The German University in Cairo (GUC)
Faculty of Media Engineering and Technology
Computer Science and Engineering
Operating Systems - CSEN 602

Multithreading - C

Team Members :
Yassin Mohamed, 58-16061

Omar Tarek, 58-10021

Omar Gamaleldin, 58-13583

Omar Magdy, 58-17147

Youssef Mohamed, 58-18247

Abdelrahman Wael, 58-10142

Team Number :
Team #60

Under Supervision of :
Dr. Eng. Catherine M. Elias

Spring 2025

Contents

Chapter 1

Project Objectives

1.1 Main Objective

The main objective of this milestone is to build the foundations of the upcoming implementations. This includes the development of :

- Different Scheduling Algorithms
- Creating and Executing Threads

The performance of the schedulers is also of interest. This report will show the performance metrics of different schedulers which consist of:

- Execution Time
- Release Time
- Start Time
- Finish Time
- Waiting Time
- Response Time
- Turnaround time
- CPU Useful Work
- CPU Utilization
- Memory Consumption

Chapter 2

Introduction

2.1 Multithreading

Multithreading is a fundamental concept in modern operating systems and software development, allowing multiple sequences of programmed instructions, known as threads, to execute concurrently within a single process. By enabling concurrent execution, multithreading improves the efficiency and responsiveness of applications. In the C programming language, multithreading is commonly implemented using the POSIX Threads (Pthreads) library, which provides an API for creating and managing threads.

2.2 Role of OS in Multithreading

Operating systems play a critical role in managing the execution of these threads through scheduling algorithms. Thread scheduling determines the order and allocation of CPU time to various threads, ensuring efficient utilization of processor resources while maintaining fairness and responsiveness. Different scheduling algorithms are designed to optimize specific performance criteria, such as minimizing waiting time, maximizing throughput, or achieving real-time responsiveness. Common scheduling algorithms include First-Come-First-Served (FCFS) and Round Robin (RR).

2.3 Aim

This report explores the use of multithreading in C alongside two scheduling algorithms, with the usage of the Pthreads library, and examines thread scheduling algorithms used in operating systems. It aims to provide a comprehensive understanding of how threads are created, synchronized, and scheduled, highlighting their significance in achieving concurrency in modern computing systems.

Chapter 3

Methodology

This section outlines the approach taken to implement and analyze multithreading concepts and thread scheduling algorithms in C. Two primary components: the practical implementation of multithreaded programs using the POSIX Threads (Pthreads) library and the measuring of each thread's performance metrics.

3.1 Multithreading Implementation Using Pthreads

The core of the multithreaded implementation is based on the POSIX Threads (Pthreads) library. Pthreads provides a standard set of APIs for creating, managing, and scheduling threads in C. In this project, threads were created using the `pthread_create()` function, which allows multiple threads to run concurrently within a single process.

Multiple thread functions were designed to simulate different workloads and behaviors. The threads executed mathematical and I/O operations, allowing for the analysis of thread performance under different scheduling scenarios.

3.2 Scheduling Algorithm

Two thread scheduling algorithms were simulated to observe their impact on thread performance and system responsiveness. Although the Pthreads library relies on the operating system's scheduler, the project used predefined scheduling policies such as First-Come-First-Served (FCFS), Round Robin (RR).

Threads were scheduled based on algorithm-specific rules, and their execution metrics were collected using a custom made thread metrics struct (`ThreadMetric.h`). The impact of each scheduling algorithm was evaluated by comparing performance metrics such as waiting time, turnaround time, and response time for each thread.

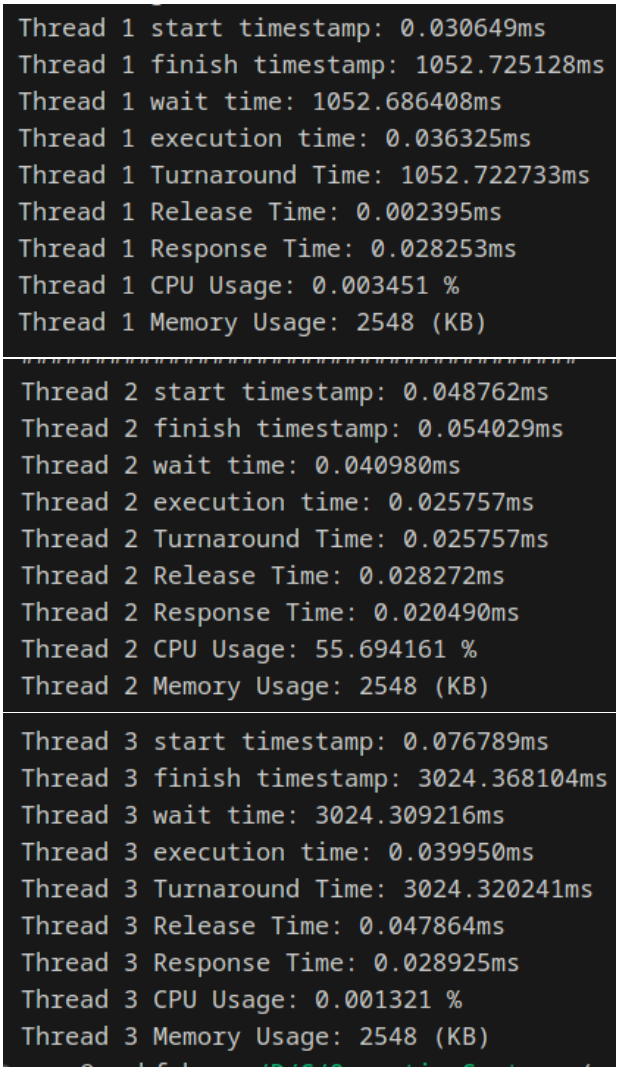
Chapter 4

Results

4.1 Scheduling Algorithms

The following images show the performance metrics for each scheduling algorithm used.

4.1.1 Round Robin



4.1.2 First Come First Serve

```
Thread 1 start timestamp: 0.037969ms
Thread 1 finish timestamp: 1004.463170ms
Thread 1 wait time: 1004.417793ms
Thread 1 execution time: 0.042720ms
Thread 1 Turnaround Time: 1004.460512ms
Thread 1 Release Time: 0.002658ms
Thread 1 Response Time: 0.035312ms
Thread 1 CPU Usage: 0.004253 %
Thread 1 Memory Usage: 2532 (KB)
```

```
Thread 2 start timestamp: 0.063604ms
Thread 2 finish timestamp: 0.068626ms
Thread 2 wait time: 0.058731ms
Thread 2 execution time: 0.034388ms
Thread 2 Turnaround Time: 0.034388ms
Thread 2 Release Time: 0.034238ms
Thread 2 Response Time: 0.029365ms
Thread 2 CPU Usage: 53.938711 %
Thread 2 Memory Usage: 2532 (KB)
```

```
Thread 3 start timestamp: 0.087118ms
Thread 3 finish timestamp: 3984.169062ms
Thread 3 wait time: 3984.104507ms
Thread 3 execution time: 0.038262ms
Thread 3 Turnaround Time: 3984.112356ms
Thread 3 Release Time: 0.056706ms
Thread 3 Response Time: 0.030413ms
Thread 3 CPU Usage: 0.000960 %
Thread 3 Memory Usage: 2536 (KB)
```

4.1.3 Results Analysis

Due to the nature of the functionss each thread was calling, certain differences between the performance of each scheduling algorithm were difficult to notice. This was due requiring the user to input characters and numbers which would increase the wait time of the threads extensively. However, one performance metric could still show the difference between the nature of the scheduling algorithms: the response time.

The response times for **First Come First Serve** were on average greater than those for **Round Robin**. **Round Robin** Thread 2 had a response time of *0.020ms*, the minimum among all thread executions. The maximum response time in **Round Robin** threads was around *0.28ms*, while the response times in **First Come First Serve** reached up to *0.035ms*. This metric goes hand in hand with the start time: A smaller response time usually implies and earlier start to the thread.

This difference in response times is as expected, since **First Come First Serve** is a ***non-preemptive*** scheduling algorithm, meaning other threads must wait for previous threads to be completed or blocked in order to start. **Round Robin** on the other hand is ***preemptive***, allowing each thread to get an initial time-slice/quanta and start execution earlier.

Furthermore, we can see a difference in **CPU Usage** when comparing Thread 2 with the

other threads (53-55% compared to 0.001-0.004%). This is due to the fact that thread 2 did not require input, decreasing wait time and making the majority of thread 2 execution time.

When it comes to the other metrics, the user input greatly increases the **wait time** of each thread. This is due to the fact that the time taken for the user to enter the start/end integer/character takes significantly more time than the CPU executes, which leads to very small CPU Usage and very exaggerated **wait times** and late **finish timestamps**.

The total **execution time** for the three threads does not change significantly from one scheduling algorithm to another due to the fact that we are executing the same functions and the same amount of time is required.

All threads had the same **memory usage**, which was the *maximum* amount of memory used at any time during the execution.

Chapter 5

Conclusion

In this report, we explored the implementation and evaluation of multithreading concepts in the C programming language using the `pthreads` library. The project demonstrated the creation and usage of multiple threads to simulate concurrent execution.

Additionally, various scheduling algorithms, including **First-Come-First-Serve** (FCFS) and **Round Robin** (RR), were implemented and tested. Each algorithm was evaluated based on performance metrics such as turnaround time, waiting time, and CPU utilization.

Overall, the project highlighted the differences between the two scheduling algorithms in terms of these performance metrics. The results were consistent with the theory behind these algorithms.

For further analysis, more complex scheduling algorithms can be used, and a larger variety of thread functions can be implemented in order to avoid the anomalies caused by requiring user input.