Institut Polytechnique des Sciences Avancées

Real Time Embedded Systems : Final Assignement

VONTHRON Pierre-Louis

Année universitaire 2024-2025

Table des matières

Introduction **2**

Task set and Scheduability **2**

Assumptions **2**

Scheduling Algorithm **3**

Task τ_5 can't miss deadlines 3

Code analysis and plot **3**

A. τ_5 can't miss deadlines 3

B. τ_5 can miss deadlines 6

Conclusion **6**

Introduction

This report presents a solution to the scheduling problem for a given task set . The implementation includes a non-preemptive scheduler that minimizes total waiting time while allowing task τ_5 to miss its deadline. The solution is implemented in Python and includes schedulability analysis, scheduling algorithm, and visualization.

Task set and Scheduability

Here are the characteristics of our task set :

Task	C	T_i
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

With these characteristics, we can calculate the total utilization of the set :

$$U = \sum_{i=1}^{7} \frac{C_i}{T_i} = \frac{2}{10} + \frac{3}{10} + \frac{2}{20} + \frac{2}{20} + \frac{2}{40} + \frac{2}{40} + \frac{3}{80} = 0.8375 \leq 1$$

We can also compute each worse case response times. For this, we have the following formula :

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \leq T_i$$

With $hp(i)$ being the set of tasks that have a higher priority than τ_i. We will consider that the lower the task, the higher the task priority (ie. τ_1 has the highest priority andτ_7 has the lowest priority). As such, we have :

$$\begin{aligned}
&R_1 = C_1 = 2 \quad (\leq T_1 = 10) \\
&R_2 = C_2 + \left\lceil \frac{R_2}{T_1} \right\rceil C_1 = C_2 + \left\lceil \frac{2}{10} \right\rceil C_1 = 3 + 1 \cdot 2 = 5 \quad (\leq T_2 = 10) \\
&R_3 = C_3 + \left\lceil \frac{R_3}{T_1} \right\rceil C_1 + \left\lceil \frac{R_3}{T_2} \right\rceil C_2 = C_3 + \left\lceil \frac{2}{10} \right\rceil (C_2 + C_1) = 2 + 5 = 7 \quad (\leq T_3 = 20) \\
&R_4 = 9 \quad (\leq T_4 = 20) \\
&R_5 = 16 \quad (\leq T_5 = 40) \\
&R_6 = 18 \quad (\leq T_6 = 40) \\
&R_7 = 30 \quad (\leq T_7 = 80)
\end{aligned}$$

With this information, we have not only confirmed that the whole set is scheduable, but also that each independent task can be scheduled.

Assumptions

For our schedule, we will use non-preemptive scheduling, give higher priority to tasks with shorter periods (Rate-monotonic scheduling or RMS) and shorter duration. If two tasks have the same period, the lowest numbered task get priority (ie. τ_4 takes priority over τ_5). Our hyperperiod is the least common multiple between all task periods (here, it is 80 time units).

Scheduling Algorithm

Let's give an overview of our scheduling algorithm. First is **job generation** : each task needs to generate all jobs within the hyperperiod (80 time units).
Second we need to set **priority assignment**. As we have said before, we will prioritize jobs based on their periods (task with shorter period takes priority), then job duration (task with shortest job takes priority), then task number (lowest numbered task takes priority, this is the most arbitrary criteria and we could change this in case we realize that it is not optimal).
The third step, **creation of the scheduling loop** is a key part of our algorithm as it will dictate the codes whole behaviour. Let's detail it step by step :

- check for tasks that are ready to launch a jo every time unit ;
- select the highest priority ready job and execute it fully ;
- track waiting time for other ready jobs during execution ;
- if new jobs are ready, the processor remains idle.

To be sure that our code works correctly we can also track that each task releases the right amount of jobs during the hyperperiod. Normally, using the formula $\frac{H}{T_i}$ (with H being the hyperperiod), we would have :

- 8 jobs for τ_1
- 8 jobs for τ_2
- 4 jobs for τ_3
- 4 jobs for τ_4
- 2 jobs for τ_5
- 2 jobs for τ_6
- 1 job for τ_7

Task τ_5 can't miss deadlines

In the case where τ_5 is not allowed to miss deadlines we will give it priority over all other tasks (the other task's order will stay the same as described earlier).

Code analysis and plot

A. τ_5 can't miss deadlines

First, we generate the task set

```
#%% Task set definition
# Task set: (C, T)
tasks = [
    (2, 10),   # τ1
    (3, 10),   # τ2
    (2, 20),   # τ3
    (2, 20),   # τ4
    (2, 40),   # τ5 — NOT allowed to miss deadline
    (2, 40),   # τ6
    (3, 80)    # τ7
]

HYPERPERIOD = 80
NUM_TASKS = len(tasks)
```

Then we generate the job queue

```
#%% Job queue Generation
# Generate job queues with deadline-critical flag for τ5
job_queues = [[] for _ in range(NUM_TASKS)]
for task_id, (C, T) in enumerate(tasks):
    for i in range(HYPERPERIOD // T):
        release = i * T
        job_queues[task_id].append({
            'release': release,
            'remaining': C,
            'C': C,
            'deadline': release + T,
            'start': None,
            'waiting': 0,
            'is_τ5': (task_id == 4)  # Flag for τ5 jobs
        })

# Initialize schedule
schedule = [['0' for _ in range(NUM_TASKS)] for _ in range(HYPERPERIOD)]
idle_time = 0
t = 0
```

The code loops through each task with each task having an id between 0 and 6. For each task, it computes the number of jobs that will be released during the hyperperiod and creates a job object for the current task with different fields :

- $'release'$ is the time at which the job becomes ready ;
- $'remaining'$ is the remaining units of computation that need to be done ;
- $'C'$ is the original computation time
- $'deadline'$ is the time by which the job must be finished ;
- $'start'$ is the time at which a job begins execution (set as $'None'$ initially) ;
- $'waiting'$ stores how long the job waited ;
- $'is\_\tau_5'$ checks if we are on a job corresponding to task 5 ;

Finally, the "*schedule*" variable will serve as a 2D grid in which the states of each task will be stores at all times (0 for idle state, $*$ when waiting, 1 when active).

Then, we can focus on the scheduling loop. This part of the loop

```
ready_jobs = []
for task_id, jobs in enumerate(job_queues):
    for job in jobs:
        if job['release'] <= t and job['remaining'] == job['C']:
            ready_jobs.append((task_id, job))
            break  # Only consider the earliest pending job per task
```

finds all the jobs that are ready to run. For each task, we find the first job that has been released but hasn't started yet and add one job per task into the *ready_jobs* list.
Then, we check if τ_5 needs urgent priority with this code :

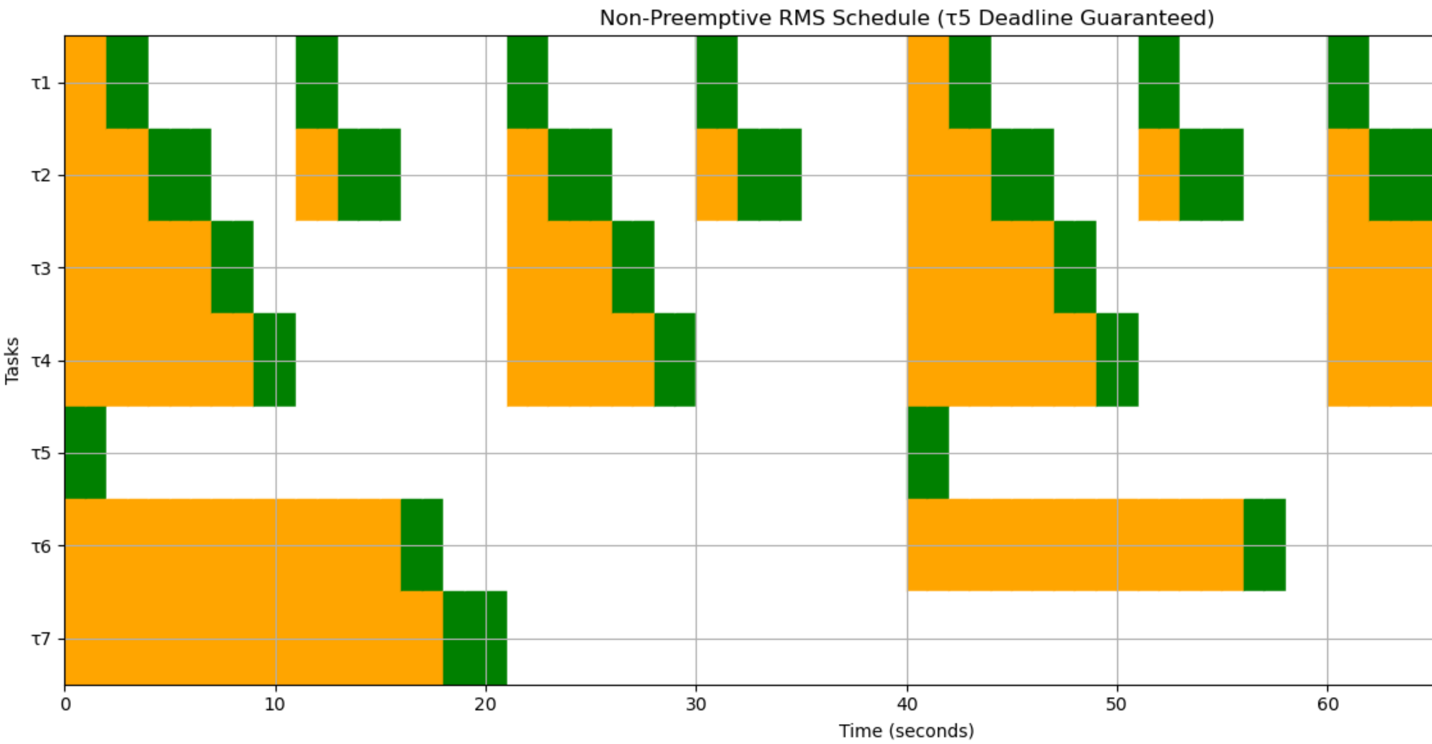
```
ready_jobs = []
for task_id, jobs in enumerate(job_queues):
    for job in jobs:
        if job['release'] <= t and job['remaining'] == job['C']:
            ready_jobs.append((task_id, job))
            break  # Only consider the earliest pending job per task
```

This snippet looks fot τ_5 in *ready_jobs* and, if it's in danger of missing its deadline, $\tau_5\_priority$ is flagged as *True*. If that's the case, we will fall back to RMS.

```python
# Sort jobs: τ5 first (if deadline at risk), then by period (Rate Monotonic)
ready_jobs.sort(key=lambda x: (
    not x[1]['is_τ5'],  # τ5 goes first if priority flagged
    tasks[x[0]][1]      # Then sort by period (T_i)
))

if ready_jobs:
    task_id, job = ready_jobs[0]
    run_time = job['C']
    job['start'] = t
    job['waiting'] = t - job['release']
    job['remaining'] = 0

    # Fill schedule
    for i in range(run_time):
        current_time = t + i
        if current_time >= HYPERPERIOD:
            break
        schedule[current_time][task_id] = '1'  # Mark as running

        # Mark other ready jobs as waiting
        for other_id, other_job in ready_jobs[1:]:
            if other_job['release'] <= current_time and other_job['remaining'] == other_job['C']:
                schedule[current_time][other_id] = '*'

    t += run_time
else:
    idle_time += 1
    t += 1
```

This part of the code picks the first job in the sorted list, record its start and waiting time, marks all time slots where the task is running as 1 while the other ready jobs are blocked and their time slots marked as $*$.

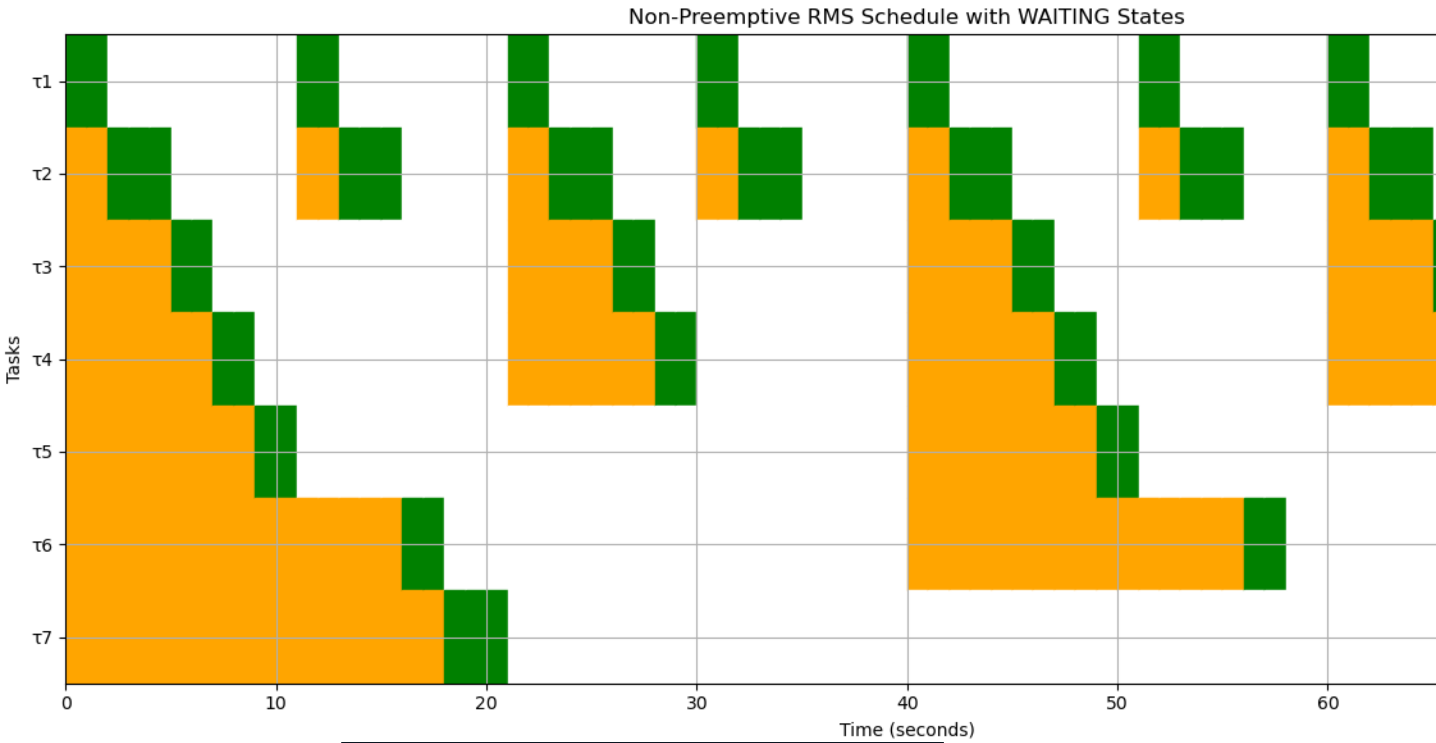
Finally, we get the following results :




```
☑ Non-preemptive schedule with τ5 deadline enforcement.
Total IDLE time: 13
Total WAITING time: 138
τ1 waiting time: 7
τ2 waiting time: 23
τ3 waiting time: 25
τ4 waiting time: 33
τ5 waiting time: 0
τ6 waiting time: 32
τ7 waiting time: 18
Job counts per task: {'τ1': 8, 'τ2': 8, 'τ3': 4, 'τ4': 4, 'τ5': 2, 'τ6': 2, 'τ7':
1}
```

B. τ_5 can miss deadlines

The version of the code for which τ_5's deadline can be missed is pretty similar to the one seen before except for the facts that we don't enforce a strict deadline check for task 5, the fact that we never deviate from RMS or the fact that there's no possibility to block or delay τ_5. With this in mind, we get the following results :




```
☑ Deadline-safe non-preemptive schedule generated.
Total IDLE time: 13
Total WAITING time: 140
τ1 waiting time: 3
τ2 waiting time: 19
τ3 waiting time: 21
τ4 waiting time: 29
τ5 waiting time: 18
τ6 waiting time: 32
τ7 waiting time: 18
```

Conclusion

As we have seen it is possible to schedule and implement our given task set regardless of if τ_5 is allowed to miss deadlines or not. It must be noted that preventing deadline misses for task 5 makes us 2 time units of waiting time. Despite this, we must wonder if it is a real necessity, especially when, you consider that it is way simpler to just code a Rate-monotonic schedule.