Ques 1

```
int lock = 0;
bool waiting[] = {false}
void increment(atomic_int *v, int i, int n) {
        int temp;
        waiting[i] = true;
        int key = 1;
        while (waiting[i] && key == 1)
                key = compare_and_swap(&lock,0,1);
        waiting[i] = false;
        /* critical section */
        do {
                temp = *v;
        } while (temp != compare_and_swap(v, temp, temp+1) );
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = 0;
        else
                waiting[j] = false;
        /* remainder section */
```

Ques 2

```
int readcount; // initialize to 0
semaphore rw_mutex, mutex, queue; // all initialized to 1
//Reader
reader() {
        //entry section
        wait(queue);
        wait(mutex);
        readcount++;
        if(readcount == 1)
                wait(rw_mutex);
        signal(queue);
        signal(mutex);

        //critical section
        //reading id done
        //exit section
        wait(mutex);
        readcount--;
        if(readcount == 0)
                signal(rw_mutex);
        signal(mutex);
}

//Writer
writer() {
        //entry section
        wait(queue);
        wait(rw_mutex);
        signal(queue);

        //critical section
        //perform writing

        //exit section
        signal(rw_mutex);
}
```

Ques 3

The “compare and compare_and_swap” works appropriately, as it ensures mutual exclusion.
Firstly we check the value of lock. If it is 0, then call compare_and_swap() which changes lock to 1. Next process cannot go into CS.
Hence, the integrity of lock is not compromised.

Ques 4

Suppose we have 3 threads/ processes P1, P2, and P3.
The value of semaphore can always be 0 for any one particular process say P1. This is because whenever it checks for the value of semaphore using getValue(&sem), another process say P2, can be using the semaphore. After it releases it, lets say the value of semaphore is changed to 1. Now some other process say P3, can get the semaphore seeing that that the value id greater than 0, and again reduce the value to 0. Thus, process P1 would never get the semaphore. Hence leading to starvation.