Higher Institute of Computer Science
University of Tunis El Manar

Software Engineering Department

OPERATING SYSTEM MINI-PROJECT

Design and Development of a Process Scheduler Simulator on Linux

By

Ahmed Takieddine Ghrib
Moataz Khabbouchi
Nessim Zemzem
Yassine Miladi

Academic Supervisor : **Yousra Najar**

Academic Year 2023/2024

Table of Content

Sprint 3 - Multi-Level

General Conclusion

Preliminary Studies

1. Subject Overview

This project proposes the development of a comprehensive system for simulating and evaluating multitasking scheduling algorithms in a Linux environment. The project aims to provide a flexible and extensible simulation tool for researchers and computer science students to explore the performance of various scheduling algorithms under controlled and varied conditions.

This project aims to:

1. Generate Random Tasks: The system will dynamically generate tasks with diverse characteristics, including varying execution times, priorities, and arrival intervals. This will allow for realistic simulations that capture the complexities of real-world workloads.

2. Implement Multitasking Scheduling Algorithms: Core functionalities will include the implementation of popular scheduling algorithms like First-Come-First-Served (FCFS), Round-Robin (RR), and Shortest Job First (SJF). Additionally, the system will be modular to enable the easy integration and comparison of other algorithms.

3. Configurable Simulation Parameters: The system will allow users to configure various simulation parameters such as the number of tasks, arrival intervals, scheduling algorithm parameters, and specific task properties. This will enable customized evaluations tailored to specific research questions or learning objectives.

4. Data Collection and Analysis: The system will automatically collect data on task execution and algorithm performance metrics like average waiting time, turnaround time, CPU utilization, and throughput. This data will be stored for later analysis and comparison across different algorithms and configurations.

5. Visualization and Reporting: The system will provide an intuitive interface for visualizing the simulation results through graphs and statistics.

The expected outcomes of this project are as follows:

Fully Functional Simulation Environment: The project aims to deliver a user-friendly and robust simulation environment implemented in C. This environment will enable users to easily conduct experiments and compare the performance of different scheduling algorithms.

Comprehensive Performance Evaluations: A series of experiments will be conducted to evaluate the performance of various scheduling algorithms under diverse workload scenarios. This will provide valuable insights into the strengths and weaknesses of each algorithm and contribute to the understanding of scheduling algorithms in the context of Linux operating systems.

Report and Recommendations: A comprehensive report will be generated documenting the project methodology, experimental results, and analysis. The report will also provide recommendations for future research directions and potential improvements to the developed simulation environment.

2. Project Framework

This mini-project forms a crucial component of the Operating Systems module in the Computer Science Engineering program at the Higher Institute of Computer Science Ariana. It goes beyond theoretical understanding and delves into the practical realm of designing and implementing a simulation environment for analyzing the performance of multitasking scheduling algorithms.

By successfully completing this mini-project, students will gain significant insights into the intricacies of multitasking scheduling, develop valuable technical skills, and enhance their problem-solving abilities. The project will contribute to:

Improved understanding of scheduling algorithms: We will gain a deeper understanding of how different algorithms work and their performance characteristics under varying workloads.

Enhanced learning and engagement: The project will provide an engaging and interactive learning experience, complementing the theoretical aspects of the Operating Systems module.

Development of valuable software engineering skills: We will acquire and hone their skills in C programming, data analysis, and system design.

Potential for future research: The project's modular design and open-ended nature encourage further exploration and experimentation, paving the way for potential future research contributions.

Chapter I : Need's Specifications

1. Scrum Management

1.1. Scrum Team

In the fast-paced world of software development, efficiency and adaptability are crucial. Scrum, a popular agile framework, relies on small, cross-functional teams called Scrum Teams to deliver projects quickly and effectively.

A Scrum Team typically consists of three key roles:

Product Owner: The voice of the customer, responsible for defining the product vision, prioritizing features, and ensuring the team builds the right product.

Scrum Master: Facilitates the Scrum process, removes roadblocks, and ensures the team adheres to Scrum principles and practices.

Development Team: A group of self-organizing individuals with diverse skills and expertise necessary to build the product increments.

Scrum Teams are characterized by their cross-functionality, allowing them to work independently without external reliance. They are self-organizing, empowered to determine the best approach to tasks within sprint cycles. Transparency reigns supreme, with all team members having access to project information and progress updates. Finally, decisions are based on data and experimentation, adhering to an empirical approach.

For this project we opted for the following team composition:

Product Owner	Scrum Master	Development Team
- Mrs Yousra Najar	- Mrs Yousra Najjar	- Ahmed Takieddine Ghrib - Moataz Khabbouchi - Nessim Zemzem - Yassine Miladi

This team composition allows us to achieve increased agility by adapting quickly to changing requirements and market trends. The cross-functional nature of the team fostered close collaboration, leading to improved quality through continuous feedback loops and iterative delivery. This resulted in a faster time to market, delivering working software increments regularly. Additionally, team members felt a strong sense of ownership and accountability, further driving motivation and project success.

1.2. Use case Diagram

After exploring the different requirements presented in the specifications file, here is the use case diagram we opted for:

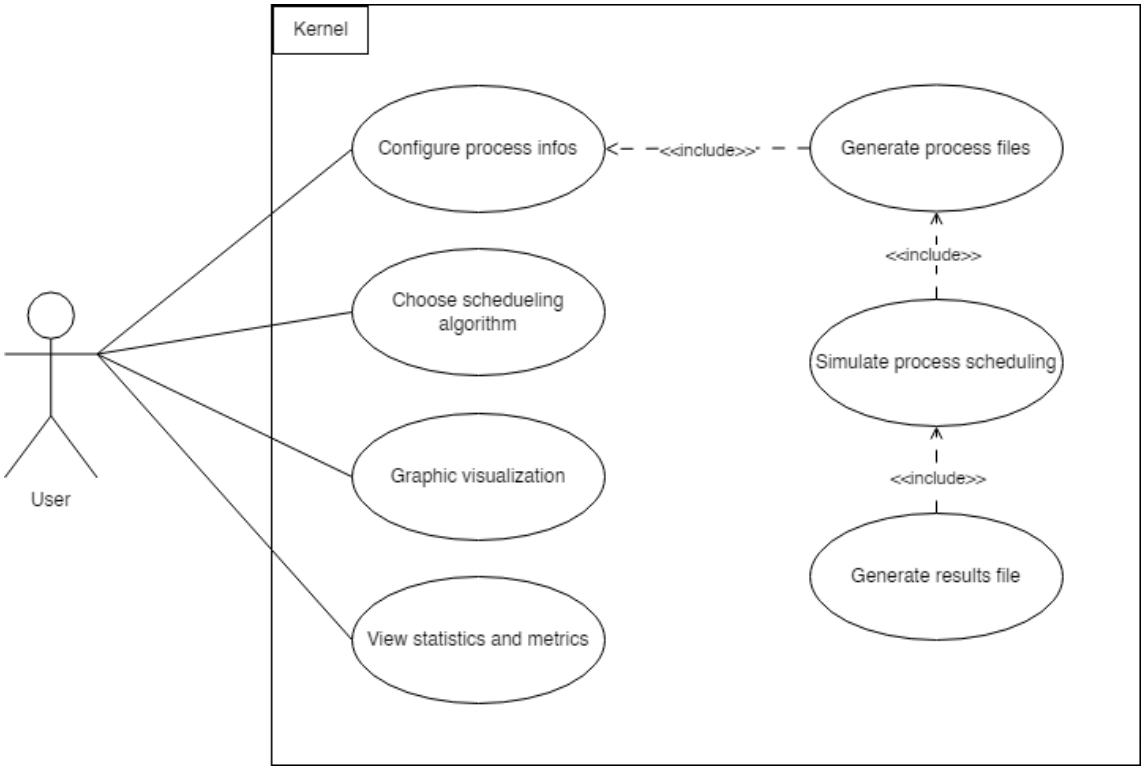


Figure.1 : Use Case Diagram

The diagram describes the following use cases:

- **Configure process information**: This use case allows the user to input information about the processes to be simulated, such as the number of processes, the arrival time, the burst time, the priority, and the deadline of each process.

- **Generate process files:** This use case enables the software to generate files for the processes based on the user's input. The files contain the process ID, the arrival time, the burst time, the priority, and the deadline of each process.

- **Choose scheduling algorithm:** This use case lets the user select the scheduling algorithm to be used in the simulation, such as first come first serve (FCFS), priority scheduling, round robin (RR), shortest remaining time first (SRTF) or Multilevel priority scheduling.

- **Simulate process scheduling:** This use case allows the software to simulate the process scheduling based on the user's input and the chosen algorithm. The software assigns the CPU (simulation) to the processes according to the algorithm and calculates the waiting time, the turnaround time, and the response time of each process.

- **Graphic visualization:** This use case enables the software to generate a graphical representation of the simulation, such as a Gantt chart, a timeline, or a pie chart. The graphic visualization shows the execution order, the duration, and the status of each process.

- **Generate results file:** This use case allows the software to generate a file with the results of the simulation, such as the average waiting time, the average turnaround time, the average response time, the CPU utilization, and the throughput.*

- **View statistics and metrics:** This use case lets the user view statistics and metrics about the simulation, such as the minimum, maximum, median, and standard deviation of the waiting time, the turnaround time, and the response time of the processes.

1.3. Product Backlog

After exploring the use case diagram and the different system requirements here is the initial Product backlog:

As a	I want to be able to	So that I can	Priority
User	Configure process information	Provide the different process information (Intervals)	M
User	Choose scheduling algorithms	Observe the process scheduling	M

User	Graphic visualization of the execution	See the process executing in-real-time	C
User	View statistics and metrics	Compare different algorithms.	C
System	Generate process file	Have a database of process to work with	M
System	Simulate process scheduling	Provide execution results file	M
System	Generate results file	Provide information about the algorithm's performances.	M

Table 2 : Product Backlog

It is important to note that we utilized the MoSCoW prioritization system during the development process of this project. MoSCoW stands for Must-Have, Should-Have, Could-Have, and Won't-Have. This system helped us to clearly define the essential features and functionalities of the project, ensuring we focused on the most critical elements first.

1.4. Sprints Planification

The project's next phase involves sprint planning, a collaborative session where the team defines the upcoming sprint's scope. This crucial meeting will determine the sprint title, identify the primary epics to be tackled, and establish estimated start and end dates. Through open discussions and collective estimations, the team will ensure a focused, achievable sprint cycle conducive to delivering tangible progress. This collaborative approach fosters transparency, alignment, and commitment amongst all team members, setting the stage for successful sprint execution.

Below is the sprints planification table:

Sprint	Title	Epics	Start Date	End Date
Sprint 1	FIFO & Round Robin Algorithms	• Process acquisition. • Process file generation. • FIFO algorithm implementation. • Round Robin algorithm implementation.	Oct 1st	Oct 15th
Sprint 2	SRT & Priority Algorithms	• Shortest remaining time algorithm implementation. • Priority algorithm. • Gantt chart implementation.	Oct 16th	Oct 31st
Sprint 3	Multilevel Priority Algorithm.	• Multilevel algorithm implementation	Nov 1st	Nov 15th
Sprint 4	Makefile & Metrics implementation	• Implement Makefile. • Add metrics functionalities.	Nov 16th	Nov 30th

Table 3 : Sprint Planification

2. Work Environment

The C programming language served as the backbone of the project's development. Its selection offered several advantages that proved instrumental to achieving the project's goals.

The project development primarily resided in the Linux environment, leveraging the power and flexibility of open-source software. This choice aligned perfectly with the project's focus on simulating multitasking scheduling algorithms in a Linux context.

For code creation and manipulation, the team utilized two popular Integrated Development Environments (IDEs): Code::Blocks and Visual Studio Code. Both platforms offered a robust set of features for C programming, including syntax highlighting, code completion, and debugging tools. This choice provided flexibility to team members, allowing them to work with their preferred IDE while maintaining consistency and collaboration.

To further enhance collaboration and code sharing, the team utilized GitHub as the repository hosting platform. GitHub provided a centralized location for storing, managing, and accessing the codebase, fostering open communication and collaboration among team members.

The combination of Linux and these IDEs facilitated a highly efficient and collaborative work environment. The open-source nature of the environment ensured accessibility and reduced potential licensing restrictions, while the chosen IDEs provided robust functionalities and customization options. This combination empowered the team to focus on the core development tasks, facilitating efficient code writing, debugging, and collaborative project progress.

3. Project Architecture

To maintain a comprehensive view of the project we opted for the following project structure:

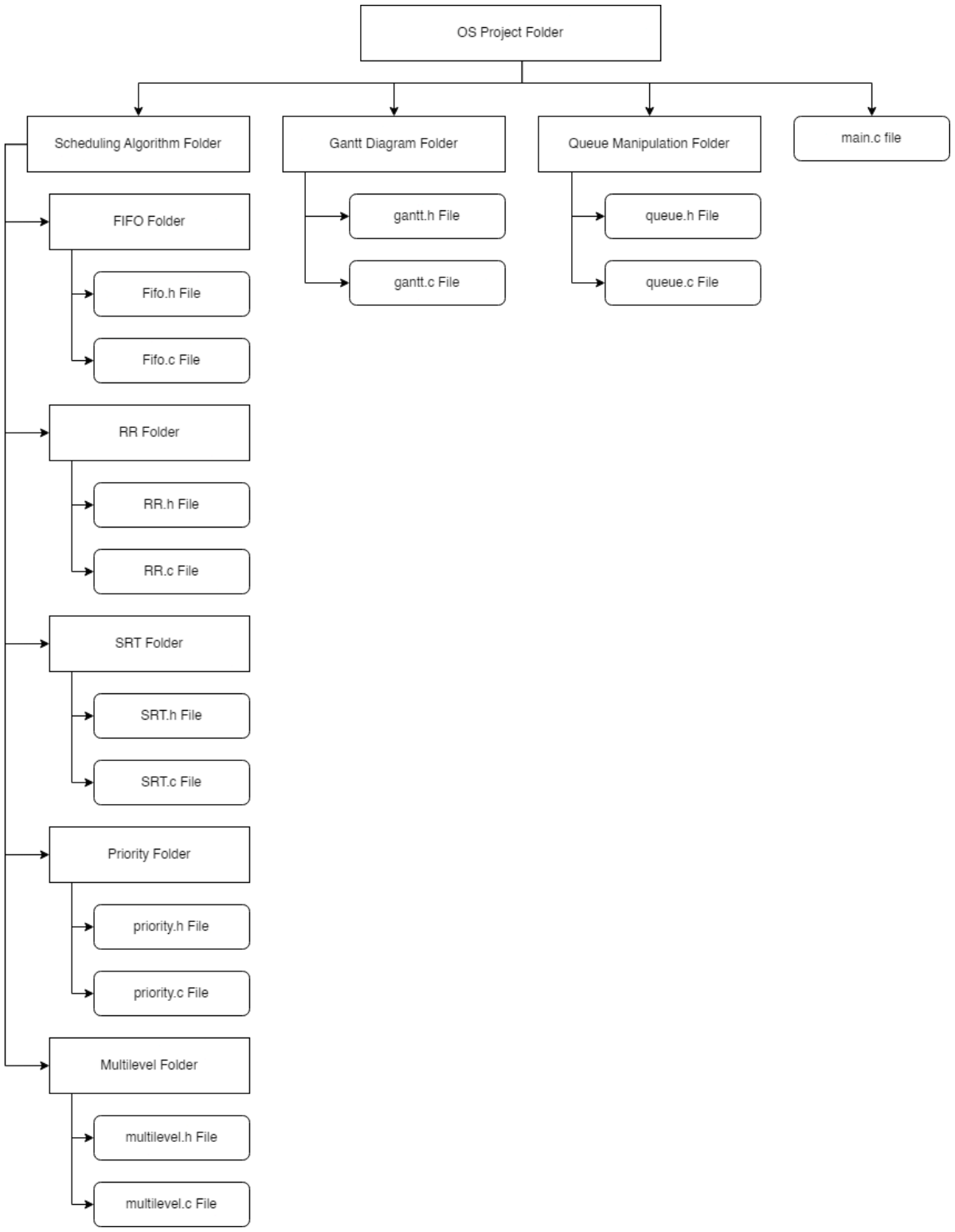


Figure.2 : Project Structure

Below is the initial structure in the IDE:

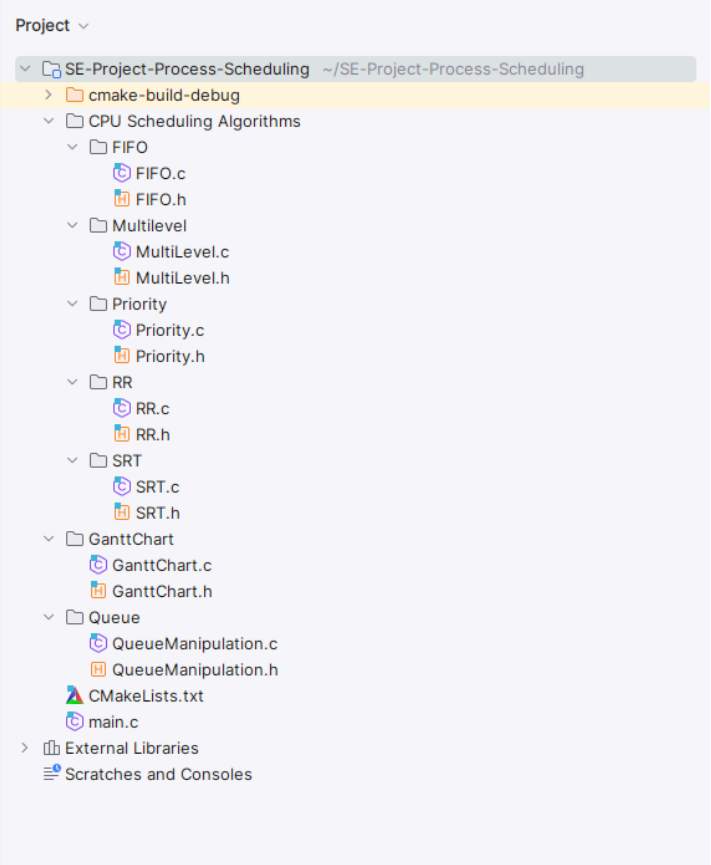


Figure.3 : Project Structure

Making modular code in C with header files and separate functions offers several advantages, contributing to better code organization, maintainability, and reusability.

Each algorithm code will reside inside a separate folder; with its own .C and header (.h) files.

The queueManipulation folder contains the file responsible for all queue manipulation functions.

The GanttChart folder contains the file responsible for displaying the execution of the process in a Gantt diagram form.

Finally, **the main** file will contain the code responsible for the menu and a reference to the algorithm functions.

Below is the architecture we get applying the modular approach:

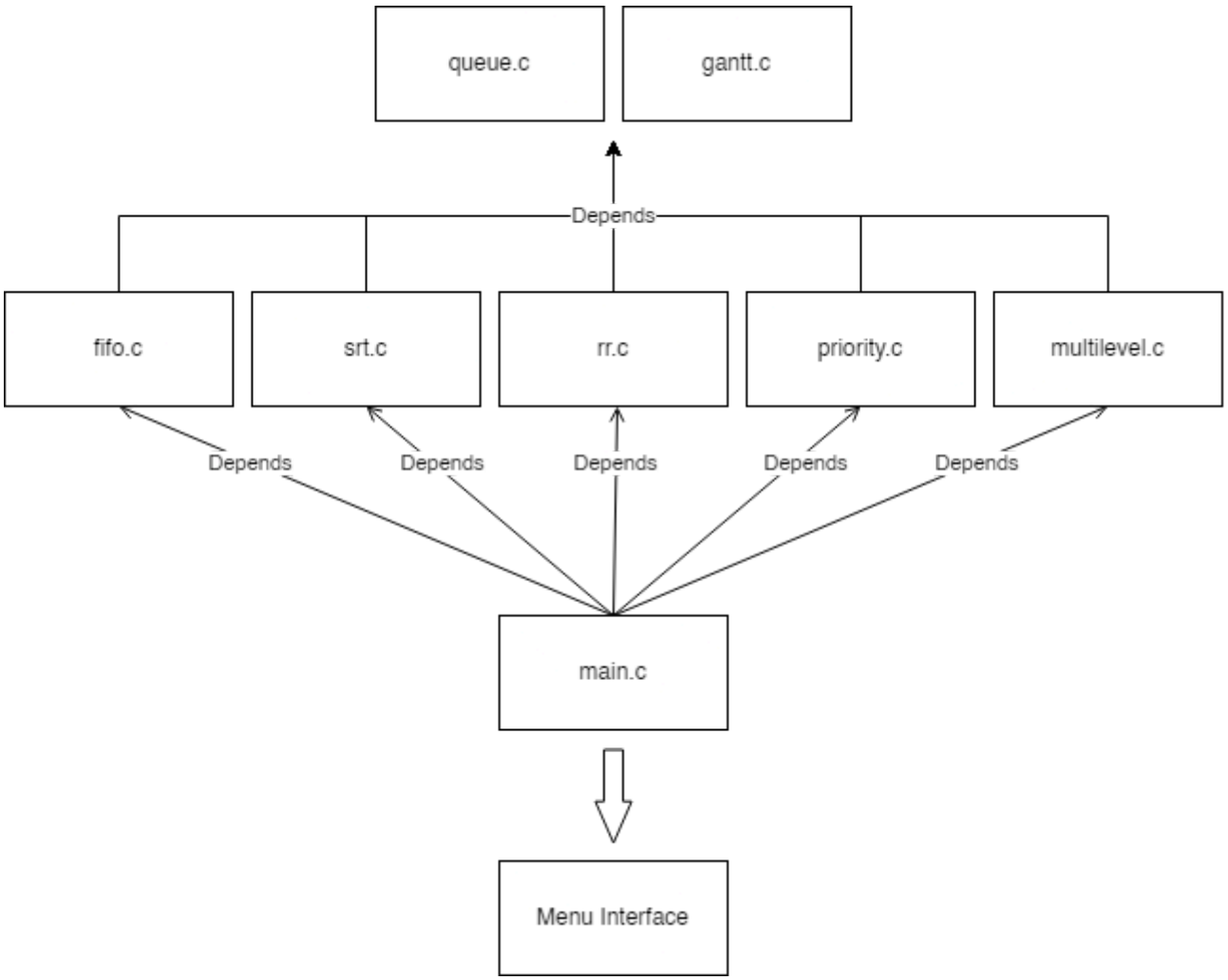


Figure.4 : Project Modular Approach

Chapter II : Sprint 1 - FIFO & Round Robin

1. FIFO

1.1. Algorithm Introduction

First-In-First-Out (FIFO) is a simple scheduling algorithm that manages the execution of processes in the order they arrive in the ready queue. The key concept behind FIFO is that the first process to arrive is the first to be executed. This scheduling algorithm operates based on a queue data structure.
Here's a step-by-step description of how the FIFO scheduling algorithm works:

Arrival of Processes:
As processes arrive, they are added to the end of the ready queue.

Selection of Next Process:
The process at the front of the ready queue is selected for execution. This is the process that arrived first and has been waiting the longest in the queue.

Execution:
The selected process is executed by the CPU until it completes its execution, is preempted, or voluntarily yields the CPU.

Completion or Removal:
Once a process completes its execution, it is removed from the system, and the next process in the ready queue becomes the new front and is selected for execution.

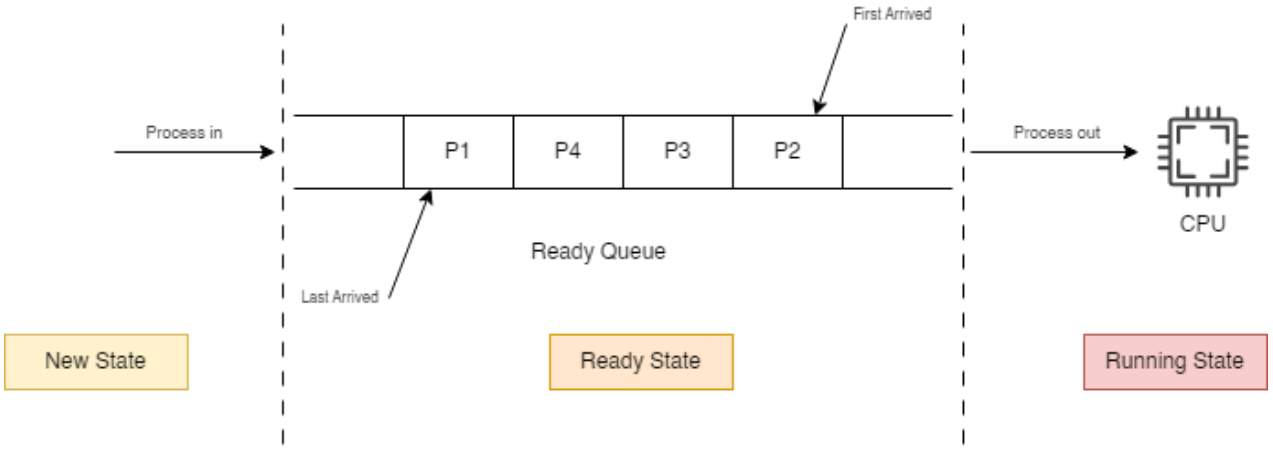


Figure.5 : FIFO Algorithm

It is to be noted that:

1. If a new process arrives while the CPU is busy executing a process, it is added to the end of the ready queue, and it will have to wait for its turn.

2. **FIFO does not involve preemption**. Once a process starts executing, it continues until it finishes or is blocked. There is no concept of priority or time-slicing in the basic FIFO algorithm.

1.2. Implementation

The implementation phase started by gathering the processes: Since we're in a simulation, the user will provide different intervals (Arrival Time, Execution Time, Priority…).
From there, the system will generate a process file containing processes with random characteristics (within the interval provided by the user)
This file will act as the database of our system, from which it will retrieve information about the processes.

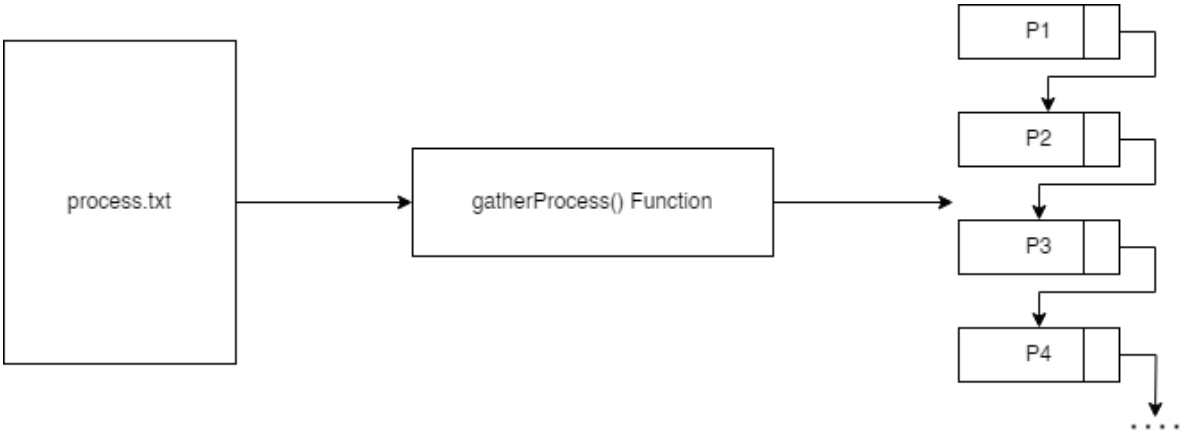


Figure.6 : Gathering processes

The gatherProcess() function will take in input the process.txt file and will extract the processes and return it in a linked list.

We end up with a linked list containing the different processes, each process is a struct containing the process characteristics:

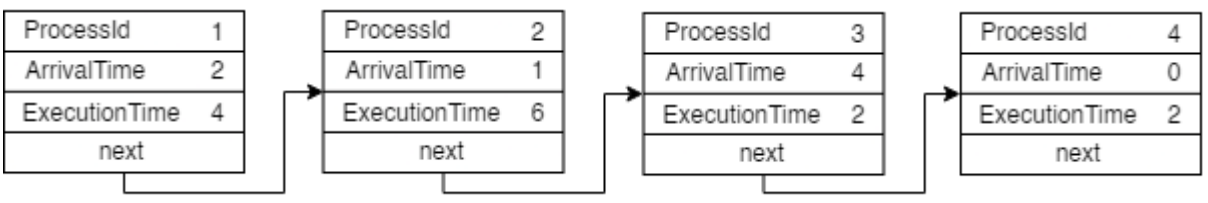


Figure.7 : Processes linked list

The FIFO function will then take the linked list as entry, and call the BubbleSort function from the queueManipulation module, in order to sort the list based on arrival time.

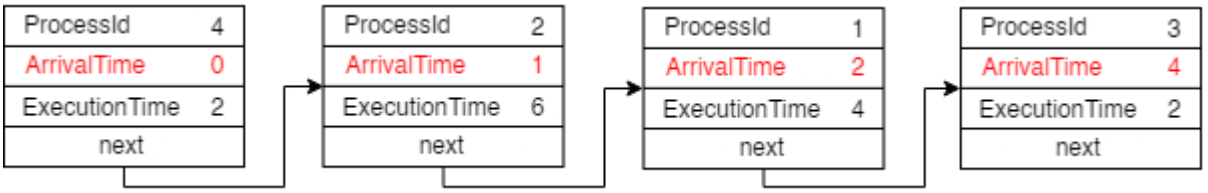


Figure.8 : Processes linked list after sort

From here, the FIFO function will create an execution list using the GanttChart Module. This list of structs will contain the start and end time of each process, making it easier to display the chart.

Below are code snippets of the implementation of the different functions:

```c
/* Function: FIFO
 Description: Implements the First-In-First-Out (FIFO) scheduling algorithm
 Parameters:
   - liste L: A linked list of processes representing the tasks to be scheduled
 Return:
   - struct exec*: Head of the Gantt chart linked list representing the execution order */

struct exec* FIFO(liste L) {

    // Initialize variables to manage the Gantt chart and process count
    struct exec* GanttChart = NULL; // Head of the Gantt chart linked list

    struct exec* tail = NULL;       // Pointer to the last element in the Gantt chart linked list

    int count = 0;                  // Variable to count the number of processes

    struct process* temp = L;       // Temporary pointer to traverse the linked list

    // Count the number of processes in the linked list
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    // Sort the linked list based on arrival time using the bubble sort algorithm
    bubbleSort(&L, count);

    printf("\033[1;34m \tExecution Details: \033[0m\n");

    // Initialize variables for tracking total turnaround and waiting time
    int totalTurnaroundTime = 0;
    int totalWaitingTime = 0;

    // Initialize the current time variable with the arrival time of the first process
    int currentTime = L->arrival;
```

```
    // Traverse the sorted linked list and create entries for the Gantt chart
    while (L != NULL) {
        // Allocate memory for a new entry in the Gantt chart
        struct exec* newEntry = (struct exec*)malloc(sizeof(struct exec));

        // Populate the new entry with process details
        newEntry->id = L->pid;
        newEntry->start = currentTime;
        newEntry->end = currentTime + L->execution;
        newEntry->next = NULL;

        // Update the Gantt chart linked list
        if (GanttChart == NULL) {
            // If the Gantt chart is empty, set the new entry as the head and tail
            GanttChart = newEntry;
            tail = newEntry;
        } else {
            // If the Gantt chart is not empty, append the new entry to the end and update the tail
            tail->next = newEntry;
            tail = newEntry;
        }

        // Calculate turnaround and waiting time for the current process
        int turnaroundTime = newEntry->end - L->arrival;
        int waitingTime = turnaroundTime - L->execution;

        // Update total turnaround and waiting time
        totalTurnaroundTime += turnaroundTime;
        totalWaitingTime += waitingTime;

        // Display information about the current process execution
        printf("\tP%d | Started at %d | Ended at %d | Turnaround Time: %d | Waiting Time: %d\n",
               L->pid, newEntry->start, newEntry->end, turnaroundTime, waitingTime);

        // Update the current time for the next iteration
        currentTime += L->execution;

        // Move to the next process in the linked list
        L = L->next;
    }

    // Calculate and display average turnaround and waiting time
    float avgTurnaroundTime = (float)totalTurnaroundTime / count;
    float avgWaitingTime = (float)totalWaitingTime / count;

    printf("\nAverage Turnaround Time: %.2f\n", avgTurnaroundTime);
    printf("Average Waiting Time: %.2f\n\n", avgWaitingTime);

    printf("\n");

    // Return the head of the Gantt chart linked list
    return GanttChart;
}
```

Figure.9 : FIFO code implementation

1.3. Pros & Cons

Pros of FIFO Scheduling:

- **Simple and Easy to Implement:** The FIFO scheduling algorithm is simple to understand and implement. It does not require complex data structures or algorithms.

- **Predictable Behavior:** The behavior of FIFO is predictable, making it easier to analyze and reason about in terms of scheduling decisions.

Cons of FIFO Scheduling:

- **Poor Turnaround Time:** FIFO may result in poor turnaround time, especially when there is a mix of short and long processes. Short processes may have to wait for a long time if they arrive after long processes have already started execution.

- **Inefficiency in Resource Utilization:** The algorithm may not be efficient in terms of resource utilization. It doesn't consider the priority or the burst time of processes, leading to potential underutilization of CPU resources.

- **No Preemption:** Once a process starts executing, it cannot be preempted. This lack of preemption means that short processes may have to wait behind long processes, even if resources become available.

- **Not Suitable for Real-Time Systems:** FIFO is not suitable for real-time systems where strict deadlines must be met. It does not take into account the urgency or priority of tasks.

2. Round Robin

2.1. Algorithm Introduction

Round Robin (RR) is a preemptive scheduling algorithm that assigns a fixed time unit per process, known as a time quantum or time slice. The primary objective of the Round Robin algorithm is to provide fair access to the CPU for all processes in the ready queue. If a process's time quantum expires, it is moved to the back of the queue, allowing the next process in line to execute. This continues in a circular fashion. Here are the key characteristics and steps of the Round Robin scheduling algorithm:

Arrival of Processes:
As processes arrive, they are added to the ready queue.

Selection of Next Process:
The scheduler selects the process at the front of the ready queue to execute. This process is given a fixed time quantum to run.

Execution:
The selected process is allowed to execute for the duration of its time quantum. If the process completes its execution within the time quantum, it is moved to the back of the queue.

Preemption:
If a process's time quantum expires before it completes execution, the process is preempted and moved to the back of the queue. The scheduler then selects the next process in line to run.

Handling New Arrivals:
When a new process arrives, it is added to the end of the ready queue.

Completion or Removal:
Once a process completes its execution, it is removed from the system.

Cycle Continues:
The process of selecting the next process, executing it for a time quantum, and potentially preempting it continues in a circular manner, giving each process a fair share of CPU time.

The following diagram explains the principle of Round Robin:

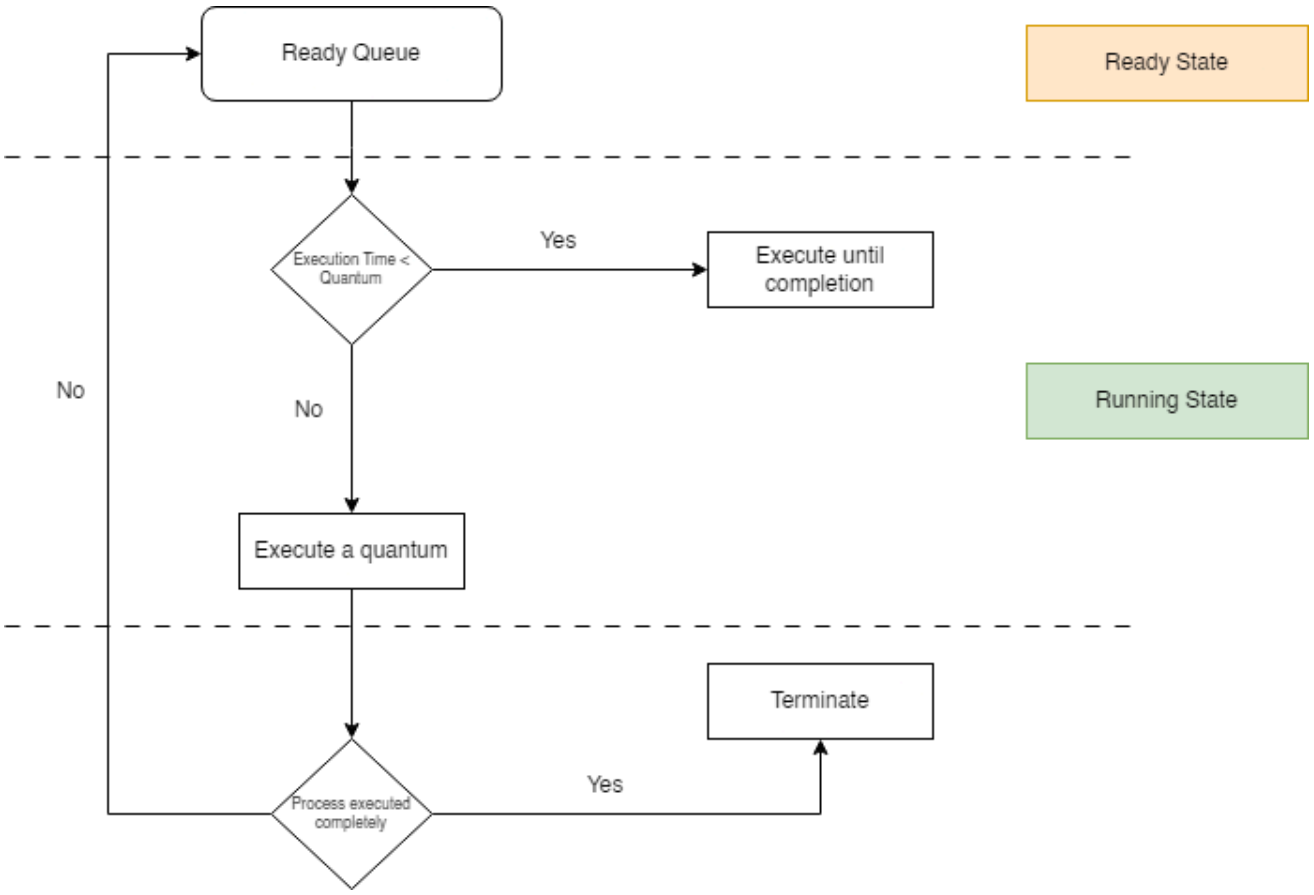


Figure.10 : Round Robin activity diagram

2.2. Implementation

Below are code snippets of the RR (Round Robin) function implementation:

```c
/* Function: RR (Round Robin)
 Description: Implements the Round Robin scheduling algorithm
 Parameters:
   - liste L: A linked list of processes representing the tasks to be scheduled
   - int quantum: The time quantum for each process execution
 Return:
   - struct exec*: Head of the Gantt chart linked list representing the order of process execution */

struct exec* RR(liste L, int quantum) {

    // Initialize variables to manage the Gantt chart, process count, and total turnaround/waiting time
    struct exec* GanttChart = NULL; // Head of the Gantt chart linked list

    struct process currentProcess;  // Temporary variable to store the current process details

    int nbProcess = getProcessNumber(L); // Number of processes in the list

    int i = 0; // Counter for processes

    int time = 0, stop = 0; // Simulation time and counter for completed processes

    liste unfinishedList = NULL; // List of processes with remaining execution time

    queue Q = createQueue(); // Queue to hold processes ready for execution

    // Variables for tracking total turnaround and waiting time
    int totalTurnaroundTime = 0;
    int totalWaitingTime = 0;

    printf("\033[1;34m \n\tExecution Details: \033[0m\n");

    // Continue simulation until all processes are completed
    while (stop < nbProcess) {
        // Enqueue processes that have arrived at or before the current time
        liste tmp = L;
        while (tmp != NULL) {
            if (tmp->arrival <= time) {
                enqueueProcess(Q, tmp);
                L = deleteProcess(L, tmp);
            }
            tmp = tmp->next;
        }

        // Enqueue processes from the unfinished list
        tmp = unfinishedList;
        liste tmp2;
        while (tmp != NULL) {
            enqueueProcess(Q, tmp);
            tmp2 = tmp;
            tmp = tmp->next;
            deleteProcess(unfinishedList, tmp2);
        }

        // Simulate the execution of processes in the queue
        if (isEmpty(Q)) {
            time++;
        } else {
            liste remainingProcess;
            // Process each item in the queue
            while (!isEmpty(Q)) {
                currentProcess = dequeueProcess(Q);
                int initialExec = currentProcess.execution;
```

```
                // Create a new entry for the Gantt chart
                struct exec* newEntry = (struct exec*)malloc(sizeof(struct exec));
                newEntry->id = currentProcess.pid;
                newEntry->start = time;

                // If the remaining execution time is less than or equal to the quantum
                if (currentProcess.execution <= quantum) {
                    time += currentProcess.execution;
                    currentProcess.execution = 0;
                    stop++;

                    // Calculate and update turnaround and waiting time
                    int turnaroundTime = time - currentProcess.arrival;
                    int waitingTime = turnaroundTime - initialExec;
                    totalTurnaroundTime += turnaroundTime;
                    totalWaitingTime += waitingTime;

                    // Display process termination information
                    printf("\tP%d | Started at %d | Ended at %d | Remaining Time: %d ",
                           currentProcess.pid, newEntry->start, time, currentProcess.execution);
                    printf("\033[38;2;249;16;16m => P%d terminated \033[0m\n", currentProcess.pid);

                    // Display detailed turnaround and waiting time information
                    printf("\033[1;34m \tP%d | Turnaround Time: %d | Waiting Time: %d \033[0m \n",
                           currentProcess.pid, turnaroundTime, waitingTime);
                } else {
                    // If the remaining execution time is greater than the quantum
                    time += quantum;
                    currentProcess.execution -= quantum;

                    // Create a new process with the remaining execution time and add it to the list
                    remainingProcess = createProcess(currentProcess.pid, currentProcess.arrival,
                                                     currentProcess.execution,currentProcess.priority);
                    addProcessToList(L, time, remainingProcess);

                    // Display process information for the current time slice
                    printf("\tP%d | Started at %d | Ended at %d | Remaining Time: %d\n",
                           currentProcess.pid, newEntry->start, time, currentProcess.execution);
                }

                // Update the end time in the Gantt chart entry
                newEntry->end = time;
                GanttChart = addExec(GanttChart, newEntry);
                i++;
            }
        }

        printf("\033[1;32m\ntime %d :\033[0m", time);
    }

    // Calculate and display average turnaround time and average waiting time
    float avgTurnaroundTime = (float)totalTurnaroundTime / nbProcess;
    float avgWaitingTime = (float)totalWaitingTime / nbProcess;

    printf("\n\nAverage Turnaround Time: %.2f\n", avgTurnaroundTime);
    printf("Average Waiting Time: %.2f\n\n", avgWaitingTime);

    // Return the head of the Gantt chart linked list
    return GanttChart;
}
```

Figure.11 : Round Robin code implementation

Here are the results of the execution:

```
 Sorted list of processes based on arrival time:
-------------------------------------------------------------------
| Process | Arrival Time | Execution Time | Priority |
-------------------------------------------------------------------
|   P 1   |      1       |       2        |     1    |
|   P 2   |      1       |       3        |     3    |
|   P 3   |      1       |       4        |     4    |
|   P 4   |      1       |       4        |     3    |
-------------------------------------------------------------------
```

Figure.12 : Execution's result for sorted list of generated processes

```
 2- Round Robin Scheduling:

Enter the time quantum for Round Robin: 2

        Execution Details:

time 1 :        P1 | Started at 1 | Ended at 3 | Remaining Time: 0  => P1 terminated
        P1 | Turnaround Time: 2 | Waiting Time: 0

time 3 :        P2 | Started at 3 | Ended at 5 | Remaining Time: 1

time 5 :        P3 | Started at 5 | Ended at 7 | Remaining Time: 2

time 7 :        P4 | Started at 7 | Ended at 9 | Remaining Time: 2

time 9 :        P2 | Started at 9 | Ended at 10 | Remaining Time: 0  => P2 terminated
        P2 | Turnaround Time: 9 | Waiting Time: 6

time 10 :       P3 | Started at 10 | Ended at 12 | Remaining Time: 0  => P3 terminated
        P3 | Turnaround Time: 11 | Waiting Time: 7

time 12 :       P4 | Started at 12 | Ended at 14 | Remaining Time: 0  => P4 terminated
        P4 | Turnaround Time: 13 | Waiting Time: 9

time 14 :

 Average Turnaround Time: 8.75
 Average Waiting Time: 5.50

        Gantt Chart:
|  P 1  *  *  |  P 2  *  *  |  P 3  *  *  |  P 4  *  *  |  P 2  *  |  P 3  *  *  |  P 4  *  *  |
```

Figure.13 : RR Scheduling's results of execution

2.3. Pros & Cons

Advantages of Round Robin Scheduling:

- **Fairness:** Round Robin provides fair access to the CPU for all processes. Each process gets an equal opportunity to execute within its time quantum.

- **Prevents Starvation:** Since processes are scheduled in a circular manner, no process is indefinitely delayed or starved. Every process eventually gets a chance to run.

- **Simple Implementation:** Round Robin is relatively simple to implement and understand compared to some other scheduling algorithms.

- **Good for Time-Sharing Systems:** Round Robin is commonly used in time-sharing systems where multiple users interact with the system concurrently.

Disadvantages of Round Robin Scheduling:

- **High Turnaround Time:** Round Robin may result in a higher turnaround time for processes compared to shorter time-slice algorithms. This is because a process may have to wait for its turn to come around again before completing execution.

- **Inefficiency for Short Processes:** If the time quantum is too large, the algorithm may not be efficient for short processes, as they might have to wait unnecessarily for their turn.

- **Overhead of Context Switching:** The frequent preemption and context switching may introduce overhead, especially in scenarios with very short time slices.

3. Conclusion

In this initial sprint, we laid a solid foundation by exploring and implementing two fundamental scheduling algorithms. The insights gained from the study of FIFO and Round Robin will serve as a valuable reference for future sprints, where we plan to delve into more advanced scheduling algorithms and optimizations. This sprint not only enhanced our understanding of process scheduling but also provided hands-on experience in algorithm implementation and analysis. As we move forward, we anticipate building upon these foundations to create more sophisticated and efficient scheduling solutions for our project.

Chapter III : Sprint 2 - SRT & Priority

1. SRT (Shortest Remaining Time)

1.1. Algorithm Introduction

Shortest Remaining Time First (SRTF) is a preemptive scheduling algorithm that selects the process with the shortest remaining burst time to execute next. Unlike non-preemptive algorithms like First-Come-First-Serve (FCFS) or Shortest Job Next (SJN), SRTF can interrupt the currently running process if a new process arrives with a shorter burst time. Here are the key characteristics and steps of the Shortest Remaining Time First (SRTF) scheduling algorithm:

Arrival of Processes:
As processes arrive, they are added to the ready queue.

Selection of Next Process:
The process with the shortest remaining burst time is selected for execution. If two processes have the same remaining time, the tie is usually broken in favor of the process that arrived first.

Execution:
The selected process is allowed to execute until it completes its burst or is preempted by a process with an even shorter burst time.

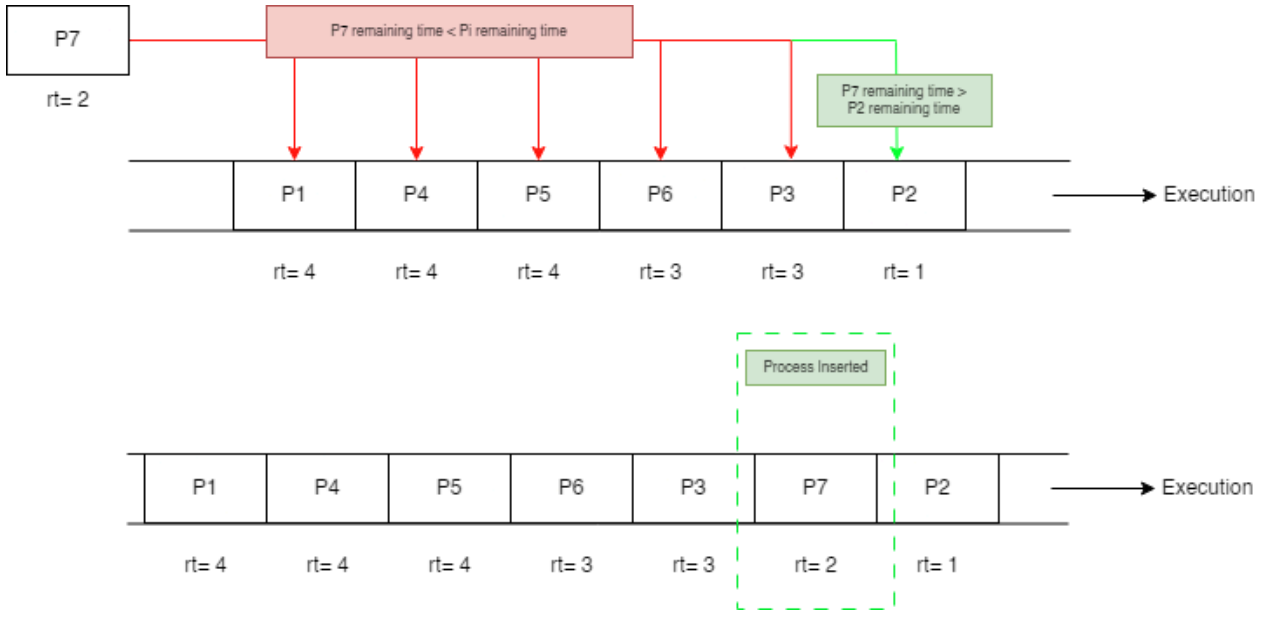


Figure.14 : Shortest Remaining Time principle

Preemption:
If a new process arrives with a shorter burst time than the currently executing process, the CPU is preempted, and the new process is scheduled. The remaining time of the preempted process is saved, and it is placed back into the ready queue.

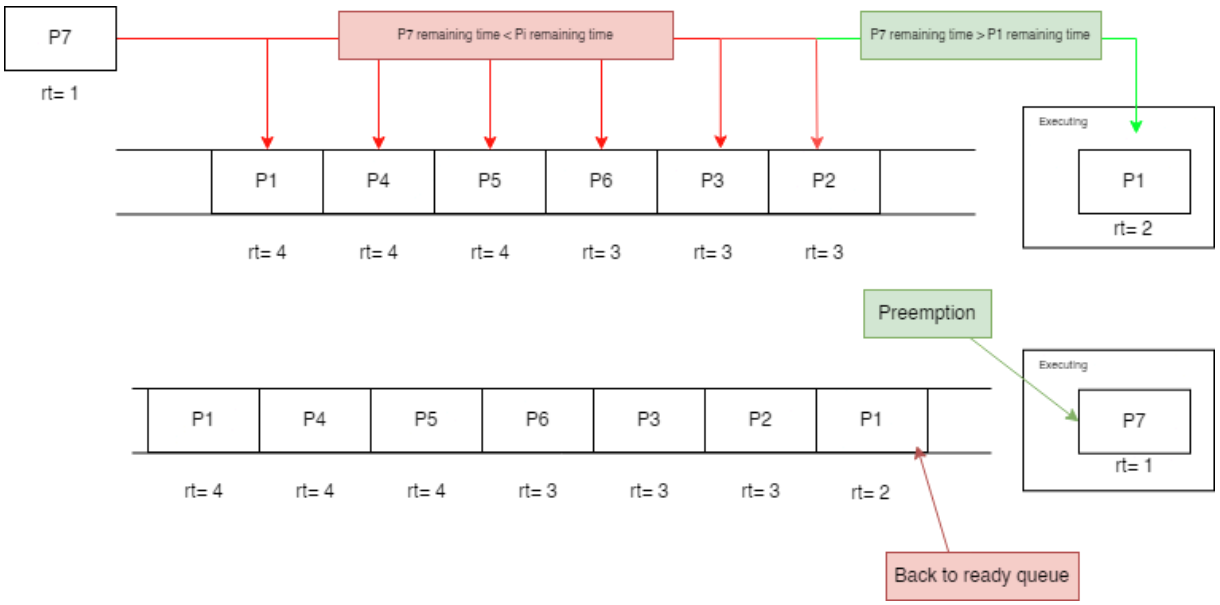


Figure.15 : Shortest Remaining Time Preemption principle

1.2. Implementation

After acquiring the processes list from the txt file, we aim to obtain the process with the minimum remaining time. For that we created a function called **minimum()** which is responsible for iterating through the list and returning the process with the shortest remaining time.

The function will take the list and the time we reached in the execution as entries.

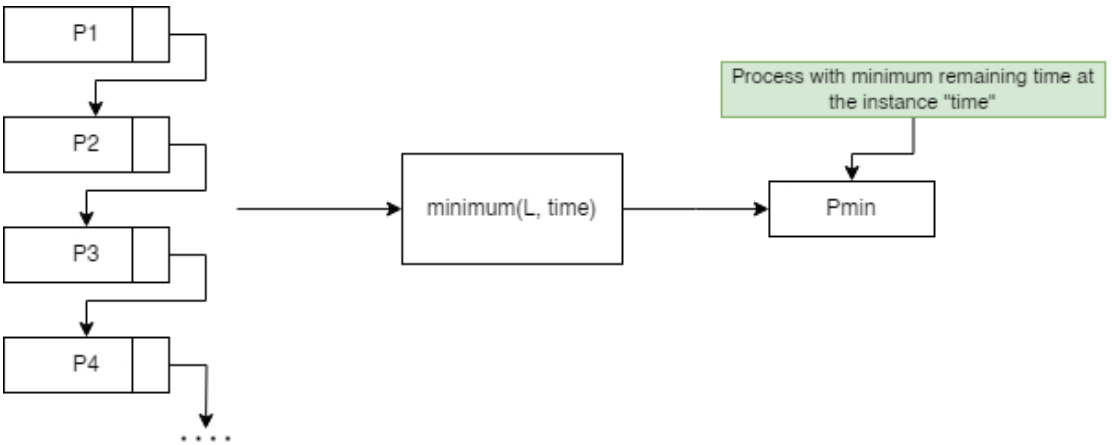


Figure.16 : Minimum remaining time function

The SRT function then will call the **minimum()** function for each instance of the execution to check for new coming processes with lower remaining time.

If the function returns a new process with lower remaining time the new process will be executed and the old one stops until it is returned again by the minimum function.

Below are code snippets of the minimum and SRT functions implementation:

```c
/* Function: minimum
 Description: Finds the process with the minimum execution time that has arrived by a given time.
 Parameters:
   - liste L: A linked list of processes representing the tasks to choose from
   - int time: The specified time until which a process must have arrived
 Return:
   - liste: A pointer to the process with the minimum execution time that has arrived by the
specified time */

liste minimum(liste L, int time) {

    // Initialize pointers for traversing the linked list
    liste tmp = L;  // Pointer to iterate through the linked list

    liste min = L;  // Pointer to the process with the minimum execution time

    // Check if the first process in the list arrives after the specified time
    if (min->arrival > time) {
        // If the first process arrives after the specified time, return NULL
        // as no process has arrived by the specified time.
        return NULL;
    } else {
        // Iterate through the linked list to find the process with the minimum execution time
        while (tmp != NULL) {
            // Check if the current process has a smaller execution time and has arrived by the
specified time
            if ((tmp->execution < min->execution) && (tmp->arrival <= time)) {
                // Update the pointer to the process with the minimum execution time
                min = tmp;
                tmp = tmp->next;  // Move to the next process in the linked list
            } else {
                tmp = tmp->next;  // Move to the next process in the linked list
            }
        }
    }

    // Return the pointer to the process with the minimum execution time
    return min;
}
```

Figure.17 : Minimum code implementation

```c
/* Function: SRT (Shortest Remaining Time)
 Description: Implements the Shortest Remaining Time scheduling algorithm
 Parameters:
   - liste L: A linked list of processes representing the tasks to be scheduled
   - int n: The total number of processes
 Return:
   - struct exec*: Head of the Gantt chart linked list representing the order of process execution */

struct exec* SRT(liste L, int n) {
    // Initialize variables to manage the Gantt chart, simulation time, and stop condition
    struct exec* GanttChart = NULL; // Head of the Gantt chart linked list

    int time = 0;                   // Simulation time

    liste m = NULL;                 // Pointer to the process with the shortest remaining time

    int stop = 0;                   // Counter for completed processes

    printf("\033[1;34m \n\tExecution Details: \033[0m\n");

    // Continue simulation until all processes are completed
    while (stop < n ) {
        // Find the process with the shortest remaining time that has arrived by the current time
        m = minimum(L, time);

        if (m != NULL) {
            // Allocate memory for a new entry in the Gantt chart
            struct exec* newEntry = (struct exec*)malloc(sizeof(struct exec));

            // Populate the new entry with process details
            newEntry->id = m->pid;
            newEntry->start = time;
            m->execution--;
            time++;
            newEntry->end = time;

            // Display detailed information about the current process execution
            printf("\033[1;32m \ntime %d : \033[0m",time-1);
            printf("\tP%d | Started at %d | Ended at %d | Remaining Time: %d ",
                   m->pid, newEntry->start, time, m->execution);

            // Update the Gantt chart linked list with the current process execution details
            GanttChart = addExec2(GanttChart, newEntry->id, newEntry->start, newEntry->end);

            // Check if the process has completed its execution
            if(m->execution == 0) {
                stop++;
                // Display termination message for the completed process
                printf("\033[38;2;249;16;16m => P%d terminated \033[0m\n", m->pid);

                // Remove the completed process from the linked list
                L = deleteProcess(L, m);
            }
        }
        else {
            time++; // If no process is ready to execute, increment simulation time
        }
    }

    // Return the head of the Gantt chart linked list
    return GanttChart;
}
```

Figure.18 : SRT code implementation

Here are the results of the execution:

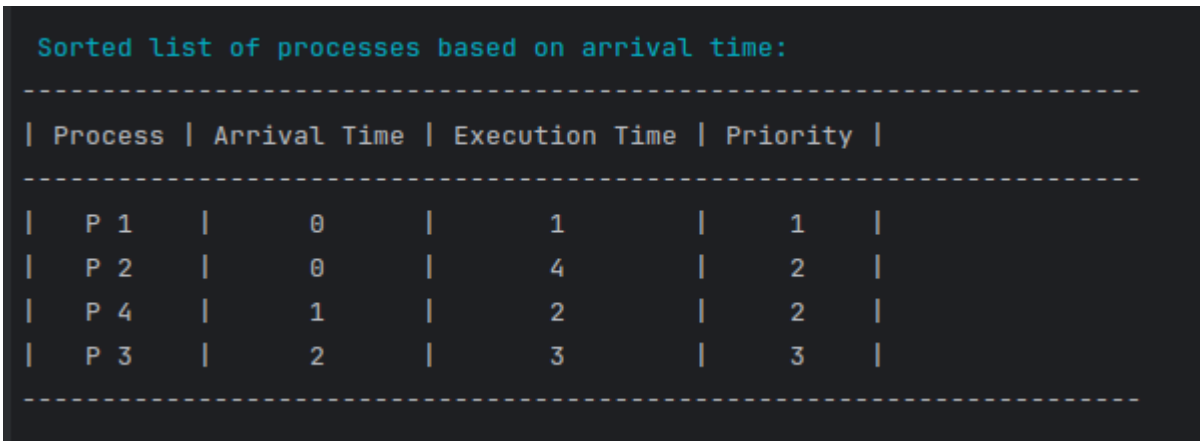


Figure.19 : Execution's result for sorted list of generated processes

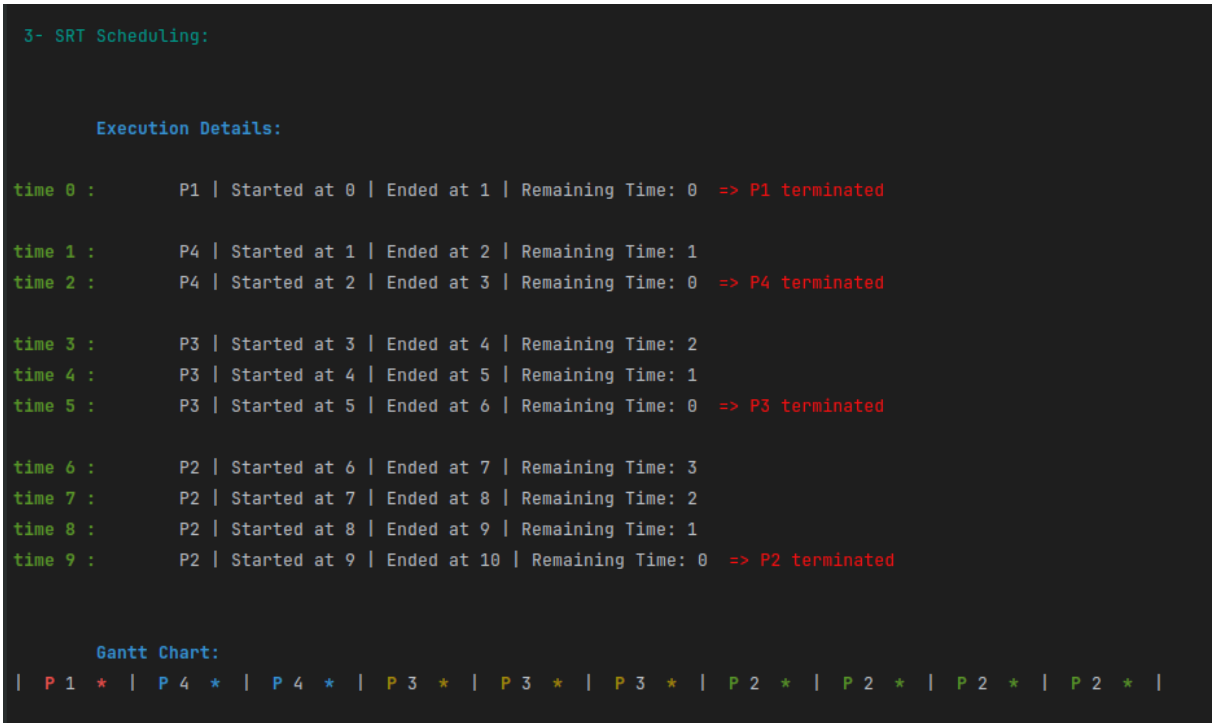


Figure.20 : SRT Scheduling's results of execution

1.3. Pros & Cons

Advantages of SRT Scheduling:

- **Minimizes Turnaround Time:** SRT aims to minimize the turnaround time by always selecting the process with the shortest remaining burst time. This results in faster completion times for shorter processes.

- **Efficient Use of CPU**: SRT tends to make efficient use of CPU time by selecting processes with the shortest burst times first, maximizing throughput.

Disadvantages of SRTF Scheduling:

- **Possibility of Starvation:** Long processes may suffer from starvation because shorter processes consistently get preference. If short processes keep arriving, long processes may be delayed indefinitely.

- **High Preemption Overhead:** The frequent preemption of processes can introduce additional overhead, as the operating system needs to save and restore the state of processes.

- **Complexity of Implementation:** Implementing a preemptive scheduling algorithm like SRTF requires careful handling of process preemption and context switching, adding complexity to the operating system.

2. Priority

2.1. Algorithm Introduction

Priority Scheduling is a preemptive scheduling algorithm that assigns priorities to each process based on certain criteria, and the process with the highest priority is selected for execution. Priority values can be assigned based on factors such as process importance, deadline urgency, or other metrics relevant to the scheduling policy.

Arrival of Processes:
As processes arrive, each process is assigned a priority value.

Selection of Next Process:
The process with the highest priority that is ready to execute is selected for execution. If a process with a higher priority arrives later, the currently running process may be preempted.

Execution:
The selected process is allowed to run for a predefined time quantum or until it voluntarily yields the CPU.

In this example, the process with the lower priority value is prioritized.

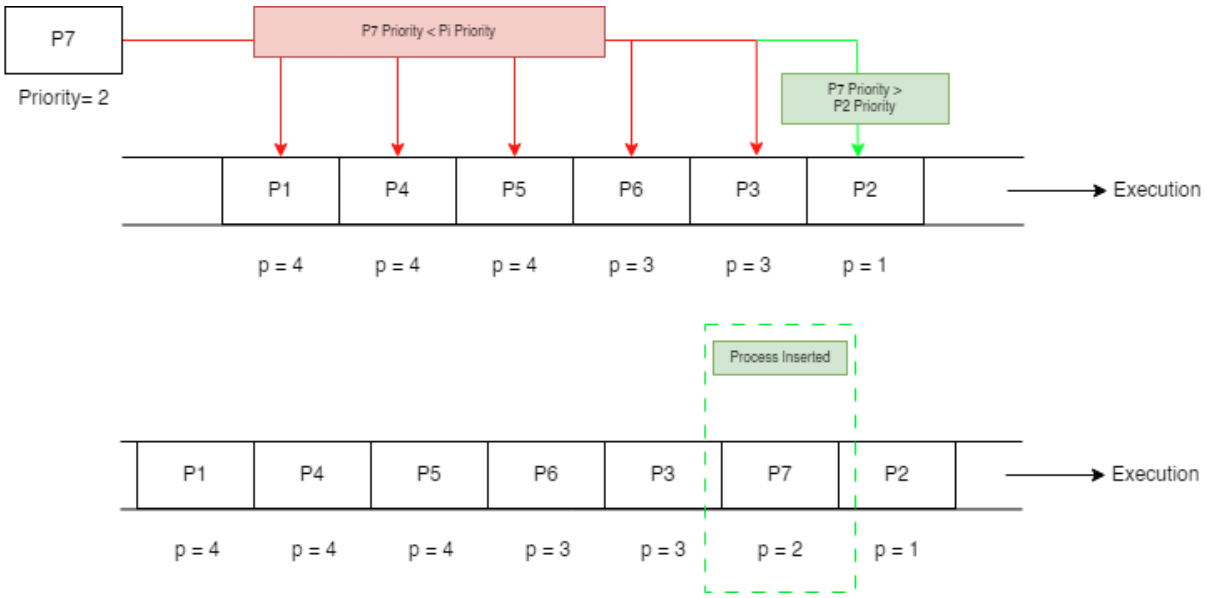


Figure.21 : Priority based algorithm principle

Preemption:

If a process with a higher priority becomes ready to execute, the currently running process may be preempted, and the higher-priority process takes over the CPU.

In this example, the process with the lower priority value is prioritized.

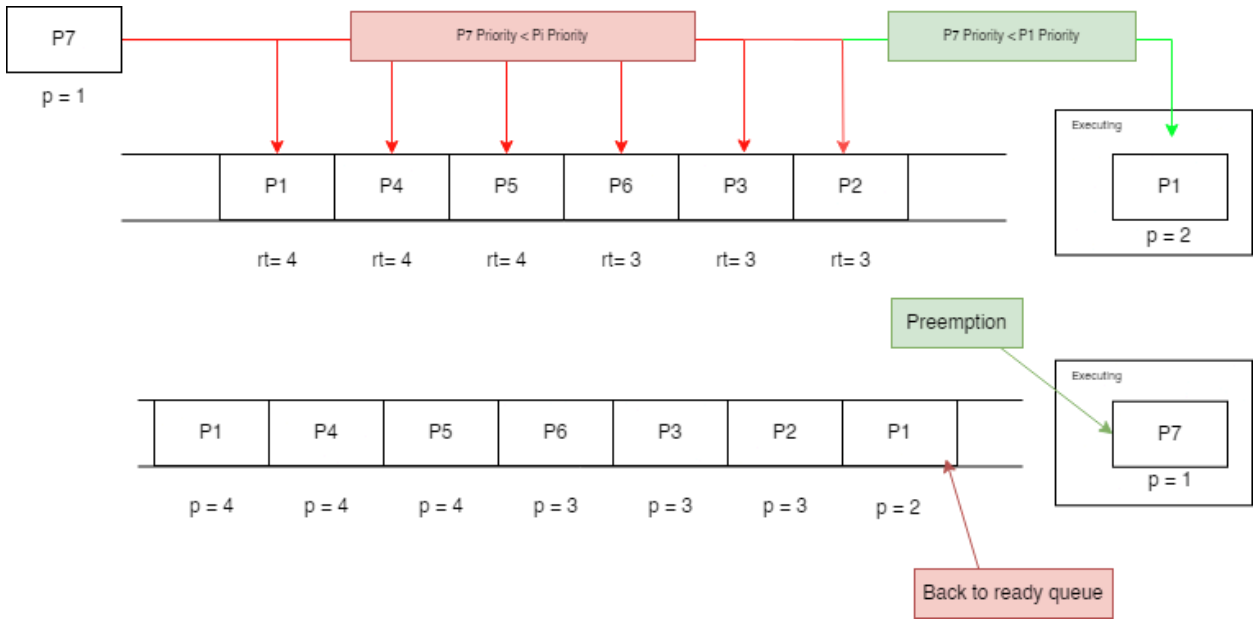


Figure.22 : Priority based algorithm preemption principle

2.2. Implementation

To implement the priority algorithm, we opted for the same principle of the SRT algorithm.

Instead of the **minimum()** function that returns the process with the lowest remaining time, we implemented a **maximumPriority()** function that returns the process with the highest priority in the list.

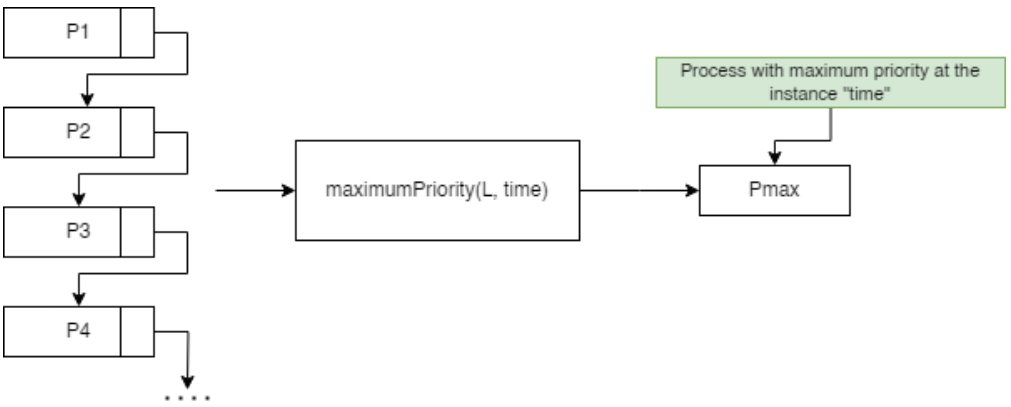


Figure.23 : Maximum priority function

The Priority function then will call **maximumPriority()** for each instance of the execution and execute the process returned.

If **maximumPriority()** returns a new process with higher priority, the new process will be executed and the old one stops until it is returned again by the function.

Below are code snippets of the maximumPriority and Priority functions implementation:

```c
/* Function: maximum
 Description: Finds the process with the maximum priority that has arrived by a given time.
 Parameters:
   - liste L: A linked list of processes representing the tasks to choose from
   - int time: The specified time until which a process must have arrived
 Return:
   - liste: A pointer to the process with the maximum priority that has arrived by the specified time */

liste maximum(liste L, int time) {

    // Initialize pointers for traversing the linked list

    liste tmp = L;  // Pointer to iterate through the linked list

    liste max = L;  // Pointer to the process with the maximum priority

    // Check if the first process in the list arrives after the specified time
    if (max->arrival > time) {
        // If the first process arrives after the specified time, return NULL
        // as no process has arrived by the specified time.
        return NULL;
    } else {
        // Iterate through the linked list to find the process with the maximum priority
        while (tmp != NULL) {
            // Check if the current process has a higher priority and has arrived by the specified time
            if ((tmp->priority > max->priority) && (tmp->arrival <= time)) {
                // Update the pointer to the process with the maximum priority
                max = tmp;
                tmp = tmp->next;  // Move to the next process in the linked list
            } else {
                tmp = tmp->next;  // Move to the next process in the linked list
            }
        }
    }

    // Return the pointer to the process with the maximum priority
    return max;
}
```

Figure.24 : Maximum code implementation

```c
/* Function: PR (Priority)
 Description: Implements the Priority scheduling algorithm
 Parameters:
   - liste L: A linked list of processes representing the tasks to be scheduled
   - int n: The total number of processes
 Return:
   - struct exec*: Head of the Gantt chart linked list representing the order of process execution */

struct exec* PR(liste L, int n) {

    // Initialize variables to manage the Gantt chart, simulation time, and stop condition
    struct exec* GanttChart = NULL; // Head of the Gantt chart linked list

    int time = 0;                   // Simulation time

    liste m = NULL;                 // Pointer to the process with the highest priority

    int stop = 0;                   // Counter for completed processes

    printf("\033[1;34m \n\tExecution Details: \033[0m\n");

    // Continue simulation until all processes are completed
    while (stop < n ) {
        // Find the process with the highest priority that has arrived by the current time
        m = maximum(L, time);

        if (m != NULL) {
            // Allocate memory for a new entry in the Gantt chart
            struct exec* newEntry = (struct exec*)malloc(sizeof(struct exec));

            // Populate the new entry with process details
            newEntry->id = m->pid;
            newEntry->start = time;
            m->execution--;
            time++;
            newEntry->end = time;

            // Display detailed information about the current process execution
            printf("\033[1;32m \ntime %d : \033[0m", time-1);
            printf("\tP%d | Started at %d | Ended at %d | Remaining Time: %d ",
                   m->pid, newEntry->start, time, m->execution);

            // Update the Gantt chart linked list with the current process execution details
            GanttChart = addExec2(GanttChart, newEntry->id, newEntry->start, newEntry->end);

            // Check if the process has completed its execution
            if(m->execution == 0) {
                stop++;
                // Display termination message for the completed process
                printf("\033[38;2;249;16;16m => P%d terminated \033[0m\n", m->pid);

                // Remove the completed process from the linked list
                L = deleteProcess(L, m);
            }
        }
        else {
            time++; // If no process is ready to execute, increment simulation time
        }
    }

    // Return the head of the Gantt chart linked list
    return GanttChart;
}
```

Figure.25 : Priority code implementation

Here are the results of the execution:

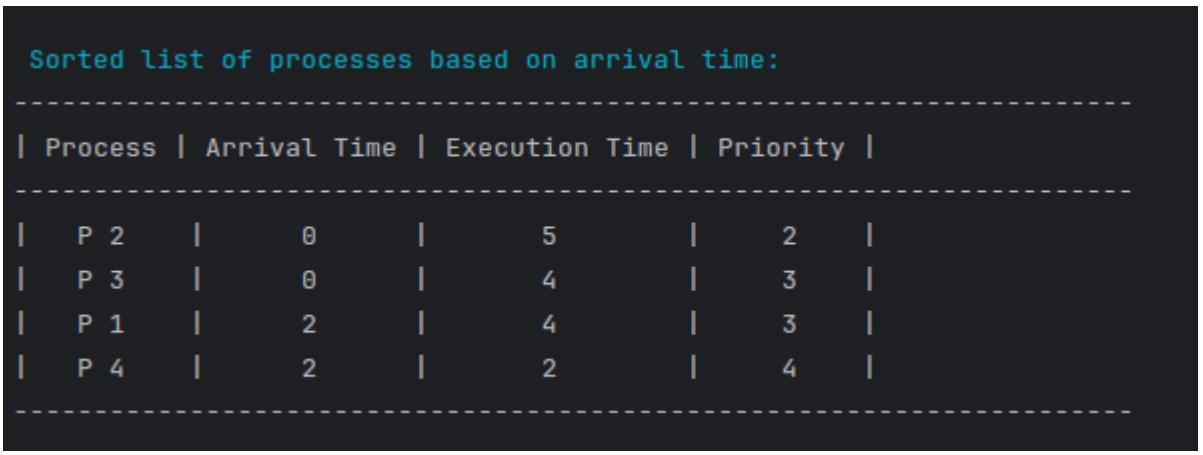


Figure.26 : Execution's result for sorted list of generated processes

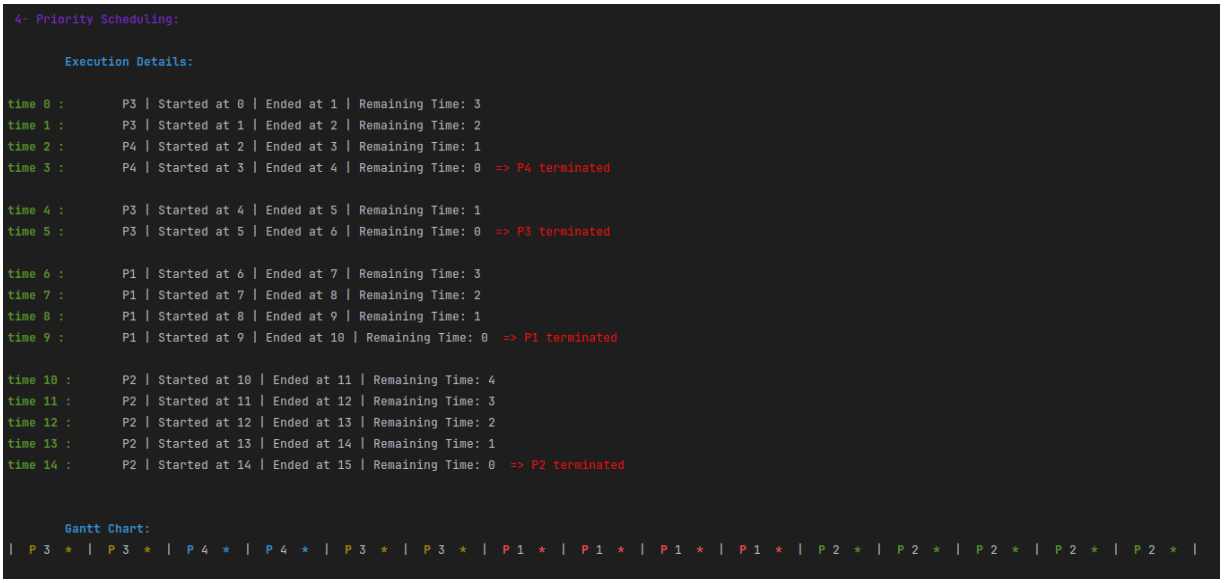


Figure.27 : Priority Scheduling's results of execution

2.3. Pros & Cons

Advantages of Priority Scheduling:

- **Flexible**: Priority scheduling is flexible and can be adapted to various scenarios by adjusting the criteria for assigning priorities.

- **Customization:** Priorities can be assigned based on business requirements, such as meeting deadlines, importance of the task, or other application-specific criteria.

Disadvantages of Priority Scheduling:

- **Starvation:** Low-priority processes may suffer from starvation if high-priority processes continuously arrive.

- **Inversion of Priorities:** Priority inversion can occur if a low-priority process holds a resource needed by a high-priority process.

- **Difficulty in Assigning Priorities:** Determining appropriate priority values for processes can be challenging, and improper assignment may lead to suboptimal performance.

- **Not Suitable for All Environments:** Priority scheduling may not be suitable for real-time systems or environments where all processes are critical, as it may lead to the neglect of lower-priority tasks.

3. Conclusion

Our exploration of SRT and Priority Scheduling has provided insights into different aspects of process scheduling in operating systems. SRT, with its focus on minimizing remaining time, is suitable for scenarios where short processes are prioritized. On the other hand, Priority Scheduling allows for a more customizable approach, where priorities can be assigned based on specific business needs.

The choice between these algorithms depends on the system's requirements, the nature of processes, and the desired optimization goals. SRT may be more suitable for scenarios where minimizing waiting times is crucial, while Priority Scheduling provides a flexible framework for addressing diverse scheduling criteria.

As we conclude this sprint, our understanding of these scheduling algorithms will contribute to making informed decisions in designing and optimizing operating systems for various computing environments.

Chapter IV : Sprint 3 - Multilevel

1. Algorithm Introduction

Multilevel scheduling algorithms are designed to manage the execution of processes in a computer system by organizing them into different priority levels or queues. The goal is to improve system responsiveness and optimize resource utilization. Here, I'll provide an overview of a basic multilevel scheduling algorithm and then discuss a popular variant known as the Multilevel Queue Scheduling algorithm.

Divide Processes into Queues:
Processes are divided into multiple priority queues based on their priority levels. Typically, there is a queue for high-priority processes and another for low-priority processes.

Scheduling Policy:
Each queue can have its scheduling policy. For example, high-priority queues may follow a Round Robin scheduling policy, while low-priority queues may use a First-Come-First-Serve (FCFS) policy.

Promotion and Demotion:
Processes can be promoted or demoted between queues based on their behavior. For instance, a process that uses CPU heavily may be demoted to a lower-priority queue to prevent it from dominating the system.

Process Migration:
Processes may be moved between queues based on their resource requirements or characteristics.

In our case, the priority is fixed and cannot be changed, thus there will be no process promotion between the different levels. Add to that, all the levels will utilize the same policy which is round robin with a fixed quantum given by the user.

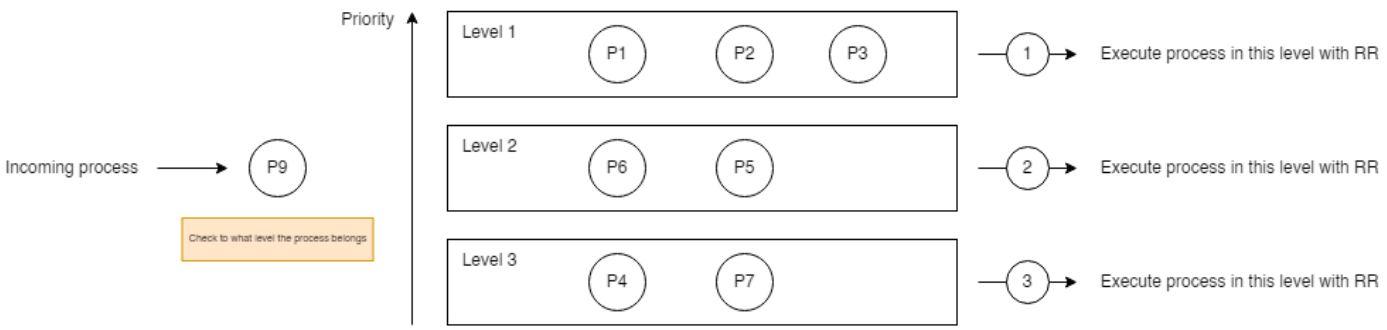


Figure.28 : Multilevel algorithm principle

2. Implementation

We start the implementation phase of the multilevel algorithm by dividing the processes list -we extracted from the txt file- into different lists, each containing the processes with the same priority.

The levels lists will be stored inside a table with a predefined size **N**.

To know the table size N, we use a function called PriorityOcc, which will return the number of priorities we have in our processes list.

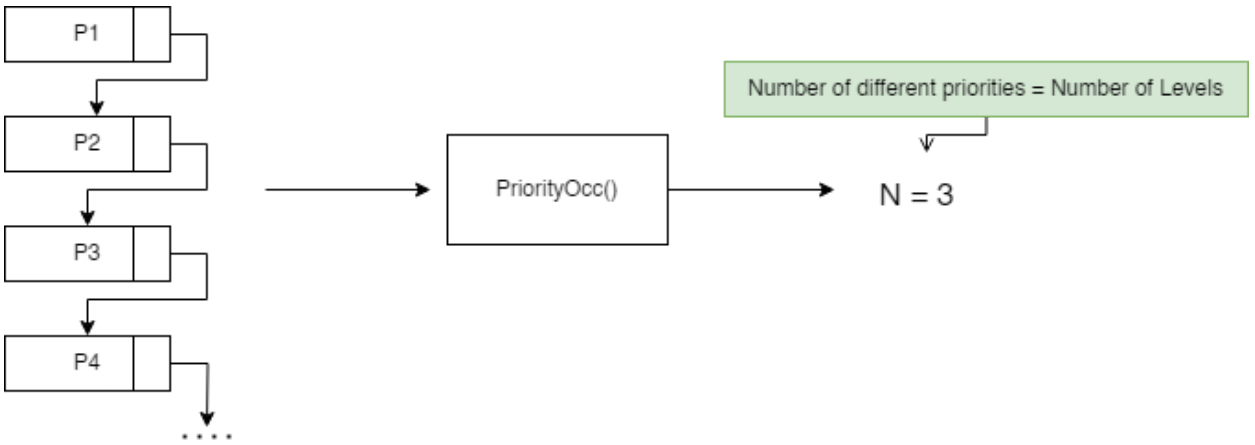


Figure.29 : PriorityOcc function

Now we can create a table with N entries. Each entry point to a list of processes with the same priority. We call the FillTable() function:

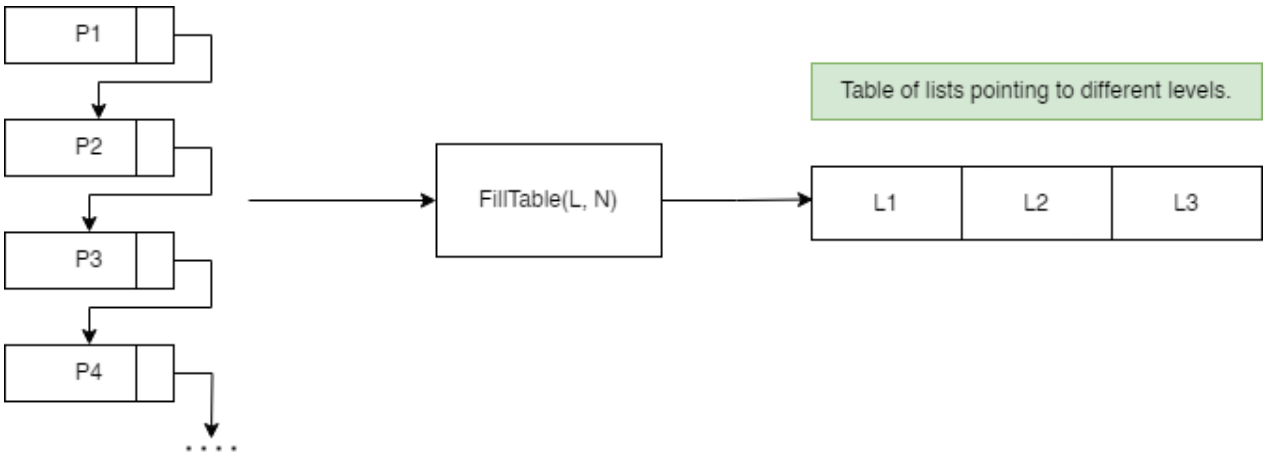


Figure.30 : FillTable function

In this case shown above, we obtain three levels of priorities. Each entry of the table will point to a list of processes, it is to be noted that the processes in each level are sorted based on their arrival time.

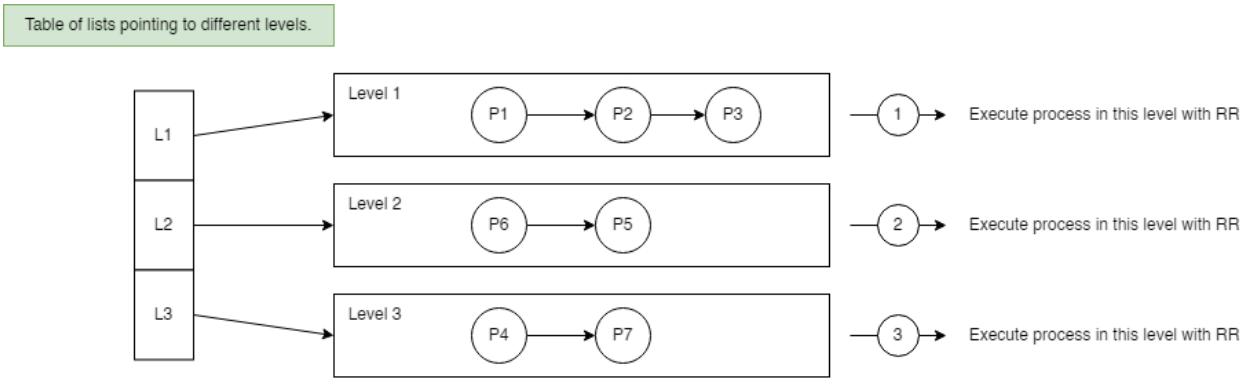


Figure.31 : Priority levels

Below are some code snippets of the functions we implemented:

```c
/* Function: priorityOcc
 Description: Counts the number of distinct priority levels in a linked list of processes.
 Parameters:
   - liste L: A linked list of processes
 Return:
   - int: The number of distinct priority levels */

int priorityOcc(liste L) {

    int occ = 1;  // Initialize the count of distinct priority levels to 1

    PrioritySort(L);  // Sort the linked list based on priority

    liste tmp = L;  // Pointer to iterate through the linked list

    // Iterate through the linked list to count distinct priority levels
    while (tmp->next != NULL) {
        // Compare the priority of the current process with the priority of the next process
        if (tmp->priority != (tmp->next)->priority) {
            occ++;  // Increment the count if the priorities are different
        }
        tmp = tmp->next;  // Move to the next process in the linked list
    }

    return occ;  // Return the count of distinct priority levels
}
```

Figure.32 : PriorityOcc code implementation

```c
/* Function: fillTable
 Description: Creates an array of linked lists, where each list contains processes
              with the same priority level. The array size is determined by the number
              of distinct priority levels in the linked list.
 Parameters:
   - liste L: A linked list of processes
   - int priorityOcc: The number of distinct priority levels in the linked list
 Return:
   - liste*: An array of linked lists, each representing processes with the same priority
level */

liste* fillTable(liste L, int priorityOcc) {

    // Allocate memory for an array of linked lists
    liste* t = malloc(priorityOcc * sizeof(liste));

    int pr;  // Variable to store the current priority level

    liste tmp = L;  // Pointer to iterate through the original linked list

    // Iterate through the priorityOcc number of distinct priority levels
    for (int i = 0; i < priorityOcc; ++i) {
        // Check if the original linked list is exhausted
        if (tmp == NULL) {
            t[i] = NULL;  // Set the i-th element of the array to NULL and continue to the
next iteration
            continue;
        }

        pr = tmp->priority;  // Get the priority of the current process
        liste p = NULL, process;  // Initialize a linked list to store processes with the same
priority

        // Iterate through the original linked list and extract processes with the same
priority
        while (tmp != NULL && tmp->priority == pr) {
            // Create a new process node with the same details as the current process
            process = createProcess(tmp->pid, tmp->arrival, tmp->execution, tmp->priority);
            // Add the new process node to the end of the linked list
            p = addProcessEnd(p, process);
            tmp = tmp->next;  // Move to the next process in the original linked list
        }

        t[i] = p;  // Store the linked list with processes of the same priority in the array
        if (tmp != NULL) {
            pr = tmp->priority;  // Update the priority for the next iteration if there are
more processes
        }
    }

    return t;  // Return the array of linked lists
}
```

Figure.33 : fillTable code implementation

3. Pros & Cons

Pros of Multilevel Scheduling Algorithm:

- **Improved Responsiveness:**
  High-priority queues ensure that interactive tasks get prompt attention, enhancing system responsiveness for users.

- **Better Resource Utilization:**
  CPU-bound tasks can be moved to lower-priority queues, preventing them from monopolizing resources and allowing other tasks to execute.

- **Fairness:**
  By dynamically adjusting the priority of processes based on their behavior, the algorithm promotes fairness and prevents any single process from dominating the system.

- **Adaptability:**
  The ability to promote or demote processes allows the system to adapt to changing workloads and prioritize tasks accordingly.

- **Efficient for Mixed Workloads**:
  Suitable for systems with a mix of short-term interactive tasks and long-term batch jobs, as it can handle different types of workloads effectively.

- **Prevention of Starvation:**
  Aging mechanisms and the ability to promote processes from lower-priority queues help prevent starvation of processes waiting in queues for a long time.

Cons of Multilevel Scheduling Algorithm:

- **Complexity:**
  Implementing and managing multiple queues with different scheduling policies and criteria can be complex, requiring careful tuning of parameters.

- **Tuning Challenges:**
  Setting appropriate thresholds for promotion, demotion, and aging can be challenging, and improper tuning may result in suboptimal performance.

- **Overhead:**
  The algorithm introduces additional overhead in terms of managing multiple queues, tracking process behavior, and deciding on promotions and demotions.

- **Potential for Starvation:**
  While the algorithm includes mechanisms to prevent starvation, improper parameter settings or inadequate design can still lead to certain processes being stuck in lower-priority queues.

- **Inefficiency for Homogeneous Workloads:**
  In scenarios where the workload is relatively uniform, a multilevel scheduling algorithm may introduce unnecessary complexity without significant benefits.

- **Difficulty in Predicting Behavior:**
  Predicting the behavior of processes and determining when to promote or demote them can be challenging, and incorrect decisions may impact overall system performance.

4. Conclusion

The conclusion of this sprint signifies a successful integration of the Multilevel Queue Scheduling algorithm into our system. With a focus on simplicity and efficiency, we adopted a strategy where processes are organized into distinct priority queues, each employing a Round Robin scheduling policy with predefined time quantums. This approach has allowed us to better manage our diverse workload, providing a fair and balanced distribution of CPU time among different types of tasks. The algorithm's straightforward design aligns with our system's requirements, and initial results show improvements in task execution and overall system responsiveness. As we reflect on this sprint, we recognize the importance of adaptability, ensuring our scheduling approach can efficiently handle varying workloads. Moving forward, we remain committed to refining and optimizing the Multilevel Queue Scheduling algorithm to further enhance our system's performance and meet the evolving demands of our users.

Chapter V : Sprint 4 - Makefile & Metrics

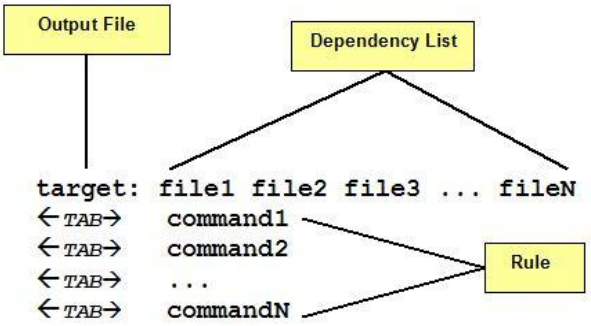
1. Makefile

1.1. Introduction to makefiles

In the realm of software development, where projects can be complex and involve numerous source files, dependencies, and compilation steps, Makefiles emerge as indispensable tools for managing the build process efficiently.

A Makefile is a script that automates the compilation and linking of source code into executable programs, providing a systematic and reproducible way to organize, build, and maintain projects. By defining rules, dependencies, and actions, Makefiles streamline the development workflow, reducing manual effort and enhancing the overall productivity of software development teams.
Below are some key components of the makefile:

Rules:
A Makefile is organized around rules, where each rule specifies a target, dependencies, and actions. The target is the output file or action to be performed, while dependencies are the files or other targets that the target relies on. The actions describe the steps to create or update the target.



Dependencies:
Makefiles leverage the concept of dependencies to determine when a target needs to be rebuilt. If a dependency is modified, the associated target is considered outdated and triggers the execution of the specified actions.

Actions:
Actions in a Makefile are the commands that the build system executes to create or update the target. These commands are typically the compilation and linking steps necessary to produce the final executable or other output.

Variables:
Makefiles allow the use of variables to store and manage project-specific settings. Variables enhance flexibility by enabling easy adjustments to compiler options, file paths, and other parameters. This simplifies the adaptation of the build system to different environments.

1.2. Implementation of makefile

```
EXECUTABLE = test

INSTALL_PATH = /usr/local/bin

SOURCE_FILES = main.c Queue/QueueManipulation.c CPU_Scheduling_Algorithms/FIFO/FIFO.c
CPU_Scheduling_Algorithms/RR/RR.c CPU_Scheduling_Algorithms/SRT/SRT.c
CPU_Scheduling_Algorithms/Priority/Priority.c CPU_Scheduling_Algorithms/Multilevel/MultiLevel.c
GanttChart/GanttChart.c

CC = gcc

CFLAGS = -Wall -Wextra -g

OBJECT_FILES = $(SOURCE_FILES:.c=.o)

MAKEFILEDIR := $(dir $(realpath $(firstword $(MAKEFILE_LIST))))

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECT_FILES)
    $(CC) $(CFLAGS) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f *.o $(EXECUTABLE)

install:
    @if [ -w $(INSTALL_PATH) ]; then \
        cp $(EXECUTABLE) $(INSTALL_PATH); \
        echo "$(EXECUTABLE) has been installed to $(INSTALL_PATH)"; \
    else \
        cp $(EXECUTABLE) ./; \
        echo "Insufficient permissions to install $(EXECUTABLE) in $(INSTALL_PATH). Installed in the
current directory instead."; \
    fi

uninstall:
    rm -f $(INSTALL_PATH)/$(EXECUTABLE
    echo "$(EXECUTABLE) has been uninstalled from $(INSTALL_PATH)"
```

Figure.34 : Makefile code implementation

1.3. Benefits of makefiles

Automation of Build Process:
One of the primary benefits of makefiles is the automation of the build process. Makefiles define the rules and dependencies for compiling and linking source code, allowing the build system to automatically determine which parts of the project need to be recompiled. This automation saves developers time and reduces the risk of errors associated with manual compilation.

Dependency Management:
Makefiles explicitly specify dependencies between source files and targets. If a source file or a dependency is modified, the make utility intelligently rebuilds only the affected components. This ensures that the build remains up-to-date and avoids unnecessary recompilation of unchanged files.

Consistency Across Environments:
By using variables in makefiles, developers can define project-specific settings, such as compiler options, file paths, and flags. This makes it easier to maintain consistency across different development environments and platforms. A well-designed makefile allows for seamless adaptation to various configurations.

Facilitates Large Projects:
In large software projects with multiple source files and complex dependencies, makefiles provide a structured way to manage the build process. They break down the build into smaller, manageable units, making it easier to organize and maintain the project as it scales.

Reproducibility:
Makefiles contribute to the reproducibility of the build process. By clearly defining rules and dependencies, developers can recreate the build environment and regenerate the executable with confidence. This is crucial for collaborative projects and when transitioning between different development environments.

Customization and Flexibility:
Makefiles are highly customizable, allowing developers to tailor the build process to suit specific project requirements. Variables, targets, and rules can be adapted to accommodate different compilers, build options, and project structures. This flexibility ensures that the build system aligns with the unique needs of the project.

Maintenance and Clean Targets:
Makefiles support maintenance activities by providing mechanisms like the "clean" target. This target allows developers to remove generated files, such as executables and object files, simplifying project cleanup. Phony targets can also be used to perform additional tasks, such as running tests or generating documentation.

2. Metrics

2.1. Introduction to metrics

In the context of process schedulers, average turnaround time and average waiting time are key metrics that measure the efficiency and performance of a scheduling algorithm. Let's explore each of these metrics:

1. Average Turnaround Time:

- **Definition:** Average Turnaround Time is the total time taken by a process to complete its execution, starting from the time it arrives in the ready queue to the time it finishes execution and exits the system.

- **Calculation:** It is computed by summing up the turnaround times of all processes and dividing by the total number of processes.

$$Average\ Turnaround\ Time\ = \frac{Total\ Turnaround\ Time\ of\ all\ Processes}{Number\ of\ Processes}$$

- **Importance:** Low average turnaround time indicates that processes are quickly moving through the system, completing their execution in a timely manner. It is a crucial metric for systems where responsiveness and quick task completion are essential, such as interactive systems.

2. Average Waiting Time:

- **Definition:** Average Waiting Time is the total time a process spends waiting in the ready queue before it gets CPU time for execution.

- **Calculation:** It is computed by summing up the waiting times of all processes and dividing by the total number of processes.

$$Average\ Waiting\ Time\ = \frac{Total\ Waiting\ Time\ of\ all\ Processes}{Number\ of\ Processes}$$

- **Importance:** Low average waiting time indicates that processes are getting access to the CPU promptly after arriving in the ready queue. Minimizing waiting time is crucial for systems aiming to achieve high throughput and effective utilization of system resources.

2.2. Implementation of metrics

```
    int initialExecution;
    int totalTurnaroundTime = 0;
    int totalWaitingTime = 0;

    // ...

    while (stop < nbProcess) {

        // ... (previous part of the function)

        if (isEmpty(Q)) {
            time++;
        }
        else {
            liste remainingProcess;
            while (!isEmpty(Q)) {
                currentProcess = dequeueProcess(Q);

                struct exec* newEntry = (struct exec*)malloc(sizeof(struct exec));
                newEntry->id = currentProcess.pid;
                newEntry->start = time;

                // Check if the remaining execution time is less than or equal to the quantum
                if (currentProcess.execution <= quantum) {
                    time += currentProcess.execution;
                    currentProcess.execution = 0;
                    stop++;

                    // Calculate turnaround time and waiting time
                    initialExecution = execT[(currentProcess.pid) - 1];
                    int turnaroundTime = time - currentProcess.arrival;
                    int waitingTime = turnaroundTime - initialExecution;

                    // Update total turnaround time and total waiting time
                    totalTurnaroundTime += turnaroundTime;
                    totalWaitingTime += waitingTime;

                    // Display information about the terminated process
                    printf("\tP%d | Started at %d | Ended at %d | Remaining Time: %d ",
                           currentProcess.pid, newEntry->start, time, currentProcess.execution);
                    printf("\033[38;2;249;16;16m => P%d terminated \033[0m\n", currentProcess.pid);

                    printf("\033[1;34m \tP%d | Turnaround Time: %d | Waiting Time: %d \033[0m \n",
                           currentProcess.pid, turnaroundTime, waitingTime);
                }
                else {
                    // The process still has execution time remaining after the quantum
                    time += quantum;
                    currentProcess.execution -= quantum;

                    // Create a new process with the remaining execution time
                    remainingProcess = createProcess(currentProcess.pid, currentProcess.arrival,
                                                     currentProcess.execution, currentProcess.priority);
                    ajouterP(L, time, remainingProcess);

                    // Display information about the process after using the quantum
                    printf("\tP%d | Started at %d | Ended at %d | Remaining Time: %d\n",
                           currentProcess.pid, newEntry->start, time, currentProcess.execution);
                }

                // Update the end time for the Gantt chart entry and add it to the Gantt chart linked list
                newEntry->end = time;
                GanttChart = addExec(GanttChart, newEntry);
                i++;
            }
        }

        printf("\033[1;32m\ntime %d :\033[0m", time);

    }

    // Calculate and display average turnaround time and average waiting time
    float avgTurnaroundTime = (float)totalTurnaroundTime / nbProcess;
    float avgWaitingTime = (float)totalWaitingTime / nbProcess;

    printf("\n\n\033[1;32m Average Turnaround Time: %.2f \033[0m\n", avgTurnaroundTime);
    printf("\033[1;32m Average Waiting Time: %.2f \033[0m\n\n", avgWaitingTime);

    return GanttChart;
```

2.3. Benefits of metrics

Implementing metrics in process scheduling systems offers several benefits, providing insights into the performance, efficiency, and fairness of the scheduling algorithms employed. Here are key advantages of incorporating metrics into process scheduling systems:

Performance Evaluation:
Identifying Bottlenecks: Metrics help in identifying potential bottlenecks and performance issues in the scheduling system. By analyzing metrics like CPU utilization and throughput, administrators can pinpoint areas that need improvement.

Optimization Opportunities:
Data-Driven Decision Making: Metrics provide quantifiable data that allows for data-driven decision-making. System administrators and developers can analyze these metrics to optimize scheduling algorithms, parameter configurations, and overall system performance.

Resource Utilization:
Efficient Resource Allocation: Metrics such as CPU utilization and resource waiting times enable better understanding of resource usage patterns. This information can be leveraged to allocate resources more efficiently, reducing idle time and enhancing overall system throughput.

Fairness and Responsiveness:
Assessing Fairness: Metrics help in evaluating the fairness of a scheduling algorithm. For example, analyzing waiting times and turnaround times allows administrators to assess how equitably resources are distributed among processes. This is particularly important for systems where fairness is a priority.

Identifying Anomalies:
Early Detection of Issues: Metrics can serve as early indicators of system anomalies or inefficiencies. Sudden spikes in waiting times, unexpected drops in throughput, or other unusual patterns can be detected through continuous monitoring, allowing for proactive problem resolution.

Tuning Parameters:
Fine-Tuning of Scheduling Parameters: Metrics aid in the fine-tuning of scheduling parameters. For instance, adjusting time quantums in Round Robin scheduling or setting priority thresholds in Multilevel Queue Scheduling can be informed by analyzing waiting times and overall system responsiveness.

3. Conclusion

The conclusion of this sprint marks a pivotal moment in our development journey as we successfully integrated both makefiles and comprehensive metrics into our system. The adoption of makefiles has brought about a paradigm shift in our build process, introducing automation that not only expedites compilation and linking but also enhances project organization and maintainability. The makefile structure, with its rules, dependencies, and variables, now forms a robust foundation for our development workflow, fostering consistency and adaptability across different environments.

Simultaneously, the incorporation of metrics into our system represents a strategic decision that aligns with our commitment to continuous improvement. These metrics, ranging from CPU utilization and throughput to waiting times and turnaround times, serve as invaluable instruments for evaluating the performance and efficiency of our process scheduling algorithms. With a data-driven approach, we can now pinpoint optimization opportunities, assess the impact of scheduling changes, and proactively address potential bottlenecks or anomalies.

General Conclusion

In conclusion, the successful completion of this project, centered around the design and development of a robust process scheduler, marks a significant achievement in advancing the efficiency and functionality of our computing system. The incorporation of diverse scheduling algorithms, including FIFO, Shortest Remaining Time (SRT), Round Robin, Priority, and Multilevel, showcases our commitment to versatility and adaptability to varying computational demands.

The implementation of makefiles adds a layer of automation to our development workflow, streamlining the compilation and linking processes. This not only enhances the maintainability of our codebase but also facilitates a more systematic approach to project management, empowering developers to focus on core functionalities rather than manual build tasks.

Furthermore, the integration of different metrics to measure process execution, including CPU utilization, throughput, waiting times, and turnaround times, positions us to gain valuable insights into the performance and responsiveness of our system. These metrics provide a quantitative basis for evaluating the efficacy of each scheduling algorithm, allowing us to fine-tune parameters and make informed decisions about algorithm selection based on specific use cases or project requirements.

As we reflect on the collaborative efforts invested in this project, we recognize the multidimensional impact on our system's overall performance. The comprehensive scheduler, coupled with the integration of makefiles and metrics, underscores our dedication to creating a dynamic and responsive computing environment. This project lays a foundation for future optimizations, iterative improvements, and the incorporation of additional features, ensuring that our system remains adaptable and efficient in the face of evolving computational challenges. The journey from conceptualization to implementation exemplifies our team's prowess in system design and the successful execution of a project that significantly contributes to the advancement of our computing infrastructure.