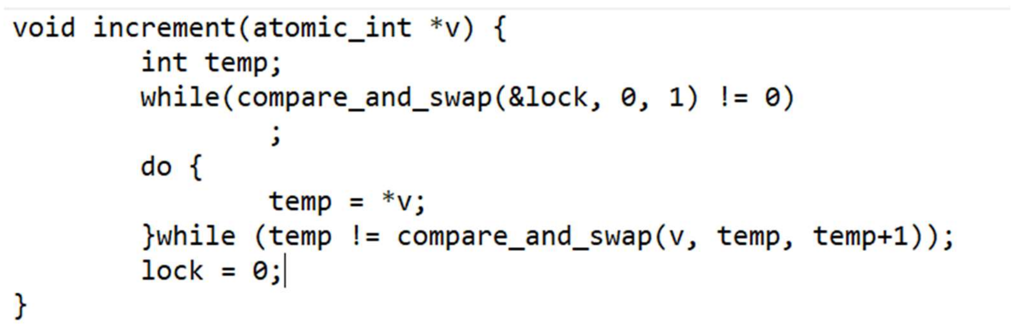
Operating Systems 2

Tanmay Garg CS20BTECH11063

Theory Assignment 1

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 invokind this function will terminate. So, can you develop an 'atomic' increment function that will eventually terminate?

Sol.



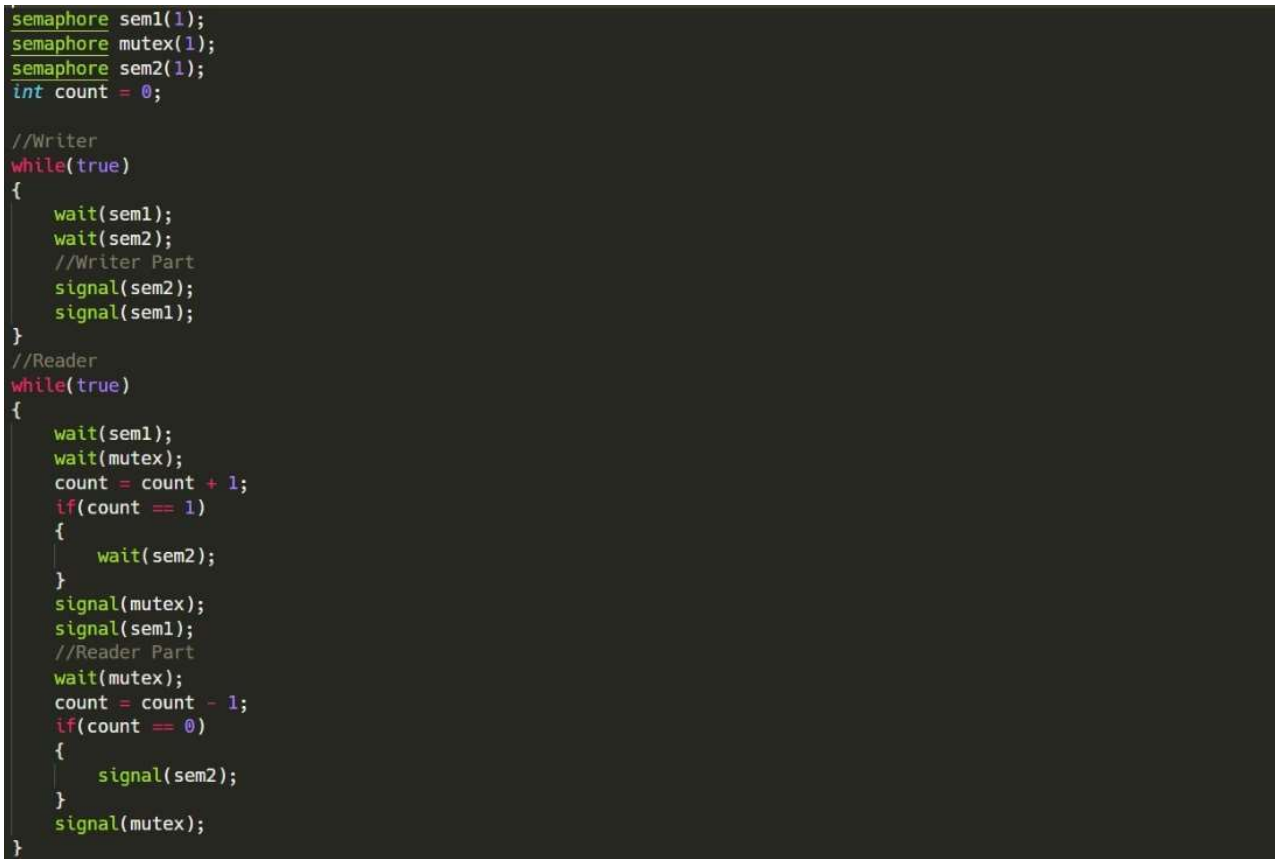
```
void increment(atomic_int *v) {
        int temp;
        while(compare_and_swap(&lock, 0, 1) != 0)
                ;
        do {
                temp = *v;
        }while (temp != compare_and_swap(v, temp, temp+1));
        lock = 0;
}
```

-
- We can use such an implementation above, where only one process thread will be allowed to increment at a time
- We can use *atomic_fetch_add()* function to help overcome the issue of infinite loops and non-terminating threads
- We may also use *++* operator to increment the atomic variable
- The above methods can be used only if the hardware provides the instructions to implement such operations on atomic variables
- Atomic variables and functions cannot be further broken down and are very similar to machine instructions
- Some architectures may or may not also provide such features
- Some atomic read, modify, and write functions may internally use CAS method
- These instructions modify variables in a single step to help avoid race conditions
- If multiple threads are continuously modifying the variable, then the thread may or may not terminate
- Another way to tackle this starvation, is by providing a method to analyze multiple threads in a fair manner
- CAS functions are lock free algorithms. If an iteration of CAS fails, then it means that a thread has modified the contents of the variable. But this may cause the thread to be "blocked" as it will continuously spin in the loop till it clears
- *Atomic_fetch_add()* functions are wait free algorithms. So, no thread will be prevented to complete itself due to failure of other threads
- Even if such functions do not work, then simple mutex locks can be used in an efficient manner to avoid failure of termination of threads

Q2. In the class we discussed the solution to the reader-writers problem using semaphores. We saw that this solution can cause the writer threads to starve. Please develop an alternative solution in which neither the reader nor the writers will starve. For this you can assume that the underlying semaphore queue is fair.

Sol.

- In this method the writer part will not starve
- The writer part now has the chance of being executed while reader parts are continuously being requested and then modifying variables and then terminating the process or thread
- We have a counter which keeps track of the second semaphore *sem2* and calls *wait()* and *signal()* as when required in the given conditions
- When a reader locks sem1 before the writer, then writer will be added to a waiting queue along with the other readers who are waiting for the semaphore
- When the writer acquires sem1 then it will wait for sem2. The readers in sem1 semaphore waiting queue will wait for writer to allow them to proceed ahead
- The other readers will finish all the tasks and wait in sem1 semaphore waiting queue
- In the end writer will lock both sem1 and sem2 and finish its critical section
- Then the reader will start executing its process and writers will wait in sem1 semaphore waiting queue
- The reader uses multiple mutex locks to prevent race conditions and for program safety
- This provides a faster and much reliable program flow to keep shared variables safe from being unknowingly modified by different threads



```
semaphore sem1(1);
semaphore mutex(1);
semaphore sem2(1);
int count = 0;

//Writer
while(true)
{
    wait(sem1);
    wait(sem2);
    //Writer Part
    signal(sem2);
    signal(sem1);
}
//Reader
while(true)
{
    wait(sem1);
    wait(mutex);
    count = count + 1;
    if(count == 1)
    {
        wait(sem2);
    }
    signal(mutex);
    signal(sem1);
    //Reader Part
    wait(mutex);
    count = count - 1;
    if(count == 0)
    {
        signal(sem2);
    }
    signal(mutex);
}
```

Q3. Exercise 6.11 from the book.

Sol.

- Yes, the “compare-and-swap” idiom works appropriately for implementing spinlocks
- Compare_and_swap takes three parameters: location, “expected_value” of that location, new value of that location
- It checks if the value of the location, matches the expected_value
- If it is same then it updates it with the new value, else it doesn’t make any change
- When the lock is 1, the while loop will continue to run till its value changes to 0 and it will break out of the loop
- When the lock is 0, there can be two situations when this occurs
- If the lock is interrupted before the statement
    - $if(*\, lock == 0)$
- Then at this moment the value will be changed to 1 and then it will continue to run in the while loop
- If the lock is interrupted after the statement
    - $if(*\, lock == 0)$
- Then at this moment the value will be changed to 1 and the inner if statement
    - $if(!\,(compare\_and\_swap(lock, 0, 1)))$
- The *compare_and_swap* will return a value of 1, so the break statement will not be executed
- Therefore, there can be only one thread in the critical section at a time while others wait in the spinlock
- All threads cannot exit while loop together due to the above structure
- The additional statement $if(*\, lock == 0)$ reduces the need for executing another atomic operation on *lock*

Q4. Exercise 6.12 from the book.

Sol.

- When using *getValue()* as described in the scenario doesn't prove to be useful
- When the function returns a particular value that it got, the value could have been changed by some other process
- If the semaphore was free and *getValue()* is called, in that case the process which had called was not waiting using *wait()*
- If we call *getValue()* and then semaphore becomes busy, in that case the process which had called will have to wait for completion as if *wait()* was called
- The function *getValue()* is not an atomic function, so it is not meant to
- Let us say that for some process and the value of the semaphore is 0 and process X
- Let another process Y is currently using the same semaphore. After Y releases and updates the value of semaphore to 1. Other process can get that semaphore by seeing that value and making it to 0
- If at this time, if X comes back and sees the value of semaphore as 0 then it will result in starvation for X
- X never gets the semaphore and will result in timing errors