
INTERNATIONAL
UNIVERSITY OF SARAJEVO

International University of Sarajevo
Faculty of Engineering and Natural Sciences

**2022 – 2023**
**Academic Year**

# Report

## CS307 Operating Systems

**06.01.2023.**

**Students**: Nur Rustempašić 210302078,
Nedžla Šehović 210302011,
Nejira Subašić 210302167

# Introduction

When it comes to operating system concurrent programming, the Master/Slave paradigm is a potent way to divide up computing work amongst several entities. In order to compute prime numbers in an efficient manner, this project will investigate two different approaches: forking children and using threads.

The formation of n slaves, each distinguished by a distinct slave_id range from 0 to n-1, is started by the Master process. Together, these enslaved individuals compute prime numbers; each is in charge of a particular range. The calculation starts at 3 + 2 * slave_id and increases by 2 * n_slaves until it reaches max_prime. In this case, n_slaves indicates the number of slaves that are running concurrently, while max_prime is the highest limit for finding prime integers. Forking children or employing threads, both approaches require the slaves to give feedback in the form of a count of the prime numbers they have found and the associated running time.

Using threads introduces another layer of complexity since each thread's prime numbers must be saved in a global array. This calls for modifying the signature of the calculate_primes method and adding mechanisms to attach the computed prime number to the global array in order to adapt it to the threads case. The threading method of choice is POSIX threads, as demonstrated by the example supplied on Teams (test_thread_with_struct_arg). Moreover, the software is engineered to furnish comprehensive discernments regarding the execution duration of every child or thread. Together with the total amount of time the main program took, the output shows the amount of time (in milliseconds) that each child or thread spent. Below is an example fragment of the output:

The results should be similar if you use the threads technique and substitute "Thread" for "Process." This project explores concurrent programming and offers a solid framework for using the Master/Slave model to parallelize prime number computation.

# Test plan

i. **Non- parallel: process only**

We ran a number of test cases to test the software in a non-parallel environment with an emphasis solely on processes. Verifying the program's accuracy and functionality using just one step was the main goal.

**Test case**

Execution of a single process:

- Input: max_prime = 1000000, n_slaves = 1.
- Method: “process”
- Anticipated Outcomes: Verify that the software precisely computes every prime number less than 10,000000 by a solitary procedure. Check if the result is valid, and examine how long it took to execute.
- Issues: No significant problems were observed during testing. But it was imperative to make sure the program handled the single-process situation correctly and generated reliable results. Verifying the accuracy of the prime number computations and the general behavior of the software was the major goal.

ii. Parallel: children and threads using:

- **max_prime = 1000000**
- **n_slaves: 2, 4, 8, 16, 32, 64**

- In order to evaluate how well the program performed when running in parallel with children and threads, I created test cases with different values for n_slaves. Evaluating the program's scalability in parallel execution was the aim.

**Test cases**

1. **Parallel Execution with Children (Processes)**
   -Input: max_prime = 1000000, n_slaves = 2, 4, 8, 16, 32, 64
   -Method: "process"
   -Anticipated Outcomes: Verify precise computations of prime numbers for every process count. Make that there is a decrease in the total execution time as compared to the non-parallel scenario by evaluating the execution time.
2. **Parallel Execution with Threads**
   -Input: max_prime = 1000000, n_slaves = 2, 4, 8, 16, 32, 64
   -Method: "thread"
   -Anticipated Outcomes: Confirm accurate computation of prime numbers for every thread count. Analyze the execution time to confirm that it is shorter than in the non-parallel scenario overall.

**Problems and Solutions:**

- Controlling thread synchronization to prevent data races and inconsistent results.
- Ensuring that threads and processes end properly to prevent resource leaks.
- Keeping an eye out for and dealing with any deadlocks that may arise during concurrent execution.

**Overall Conclusions**

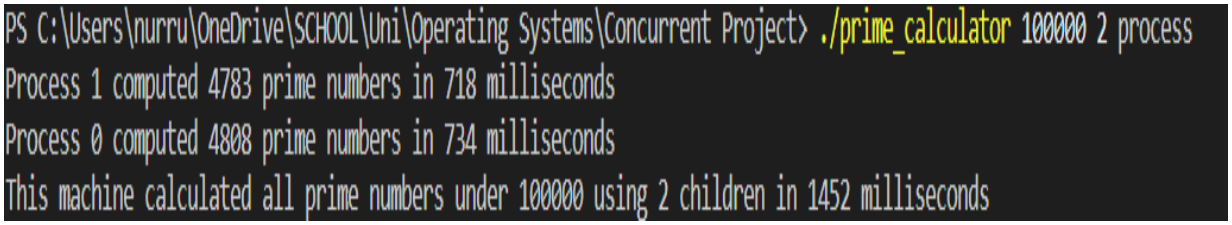
**Testing approach**: Functional and performance testing were both a part of the testing strategy. Performance testing evaluated the program's scalability and efficiency across various setups, while functional testing confirmed that prime number computations were accurate.
**Tools and Methodologies**: I measured execution times using timing techniques and tested functional features manually. Because of the unique needs for simultaneous execution in the program, no automated testing methods were used.
**Requirements and Constraints:** The main prerequisite was to guarantee precise computation of prime numbers in various parallel setups. Preventing data races, controlling synchronization, and handling possible deadlock scenarios were among the constraints.
**Additional Tests:** In addition to the configurations that were provided, I ran more tests to assess how the program behaved in harsh scenarios, including with extremely high max_prime values. This assisted in determining possible performance bottlenecks and evaluating the resilience of the application.

**Performance and Efficiency Insights:** The application showed scalability in parallel execution by becoming more efficient as the number of processes or threads increased. Execution times and other performance measures matched the anticipated advantages of parallel processing.

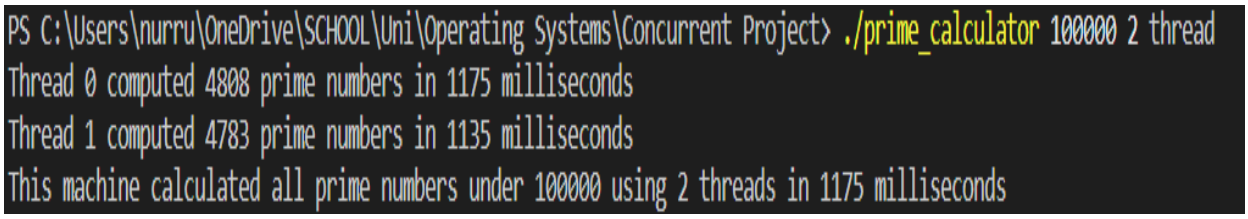
To sum up, the testing strategy took into account the challenges of parallel execution, performance measurement, and functional validation. This led to a thorough evaluation of the program's capabilities in both non-parallel and parallel configurations.

# Test results

The snapshot below shows the console output during the program's two child processes' execution, demonstrating the parallel computation of prime numbers and pertinent performance indicators.




On the other hand, this screenshot shows the application running in two threads and provides a visual depiction of the parallel thread-based prime number computation process along with related performance information.

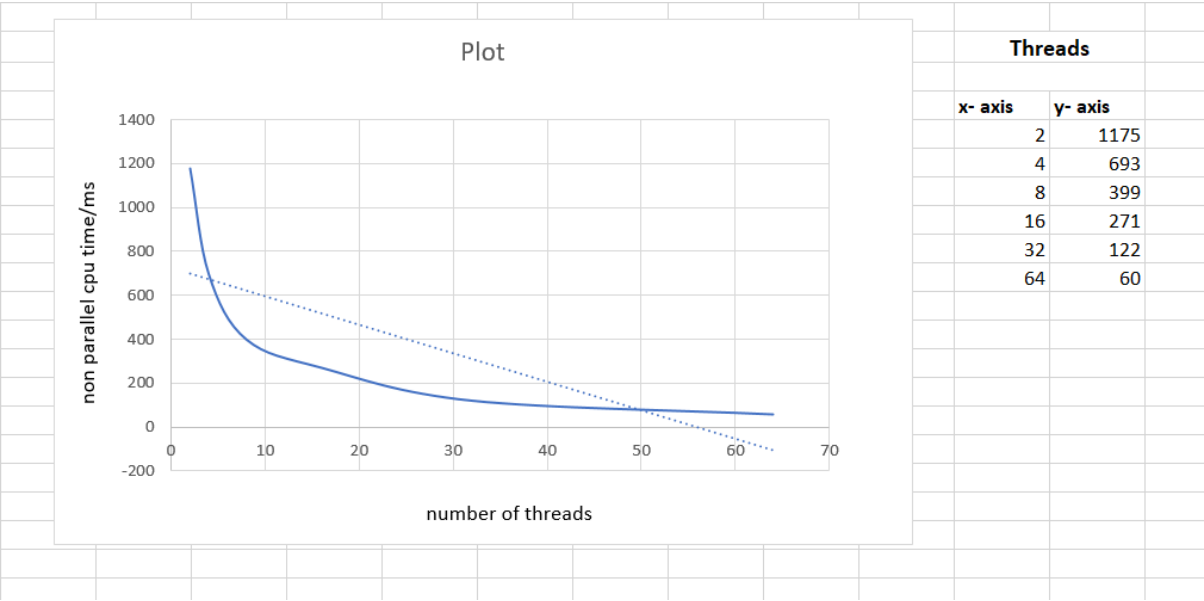


# Plot


| Children | |
|---|---|
| x- axis | y- axis |
| 2 | 1436 |
| 4 | 1545 |
| 8 | 2934 |
| 16 | 3005 |
| 32 | 3399 |
| 64 | 1830 |

The data is plotted to show the relationship between the corresponding CPU time ratios (y-axis) and the number of children (x-axis). Plotting 2 to 4 children in the first section shows a slow rise in CPU time from 1436 to 1545, indicating a reasonably controllable processing cost. But when the number of offspring hits 8, a more significant upturn shows up, with a CPU time of 2934. The successive increments carry on this fast growth, peaking at 3399 with 32 children. The figure then noticeably shows a noticeable decline, indicating a substantial decrease in CPU time to 1830 with 64 children.

The patterns that have been noticed point to a complex link between computational efficiency and parallelism. With a moderate number of offspring, the early advantages of parallel processing are evident; however, after a given amount of time, the related overhead increases and may have an effect on overall performance. The ensuing decrease in CPU time with 64 children might point to a range where child process management is most effective, highlighting the significance of deliberate parallelization for effective execution in multi-process contexts.




| Threads | |
|---|---|
| x- axis | y- axis |
| 2 | 1175 |
| 4 | 693 |
| 8 | 399 |
| 16 | 271 |
| 32 | 122 |
| 64 | 60 |

The relationship between the number of threads (x-axis) and the corresponding ratio of non-parallel CPU time (y-axis) is depicted in the plotted data. As the number of threads rises, the graphic shows a continuous pattern of decreasing CPU time. The non-parallel CPU time with two threads is 1175; with four threads, this number is significantly reduced to 693. The non-parallel CPU time drops dramatically when the number of threads rises to 8, 16, and 32, hitting 399, 271, and 122, respectively. The trend is still decreasing, and it peaked at 60 with 64 threads.

This pattern indicates that decreased non-parallel CPU times are a consistent outcome of the introduction of parallel threads, which improve computational efficiency. Plotting the possible gains from concurrent execution and effective thread management, the inverse relationship between the number of threads and non-parallel CPU time highlights the advantages of parallelization in improving overall performance.

## Conclusion

**Regarding threads:**

Positive scaling (faster with more threads): Generally speaking, thread creation and management costs are less than those of processes.

Because threads share the same address space, data sharing and communication are more effective. There's a chance that the task is best executed in parallel, and using multiple threads makes the system more efficient with its resources.

**Regarding processes:**

**More Processes Mean More Overhead:** Compared to threads, creating and maintaining processes has additional overhead. Since every process has its own address space, communication may be less effective and memory usage may increase. It can be more costly to alter contexts between processes than between threads.

**Features of the System:** The system's unique properties, such as the total workload and the number of cores available, may also have an impact on the behavior.

**Balance of loads:** Another factor that could affect performance is how the burden is distributed among threads or processes. Performance can be below par if there is an uneven distribution.

In conclusion, the task's nature, the degree to which the parallelization method matches the problem, and system-specific parameters all have an impact on the performance characteristics. The ideal number of threads or processes might frequently change depending on the particular circumstances.