Homework Week 9

Tanu Shree

1. The bugs in the implementation are at**:**

- Line 27: Before throwing the exception, the lock must be released, otherwise no other process/thread would be able to access the stack ever in the future. So lock.unlock() should be called before that.
- Line 30: Stack should be incremented before the lock is released, otherwise there will be incorrect reference to the stack by other processes.
- Line 37: Like the error at line 27, the lock must be released before throwing the exception.

2. Considering Peterson's solution to the mutual exclusion problem. Suppose we make a small change to the solution and change the line "turn = other" to "turn = self", both processes can be in their critical section at the same time:

Thread0	**Thread1**
Running …	Blocked or ready…
Busy [0] = true;	**-**
Other = 1	**-**
Turn = 0;	**-**
-	Thread1 took CPU
-	**Busy [1] = true**
Thread0 took CPU	**-**
Busy [1] = true && turn! =0 ?	**-**
(Since turn = 0, so false, hence no busy waiting)	**-**
Thread1 acquires lock and enters critical section.	**-**
-	Thread 1 took CPU
-	**Other = 0**
-	**Turn = 1**
-	**Busy [0] = true && turn! = 1 ?**
-	(Since turn = 1, so false, hence no busy waiting)
-	Thread1 acquires lock and enters critical section.

In the above scenario, both the threads end up acquiring the lock and entering the critical section. Therefore, mutual exclusion is not achieved here.

Homework Week 9
Tanu Shree

3. The kernel in the uniprocessor can be accessed by interrupts. When the interrupt occurs and a thread is running inside the kernel, that thread could get preempted when it is executing some critical sections of code like modifying some data structure. This can lead to the discrepancy in the invariants. Disabling/masking interrupts before getting inside the critical section can help preventing such unwanted preemption of thread and hence invariants can be maintained till the completion of that section. After exiting the critical section, interrupts can be enabled/unmasked so that other threads could use the critical section.

4. The hardware solution mentioned in the question can help implement a solution the critical Section as follows.

```
Atomic_Counter  a = 0;          // a is an atomic counter
int j;                          // j is the variable used to sample the atomic counter a

do {                            //lock
        j = a++ ;
        while(j != 1)
}

/* Enter Critical Section */
....

/* Exit Critical Section */
a = 0;                          // Unlock
```