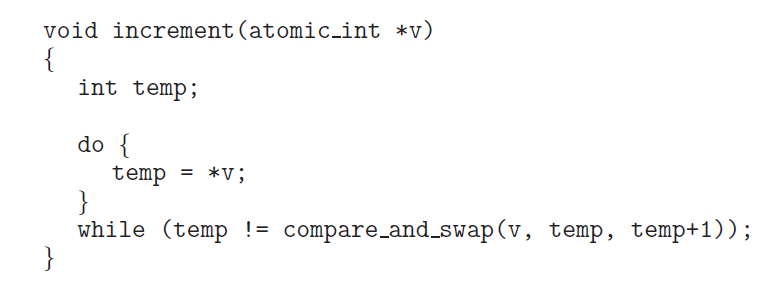
CS3523: Operating Systems - II

Theory Assignment 1

# Soumi Chakraborty

ES19BTECH11017

# Question 1



We can ensure that this atomic function terminates without fail via the usage of mutex locks. This function can be treated as critical section code since it is updating a variable that is possibly shared amongst many processes like we saw in the book. Then, we can only allow processes to execute this function after it uses the acquire() function to acquire the lock. Once it finished executin and exits the while loop, it should call the release() function to release the lock and terminate the function while simultaneously allowing another process to access the updated value of the variable and the function.

# 

# 

# 

# 

# Question 2

The base template for this code was taken from section 7.1.2 in the textbook. A solution which will ensure neither the readers or the writers starve is as follows:

// required data structures

semaphore rw\_mutex = 1;

semaphore mutex = 1;

semaphore fairness\_queue = 1;

int read\_count = 0;

// writer

while (true) {

wait(fairness\_queue);

wait(rw\_mutex);

signal(fairness\_queue);

. . .

/\* writing is performed \*/

. . .

signal(rw\_mutex);

}

// reader

while (true) {

wait(fairness\_queue);

wait(mutex);

read\_count++;

if (read\_count == 1)

wait(rw\_mutex);

signal(fairness\_queue);

signal(mutex);

. . .

/\* reading is performed \*/

. . .

wait(mutex);

read\_count--;

if (read\_count == 0)

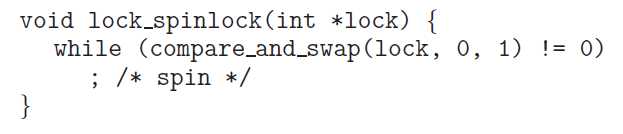
signal(rw\_mutex);

signal(mutex);

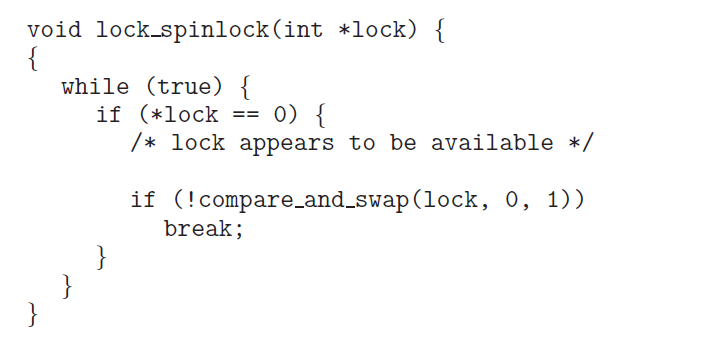
}

# Question 3

The “compare and compare-and-swap” idiom does work appropriately for implementing a spinlock.



On studying the implementation of the spinlock, it becomes clear that when lock == 0, its value is set to one, and compare\_and\_swap() returns 0 which allows the while condition to evaluate to false and break out of the loop. On the other hand, when lock == 1, the value of lock remains unchanged and the function returns 1 and thus doesn’t break out of the while loop.



On examining the code for the “compare and compare-and-swap” method, we realise that it achieves the same thing: it only breaks out of the loop if lock == 0 after setting it to 1, otherwise it remains stuck inside the loop.

# 

# 

# Question 4

In the regular implementation of a semaphore, a process remains stuck inside the wait() while loop until the value of S is incremented by another process after its execution. This waiting process can then immediately begin executing once there is a free resource. However with the proposed getValue() method, it isn’t necessary that a waiting process will get to execute its critical section right after the resource becomes available. If, for instance, two processes try to access the critical section at the same time, only one of them will get it regardless of which process requested it first. Thus, due to this solution, it may happen that a process keeps waiting for a much longer time than it would have otherwise.