

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

December 10, 2023

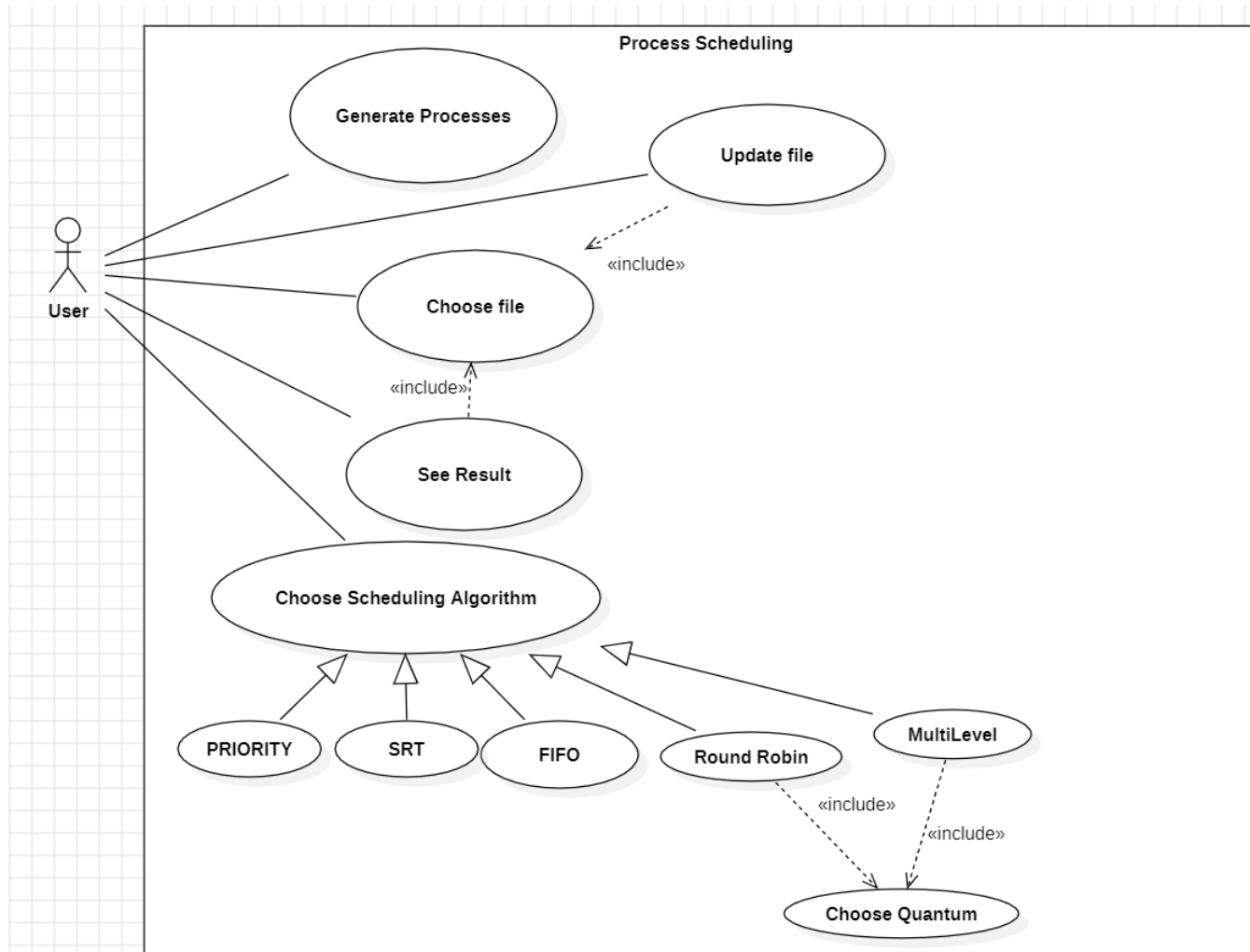
## 1- Product Backlog

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



## 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.

```
1 const gchar *selected_item =  
    gtk_combo_box_text_get_active_text(GTK_COMBO_BOX_TEXT(entries[1]))  
    ;  
2  
3 struct Process* results;  
4 if (selected_item != NULL) {  
5     if (strcmp(selected_item, "FIFO") == 0) {  
6         results = fifoScheduling(processes, numProcesses);  
7         print_results(results,selected_item,processes,  
            numProcesses);  
8     }  
9  
10    else if(strcmp(selected_item, "SRT") == 0){  
11        results= srtfScheduling(processes, numProcesses) ;  
12        print_results(results, selected_item,processes,  
            numProcesses);  
13  
14    }else if(strcmp(selected_item, "PRIORITY") == 0){  
15        results= priorityScheduling(processes, numProcesses) ;  
16        print_results(results, selected_item,processes,  
            numProcesses);  
17  
18    }  
19  
20 }
```

## 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 struct as follows (screenshot on your left side)

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

```

1 process_generator.h
2 #ifndef PROCESS_GENERATOR_H
3 #define PROCESS_GENERATOR_H
4
5 #include <stdio.h>
6 #include <stdlib.h>
7
8 struct Process {
9     char nom[50];
10    int dateArrivee;
11    int dureeExecution;
12    int priorite;
13    int tempsExecutionRestant;
14    int waitingTime;
15    int turnAroundTime;
16    int endTime;
17 };
18
19 #endif /* PROCESS_GENERATOR_H
20 */

```

```

1 process_generator.c
2 #include "process_generator.h"
3 #include <time.h>
4 #include <string.h>
5
6 void generateParameterizedProcesses(struct Process processes[], int n,
7                                     int minArrival, int maxArrival,
8                                     int minExecution, int maxExecution,
9                                     int minPriority, int maxPriority) {
10     for (int i = 0; i < n; ++i) {
11         sprintf(processes[i].nom, "P%d", i + 1);
12         processes[i].dateArrivee = rand() % (maxArrival - minArrival + 1) + minArrival;
13         processes[i].dureeExecution = rand() % (maxExecution - minExecution + 1) +
14         minExecution;
15         processes[i].priorite = rand() % (maxPriority - minPriority + 1) + minPriority;
16     }
17 }
18
19 void saveToFile(struct Process processes[], int n, const char *filename) {
20     FILE *file = fopen(filename, "w");
21     if (file == NULL) {
22         perror("Error opening file");
23         exit(EXIT_FAILURE);
24     }
25     fprintf(file, "Process | Arrival Time | Execution Time | Priority |\n");
26     fprintf(file, "-----\n");
27     for (int i = 0; i < n; ++i) {
28         fprintf(file, "%7s | %13d | %15d | %8d\n", processes[i].nom,
29         processes[i].dateArrivee,
30         processes[i].dureeExecution, processes[i].priorite);
31         fprintf(file, "-----\n");
32     }
33     fclose(file);
34 }
35
36

```

**Parameters**

## 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.

```

1 #include "process_generator.h"
2 #include <time.h>
3 #include <string.h>
4
5 void readResultsFromFile(const char *filename, struct Process *processes, size_t
   *numProcesses) {
6     // Retrieve processes from the file
7     FILE *file = fopen(filename, "r");
8     if (file == NULL) {
9         perror("Error opening file");
10        exit(EXIT_FAILURE);
11    }
12
13    // Skip the header lines
14    char line[256];
15    fgets(line, sizeof(line), file); // Skip the first header line
16    fgets(line, sizeof(line), file); // Skip the second header line
17
18    // Read data lines
19    while (fgets(line, sizeof(line), file) != NULL && *numProcesses < 100) {
20        // Use sscanf to parse the line into a Process struct
21        int result = sscanf(line, "%99[^\n]| %d | %d | %d",
22                            processes[*numProcesses].nom,
23                            &processes[*numProcesses].dateArrivee,
24                            &processes[*numProcesses].dureeExecution,
25                            &processes[*numProcesses].priorite);
26
27        if (result == 4) {
28            // Successfully parsed the line, increment the counter
29            (*numProcesses)++;
30        } else {
31            fprintf(stderr, "Error parsing line: %s", line);
32        }
33    }
34    fclose(file);
35}

```

## 7- MAKEFILE

Every programmer needs a build system, and often, multiple build systems prove 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:

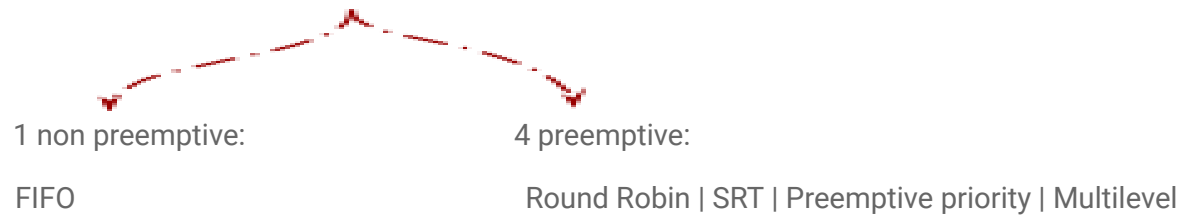
- 1- Open Notepad or any text editor.
  - 2- Create our own Makefile code.
  - 3- Save the file with the name "Makefile"
- (without any file extension).

```

1 # Makefile
2
3 # Compiler
4 CC = gcc
5
6 # Compiler flags
7 CFLAGS = -Wall pkg-config --cflags gtk+-3.0
8
9 # Linker flags
10 LDFLAGS = pkg-config --libs gtk+-3.0
11
12 # Source files
13 SRC = Main.c process_generator.c callbacks.c Politiques/FIFO.c Politiques/SRT.c
    Politiques/PRIORITY.c
14
15 # Object files
16 OBJ = $(SRC:.c=.o)
17
18 # Executable
19 EXECUTABLE = process_generator
20
21 # Targets
22 all: $(EXECUTABLE)
23
24 $(EXECUTABLE): $(OBJ)
25     $(CC) $(OBJ) -o $@ $(LDFLAGS)
26
27 callbacks.o: Politiques/FIFO.o Politiques/SRT.o Politiques/PRIORITY.o
28
29 %.o: %.c
30     $(CC) -c $< -o $@ $(CFLAGS)
31
32 Politiques/FIFO.o: Politiques/FIFO.c
33     $(CC) -c $< -o $@ $(CFLAGS)
34 Politiques/SRT.o: Politiques/SRT.c
35     $(CC) -c $< -o $@ $(CFLAGS)
36 Politiques/PRIORITY.o: Politiques/PRIORITY.c
37     $(CC) -c $< -o $@ $(CFLAGS)
38 clean:
39     rm -f $(OBJ) $(EXECUTABLE)
  
```

## 8- Politiques

In our project, we implemented 5 algorithms:



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.

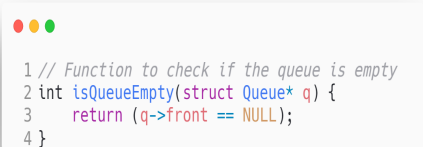
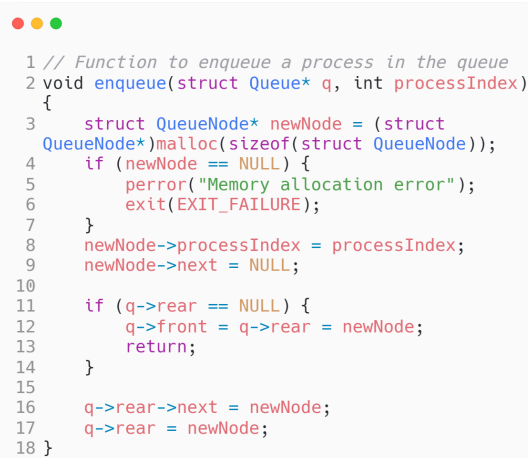
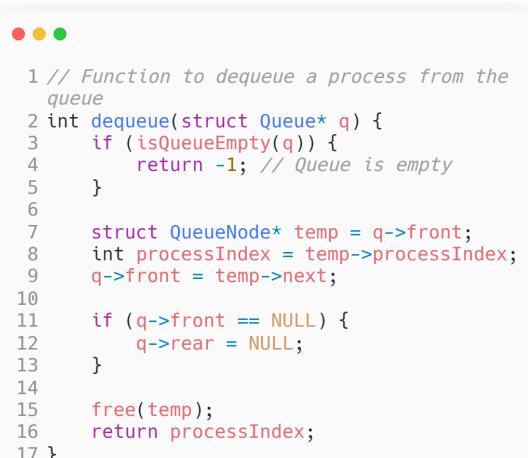
```
``for(int j=ganttIndex;j<currentTime;j++)
  results[j]=processes[index];
``
```



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.

```
1 struct QueueNode {
2     int processIndex;
3     struct QueueNode* next;
4 };
5
6 struct Queue {
7     struct QueueNode* front;
8     struct QueueNode* rear;
9 };
10
11 // Function to initialize a queue
12 void initializeQueue(struct Queue* q)
13 {     q->front = q->rear = NULL;
14 }
```

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
 <pre> 1 // Function to check if the queue is empty 2 int isQueueEmpty(struct Queue* q) { 3     return (q-&gt;front == NULL); 4 } </pre>	 <pre> 1 // Function to enqueue a process in the queue 2 void enqueue(struct Queue* q, int processIndex) 3 { 4     struct QueueNode* newNode = (struct 5     QueueNode*)malloc(sizeof(struct QueueNode)); 6     if (newNode == NULL) { 7         perror("Memory allocation error"); 8         exit(EXIT_FAILURE); 9     } 10    newNode-&gt;processIndex = processIndex; 11    newNode-&gt;next = NULL; 12    if (q-&gt;rear == NULL) { 13        q-&gt;front = q-&gt;rear = newNode; 14        return; 15    } 16    q-&gt;rear-&gt;next = newNode; 17    q-&gt;rear = newNode; 18 } </pre>	 <pre> 1 // Function to dequeue a process from the 2 // queue 3 int dequeue(struct Queue* q) { 4     if (isQueueEmpty(q)) { 5         return -1; // Queue is empty 6     } 7     struct QueueNode* temp = q-&gt;front; 8     int processIndex = temp-&gt;processIndex; 9     q-&gt;front = temp-&gt;next; 10 11     if (q-&gt;front == NULL) { 12         q-&gt;rear = NULL; 13     } 14 15     free(temp); 16     return processIndex; 17 } </pre>

## 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)