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

1.1.1 Efficiency

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:

Table 1

Test Name	Atomic	Intrinsic	Extrinsic	Semaphore	Result
1A	23.364	8.393	47.181	9.741	Intrinsic
1B	0.815	0.690	0.719	1.146	Intrinsic
2A	0.127	0.124	0.125	0.107	Semaphore
2D	2.040	1.738	1.753	2.796	Intrinsic
3A	1.794	2.630	2.624	2.870	Atomic

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

1. Intrinsic
2. Semaphore & Atomic
3. Extrinsic

1.1.2 Responsiveness

AtomicSync and IntrinsicSync were certainly the easiest to implement, however these, and ExtrinsicSync appeared to really suffer performance issues when trying to synchronize larger numbers of threads, SemaphoreSync on the other hand was quite challenging to get working without errors. Once SemaphoreSync was working, it appeared to perform only slightly worse as the number of threads went up, compared to the other synchronization classes which would get vastly worse performance each time you doubled the number of threads.

Below is the teams ranking of the four different implementations based on responsiveness. The reason these rankings vary from efficiency is due to the fact the team experienced greatly different results from different hardware testing and thus believe this is overall the correct ranking.

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

1.2 Readability

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

Below is a selection of common errors we as a team believe are likely to be encountered when implementing the concurrent classes :

Null pointer exceptions in phase 2 & 3 - Thread groups are stored inside an array where elements are based on their group id, that is for example group 7 is stored as the 7th element inside the array. The area of concern is when a group arrives with an id that is greater than the current length of the array, a deep copy occurs creating a new larger array. A null pointer exception can occur if synchronization is wrongly implemented as if the array is modified by multiple threads simultaneously; two threads could perceive the array to be of a different size.

Infinite loop problems - Very rarely a programmer could encounter the behaviour of an infinite loop problems in atomic sync and semaphore sync where threads where stuck in the waiting state which is most likely caused by cpu scheduling. This is also an issue when testing as you could run a test thousands of times and have it pass every time but once in a blue moon could get stuck in an infinite loop.

Misusing Notify and NotifyAll - Programmers could misunderstand the difference between notify and notifyAll and use notify with the intention to release all threads that were waiting to be released. But since we want to release multiple threads we need to notify all threads thus using NotifyAll.

Confusing wait with await between Intrinsic and Extrinsic respectively - They mostly offer the same functionality but Intrinsic wait can only be related to a single lock instance whereas Extrinsic await can have multiple Conditions with multiple locks.

A Test descriptions

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).

Table 2: Test 1A results in seconds

Test run	Atomic	Intrinsic	Extrinsic	Semaphore
1	12.507	5.238	45.412	9.897
2	22.479	6.870	47.496	9.358
3	13.266	12.399	32.972	9.979
4	45.441	12.650	55.147	9.759
5	23.128	4.810	54.880	9.971
Averages	23.364	8.393	47.181	9.741

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.

Table 3: Test 1B results in seconds

Test run	Atomic	Intrinsic	Extrinsic	Semaphore
1	0.656	0.688	0.693	1.196
2	0.730	0.682	0.695	1.130
3	0.671	0.696	0.708	1.111
4	1.350	0.692	0.797	1.105
5	0.667	0.692	0.696	1.190
Averages	0815	0.690	0.717	1.146

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.

Table 4: Test 2A results in seconds

Test run	Atomic	Intrinsic	Extrinsic	Semaphore
1	0.135	0.122	0.121	0.124
2	0.126	0.126	0.130	0.104
3	0.124	0.123	0.124	0.102
4	0.126	0.126	0.122	0.101
5	0.125	0.122	0.126	0.102
Averages	0.127	0.124	0.125	0.107

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

Table 5: Test 2D results in seconds

Test run	Atomic	Intrinsic	Extrinsic	Semaphore
1	1.771	1.739	1.753	2.815
2	1.970	1.1741	1.752	2.746
3	1.788	1.740	1.751	2.857
4	2.920	1.738	1.758	2.803
5	1.736	1.734	1.750	2.760
Averages	2.037	1.738	1.753	2.796

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.

Table 6: Test 3A results in seconds

Test run	Atomic	Intrinsic	Extrinsic	Semaphore
1	1.785	2.629	2.622	3.130
2	1.774	2.634	2.620	2.927
3	1.914	2.627	2.616	2.843
4	1.768	2.628	2.636	2.794
5	1.729	2.632	2.625	2.775
Averages	1.794	2.630	2.624	2.870

B Readability ranking table

Table 7: Voting results from the ranking of readability

	1st	2nd	3rd	4th
Intrinsic	1	3	0	0
Extrinsic	0	0	3	1
Atomic	3	1	0	0
Semaphore	0	0	1	3