and repeat from<br>line 2 to line 7 continuously |

If the scheduler works in this way, two processes can never reach critical sections. Progress is violated.

(c)

Under the following situation, process 0 can enter critical section, while process 1 keeps looping and can never access critical section.

| P0 | P1 |
|---|---|
| 1. Flag[0] = true; | |
| | 1. Flag[1] = true; |
| | 2. While(flag[0] = true) enters loop |
| | 3. Flag[1] = false; |
| 2. While(flag[1]) skips loop | |
| 9. critical section | |

| | |
|---|---|
| | 4. While(flag[0]) enters and loops |
| 10. flag[0] = false; | |
| | Jumps out of loop between line 4 and line 5 |
| | 7. Flag[1] = true; |
| 11. remainder_section | |
| 1. Flag[0] = true | |
| | 2. While(flag[0]) enters loop |
| | 3. Flag[1] = false; |
| 2.While(flag[1]) skips loop | |
| 9. critical section | |

It starts repeating now. If it runs and repeats like the above and repeats, process 1 can never reach its critical section. It waits forever.

3.

```
Monitor bounded_buffer{
        Int items[MAX_ITEMS];
        Int numItems = 0;
        Condition full, empty;
       Void produce(int v);
       Int consume();
        Void produce(int v){
             While(numItems >= MAX_ITEMS){
```

```
            Full.wait();}
        Items[numItems] = v;
        numItems++;
        empty.signal()
    }
    Int consume(){
        Int temp;
        While(numItems == 0){
            Empty.wait();}
        Temp = items[numItems - 1];
        numItems--;
        full.signal();
        return temp;
    }
}
```

4.

S1 = 0; s2 = 0; s3 = 0; s4 = 0; s5 = 0; s6 = 0; s7 = 0; s8 = 0;

Pr1: body; V(s1); V(s1); V(s1);

Pr2: P(s1); body; V(s2); V(s2);

Pr3: P(s2); body; V(s3);

Pr4: P(s2); body; V(s4);

Pr5: P(s1); body; V(s5);

Pr6: P(s1); body; V(s6);

Pr7: P(s6); body; V(s7)

Pr8: P(s4); P(s5); body; V(s8);

Pr9: P(s3); P(s8); P(s7); body;