Lab 1 - Basic Concurrency in Java
Parallel and Distributed Computing
DD2443 - Pardis24

Names:
Casper Kristiansson
Nicole Wijkman

Group 14

September 3, 2024

1 Simple Synchronisation

1.1 Race conditions

1.1.1 Information

Source files:

- ‘task1/MainA.java‘ (main file)

To compile and execute:

```
javac MainA.java
java MainA
```

1.1.2 Program Explanation

We began by creating a shared, volatile counter variable that multiple threads could access. Then, we defined a class that implements the `Runnable` interface, where each thread would increment this counter a million times.

In the `main` method, we created and started four threads to run the incrementing task simultaneously. After all threads completed their execution, we printed the final value of the counter.

1.1.3 Discussion

When running this program we can see that we get the output: "`Counter value: 1091182`", note that this value differs every time. This is substantially less than the expected value which should be 4,000,000. The reason for this result can be attributed to the nature of both just using the volatile keyword and the fundamentals of how the different threads will read the value of the shared counter. The volatile keyword ensures that each thread can read the latest value of the shared value but its operation is not atomic. This means it will first read the value, increment the value, and then write. Doing this will mean that a lot of the operations will be ignored and increments operations will be lost. As in the title, this is what we call a race condition.

1.2 Synchronized keyword

1.2.1 Information

Source files:

- 'task1/MainB.java' (main file)

To compile and execute:

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

1.2.2 Program Explanation

We re-used the code from the previous task but modified it by introducing the `synchronized` keyword to the method responsible for incrementing the shared counter. This ensured that the counter was incremented safely by each thread, preventing the race conditions present in the previous task.

In the `main` method, we once again created and started four threads to run the incrementing task concurrently. After all threads completed their execution, we measured the execution time and printed the final counter value.

1.2.3 Discussion

When running this program we can see that we get the output: "`Counter Value: 4000000 Time: 211451375 ns`". By using this strategy with the synchronized keyword we can make sure that the result of the shared counter value to always be 4,000,000. Utilizing the synchronized keyword prevents the race condition from overwriting certain increment operations which we saw happening in the previous task.

1.3 Synchronization performance

Source files:

- 'task1/MainC.java' (main file)

To compile and execute:

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

1.3.1 Program Explanation

In this task, we measured the performance impact of synchronization across varying numbers of threads. We reused the synchronized increment method from earlier and implemented a `run_experiment` method to spawn and join `n` threads, each incrementing the shared counter.

We chose `X = 10` warm-up iterations to allow the JVM to optimize and stabilize before measurements begin, and `Y = 20` measurement iterations to ensure reliable

averages while keeping the total runtime of the program manageable. This setup allowed us to observe the performance implications of increasing thread counts from 1 to 64.

To analyze the performance, we calculated the average execution time and standard deviation across multiple runs to assess both consistency and performance overhead. The standard deviation was computed using:

$$\text{Standard Deviation} = \sqrt{\left(\frac{\sum(\text{time}^2)}{Y}\right) - (\text{Average Time})^2}$$

This allowed us to assess both the consistency and performance overhead introduced by synchronization as the number of threads increased.

We then executed the program on PDC, for the same range of threads, to get a result we could compare with the results of our local machine.

1.3.2 Result

As the number of threads increases, we observe a significant rise in execution time. For instance, the average time with 1 thread is about 2.29 milliseconds, but with 64 threads, it reaches over 532 milliseconds. This is due to the overhead of the `synchronized` keyword, which forces threads to wait their turn to increment the counter.

Threads	**Average Time (ns)**	**Standard Deviation (ns)**
1	2,227,666.7	28,849.36
2	21,996,996.05	877,982.09
4	34,643,375.05	2,115,645.74
8	69,924,668.7	8,190,083.8
16	131,321,616.6	3,794,384.82
32	262,143,381.25	15,229,219.75
64	508,303,397.9	15,938,126.61

Table 1: Local Machine (MacBook Pro M1) Results

Threads	Average Time (ns)	Standard Deviation (ns)
1	2,723,026.15	148,999.88
2	35,170,916.9	1,150,519.76
4	91,405,393.9	12,584,853.82
8	244,587,758.45	50,243,364.12
16	467,120,508.05	117,124,377.22
32	994,722,323.6	146,300,986.62
64	2,249,634,642.4	329,908,076.21

Table 2: PDC Results

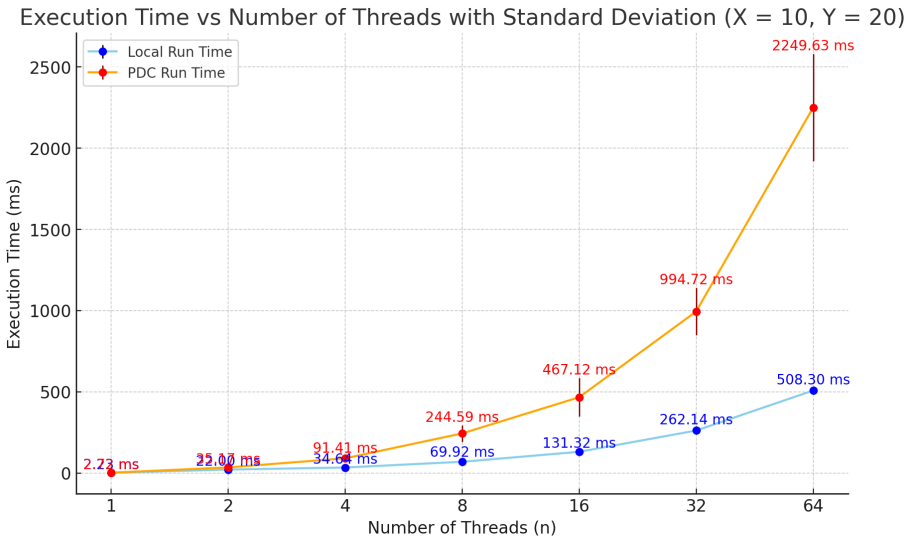

Figure 1: Execution Time vs Number of Threads with Standard Deviation

1.3.3 Difference in Performance

Single-Thread Performance:

The MacBook Pro M1 is slightly faster with 2,227,666.7 ns compared to the PDC's 2,723,026.15 ns. This is expected as a desktop CPU can often have lower latency for single-threaded operations.

Multi-Thread Performance:

As the number of threads increases, the average execution time grows more rapidly on the PDC than on the local MacBook. For example, with 64 threads,

the average time on the PDC is approximately 2.25 billion ns, while it is around 508 million ns on the MacBook.

The PDC shows a significantly larger increase in time and standard deviation with more threads, indicating a higher synchronization overhead.

1.3.4 Discussion

- **System Architecture Differences:** - **MacBook Pro M1:** Uses an ARM-based architecture optimized for single-threaded performance and power efficiency. It has a high-performance cache hierarchy and is designed for lower power consumption, contributing to faster single-thread performance and relatively consistent multi-threaded performance up to a certain number of threads. - **PDC (HPE Cray EX):** A high-performance computing system designed for parallel workloads across many CPU cores. The Cray EX system has more complex interconnects and potentially higher overhead for thread synchronization.

- **Overhead of Synchronization:** - **MacBook Pro M1:** The lower thread count performance suggests lower synchronization overhead. - **PDC (HPE Cray EX):** The increasing overhead, as threads increase, is likely due to the distributed nature of the system.

2 Guarded blocks using wait()/notify()

2.1 Asynchronous sender-receiver

2.1.1 Information

Source files:

- ‘task2/MainA.java‘ (main file)

To compile and execute:

```
javac MainA.java
java MainA
```

2.1.2 Program Explanation

In this task, we implemented a simple message-passing scenario using two threads: an `incrementingThread` and a `printingThread`. The `incrementingThread` is responsible for incrementing a shared integer, `sharedInt`, one million times. The `printingThread` prints the value of `sharedInt`.

We deliberately started the `printingThread` without ensuring that the `incrementingThread` had finished its task. Since there is no synchronization between the threads, the `printingThread` may print a value of `sharedInt` that is neither 0 nor 1,000,000, but some intermediate value. Setting the program up this way demonstrates the lack of synchronization, where the printing operation can occur before the incrementing operation is completed.

2.1.3 Discussion

When running this program we can see that we get some different outputs of the final value of `sharedInt` such as: `7474`, `4252`, and `3204`. The reason for this variation is that the `printingThread` reads and prints the value of `sharedInt` without ensuring that the `incrementingThread` has completed its task. Since there is no synchronization between the threads, the `printingThread` may print an intermediate value of `sharedInt` before the incrementing process finishes. This behavior is a direct result of the race condition between the threads, leading to inconsistent and unpredictable results.

2.2 Busy-waiting receiver

2.2.1 Information

Source files:

- ‘task2/MainB.java‘ (main file)

To compile and execute:

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

2.2.2 Program Explanation

In this task, we modified the program from the previous task to implement a busy-waiting mechanism. We introduced a new shared boolean variable, `done`, which is declared as `volatile` and initialized to `false`. The `incrementingThread` increments `sharedInt` one million times and then sets `done` to `true` once it has completed its task.

The `volatile` keyword ensures that changes to `done` are immediately visible to all threads, preventing the `printingThread` from getting stuck in an infinite loop due to cached values. The `printingThread` continuously checks the value of `done` in a while-loop, busy-waiting until `done` becomes `true`. Once `done` is true, the `printingThread` prints the final value of `sharedInt`. This approach ensures that the final value is only printed after the incrementing is finished, eliminating the race condition observed in the previous version.

2.2.3 Discussion

When running this program we can see that we consistently get the output: “*Final value of sharedInt: 1000000*”. What this indicates is that the busy-waiting mechanism, combined with the `volatile` keyword for the `done` variable, effectively ensures that the `printingThread` waits for the `incrementingThread` to complete its task before printing the value of `sharedInt`. The `volatile` keyword prevents the `done` variable from being cached, ensuring that the loop in `printingThread` accurately detects when `done` becomes `true`. As a result, the final value of `sharedInt` is always the expected 1,000,000, demonstrating that this method successfully eliminates the race condition that was present in the previous task.

2.3 Waiting with guarded block

2.3.1 Information

Source files:

- ‘task2/MainC.java‘ (main file)

To compile and execute:

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

2.3.2 Program Explanation

In this task, we modified the previous program further by replacing the busy-waiting mechanism with a guarded block using `synchronized`, `wait()`, and `notify()`. The `incrementingThread` increments the shared integer `sharedInt` one million times. Once it completes, it enters a synchronized block to set the `done` flag to `true` and calls `notify()` to signal the `printingThread` that the incrementing is complete.

The `printingThread` now uses a synchronized block and a `while` loop to wait for the notification. It calls `wait()` and only proceeds to print the final value of `sharedInt` when it receives the notification from the `incrementingThread`. This approach replaces the inefficient busy-waiting with a more efficient wait-notify mechanism, ensuring that the `printingThread` only proceeds after the incrementing is finished, while also handling potential spurious wake-ups by rechecking the condition in the `while` loop.

2.3.3 Discussion

When running the program for this task we can see that we consistently get the output: "*Final value of sharedInt: 1000000*". This result indicates that using the `synchronized`, `wait()`, and `notify()` mechanism effectively coordinates the execution of the `incrementingThread` and `printingThread`. The `printingThread` waits correctly for the `incrementingThread` to complete its task before printing the value of `sharedInt`. The use of a synchronized block and the `wait()` method ensures that the `printingThread` only proceeds after receiving the notification, eliminating any race conditions and avoiding unnecessary CPU usage caused by busy-waiting. As a result, the final value of `sharedInt` is always the expected 1,000,000, demonstrating that this approach is both efficient and reliable.

2.4 Guarded block on performance

2.4.1 Information

Source files:

- ‘task2/MainD.java‘ (main file)

To compile and execute:

```
javac MainD.java
java MainD
```

2.4.2 Program Explanation

In this task, we compared the performance of a busy-waiting approach against a guarded block approach by modifying the programs from Section 2.2 and Section 2.3. We measured the delay between the completion of the `incrementingThread` and the `printingThread` receiving the notification, using `System.nanoTime()`.

We implemented two methods: `runExperimentsBusyWaiting()` and `runExperimentsGuardedBlock()`. The former method employs a busy-waiting mechanism where the `printingThread` continually checks the `done` flag until the `incrementingThread` completes. The latter uses a guarded block, where the `printingThread` waits for a notification from the `incrementingThread` before proceeding.

To assess the performance, we ran both methods locally, with a warm-up phase followed by multiple measurement iterations. We recorded and compared the delays for both approaches to analyze the impact of guarded blocks on performance.

2.4.3 Discussion

From what we can see after running a test both comparing the average time of running up to 1000 tests is that the difference is really small but most of the time the guarded block has better performance. We generally saw around a 5-10% faster execution time. Using a guarded block is extremely better in terms of the performance of the CPU because we are not constantly checking a flag as in part B. Checking a busy flag in a while loop over and over is extremely CPU heavy which then can affect the actual counting process. However, we did see some outlier tests where the guarded block approach took 5-10% longer to execute.

3 Producer-Consumer Buffer using Condition Variables

3.1 Producer-consumer buffer and Buffer class

3.1.1 Information

Source files:

- ‘task3/Main.java‘ (main file)

To compile and execute:

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

3.1.2 Program Explanation

In this task, we implemented a producer-consumer buffer using Java's `ReentrantLock` and `Condition` variables. The `Buffer` class manages a queue with a fixed capacity N and ensures thread-safe interactions between producer and consumer threads.

The `Buffer` class includes three primary methods:

- `add(int i)`: Allows the producer to insert integers into the buffer. If the buffer is full, the producer waits until space is available.
- `remove()`: Allows the consumer to remove and return integers from the buffer. If the buffer is empty, the consumer waits until an item is available.
- `close()`: Closes the buffer, preventing any further additions but allowing remaining items to be consumed.

The `Producer` thread adds 1,000,000 integers to the buffer, then closes it. The `Consumer` thread continuously removes integers from the buffer and prints them. All of the synchronization is handled within the `Buffer` class, which ensures safe interaction between the producer and consumer without direct synchronization in their respective code.

3.1.3 Discussion

In short, the program segments the generation of numbers between `x` and `x + buffer size`. A consumer then consumes these numbers. If the buffer becomes full, the producer will wait until space becomes available. If the buffer becomes empty, the consumer will wait until new items are added. Both the producer and consumer wake up using the `ReentrantLock` and `Condition` classes, which handle the synchronization efficiently.

4 Counting Semaphore

4.1 Basic Semaphore Counting and Testing Semaphore with Various Counter Values and Thread Counts

4.1.1 Information

Source files:

- ‘task4/Main.java‘ (main file)

To compile and execute:

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

4.1.2 Program Explanation

In this task, we implemented a counting semaphore using Java's `synchronized`, `wait()`, and `notify()` mechanisms. The `CountingSemaphore` class manages a counter that controls access to a shared resource. The semaphore supports two primary operations: `s_wait()` and `signal()`.

- `s_wait()`: Decrements the semaphore counter. If the counter is negative, the thread waits until a resource becomes available.
- `signal()`: Increments the semaphore counter. If the counter is non-positive, it notifies a waiting thread that a resource is now available.

We tested the semaphore with a `Runner` class, where multiple threads attempt to acquire the semaphore, perform a task we have simulated, and then release the semaphore. The main method initializes the semaphore with a count of 3 and starts with 5 threads to observe the semaphore's behavior under concurrent conditions.

4.1.3 Discussion

During this exercise, we identified a potential issue in regard to the exercise instruction: "If the value was negative prior to the increment, wake one thread waiting on the semaphore." The issue with this is that with this structure a deadlock could occur. Specifically, with our current solution, if the count is -1 and a resource releases a semaphore while a thread is still waiting, the waiting thread will be notified. However, when the thread that was sleeping rechecks the condition `count < 0`, it may go back to sleep and wait again, potentially leading to a situation where no progress is made. If the current count is 0 and we increase it, a thread might still be asleep. This means that running the program multiple times with 3 semaphores and 5 threads will sometimes cause a deadlock. It does, however, not seem to get stuck for other inputs.

5 Dining Philosophers

5.1 Model the Dining Philosophers

5.1.1 Information

Source files:

- ‘task5/MainA.java‘ (main file)

To compile and execute:

```
javac MainA.java
java MainA
```

5.1.2 Program Explanation

In this task, we modeled the classic Dining Philosophers problem, where a group of philosophers alternates between thinking and eating. The philosophers sit around a circular table, with one chopstick placed between each pair of adjacent philosophers. To eat, a philosopher must first pick up both chopsticks adjacent to them. However, they can only eat if they have successfully picked up both chopsticks. After eating, they place the chopsticks back on the table and return to thinking.

Our implementation aims to solve this problem using Java's `ReentrantLock` to represent each chopstick. Each philosopher is modeled as a thread that alternates between thinking and attempting to eat. To prevent deadlock, philosophers pick up the left chopstick first, then the right chopstick. This approach, however, can still lead to deadlock if all philosophers pick up their left chopstick simultaneously and then wait for the right one.

In our solution, we assign each philosopher two chopsticks—the left and right—by using modular arithmetic to ensure the last philosopher correctly shares chopsticks with the first philosopher. Each philosopher thread runs in an infinite loop, simulating the process of thinking, attempting to acquire the chopsticks, eating, and then releasing the chopsticks. This setup allows us to observe the deadlocks and test different strategies for preventing them.

5.1.3 Discussion

When testing with five philosophers, we observed that the program could lead to a deadlock, particularly when each philosopher picks up their left chopstick simultaneously and then waits indefinitely for the right one. This deadlock occurs because all philosophers are holding one chopstick, preventing any from acquiring the second chopstick needed to eat.

As the number of philosophers increases, the likelihood and speed of encountering a deadlock rise due to more threads competing for the limited chopsticks. This emphasizes the need for strategies to prevent deadlock, such as implementing resource acquisition hierarchies or introducing random delays before picking up chopsticks. But we also found that when minimizing both the delay for thinking and eating to 1ms we would sometimes with just 5 philosophers reach a deadlock on the first iteration of the code.

5.2 Java Debug

We are using Visual Studio Code to develop our solution, in which we additionally also have the Java extension pack. This allows to debug the code directly in the editor. By pausing the execution when a deadlock occurs, we can inspect the state of each thread to understand the cause of the issue. For example, by pausing the execution in our code we can see that each thread is parked and is trying to acquire the right chopstick. By pausing we also acquire additional information, such as the current stack trace. This allows us to see the progress of the program and the different values of objects. For instance, we can identify specific instances of `ReentrantLock`, labeled as `ReentrantLock@id` (e.g., `ReentrantLock@39`), which allows us to determine which thread currently holds a particular lock.

In addition to the IDE-based debugging, there is the `jstack` tool, which is part of the Java Development Kit (JDK). `jstack` generates a thread dump that shows the current state of threads in a Java process. When a deadlock occurs, `jstack` can help identify the specific threads involved and the resources they are waiting for. This makes it easier to pinpoint the cause of the deadlock. By analyzing the thread dump, we can see which locks are held and which ones are being waited on. This thus provides a clear view of the deadlock situation.

5.3 Solution to the Dining Philosophers problem

5.3.1 Information

Source files:

- ‘task5/MainB.java‘ (main file)

To compile and execute:

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

5.3.2 Program Explanation

In this task, we implemented a solution to the Dining Philosophers problem, focusing on ensuring that our solution is both deadlock-free and starvation-free [1].

To prevent deadlock, we utilized a semaphore to limit the number of philosophers who can attempt to pick up chopsticks simultaneously. More specifically, the semaphore allows `numberOfPhilosophers - 1` philosophers to access the chopsticks at any given time. What this ensures is that at least one philosopher will always be able to pick up both of the chopsticks to eat, which breaks the potential circular wait condition that could lead to a deadlock.

Our solution also prevents starvation and guarantees fairness by utilizing the semaphore's FIFO (First-in, First-out) queue. This will ensure that each philosopher gets a turn to eat, without being indefinitely blocked by others. The result is that no philosopher will be forced to wait infinitely long, and this ensures that all philosophers eventually get to eat. This makes the solution both deadlock-free and starvation-free.

5.3.3 Discussion

The reason why this solution is effective is because of the structure of using semaphores. By allowing `n-1` semaphores to be acquired there will always be at least one that can eat with both chopsticks essential to dealing with the issue that all left chopsticks will be picked up. Secondly, the semaphore allows for fairness because it follows the FIFO queue system which means that no thread will end up starving.

References

[1] *8.5. dining philosophers problem and deadlock — computer systems fundamentals*, https://w3.cs.jmu.edu/kirkpams/OpenCSF/Books/csf/html/DiningPhil.html, (Accessed on 09/01/2024).