Real-Time Embedded Systems

FINAL ASSIGNMENT: SCHEDULE SEARCH

PATTE Julia-Marie

AERO 4 SET – 4TS1

April 2025

Context of the study

This study focuses on verifying the schedulability of a set of periodic tasks and, if the system is schedulable, generating a non-preemptive job-level schedule. The primary objectives are to ensure that no deadlines are missed and to minimize the total waiting time, the delay experienced by jobs due to the execution of other jobs. This minimization is expected to maximize total processor idle time, and this hypothesis will be verified. The implementation is carried out in C++.

Contents

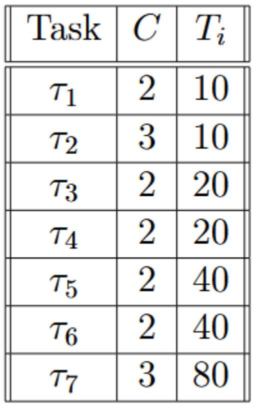
Is it schedulable ?

The first step of this study consists in determining whether the given set of tasks is schedulable.

Schedulability analysis is the process of obtaining functional guarantees for a real-time system. It aims to provide a guarantee that each task in the system will complete its execution within its deadline throughout the operational lifetime of the embedded system. Ensuring schedulability is critical, as missing deadlines may lead to system instability or failure, especially in safety-critical applications.

Therefore, we begin by analyzing the task set to determine whether a feasible non-preemptive schedule exists that satisfies all timing constraints.

The tasks set is the following:



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

Figure 1: Tasks set of the project

Schedulability

The CPU utilization for task τ_i is given by: $U_i = \frac{C_i}{T_i}$.

We have:

$$U_1 = \frac{2}{10},\ U_2 = \frac{3}{10},\ U_3 = \frac{2}{20},\ U_4 = \frac{2}{20},\ U_5 = \frac{2}{40},\ U_6 = \frac{2}{40},\ U_7 = \frac{3}{80}$$

Total utilization $= U_1 + U_2 + U_3 + U_4 + U_5 + U_6 + U_7 = \frac{67}{80} \approx 0.84 < 1$

Total utilization < 1, therefore it is **schedulable**.

Remark. We can easily deduce that if it is schedulable for all tasks then it will also be schedulable for 3 or 4 tasks.

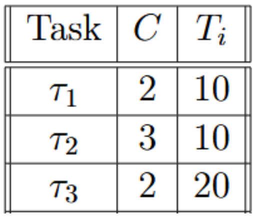
Implementation

After performing the schedulability analysis, we found that the given set of tasks is schedulable. Therefore, the next step is to construct a non-preemptive schedule at the job level. The aim of this schedule is twofold: first, to ensure that no deadlines are missed, and second, to minimize the total waiting time. This is expected to implicitly maximize the processor's idle time, which will be verified during the implementation.

For this purpose, I have chosen to implement the scheduling algorithm in **C++**, which offers both performance and flexibility for this type of low-level scheduling problem.

Initial implementation with three tasks:

To begin with, I implemented and tested the scheduling algorithm using the first three tasks: τ_1, τ_2, and τ_3.



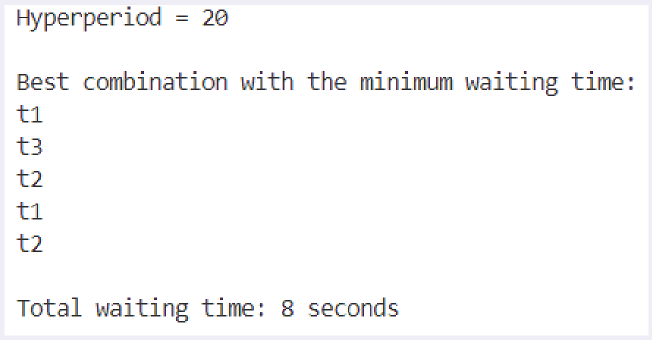
Task	C	T_i
τ_1	2	10
τ_2	3	10
τ_3	2	20

Figure 2: Set for 3 tasks

The algorithm starts by computing the **hyperperiod**, which is defined as the least common multiple (LCM) of the tasks' periods. The hyperperiod represents the time interval after which the schedule pattern repeats, and it serves as the time window within which all possible job instances must be considered.

Once the hyperperiod is computed, the algorithm generates all possible non-preemptive permutations of job execution orders within that interval. For each valid combination, the total waiting time is calculated, defined as the cumulative delay each job experiences due to other jobs executing before it. Finally, the algorithm selects the schedule that results in the **minimum total waiting time**, ensuring all deadlines are met. This approach allows an exhaustive search to identify the most efficient job sequence for the given task set. We find:

Result of the implementation with 3 tasks



```
Hyperperiod = 20

Best combination with the minimum waiting time:
t1
t3
t2
t1
t2

Total waiting time: 8 seconds
```

We observe that the selected order tends to schedule shorter tasks earlier when possible, which reduces the waiting time experienced by subsequent jobs. Moreover, spreading out instances of the same task avoids unnecessary accumulation of waiting times.

Implementation with four tasks:

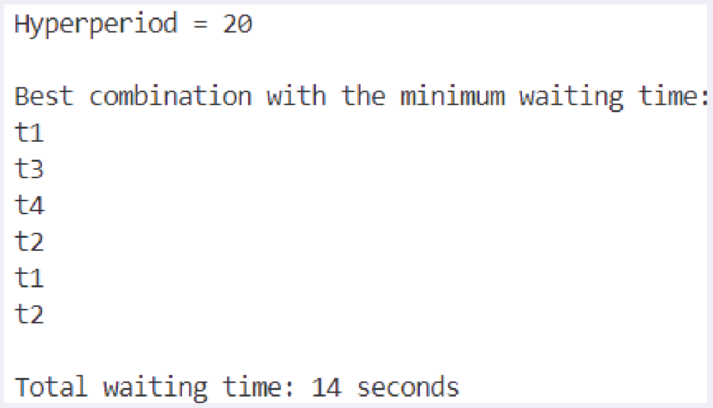
Now, we will see if our algorithm works for four tasks and in particular the first four.

Task	C	T_i
τ_1	2	10
τ_2	3	10
τ_3	2	20
τ_4	2	20

Figure 3: Set for 4 tasks

We obtain:

Result of the implementation with 4 tasks



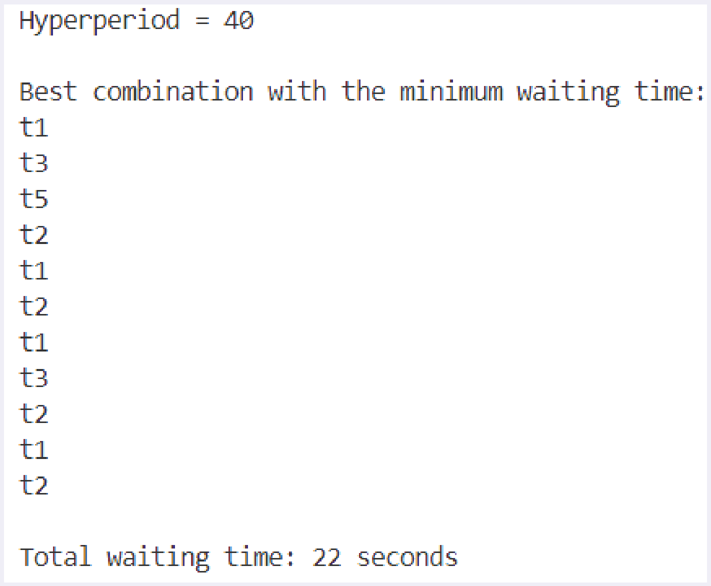
```
Hyperperiod = 20

Best combination with the minimum waiting time:
t1
t3
t4
t2
t1
t2

Total waiting time: 14 seconds
```

When extending the task set to four tasks, the algorithm was still able to find the optimal non-preemptive schedule within a reasonable computation time. However, it is worth noting that the number of possible permutations grows factorially with the number of jobs, which poses a scalability challenge for larger task sets.

Compared to the previous configuration with three tasks (where the total waiting time was 8 seconds), the total waiting time has increased to 14 seconds after introducing a fourth task. This increase is expected due to the higher number of jobs and constraints, but the algorithm still manages to find the sequence with the lowest possible waiting time.

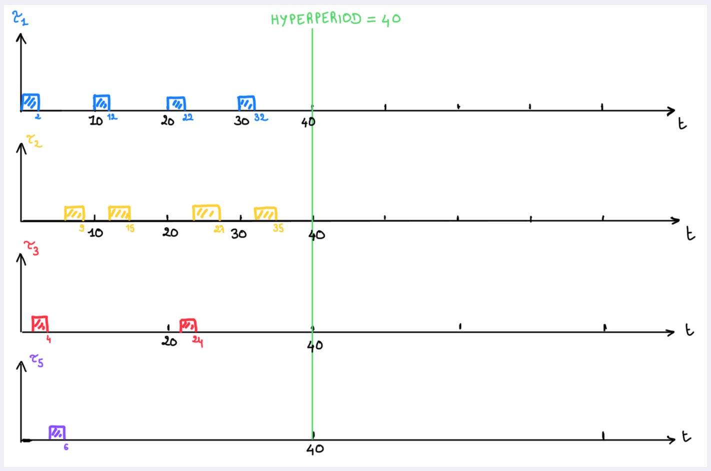
We also try the algorithm for 4 tasks with tasks τ_1, τ_2, τ_3 and especially τ_5 in order to verify that our algorithm works well.

Result of the implementation with 4 tasks and especially τ_5



```
Hyperperiod = 40

Best combination with the minimum waiting time:
t1
t3
t5
t2
t1
t2
t1
t3
t2
t1
t2

Total waiting time: 22 seconds
```

We have plotted all the tasks in order to visualize the proposed combinations:



According to the project specifications, we also investigated a scenario in which task τ_5 is allowed to miss a deadline. This relaxation is particularly useful in systems where certain tasks are non-critical or can tolerate occasional deadline violations, as it enables the scheduler to potentially reduce the total waiting time for the remaining tasks.

However, due to the limitations of our exhaustive search algorithm, we were only able to apply this to a subset of tasks, specifically τ_1, τ_2, τ_3, and τ_5, as running the algorithm with more tasks leads to unbounded execution time. With this reduced set, the computed hyperperiod remains relatively small (40), and even under the relaxed constraint for τ_5, no actual deadline misses occurred as we can see in the final schedule.

This indicates that the system was still schedulable within the given hyperperiod, and that the allowed relaxation did not need to be exploited.

Limitations of the exhaustive search approach:

We then attempted to run the algorithm with more than four tasks. However, we observed that the execution time grows drastically and the program runs indefinitely without producing a result. This behavior highlights the primary limitation of the exhaustive search method, its **computational complexity**. Since the algorithm evaluates all possible permutations of job sequences within the hyperperiod, the number of combinations increases factorially with the number of tasks and their instances.

Moreover, a large **hyperperiod** significantly increases the number of job instances to consider, which further amplifies the number of possible permutations. As a result, not only does the task count impact runtime, but so does the alignment of task periods, especially when they lead to a large least common multiple.

Advantages of the exhaustive search approach:

- Guarantees the **optimal solution**, the combination with the absolute minimum total waiting time.
- Easy to understand and implement for small task sets.
- Provides a reliable reference baseline for testing other heuristics.

Drawbacks of the exhaustive search approach:

- **Not scalable**, the number of permutations explodes with more tasks or a longer hyperperiod.
- **High computation time**, becomes infeasible for real-time use or large systems.
- Difficult to integrate into real embedded systems where quick decisions must be made.

Therefore, while the exhaustive method is effective for small cases, alternative scheduling algorithms or heuristics must be considered for larger task sets to balance between solution quality and computational efficiency.

Response time analysis:

To formally verify the schedulability of the system, we conducted a response time analysis for each individual job. The response time is defined as the difference between the finish time and the release time of a job. For a task set to be considered schedulable, the response time of every job must be strictly less than its corresponding deadline.

For each job instance generated within the hyperperiod, we computed its response time based on the selected non-preemptive schedule. We then verified that this value was less than or equal to the job's deadline. The results obtained for 3 or 4 tasks confirmed that all jobs met their deadlines, thereby validating the correctness and schedulability of the computed schedule under the given constraints.

Conclusion

This study successfully demonstrated the feasibility of non-preemptive scheduling for a set of periodic tasks while minimizing waiting time. Our implementation using exhaustive search in C++ provided optimal schedules for smaller task subsets (3-4 tasks) and confirmed the theoretical schedulability analysis with a CPU utilization of approximately 0.84.

Strengths:

- Guarantees optimal scheduling solutions for the tested task sets
- Provides precise response time analysis confirming deadline compliance
- Successfully minimizes total waiting time in the tested configurations
- Demonstrates that shorter tasks tend to be scheduled earlier when possible, reducing overall system latency

Limitations:

- Exponential computational complexity makes this approach infeasible for larger task sets
- Unable to scale beyond 4 tasks due to the factorial growth of possible permutations
- Impractical for real-world embedded systems with numerous tasks or long hyperperiods

Future improvements:

A more scalable approach using algorithms like Earliest Deadline First (EDF) could handle larger task sets more efficiently. EDF would offer polynomial complexity compared to our exhaustive search, enabling the scheduling of all seven tasks and potentially more complex real-time systems, though potentially sacrificing optimal waiting time minimization for computational feasibility.