Distributed Real-Time Systems (DRS855)
Matlab Exercise # 1

Lab Assistant

Name:	Andreas Persson
Email:	`andreas.persson@ide.hh.se`
Room:	E527

The assignments should be solved individually or in groups of two.

Notation

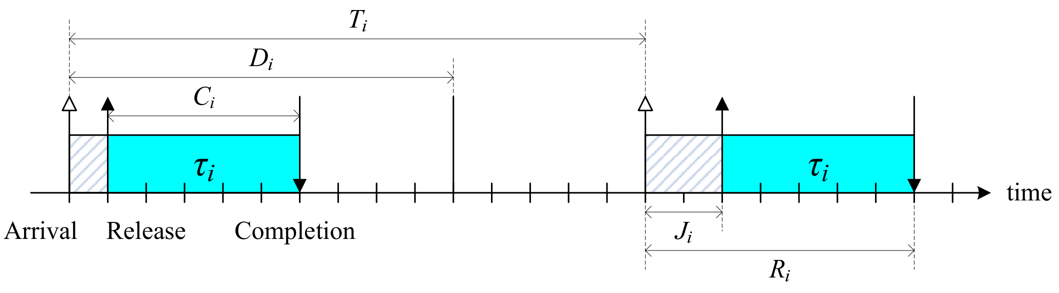

Figure 1: Typical parameters of a real-time task

The timely behavior of a real-time task can be illustrated using a time line as in Figure 1, where the filled regions indicate that task τ_i is executing. In this exercise, we will use the following notation for describing tasks:

T_i is the period of task τ_i. That is, the *inter-arrival time* of instances of task τ_i.

C_i is the Worst Case Execution Time (WCET) of task τ_i.

D_i is the relative deadline of task τ_i. That is, the maximum allowed time between the arrival and the completion of an instance of task τ_i.

J_i is the worst case jitter for task τ_i.

R_i is the worst case response time for task τ_i.

The goal of schedulability analysis is to find R_i and to verify that $R_i \leq D_i$.

The TORSCHE Scheduling Toolbox for Matlab

TORSCHE (Time Optimization of Resources, Scheduling) is a freely (GNU GPL) available toolbox for Matlab developed at the Czech Technical University in Prague[1]. TORSCHE is written in object oriented Matlab Programming Language and provides functions and structures for describing tasks and scheduling problems.

The toolbox was originally not intended for real-time scheduling, but for optimization of control applications and parallel algorithms. The goal for most scheduling algorithms shipped with TORSCHE is to minimize the total execution time for a set of tasks, where each task is executed only once (e.g. for finding the shortest achievable sampling period of a filter implemented on given set of processors).

However, the latest release of the toolbox includes (primitive) support for real-time scheduling of periodic tasks. This is the part of the toolbox that we will be using today. The toolbox has been extended with a simple simulator that allows you to implement and evaluate your own scheduling algorithms.

Getting started

- Download the scheduling toolbox from the course home page (`http://www.hh.se/staff/tola/papers/lab1/scheduling.zip`)
- Extract the folder somewhere on your computer.
- Start Matlab.
- Add the toolbox and the extensions to the Matlab search path:
  1. Open the “Set Path” dialog (“File” → “Set Path...”).
  2. Browse for the folder and add it to the folder list.
- Test your setup by typing `help scheduling` in the Command Window.

Tasks

The fundamental construct in the toolbox is the task object. Periodic tasks are tasks objects with additional attributes. Periodic tasks are created using the `ptask` command, with the following syntax (parameters contained inside the square brackets are optional).

```
t = ptask(Name,ProcTime,Period[,ReleaseTime[,Deadline
         [,Duedate[,Weight[,Processor]]]]])}
```

The `ptask` command is a constructor of object `ptask` and returns a new object. In the following example, a new `ptask` object is bound to the variable `t1`. Type the following in the Matlab Command Window:

```
>> t1 = ptask('task1', 2, 10)
```

The following should be printed on the screen:

```
Task "task1"
 Processing time:  2
 Release time:     0
 Period:          10
```

As seen in the example, `ReleaseTime` is automatically set to zero. The other optional parameters will be undefined if they are not given as arguments to the constructor command.

[1]http://rtime.felk.cvut.cz/scheduling-toolbox/

`Name`	Label of the task
`ProcTime`	Processing time (C_i in Figure 1)
`Period`	Period (T_i in Figure 1)
`ReleaseTime`	Release time, or arrival time
`DeadLine`	Deadline (D_i in Figure 1)
`DueDate`	A ”softer” deadline. Useful for value-based scheduling
`Weight`	The priority of the task with respect to other tasks
`Processor`	Specifies dedicated processor on which the task must be executed

Table 1: Attributes of the `ptask` object.

Task attributes can be accessed using “dot notation”, familiar from other object oriented programming languages. The available attributes are given in Table 1. Try the following commands:

```
>> t1.Deadline

ans =

   Inf

>> t1.DeadLine = 8
Task "task1"
 Processing time: 2
 Release time:    0
 Deadline:        8
 Period:          10
```

A graphical interpretation of a task is provided by the `plot` command. The resulting plot should look like the one in Figure 2. For more information, type `help ptask/plot` in the Command Window.

```
>> plot(t1)
```

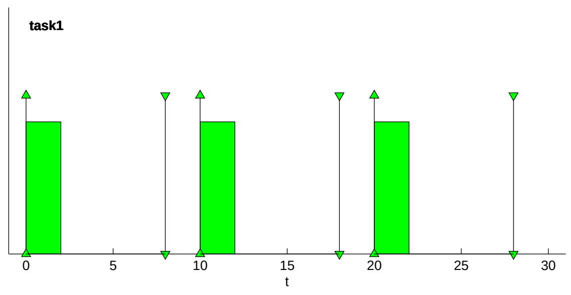

Figure 2: Graphical representation of a single periodic task

Sets of tasks

`ptask` objects can be grouped into sets of tasks. A set of tasks is an object of type `taskset` and is created by the `taskset` command. Enter the following commands in the Command Window:

```
>> t1 = ptask('task1', 2, 10, 0, 10);
>> t2 = ptask('task2', 4, 15, 0, 15);
>> t3 = ptask('task3', 10, 35, 0, 35);
>> T  = taskset([t1,t2,t3])
Set of 3 tasks
```

Sets of tasks can be graphically represented using `plot`. The resulting plot should look something like the one shown in Figure 3. For more information, type `help taskset/plot` in the Command Window.

```
>> plot(T)
```

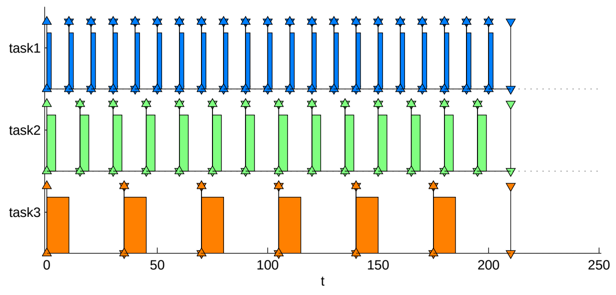

Figure 3: Graphical representation of a set of tasks

From a `taskset` object, you can access task attributes as arrays using “dot notation”. Items order in the arrays is the same as tasks order in the set of the tasks. This works for all task attributes listed in Table 1. Try the following two commands:

```
>> T.Name

ans =

    'task1'    'task2'    'task3'

>> T.ProcTime

ans =

     2     4    10
```

The `count` command returns the number of tasks contained in a given `taskset` object:

```
>> count(T)

ans =

     3
```

Rate Monotonic Scheduling

The *Rate Monotonic* (RM) priority ordering assigns priorities to tasks according to their periods. Specifically, task with smaller periods gets higher priority. Tasks with higher priority can preempt lower priority tasks. For RM schedulability analysis to work, all tasks must be periodic, independent and have deadlines equal to their periods.

Schedulability analysis

The *utilization* of a system of n tasks is the fraction of total execution time spent on executing the tasks, i.e. the computational load of the system. An upper utilization bound U of a system with n periodic tasks can be derived as:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}.$$

It can be shown that a set of n independent periodic tasks, with deadlines equal to periods, will always meet its deadlines when scheduled using the RM algorithm, if

$$U \leq n\left(2^{\frac{1}{n}} - 1\right).$$

Exercise 1

In this exercise, you will define a Matlab function `utilization` for testing this *sufficient* condition for RM schedulability. The function should take a `taskset` object `T` as argument and return the utilization `u`, and a boolean value `s`, indicating whether the condition is satisfied or not. The function prototype is:

```
function [u, s] = utilization(T)
```

1. Put the function prototype in a file called `utilization.m` and define the function by assigning appropriate values to `u` and `s` (*hint:* In Matlab, boolean variables are represented as normal numbers, zero is interpreted as `False` and all non-zero values are interpreted as `True`).
2. Test your function by calculating the utilization for the set of tasks that we defined in the previous sections. The taskset should have the following specification:

Name	C	T
`task1`	2	10
`task2`	4	15
`task3`	10	35

   If the function is correct, the result should be:

```
>> [u, s] = utilization(T)

u =

    0.7524


s =

     1
```

3. What happens if we increase the WCET of `task3` from 10 to 18? What is the new utilization bound? Is the condition for RM schedulability still satisfied?

Earliest-Deadline-First Scheduling

Earliest deadline first (EDF) scheduling is a dynamic scheduling algorithm. Whenever a scheduling event occurs (a task finishes or a new task is released) the set of tasks ready for execution will be searched for the task closest to its deadline. The task with the earliest deadline is selected for execution.

When scheduling periodic tasks that have deadlines equal to their periods, EDF has a utilization bound of 1. That is, EDF can guarantee that all deadlines are met provided that the total processor utilization is not more than 100%. Compared to fixed priority scheduling techniques (like Rate Monotonic Scheduling), EDF can guarantee all deadlines in systems with higher computational loads.

Exercise 2

In this exercise we will use a simulation framework built on top of the TORSCHE toolbox. Your task is to implement and evaluate the EDF scheduling algorithm. There will be a demonstration of the simulation framework during the lab session.

At `http://www.hh.se/staff/tola/papers/lab1/`, you will find three Matlab m-files. Download these files and save them in your Matlab working directory.

`simulate.m` is the scheduling simulation engine.

`rm.m` is an implementation of the Rate Monotonic scheduling algorithm using the simulation engine.

`lab1.m` is a test bench for user-defined scheduling algorithms.

Simulations are performed by calling the `simulate` function. This function takes a `taskset` object as argument and returns a new `taskset` object containing scheduling information (start and stop times of task instances). Apart from the input set of tasks, the `simulate` function needs three *function references*. Function references are pointers to Matlab functions. It is these functions that define the scheduling algorithm. The file `rm.m` contains an implementation of the Rate Monotonic scheduling algorithm using the `simulate` function. This file contains a lot of comments on how to use the simulation framework.

1. run the simulation test bench by typing `lab1` in the Matlab Command Window. This will start a simulation using the RM algorithm. After a while, you should see something like:

   ```
   >> lab1
   *** Simulation finished ***
     - Base period is 210
     - Processor utilization is 0.7524
     - Worst-case response times:
       task1 : 2
       task2 : 6
       task3 : 24
     Taskset is schedulable
   ***************************
   ```

   and a plot of the resulting schedule.
2. Now its time to implement your own scheduling algorithm! create a file called `edf.m` (use `rm.m` as a base for the new file). The function prototype on the fist line of the new file should be:

   ```
   function TS = edf(T)
   ```

   redefine the `priority` subfunction in `edf.m` such that it computes task priorities according to the Earliest Deadline First dynamic priority ordering.
3. In `lab1.m`, remove the % character before lines 13 and 25. Now, the test bench will run your EDF scheduling algorithm as well. Run the test bench again. For the taskset given in `lab1.m`, the EDF algorithm should generate the same schedule as the RM algorithm.

4. Now, increase the WCET of `task3` from 10 to 18 time units and run the simulation again. The new taskset should be schedulable using EDF, but not with RM. If your algorithm is correct, the resulting schedule should be the one depicted in Figure 4.

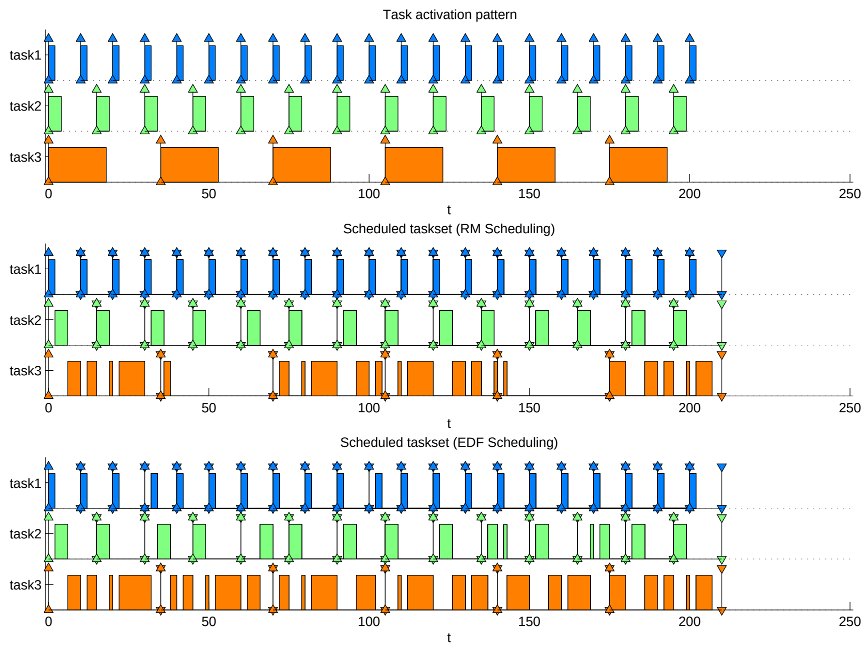

Figure 4: Taskset with $U = 0.9810$, scheduled using the RM and EDF scheduling algorithms

Submission

Send the files `utilization.m` and `edf.m` to `andreas.persson@ide.hh.se` or leave printed copies in the box outside room E527.