Concurrency:
Multi-core Programming and Data Processing

Assignment 02

Professor: Pascal Felber
Assistant: Isabelly Rocha

March 19, 2020

Submission: File *assignment02.zip* containing Java code and a pdf file with your answers.

1 Peterson's Algorithm

Case 1 Write a program that invokes n threads using Peterson's generalized algorithm in order to protect a shared counter. Each thread increments the counter. In addition, each thread will count its own accesses to the shared counter separately in one field of an array of size n (each thread will have associated one element of the array). Define the shared counter as volatile. The threads will stop when the counter reaches 300000. The program prints the final value of the counter and the number of accesses per thread (the values from the mentioned array). Peterson's algorithm for n threads is described in the lecture's notes (the Filter algorithm).

Case 2 Modify the previous program by making the shared variables non-volatile.

For each of the cases above, run the program sequentially (i.e., with 1 thread) and with t threads, where t is the number of cores in your computer. Report your statistics and the interpretation of the results. The statistics data should include: case, counter value, lowest number of accesses in the critical section for a thread, highest number of accesses in the critical section for a thread, number of threads, the execution time and speedup.

Note: You should not use other Java synchronization mechanisms in your solution besides declaring variables as *atomic types* or *volatile* when/if you think this is needed.

2 Peterson's Algorithm: Fairness

Why is the generalized Peterson algorithm unfair? Try to describe an example that shows the lack of fairness. Do you have any idea of how the algorithm can be enhanced to provide fairness? Give your answer and explain your reasoning.

3 Linearizability and Sequential Consistency

Are the following histories linearizable or sequentially consistent? Explain your answers and write the equivalent linearizable/sequential consistent histories where applicable.

3.1 Stack

We have the following operations:

push(x)	pushes element x on the stack, returns void;
pop()	retrieves an element from the stack;
empty()	returns true if stack is empty and false otherwise.

Concurrent threads A, B, C, stack s.
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

3.2 Queue

We have the following operations:

enq(x)	inserts element x in the queue, returns void;
deq()	retrieves an element from the queue.

Concurrent threads A, B, C, queue q.
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