Real-time Concepts for Embedded Systems
WT 18/19

Assignment 4

Submission Deadline: Sunday, 06.01.2019, 23:55

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

1 Dynamic Priority Scheduling [14 points]

a) [4 points] Consider a task set $\mathbb{T}_1$ that consists of three periodic and preemptable tasks A, B, and C. Their corresponding execution times and periods are given in Table 1. Use the time supply line t given in Figure 1 to check the schedulability of the task set $\mathbb{T}_1$ under rate monotonic assignment on a uniprocessor system by drawing the time-demand function.
Note: Assume all the tasks have zero phasing at $t = 0$

Table 1: Task Set $\mathbb{T}_1$

Task	Execution Time	Period/Deadline
A	1	3
B	1	4
C	2.25	6

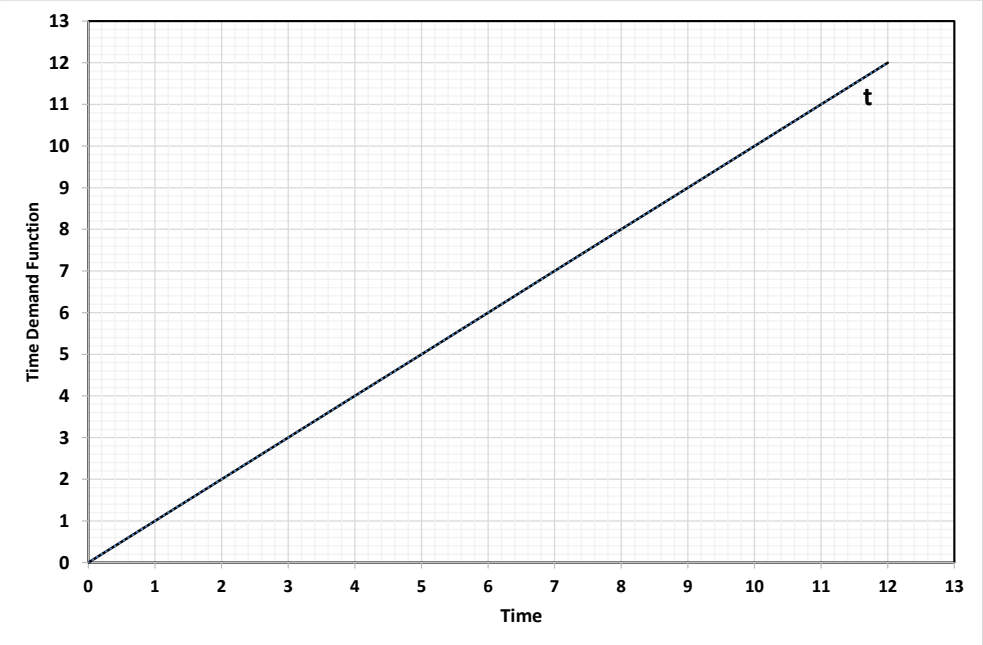

Figure 1: Timing Diagram for Time Demand Function

b) [6 points] Schedule the same task set given in part a) using the two dynamic scheduling algorithms Earliest Deadline First (EDF) and Least Slack Time First (LST) by completing the timing diagram given below in Figure 3 and Figure 4.

While making a scheduling decision if two tasks have the same deadline in case of EDF or if two tasks have equal slack time in case of LST then the two tasks are scheduled in alphabetical order (e.g. Task A followed by Task B and so on).

Also in case of Least Slack Time first, a scheduling decision based on slack time is made only in 3 cases

- scheduling decision based on slack time is done every 1 time unit
- if a task is completed
- if a new task arrives

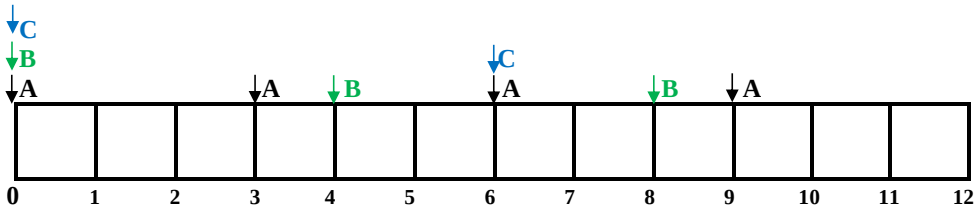

Figure 3: Solution Space for EDF Scheduling

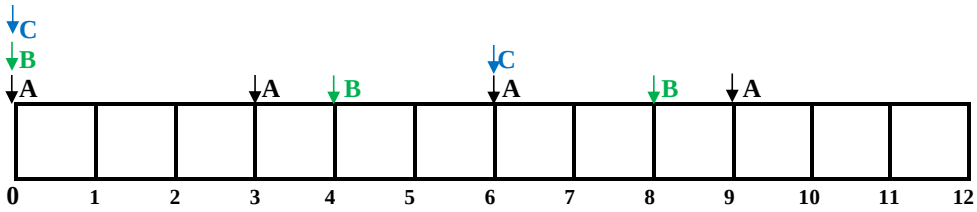

Figure 4: Solution Space for LST Scheduling

c) [4 points] Consider a new task set $\mathbb{T}_2 = \{A, B, C\}$ with three periodic tasks where the corresponding computation times and periods of the tasks are given in Table 2.

Table 2: Task Set $\mathbb{T}_1$

Task	Computation Time	Period/Deadline
A	1	3
B	1	4
C	1.5	6

In addition to these periodic tasks, there are two aperiodic tasks $X = (r = 3, e = 1.5)$ and $Y = (r = 4, e = 0.5)$ that are to be scheduled using a deferrable server $T_{DS} = (p_s = 4, e_s = 1)$. Schedule this task set $\mathbb{T}_2$ along with the aperiodic tasks

X and Y using the given deferrable server T_{DS} with EDF Scheduling by completing the timing diagram given in Figure 7.

Similar to the previous part, while making a scheduling decision if two periodic tasks have the same deadline, then the tasks are scheduled in alphabetical order. Also if the deferrable server and any of the periodic task has equal deadlines then the deferrable server is scheduled first.

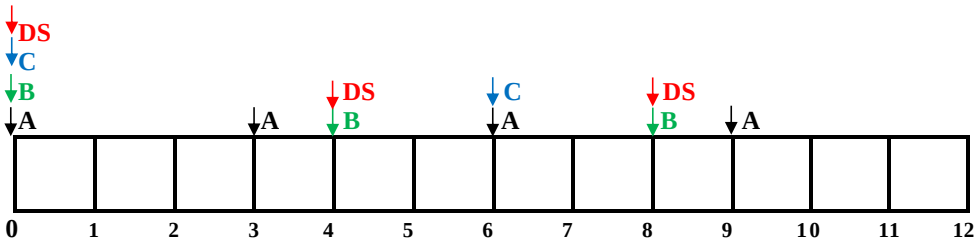

Figure 7: Solution Space for EDF and Deferrable Server Scheduling

2 Time Management [8 points]

a) [8 points] Consider a 4-layer hierarchical timing wheel as given below.

$$\circlearrowright^{i_4}_{(60,1m)},\ \circlearrowright^{i_3}_{(60,1s)},\ \circlearrowright^{i_2}_{(10,100ms)},\ \circlearrowright^{i_1}_{(10,10ms)}$$

Now, it is necessary to install three timers with timing requirements as given in Table 3.

Suppose that the *reference context* saved for each of these timers is [36, 34, 7, 3]. According to the lecture, all the three timers get installed initially in the minutes wheel in corresponding slots according to their timing requirements. State in which timing wheel and in which slot of the timing wheel are the three timers installed when the dials of the timing wheels are in each of the contexts given below. Write your answers in Table 4. Support your answer with calculations and justification.

Table 3: The timing requirements for three different timers

Timer 1	3min, 25s, 360ms
Timer 2	15min, 18s, 280ms
Timer 3	24min, 27s, 560ms

Context 1: [36, 34, 7, 3]
Context 2: [40, 0, 0, 8]
Context 3: [51, 24, 7, 0]
Context 4: [1, 0, 1, 2] in the next run

Table 4: Timing Wheels Solution Space

Context	**Timer 1**	**Timer 2**	**Timer 3**
Context 1			
Context 2			
Context 3			
Context 4			

3 Interrupts Handling [9 points]

a) [3 points] You have learned in the lecture that external interrupt sources can operate in either *level-sensitive mode* or *edge-triggered mode*. We consider edge-triggered devices for this part of the question.

Two Edge-triggered I/O Devices (A and B) share the same IRQ line (say $IRQ1$) of a PIC, as shown in Figure 9. Show using a signal space diagram a scenario where an interrupt request from one of the devices could be missed.

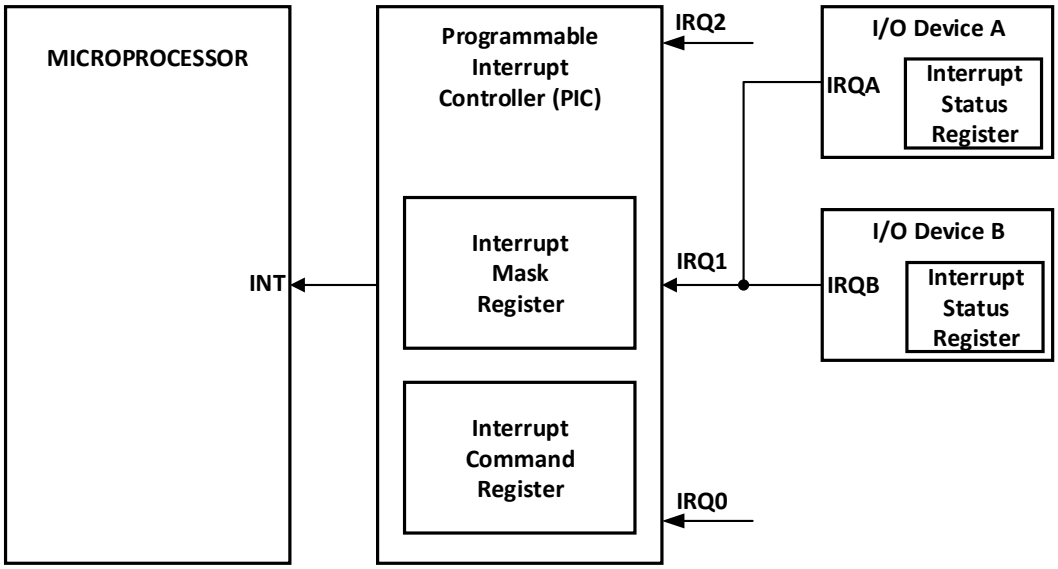

Figure 9: PIC Architecture

b) [2 points] You have learned from the lecture that, at the start and end of an ISR, interrupts are disabled during context switching. Briefly explain what happens if interrupts are **not** disabled in these cases.

c) [4 points] You have learned the execution steps of Vectored Interrupting in the lecture. Table 5 shows the steps involved in *vectored interrupting* in an arbitrary order. Indicate the correct order of the given execution steps of Vectored Interrupting in the order column in Table 5.

Table 5: Vectored Interrupt Handling

Steps	Order
A device drives an IRQ line of the PIC which is then converted to a vector number corresponding to that device and stores it in a register	
The INT line is deasserted by the PIC.	
ISR is executed and after completion of the ISR, the top frame of the interrupt stack is popped and the context of the original task is restored	
The commonly used registers are saved to the interrupt stack.	
The current program is suspended, and processor asserts INTA to PIC and PIC drives the interrupt vector number to the sytem bus	
The INT line of the processor is asserted by the PIC.	
Status Register and PCR are pushed to the stack and PCR is loaded with address of the corresponding interrupt vector	
Execution of instruction i is completed.	
While instruction i of the current program is executed, the processor samples INT and detects an assertion.	

4 (Immediate) Ceiling Priority Protocol [29 points]

In this question, the ceiling priority blocking is addressed for the task set given in Table 6. As in FreeRTOS, a higher numerical priority value indicates higher priority. The release times of the tasks are indicated by upward arrows in Figure 10 in this context. The execution time is the total number of time units, that each execution of the task, i.e. each job, will take.

Table 6: Task set with priorities, execution time (Ex. Time) and time of locking L(X) and unlocking U(X) of the semaphores A, B & C

Task	Priority	Ex. Time	L(A)	U(A)	L(B)	U(B)	L(C)	U(C)
1	5	5			1	2	3	4
2	4	7	5	6			1	3
3	3	8	3	5	2	7		
4	2	9	2	8	4	6		

The time of locking L(X) indicates after how much time units of its execution, the task will try to lock resource X, i.e. acquire a semaphore. For example, task τ_4 executes for 9 time units in total. In the first two time units (here: 0-2), τ_4 executes without locks on any resources, after that it acquires resource A (here: at t=2), executes for one more time unit (here: 2-3) before being preempted by τ_2. The time of unlocking U(X) indicates after how much time units of its execution, the task will release the lock on resource X. Be aware, that the tasks from Table 6 might have nested resource locks. For example, in the provided parts of Figure 10, consider τ_4 which acquires resource A as indicated by L(A) at t=2 and then, while holding the lock for resource A, locks resource B at t=5. The lock for resource B is then released at time t=7, before τ_4 releases resource A at t=9 indicated by U(A) in this example.

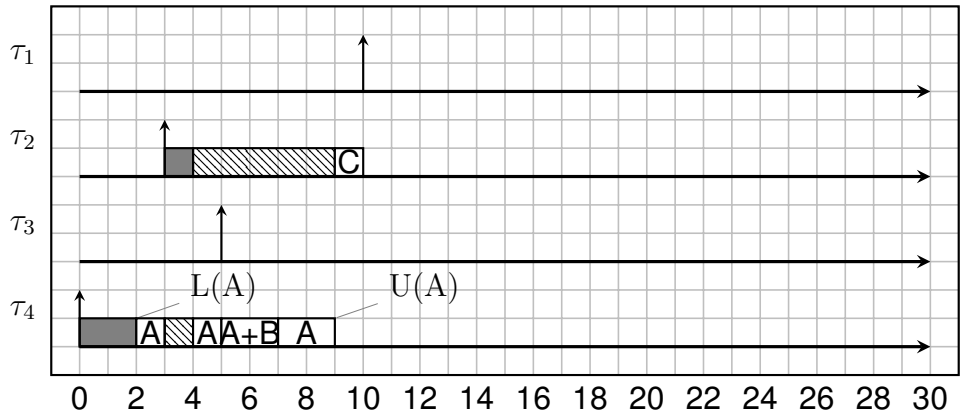

Figure 10: Incomplete timing diagram for the task set from Table 6 under PCP; time intervals where tasks are executed without locks on any resources are marked by solidly filled boxes; time intervals where tasks are ready but not executed are marked with shaded/hatched boxes; time intervals where tasks are executed while holding one or more resources are marked by boxes containing the resource identifier

Locking and unlocking times depend on the number of time units the task has been

actually executing which is influenced by blocking. For example, in the scenario depicted in Figure 10, Task τ_2 with L(C)=1 acquires resource C at t=9, because it had been only running for one time unit up to this point. Then, from t=9 on, τ_2 will execute for two more time units holding the lock for resource C (one time unit (9-10) already depicted in Figure 10). Then U(C)=3 for τ_2 is reached after τ_2 has been executed for a total of 3 time units and then the lock on C will be released.

a) [6 points] In the lecture, the priority ceiling protocol (PCP) was introduced. Give the priority ceilings for each semaphore. Then, complete the scenario in Figure 10 for the tasks in Table 6 using PCP. Mark the time intervals where a task executes without having a lock on any resource by gray boxes. Mark the time intervals where a task executes while having acquired one or more resource by giving the name of the respective resources (cf. Figure 10). Also give the priority for each of the tasks during their execution.

b) [6 points] The **immediate** ceiling priority protocol (ICPP) is a simplified version of PCP. Similar to PCP each semaphore has an assigned priority ceiling which is the highest priority of the tasks that can lock it.

   However, ICPP differs from PCP in that the priority of a task is elevated immediately to the priority ceiling when the task acquires the semaphore, *regardless of whether another task attempts a conflicting access.* Similary, the priority is restored to the previous priority as soon as the lock on the resource is released. Therefore, ICPP is easier to implement compared to PCP.

   As in the previous question part, draw a timing diagram for the scenario defined by the tasks in Table 6 but now using ICPP. Mark the time intervals where a task executes without having a lock on any resource by gray boxes. Mark the time intervals where a task executes while having acquired one or more resource by giving the name of the respective resources (cf. Figure 10). Also give the priority for each of the tasks during their execution.

   Note: In case a currently executing task lowers its priority to the priority of an ready task, the currently executed task is continued, but in general, this depends on the implementation of the scheduler. In case a scheduling decision has to be made between tasks with the same priority that are ready for execution, the task with lower task index is executed first.

c) In this question part, we will emulate the behaviour of ICPP in FreeRTOS by implementing a function wrapper around FreeRTOS provided semaphore methods. For FreeRTOS semaphores, what has been introduced in the lecture as wait()-operation for semaphores is called take()-operation in FreeRTOS. Similarly, the signal()-operation is referred to as give()-operation in FreeRTOS. The wrappers, which have to be implemented in this question part, execute the necessary priority adjustments in addition to calling the semaphore functions.

   We will again use the task set from Table 6. To store references to semaphores and the associated priority values, you may create a data structure, where you can refer to the semaphores with a numerical index for convenience (e.g. an array of structs, where the array element with index 0 maps to semaphore A from Table 6, element with index 1 maps to semaphore B, and so on).

   As usual, the provided zip-file contains a FreeRTOS project. The `main_exercise.c`-

file already contains a function `vUselessLoad(uint32_t ulTimeUnits)` which keeps the processor busy for `ulTimeUnits` time units.

i. [5 points] Create a function `usPrioritySemaphoreWait()` which emulates the behaviour of the ICPP when trying to acquire a resource (here: semaphore). That means the priority of the calling task (i.e. the task that wants to acquire the resource) has to be raised to the priority ceiling of the respective semaphore and the take-function of the semaphore is called. When the calling task acquired the resource print the following string: `Task <x> acquired resource <y> and changed its priority from <i> to <j>.`
Hint: Think about the order of these operations and whether it is required to track the task's priorities.

ii. [5 points] Create a function `usPrioritySemaphoreSignal()` which emulates the behaviour of ICPP when releasing a semaphore. That means the priority of the calling task (i.e. the task that wants to release the resource) has to be restored and the give-function of the semaphore is called. When the calling task released the resource, print a string with similar to information as requested in previous question part.
Hint: Think about the order of these operations and when to call ***`printf()`****.*

iii. [4 points] Create four Task-functions which behave similar to the tasks given in Table 6, where the `vUselessLoad()` is used to mimic the timing behaviour, e.g. for τ_1:

```
vUselessLoad(1); //emulate Task doing something for 1 time unit
usPrioritySemaphoreWait(...);
vUselessLoad(1);
usPrioritySemaphoreSignal(...);
vUselessLoad(1);
usPrioritySemaphoreWait(...);
vUselessLoad(1);
usPrioritySemaphoreSignal(...);
vUselessLoad(1);
```

Tasks should execute periodically with a sufficiently long period of 10 seconds and tasks should be ready relative to the start of each period according to the release times given in Figure 10. Create and initialize all other necessary data structures for this scenario (e.g. semaphores, prioritiy ceiling, and so on). Complete the `main_exercise()`-Function in `main_exercise.c` such that you can run this ICPP-scenario
Note: Due to the timing behaviour of the FreeRTOS simulator, the actual execution of the task instance may be different from your results in the question part b).

d) [3 points] Assume future versions of FreeRTOS would provide an ICPP-compatible semaphore, where there exists *one* system call that does the semaphore operations and priority adjustments. Is there a scenario where the wrapper functions from question part c) behave differently than these system calls. Justify your answer.
Hint: A system call can be implemented atomically with regards to task arrivals.