Operating Systems–2: CS3523 2019
Theory Assignment 2
Report

Sai Harsha Kottapalli
CS17BTECH11036

March 3, 2019

1

Note: Full semaphore is initialized to 0 and empty semaphore is initialized to n. (a) When n "things" are produced with no consumption, the value of empty becomes "0".
Before writing the next "thing" produced to the buffer, the producer waits for empty semaphore which then becomes "-1" after which the producer is put to sleep. Therefore, the minimum value of empty is -1.
Similarly, the minimum value of full is -1, which happens when the consumer is waiting for at least one entry in the buffer.
(b) The maximum value of empty is n. This happens when the buffer is empty and the consumer cannot increase this value since it is stuck at wait(full).
Therefore, the maximum value of full is also n(when the buffer is full) - symmetrical to previous scenario.
(c) The actual bounds depend upon the number of producers and consumers.
For one thread the sum is always grater than n-1 and less than n.
For No busy waiting, this number can be even lesser or go to negative.

2

Given, the underlying semaphore queue is fair.
If we favor readers over writers, then this would result in the starvation of writers.
If we favor writers over readers, then the resultant throughput i.e. amount of work that can be performed in a given period of time, would be reduced.
Instead we could do the following:

- We maintain timestamps of readers/writers requesting access to critical section.
- When a reader requests for access to critical section, we first check if any writers are waiting to access the same.
- If no writers are waiting then it would proceed else it would wait until there are no more writers.
- If a writer is done executing, we will check from the collected timestamps for the oldest process i.e. the reader/writer which has waited for the longest duration and give it access to the critical section.

This way, neither readers not the writers will starve.

3

We know that compare_and_swap compares the lock with expected value first and then assigns the required value i.e. new value only if the comparison earlier is true.
Note: compare_and_swap is atomic.
In the given implementation of "compare and compare_and_swap", calling compare_and _swap inside the **if** condition (*lock == 0), only lets us know whats the status of lock value.
It is possible that after the condition is checked, the value is changed and before compare_and_swap() is started. However this change doesn't matter since the function intrinsically checks the lock value before making any changes to it.

Therefore, we can say that, this implementation will work.

4

- If two threads were to access the **if** condition of the program concurrently, given that the lock was free, this would result in one thread acquiring the lock while the other thread being blocked waiting for the semaphore.
- If getValue(&sem) returns 0, then it would directly enter the critical section, which is completely against the requirement.