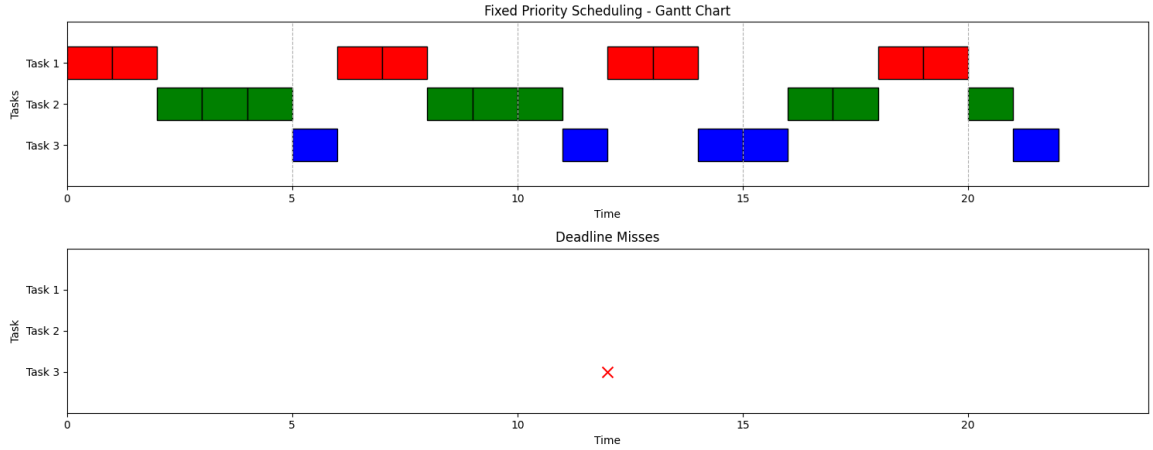
CSC112 Lab2: Real-Time Scheduling Algorithms

You are given a Python program "FP Scheduling.py" that simulates the Fixed-Priority Rate Monotonic scheduling algorithm for a given taskset in one hyperperiod, detects any possible deadline misses, and plots both the scheduling Gantt chart and deadline misses using Matplotlib. Assume all tasks are released at time 0. For example, for this taskset:

```
tasks = [
    Task(1, 6, 2, 6, 3),  # Task 1: Period=6ms, Execution Time=2ms,
Deadline=6ms, Priority=3 (highest)
    Task(2, 8, 3, 8, 2),  # Task 2: Period=8ms, Execution Time=3ms,
Deadline=8ms, Priority=2
    Task(3, 12, 3, 12, 1)  # Task 3: Period=12ms, Execution Time=3ms,
Deadline=12ms, Priority=1 (lowest)
]
```

The resulting Gantt chart is shown below, with a deadline miss for Task 3 at time instant 12ms. Here Task 1 has the highest priority of 3, and Task 3 has the lowest priority of 1, based on Rate Monotonic priority assignment. At time 12ms, Task 3 has not finished execution at its deadline, since it has executed for 2ms but its execution time is 3ms.



Explanations of this line:

`current_task = max(ready_tasks, key=lambda t: t.priority)`

The `max()` function iterates over all elements in `ready_tasks`. For each task `t`, it evaluates the value of `t.priority` using the `lambda` function, which is an anonymous (inline) function that takes one argument (`t`, representing a task object) and returns its priority attribute. It compares these priority attribute values and identifies the task with the highest priority, and finally assigns this task to the variable `current_task`. Note that the `key` parameter of the `max()` function expects a `callable`, a function that takes an element from the `iterable` (`ready_tasks`) and returns a value to be used for comparison. An alternative method is to define a separate function to extract the priority:

```
def get_priority(task):
      return task.priority
current_task = max(ready_tasks, key=get_priority)
```

Your job is to implement (1) *Earliest Deadline First (EDF)* scheduling, where the task with the smallest absolute deadline has highest priority, assuming that each task's relative deadline is equal

to its period. (2) *Least Laxity First (LLF)* scheduling, where the task with the smallest laxity has highest priority, defined as:

```
Laxity=AbsoluteDeadline-(Current Time+Remaining Execution Time)
```

Hints: you need to:

1. remove the task attribute *priority*, since it is not useful for EDF or LLF scheduling.
2. select the ready task in the queue with the smallest *absolute deadline* (or smallest *laxity*) as the highest priority task to run at each time instant within the hyperperiod.

Please submit the following on Canvas:

1. For EDF scheduling: Two Python files named “EDF Scheduling nodeadlinemiss.py”, and “EDF Scheduling deadlinemiss.py”, each with the same Python code but containing a different taskset as the test case: one taskset that is schedulable, and the othe taskset that is not schedulable and has deadline misses in the hyperperiod. (Please use a small number of tasks with small hyperperiod, perhaps by reusing the above taskset and modifying the attributes slightly.)
2. For LLF scheduling: Two Python files named “LLF Scheduling nodeadlinemiss.py”, and “LLF Scheduling deadlinemiss.py”, similar to EDF scheduling.
3. A short PDF report explaining the code you have written and the execution results, including each taskset’s parameters and the corresponding Gantt chart.