Assignment Three

Objectives

- Understand how the binomial heap-based priority queue works
- Understand how to use priority queues to solve a problem

Admin

Marks 10 marks. Marking is based on the correctness and efficiency of your code. Your code must be well commented.
Group? This assignment is completed individually.
Due Time 23:59:59pm Wednesday 17 April 2019.
Late Submissions Late submissions will not be accepted!

In this assignment, you 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 $V=\{v_1, v_2, \ldots, v_n\}$ 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 v_i (i=1, 2, …, n) has four attributes: a unique task name v_i, an execution time c_i, a release time r_i, and a deadline d_i ($d_i>r_i$). 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 v_i (i=1, 2, …, n) is ready at a time t if $t \geq r_i$ holds.

The EDF scheduling heuristic works as follows:

- At each scheduling point t_i ($t_i \leq t_i+1$, 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 S_1 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 S_1 of 6 tasks with individual release times and deadlines

Task	Execution time	Release time	Deadline
v_1	4	0	4
v_2	4	1	5
v_3	5	3	10
v_4	6	4	11
v_5	4	6	13
v_6	5	6	18

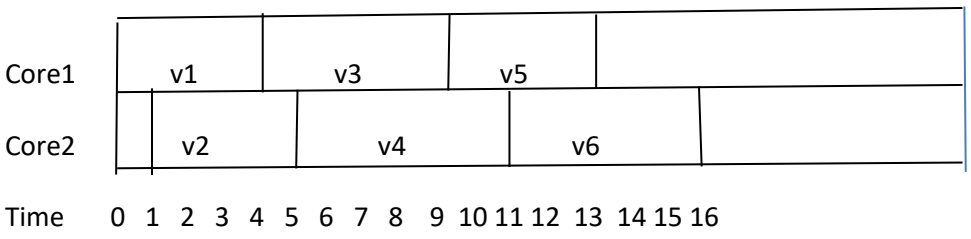

Figure 1: A feasible schedule for S_1

Example Two

Consider a set S_2 of 6 independent tasks whose release times and deadlines are shown in Table 2. 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, v_6 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
v_1	4	0	4
v_2	4	1	5
v_3	5	3	10
v_4	6	4	11
v_5	4	9	16
v_6	5	10	15

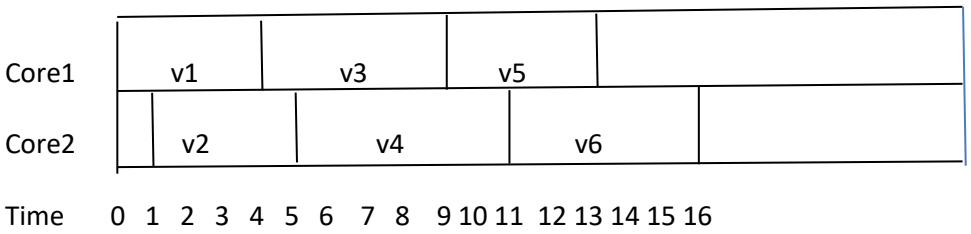

Figure 2: An infeasible schedule constructed by the EDF scheduling strategy

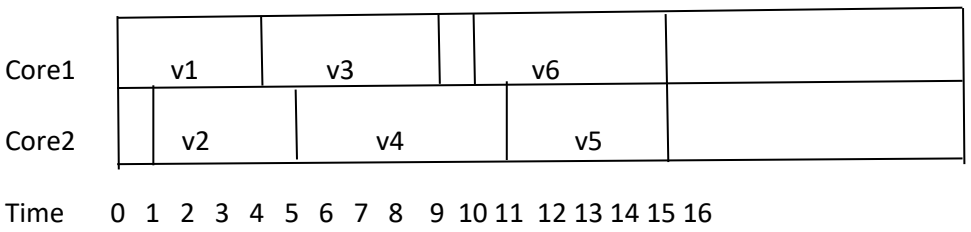

Figure 3: A feasible schedule for S_2

The TaskScheduler function

You need to write a task scheduler function named **TaskScheduler**. A prototype of **TaskScheduler** is shown as follows:

```
int TaskScheduler(char *file1, char *file2, int m) {};
```

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:

$v_1\ c_1\ r_1\ d_1\ v_2\ c_2\ r_2\ d_2\ \ldots\ v_n\ c_n\ r_n\ d_n$

Two adjacent attributes (task name, execution time, release time and deadline) are separated by one or more white space characters or a newline character. A sample file1 is shown here.

For simplicity, you may assume that all the task names in file1 are distinct. Therefore, you 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.
2. file2 already exists. In this case, overwrite the old file2.
3. 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:

$v_1\ p_1\ t_1\ v_2\ p_2\ t_2\ \ldots\ v_n\ p_n\ t_n$

where each v_i (i=1, 2, …, n) is the task name, p_i is the name of the core where v_i is scheduled, and t_i is the start time of the task v_i in the schedule. In file2, all the tasks must be sorted in non-decreasing order of start times. A sample file2 is shown here.

Your 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. The data type of a heap header is defined as follows:

```
typedef struct BinomialHeap{
        int  size;     // count of tasks in the heap
        …              // you need to add additional fields here
} BinomialHeap;
```

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.

The data type for heap nodes is defined as follows:

```
typedef struct HeapNode {
        int key; // key of this task
```

```
        int TaskName; // task name
        int Etime; // execution time of the task
        int Rtime; // release time of the task
        int Dline; // deadline of the task
        …          // you need to add additional fields here
} HeapNode;
```

Therefore, you also need to implement the following functions of a binomial heap-based priority queue:

1. void Insert(BinomialHeap *T, int k, int n, int c, int r, int d). This function inserts a new task into a heap T, where k, n, c, r and d are the key, name, execution time, release time, and deadline of the task, respectively.

2. HeapNode *RemoveMin(BinomialHeap *T). This function removes a task with the smallest key from the heap T and returns the node containing the task.

3. int Min(BinomialHeap *T). This function returns the smallest key of all the tasks in the heap T without modifying the heap T.

The prototypes of all the above-mentioned function and some other basic functions are included in the template file **MyTaskScheduler.c**. In addition to the above functions, you may add your own helper functions and fields (variables) to the file MyTaskScheduler.c.

Note that you must implement a binomial heap-based priority queue. Any other priority queues are not acceptable.

Time complexity requirement

You need to include the time complexity analysis of each function as comments in your program. The time complexity of your scheduler is required to be no higher than O(n*log n), where n is the number of tasks (**Hints: use two binomial heap-based priority queues, one priority queue with release times as keys and one priority queue with deadlines as keys**). There is no specific requirement on space complexity. However, try your best to make your program space efficient.

When analysing the time complexity of your task scheduler, you can assume that the time complexity of each function (insertion, deletion, removeMin and merge) on a binomial heap is O(log n), where n is the number of tasks in the binomial heap.

How to submit your code?

a. Go to the Assignment page
b. Click on Assignment Specifications
c. Click on Make Submission
d. Submit your MyTaskScheduler.c file that contains all the code.

Plagiarism

This is an individual assignment. Each student will have to develop their own solution without help from other people. In particular, it is not permitted to exchange code or pseudocode.

You are not allowed to use code developed by persons other than yourself. All work submitted for assessment must be entirely your own work. We regard unacknowledged copying of material, in whole or part, as an extremely serious offence. For further information, see the Course Information.