CS149
HW3
Jooyul Yoon
014597154

Q1.

a. Mutual Exclusion
   - This is satisfied because this algorithm is preventing both processes accessing critical section at the same time by using flags.

b. Progress
   - This is not satisfied. When no one is in critical section and when both processes try to flip their flags at the same time, they will block each other.

P0 – flag[0]	P1 - flag[1]
1: true	
	1: true
3: false	3: false
7: true	7: true
3: false	3: false
7: true	7: true
4: infinite while loop	4: infinite while loop

c. Bounded waiting
   - This is not satisfied since P0 can access the critical section endlessly.

P0	P1
1: true	
9: critical section	
10: false	
1: true	
9: critical section	
10: false	

Q2.

a. There is a race condition on the integer variable ‘number_of_processes’ in line 2.

b.

```
#define MAX_PROCS 1023
int num_of_procs = 0; /* the implementation of fork() calls this function */
mutex_lock mx; // mutex lock

int allocate_process() {
        int new_pid;
        mx.lock();
        if (num_of_procs == MAX_PROCS) {
                mx.unlock();
                return -1;
        }
        else {/* allocate process resources and assign the PID to new_pid */
                ++num_of_procs;
                mx.unlock();
                return new_pid;
        }
}

/* the implementation of exit() calls this function */
void release_process() {
        /* release process resources */
        mx.lock();
        --num_of_procs;
        mx.unlock();
}
```

c. No, we can’t. “allocated_process()” is the place where race occurs. The process needs to go to the if statement first and get tested.

Q3.

```
s1=0, s2=0, s3=0, s4=0, s5=0, s6=0
Pr1: body; V(s1); V(s1); V(s1);
Pr2: P(s1); body; V(s2);
Pr3: P(s1); body; V(s3);
Pr4: P(s1); body; V(s4);
Pr5: P(s3); body; V(s5);
Pr6: P(s3); body; V(s6);
Pr7: P(s2); P(s4); P(s5); P(s6); body;
```

s1 semaphore would ensure that P2, P3, P4 are executed after P1 is executed completely.
s2 semaphore would ensure that P7 is executed after P2 is executed completely.
s3 semaphore would ensure that P5, P6 are executed after P3 is executed completely.
s4 semaphore would ensure that P7 is executed after P4 is executed completely.
s5 semaphore would ensure that P7 is executed after P5 is executed completely.
s6 semaphore would ensure that P7 is executed after P6 is executed completely.

Q4.

```
monitor bounded_buffer {
        int items[MAX_ITEMS]; /* MAX_ITEMS is a constant defined elsewhere; not a circular buffer */
        int numItems = MAX_ITEMS; /* # of items in the items array, 0 ≤ numItems ≤ MAX_ITEMS */
        condition full, empty;
        /* both produce() and consume() use numItems as index to access the array */

        void produce(int v); /* deposit the value v to the tail of the items array */
        int consume(); /* remove an item from the tail of items array, and return the value */
}
```

```
produce(int v){
        while(1){
                Resource buffer[MAX_ITEMS];         // Produce new source
                wait(empty);                        // wait for empty buffer
                wait(mutex);                        // lock buffer list
                buffer[MAX_ITEMS] = v;               // Add resource to an empty bugger
                signal(mutex);                      // unlock buffer list
                signal(full);                       // note a full buffer
        }
}

consume(){
        while(1){
                wait(full);                         // wait for a full buffer
                wait(mutex);                        // lock buffer list
                buffer[MAX_ITEMS] = null;           // Remove resource from a full buffer
                signal(mutex);                      // unlock buffer list
                signal(empty);                      // note an empty buffer
                free(buffer)                        // consume resource
        }
}
```