CMPE 436

HW1

Mahir Efe KAYA - 2016400195

1)

In the TestAndSet question, I used an int variable called “next” to settle the starvation.

The next variable sets which thread enters the Critical Section next. One thread can only enter the CS if the “next” variable has been set to its id or the variable initial value,-1.

“next” variable has been set as like this:

If one thread requests the lock, but another has already been inside, “next” has been set to its id. So whenever the threads exits and requests the lock again, the lock will let inside whoever next variable points to.

If the thread inside releases the lock when no one requests, since it can check with a condition sentence of whether “next” has still been set to itself, it will set “next” to its initial value, -1, so that next time one wants to enter, be able to enter the CS. With that Progress principle has been achieved.

Mutual Exclusion has been handled with the previous lock, TestAndSet.

2)

In the swapLock, I have given the simple lock design, which gives the permissions by swapping the values between one thread and the lock. I have created an array to store the Boolean values of the each thread, in other words permissions, and one Boolean value to set lock’s state.

When lock value is false, no thread has been let inside, no thread’s value has been swapped to true.

When lock value is true, the thread comes first will be swapping its false value with the lock’s true value and be let inside.

With its being atomic, the previous example’s being not mutual exclusive has been handled. However it is still not starvation-free.

I had to change the swap function for it to be able to work in java. Since java is pass-by-value, the changings made inside does not effect the outside its scope. To handle it, I have given the values inside an array. With that, the function become able to work.