

Lab 1 - Basic Concurrency in Java

Group 5

Hanna Peters & Iley Alvarez Funcke

1. Creating and joining threads

Source files:

- task1/Main.java (main file)

To compile and execute:

```
javac Main.java
java Main
```

In the Join class, we initialized the threads and started and joined them. When the thread was started, the MyThreadedCode class began running the run method which printed out the “hello world” message, and in the C assignment, also the thread number.

2. Simple Synchronization

2a: Race conditions

Source files:

- task2/MainA.java (main file)

To compile and execute:

```
javac MainA.java

java MainA
```

We expect to get a value that's below 4 000 000 since the threads aren't synchronized and can overwrite each other.

We obtained the value 1596756 when running the program.

2b: Synchronized keyword

Source files:

- task2/MainB.java (main file)

To compile and execute:

```
javac MainB.java  
java MainB
```

We expected to get 4000 000 since we now synchronize the threads

2c: Synchronization performance

Source files:

- task2/MainC.java (main file)

To compile and execute:

```
javac MainC.java  
java MainC
```

How to execute on the PDC computer:

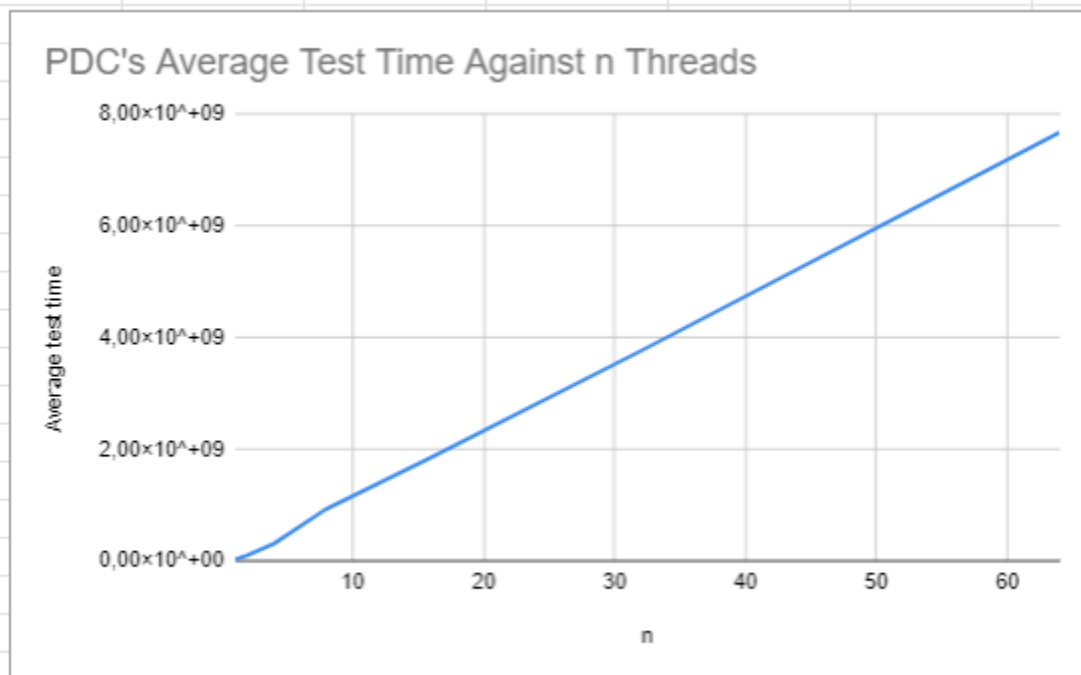
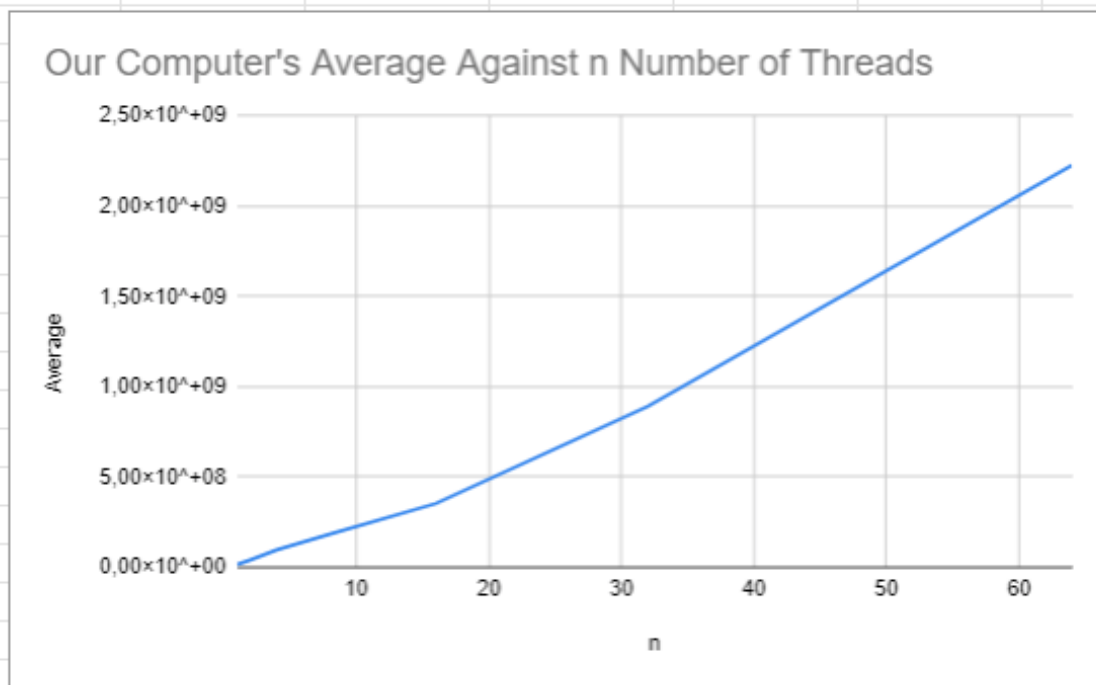
Copy source file from local computer to pdc home folder:

```
scp MainC.java hapeters@dardel.pdc.kth.se:~
```

The same compile and execute commands where used on the PDC

note: execution time increases with the number of threads. For a comparison between our computer's performance and the PDC read further down in the document where the test results are discussed.

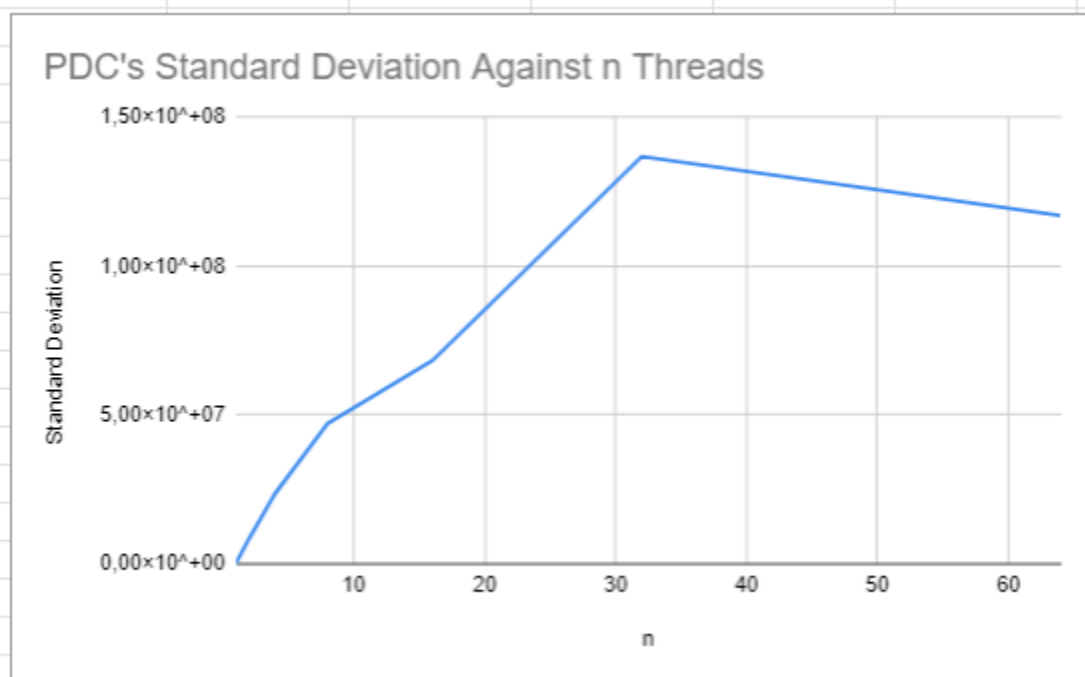
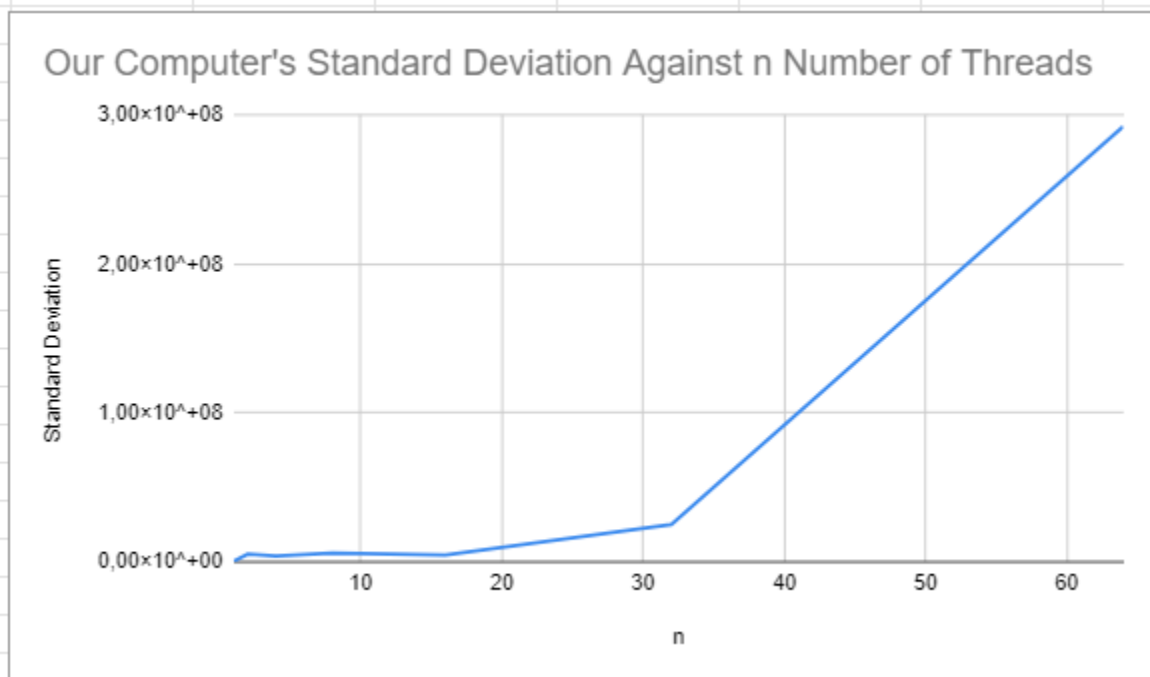
The first findings which can be compared between our own computers performance and the PDC's is the average execution time. The graphs made from the results of these tests can be seen in the 2 figures below:



When looking at the execution time it is clear that the average execution time for our own computer is better for any amount of threads used which was tested by us. We think the reason for this might be that we did not utilize enough of the supercomputer's resources to make any significant improvements compared to what an already good computer would do. On our system

we had an i9 processor which is very good for conventional computers and therefore this may be why our performance was better on this test using our own computer.

When it comes to the average standard deviation of the tests both on our own computers and on the PDC the results for this can be seen in the two figures below:



It is a bit hard to tell in the graph but when looking at the numbers we can see that the standard deviation for 1 thread is lower with the PDC and also with 64 threads and otherwise our own computer performed slightly better in this regard. However the results were very similar and considering this we don't think any solid conclusions can be made.

Discussion of the experiments

The experiment results can be found here:

https://docs.google.com/spreadsheets/d/1p3WflwglFSQcNFE6qU0jezr1tHO_8HJ8KdKp2YmKpn4/edit?usp=sharing

3: Guarded blocks using wait()/notify()

3a asynchronous sender-receiver:

Source files:

- task3/MainA.java (main file)

To compile and execute:

```
javac MainA.java
```

```
java MainA
```

We expected a lower value than 1000 000 since synchronization was not used. We obtained the value 2766.

3b busy-waiting receiver:

Source files:

- task3/MainB.java (main file)

To compile and execute:

```
javac MainB.java
```

```
java MainB
```

3c waiting with guarded block:

Source files:

- task3/MainC.java (main file)

To compile and execute:

```
javac MainC.java
```

```
java MainC
```

3c effects of guarded block on performance:

Source files:

- task3/MainD.java (main file)

To compile and execute:

```
javac MainD.java
```

```
java MainD
```

busy-waiting is faster than guarded blocks

We measured the time it took for the busy-waiting receiver and the guarded block implementation to finish its job. The results can be seen below:

30					
31	3B Tests (Busy Waiting)			3C Tests (Guarded Blocks)	
32	1	8200		1	47700
33	2	4200		2	53100
34	3	6800		3	36000
35	4	10300		4	34800
36	5	4700		5	67000
37	6	3900		6	44800
38	7	6300		7	74900
39	8	3100		8	56600
40	9	11300		9	30300
41	10	11100		10	21200
42	Average	6990		Average	46640
43					

From these results it is pretty clear that the busy-waiting implementation was executed faster, with the average improvement being 39650 nanoseconds which is 0,03965 milliseconds.

4. Producer-Consumer Buffer using Condition Variables

Source files:

- task4/Main.java: main file

- `task4/Buffer.java`: **our producer-consumer buffer**

To compile and execute:

```
javac Buffer.java
```

```
javac Main.java
```

```
java Main
```

We implemented the LIFO consumption. There was no need for synchronization or await signals since we used them in the buffer class.

5: Counting semaphores

We implemented a semaphore by following the instructions for signal and s_wait. We used synchronized together with notify and wait.

Source files:

- `task5/Main.java`: **main file**
- `task5/CountingSemaphore.java`

To compile and execute:

```
javac Main.java
```

```
javac CountingSemaphore.java
```

```
java Main
```

6. Dining Philosophers

6a: deadlocking version

Source files:

- `task6/MainA.java` (**main file**)

To compile and execute:

```
javac MainA.java
```

```
java MainA
```

6b: deadlock and starvation free algorithm

Source files:

- task6/MainB.java (main file)

To compile and execute:

```
javac MainB.java
```

```
java MainB
```

The more philosophers we added the longer it took for the program to deadlock. The reason for this is that the program went into a deadlock state when all left chopsticks had been picked up since then none of the philosophers could pick up a right chopstick.

Deadlock freedom proof by contradiction:

Assume no philosopher can make a move, then each one has to have picked at most one chopstick (the first four has picked the one to their right) and no other stick can be picked by any other. For the last philosopher there are 2 cases:

Case 1:

He has not picked a chopstick. This means that the philosopher next to him must have the chopstick to his left free to pick and since he already has a stick in his hand, he now has two chopsticks and can eat, hence a contradiction to the claim that no one can make a move

Case 2:

He has picked the chopstick to his left. In this case, the chopstick to his right is free since the other philosophers has at most picked one chopstick (the one to their right), meaning he now has two chopsticks and can eat, which also contradicts the claim that no one can make a move. chopsticks

Starvation freedom proof:

If a philosopher is holding his right chopstick waiting for the left to be freed in order to eat, he will get the opportunity to pick the left one up and eat as soon as the philosopher to his left has finished eating and puts down his right chopstick, making the left one available to pick up for him. This means any philosopher waiting for a chopstick to eat will eventually get it, showing that the algorithm is starvation-free.

7

Source files:

- task7/Deadlock.java (main file)

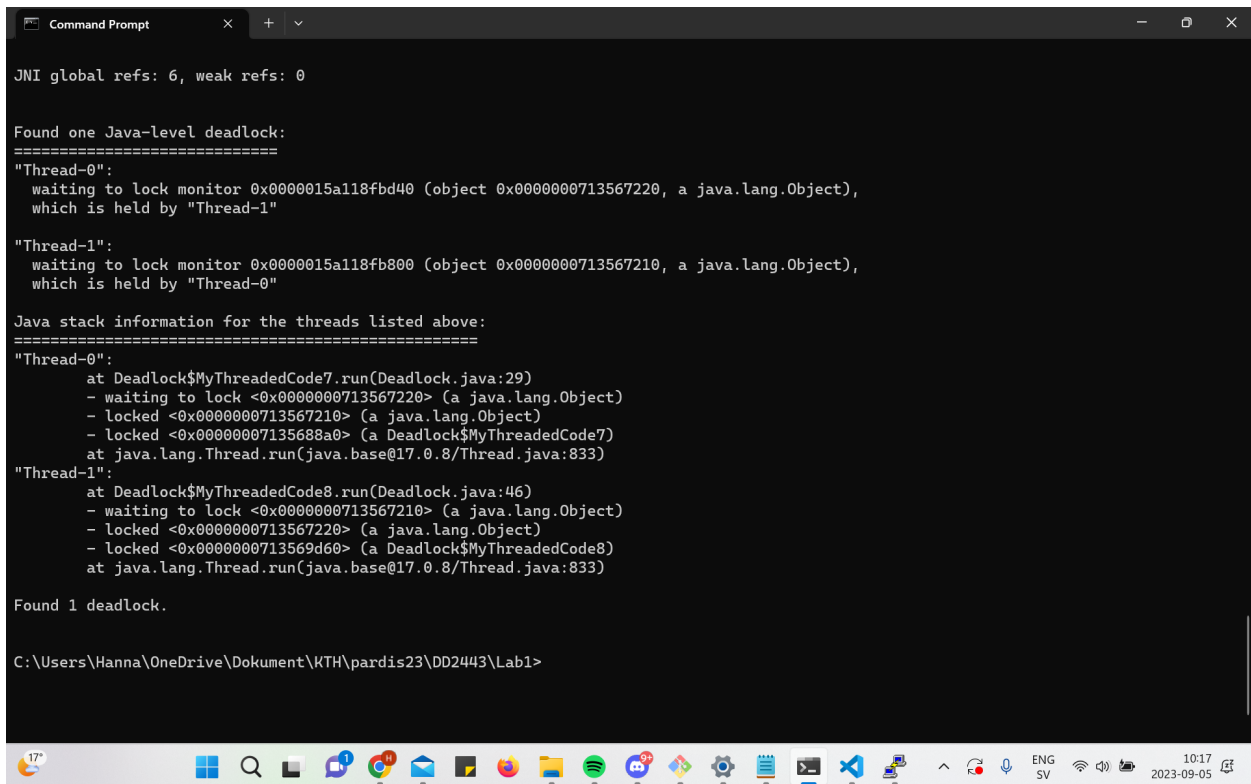
To compile and execute:

```
javac Deadlock.java
```

```
java Deadlock
```

The java.lang.management package has an interface ThreadMXBean that provides findMonitorDeadlockedThreads() and findDeadlockedThreads() methods to find deadlocks in the running application.

It is also possible to look at the thread dump output for analyzing deadlock situations. By using jcmd \$PID Thread.print we can get the thread dump for a program:



```
Command Prompt

JNI global refs: 6, weak refs: 0

Found one Java-level deadlock:
=====
"Thread-0":
  waiting to lock monitor 0x0000015a118fbd40 (object 0x0000000713567220, a java.lang.Object),
  which is held by "Thread-1"

"Thread-1":
  waiting to lock monitor 0x0000015a118fb800 (object 0x0000000713567210, a java.lang.Object),
  which is held by "Thread-0"

Java stack information for the threads listed above:
=====
"Thread-0":
  at Deadlock$MyThreadedCode7.run(Deadlock.java:29)
  - waiting to lock <0x0000000713567220> (a java.lang.Object)
  - locked <0x0000000713567210> (a java.lang.Object)
  - locked <0x00000007135688a0> (a Deadlock$MyThreadedCode7)
  at java.lang.Thread.run(java.base@17.0.8/Thread.java:833)
"Thread-1":
  at Deadlock$MyThreadedCode8.run(Deadlock.java:46)
  - waiting to lock <0x0000000713567210> (a java.lang.Object)
  - locked <0x0000000713567220> (a java.lang.Object)
  - locked <0x0000000713569d60> (a Deadlock$MyThreadedCode8)
  at java.lang.Thread.run(java.base@17.0.8/Thread.java:833)

Found 1 deadlock.

C:\Users\Hanna\OneDrive\Dokument\KTH\pardis23\DD2443\Lab1>
```

Where \$PID for the deadlock program is fetched by running jps in cmd while the program is running