ECOLE CENTRALE DE NANTES

MASTER CONTROL AND ROBOTICS

EMBEDDED REAL-TIME SYSTEMS SECTION

Simulation of a preemptive fixed-priority Scheduler

Project Report

Author
Erwin LEJEUNE

Superviser
Maryline CHETTO

November 15, 2020

Contents

Chapter 1

Introduction

Initially, the project's goal was to simulate the behavior of a Preemptive Fixed Priority Scheduler (Rate Monotonic) only, with tasks strictly restricted with the constraint $T(period) = D(deadline)$. I decided to leave it customizable to the user by not hard-coding this constraint, and this way allow the project to be scalable to other scheduling policies. The language used is C++, for its object-oriented obvious advantage over C. I initially wanted to use a personal project similar to this one last year that I wrote in Python, but decided that it would be much more gratifying for me to use standards closer to the industry's.

The report will be divided in a few parts: first the Architecture to describe the general idea behind how I built the simulator and why I made the choices that I made, then the typical automaton for the simulator's usage, and finally we will dig into the details of the methods and functions. All steps have been completed, from data acquisition to the display of metrics.

The project sources can be found here.

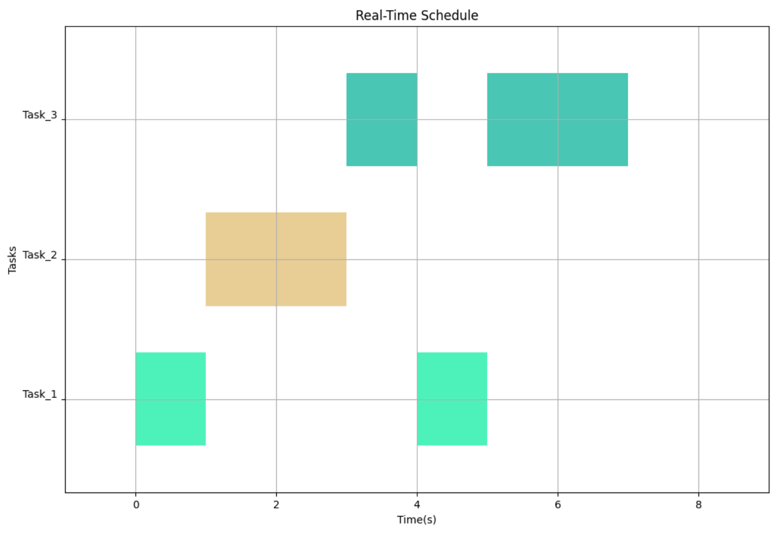


Figure 1.1: Simulated Gantt Plot for 3 Tasks

Chapter 2

Architecture

I used Classes to define structures that needed to perform actions, whether it was computation or calls from other structures. The first class built was the Task class, that obviously was the first stone to the project as it is the structure at the very low level of our architecture.

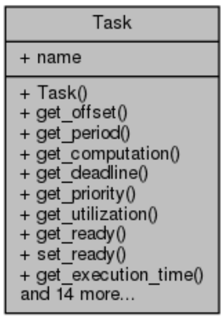


Figure 2.1: Task Class

The task structure has a lot of getter and setter methods because of the amount of protected attributes it has. Initially I only implemented the Offset, Period, Computation (WCET) and Deadline and there respective methods, but as I kept developing the project I realized that it needed more for the scheduling part : Id, Priority, Utilization, Ready, Current Execution... It is the structure you fill when registering a task to the task set, which also has its own structure as followed.

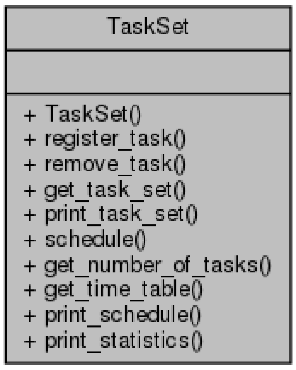


Figure 2.2: Task Set Class

This is the cornerstone of the project as it handles the entire scheduling simulation. It is used to register and remove tasks, compute the hyper period, define if the necessary and sufficient conditions are met depending on the chosen scheduling policy, and finally schedule the registered tasks on a time table. In section , we established that the project was scalable: indeed, it is built for anyone to easily add bricks to it and it has high readability. To add a scheduling policy, one simply has to add a Class for the desired scheduling algorithm and implement the corresponding methods, as done for the Rate Monotonic algorithm:

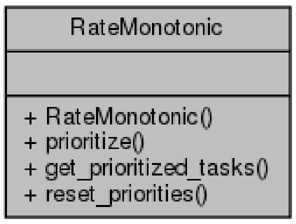


Figure 2.3: Rate Monotonic Class

The contributor also would have to fill in the switch case for prioritizing and scheduling in the Task Set Class (fig. 2.1) and nothing else.

The statistics are computed for each Task and filled in the Statistics structure of the Task Class as follows:

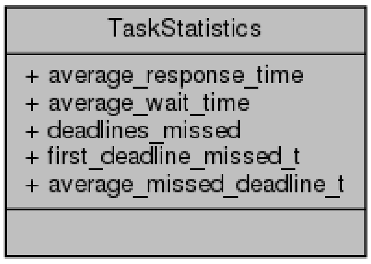


Figure 2.4: Task Statistics Structure

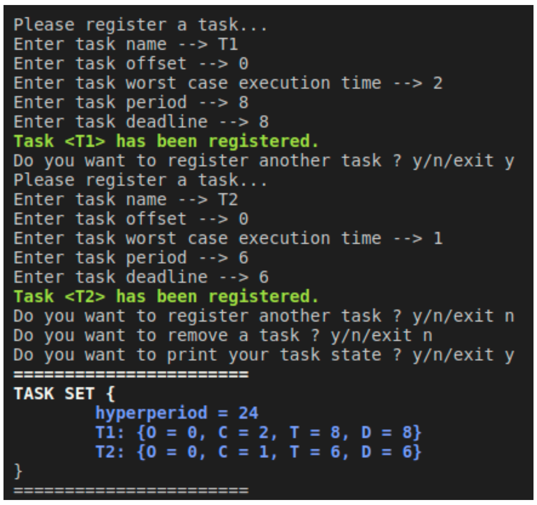
Statistics are computed using the Time Table vector structure, that is filled after setting tasks' priorities: the highest priority task cannot be preempted, so we schedule it without delay first at its defined ready time. The second highest priority task will be schedule at the first available task slot after its ready time is reached for each period, and so on. We ensure real-time constraints using that polling technique, even though it isn't optimal.

Chapter 3

Automaton

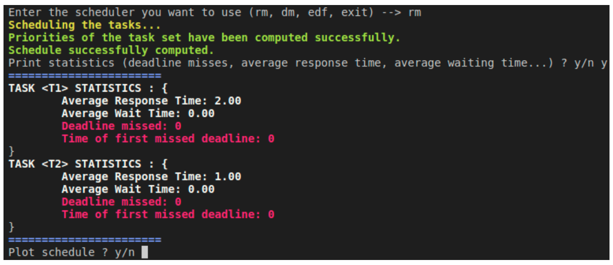
3.1 User Interface

In order to build a robust user state machine, I first wrote a pseudo code of what behavior I wanted to have, which I then changed here and there in order to match unpredictable changes in the project. The idea is to be able to follow strictly the states that are defined and still be able to exit at any point safely. The execution must be sequential in that way: Register/Remove, Schedule, Plot/Print Statistics. As you can see it's a very easy usage for the user that don't necessarily know how to use the scheduler or what a scheduler.



```
Please register a task...
Enter task name --> T1
Enter task offset --> 0
Enter task worst case execution time --> 2
Enter task period --> 8
Enter task deadline --> 8
Task <T1> has been registered.
Do you want to register another task ? y/n/exit y
Please register a task...
Enter task name --> T2
Enter task offset --> 0
Enter task worst case execution time --> 1
Enter task period --> 6
Enter task deadline --> 6
Task <T2> has been registered.
Do you want to register another task ? y/n/exit n
Do you want to remove a task ? y/n/exit n
Do you want to print your task state ? y/n/exit y
=======================
TASK SET {
        hyperperiod = 24
        T1: {O = 0, C = 2, T = 8, D = 8}
        T2: {O = 0, C = 1, T = 6, D = 6}
}
=======================
```

Figure 3.1: Registering States



```
Enter the scheduler you want to use (rm, dm, edf, exit) --> rm
Scheduling the tasks...
Priorities of the task set have been computed successfully.
Schedule successfully computed.
Print statistics (deadline misses, average response time, average waiting time...) ? y/n y
=======================
TASK <T1> STATISTICS : {
        Average Response Time: 2.00
        Average Wait Time: 0.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T2> STATISTICS : {
        Average Response Time: 1.00
        Average Wait Time: 0.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
=======================
Plot schedule ? y/n
```

Figure 3.2: Scheduling States

3.2 Pseudo-Code

Algorithm 1: Pseudocode User Usage

```
stop <= false;
state <= 0;
taskSet <= TaskSet();
while not stop do
    if state = 0 then
        taskSet.RegisterTasks();
        if done <= true then
            state <= 1;
        end
    end
    if state = 1 then
        taskSet.RemoveTasks();
        if done <= true then
            state <= 2;
        end
    end
    if state = 2 then
        Input(schedulingPolicy);
        taskSet.Schedule(schedulingPolicy);
        state <= 3;
    end
    if state = 3 then
        Input("print statistics ?");
        taskSet.PrintStatistics(taskSet);
        state <= 4;
    end
    if state = 4 then
        taskSet.PlotGantt();
        state <= 5;
    end
    if state = 5 then
        Input("exit ?");
        if exit = true then
            stop <= true;
        else
            state <= 0;
            taskSet <= TaskSet();
        end
    end
end
```

Chapter 4

Functions and Methods

4.1 Hyper Period

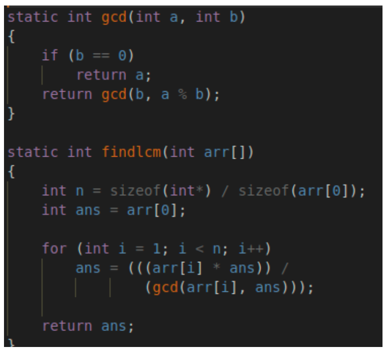
Calculating the hyper period calculation required a few functions that were not built-in for C++. In my implementation, the LCM calculation isn't straight forward: we need to calculate the Greater Common Divisor too.

We know that: $LCM(a, b) = \frac{a*b}{gcd(a,b)}$
We also know that: $LCM(a, b, c) \neq \frac{a*b}{gcd(a,b,c)}$

The main steps of our algorithm are:

- Initialize $currentElem = list[0]$.
- For every elem:

$$\begin{aligned} currentElem &= LCM(list[0], list[1], \ldots\ldots.., list[i-1]) \\ &= LCM(currentElem, list[i]) \\ &= \frac{currentElem \times list[i]}{gcd(currentElem, list[i])} \end{aligned} \tag{4.1}$$



```
static int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

static int findlcm(int arr[])
{
    int n = sizeof(int*) / sizeof(arr[0]);
    int ans = arr[0];

    for (int i = 1; i < n; i++)
        ans = (((arr[i] * ans)) /
                (gcd(arr[i], ans)));

    return ans;
}
```

Figure 4.1: C++ Implementation of LCM

4.2 Rate Monotonic Conditions

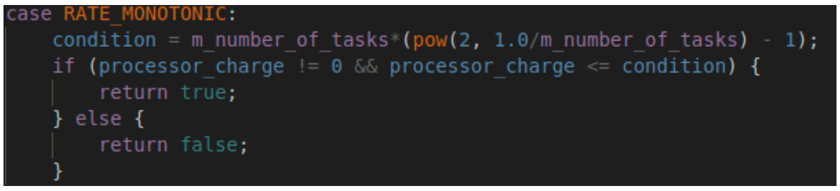
First we check if the Utilization Factor is below 1 (necessary condition):

$$U = \sum_{i}^{n} \frac{Ci}{Ti} <= 1$$

If the condition is not met, then the Task Set is definitely not schedulable and we exit no matter what. If the necessary condition is met, we can compute the sufficient one:

$$U = \sum_{i}^{n} \frac{Ci}{Ti} <= n \times (2^{\frac{1}{n}} - 1)$$

If it is not met, we will still ask the user if he wants to try and schedule it anyway.

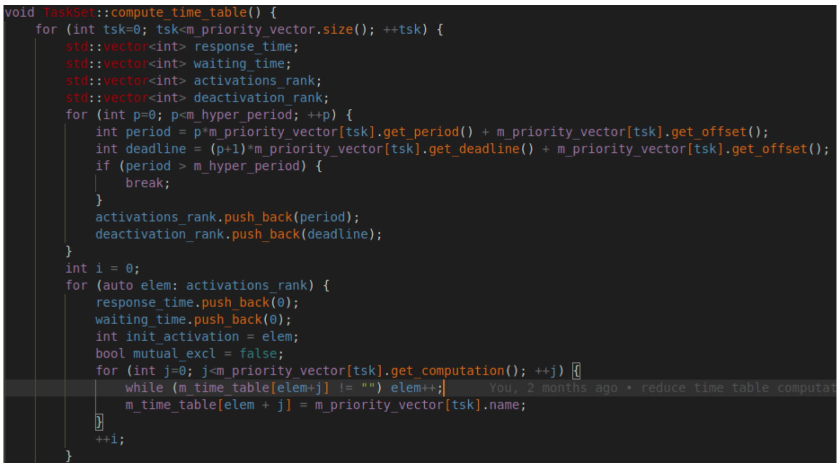


```cpp
case RATE_MONOTONIC:
    condition = m_number_of_tasks*(pow(2, 1.0/m_number_of_tasks) - 1);
    if (processor_charge != 0 && processor_charge <= condition) {
        return true;
    } else {
        return false;
    }
```

Figure 4.2: C++ Implementation of RM Sufficient Condition

4.3 Building the schedule

In order to build the schedule, I created a time table structure that has the length of the hyper period with one element equal to one unit of time. For each task, we compute the ready times and place it in a temporary structure. We know know when the task is going to try and activate (and maybe it won't be able to). Now that we have that information, for each task in the set (highest priority first), we fill the time table from activation to relative computation time when it's available in the time table. That way we have at the end a one line schedule with tasks executions and idle times.

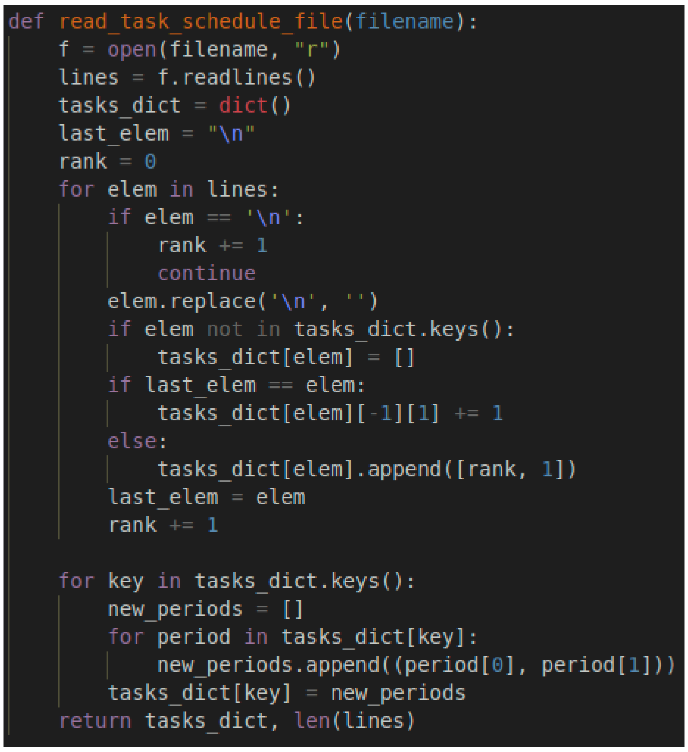


```cpp
void TaskSet::compute_time_table() {
    for (int tsk=0; tsk<m_priority_vector.size(); ++tsk) {
        std::vector<int> response_time;
        std::vector<int> waiting_time;
        std::vector<int> activations_rank;
        std::vector<int> deactivation_rank;
        for (int p=0; p<m_hyper_period; ++p) {
            int period = p*m_priority_vector[tsk].get_period() + m_priority_vector[tsk].get_offset();
            int deadline = (p+1)*m_priority_vector[tsk].get_deadline() + m_priority_vector[tsk].get_offset();
            if (period > m_hyper_period) {
                break;
            }
            activations_rank.push_back(period);
            deactivation_rank.push_back(deadline);
        }
        int i = 0;
        for (auto elem: activations_rank) {
            response_time.push_back(0);
            waiting_time.push_back(0);
            int init_activation = elem;
            bool mutual_excl = false;
            for (int j=0; j<m_priority_vector[tsk].get_computation(); ++j) {
                while (m_time_table[elem+j] != "") elem++;        You, 2 months ago • reduce time table computat
                m_time_table[elem + j] = m_priority_vector[tsk].name;
            }
            ++i;
        }
```

Figure 4.3: Time Table Computation

4.4 Plotting the schedule

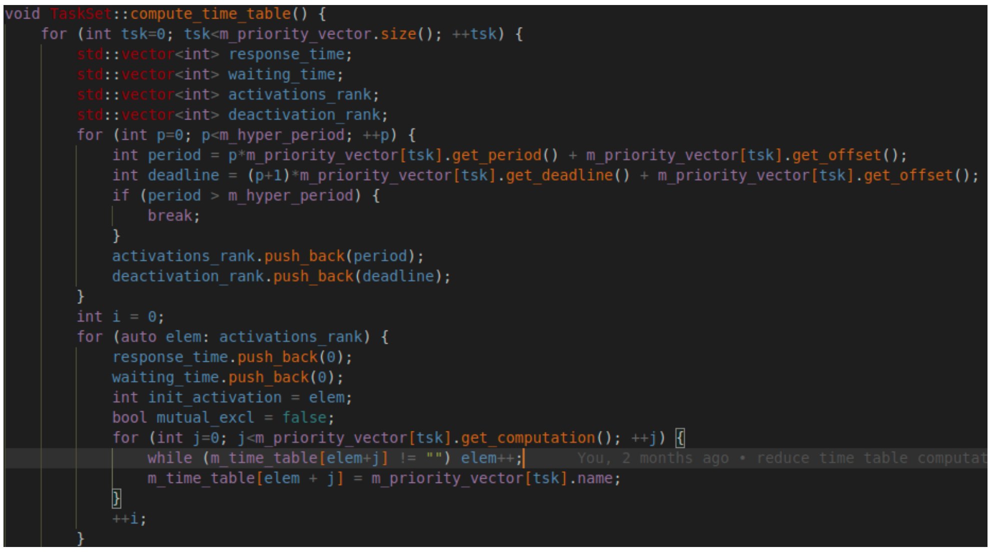
To plot the schedule, I used a Python/C++ Bridge. That way I can call a Python program from C++ and use matplotlib in an easier way than on C++. I stream the time table in a file, and each line is one unit of time with a task id written. Then for each line, I count how many units each task take and at which time each tasks are ready. I can then plot the schedule easily using the horizontal bar plots.



```python
def read_task_schedule_file(filename):
    f = open(filename, "r")
    lines = f.readlines()
    tasks_dict = dict()
    last_elem = "\n"
    rank = 0
    for elem in lines:
        if elem == '\n':
            rank += 1
            continue
        elem.replace('\n', '')
        if elem not in tasks_dict.keys():
            tasks_dict[elem] = []
        if last_elem == elem:
            tasks_dict[elem][-1][1] += 1
        else:
            tasks_dict[elem].append([rank, 1])
        last_elem = elem
        rank += 1

    for key in tasks_dict.keys():
        new_periods = []
        for period in tasks_dict[key]:
            new_periods.append((period[0], period[1]))
        tasks_dict[key] = new_periods
    return tasks_dict, len(lines)
```

Figure 4.4: Python Script - Reading the Time Table file



```cpp
void TaskSet::compute_time_table() {
    for (int tsk=0; tsk<m_priority_vector.size(); ++tsk) {
        std::vector<int> response_time;
        std::vector<int> waiting_time;
        std::vector<int> activations_rank;
        std::vector<int> deactivation_rank;
        for (int p=0; p<m_hyper_period; ++p) {
            int period = p*m_priority_vector[tsk].get_period() + m_priority_vector[tsk].get_offset();
            int deadline = (p+1)*m_priority_vector[tsk].get_deadline() + m_priority_vector[tsk].get_offset();
            if (period > m_hyper_period) {
                break;
            }
            activations_rank.push_back(period);
            deactivation_rank.push_back(deadline);
        }
        int i = 0;
        for (auto elem: activations_rank) {
            response_time.push_back(0);
            waiting_time.push_back(0);
            int init_activation = elem;
            bool mutual_excl = false;
            for (int j=0; j<m_priority_vector[tsk].get_computation(); ++j) {
                while (m_time_table[elem+j] != "") elem++;      You, 2 months ago • reduce time table computat
                m_time_table[elem + j] = m_priority_vector[tsk].name;
            }
            ++i;
        }
```

Figure 4.5: Python Script - Gantt Plotting script

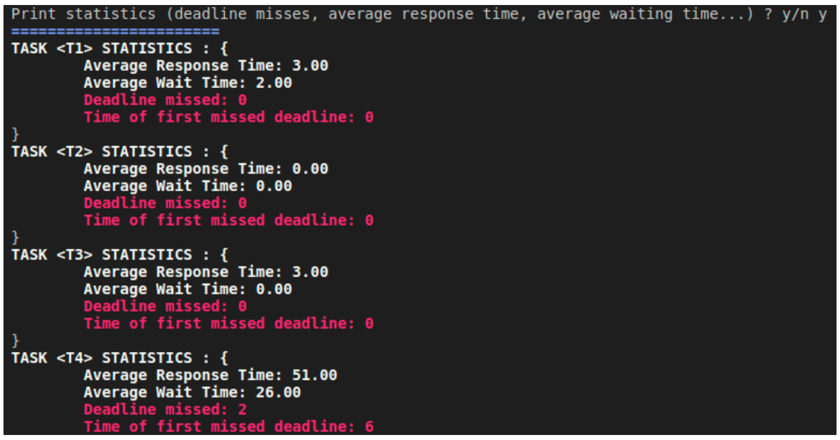
The result can be seen on Fig. 1.1. If the sufficient condition isn't met, and the user decides to plot the schedule anyway and it fails, you will see on the statistics the deadlines missed etc. If a task is never scheduled, its statistics will all show 0. The statistics are composed of the Average Response Time (ART), the Average Wait Time (AWT), the amount of Deadlines missed and the time of the first missed deadline, such as:

$$ART(T_j) = \frac{\sum_i^n (FinishingTime(T_j) - ReleaseTime(T_j))}{n}$$

$$ART = \frac{\sum_j^n (ART(T_j))}{n}$$

$$AWT(T_j) = \frac{\sum_i^n (StartingExecTime(T_j) - ReleaseTime(T_j))}{n}$$

$$AWT = \frac{\sum_j^n (AWT(T_j))}{n}$$



```
Print statistics (deadline misses, average response time, average waiting time...) ? y/n y
=======================
TASK <T1> STATISTICS : {
        Average Response Time: 3.00
        Average Wait Time: 2.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T2> STATISTICS : {
        Average Response Time: 0.00
        Average Wait Time: 0.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T3> STATISTICS : {
        Average Response Time: 3.00
        Average Wait Time: 0.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T4> STATISTICS : {
        Average Response Time: 51.00
        Average Wait Time: 26.00
        Deadline missed: 2
        Time of first missed deadline: 6
```

Figure 4.6: Unschedulable Statistics

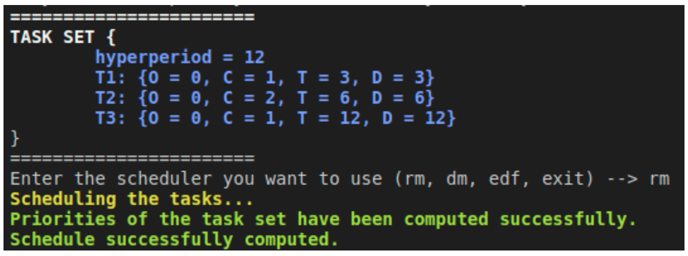
Chapter 5

Scenarios

5.1 Usecase - Ideal

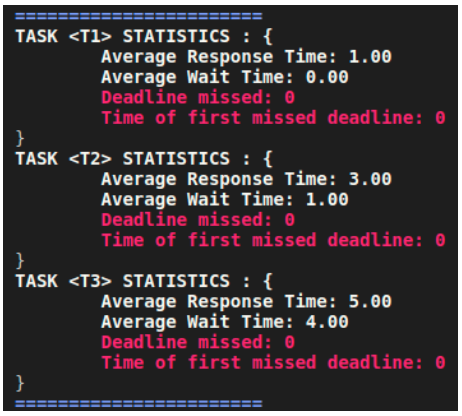
$$\begin{aligned}
T1 &= (O = 0, C = 1, T = 3, D = 3)\\
T2 &= (O = 0, C = 2, T = 6, D = 6)\\
T3 &= (O = 0, C = 2, T = 12, D = 12)\\
U &= 0.75\\
H &= 12
\end{aligned} \tag{5.1}$$

In the example at 5.1, we expect the simulator to schedule it straight away, as both the necessary and sufficient condition are met.



```
======================
TASK SET {
        hyperperiod = 12
        T1: {O = 0, C = 1, T = 3, D = 3}
        T2: {O = 0, C = 2, T = 6, D = 6}
        T3: {O = 0, C = 1, T = 12, D = 12}
}
======================
Enter the scheduler you want to use (rm, dm, edf, exit) --> rm
Scheduling the tasks...
Priorities of the task set have been computed successfully.
Schedule successfully computed.
```

Figure 5.1: Task Set 5.1 output



```
======================
TASK <T1> STATISTICS : {
        Average Response Time: 1.00
        Average Wait Time: 0.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T2> STATISTICS : {
        Average Response Time: 3.00
        Average Wait Time: 1.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T3> STATISTICS : {
        Average Response Time: 5.00
        Average Wait Time: 4.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
======================
```

Figure 5.2: Task Set 5.1 statistics

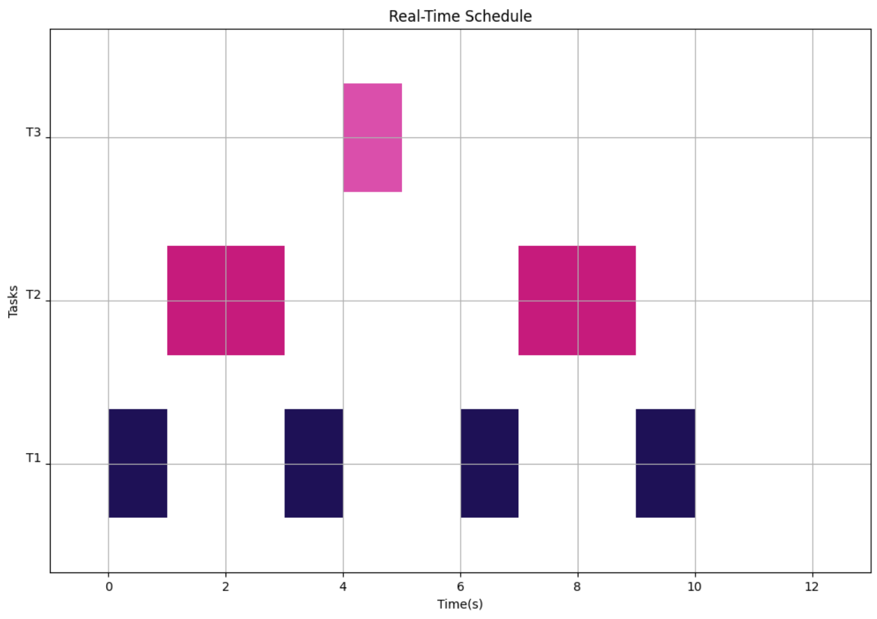


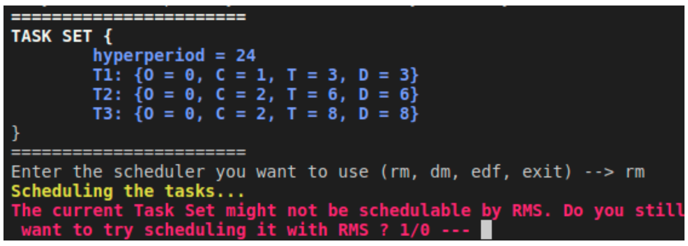
Figure 5.3: Task Set 5.1 schedule

The simulator behave as predicted and the schedule is valid with no deadline missed.

5.2 Usecase - Sufficient condition not met

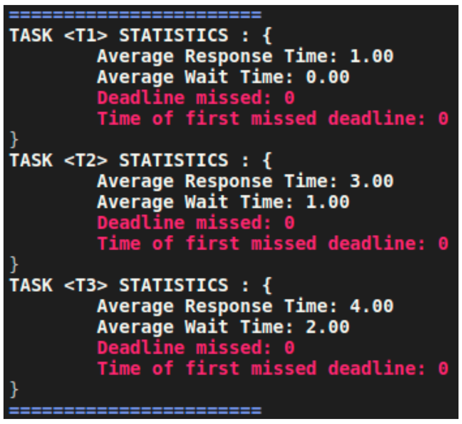
$$\begin{aligned} T1 &= (O = 0, C = 1, T = 3, D = 3) \\ T2 &= (O = 0, C = 2, T = 6, D = 6) \\ T3 &= (O = 0, C = 2, T = 8, D = 8) \\ U &= 0.916 \\ H &= 24 \end{aligned} \tag{5.2}$$

In the example at 5.2, we expect the simulator to say that the Task Set might not be schedulable, so the only way to know for sure is to build the schedule anyway.



```
=======================
TASK SET {
        hyperperiod = 24
        T1: {O = 0, C = 1, T = 3, D = 3}
        T2: {O = 0, C = 2, T = 6, D = 6}
        T3: {O = 0, C = 2, T = 8, D = 8}
}
=======================
Enter the scheduler you want to use (rm, dm, edf, exit) --> rm
Scheduling the tasks...
The current Task Set might not be schedulable by RMS. Do you still
 want to try scheduling it with RMS ? 1/0 ---
```

Figure 5.4: Task Set 5.2 output



```
=======================
TASK <T1> STATISTICS : {
        Average Response Time: 1.00
        Average Wait Time: 0.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T2> STATISTICS : {
        Average Response Time: 3.00
        Average Wait Time: 1.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T3> STATISTICS : {
        Average Response Time: 4.00
        Average Wait Time: 2.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
=======================
```

Figure 5.5: Task Set 5.2 statistics

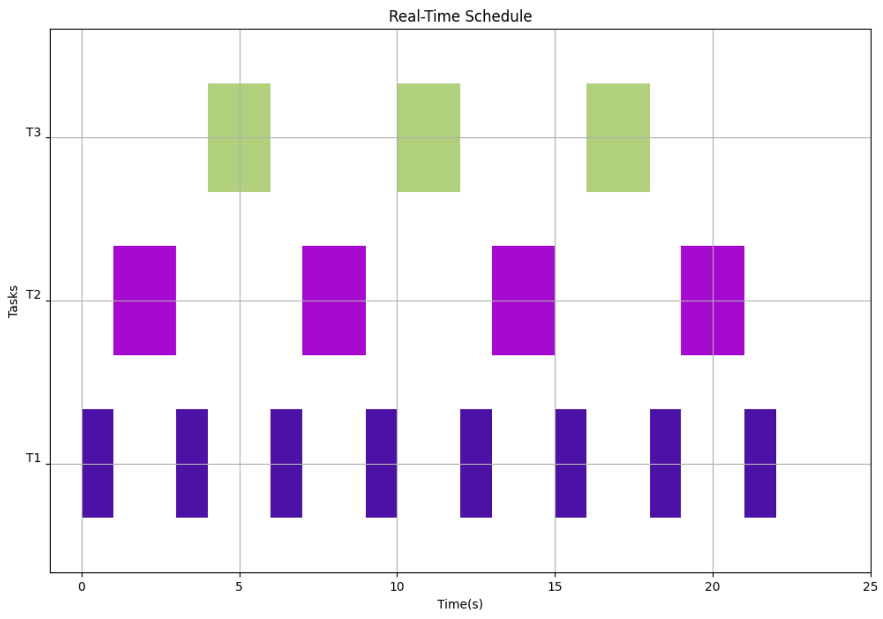
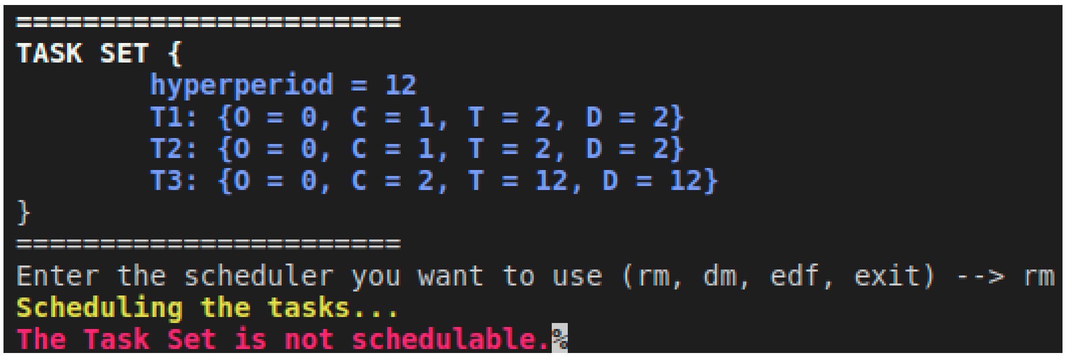


Figure 5.6: Task Set 5.2 schedule

The simulator behave as predicted and the schedule proves that the task set is schedulable.

5.3 Usecase - Necessary condition not met

$$\begin{aligned}T1 &= (O=0, C=1, T=2, D=2)\\ T2 &= (O=0, C=2, T=2, D=2)\\ T3 &= (O=0, C=2, T=12, D=12)\\ U &= 1.16\\ H &= 12\end{aligned} \tag{5.3}$$



```
=======================
TASK SET {
        hyperperiod = 12
        T1: {O = 0, C = 1, T = 2, D = 2}
        T2: {O = 0, C = 1, T = 2, D = 2}
        T3: {O = 0, C = 2, T = 12, D = 12}
}
=======================
Enter the scheduler you want to use (rm, dm, edf, exit) --> rm
Scheduling the tasks...
The Task Set is not schedulable.%
```

Figure 5.7: Task Set 5.3 schedule

As expected, after one confirms that the tasks are registered and that it's time to try and schedule, the program exits at once and notifies that the Task Set is not schedulable.

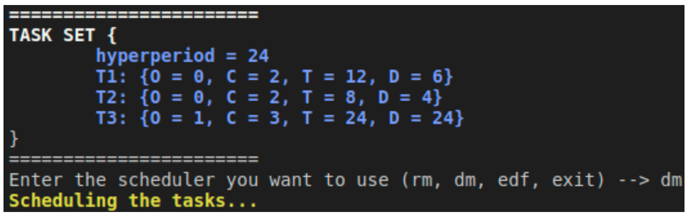
The simulator behave as predicted and the schedule is valid with no deadline missed.

5.4 Bonus Usecase - Deadline Monotonic

$$\begin{aligned} T1 &= (O = 0, C = 2, T = 12, D = 6) \\ T2 &= (O = 0, C = 2, T = 8, D = 4) \\ T3 &= (O = 0, C = 3, T = 24, D = 24) \\ Ch &= 0.96 \\ U &= 0.54 \\ H &= 24 \end{aligned} \tag{5.4}$$

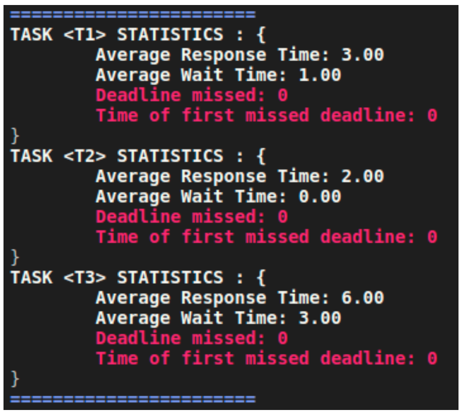
$$U = \sum_{i}^{n} \frac{Ci}{Ti} <= n \times (2^{\frac{1}{n}} - 1)$$

In the example at 5.2, we expect the simulator to say that the Task Set might not be schedulable, as the sufficient condition for the Deadline Monotonic algorithm is not met. The task set will then be validated by the built schedule.



```
=======================
TASK SET {
        hyperperiod = 24
        T1: {O = 0, C = 2, T = 12, D = 6}
        T2: {O = 0, C = 2, T = 8, D = 4}
        T3: {O = 1, C = 3, T = 24, D = 24}
}
=======================
Enter the scheduler you want to use (rm, dm, edf, exit) --> dm
Scheduling the tasks...
```

Figure 5.8: Task Set 5.4 output



```
=======================
TASK <T1> STATISTICS : {
        Average Response Time: 3.00
        Average Wait Time: 1.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T2> STATISTICS : {
        Average Response Time: 2.00
        Average Wait Time: 0.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
TASK <T3> STATISTICS : {
        Average Response Time: 6.00
        Average Wait Time: 3.00
        Deadline missed: 0
        Time of first missed deadline: 0
}
=======================
```

Figure 5.9: Task Set 5.4 statistics

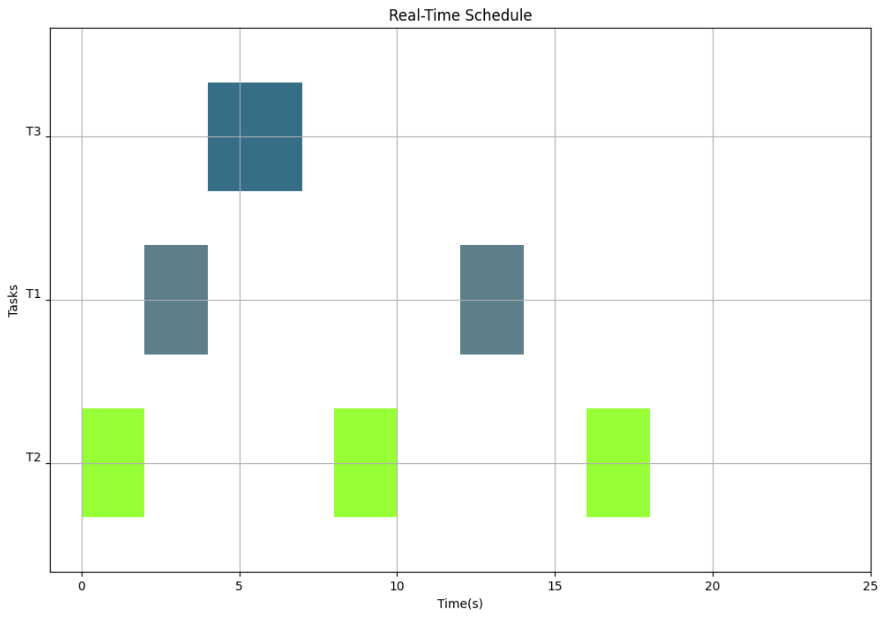


Figure 5.10: Task Set 5.4 schedule

The simulator behave as predicted, the schedule is indeed typical of deadline monotonic with the task with the smallest relative deadline being the most prioritized.

Chapter 6

Conclusion

The most challenging part of the scheduler was to figure out how to compute the schedule. How to handle the preemption and when to activate tasks, how to create a relevant structure to handle the schedule. I was fortunate to have done a personal project last year to plot schedules from Trampoline traces in PRETS and had a lot of the thought process already figured out, as I had implemented RM, DM and EDF schedulers. The project was very robust (sources here. It's very interesting from an architectural problem, as the formulas aren't very hard to implement, what's really hard is figuring out the Software Engineering part of the issue.

Appendix A

Source Code

A.1 Task.h

```
#ifndef TASK_H
#define TASK_H

#define RESET       "\033[0m"
#define BLACK       "\033[30m"      /* Black */
#define RED         "\033[31m"      /* Red */
#define GREEN       "\033[32m"      /* Green */
#define YELLOW      "\033[33m"      /* Yellow */
#define BLUE        "\033[34m"      /* Blue */
#define MAGENTA     "\033[35m"      /* Magenta */
#define CYAN        "\033[36m"      /* Cyan */
#define WHITE       "\033[37m"      /* White */
#define BOLDBLACK   "\033[1m\033[30m"      /* Bold Black */
#define BOLDRED     "\033[1m\033[31m"      /* Bold Red */
#define BOLDGREEN   "\033[1m\033[32m"      /* Bold Green */
#define BOLDYELLOW  "\033[1m\033[33m"      /* Bold Yellow */
#define BOLDBLUE    "\033[1m\033[34m"      /* Bold Blue */
#define BOLDMAGENTA "\033[1m\033[35m"      /* Bold Magenta */
#define BOLDCYAN    "\033[1m\033[36m"      /* Bold Cyan */
#define BOLDWHITE   "\033[1m\033[37m"      /* Bold White */

#include <stdlib.h>
#include <iostream>
#include <string>
typedef struct {
    double average_response_time;
    double average_wait_time;
    int deadlines_missed = 0;
    int first_deadline_missed_t = 0;
    double average_missed_deadline_t = 0;
} TaskStatistics;

class Task {
    public:
        Task(const char* name, int offset, int period, int
           computation, int deadline);
        const char* name;
        int get_offset() const;
        int get_period() const;
        int get_computation() const;
        int get_deadline() const;
        int get_priority() const;
        double get_utilization() const;
        double get_ch() const;

        bool get_ready() const;
        const void set_ready(bool r);
        int get_execution_time() const;
        const void set_execution_time(int ex_t);

        int get_arrival_time() const;
        const void set_arrival_time(int at);

        int get_absolute_deadline() const;
        const void set_absolute_deadline(int ad);

        const void set_priority(int p);
        void print_task() const;

        TaskStatistics get_statistics() const;
        const void set_average_response_time(double art);
        const void set_average_waiting_time(double awt);
        const void set_deadlines_missed(int dm);
        const void set_first_deadline_missed_t(int fdmt);
        const void set_deadline_missed_average_t(double dmat);

        void pretty_print_statistics() const;

    private:
        int m_offset;
        int m_period;
        int m_computation;
        int m_deadline;
        int m_priority = 0;
        bool m_ready = false;
        int m_execution_time;
        int m_absolute_deadline;
        int m_arrival_time;
        double m_ch;
        double m_utilization;
        TaskStatistics m_statistics;
};

#endif //TASK_H
```

A.2 Task.cpp

```
#include "Task.h"

/**
 * @brief Construct a new Task:: Task object
 *
 * @param name
 * @param offset
 * @param computation
 * @param period
 * @param deadline
 */
Task::Task(const char* name, int offset, int computation, int period,
    int deadline) {
    this->name = name;
    m_computation = computation;
    m_deadline = deadline;
    m_offset = offset;
    m_period = period;
    m_utilization = double(computation)/double(period);
    m_ch = double(computation)/double(deadline);
}

/**
 * @brief get task offset
 *
 * @return int
 */
int Task::get_offset() const {
    return m_offset;
}

/**
 * @brief get task period
 *
 * @return int
 */
int Task::get_period() const {
    return m_period;
}

/**
 * @brief get task wcet
 *
 * @return int
 */
int Task::get_computation() const {
    return m_computation;
}

/**
 * @brief get deadline
 *
 * @return int
 */
int Task::get_deadline() const {
    return m_deadline;
}

/**
 * @brief get utilization factor
 *
 * @return double
 */
double Task::get_utilization() const {
    return m_utilization;
}

/**
 * @brief get ch factor
 *
 * @return double
 */
double Task::get_ch() const {
    return m_ch;
}

/**
 * @brief get priority
 *
 * @return int
 */
int Task::get_priority() const {
    return m_priority;
}

/**
 * @brief set priority
 *
 * @param p
 */
const void Task::set_priority(int p) {
    m_priority = p;
}

/**
 * @brief get ready state
 *
 * @return true
 * @return false
 */
bool Task::get_ready() const {
    return m_ready;
}

/**
 * @brief set ready state
 *
 * @param r
 */
const void Task::set_ready(bool r) {
    m_ready = r;
}

/**
 * @brief get execution time
 *
 * @return int
 */
int Task::get_execution_time() const {
    return m_execution_time;
}

/**
 * @brief set execution time
 *
 * @param ex_t
 */
const void Task::set_execution_time(int ex_t) {
    m_execution_time = ex_t;
}

/**
 * @brief get arrival time
 *
 * @return int
 */
int Task::get_arrival_time() const {
    return m_arrival_time;
}

/**
 * @brief set arrival time
 *
 * @param at
 */
const void Task::set_arrival_time(int at) {
    m_arrival_time = at;
}

/**
 * @brief get absolute deadline
 *
 * @return int
 */
int Task::get_absolute_deadline() const {
    return m_absolute_deadline;
}

/**
 * @brief set absolute deadline
 *
 * @param ad
 */
```

```
const void Task::set_absolute_deadline(int ad) {
    m_absolute_deadline = ad;
}

/**
 * @brief task structure printer
 *
 */
void Task::print_task() const {
    if (m_priority == 0) {
        printf("%s: {O = %d, C = %d, T = %d, D = %d}\n", this->name,
            m_offset, m_computation, m_period, m_deadline);
    } else {
        printf("%s: {O = %d, C = %d, T = %d, D = %d, Prio = %d}\n",
            this->name, m_offset, m_computation, m_period, m_deadline,
            m_priority);
    }
}

/**
 * @brief get statistics
 *
 * @return TaskStatistics
 */
TaskStatistics Task::get_statistics() const {
    return m_statistics;
}

/**
 * @brief set average response time
 *
 * @param art
 */
const void Task::set_average_response_time(double art) {
    m_statistics.average_response_time = art;
}

/**
 * @brief set average waiting time
 *
 * @param awt
 */
const void Task::set_average_waiting_time(double awt) {
    m_statistics.average_wait_time = awt;
}

/**
 * @brief set deadline missed
 *
 * @param dm
 */
const void Task::set_deadlines_missed(int dm) {
    m_statistics.deadlines_missed = dm;
}
```

```
/**
 * @brief set time of the first deadline missed
 *
 * @param fdmt
 */
const void Task::set_first_deadline_missed_t(int fdmt) {
    m_statistics.first_deadline_missed_t = fdmt;
}

/**
 * @brief set average time by which task missed deadlines
 *
 * @param dmat
 */
const void Task::set_deadline_missed_average_t(double dmat) {
    m_statistics.average_missed_deadline_t = dmat;
}

/**
 * @brief task statistics printer
 *
 */
void Task::pretty_print_statistics() const {
    printf("%sTASK <%s> STATISTICS : {\n\tAverage Response Time:
        %.2f\n\tAverage Wait Time: %.2f\n\t%sDeadline missed:
        %d\n\tTime of first missed deadline: %d\n%s}%s\n",
            BOLDWHITE, this->name, m_statistics.average_response_time,
               m_statistics.average_wait_time, BOLDRED,
               m_statistics.deadlines_missed,
               m_statistics.first_deadline_missed_t, RESET, RESET/*,
               m_statistics.average_missed_deadline_t*/);
}
```

A.3 Helpers.h

```
#ifndef HELPER_H
#define HELPER_H

#include <fstream>
#include <iostream>
#include <string.h>
#include "Task.h"

/**
 * @brief
 *
 * @param tsk1
 * @param tsk2
 * @return true
 * @return false
 */
static bool rateMonotonicSorter(Task tsk1, Task tsk2) {
    return tsk1.get_period() < tsk2.get_period();
}

/**
 * @brief
 *
 * @param tsk1
 * @param tsk2
 * @return true
 * @return false
 */
static bool deadlineMonotonicSorter(Task tsk1, Task tsk2) {
    return tsk1.get_deadline() < tsk2.get_deadline();
}

static int gcd(int a, int b)
{
    if (b == 0)
        return a;
    return gcd(b, a % b);
}

static int findlcm(int arr[], int n)
{
    int ans = arr[0];

    for (int i = 1; i < n; i++)
        ans = (((arr[i] * ans)) /
                (gcd(arr[i], ans)));

    return ans;
}

/**
 * @brief
 *
 * @param input
 */
static void print_task_vector(std::vector<const char*> const &input) {
    std::cout << "job == ";
    int i = 0;
    for (auto elem : input) {
        if (i != input.size() - 1) {
            printf("%s(%d) ->", elem, i);
        } else {
            printf("%s(%d)\n", elem, i);
        }
        i++;
    }
}

/**
 * @brief
 *
 * @param input
 * @param filename
 */
static void stream_schedule_to_file(std::vector<const char*> const
   &input, const char* filename) {
    std::ofstream myfile;
    char buf[30];
    sprintf(buf, "data/%s", filename);
    myfile.open(buf);
    for (auto elem : input) {
        myfile << elem;
        myfile << "\n";
    }
    myfile.close();
}

/**
 * @brief
 *
 * @param c
 * @return wchar_t*
 */
static wchar_t* GetWC(const char *c) {
    const size_t c_size = strlen(c) + 1;
    wchar_t* wc = new wchar_t[c_size];
    mbstowcs(wc, c, c_size);
    return wc;
}

/**
 * @brief Get the Const W C object
 *
 * @param c
 * @return const wchar_t*
 */
static const wchar_t* GetConstWC(const char *c) {
    const size_t c_size = strlen(c) + 1;
    wchar_t* wc = new wchar_t[c_size];
    mbstowcs(wc, c, c_size);
    return wc;
}

/**
 * @brief
 *
 * @param argc
 * @param argv
 */
static void plot_gantt_from_python(int argc, char *argv[]) {
    char buffer[50];
    sprintf(buffer, "python3 src/gantt.py --filename data/%s",
       argv[1]);
    int x = system(buffer);
    if (x != 0) {
        std::cout << "Failure: python script didn't execute correctly"
           << std::endl;
    }
}

#endif //HELPER_H
```

A.4 DeadlineMontonic.h

```
#ifndef DEADLINE_MONOTONIC_H
#define DEADLINE_MONOTONIC_H

#include <cmath>
#include <vector>
#include <algorithm>
#include <map>
#include "Helpers.h"

class DeadlineMonotonic {
    public:
        DeadlineMonotonic();
        std::map<const char*, Task> prioritize(std::map<const char*,
           Task> tasks);
        std::vector<Task> get_prioritized_tasks() const;
        void reset_priorities();
        bool compute_sufficient_condition(int ntasks, double charge);

    private:
        std::vector<Task> m_task_vector;
};

#endif //RATE_MONOTONIC_H
```

A.5 DeadlineMonotonic.cpp

```
#include "DeadlineMonotonic.h"

/**
 * @brief Construct a new Deadline Monotonic:: Deadline Monotonic
     object
 */
DeadlineMonotonic::DeadlineMonotonic() {}

/**
 * @brief Compute priority according to the DM policy
 * @param tasks
 * @return std::map<const char*, Task>
 */
std::map<const char*, Task>
   DeadlineMonotonic::prioritize(std::map<const char*, Task> tasks) {
    int max_priority = tasks.size();
    for (auto pair : tasks) {
        m_task_vector.push_back(pair.second);
    }
    std::sort(m_task_vector.begin(), m_task_vector.end(),
       deadlineMonotonicSorter);
    for (auto elem : m_task_vector) {
        (tasks.at(elem.name)).set_priority(max_priority);
        --max_priority;
    }

    return tasks;
}

/**
 * @brief Prioritized tasks structure getter
 * @return std::vector<Task>
 */
std::vector<Task> DeadlineMonotonic::get_prioritized_tasks() const {
    return m_task_vector;
}

/**
 * @brief Reset priorities by clearing the structure
 *
 */
void DeadlineMonotonic::reset_priorities()  {
    m_task_vector.clear();
}

bool DeadlineMonotonic::compute_sufficient_condition(int ntasks,
   double charge) {
    double condition = ntasks*(pow(2, 1.0/ntasks) - 1);
    if (charge != 0 && charge <= condition) {
        return true;
    }
    return false;
}
```

A.6 RateMonotonic.h

```
#ifndef RATE_MONOTONIC_H
#define RATE_MONOTONIC_H

#include <cmath>
#include <vector>
#include <algorithm>
#include <map>
#include "Helpers.h"

class RateMonotonic {
    public:
        RateMonotonic();
        std::map<const char*, Task> prioritize(std::map<const char*,
           Task> tasks);
        std::vector<Task> get_prioritized_tasks() const;
        void reset_priorities();
        bool compute_sufficient_condition(int ntasks, double charge);

    private:
        std::vector<Task> m_task_vector;
};

#endif //RATE_MONOTONIC_H
```

A.7 RateMonotonic.cpp

```
#include "RateMonotonic.h"

/**
 * @brief Construct a new Rate Monotonic:: Rate Monotonic object
 */
RateMonotonic::RateMonotonic() {}

/**
 * @brief Compute priority according to the RM policy
 *
 * @param tasks
 * @return std::map<const char*, Task>
 */
std::map<const char*, Task> RateMonotonic::prioritize(std::map<const
   char*, Task> tasks) {
    int max_priority = tasks.size();
    for (auto pair : tasks) {
        m_task_vector.push_back(pair.second);
    }
    std::sort(m_task_vector.begin(), m_task_vector.end(),
       rateMonotonicSorter);
    for (auto elem : m_task_vector) {
        (tasks.at(elem.name)).set_priority(max_priority);
        --max_priority;
    }

    return tasks;
}

/**
 * @brief Prioritized tasks structure getter
 * @return std::vector<Task>
 */
std::vector<Task> RateMonotonic::get_prioritized_tasks() const {
    return m_task_vector;
}

/**
 * @brief Reset priorities by clearing the structure
 */
void RateMonotonic::reset_priorities()  {
    m_task_vector.clear();
}

bool RateMonotonic::compute_sufficient_condition(int ntasks, double
   charge) {
    double condition = ntasks*(pow(2, 1.0/ntasks) - 1);
    if (charge != 0 && charge <= condition) {
        return true;
    }
    return false;
}
```

A.8 TaskSet.h

```
#ifndef TASK_SET_H
#define TASK_SET_H

#include "RateMonotonic.h"
#include "DeadlineMonotonic.h"
#include "vector"
#include <numeric>

#define RATE_MONOTONIC          0
#define DEADLINE_MONOTONIC      1
#define EARLIEST_DEADLINE_FIRST 2

class TaskSet {
    public:
        TaskSet();
        void register_task(Task tsk);
        void remove_task(const char* task_id);
        std::map<const char*, Task> get_task_set() const;
        void print_task_set();
        void schedule(int scheduler);
        int get_number_of_tasks() const;
        std::vector<const char*> get_time_table() const;
        void print_schedule() const;
        void print_statistics() const;

    private:
        std::map<const char*, Task> m_tasks;
        std::vector<const char*> m_time_table;
        std::vector<Task> m_priority_vector;
        int m_number_of_tasks = 0;
        int m_hyper_period;
        void compute_time_table();
        void compute_hyper_period();
};

#endif // TaskSet
```

A.9 TaskSet.cpp

```
#include "TaskSet.h"

TaskSet::TaskSet() {}

/**
 * @brief register a task in the set
 *
 * @param tsk
 */
void TaskSet::register_task(Task tsk) {
    if (m_tasks.find(tsk.name) == m_tasks.end()) {
        // Task not registered, free to go
        m_tasks.insert(std::pair<const char*, Task>(tsk.name, tsk));
        ++m_number_of_tasks;
        this->compute_hyper_period();
        printf("%sTask <%s> has been registered.\n%s", BOLDGREEN,
            tsk.name, RESET);
    } else {
        std::cout << BOLDRED << "Failed to register task: id already
            registered." << RESET << std::endl;
    }
}

/**
 * @brief remove a task from the set
 *
 * @param task_id
 */
void TaskSet::remove_task(const char* task_id) {
    if (m_tasks.find(task_id) == m_tasks.end()) {
        printf("%sTask <%s> has not been registered, failed to
            remove.\n%s", BOLDRED, task_id, RESET);
    } else {
        m_tasks.erase(task_id);
        --m_number_of_tasks;
        this->compute_hyper_period();
        printf("%sTask <%s> has been removed\n%s", BOLDGREEN, task_id,
            RESET);
    }
}

/**
 * @brief compute hyper period of the set
 *
 */
void TaskSet::compute_hyper_period() {
    int hyper_periods[m_number_of_tasks];
    int i = 0;
    for (auto const& [key, val] : m_tasks) {
        hyper_periods[i] = val.get_period();
        ++i;
    }
    int n = sizeof(hyper_periods) / sizeof(hyper_periods[0]);
    m_hyper_period = findlcm(hyper_periods, n);
    std::vector<const char*> tempVec(m_hyper_period, "");
    m_time_table = tempVec;
}

/**
 * @brief get the task set
 *
 * @return std::map<const char*, Task>
 */
std::map<const char*, Task> TaskSet::get_task_set() const {
    return m_tasks;
}

/**
 * @brief get number of tasks registered
 *
 * @return int
 */
int TaskSet::get_number_of_tasks() const {
    return m_number_of_tasks;
}

/**
 * @brief tasks set printer
 *
 */
void TaskSet::print_task_set() {
    printf("%s======================\n", BOLDWHITE);
    printf("TASK SET {\n");
    printf("%s\thyperperiod = %d\n", BOLDBLUE, m_hyper_period);
    for (auto it = m_tasks.cbegin(); it != m_tasks.cend(); ++it) {
        printf("\t");
        (it->second).print_task();
    }
    printf("%s}\n======================\n%s", RESET, RESET);
}

/**
 * @brief schedule task set according to a chosen policy
 *
 * @param scheduler
 */
void TaskSet::schedule(int scheduler) {
    std::cout << BOLDYELLOW << "Scheduling the tasks..." << RESET <<
        std::endl;
    bool ok;
    double processor_charge = 0;
    double ch = 0;
    for (auto it = m_tasks.cbegin(); it != m_tasks.cend(); ++it) {
        processor_charge += (it->second).get_utilization();
        ch += (it->second).get_ch();
    }
    if (processor_charge > 1) {
        std::cout << BOLDRED << "The Task Set is not schedulable." <<
```

```
            RESET;
        exit(EXIT_FAILURE);
    }
    switch(scheduler) {
        case RATE_MONOTONIC: {
            auto rm = RateMonotonic();

            ok =
                rm.compute_sufficient_condition(this->m_number_of_tasks,
                processor_charge);
            if (ok) {
                this->m_tasks = rm.prioritize(this->m_tasks);
                this->m_priority_vector = rm.get_prioritized_tasks();
                std::cout << BOLDGREEN << "Priorities of the task set
                    have been computed successfully." << RESET <<
                    std::endl;
            } else {
                int yn;
                std::cout << BOLDRED << "The current Task Set might
                    not be schedulable by RMS. Do you still want to try
                    scheduling it with RMS ? 1/0 --- " << RESET;
                std::cin >> yn;
                if (yn == 1) {
                    this->m_tasks = rm.prioritize(this->m_tasks);
                    this->m_priority_vector =
                        rm.get_prioritized_tasks();
                } else exit(1);
            }
            this->compute_time_table();
            std::cout << BOLDGREEN << "Schedule successfully
                computed." << RESET << std::endl;
        } break;
        case DEADLINE_MONOTONIC: {
            auto dm = DeadlineMonotonic();

            ok =
                dm.compute_sufficient_condition(this->m_number_of_tasks,
                ch);
            if (ok) {
                this->m_tasks = dm.prioritize(this->m_tasks);
                this->m_priority_vector = dm.get_prioritized_tasks();
                std::cout << BOLDGREEN << "Priorities of the task set
                    have been computed successfully." << RESET <<
                    std::endl;
            } else {
                int yn;
                std::cout << BOLDRED << "The current Task Set might
                    not be schedulable by DMS. Do you still want to try
                    scheduling it with RMS ? 1/0 --- " << RESET;
                std::cin >> yn;
                if (yn == 1) {
                    this->m_tasks = dm.prioritize(this->m_tasks);
                    this->m_priority_vector =
                        dm.get_prioritized_tasks();
                } else exit(1);
            }
            this->compute_time_table();
            std::cout << BOLDGREEN << "Schedule successfully
                computed." << RESET << std::endl;
        } break;
        case EARLIEST_DEADLINE_FIRST: {
            std::cout << BOLDRED << "Scheduler not supported yet" <<
                RESET << std::endl;
        } break;
        default: {
            std::cout << BOLDRED << "Scheduler not supported yet" <<
                RESET << std::endl;
        } break;
    }
}

/**
 * @brief get availability time table
 *
 * @return std::vector<const char*>
 */
std::vector<const char*> TaskSet::get_time_table() const {
    return m_time_table;
}

/**
 * @brief compute availability time table (schedule)
 *
 */
void TaskSet::compute_time_table() {
    for (int tsk=0; tsk<m_priority_vector.size(); ++tsk) {
        std::vector<int> response_time;
        std::vector<int> waiting_time;
        std::vector<int> activations_rank;
        std::vector<int> deactivation_rank;
        for (int p=0; p<m_hyper_period; ++p) {
            int period = p*m_priority_vector[tsk].get_period() +
                m_priority_vector[tsk].get_offset();
            int deadline = (p+1)*m_priority_vector[tsk].get_deadline()
                + m_priority_vector[tsk].get_offset();
            if (period >= m_hyper_period) {
                break;
            }
            activations_rank.push_back(period);
            deactivation_rank.push_back(deadline);
        }
        int i = 0;
        for (auto elem: activations_rank) {
            response_time.push_back(0);
            waiting_time.push_back(0);
            int init_activation = elem;
            bool mutual_excl = false;
            for (int j=0; j<m_priority_vector[tsk].get_computation();
                ++j) {
                while (m_time_table[elem+j] != "") {
                    if (elem+1+j >= m_hyper_period) {
                        std::cout << BOLDRED << "Failed to schedule
                           task set" << RESET << std::endl;
                        exit(EXIT_FAILURE);
                    }
                    elem++;
                }
                if (elem + j > m_hyper_period) {
                    response_time.pop_back();
                    waiting_time.pop_back();
                    break;
                }
                if (j == 0) {
                    waiting_time[i] = elem - init_activation;
                    if (waiting_time[i] >
                       m_priority_vector[tsk].get_deadline() &&
                       !mutual_excl) {
                        m_tasks.at(m_priority_vector[tsk].name).set_deadlines_missed
                           + 1);
                        if
                           (m_tasks.at(m_priority_vector[tsk].name).get_statistics(
                           == 1) {
                            m_tasks.at(m_priority_vector[tsk].name).set_first_deadli
                        }
                        mutual_excl = true;
                    }
                }
                if (j == (m_priority_vector[tsk].get_computation() -
                   1)) {
                    response_time[i] = elem + j - init_activation + 1;
                    if (response_time[i] >
                       m_priority_vector[tsk].get_deadline() &&
                       !mutual_excl) {
                        m_tasks.at(m_priority_vector[tsk].name).set_deadlines_missed
                           + 1);
                        if
                           (m_tasks.at(m_priority_vector[tsk].name).get_statistics(
                           == 1) {
                            m_tasks.at(m_priority_vector[tsk].name).set_first_deadli
                        }
                        mutual_excl = true;
                    }
                }
                m_time_table[elem + j] = m_priority_vector[tsk].name;
            }
            ++i;
        }
        if (response_time.size() != 0) {
            double art = std::accumulate(response_time.begin(),
               response_time.end(), 0) / response_time.size();
            m_tasks.at(m_priority_vector[tsk].name).set_average_response_time(art);
        }
        if (waiting_time.size() != 0) {
            double awt = std::accumulate(waiting_time.begin(),
               waiting_time.end(), 0) / waiting_time.size();
```

```
                m_tasks.at(m_priority_vector[tsk].name).set_average_waiting_time(awt);
            }
        }
    }

    /**
     * @brief print statistics of each task in the set
     *
     */
    void TaskSet::print_statistics() const {
        printf("%s=======================%s\n", BOLDBLUE, RESET);
        for (auto &pair : m_tasks) {
            pair.second.pretty_print_statistics();
        }
        printf("%s=======================%s\n", BOLDBLUE, RESET);
    }

    /**
     * @brief ugly schedule printer
     *
     */
    void TaskSet::print_schedule() const {
        printf("\n==SCHEDULE==\n");
        for ( auto &pair : m_tasks ) {
            printf("%s |", pair.first);
            for ( auto &elem : m_time_table ) {
                if (pair.first == elem) {
                    printf("   ");
                } else {
                    printf("_");
                }
            }
            printf("\n");
        }
    }
```

A.10 gantt.py

```
import matplotlib.pyplot as plt
import numpy as np
import argparse

class GanttPlot():
    def __init__(self, ylim, xlim, title=None):
        self.fig, self.gnt = plt.subplots(figsize=(12, 8))
        self.gnt.set_ylim(0, ylim+1)
        self.gnt.set_xlim(-1, xlim+1)
        self.ylim = ylim
        self.xlim = xlim
        self.tasksYticks = {}
        self.tasksColors = {}
        self.tasks = {}

        # Setting labels for x-axis and y-axis
        self.gnt.set_xlabel('Time(s)')
        self.gnt.set_ylabel('Tasks')

        # Setting graph attribute
        self.gnt.grid(True)

        if title:
            self.gnt.set_title(title)

        # Define available y position
        self.available_y = []
        index = 1
        while index < ylim:
            self.available_y.append((index, 2))
            index += 3

        # Initiate labels
        self.ylabels = [str(_) for _ in range(ylim)]
        self.gnt.set_yticks([_[0]+1 for _ in self.available_y])

        self.numberTasks = 0

    def addTask(self, name, runningPeriods):
        y_index = self.available_y[self.numberTasks]
        self.tasksColors[name] = np.random.rand(3,)
        self.tasksYticks[name] = y_index
        self.ylabels[self.numberTasks] = name
        self.gnt.set_yticklabels(labels=self.ylabels)
        self.numberTasks += 1
        self.gnt.broken_barh(
                runningPeriods, self.tasksYticks[name],
                   facecolor=self.tasksColors[name])

    def show(self):
        plt.show()
```

```python
def read_task_schedule_file(filename):
    f = open(filename, "r")
    lines = f.readlines()
    tasks_dict = dict()
    last_elem = "\n"
    rank = 0
    for elem in lines:
        if elem == '\n':
            rank += 1
            continue
        elem.replace('\n', '')
        if elem not in tasks_dict.keys():
            tasks_dict[elem] = []
        if last_elem == elem:
            tasks_dict[elem][-1][1] += 1
        else:
            tasks_dict[elem].append([rank, 1])
        last_elem = elem
        rank += 1

    for key in tasks_dict.keys():
        new_periods = []
        for period in tasks_dict[key]:
            new_periods.append((period[0], period[1]))
        tasks_dict[key] = new_periods
    return tasks_dict, len(lines)

def main():
    parser = argparse.ArgumentParser(description="Real-Time tasks
       Gantt plotter from data file")
    parser.add_argument('--filename', type=str,
       default='data/schedule', help='Register the path to the
       schedule description file')
    args = parser.parse_args()
    tasks_dict, hyperperiod = read_task_schedule_file(args.filename)

    gnt = GanttPlot(len(tasks_dict.keys())*3, hyperperiod,
       title="Real-Time Schedule")
    for key in tasks_dict.keys():
        gnt.addTask(key, tasks_dict[key])
    gnt.show()

if __name__ == "__main__":
    main()
```

A.11 main.cpp

```
#include "TaskSet.h"
#include <Python.h>
#include <unistd.h>
#include <stdio.h>
#include <string>
#include <signal.h>

volatile sig_atomic_t stop;

void inthand(int signum) {
    stop = 1;
}

int main(int argc, char *argv[])
{
    int state = 0;
    TaskSet ts = TaskSet();
    signal(SIGINT, inthand);
    std::string q;

    while (!stop) {
        switch(state) {
            case 0:
                while (1) {
                    std::cout << "Please register a task..."  <<
                       std::endl;
                    char* tname = new char[25];
                    int offset, wcet, period, deadline;
                    std::cout << "Enter task name --> ";
                    std::cin >> tname;

                    std::cout << "Enter task offset --> ";
                    std::cin >> offset;

                    std::cout << "Enter task worst case execution time
                       --> ";
                    std::cin >> wcet;

                    std::cout << "Enter task period --> ";
                    std::cin >> period;

                    std::cout << "Enter task deadline --> ";
                    std::cin >> deadline;

                    ts.register_task(Task(tname, offset, wcet, period,
                       deadline));

                    std::cout << "Do you want to register another task
                       ? y/n/exit ";
                    std::cin >> q;

                    if (q.compare("n") == 0) {
                        state = 1;
                        break;
                    } else if (q.compare("y") == 0) {
                        // nothing to do
                    } else if (q.compare("exit") == 0) {
                        state = 6;
                        break;
                    }  else {
                        state = 100;
                        break;
                    }
                }
                break;

            case(1):
                while(1) {

                    std::cout << "Do you want to remove a task ?
                       y/n/exit ";
                    std::cin >> q;

                    if (q.compare("y") == 0) {
                        char* rm_tname;
                        std::cout << "Enter task name to remove --> ";
                        std::cin >> rm_tname;

                        ts.remove_task(rm_tname);
                    } else if (q.compare("exit") == 0) {
                        state = 6;
                        break;
                    } else if (q.compare("n") == 0) {
                        state = 2;
                        break;
                    } else {
                        state = 100;
                        break;
                    }
                }
                break;

            case(2):
                std::cout << "Do you want to print your task state ?
                   y/n/exit ";
                std::cin >> q;

                if (q.compare("y") == 0) {
                    ts.print_task_set();
                } else if (q.compare("exit") == 0) {
                    state = 6;
                    break;
                } else if (q.compare("n") == 0) {
                    state = 3;
                    break;
                } else {
                    state = 100;
                    break;
                }

            case(3):
                std::cout << "Enter the scheduler you want to use (rm,
                    dm, edf, exit) --> ";
                std::cin >> q;

                if (q == "rm") {
                    ts.schedule(RATE_MONOTONIC);
                    state = 4;
                } else if (q == "dm") {
                    ts.schedule(DEADLINE_MONOTONIC);
                    state = 4;
                } else if (q == "edf") {
                    ts.schedule(EARLIEST_DEADLINE_FIRST);
                    state = 3;
                    std::cout << "Choose rm" << std::endl;
                } else if (q.compare("exit") == 0) {
                    state = 6;
                } else {
                    state = 100;
                }
                break;

            case(4):
                std::cout << "Print statistics (deadline misses,
                    average response time, average waiting time...) ?
                    y/n ";
                std::cin >> q;

                if (q.compare("y") == 0) {
                    ts.print_statistics();
                    state = 5;
                    break;
                } else if (q.compare("exit") == 0) {
                    state = 6;
                    break;
                } else if (q.compare("n") == 0) {
                    state = 5;
                    break;
                } else {
                    state = 100;
                    break;
                }

            case(5):
                std::cout << "Plot schedule ? y/n ";
                std::cin >> q;

                if (q.compare("y") == 0) {
                    stream_schedule_to_file(ts.get_time_table(),
                        argv[1]);
                    plot_gantt_from_python(argc, argv);
                    state = 6;
                    break;
```

```
                } else if (q.compare("n") == 0) {
                    state = 6;
                    break;
                } else {
                    state = 100;
                    break;
                }
            case(6):
                std::cout << "Restart a new simulation ? y/n ";
                std::cin >> q;

                if (q.compare("y") == 0) {
                    ts = TaskSet();
                    state = 0;
                    break;
                } else if (q.compare("n") == 0) {
                    exit(EXIT_SUCCESS);
                } else {
                    state = 100;
                    break;
                }
            default:
                std::cout << "Unreachable state !" << std::endl;
                exit(EXIT_FAILURE);
        }
    }

    return 0;
}
```