16-124-836
Marcel ZAUDER

1 Peterson's Algorithm

CASE 01

Single Threaded

Counter value: 300'000
lowest number of accesses in critical section for a thread: 300'000
highest number of accesses in critical section for a thread: 300'000
Number of Threads: 1
Execution Time: 15ms

Multi-Threaded

Counter value: 300'000
lowest number of accesses in critical section for a thread: 33'889
highest number of accesses in critical section for a thread: 42'733
Number of Threads: 8
Execution Time: 190ms
SpeedUp: 15ms/190ms = 0,08

CASE 02

Single Threaded

Counter value: 300'000
lowest number of accesses in critical section for a thread: 300'000
highest number of accesses in critical section for a thread: 300'000
Number of Threads: 1
Execution Time: 15ms

Multi-Threaded

Counter value: 300'000
lowest number of accesses in critical section for a thread: 32'111
highest number of accesses in critical section for a thread: 41'052
Number of Threads: 8
Execution Time: 164ms
SpeedUp: 15ms/164ms = 0,09

2 Peterson's Algorithm: Fairness

The Peterson's Algorithm is *starvation free*. That means that every thread that would like to acquire the lock will eventually be able to access the critical section.
The problem for fairness with this problem lies within the *bounded waiting* property which says that when a thread t is waiting to enter a critical section, the maximum number of times any other thread is allowed to enter the critical section before t is bounded by a function of the number of contending threads. This is because the *Filter* algorithm does not consider that a thread can be interrupted i.e. by the scheduler of the OS. Therefore an unbounded amount of threads can access the critical section before the interrupted thread. This behaviour is visible if you run the program with more threads than cores you have (i.e. 16 threads on an octacore machine). The lowest number of accesses to the critical section is 9'947 and the highest number is 27'574, which is clearly showing that the algorithm is not fair in all instances.

3 Linearizability and Sequentially Consistency

3.1 Stack

(OLD) History

C: s.empty()
A: s.push(10)
B: s.pop()
A: s:void
A: s.push(20)
B: s:10
A: s:void
C: s:true

This history is linearizable and therefore as well sequential consistent because C's s.empty() call can finish before any value is pushed towards the stack. Furthermore the s.push(10) operation must happen before the pop() command and before the push(20) operation, because s.pop() will return 10. Therefore we get the following hystory.

(NEW) History

C: s.empty()
C: s:true
A: s.push(10)
A: s:void
B: s.pop()
B: s:10
A: s.push(20)
A: s:void

3.2 Queue

(OLD) History

A: q.enq(x)
B: q.enq(y)
A: q:void
B: q:void
A: q.deq()
C: q.deq()
A: q:y
C: q:y

This history is not linearizable and not sequentially consistent, because it is not possible that only one time y is enqueued but two dequeue operations both get y as return value.