IN4330 Oblig 5 report
Second attempt

Intro

For this assignment we were tasked with making a wait and swap program, by synchronizing threads by only using semaphores. The idea is that the first thread waits, then the second passes through, and then the third one waits. This pattern then repeats itself.

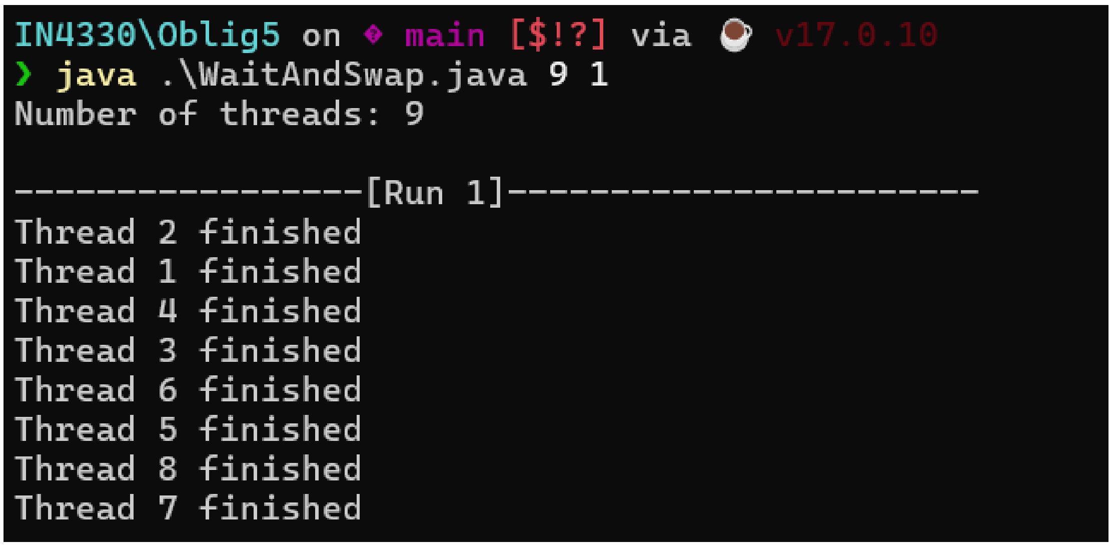
Solution

I again ended up having 2 solutions to solve the problem. The first solution is a bit messy but follows the algorithm. It starts with an enter semaphore so that only 2 threads are working beyond it at once. We then have an odd check that checks the boolean for odd, then flips it and whether the thread is first or not. Once this happens it releases the enter semaphore allowing the second semaphore through. The next set of operations only occurs if the thread number is odd. Odd number thread then attempts to acquire an escape semaphore but it stops. Then if an incoming thread is not first and an odd number waiting, then it releases the escape allowing the previous odd numbered through so it finishes, while doing so it releases the neter semaphore for the next odd number allowing it entry, this next odd numbered thread is responsible for kicking the previous odd numbered out before its entry.

The second solution has also been revised to match the oblig description, it differentiates from the first solution by not using the thread id's to dictate passing or the stopping of threads. It does this by introducing a new semaphore and second bool that controls the behavior of odd threads. Firstly it start off with an outer semaphore that controls the amount of threads working within the procedure at once. We are then met with a first thread check, that releases the entry semaphore allowing a second thread to enter. We then have a semaphore which acquires the permit to change the odd bool which is used for the next set of operations. We then have an if check that checks the flipped bool. If the thread is an odd thread then. If a thread isn't odd then it simply let through. If a thread is odd it it is halted by the a semaphore which is set to (-1) initially. This so that the thread would need to be released twice in order to reach single permit. this way it is only released when another odd thread hits the release for the second time, effectively pushing it out while maintaining a number of threads within the outer semaphore at 2. The second solution the one called waitAndSwap2 ended up being the algorithm that ran the most consistent and I would on that as the proper solution the oblig, if i were to choose one.

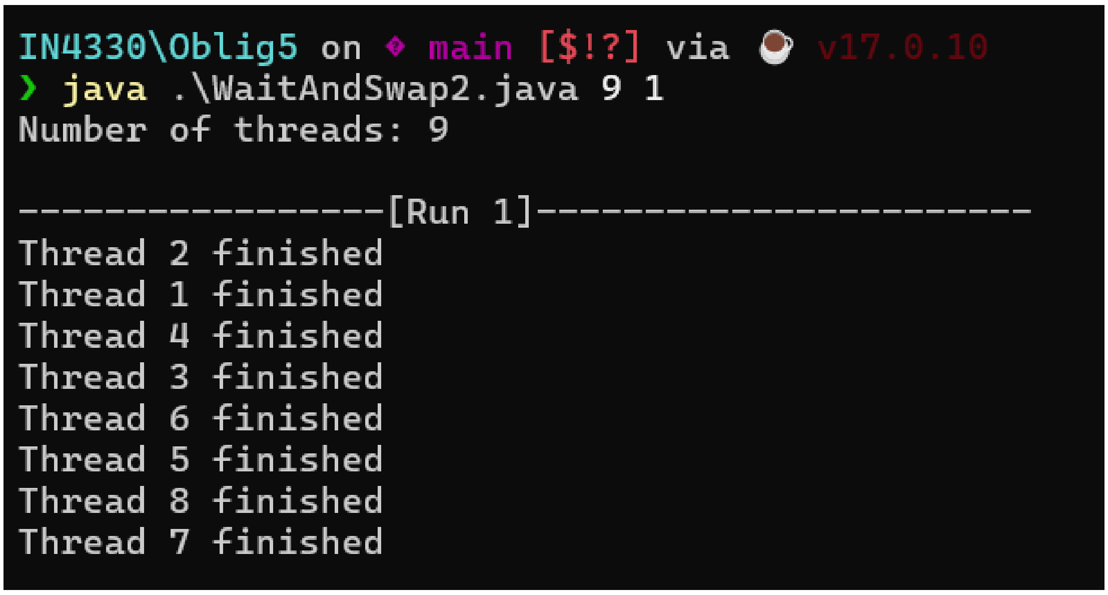
Results

The output for multiple runs of both solutions gives the desired output the oblig described. The program is set to run WaitAndSwap a specified amount of times, to ensure the results are consistent.The current default is set two 3 but is changeable through arguments as seen in the run.



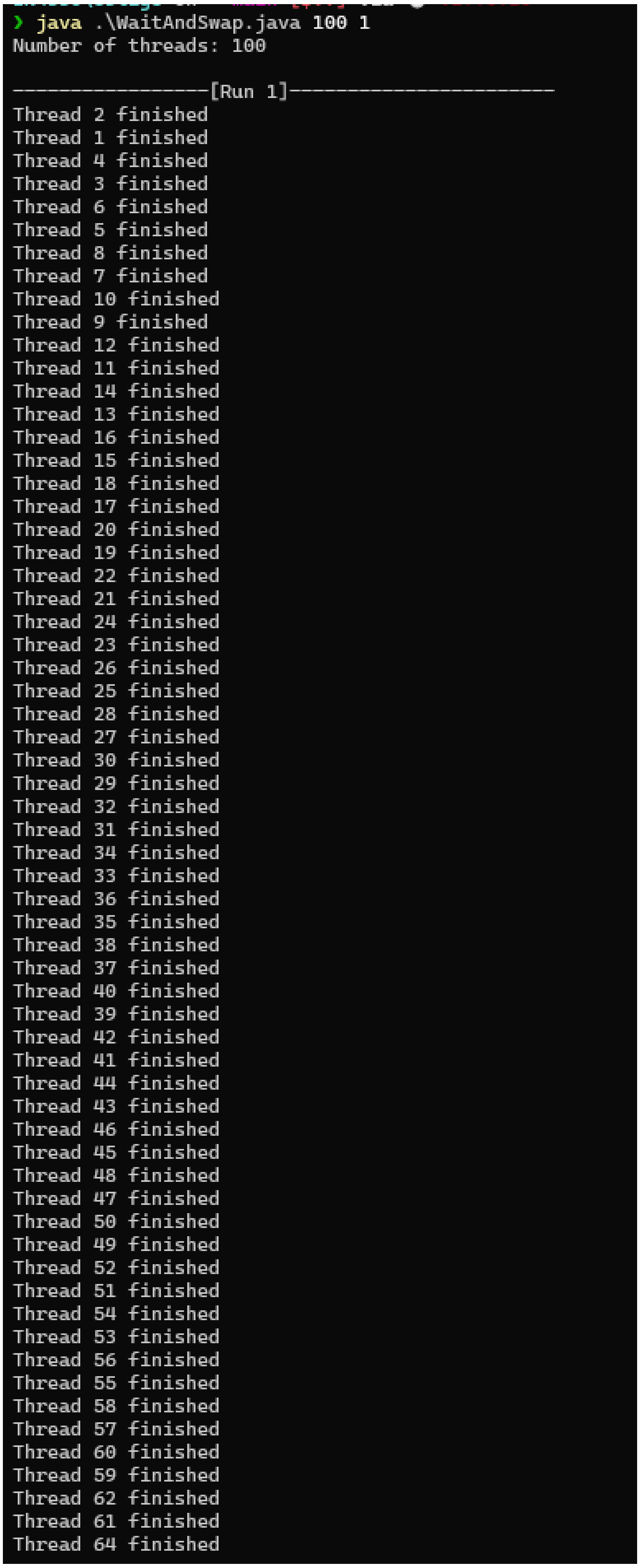
```
IN4330\Oblig5 on ◆ main [$!?] via ☕ v17.0.10
❯ java .\WaitAndSwap.java 9 1
Number of threads: 9

-----------------[Run 1]-----------------------
Thread 2 finished
Thread 1 finished
Thread 4 finished
Thread 3 finished
Thread 6 finished
Thread 5 finished
Thread 8 finished
Thread 7 finished
```



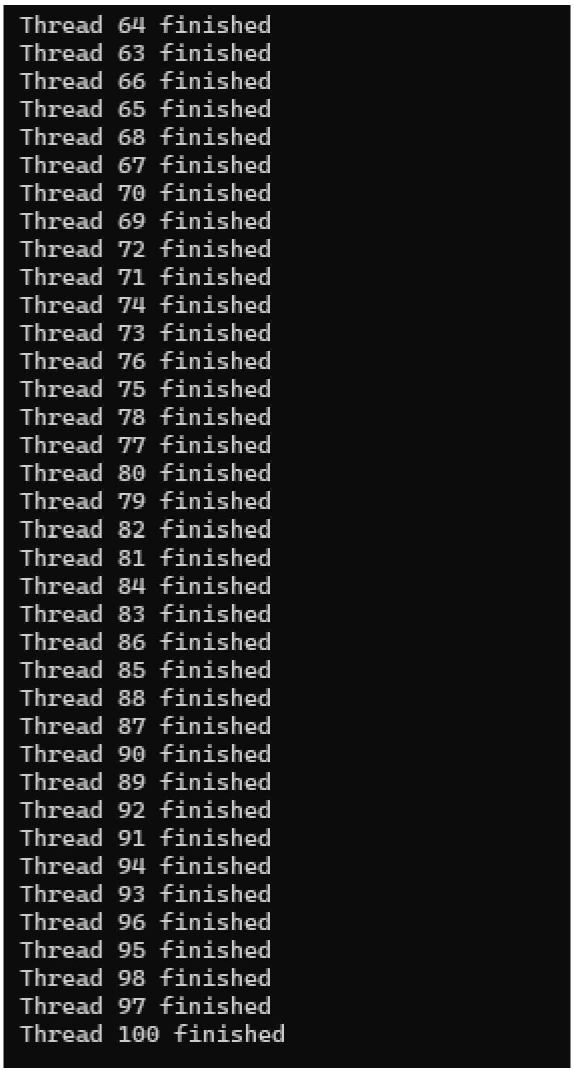
```
IN4330\Oblig5 on ◆ main [$!?] via ☕ v17.0.10
❯ java .\WaitAndSwap2.java 9 1
Number of threads: 9

-----------------[Run 1]-----------------------
Thread 2 finished
Thread 1 finished
Thread 4 finished
Thread 3 finished
Thread 6 finished
Thread 5 finished
Thread 8 finished
Thread 7 finished
```



```
❯ java .\WaitAndSwap.java 100 1
Number of threads: 100

-----------------[Run 1]-----------------------
Thread 2 finished
Thread 1 finished
Thread 4 finished
Thread 3 finished
Thread 6 finished
Thread 5 finished
Thread 8 finished
Thread 7 finished
Thread 10 finished
Thread 9 finished
Thread 12 finished
Thread 11 finished
Thread 14 finished
Thread 13 finished
Thread 16 finished
Thread 15 finished
Thread 18 finished
Thread 17 finished
Thread 20 finished
Thread 19 finished
Thread 22 finished
Thread 21 finished
Thread 24 finished
Thread 23 finished
Thread 26 finished
Thread 25 finished
Thread 28 finished
Thread 27 finished
Thread 30 finished
Thread 29 finished
Thread 32 finished
Thread 31 finished
Thread 34 finished
Thread 33 finished
Thread 36 finished
Thread 35 finished
Thread 38 finished
Thread 37 finished
Thread 40 finished
Thread 39 finished
Thread 42 finished
Thread 41 finished
Thread 44 finished
Thread 43 finished
Thread 46 finished
Thread 45 finished
Thread 48 finished
Thread 47 finished
Thread 50 finished
Thread 49 finished
Thread 52 finished
Thread 51 finished
Thread 54 finished
Thread 53 finished
Thread 56 finished
Thread 55 finished
Thread 58 finished
Thread 57 finished
Thread 60 finished
Thread 59 finished
Thread 62 finished
Thread 61 finished
Thread 64 finished
```



```
Thread 64 finished
Thread 63 finished
Thread 66 finished
Thread 65 finished
Thread 68 finished
Thread 67 finished
Thread 70 finished
Thread 69 finished
Thread 72 finished
Thread 71 finished
Thread 74 finished
Thread 73 finished
Thread 76 finished
Thread 75 finished
Thread 78 finished
Thread 77 finished
Thread 80 finished
Thread 79 finished
Thread 82 finished
Thread 81 finished
Thread 84 finished
Thread 83 finished
Thread 86 finished
Thread 85 finished
Thread 88 finished
Thread 87 finished
Thread 90 finished
Thread 89 finished
Thread 92 finished
Thread 91 finished
Thread 94 finished
Thread 93 finished
Thread 96 finished
Thread 95 finished
Thread 98 finished
Thread 97 finished
Thread 100 finished
```

Run WaitAndSwap with 8 threads, and 3 runs

Conclusion

Through use of semaphores we can synchronize running threads in various ways to control the running of a program that solves a problem in parallel. That being said using them to solve this challenge presented a couple of problems.I found the details of the problem i bit difficult to follow, but i ended up building what i believe is the correct solution given the desired output described in the Oblig text.

Disclaimer about the algorithm. It is built in a way where depending on the if the program is run with an odd number or even number of threads, there's always a thread left hanging. With odd numbers the very last one is stuck, and on even numbers the second to last thread doesn't get released. This made running the algorithm multiple times difficult and simply terminating the hanging threads after joining, had consequences on proceeding runs, breaking the entire program.

I was told in the feedback to run it without sleep, but here there's om issues. There's no controlling which threads enter the procedure in the first place, causing the order of threads completely incorrect. That being said, the number of threads working within the procedure is still only 2 at the time, which I assume is the goal. Furthermore, 2 threads that enter do swap each other, there's just no control which 2 threads enter when there's no sleep.