Project Report

In422 – Real-Time Information Systems

Final Assignment: Schedule Search

Students:
Simon Lignac

Instructor:
Jasdeep Singh

Promotion 2027 – Aero 4
April 2026
Academic Year 2025–2026

Table of Contents

1 Introduction **1**

2 WCET Measurement of Task τ_1 **2**
2.1 Task Description . . . 2
2.2 Measurement Protocol . . . 2
2.3 Results . . . 2

3 Schedulability Analysis **3**
3.1 Task Set . . . 3
3.2 Assumptions . . . 3
3.3 Hyperperiod . . . 3
3.4 Utilisation Bound (Liu & Layland) . . . 3
3.5 Response Time Analysis . . . 4

4 Optimal Non-Preemptive Schedule **5**
4.1 Objective . . . 5
4.2 Algorithm: Branch and Bound . . . 5
4.3 Results . . . 6

5 Schedule with τ_5 Allowed to Miss Its Deadline **7**
5.1 Problem . . . 7
5.2 Approach . . . 7
5.3 Results and Discussion . . . 7

6 Conclusion **8**

1. Introduction

This report presents the analysis and scheduling of a set of seven periodic real-time tasks on a single processor. The work is split into four parts.

First, we measure the WCET of task τ_1 through statistical sampling. Then we verify that the task set is schedulable. We then build a non-preemptive schedule that minimises total waiting time, and finally we look at how the schedule changes when τ_5 is allowed to miss its deadline.

The full source code is available at: `https://github.com/SimonLignac/Real-Time_Information_Systems_LIGNAC`

Notation. τ_i denotes the i-th task with WCET C_i, period T_i, and implicit deadline $D_i = T_i$. Job $J_{i,k}$ is the k-th instance of τ_i, released at $r_{i,k} = (k-1) \cdot T_i$. All times are in milliseconds unless stated otherwise.

2. WCET Measurement of Task τ_1

2.1 Task Description

Task τ_1 computes the product of two randomly generated 512-bit integers. Execution time is not constant: it varies with operand values and OS-level events such as cache misses or interrupt handling. We therefore estimate the WCET by running a large number of measurements and taking the observed maximum.

2.2 Measurement Protocol

We ran $N = 10{,}000$ trials in Python using `random.getrandbits(512)` to generate operands and `time.perf_counter` for timing. The core measurement loop is as follows:

```
for _ in range(N):
    a = random.getrandbits(512)
    b = random.getrandbits(512)
    t_start = time.perf_counter()
    _ = a * b                              # timed operation
    elapsed_us = (time.perf_counter() - t_start) * 1e6
    timings.append(elapsed_us)

wcet_us = math.ceil(max(timings))          # round up for safety margin
```

Listing 2.1: Core measurement loop (extract from `generate_number/wcet_tau1.py`)

2.3 Results

Table 2.1: Execution time statistics for τ_1 ($N = 10{,}000$)

Indicator	**Value (μs)**
Min	0.500
Q1	0.900
Q2 (median)	1.100
Q3	1.200
Max	83.100
WCET C_1	**84** (rounded up)

The WCET is set to $C_1 = 84$ μs $= 0.084$ ms, rounded up from the observed maximum to add a small safety margin. The distribution is right-skewed: the median is close to 1.1 μs but the worst case reaches 83.1 μs, which illustrates why average execution time cannot be used as a safe bound in real-time systems.

3. Schedulability Analysis

3.1 Task Set

Table 3.1: Periodic task set

Task	C_i (ms)	T_i (ms)	D_i (ms)
τ_1	0.084	10	10
τ_2	3	10	10
τ_3	2	20	20
τ_4	2	20	20
τ_5	2	40	40
τ_6	2	40	40
τ_7	3	80	80

3.2 Assumptions

- Tasks are independent (no shared resources, no precedence).
- All tasks are released simultaneously at $t = 0$ (synchronous model).
- Deadlines are implicit: $D_i = T_i$.
- The system is single-processor and non-preemptive.
- No job exceeds its WCET C_i.

3.3 Hyperperiod

We analyse the schedule over one hyperperiod H, i.e. the least common multiple of all periods. The task set behaviour repeats exactly every H milliseconds, so it is sufficient to check schedulability on $[0, H)$.

$$H = \text{lcm}(10, 10, 20, 20, 40, 40, 80) = 80 \text{ ms}$$

Total number of jobs in one hyperperiod:

$$J = 8 + 8 + 4 + 4 + 2 + 2 + 1 = 29$$

3.4 Utilisation Bound (Liu & Layland)

$$U = \sum_{i=1}^{7} \frac{C_i}{T_i} = \frac{0.084}{10} + \frac{3}{10} + \frac{2}{20} + \frac{2}{20} + \frac{2}{40} + \frac{2}{40} + \frac{3}{80} \approx 0.646$$

For $n = 7$ tasks, the Liu & Layland sufficient condition for Rate Monotonic schedulability is:

$$U \leq n\left(2^{1/n} - 1\right) = 7\left(2^{1/7} - 1\right) \approx 0.729$$

Since $0.646 < 0.729$, the condition is satisfied. The task set is schedulable under RM, and a fortiori under EDF ($U < 1$). The non-preemptive case is confirmed by the response time analysis below.

3.5 Response Time Analysis

The response time of a job in the non-preemptive case is computed using the iterative formula from the assignment:

$$R_{ij}^{n} = C_{ij}^{n} + \max\Big(R_{ij}^{n-1} - \big(a_{ij}^{n} - a_{ij}^{n-1}\big),\ 0\Big) \tag{3.1}$$

Table 3.2: Response time verification – first job of each task

Task	r (ms)	C (ms)	Start (ms)	Finish (ms)	D (ms)	R (ms)
τ_1	0	0.084	0.000	0.084	10	0.084
τ_2	0	3	0.084	3.084	10	3.084
τ_3	0	2	3.084	5.084	20	5.084
τ_4	0	2	5.084	7.084	20	7.084
τ_5	0	2	7.084	9.084	40	9.084
τ_6	0	2	9.084	11.084	40	11.084
τ_7	0	3	14.168	17.168	80	17.168

All response times satisfy $R_i \leq D_i$: **no deadline is missed.**

4. Optimal Non-Preemptive Schedule

4.1 Objective

We want a non-preemptive schedule over $[0, H)$ such that:

1. No deadline is missed.
2. Total waiting time $W = \sum_j (t_{\text{start},j} - r_j)$ is minimised.

Since total busy time $B = \sum_i (H/T_i) \cdot C_i = 51.184$ ms is fixed regardless of the schedule, minimising W directly maximises processor idle time I. We can verify: $B + I = 51.184 + 28.816 = 80$ ms $= H$, which confirms that minimising W is equivalent to maximising I.

4.2 Algorithm: Branch and Bound

The scheduler (`scheduling.py`) uses a Branch-and-Bound approach. At each step it selects a ready job and recurses; branches are pruned as soon as their optimistic lower bound on remaining waiting time exceeds the best solution found so far. Jobs are explored in EDF order to find good solutions early and improve pruning effectiveness.

```
def lower_bound_wait(time, remaining):
    # Optimistic bound: every future job starts as soon as released
    lb, t = 0, time
    for j in sorted(remaining, key=lambda x: x["release"]):
        start = max(t, j["release"])
        lb += start - j["release"]
        t   = start + j["C"]
    return lb

def branch(time, remaining, schedule, total_wait):
    # Cut branch if it cannot beat the current best solution
    if total_wait + lower_bound_wait(time, remaining) >= best_wait:
        return
    # Explore ready jobs in EDF order
    for job in sorted(ready, key=lambda x: x["deadline"]):
        if finish > job["deadline"]:
            continue    # hard deadline violated -- skip
        branch(finish, new_remaining, new_sched, total_wait + wait)
```

Listing 4.1: Pruning and dispatch logic (extract from `scheduling.py`)

Complexity. With $J = 29$ jobs the search space is bounded but potentially large. The EDF ordering and lower-bound pruning keep the number of explored nodes manageable in practice. In the worst case the complexity is $\mathcal{O}(J!)$, but the pruning reduces this significantly.

4.3 Results

Table 4.1: Non-preemptive schedule – summary

Metric	Value
Total waiting time W	82.184 ms
Total idle time I	28.816 ms
Total busy time B	51.184 ms
Missed deadlines	0

5. Schedule with τ_5 Allowed to Miss Its Deadline

5.1 Problem

We now relax the deadline constraint on τ_5: its jobs may be delayed past their deadline if this reduces the total waiting time W. All other tasks must still meet their deadlines.

5.2 Approach

The modified scheduler (`scheduling_relax_t5.py`) is identical to the standard version, with a single change in the feasibility check: the hard deadline is only enforced for tasks other than τ_5.

```
SOFT_TASK = "T5"    # this task is allowed to miss its deadline

# Original check (hard deadline enforced for all tasks):
#    if finish > job["deadline"]: continue

# Modified check (T5 deadline is now a soft constraint):
if job["task"] != SOFT_TASK and finish > job["deadline"]:
    continue     # hard deadline violated -- skip

# Record tardiness for reporting
tardiness = max(0, finish - job["deadline"])
```

Listing 5.1: Modified feasibility check (extract from `scheduling_relax_t5.py`)

5.3 Results and Discussion

Running the modified scheduler yields $W' = 82.184$ ms, identical to the standard result. No τ_5 job misses its deadline in the optimal solution: the scheduler already places τ_5 jobs within their deadlines while achieving the minimum total waiting time. This means that relaxing τ_5's deadline constraint brings no benefit for this particular task set – the slack in τ_5's period ($T_5 = 40$ ms) is already sufficient to accommodate it without conflicting with tighter tasks.

6. Conclusion

This project covered the full scheduling pipeline for a set of seven periodic tasks. The WCET of τ_1 was measured at 84 μs over 10,000 runs. The task set has a utilisation of 0.646, well below the Liu & Layland bound of 0.729, confirming schedulability under both RM and EDF. The non-preemptive Branch-and-Bound schedule produces no deadline misses, a total waiting time of 82.184 ms, and leaves 28.816 ms of idle time over the 80 ms hyperperiod. Allowing τ_5 to miss its deadline yields no improvement for this task set, as the optimal schedule already respects all deadlines including τ_5.

Bibliography

[1] C.L. Liu and J.W. Layland, *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*, Journal of the ACM, 20(1):46–61, 1973.

[2] G. Buttazzo, *Hard Real-Time Computing Systems*, 3rd ed., Springer, 2011.