1ING02

SEA - Project

Imen Mraidi
Eya Bouden
Âlaa Zekri
Mohamed El Arbi Werghi
Oussama Elhaddad

December 10, 2023

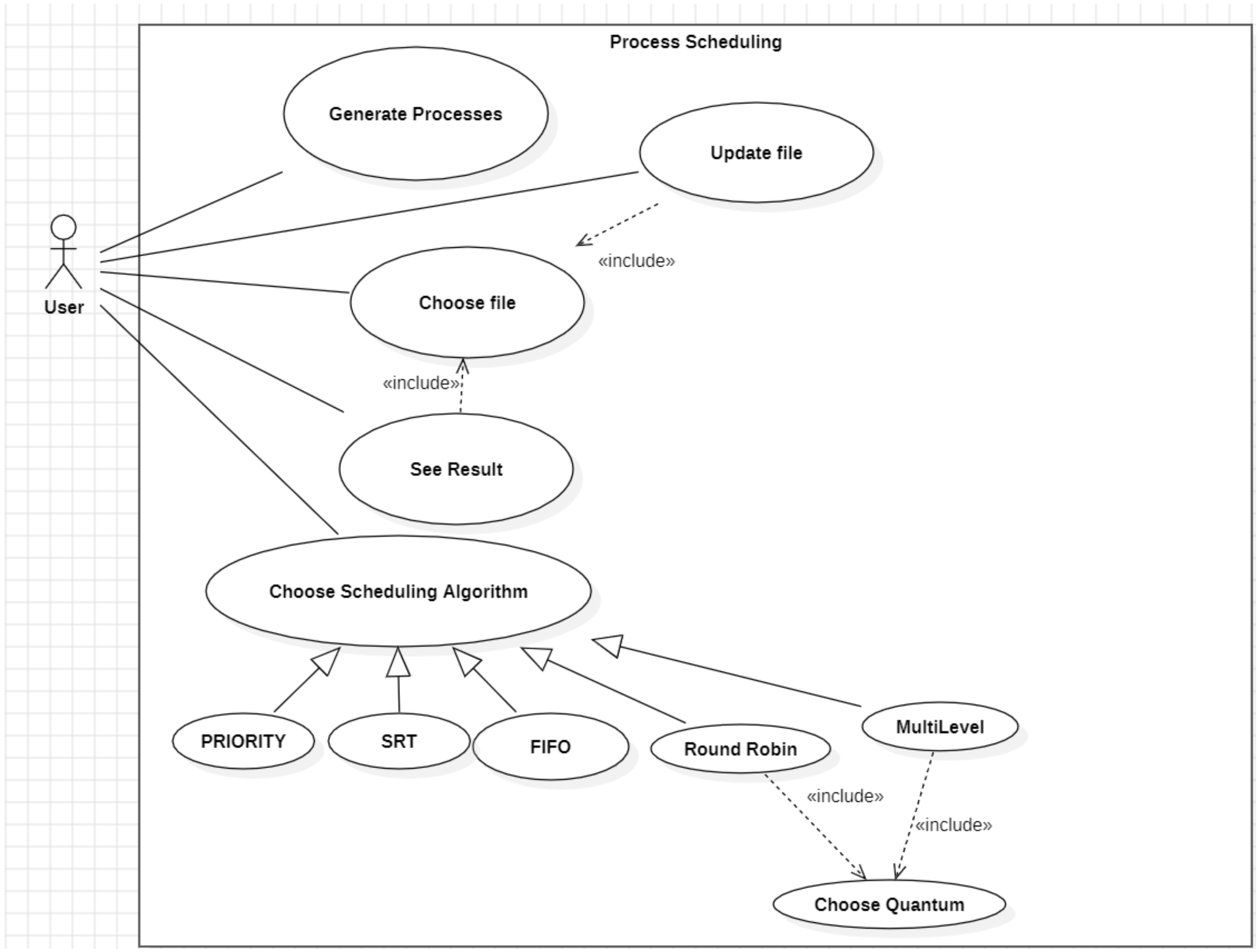
1- Product Backlog

ID	User story	Priority	Estimation
1	As a user, I can generate tables of random processes.	5	2 days
2	As a user, I want to select a scheduling algorithm.	5	7 days
3	As a user, I want to parameterize my random parameter intervals, choose the number of processes to be generated, and specify the location to save the file.	3	4 days
4	As a user, I want to consult my generated file results.	3	1 day
5	As a user, I want to update my file results.	1	1 day
6	As a user, I want to see the average waiting time, total execution time results.	4	3 days
7	As a user, I want to be able to visualize simulation results in the form of a gantt chart.	2	10days

2- Sprints planification

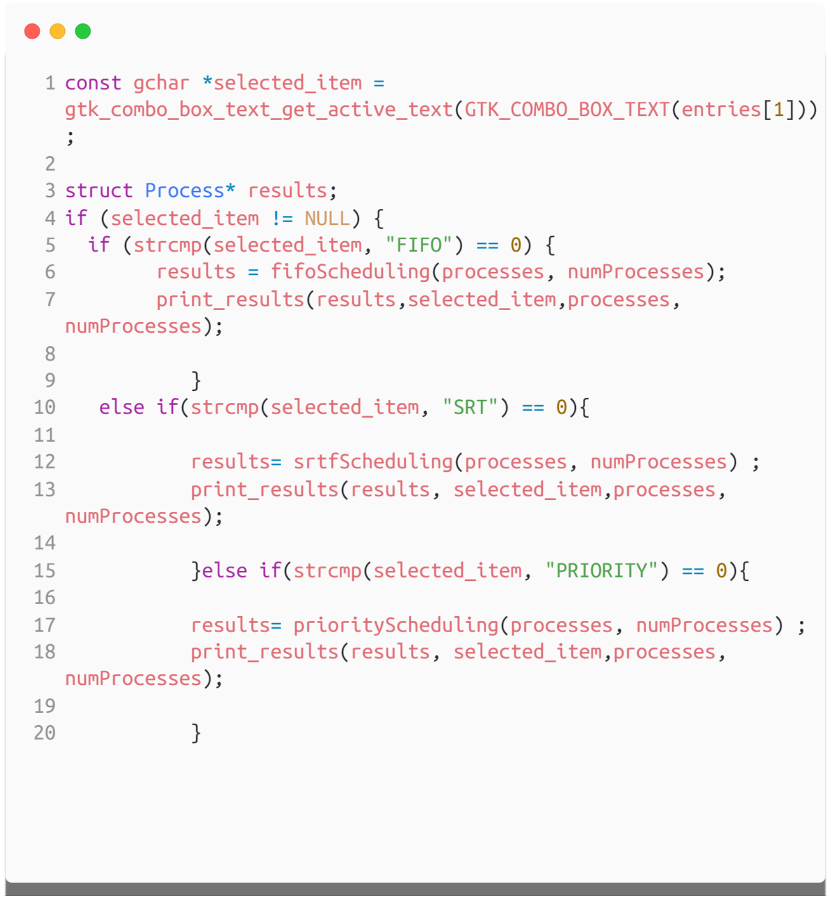
ID	Objectif	ID User story
1	Random process generation.	1
2	Implementation of scheduling algorithms.	2
3	Simulation parameter configuration.	3 & 4
4	Performance metric collection.	6
5	Results visualization.	5 & 7

3- USE CASE


Process Scheduling
User
Generate Processes
Update file
«include»
Choose file
«include»
See Result
Choose Scheduling Algorithm
PRIORITY
SRT
FIFO
Round Robin
MultiLevel
«include»
«include»
Choose Quantum

4- Scheduling Algorithm Selection

To choose the right algorithm, we used the GtkComboBox. It displays a popup which allows the user to make their choice. And then we retrieve that value and execute the according scheduling algorithm.

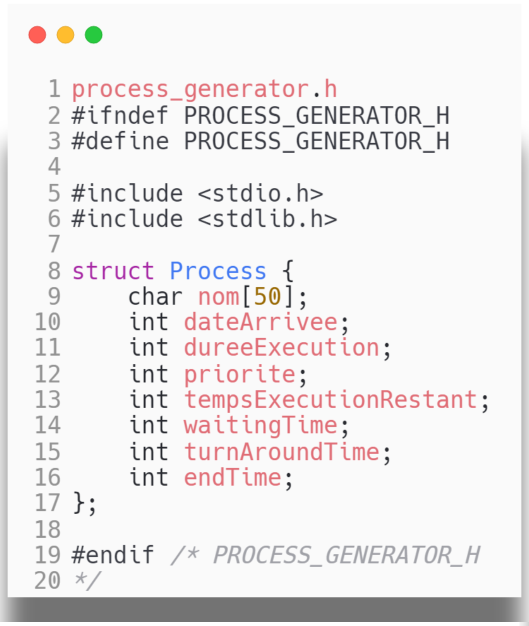


```c
const gchar *selected_item = gtk_combo_box_text_get_active_text(GTK_COMBO_BOX_TEXT(entries[1]));

struct Process* results;
if (selected_item != NULL) {
  if (strcmp(selected_item, "FIFO") == 0) {
        results = fifoScheduling(processes, numProcesses);
        print_results(results,selected_item,processes, numProcesses);

           }
   else if(strcmp(selected_item, "SRT") == 0){

           results= srtfScheduling(processes, numProcesses) ;
           print_results(results, selected_item,processes, numProcesses);

           }else if(strcmp(selected_item, "PRIORITY") == 0){

           results= priorityScheduling(processes, numProcesses) ;
           print_results(results, selected_item,processes, numProcesses);

           }
```

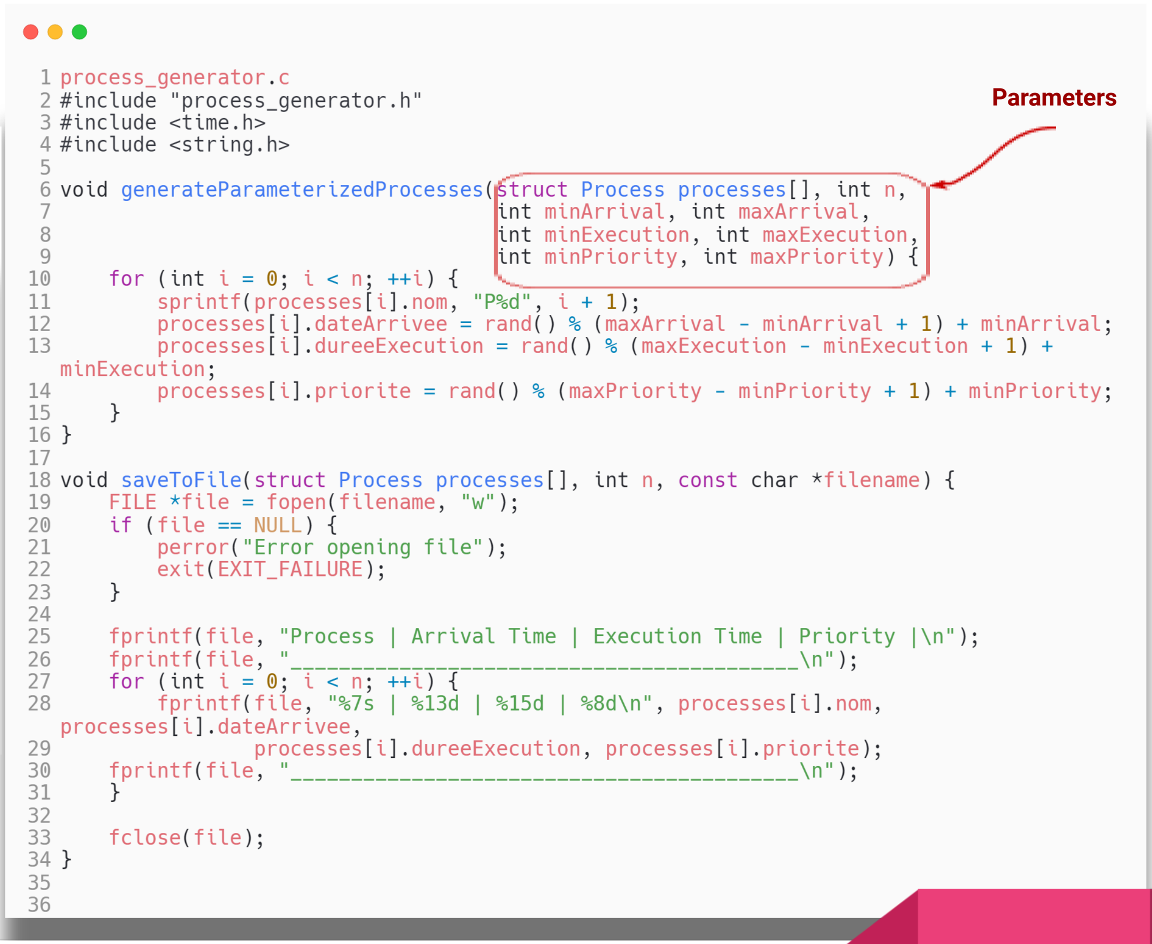
5- Process generator

Before creating our process generator program, we decided how our data structure will look. Each line of the config file (process) will be represented by a process structure as indicated in the screenshot on the left.

Then we started creating the process generator algorithm that will create a table of randomly parametrized process (screenshot on the right)



```c
process_generator.h
#ifndef PROCESS_GENERATOR_H
#define PROCESS_GENERATOR_H

#include <stdio.h>
#include <stdlib.h>

struct Process {
    char nom[50];
    int dateArrivee;
    int dureeExecution;
    int priorite;
    int tempsExecutionRestant;
    int waitingTime;
    int turnAroundTime;
    int endTime;
};

#endif /* PROCESS_GENERATOR_H
*/
```

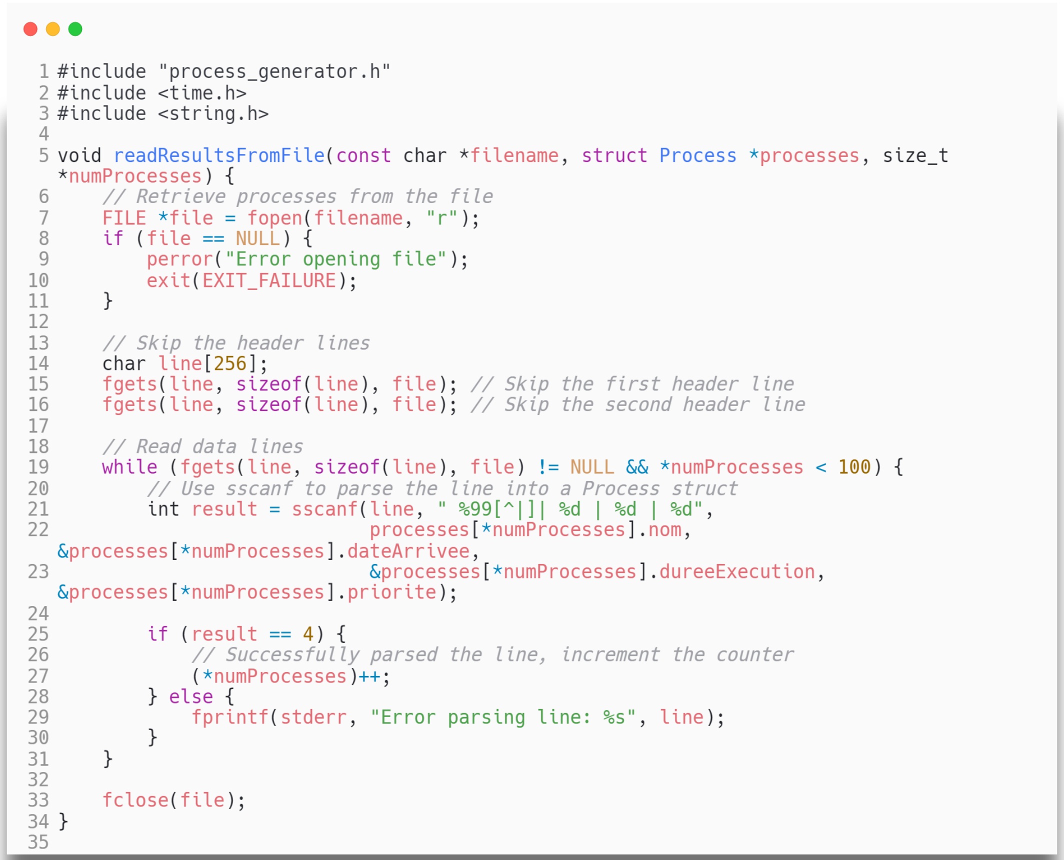


```c
process_generator.c
#include "process_generator.h"
#include <time.h>
#include <string.h>

void generateParameterizedProcesses(struct Process processes[], int n,
                                    int minArrival, int maxArrival,
                                    int minExecution, int maxExecution,
                                    int minPriority, int maxPriority) {
    for (int i = 0; i < n; ++i) {
        sprintf(processes[i].nom, "P%d", i + 1);
        processes[i].dateArrivee = rand() % (maxArrival - minArrival + 1) + minArrival;
        processes[i].dureeExecution = rand() % (maxExecution - minExecution + 1) +
minExecution;
        processes[i].priorite = rand() % (maxPriority - minPriority + 1) + minPriority;
    }
}

void saveToFile(struct Process processes[], int n, const char *filename) {
    FILE *file = fopen(filename, "w");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    fprintf(file, "Process | Arrival Time | Execution Time | Priority |\n");
    fprintf(file, "__________________________________________\n");
    for (int i = 0; i < n; ++i) {
        fprintf(file, "%7s | %13d | %15d | %8d\n", processes[i].nom,
processes[i].dateArrivee,
                processes[i].dureeExecution, processes[i].priorite);
    fprintf(file, "__________________________________________\n");
    }

    fclose(file);
}
```

6- Reading File Contents

The purpose of this function is extract relevant process data from a specified text file and use it when needed (such as displaying results while clicking on “See Results” button , or generating a gantt chart).

It takes the file path, an array of struct Process objects, and a pointer to a variable representing the number of processes as input parameters.



```c
#include "process_generator.h"
#include <time.h>
#include <string.h>

void readResultsFromFile(const char *filename, struct Process *processes, size_t
*numProcesses) {
    // Retrieve processes from the file
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    // Skip the header lines
    char line[256];
    fgets(line, sizeof(line), file); // Skip the first header line
    fgets(line, sizeof(line), file); // Skip the second header line

    // Read data lines
    while (fgets(line, sizeof(line), file) != NULL && *numProcesses < 100) {
        // Use sscanf to parse the line into a Process struct
        int result = sscanf(line, " %99[^|]| %d | %d | %d",
                            processes[*numProcesses].nom,
&processes[*numProcesses].dateArrivee,
                            &processes[*numProcesses].dureeExecution,
&processes[*numProcesses].priorite);

        if (result == 4) {
            // Successfully parsed the line, increment the counter
            (*numProcesses)++;
        } else {
            fprintf(stderr, "Error parsing line: %s", line);
        }
    }

    fclose(file);
}

```

7- MAKEFILE

Every programmer needs a build system, and often, multiple build systems which prove to be beneficial for various projects. In this context, we opt for one of the most universal build systems available: make.

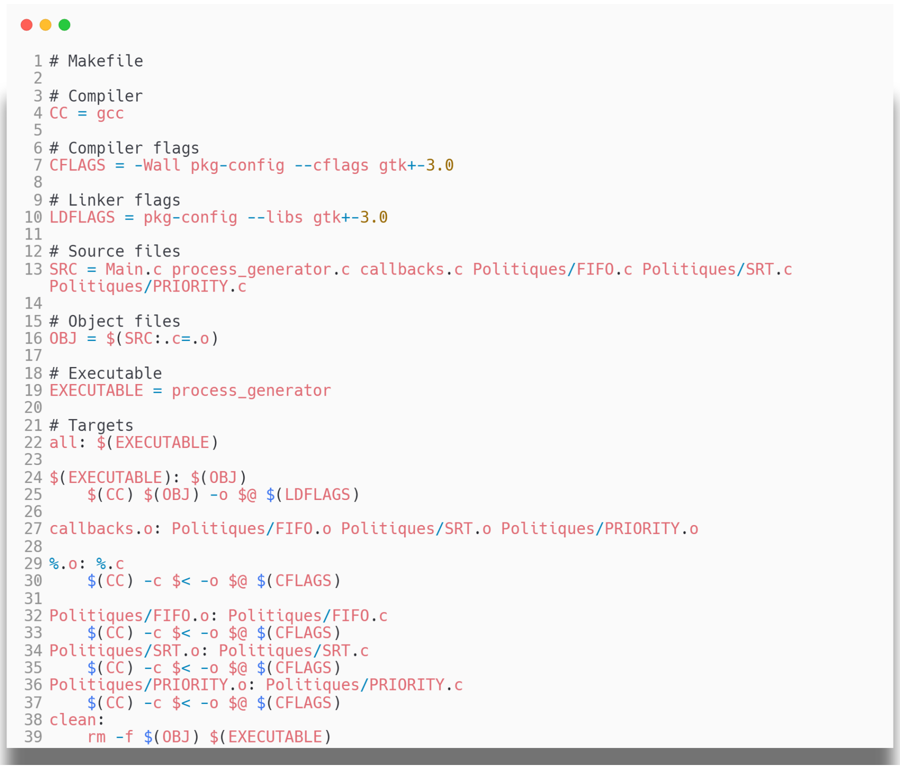
Essentially, make is a script that outlines the compilation and construction process of a project.

Here's a quick summary on how we create it:

1- Open Notepad or any text editor.

2- Create our own Makefile code.

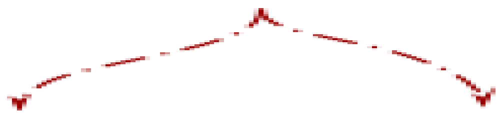
3- Save the file with the name "Makefile" (without any file extension).



```
# Makefile

# Compiler
CC = gcc

# Compiler flags
CFLAGS = -Wall pkg-config --cflags gtk+-3.0

# Linker flags
LDFLAGS = pkg-config --libs gtk+-3.0

# Source files
SRC = Main.c process_generator.c callbacks.c Politiques/FIFO.c Politiques/SRT.c Politiques/PRIORITY.c

# Object files
OBJ = $(SRC:.c=.o)

# Executable
EXECUTABLE = process_generator

# Targets
all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJ)
    $(CC) $(OBJ) -o $@ $(LDFLAGS)

callbacks.o: Politiques/FIFO.o Politiques/SRT.o Politiques/PRIORITY.o

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

Politiques/FIFO.o: Politiques/FIFO.c
    $(CC) -c $< -o $@ $(CFLAGS)
Politiques/SRT.o: Politiques/SRT.c
    $(CC) -c $< -o $@ $(CFLAGS)
Politiques/PRIORITY.o: Politiques/PRIORITY.c
    $(CC) -c $< -o $@ $(CFLAGS)
clean:
    rm -f $(OBJ) $(EXECUTABLE)
```

8- Politiques

In our project, we implemented 5 algorithms:



1 non preemptive:

FIFO

4 preemptive:

Round Robin | SRT | Preemptive priority | Multilevel

The data structure used to store the processes is an array of structures.

```
`` struct Process* results = malloc(s * sizeof(struct Process)); ``
```

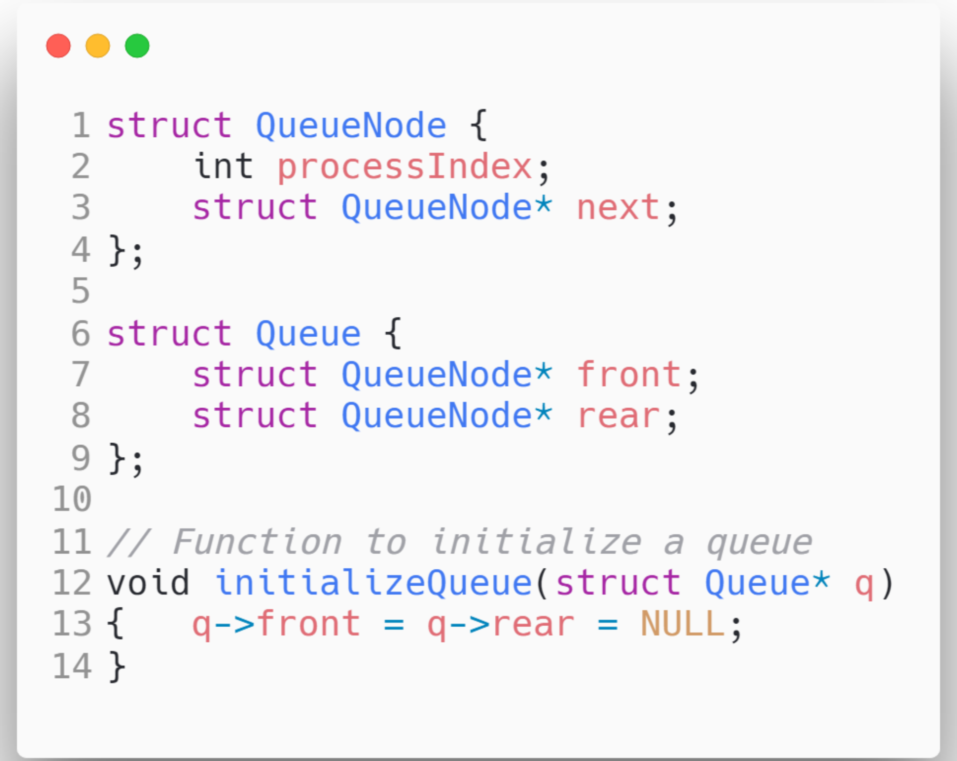
We used a dynamic array (results) that will be allocated using malloc to store the order in which processes are executed(based on the algorithm used). This array is later returned by the function.
E.g of FIFO: the order of execution is determined by the FIFO principle, where processes are executed in the order of their arrival time.
→ We used an array of structures to represent and manipulate processes.

The array in RR stores information about the processes as they are scheduled with the start and end times of each process.

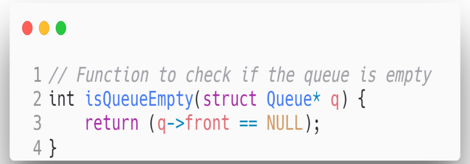
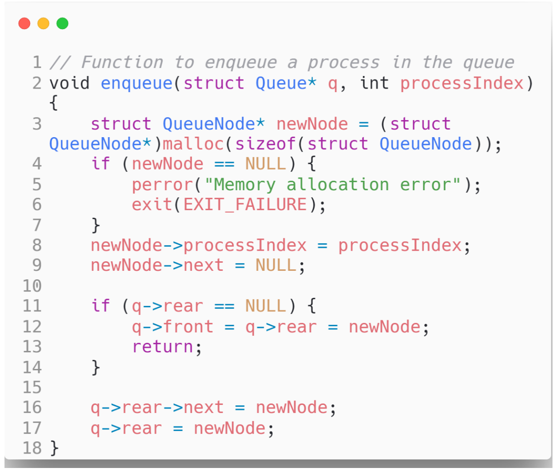
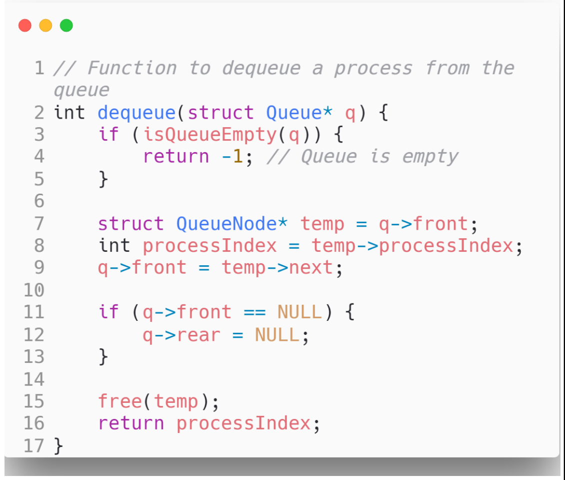
```
``for(int j=ganttIndex;j<currentTime;j++)
   results[j]=processes[index];              ``
```

The RR iterates through processes, selecting the one with the earliest arrival time and executes it for a fixed quantum(selected by the user). If a process isn't completed within the quantum, it's rescheduled, and the array records the process entries in the Gantt chart.

Finally, for the last scheduling algorithm “MULTILEVEL” we implemented the Queue data structure to manage processes based on their priority levels.



```c
struct QueueNode {
    int processIndex;
    struct QueueNode* next;
};

struct Queue {
    struct QueueNode* front;
    struct QueueNode* rear;
};

// Function to initialize a queue
void initializeQueue(struct Queue* q)
{   q->front = q->rear = NULL;
}
```

IsQueueEmpty function: check if the queue is empty or not	Enqueue function: add a process into our list	Dequeue function: remove a process from our list
		

```c
// Function to check if the queue is empty
int isQueueEmpty(struct Queue* q) {
    return (q->front == NULL);
}
```

```c
// Function to enqueue a process in the queue
void enqueue(struct Queue* q, int processIndex)
{
    struct QueueNode* newNode = (struct
QueueNode*)malloc(sizeof(struct QueueNode));
    if (newNode == NULL) {
        perror("Memory allocation error");
        exit(EXIT_FAILURE);
    }
    newNode->processIndex = processIndex;
    newNode->next = NULL;

    if (q->rear == NULL) {
        q->front = q->rear = newNode;
        return;
    }

    q->rear->next = newNode;
    q->rear = newNode;
}
```

```c
// Function to dequeue a process from the
queue
int dequeue(struct Queue* q) {
    if (isQueueEmpty(q)) {
        return -1; // Queue is empty
    }

    struct QueueNode* temp = q->front;
    int processIndex = temp->processIndex;
    q->front = temp->next;

    if (q->front == NULL) {
        q->rear = NULL;
    }

    free(temp);
    return processIndex;
}
```

9- Perspectives:

Diversified Scheduling Algorithms: (such as LIFO, MULTILEVEL Q,dynamic Priority...)

Interface Upgrade with Detailed Statistics: (include detailed statistics on process execution, resource utilization, and algorithm efficiency)