Real-time Concepts for Embedded Systems
WT 18/19

Assignment 2

Submission Deadline: Sunday, 18.11.2018, 23:55

- *Submit the solution in PDF via Ilias (only one solution per team).*
- *Respect the submission guidelines (see Ilias).*

---

1 Frame-based Scheduling [23 points]

In this question, frame-based scheduling on uniprocessors will be adressed. Frame-based scheduling belongs to the family of of cyclic scheduling. Throughout this question, task i of a task set $\mathbb{T}$ is characterized by the following 2-tuple (p_i, e_i), i.e. we assume all tasks are immediately released ($r_i = 0$), and tasks have to be completed within their period ($d_i = p_i$). In this question, we do not consider splitting tasks.

a) [3 points] Suppose $\mathbb{T}$ consists of three periodic tasks, with CPU utilization less than 1. Suppose that there exists an appropriate frame size that satisfies the three constraints for the frame size. Does this ensure that we can feasibly schedule the three tasks using the frame-based scheduler? If yes, prove it, else provide a counter example.

b) You are given the task set $\mathbb{T}_A$ with 10 tasks in Table 1.

Table 1: Task set $\mathbb{T}_A$.

Task i	p_i	e_i
0	20	3
1	24	2
2	30	3
3	32	3
4	28	1
5	27	1
6	35	4
7	21	1
8	42	5
9	36	4

i. [1 point] What is the hyper-period of task set $\mathbb{T}_A$?

ii. [3 points] Through only considering the first frame size constraint given in the lecture ($H \mod F = 0$), how many possible frame sizes F satisfy the first constraint for task set $\mathbb{T}_A$?

iii. [1 point] Through only considering the second frame size constraint given in the lecture ($F \geq \max_{T_i} e_i$), what is the smallest frame size that satisfies *only* the second constraint for task set $\mathbb{T}_A$?

iv. [2 points] Through only considering the third frame size constraint given in the lecture ($2F - \gcd(p_i, F) \leq d_i$), what is the largest frame size that satisfies *only* the third constraint for task set $\mathbb{T}_A$?

v. [1 point] Assuming a frame size of 20, how many frames will the frame list contain for task set $\mathbb{T}_A$?

c) [7 points] Consider task set $\mathbb{T}_B$ (cf. Table 2) with hyper-period of 600. The tasks shall be scheduled in the order of increasing index i. Fill-in the starting times of all executions of all tasks of task set $\mathbb{T}_B$ in Table 2 within the hyper-period.

Table 2: Task set $\mathbb{T}_B$.

Task i	p_i	e_i	1st exec.	2nd exec.	3rd exec.
0	200	1	0	200	
1	200	2	1		
2	300	3	3		
3	300	4			
4	200	5			
5	600	6			

d) [5 points] Complete the frame list given in Table 3 by inserting the tasks from task set $\mathbb{T}_B$ according to their start times (cf. Table 2) into the right frame in the right order. The frame size is set to 120.

Table 3: Frame list $\mathbb{T}_B$.

Frame #	frame start	frame end	tasks
0	0	120	
1	120	240	
2	240	360	
3	360	480	
4	480	600	

2 Practical Task: FreeRTOS implementation [13 points]

In this part, you shall emulate the behaviour of a frame-based scheduler in FreeRTOS for the frame list from the previous question (cf. Table 3). For this question, consider only the order of tasks in the frame and not their release time relative to the start of the frame. In other words, tasks are executed (in the order from Table 3) right after the start of the frame, ignoring the calculated start times from Table 2.

a) [13 points] Use the project from the provided .zip-file. As in the previous exercise, this exercise only requires changes in the `main_exercise.c`-file (cf. Figure 1), which gets called from the `main()`-function in the file `main.c`. To emulate the frame-based

```
exercise_01
└─FreeRTOS
  ├─FBS-App
  │ └─WIN32-<Compiler>
  │   ├─main_exercise.c
  │   ├─main.c
  │   └─
  └─Source
    └─
```

Figure 1: Folder structure for FBS-Application.

scheduling algorithm without having to change the FreeRTOS kernel, we will use the following approach in this question: There is one *Scheduler*-Task and multiple *Worker*-Tasks, which correspond to the tasks from task set $\mathbb{T}_B$. *Worker*-Tasks do their actual computation (in this question, simply increasing a task-local counter), before suspending themselves. Each time the *Scheduler*-Task is executed, it checks that all tasks from the previous frame are suspended (i.e. no overruns) and loops over the frame list indefinitely to resume only those tasks, which are allowed to run in the next frame. Then the *Scheduler*-Task waits until the next frame is to be scheduled. For this to work, the *Scheduler*-Task has to run at higher priority than the *Worker*-Tasks, to be able to interrupt possible overrunning *Worker*-Tasks. In this question, Task 5 shall misbehave, i.e. Task 5 ends up in an infinite loop, and therefore will not suspend itself.

To implement the FBS-App, extend the `main_exercise.c`-file considering the requirements below:

- Implement one “well-behaved”, self-suspending *Worker*-Task function, with the computational task of counting up to $e_i \cdot 1000000$, (i.e. Task 0 counts to 1E6).
- Implement the “misbehaving” *Worker*-Task function, which indefinitely changes the counter value.
- *Worker*-Tasks print some identifying string (i.e. the value they have to count up to) each time they finish their computation.
- Implement and initialize the data structure for the frame list in the `main_exercise()`-function. Your data structure should be able to support varying number of frames and varying number of tasks per frame.

- Implement the *Scheduler*-Task. The *Scheduler*-Task runs every 120 milliseconds (in between the frames). It first checks for overruns in the previous frame, deleting every detected overrun task[1], then resumes the respective *Worker*-Tasks for the next frame, and finally "waits" (in Blocked-State) until the next frame starts. Take care to handle the very *first* frame, *empty frames* and *deleted tasks* correctly. The *Scheduler*-Task prints a string identifying overrun tasks, as well as a string containing the frame number, each time it starts to schedule the tasks for the next frame.
- Create all the *Worker*-Task instances (regular as well as misbehaved) and the *Scheduler*-Task instance. Take care to pass the necessary parameters and to create the *Scheduler*-Task instance with higher priority than the *Worker*-Task instances.
- Start the scheduler, so that the system starts.

A (possible) example output is given below.

```
It's the very first frame.
Scheduling for frame 0.
I counted 1000000 cycles, took ~ 121 ticks.
(...)
Checking frame 0.
Task 5 in frame 0 was not suspended. Sad.
Scheduling for frame 1.
(...)
```

[1]In this exercise, we consider each task not suspended at the expiration of the frame it was scheduled in, as an overrun task.