Operating Systems & Concurrency Assignment-2

Hudson Zhong — hsz1@hw.ac.uk
Kamil Symczak — ks83@hw.ac.uk
Lewis Wilson — lw52@hw.ac.uk
Saad Badshah — sb135@hw.ac.uk
Sam Fay-Hunt — sf52@hw.ac.uk

April 23, 2020

Contents

1 Comparison of Different Methods Used to Achieve Synchronization

1.1 Efficiency Vs. responsiveness

When developing our solutions we have experienced significant differences between the results on a system by system basis. We have selected a single system to run all the tests on, so that their relative performance in that environment can be compared. The system in question has a 4 core 4.5Ghz i7 processor, 16GB of DDR4 RAM and is running the latest version of Windows 10. It is important to emphasize that the tests are not representative of anything beyond the system they ran on.

The table below shows the average time in seconds to complete 5 runs of the tests, for each of the sync classes. A description of each test and the full test results have been provided in the appendix:

Thus a direct ranking based purely on an efficiency perspective results in the following ranking:

1. Intrinsic
2. Semaphore & Atomic
3. Extrinsic

Test Name	Atomic	Intrinsic	Extrinsic	Semaphore
1A	23.364	8.393	47.181	
1B	0.815	0.690	0.719	
2A	0.127	0.124	0.125	
2D	2.040	1.738	1.753	
3A	1.794	2.630	2.624	

1.2 Readability

discussion about Readability (about half a page)

Below is a discussion by the team about the readability of the four classes:

AtomicSync - The team thought overall the code was well formatted and made good use of white space. Members found that conceptually busy waits are easier to follow than some of the other implementations. Members commented on the fact when reading the code, compare and set could be potentially misleading if you have not read and correctly understood the documentation.

IntrinsicSync - The team agreed method names could be improved - "call, put, take". Members found it easy to understand where only a single thread executes code and where concurrency is permitted. Members commented on the finish method stating that it was well commented as well as the logic being easy to understand.

ExtrinsicSync - The team agreed like in IntrinsicSync method names could be improved "put and take". Members also commented on the fact time needed to be spent on familiarising yourself with the .await() and .signalAll() functionally to understand the implementation. Members found the code well formatted and thought that it made good use of comments.

SemaphoreSync - The team thought as a whole that Semaphore Sync was the most complex implementation out of all the classes, with it taking the largest amount of time to familiarise with the implementation. Members commented on the fact that the solution could perhaps be simplified if given more time.

The teams ranking of the classes based on readability and sharing code within by team:

1. AtomicSync
2. IntrinsicSync
3. ExtrinsicSync
4. SemaphoreSync

1.3 Error-proneness

discussion about Error-proneness (about half a page)

A Test description

Test 1A description: A stress test that creates 10,000 threads that run phase one. We wait till all threads terminate (or they timeout via our built in timeout feature) and count the threads that terminated. We then check that the number of threads terminated is equal to the number of threads we expected to terminate (the largest multiple of created threads).

Test 1B description: Same logic as Test 1A but runs multiple tests with a different number of threads using a parameterized input. In the test we have used the following input : 0, 1, 3, 4, 7, 8, 10, 12, 25, 36, 50, 100, 123, 523. This test is predominantly testing the edge cases of the specification.

Test 2A description : We created 10 threads that run phase two. We assign the first half of threads to a group with id 0 and the second half to a group with id 1. We wait till threads in each group terminate and count them. We then expect the number of threads terminated in each group to equal our expected number of threads to be terminated.

Test 2D description: Same as 2A but runs using parameterized testing, with the same range of inputs as in Tests 1B.

Test 3A description: This test method creates 2 groups, and assigns a fixed number of threads from the testing parameters to each group. It then instantiates a thread that will simulate doing some work when it runs (a simple timed busywait 5ms). Finally it performs JUnit assertion operations on a stack containing all the terminated threads. Note that this test does not test the functionality of the finished method, its purely a stability and performance test.

B Readability ranking table