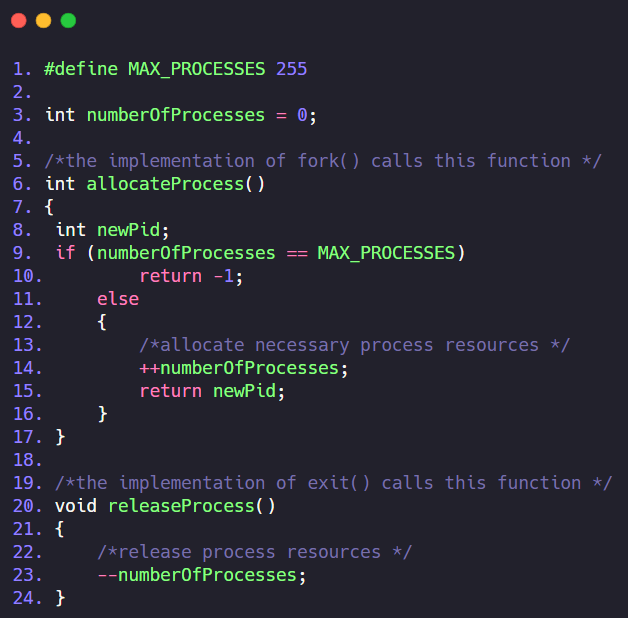
Operating System (CS301)

Assignment - 7

**U19CS012**

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



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 - 

Line Number 23 - 

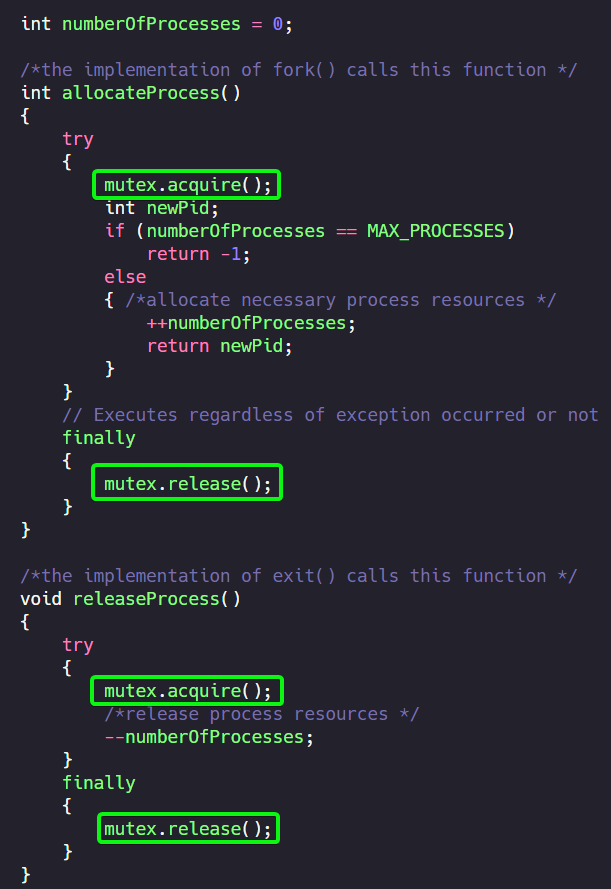
The code that **decrements** number\_of\_processes and the code that **increments** number\_of\_processes are the statements that could be involved in Race Conditions.

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



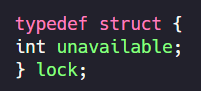
Method 2

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:



(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;

}

*// (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