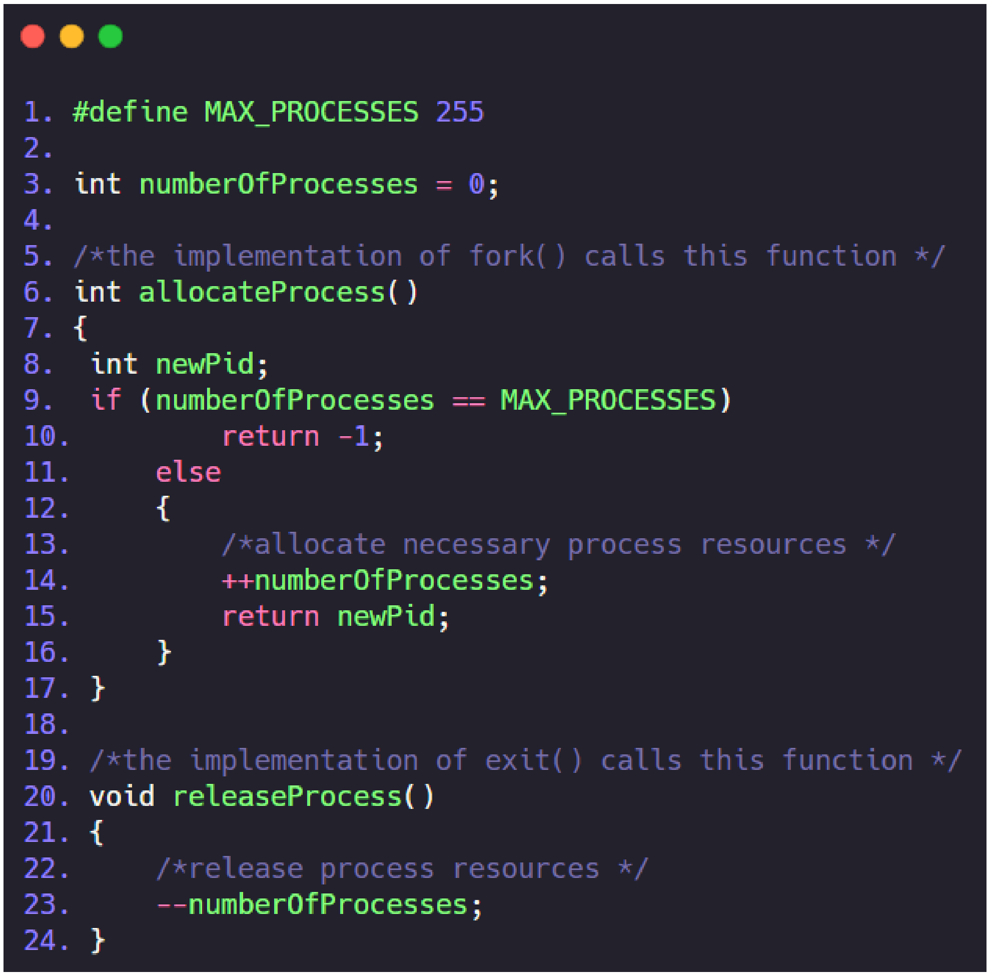
Operating System (CS301)

Assignment - 7

U19CS012

1. Consider this code example for allocating and releasing processes:



```
1. #define MAX_PROCESSES 255
2.
3. int numberOfProcesses = 0;
4.
5. /*the implementation of fork() calls this function */
6. int allocateProcess()
7. {
8.  int newPid;
9.  if (numberOfProcesses == MAX_PROCESSES)
10.         return -1;
11.     else
12.     {
13.         /*allocate necessary process resources */
14.         ++numberOfProcesses;
15.         return newPid;
16.     }
17. }
18.
19. /*the implementation of exit() calls this function */
20. void releaseProcess()
21. {
22.     /*release process resources */
23.     --numberOfProcesses;
24. }
```

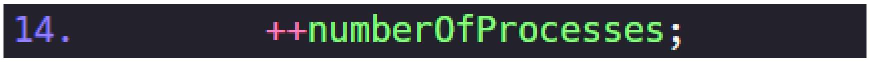
A) Identify the race condition(s). Be specific — refer to the code.

Race condition occurs when multiple threads **read** and **write** the *same variable* i.e. they have access to some shared data and they try to change it at the same time. In such a scenario threads are "racing" each other to access/change the data.

Data Involved in Race Condition: The Variable "**numberOfProcesses**"

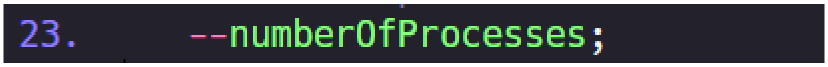
The Location in the Code where the Race Condition occurs:

Line Number 14 -



```
14.         ++numberOfProcesses;
```

Line Number 23 -

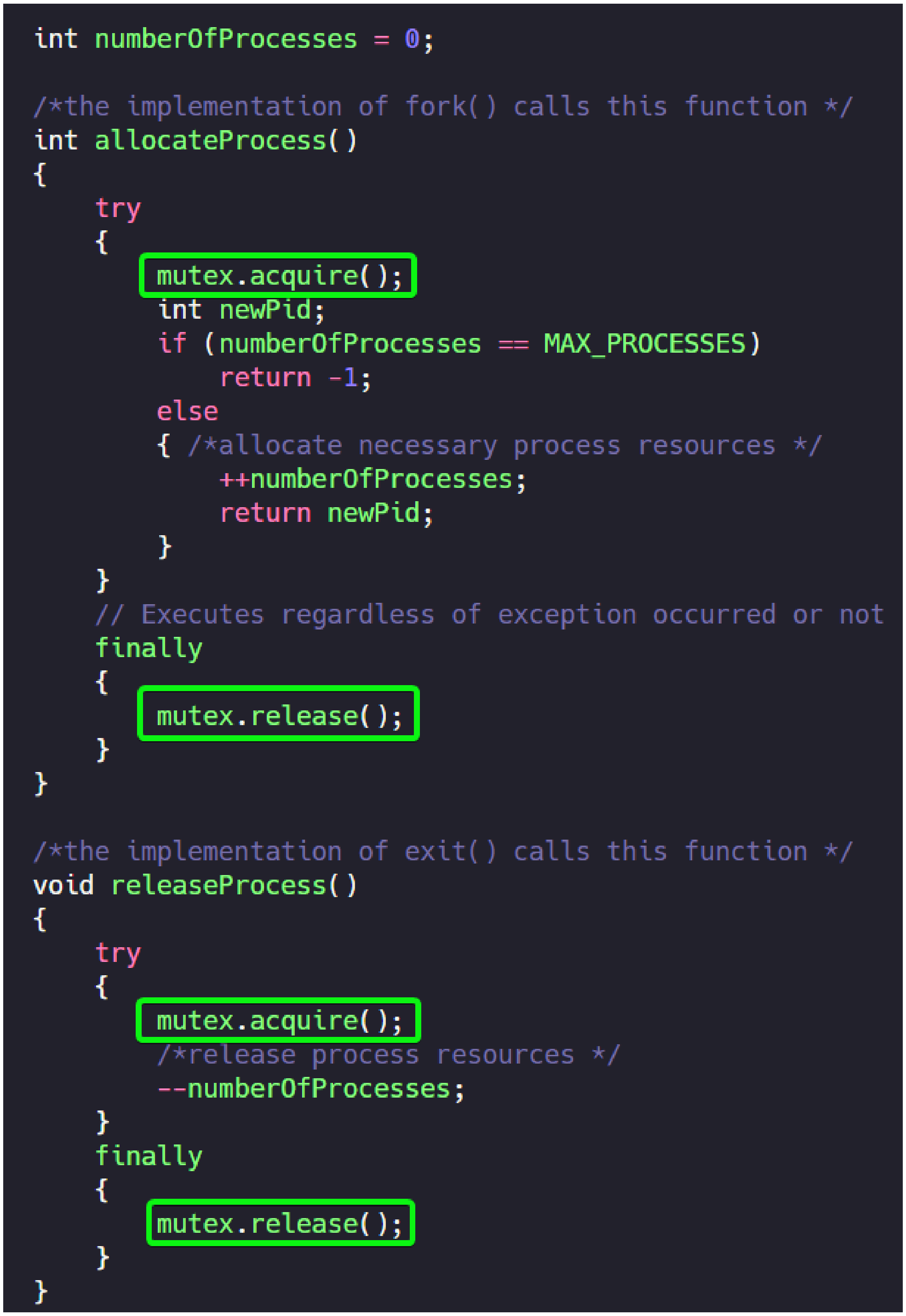


```
23.     --numberOfProcesses;
```

The code that **decrements** number_of_processes and the code that **increments** number_of_processes are the statements that could be involved in Race Conditions.

B) Assume that you have a mutex lock named `mutex` with the operations `acquire ()` and `release()`. Indicate where in the code above that the locking/unlocking needs to be placed to prevent the race condition(s).

Method 1



```
int numberOfProcesses = 0;

/*the implementation of fork() calls this function */
int allocateProcess()
{
    try
    {
        mutex.acquire();
        int newPid;
        if (numberOfProcesses == MAX_PROCESSES)
            return -1;
        else
        { /*allocate necessary process resources */
            ++numberOfProcesses;
            return newPid;
        }
    }
    // Executes regardless of exception occurred or not
    finally
    {
        mutex.release();
    }
}

/*the implementation of exit() calls this function */
void releaseProcess()
{
    try
    {
        mutex.acquire();
        /*release process resources */
        --numberOfProcesses;
    }
    finally
    {
        mutex.release();
    }
}
```

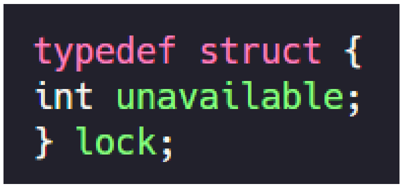
Method 2

The programmer can place the two operations at the very first and end of a function call as well.

The **acquire()** function should be placed in the beginning of function call.

Whereas, **release()** operation call should be placed just before the end of function call.

2. Consider how to implement a `mutex` lock using an Atomic Hardware Instruction. Assume that the following structure defining the mutex lock is available:



```
typedef struct {
int unavailable;
} lock;
```

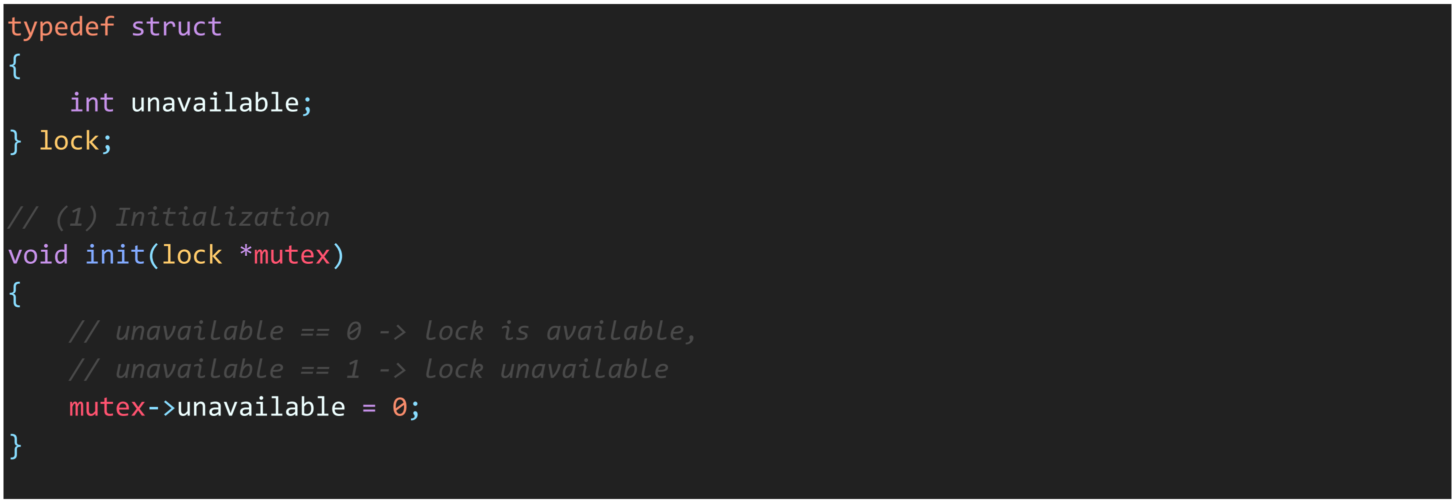
(unavailable == 0) indicates that the lock is available &

(unavailable == 1) indicates that the lock is unavailable.

Using this struct, illustrate how the following functions can be implemented using the test_and_set() instruction and compare_and_swap() instructions:

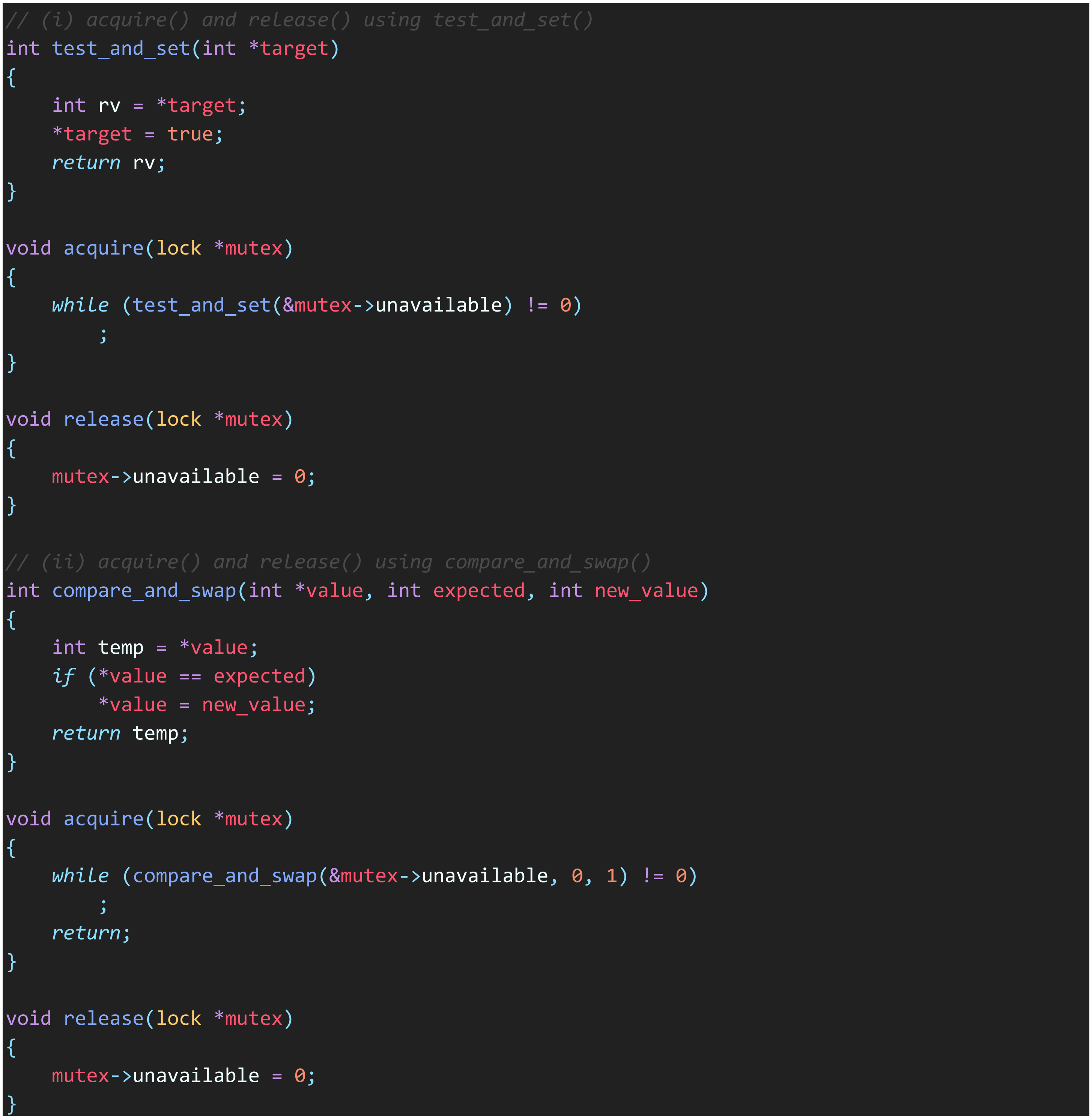
- void acquire(lock *mutex)
- void release(lock *mutex)

Be sure to include any initialization that may be necessary.



```
typedef struct
{
    int unavailable;
} lock;

// (1) Initialization
void init(lock *mutex)
{
    // unavailable == 0 -> lock is available,
    // unavailable == 1 -> lock unavailable
    mutex->unavailable = 0;
}
```



```c
// (i) acquire() and release() using test_and_set()
int test_and_set(int *target)
{
    int rv = *target;
    *target = true;
    return rv;
}

void acquire(lock *mutex)
{
    while (test_and_set(&mutex->unavailable) != 0)
        ;
}

void release(lock *mutex)
{
    mutex->unavailable = 0;
}

// (ii) acquire() and release() using compare_and_swap()
int compare_and_swap(int *value, int expected, int new_value)
{
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}

void acquire(lock *mutex)
{
    while (compare_and_swap(&mutex->unavailable, 0, 1) != 0)
        ;
    return;
}

void release(lock *mutex)
{
    mutex->unavailable = 0;
}
```

SUBMITTED BY:

U19CS012

BHAGYA VINOD RANA