ECE/CS 5510 Multiprocessor Programming
Homework 4

Mincheol Sung

October 22, 2018

Part I

Problem 1.
(1) Mutual exclusion: Yes. Roll is a single register that it can have only a specific value. In addition, read, and write operations are atomic. So a single thread can enter the critical section.
(2) Livelock-free: Yes.
Let there be thread1, 2, 3. All of thread 1, 2, 3 are waiting for ROLL == -1, and try to set ROLL to its PID. Let's assume that thread 3 is always the last one writing its PID to ROLL. Thread 3 can always get into the critical section. Thread 3 finishes its jobs in the critical section and sets ROLL to -1. Other two threads are spinning on ROLL. Then, all threads see ROLL == -1. All try to write ROLL. Finally, Threads 3 succeeds to be in the critical section again. Thread3 always wins. Others keep changing their status but they can't be in the critical section. In this corner case, we still can say this is livelock-free because a thread (thread 3) can always proceed at least.
(3) Performance drawbacks: Everyone spins on a same register causing severe cache contention (everyone will suffer when a cache miss happens). In addition, CAS operation is costly in general. So spinning with CAS operation would be bad. Requirements: In order to operate multiple operations within one time unit, this algorithm requires some systems supports such as timestamp.

Problem 2.
(a)
TAS: Overhead of broadcasting increases
TTAS: Overhead increases during setting flags
ALock: Space hog
CLH: If $n > 10$, thread is spinning on the remote node.
MCS: Thread spins on local node with reading. o performance affected.
(b)
MCS: It writes once on lock() and keep reading.

Problem 3.
Correct. Let there be 4 threads calling lock() in order: $T0 \rightarrow T1 \rightarrow T2 \rightarrow T3$. At the

beginning, only $T0, T1, T2$ will find idle spots due to the array size of 3. $T0$ executes first. $T0$ finishes critical section and does the line 25 in unlock(). In the worse case, $T0$ stops after invoking line 25. $T3$ now can see an idle spot in the array, and it eventually spins at line 21. At this moment, all threads are spinning at line 21 besides $T0$. $T0$ will eventually execute line 24, and finishes all the tasks. $T1$ are triggered to enter the critical section. Therefore the implementation is correct even if $T1$ is interfered in unlock().

Problem 4.
(a) Incorrect. Many can inset to a same spot due to they get the same index. For example, there is a case where:
A get(): A gets index 1
B get(): B gets index 1
A set(): A sets its Qnode to index 1
B set(): B sets its Qnode to index 1

(b) Incorrect. It's deadlock. Threads spinning might be removed from the list.
$T1$ comes, gets the lock, finishes the critical section and unlock. It finds Qnode.next == NULL and tail == itself, preparing to do swap. That is tail = NULL. $T2$ inserts to the tail of the queue. $T3$ comes and finishes all setting, spinning on the lock. $T1$ keeps doing, swap, setting the tail point to NULL. And $T2$ finishes all the setting, spinning on its lock. $T4$ arrives and finds the tail is NULL and acquires the lock. $T5$ arrives behind $T4$. $T2$ and $T3$ spin forever eventually.

(c) Deadlock can only happen where a thread stuck in while() loop making no progress. This will never happen. There are only two cases where Qnode.next is NULL. The first case is when Thread 2 arrives and sets the tail to point itself, but not yet done pre.next = Qnode(). This is where thread 2 is on the half way of lock(). It will eventually finish and trigger the current thread to break the while loop.
The second case is when a thread is the tail, its next points to NULL which can make the thread stuck in the while loop. However, when the thread is the tail, it will leave the unlock when it does the CAS operation. There is no chance to do the while loop.

(d) Incorrect. Let there be a thread executing line 17 and then stops. Someone else comes line 17 and can also see the tail is false meaning that it can get the lock. Therefore, both threads get into the critical section.

Part II

Problem 1.
Experimental environment
CPU: Intel(R) Xeon(R) CPU E5-2470 v2 @ 2.40GHz
Num of Cores: 40
Memory: 100GB
Command: $ java Q1.Test

Execution results:
2 threads:
1 enters foo()
0 enters foo()
1 enters bar()
0 enters bar()
Average time per thread is 0ms

4 threads:
1 enters foo()
3 enters foo()
0 enters foo()
2 enters foo()
3 enters bar()
0 enters bar()
1 enters bar()
2 enters bar()
Average time per thread is 0ms

8 threads:
2 enters foo()
1 enters foo()
0 enters foo()
3 enters foo()
4 enters foo()
5 enters foo()
6 enters foo()
7 enters foo()
4 enters bar()
6 enters bar()
5 enters bar()
0 enters bar()
1 enters bar()
2 enters bar()
3 enters bar()

7 enters bar()
Average time per thread is 1ms

16 threads:
3 enters foo()
0 enters foo()
2 enters foo()
1 enters foo()
4 enters foo()
6 enters foo()
8 enters foo()
5 enters foo()
7 enters foo()
9 enters foo()
11 enters foo()
10 enters foo()
13 enters foo()
12 enters foo()
15 enters foo()
14 enters foo()
3 enters bar()
2 enters bar()
1 enters bar()
14 enters bar()
0 enters bar()
7 enters bar()
6 enters bar()
5 enters bar()
12 enters bar()
11 enters bar()
9 enters bar()
8 enters bar()
15 enters bar()
10 enters bar()
4 enters bar()
13 enters bar()
Average time per thread is 4ms

Problem 2.
Command: $ java Q2.Test

Table 1: Average waiting time in ns

Threads	2	4	6	8	10	12	14	16
TAS	107.40	257.73	378.78	460.45	584.21	689.07	807.31	869.89
TTAS	113..28	261.28	360.69	426.72	508.31	513.75	547.84	579.99
CLH	348.62	379.38	390.72	355.81	373.95	377.13	357.83	366.62
MCS	528.12	349.08	328.88	346.02	331.82	321.84	352.31	356.62

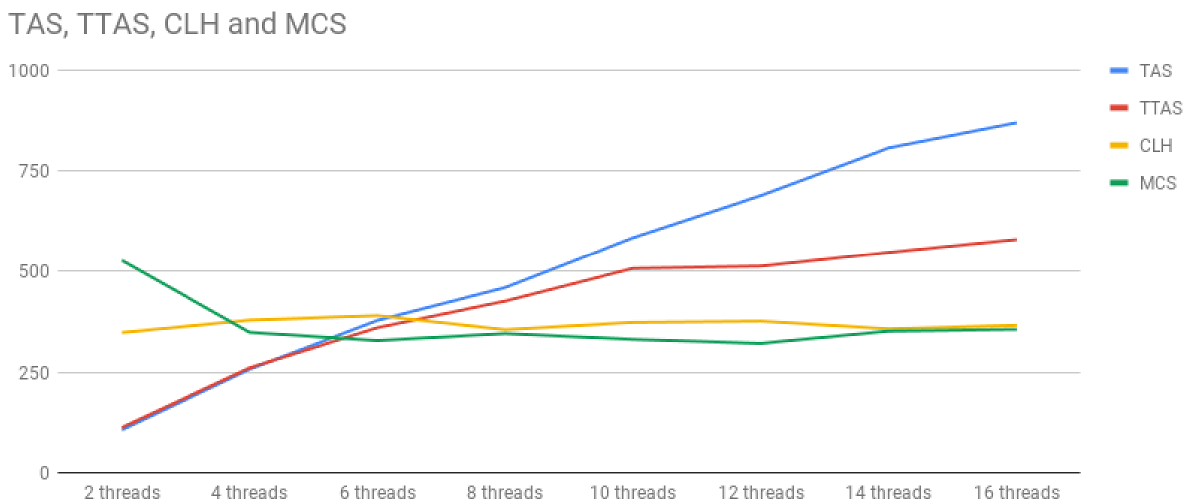

Figure 1: Average waiting time in ns

Problem 3.
Command: $ java Q3.Test (lock)
Command: $ java Q3.Test2 (trylock)
Execution result:
Thread 8(prior: 1) waiting time: 16ms, multiplied wait time: 16ms
Thread 6(prior: 2) waiting time: 59ms, multiplied wait time: 118ms
Thread 7(prior: 2) waiting time: 59ms, multiplied wait time: 118ms
Thread 0(prior: 2) waiting time: 60ms, multiplied wait time: 120ms
Thread 2(prior: 2) waiting time: 60ms, multiplied wait time: 120ms
Thread 1(prior: 2) waiting time: 62ms, multiplied wait time: 124ms
Thread 14(prior: 3) waiting time: 63ms, multiplied wait time: 189ms
Thread 3(prior: 4) waiting time: 83ms, multiplied wait time: 332ms
Thread 5(prior: 4) waiting time: 83ms, multiplied wait time: 332ms
Thread 15(prior: 4) waiting time: 82ms, multiplied wait time: 328ms

Thread 10(prior: 5) waiting time: 98ms, multiplied wait time: 490ms
Thread 12(prior: 5) waiting time: 99ms, multiplied wait time: 495ms
Thread 9(prior: 5) waiting time: 100ms, multiplied wait time: 500ms
Thread 13(prior: 5) waiting time: 105ms, multiplied wait time: 525ms
Thread 11(prior: 5) waiting time: 104ms, multiplied wait time: 520ms
Thread 4(prior: 5) waiting time: 105ms, multiplied wait time: 525ms
Average waiting time per thread is 77ms
Average multiplied waiting time per thread is 303ms

Table 2: Multiplied waiting time in μs

Threads	2	4	6	8	10	12	14	16
CLH	14	28	39	41	50	59	72	82
P-CLH	40	85	113	170	213	237	271	303

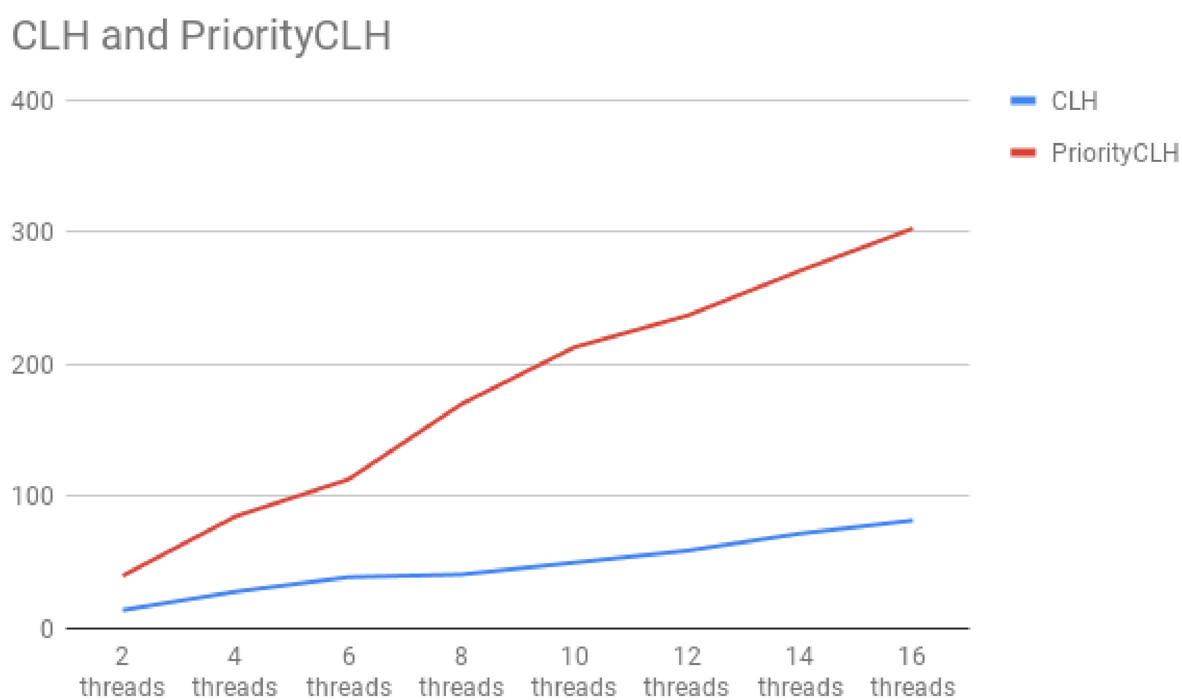

Figure 2: Multiplied waiting time in μs