Institut Polytechnique des Sciences Avancées

Embedded system

Final Embedded Systems Project Report
Schedule Search

Written by :

COMTET Louis

Tuesday, April 29

Academic year 2024-2025

Introduction

This report was prepared as part of the final assignment for the Embedded Systems course at IPSA. The objective was to analyze a set of periodic tasks, each defined by its computation time and period, verify their schedulability, and determine an optimal job-level schedule using the C programming language.

To accomplish this, a non-preemptive scheduling approach was required, meaning that tasks cannot be interrupted once they have started executing.

The schedule must ensure that:

- No task misses its deadline.
- The total waiting time (delay due to other jobs running) is minimized.
- The processor's idle time is also maximized.

Code Explanation

In this section, we will explain the main functionality of the code implemented to simulate task scheduling and evaluate whether deadlines are met. For readability purposes, we will not present the entire code here, but will focus on the most important parts.

The main functions used for this simulation are:

- `simulate`: This function simulates the execution of tasks over a given period (hyperperiod) and calculates the score based on task completion.
- `permute`: This function generates all possible permutations of the execution order of tasks and determines which order provides the best result in terms of deadline satisfaction and score minimization.

For clarity and conciseness, we will provide a detailed explanation of these two main functions in the appendix of the report. The rest of the code, including auxiliary functions and the full program structure, is available in our GitHub repository (link in conclusion).

Main part of the code:

First, we created a structure associated with each task, which includes the task name, period, execution time, remaining time before the task completes, and a boolean indicating whether the task has been executed or not.

main() This is the entry point of the program:

- It initializes a predefined set of tasks with periods and computation times.
- Calculates the hyperperiod using `lcm()`.
- Calls `permute()` to explore all valid task orders.
- Displays the best task order and schedule.
- Generates the visual HTML file using `generate_html()`.

gcd(int a, int b) Computes the greatest common divisor (GCD) of two integers using the Euclidean algorithm. This is essential for calculating the least common multiple (LCM) of task periods.

lcm(int a, int b) Returns the least common multiple of two integers. It is used to compute the hyperperiod of the task set, which represents the smallest interval over which the schedule repeats.

copy_tasks(Task dest[N], Task src[N]) Copies the content of one task array into another. This is used to avoid modifying the original task set during simulations.

copy_schedule(int dest[], int src[], int len) Similar to `copy_tasks`, but for copying scheduling arrays that store which task is run at each time unit.

simulate(Task tasks[N], int hyperperiod, int schedule[]) This is the core simulation function:

- It initializes the task state (remaining execution time, release time, execution flag).
- For each time unit of the hyperperiod, it checks which tasks are ready to execute.
- It selects a task that can be executed without missing its deadline.
- It tracks execution order and calculates a score based on cumulative waiting time.
- If any task fails to meet its deadline, the function returns `-1`.

permute(...) This recursive function generates all permutations of task orders using backtracking. For each permutation, it calls `simulate()` to determine if it's valid and to calculate the performance score. If a new best score is found, it saves the current permutation and schedule.

generate_html(...) Creates an HTML file to visualize the final optimal task schedule. Each task execution is represented as a colored rectangle in a timeline using SVG. The result is a clear and interactive graphical representation.

Results

Example 1:

Task	Computation time (C)	Period (T)
τ_1	2	10
τ_2	3	10
τ_3	2	20
τ_4	2	20
τ_5	2	40
τ_6	2	40
τ_7	3	80

For this first example consisting of 7 tasks (example given in the statement), the result is shown below. We can see that all tasks are completed within their allotted time. Additionally, the execution order of the different tasks is clearly visible.

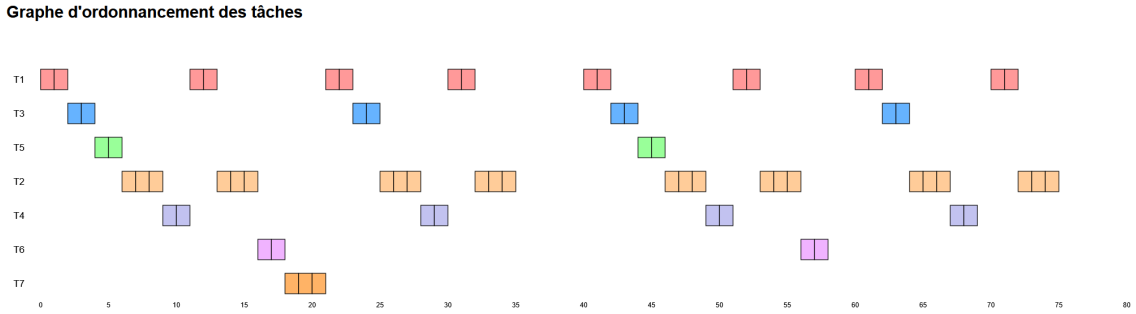

Figure 1: Example of generated scheduling graph.

Exemple 2:

We will now test our program on other task lists to schedule.

Task	Computation time (C)	Period (T)
τ_1	3	8
τ_2	2	15
τ_3	4	20

For this second example, composed of 3 tasks as shown above, we obtain the following schedule. The result is shown below. As in the previous example, all tasks are completed within their allotted time.

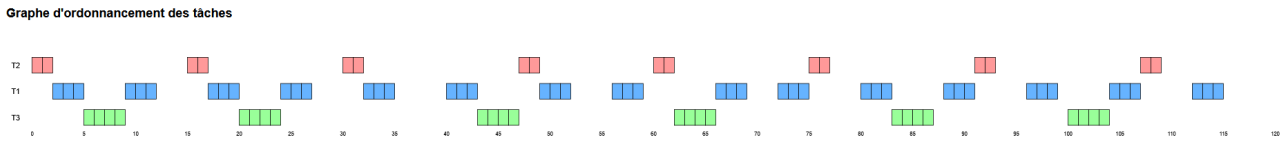

Figure 2: Example of generated scheduling graph.

Exemple 3:

We will now test our program on other task lists to schedule.

Task	Computation time (C)	Period (T)
τ_1	1	10
τ_2	2	10
τ_3	5	20
τ_4	3	20
τ_5	5	40
τ_6	8	40
τ_7	4	80

In this third example, composed of 7 tasks as shown above, we obtain the following schedule. The result is shown below. We can see that all tasks are completed within their allocated time, just like in the previous example.

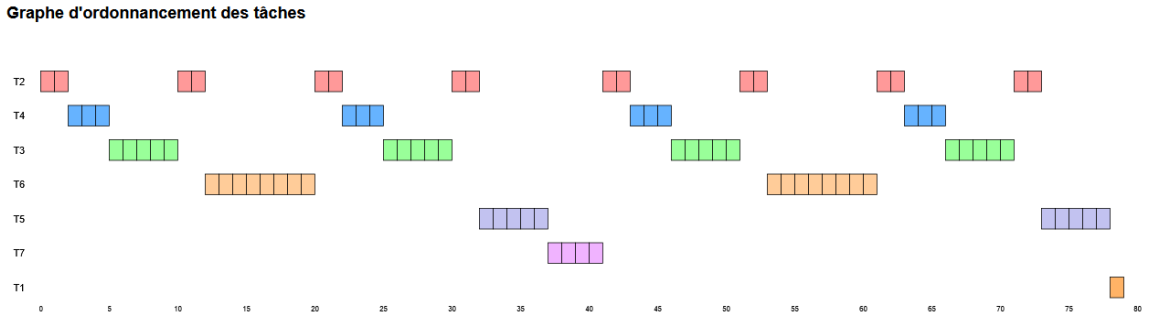

Figure 3: Example of generated scheduling graph.

Exemple 4:

Task	Computation time (C)	Period (T)
τ_1	5	10
τ_2	5	15
τ_3	6	30

Finally, with the example above, we can manually calculate that it is not schedulable in a non-preemptive manner. Indeed, we can see that the program confirms that it is not schedulable.

Conclusion

In this report, we have presented an approach for simulating and evaluating the schedulability of a set of periodic tasks using a non-preemptive scheduling algorithm. By analyzing various task sets and their execution orders, we were able to determine the best scheduling strategy that ensures all tasks are completed within their deadlines.

Through the use of key functions such as `simulate` and `permute`, we were able to efficiently evaluate task execution orders and select the one that minimizes the overall score while meeting all deadlines. The results demonstrated that the algorithm works as intended, successfully scheduling tasks and identifying non-schedulable cases.

For those interested in exploring the full implementation, the complete code is available on our GitHub repository, where additional details and functions can be found.

GitHub Link: `https://github.com/your-repo-here`

I Appendix: Code Implementation

In this appendix, the code for the task scheduling simulation and permutations is provided. Below are the key functions used in our implementation.

Task Structure Definition

Listing 1: Task Structure Definition

```
typedef struct {
    char name[10];        // Task name.
    int T;                // Task period.
    int C;                // Task computation time.
    int remaining_time;   // Remaining time before execution finishes.
    int next_release;     // Time remaining before the task can be executed
       again, calculated from the start of the simulation.
    bool executed;        // Boolean to indicate if the task has been
       executed during the period.
} Task;
```

Simulate Function

In this first part, we will initialize the different variables in our code, such as setting the score to 0, the time to 0, and the parameters for each task:

Listing 2: Simulating task scheduling

```
int simulate(Task tasks[N], int hyperperiod, int schedule[MAX_HYPERPERIOD])
   {
    Task temp[N];
    copy_tasks(temp, tasks);

    int score = 0;
    int current_duration = 0;
    int current_task = -1;

    // Initialize all parameters before the simulation
    for (int i = 0; i < N; i++) {
        temp[i].remaining_time = temp[i].T;
        temp[i].next_release = temp[i].T;
        temp[i].executed = false;
    }
```

Next, we will simulate our tasks to check if they are schedulable in this permutation of tasks.

Listing 3: Simulating task scheduling

```
    // Start the simulation for the duration corresponding to the
       hyperperiod
    for (int t = 0; t < hyperperiod; t++) {
        schedule[t] = -1; // -1 means no task is executed (processor is idle
           )

        // Create a new period if the task has finished its previous period
        for (int i = 0; i < N; i++) {
            if (t == temp[i].next_release) {
                temp[i].remaining_time = temp[i].T;
                temp[i].next_release += temp[i].T;
                temp[i].executed = false;
            }

            // Abort the simulation if a deadline has been missed or there
               is not enough remaining time to finish the task
            if (!temp[i].executed && temp[i].remaining_time < temp[i].C)
                return -1;
            temp[i].remaining_time--;
        }

        if (current_duration > 0) {
            current_duration--;
            schedule[t] = current_task;
            continue;
        }
```

Once a task finishes executing, we will execute the next one and update the score, which corresponds to each task's period time minus the remaining time before its deadline. We will then aim to minimize this value. The goal is to find the scheduling order that allows all tasks to be completed as early as possible relative to their period.

Listing 4: Simulating task scheduling

```
        // When a task finishes execution, it looks for the next task to
           execute
        for (int i = 0; i < N; i++) {
            if (temp[i].remaining_time >= temp[i].C && !temp[i].executed) {
                current_task = i;
                current_duration = temp[i].C - 1;
                temp[i].executed = true;
                schedule[t] = current_task;
                // Calculate the score, which corresponds to executing the
                   task as soon as possible within its period
                // The score is calculated as the period minus the remaining
                    time of each task, and the task with the lowest score is
                    chosen
                score += temp[i].T - temp[i].remaining_time;
                break;
            }
        }
    }

    return score;
}
```

Permutation Function

Listing 5: Permuting task orders

```
void permute(Task tasks[N], int l, int r, int hyperperiod, int* best_score,
    Task best_order[N], int best_schedule[MAX_HYPERPERIOD]) {
    // Base case of the recursion, selects the best permutation that
       simulates the tasks within the hyperperiod without missing deadlines
       and minimizes the score.
    if (l == r) {
        int schedule[MAX_HYPERPERIOD];
        int score = simulate(tasks, hyperperiod, schedule);
        nbre++;
        if (score == -1) return;
        if (score < *best_score) {
            *best_score = score;
            copy_tasks(best_order, tasks);
            copy_schedule(best_schedule, schedule, hyperperiod);
        }
        return;
    }

    // Recursive part for permuting tasks.
    for (int i = l; i <= r; i++) {
        Task temp = tasks[l];
        tasks[l] = tasks[i];
        tasks[i] = temp;

        permute(tasks, l + 1, r, hyperperiod, best_score, best_order,
           best_schedule);

        temp = tasks[l];
        tasks[l] = tasks[i];
        tasks[i] = temp;
    }
}
```