OS2 Theory Assignment-1
Vanga Aravind Shounik-CS20BTECH11055

Q1

Consider the atomic ‘increment’ function discussed in the book (page 270 of the pdf) using hardware instruction ‘compare_and_swap’. As of now, there is no guarantee that a thread invoking this function will terminate. So, can you develop an ‘atomic’ increment function that will eventually terminate?

Solution:

The code is

```
void Increment(atomic_int *v){
      int temp;
      do{
            temp = *v;
      }
      while(temp!=compare_and_swap(v,temp,temp+1));
}
```

Here, it is not correct in some cases when the Increment function is called twice at the same time in 2 different instances. Then if the first process increments the value when the second process is at temp=*v, then the second process doesn’t contain the same value as v. So, the while loop may run infinitely.

To solve this problem, we can add a Semaphore variable named `lock` and then we can call wait() and signal() operations before and after the process.

```
void Increment(atomic_int *v){
      int temp;
      wait(&lock);
      do{
            temp = *v;
      }
      while(temp!=compare_and_swap(v,temp,temp+1));
      signal(&lock);
}
```

This will solve the issue by letting only one process run at a time.

Q3

Here, we can use the 2nd process where we check the status of the lock before implementing compare_and_swap(lock,0,1). It works perfectly fine because in this case, it checks whether the lock is open and if it is open, then it checks for the compare_and_swap() process again to check whether any other process has taken over the lock. Here, after we enter the if statement, we check whether the lock is open or closed again. Here, the second checking of the lock is important because if we didn;t check the lock the second time, consider a case where 2 processes check whether the lock is open and 2 processes goes inside the if statement and here if there just a break statement without the lock again, then maybe 2 processes may run the critical section together which may cause data unsynchronization. But in this code, we check for the lock again. So, we can see that no 2 processes run their critical section together.

So, The compare_and_compare_and_swap works properly.

Q4

Here, the given code is

```
if(getValue(&sem)>0)
      wait(&sem);
```

Here, let us consider a case where the initial value of sem is 1.
Let P1,P2 are processes which come and execute the getValue and here, they wait(). If P2 executes first then P1 and decrements the value of sem to 0. Then , P1 has to run again in the loop and before it executes the wait function, if P2 completes its process and increments the sem then P1 or any other process may take place at that instance. Here, we can see that if P1 doesn’t execute in such cycles for a long time, it may lead to starvation.