**Task scheduling for embedded system**

We will implement a binomial heap-based priority queue and solve a task scheduling problem using priority queues.

# Background

An embedded system is a computer system performing dedicated functions within a larger mechanical or electrical system. Embedded systems range from portable devices such as Google Glasses, to large stationary installations like traffic lights, factory controllers, and complex systems like hybrid vehicles, and avionic. Typically, the software of an embedded system consists of a set of tasks (threads) with timing constraints. Typical timing constraints are release times and deadlines. A release time specifies the earliest time a task can start, and a deadline is the latest time by which a task needs to finish. One major goal of embedded system design is to find a feasible schedule for the task set such that all the timing constraints are satisfied.

# Task scheduler

We assume that the hardware platform of the target embedded systems is a single processor with m identical cores, Core1, Core2, …, Corem. The task set V={v1, v2, …, vn} consists of n independent, non-pre-emptible tasks. A non-pre-emptible task cannot be pre-empted by another task during its execution, i.e., it runs to completion without being interrupted. Each task vi (i=1, 2, …, n) has four attributes: a unique task name vi, an execution time ci, a release time ri, and a deadline di (di>ri). All the execution times, the release times and deadlines are non-negative integers. You need to design a task scheduler and implement it in C. Your task scheduler uses EDF (Earliest Deadline First) heuristic to find a feasible schedule for a task set. A schedule of a task set specifies when each task starts and on which core it is executed. A feasible schedule is a schedule satisfying all the release time and deadline constraints.

The problem of finding a feasible schedule for this task scheduling problem is NP-complete. It is widely believed that an NP-complete problem has no polynomial-time algorithm. However, nobody can prove it.

First, we introduce two definitions: scheduling point and ready task.

* A scheduling point is a time point at which a task can be scheduled on a core, In other words, a scheduling point is either the release time or the completion time of a task.
* A task vi (i=1, 2, …, n) is ready at a time t if tri holds.

The EDF scheduling heuristic works as follows:

* At each scheduling point ti (titi+1, i=1, 2, …), among all the ready tasks, find a task with the smallest deadline, and schedule it on an idle core such that its start time is minimized. Ties are broken arbitrarily.

Since this task scheduling problem is NP-complete, the EDF heuristic is not guaranteed to find a feasible schedule even if a feasible schedule exists.

# Example One

Consider a set S1 of 6 independent tasks whose release times and deadlines are shown in Table

1. The target processor has two identical cores. A feasible schedule of the task set by using EDF scheduling strategy is shown in Figure 1.

Table 1: A set S1 of 6 tasks with individual release times and deadlines

|  |  |  |  |
| --- | --- | --- | --- |
| Task | Execution time | Release time | Deadline |
| v1 | 4 | 0 | 4 |
| v2 | 4 | 1 | 5 |
| v3 | 5 | 3 | 10 |
| v4 | 6 | 4 | 11 |
| v5 | 4 | 6 | 13 |
| v6 | 5 | 6 | 18 |

Core1

v1

v3

v5

v2

v4

v6

Core2

Time 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figure 1: A feasible schedule for S1

# Example Two

Consider a set S2 of 6 independent tasks whose release times and deadlines are shown in Table

1. The target processor has two identical cores. A schedule of the task set by using EDF scheduling strategy is shown in Figure 2. As we can see, in the schedule, v6 finishes at time 16

and thus misses its deadline. Therefore, the schedule is not feasible. However, a feasible schedule, as shown in Figure 3, does exist.

Table 2: A set of tasks with individual release times and deadlines

|  |  |  |  |
| --- | --- | --- | --- |
| Task | Execution time | Release time | Deadline |
| v1 | 4 | 0 | 4 |
| v2 | 4 | 1 | 5 |
| v3 | 5 | 3 | 10 |
| v4 | 6 | 4 | 11 |
| v5 | 4 | 9 | 16 |
| v6 | 5 | 10 | 15 |

Core1

v1

v3

v5

v2

v4

v6

Core2

Time 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figure 2: An infeasible schedule constructed by the EDF scheduling strategy

Core1

v1

v3

v6

v2

v4

v5

Core2

Time 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Figure 3: A feasible schedule for S2

# The TaskScheduler function

This function gets a task set from a file named file1, constructs a feasible schedule for the task set on a processor with m identical cores by using the EDF strategy, and writes the feasible schedule to file2. If a feasible schedule is found, it returns 1. Otherwise, it returns 0.

Both file1 and file2 are text files. file1 contains a set of independent tasks each of which has a name, an execution time, a release time and a deadline in that order. Task names, execution times, release times and the deadlines are a string of digits between 0 and 9. All the release times are non-negative integers, and all the execution times and the deadlines are natural numbers. The format of file1 is as follows:

v1 c1 r1 d1 v2 c2 r2 d2 … vn cn rn dn

Two adjacent attributes (task name, execution time, release time and deadline) are separated by one or more white space characters or a newline character.

For simplicity, we assume that all the task names in file1 are distinct. Therefore, we

don’t need to check for duplicates.

The TaskScheduler function needs to handle the following possible cases properly when reading from file1 and writing to file2:

* 1. file1 does not exist. In this case, print “file1 does not exist” and the program

terminates.

* 1. file2 already exists. In this case, overwrite the old file2.
  2. The task attributes (task name, execution time, release time and deadline) of file1 do not follow the formats as shown before. In this case, print “input error when reading the attribute of the task X” and the program terminates, where X is the name of the task with an incorrect attribute.

file2 has the following format: v1 p1 t1 v2 p2 t2 … vn pn tn

where each vi (i=1, 2, …, n) is the task name, pi is the name of the core where vi is scheduled, and ti is the start time of the task vi in the schedule. In file2, all the tasks must be sorted in non-decreasing order of start times.

The task scheduler needs to use binomial heap-based priority queues. A priority queue has a header which stores the number of tasks in the heap and other implementation dependent information.

Each node in the heap stores the priority (key) and the attributes of a task. The attributes of a task include its name, execution time, release time and deadline. For simplicity, we use an integer to denote the name of task.