Real-time Concepts for Embedded Systems
WS 2023/24

Assignment 2

Submission Deadline: 01.12.2023 23:55

- *Submit the solution in PDF via Ilias (only one solution per group).*
- *Respect the submission guidelines (see Ilias).*
- *This assignment has a total of 45 points.*

---

1 Network Flow Graph [18 points]

a) [5 points] Figure 1 shows a set of jobs with their precedence constraints. The values in the parenthesis above each job are the given release times and deadlines. Fill in the empty boxes with the effective release times and deadlines for each job. Is it possible to create a schedule for this set of jobs?

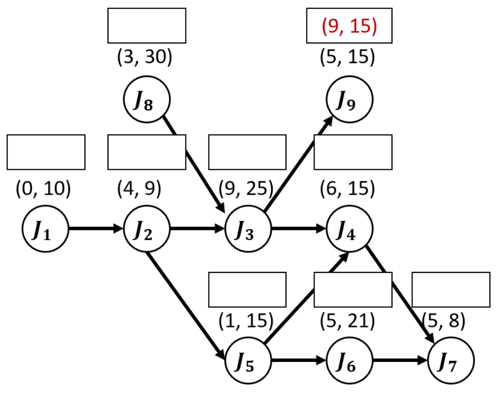


Figure 1: Effective Release Time and Deadline

b) In this exercise we want to create a network-flow graph in order to find a pre-emptive cyclic schedule for frame-based scheduling. Given are the periodic tasks $T_A = (4, 15)$, $T_B = (5, 20)$, $T_C = (8, 30)$ with $T_i(e_i, p_i)$. The deadline d_i equals the period p_i.

   i. [1 point] First, calculate the hyper period for the given tasks.

   ii. [2 points] Now calculate the maximum frame size for the given tasks. Justify your answer by showing your calculations.

   iii. [8 points] Now draw a network-flow graph for the given tasks and annotate the edges with their assigned flows and capacities. Use a frame size of 10.

   iv. [2 points] Finally, use the concept of max-flow discussed in the lecture and check whether the task set is schedulable or not. If the task set is schedulable show a schedule by drawing a timing diagram and include the tasks' periods/deadlines, otherwise justify why it is not schedulable.

2 Aperiodic and sporadic tasks [8 points]

In this question, we deal with scheduling of aperiodic and sporadic tasks.

a) [5 points] Consider the given frame-based schedule depicted in Figure 4. Each frame is 6 time units long, and the schedule consists of 8 frames in total. The figure shows the schedule that spans for 48 time units. The schedule is split into two rows with the first row starting at time 0 and the second row starting at time 24. Given in Table 1 are preemptable sporadic tasks that are to be scheduled to the existing schedule. S 1 has the highest priority from the sporadic tasks, S 5 the lowest. Decide for every sporadic task if it can be scheduled. Give the schedule including the schedulable sporadic tasks.

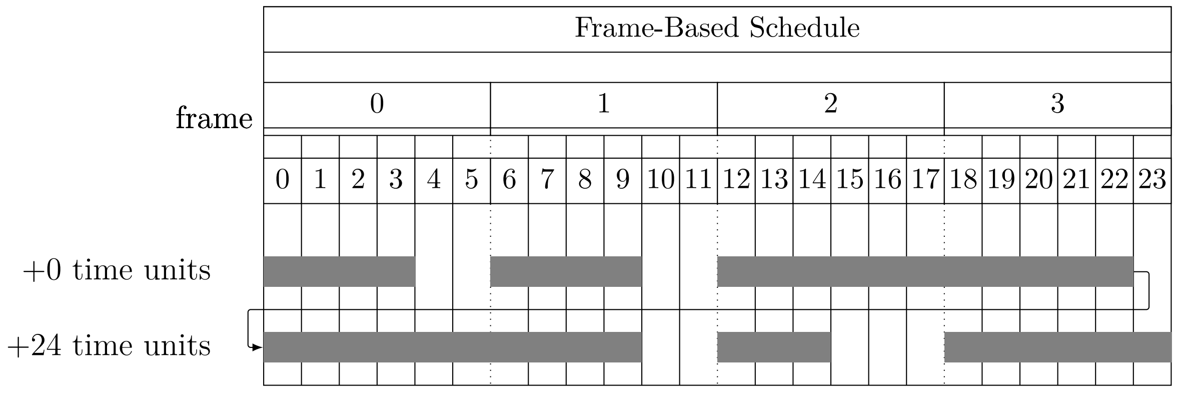


Figure 4: Frame-based schedule with periodic tasks depicted in grey.

Table 1: Sporadic Task set

Sporadic Task i	Release time r_i	deadline d_i	execution time e_i
S 1	4	7	2
S 2	5	27	3
S 3	6	39	3
S 4	10	35	2
S 5	20	36	2

b) [3 points] Consider a scenario with periodic tasks and aperiodic tasks. The hyper-period for the period tasks is 45 time units, and the sum of execution times of all periodic tasks is 24. We have the following information given in Table 2 about the aperiodic tasks, which occur in the system. What is the average response time W of any aperiodic task in this scenario? Explain your calculation.

Table 2: Aperiodic Task set

Task i	E[β_i]	E[β_i^2]	λ_i
Task 1	3.125	12.375	$\frac{1}{43}$
Task 2	1.25	2.0	$\frac{1}{12}$
Task 3	4.125	22.875	$\frac{1}{34}$
Task 4	2.375	7.625	$\frac{1}{45}$

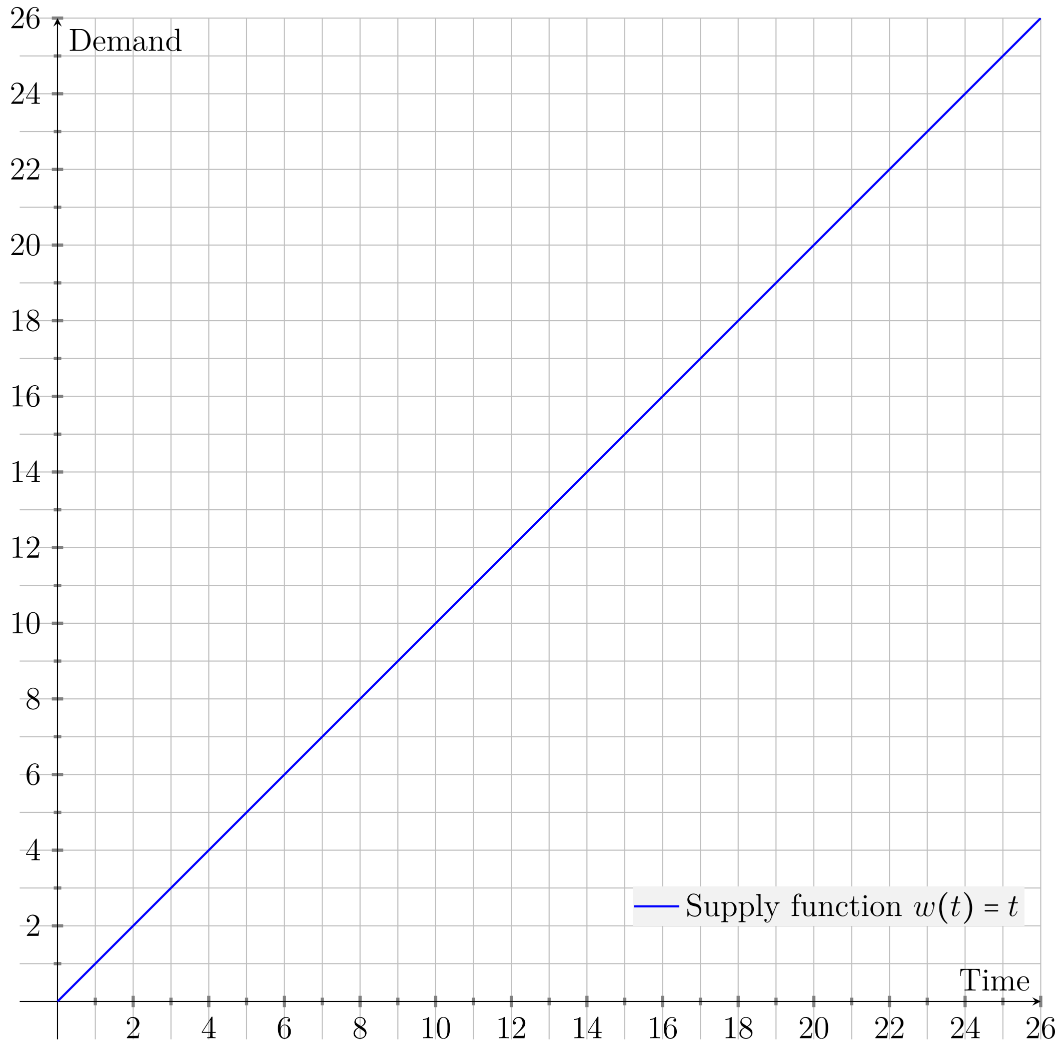
3 Graphical Time-Demand Analysis [9 points]

In this question, static priority-driven scheduling with rate-monotonic priority assignment (RMA) is considered. Thereby, we use the graphical time-demand analysis to check if a task set is schedulable. The priorities are the inverse of the task period, i.e., $prio_i = \frac{1}{p_i}$. Tasks with a higher numerical value of $prio_i$ have higher priority.

a) [6 points] Draw the time-demand function for every task in task set $\mathbb{T}_1$ (cf. Table 3). Consider a uni-processor system. *Assume zero phasing for all tasks.*

Table 3: Task Set $\mathbb{T}_1$

Task	**Computation Time**	**Period/Deadline**
A	2	5
B	3	7
C	1	17



b) [3 points] Now justify for every task whether it is schedulable or not using the graph.

4 Completion-Time Test (Worst-Case Simulation) [10 points]

a) [3 points] A task set $\mathbb{T}_a$ with corresponding computation times and periods of the tasks is given in Table 4. Prove with the **utilization bound test** whether the task set can be scheduled on a uni-processor system using the Rate Monotonic Scheduling algorithm. If the task set is schedulable, also show via timing diagram the task schedule. (Note: Assume the Release times for all tasks to be at 0 time units.)

Table 4: Task Set $\mathbb{T}_a$

Task	**Computation Time**	**Period/Deadline**
A	5	20
B	8	30
C	10	40

b) [7 points] Consider the task set $\mathbb{T}_{a'}$ given in Table 5, which is a modified version from the task set before. Now check the schedulability of the new task set using the **utilization bound test**. If not satisfied, then perform a **Completion-Time Test** (Worst-Case Simulation) and show whether the changed task set is schedulable or not. Show all your calculations for the two tests.

Table 5: Task Set $\mathbb{T}_{a'}$

Task	**Computation Time**	**Period/Deadline**
A	5	20
B	8	30
C	14	40

5 [Practical Task] FreeRTOS: frame-based scheduling [0 points]

In this part, you shall emulate the behavior of a frame-based scheduler in FreeRTOS for the frame list from Table 6. Tasks are executed (in the order from Table 6) right after the start of the frame.

Frame #	frame start	frame end	tasks
0	0	120	0,1,2,3,4,5
1	120	240	
2	240	360	0,1,4
3	360	480	2,3
4	480	600	0,1,4

Table 6: Frame list

a) Use the project from the provided .zip-file. As in the previous practical exercise, this exercise only requires changes in the `main_exercise.c`-file (cf. Figure 5), which gets called from the `main()`-function in the file `main.c`. To emulate the frame-based

```
fbs
├── example
├── fbs_app
│   ├── main.c
│   ├── main_exercise.c
│   └──
├── src
├── CMakeLists.txt
└──
```

Figure 5: 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 $\mathbb{T}$. *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 ms (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.