16-124-836
Marcel ZAUDER

1 CASLock and CCASLock

1.1 Execution with 4 Threads

	CASLock	CCASLock
Counter Value	300'000	300'000
Executiont Time	39ms	27ms
Lowest Number of Accesses	$\sim$ 50'000	$\sim$ 35'000
Highest Number of Accesses	$\sim$ 100'000	$\sim$ 155'000

1.2 Execution with 8 Threads

	CASLock	CCASLock
Counter Value	300'000	300'000
Executiont Time	60ms	51ms
Lowest Number of Accesses	$\sim$ 9'000	$\sim$ 7'000
Highest Number of Accesses	$\sim$ 75'000	$\sim$ 85'000

1.3 Peterson's Algorithm - 8 Threads (*Comparison*)

Counter Value	300'000
Executiont Time	164ms
Lowest Number of Accesses	$\sim$ 32'000
Highest Number of Accesses	$\sim$ 41'000

1.4 Conclusion

In all executions the counter did not exceed the value of 300'000 and all accesses to the critical section sum up to 300'000 as well, which leads to the conclusion that the three different locks work correctly. The differences between the *Peterson's Algorithm* and the newly implemetned *CASLock* and *CCASLock* lies within the fairness which was fairly given in the implementation of the former. The values between the lowest and highest access numbers in the algorithms of *CASLock* and *CCASLock* differ tremendously.
Furthermore we can see that the fairness comes with a huge overhead which leads to a slightly higher execution time for the *Peterson's Algorithm*, whereas the *CASLock* and *CCASLock* are much faster.
When comparing the 4 threads and 8 threads execution of *CASLock* and *CCASLock* we can conclude that the more threads are involved the more overhead and waiting time is created which affects the execution time to be slower for the execution with more threads.

16-124-836
Marcel ZAUDER

2 Unbounded Lock

The necessaty for a *non-empty queue* check in the *deq()*-method follows from the following otherwise occuring problem:

- *queue* has **one** element enqueued
- *Thread B* checks for empty queue → FALSE
- *Thread B* enters critical section
- *Thread A* checks for empty queue → FALSE
- *Thread B* dequeues the last element in the *queue* inside critical section
  ⇒ *queue* is now empty
- *Thread A* enters critical section
- *Thread A* tries to dequeue from an empty queue

3 Queue

A first problem could occur within the *enqueue* method. Because we are always incrementing the integer i, we could exceed the limit of an integer ($2^{31} - 1 = 2'147'483'647$ for a signed integer) and try to increment again the value gets reset to to $-2'147'483'648$. Therefore if we try to enqueue $2^{32} + 1$ times, and do not consider this problem and there is no exception thrown when trying to enqueue at a negative value position, the value at position 0 is overwritten. Therefore the FIFO property is violated because the value at position 0 was enqueued last in this example but is dequeued first. This is but a very unlikely issue which nevertheless must be accounted for.
The much more likely problem of this algorithm is described in the following paragraph (we assume that two values were enqueued before and two threads trying to dequeue concurrently):

Thread A gets the value at position x, sets the value in the list to null and then is put to sleep. *Thread B* gets the value at position $x + 1$ and directly returns it. Then *Thread A* returns its value. Therefore the value at position $x + 1$ is returned before the value at position x. This violates the FIFO property because the value at position x was clearly enqueued before the value at position $x + 1$.