Real-time embedded systems

Final assignment: Schedule search

This project aims to design a scheduler for a set of periodic real-time tasks with strict timing constraints. The goal is to minimize the total waiting time while respecting each job's deadline.

Given this 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

Schedulability

First, we check the schedulability of our task set. For that, we need to find:

$$U_i = \frac{C_i}{T_i}$$

Thus, we have:

Task	$\boldsymbol{C_i}$	$\boldsymbol{T_i}$	$\boldsymbol{U_i}$
$\boldsymbol{\tau_1}$	2	10	0.2
$\boldsymbol{\tau_2}$	3	10	0.3
$\boldsymbol{\tau_3}$	2	20	0.1
$\boldsymbol{\tau_4}$	2	20	0.1
$\boldsymbol{\tau_5}$	2	40	0.05
$\boldsymbol{\tau_6}$	2	40	0.05
$\boldsymbol{\tau_7}$	3	80	0.0375

If the sum of all the U_i's is inferior to 1, we can say that the set is schedulable:

$$\sum U_i = 0.2 + 0.3 + 0.1 + 0.1 + 0.05 + 0.05 + 0.0375 = 0.8375 < 1$$

Schedulability analysis

To find the hyperperiod, we need the least common multiple of all the periods of all tasks. Here, 80 can be divided by 10, 20, and 40, so this is the hyperperiod.

We can define how many jobs each task has. For example, τ_1 is executed on a period of 10, so on the hyperperiod, $\tau_1$has 8 jobs.

We have:

Task	C_i	T_i	Number of jobs
τ_1	2	10	8
τ_2	3	10	8
τ_3	2	20	4
τ_4	2	20	4
τ_5	2	40	2
τ_6	2	40	2
τ_7	3	80	1

Total number of jobs for one hyperperiod: 29

We can now compute the response time for each task for the hyperperiod in units of time:

$$RT_i = Number\ of\ job\ s_i \times C_i$$

$$RT_1 = 8 \times 2 = 16\ units$$

$$RT_2 = 8 \times 3 = 24\ units$$

$$RT_3 = 4 \times 2 = 8\ units$$

$$RT_4 = 4 \times 2 = 8\ units$$

$$RT_5 = 2 \times 2 = 4\ units$$

$$RT_6 = 2 \times 2 = 4\ units$$

$$RT_7 = 1 \times 3 = 3\ units$$

We can now determine the total response time for the hyperperiod:

$$RT = \sum RT_i = 16 + 24 + 8 + 8 + 4 + 4 + 3 = 67\ units < \text{hyperperiod (80 units)}$$

Assumptions

To realise this schedule, we make these assumptions:

- Each task is periodic, with its relative deadline equal to its period.
- The execution time is deterministic and equal to the Worst-Case Execution Time.
- All tasks are independent.
- Tasks are released periodically starting from $t = 0$.

Computational complexity

Step 1: 3 tasks

We test with only three tasks to understand the system and optimize the code. We consider a hyperperiod of 30 units of time:

Task	C_i	T_i	Number of jobs
τ_1	7	10	3
τ_2	2	15	2
τ_3	2	30	1

First, the total number of permutations is $6! = 720$ possibilities (for 6 jobs). However, we must consider that jobs within a single task must have a specific order.

That means:

- J1,1 must come before J1,2, which must come before J1,3
- J2,1 must come before J2,2
- J3,1 has no constraint (only one job)

We can therefore calculate for each chain the number of distinct ways of choosing k elements from n elements:

$$C_i = \frac{n!}{k!\,(n-k)!}$$

For the first task of 3 jobs, we look for all possible combinations in 3 positions (3 jobs for the first task) for 6 elements (6 jobs):

$$C_1 = \frac{6!}{3!\,(6-3)!} = \frac{720}{36} = 20$$

We now consider that 3 out of 6 jobs are fixed. For the second task of 2 jobs, we look for all possible combinations in 2 positions (2 jobs for the second task) for 3 elements (6-3 fixed jobs):

$$C_2 = \frac{3!}{2!\,(3-2)!} = \frac{6}{2} = 3$$

So, we now consider that 5 out of 6 jobs are fixed. For the third task of 1 job, we look for all possible combinations in 1 position (1 job for the third task) for 1 element (6-5 fixed jobs):

$$C_3 = \frac{1!}{1!\,(1-1)!} = \frac{1}{1} = 1$$

Finally, to have the total of possible combinations on all the spots:

$$C = C_1 \times C_2 \times C_3 = 60$$

With this technique, we model each task with multiple jobs as an ordered chain (a sequence where the order must be respected).

Now we must impose a second condition after the order of the jobs, so as not to keep the option if the deadline is not met.

Step 2: 7 tasks

To calculate the number of combinations considering just the order of the jobs we have:

$$C_1 = \frac{29!}{8!\,(29-8)!} = 4\,292\,145$$

$$C_2 = \frac{21!}{8!\,(21-8)!} = 203\,490$$

$$C_3 = \frac{13!}{4!\,(13-4)!} = 715$$

$$C_4 = \frac{9!}{4!\,(9-4)!} = 126$$

$$C_5 = \frac{5!}{2!\,(5-2)!} = 10$$

$$C_6 = \frac{3!}{2!\,(3-2)!} = 3$$

$$C_7 = \frac{1!}{1!\,(1-1)!} = 1$$

$$C_{tot} \cong 2.36 \times 10^{18}$$

The number of possibilities is too large to be searched with a computer, so we must impose additional restrictions to lighten the code.

Algorithm

Deadlines

We want our system to respect the deadlines imposed in each period.

Every job must complete its execution before its absolute deadline, which is calculated as: $job\_number * Ti$ where Ti is the period of the corresponding task. Violating this constraint normally results in system failure in a hard real-time context. However, in some variants (like in the example where $\tau 5$ is allowed to miss its deadline), soft real-time scheduling is applied, in which some violations are tolerated to optimize other metrics like the overall waiting time. This compromise allows for greater flexibility but must remain limited to certain non-critical tasks.

Waiting time

For each job, its waiting time is defined as: $Waiting\ time\ =\ Start\ time\ -\ Release\ time$

The goal is to minimize the sum of these waiting times across all tasks. This improves system responsiveness and allows for smoother use of CPU resources, which is especially important for real-time embedded systems where idle times must be minimized.

Backtrack

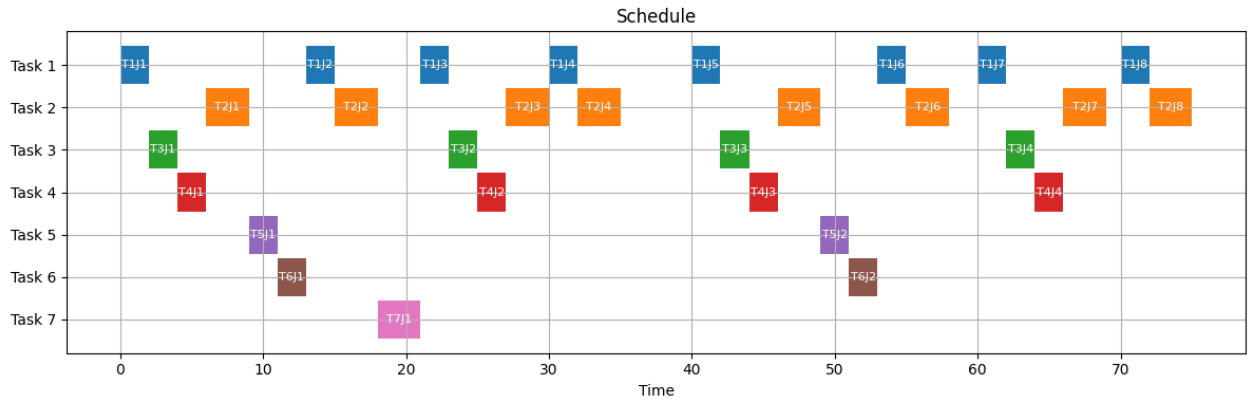
The backtracking algorithm exhaustively explores all valid permutations of jobs while respecting two constraints: the deadlines and the order of arrival of the same task jobs (ex, J1,1 must preceed J1,2). Each step attempts to add a new job in the current sequence and evaluates whether it remains valid. It abandons this branch if it detects that a sequence cannot lead to an optimal solution (for example, if it already exceeds a known waiting time). This backtracking principle, combined with the exploration of state memorization, makes it possible to drastically reduce the number of combinations to explore while guaranteeing the optimality of the solutions found.

Progressive Scheduling

The progressive scheduling approach limits combinatorial complexity by introducing tasks one at a time. A subset of tasks is ordered (e.g., the first 4), and then the next ones are progressively added, recalculating the best permutations at each step. This strategy avoids having to explore the entire search space from the start, while maintaining the quality of the solutions. It is particularly effective in systems with many tasks, exploiting optimal partial solutions to build a global solution.

Results

We plot on a Gantt-like plot to visualize the best schedule (the one that minimizes the waiting time).



Best waiting time: 130 units time, 7 tasks

In the most optimized case, all deadlines are respected, and the jobs within the same task are in order. We display one situation for each minimized waiting time. In this case, if we swap jobs from different tasks that are the same length and consecutive, we will obtain the same waiting time (ex, if we exchange 3.1 and 4.1, nothing changes for the waiting time). We can see that for the minimal waiting time case, we have 130 and 8 units during the process, where no task is performed. (We don't consider the time for 5 units at the end, where all tasks are done)

Case of task 5 missed deadline

This part studies real-time tasks scheduling in scenarios where Task 5 can miss its deadline once, while all other tasks are strictly constrained by their deadlines.

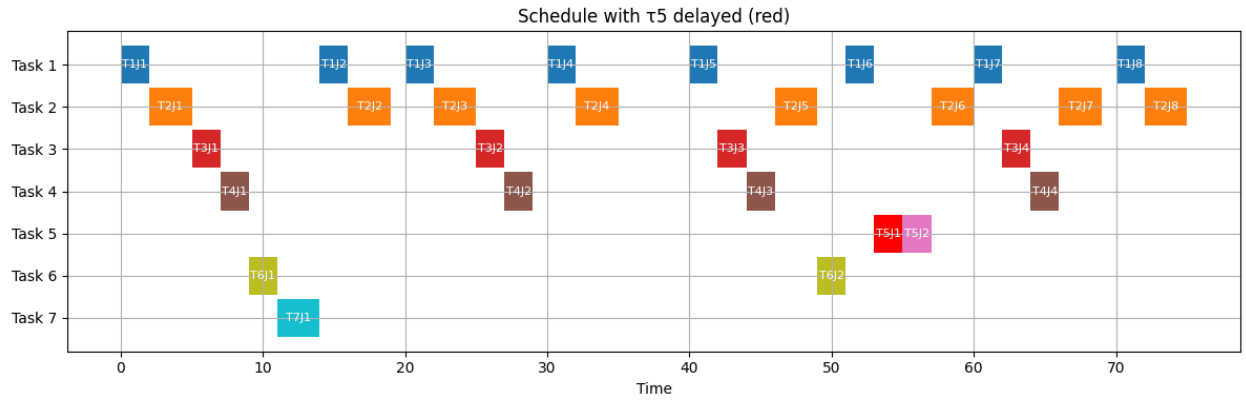
Soft real-time

Now, the scheduler uses a recursive backtracking approach to explore all valid task sequences within the hyper period. The jobs are sorted by release time and tight deadline. Deadline misses are tolerated only for Task 5.

From all generated valid permutations, only sequences are selected that miss exactly one deadline (and that missed deadline is for Task 5) and that have the lowest possible total waiting time. We choose to keep the permutation where the deadline is missed to compare with when it is not. This way, the algorithm is lighter and has fewer permutations to study. We prioritize Task 5 to ensure it is executed by the missed deadline.

This strategy remains exponential in complexity, since it explores permutations of job schedules. If J is the total number of jobs across all tasks in the hyper period, then the search space grows as $J!$. To manage this, we use memorization to prune redundant branches and avoid recalculating known suboptimal paths, and progressive scheduling adds tasks incrementally.

For the best valid permutation found, all tasks except $\tau 5$ meet their deadlines and $\tau 5$ misses exactly one deadline but still scheduled and executed. It is shown in the Gantt chart below. The execution of the late Task 5 is marked in red for visibility.



Best waiting time: 171 units of time

This approach shows that it's possible to schedule all tasks optimally while relaxing one task's deadline. Task 5's deadline miss is controlled and scheduled in a way that doesn't disrupt the feasibility of higher-priority tasks. However, the total waiting time in this relaxed scenario is worse than in the strictly feasible case where no deadlines are missed.

Although allowing a task to miss its deadline increases scheduling flexibility, it may lead to suboptimal placement of its jobs in the schedule. Because $\tau 5$ is allowed to finish late, it may be delayed until the end, accumulating significant waiting time. Other tasks must still meet their deadlines, constraining their placement and limiting opportunities for Task 5 to be efficiently interleaved. Finally, the backtracking algorithm may prioritize feasibility over optimality, so it may accept permutations with higher overall delay in ensuring the one miss.

To sum up, relaxing constraints does not guarantee a better overall performance. In this case, the ability to delay Task 5 introduces a gain in feasibility but often at the cost of greater total waiting time.

Hard real-time

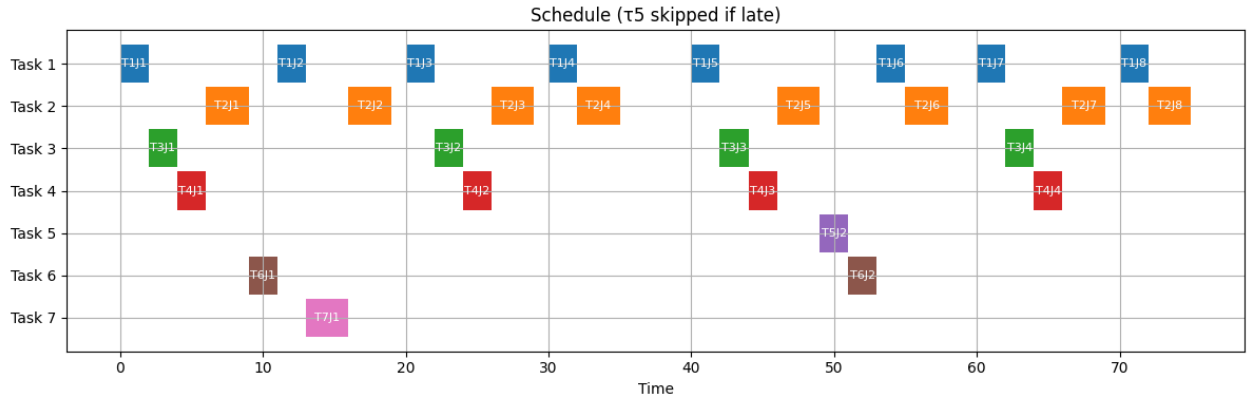
This part analyzes the scheduling under the condition that Task 5 must not be executed if it would miss its deadline, in contrast to the previous relaxed scenario. No deadline violations are allowed.

In this configuration, the scheduling algorithm uses recursive backtracking to explore all feasible job sequences over the system's hyperperiod. However, if the algorithm encounters a situation where Task 5 cannot be completed before its deadline, that job is dropped entirely from the schedule. This drop simulates strict real-time enforcement, where late jobs are considered failed and not executed. It allows for reducing the search space and ensuring strict schedulability.

The backtracking approach still exhibits factorial time complexity with respect to the number of jobs. To manage this, the implementation uses memorization to avoid redundant computations and progressive scheduling, which introduces tasks incrementally.

This aims to minimize the total waiting time across all executed jobs. If a job (like τ5) is skipped, it does not contribute to waiting time. Thus, the scheduler prefers sequences where job executions are feasible and tightly packed to reduce delays.

The best schedule found for this scenario executes all tasks within their deadlines, dropping any τ5 instances that would violate timing constraints. The Gantt chart below shows the selected permutation, where τ5 is absent from any position where its deadline would be missed.



Best waiting time: 109 units of time

This schedule demonstrates the efficiency gained from strict enforcement. By removing infeasible jobs, the system focuses solely on those that can be completed in time, reducing scheduling complexity and idle periods. While the relaxed case (where τ5 is allowed to miss its deadline once) offers more flexibility and a broader range of valid permutations, it often leads to worse total waiting time. In contrast, the strict policy eliminates problematic jobs early, resulting in a more compact and optimal schedule. Allowing a task to exceed its deadline provides additional flexibility to the backtracking algorithm, allowing it to explore more solutions that minimize the total waiting time, while still meeting the deadlines of larger tasks.