

Real-time Concepts for Embedded Systems

WT 18/19

Assignment 1

Submission Deadline: Sunday, 04.11.2018, 23:55

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

1 Timing Parameters and Resource Constraints [8 points]

- a) [3 points] Two task sets with their corresponding release time, computation time and period are given in Table 1 and 2. The period also represents the relative deadline of that task. Use timing diagrams (cf. Figure 1 and 2) to show whether the two task sets can be scheduled in a uniprocessor system without any of the tasks missing their deadlines. Consider a non-preemptive priority-based system where A has the highest priority and C has the lowest priority. (Hint: You can verify your answer by using the formula for processor utilization $U = \sum_i \frac{C_i}{P_i}$; where C_i and P_i are the computation times and period of the i^{th} task. If $U \leq 1$ then it is schedulable else it is not.)

Table 1: Task Set 1

Task	Release Time	Computation Time	Period/Deadline
A	0	2	6
B	3	2	5
C	5	1	3

Table 2: Task Set 2

Task	Release Time	Computation Time	Period/Deadline
A	0	2	6
B	3	2	5
C	5	1	4

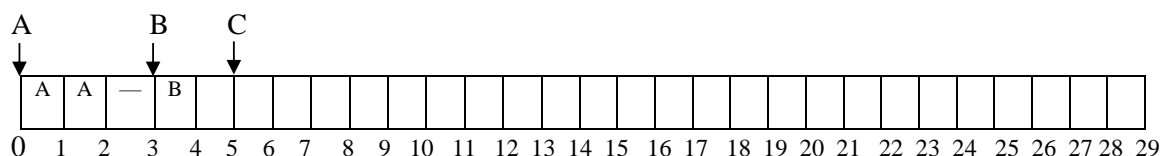


Figure 1: Solution Space for 1a

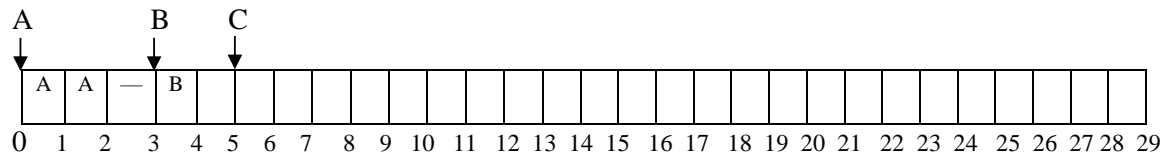


Figure 2: Solution Space for 1b

- b) [2 points] In a task set consisting of n tasks ($n > 1$) what happens if for one task the computation time is equal to its period (For example $P_1 = C_1$).
- c) [3 points] Two tasks P_1 and P_2 write to a shared variable a that is initialized to 0 ($a = 0$). The pseudo-code of the two tasks is shown in Figure 3. Give all possible values for the shared variable a after the execution of P_1 and P_2 assuming preemption is allowed and there is no mutual exclusion. Also assume that a task gets preempted only after completing the execution of the current instruction.

Task P_1	Task P_2
1: $x = 5$	1: $y = 6$
2: $a = x$	2: $a = a + y$
3: $a = a * 3$	

Figure 3: Pseudo-code of Tasks P_1 and P_2

2 Worst-Case Execution Times

[10 points]

The worst-case execution time (WCET) is a guaranteed upper bound on the time between the start of the execution of a task, and the time when the execution of the task is completed. The WCET shall not be exceeded for any possible input data and execution scenario. Additionally, the WCET should be a tight bound, ie. the least possible upper bound.

- a) [3 points] Assume, that the tasks are written in language where the maximum execution times of basic language constructs (loops, conditional statements, etc.) are known and context independent. To make the problem of determining the WCET a tractable problem, some prerequisites¹ must be met:

- no unbounded control statements at the beginning of a loop,
- no arbitrary recursive function calls,
- no arbitrary dynamic data structure

Explain briefly, why these prerequisites are necessary.

foo(a,b)	
1:	$s = -1$
2:	$i = 0$
3:	if $b > 0$ then
4:	if $a > 0$ then
5:	while $i < 5$ do
6:	if $i < b$ then
7:	$s = 0.5 \cdot \left(s + \frac{a}{s}\right)$
8:	end if
9:	$i = i + 1$
10:	end while
11:	else if $a == 0$ then
12:	$s = 0$
13:	end if
14:	end if
15:	store(s)

Figure 4: Task with two parameter a and b.

- b) You are given the pseudo-code in Figure 4. On a system, the maximum execution times (MET) of basic language constructs are as follows:

assignment : 1 time unit

if-statement(test and branch) : 2 time units

while-statement(test and branch) : 2 time units

addition : 2 time units

division : 4 time units

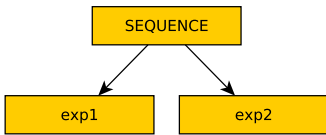
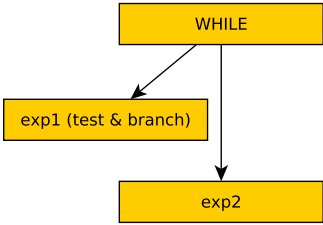
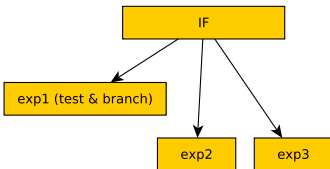
multiplication : 3 time units

¹H. Kopetz, Real-Time Systems: Design Principles for Distributed Embedded Applications

store() : 3 time units

- i. [4 points] For the WCET computation, we will use a tree-based approach. In Table 3, for each language construct, the associated syntax tree is given. In the last row, the rule for computing the WCET is stated, assuming that $T(\text{exp1})$ is the maximum time it takes to execute expression exp1 . Draw the full syntax tree for the pseudo-code given in Figure 4.

Table 3: Rules for syntax-tree generation and WCET computation of root node

exp1 exp2	while exp1 do exp2 end while	if exp1 then exp2 else exp3 end if
		
$T(\text{exp1}) + T(\text{exp2})$	$T(\text{exp1}) + T(\text{exp2}) \cdot \text{iter}_{\max}$	$T(\text{exp1}) + \max(T(\text{exp2}), T(\text{exp3}))$

- ii. [3 points] Compute the WCET using the syntax tree by applying the rules from Table 3. Start from the leave nodes and annotate each node in the syntax tree with the maximum execution time up to that node. For each node, give the overall maximum execution time and show how you calculated this value with respect to the maximum execution times of the child nodes (cf. Figure 5)

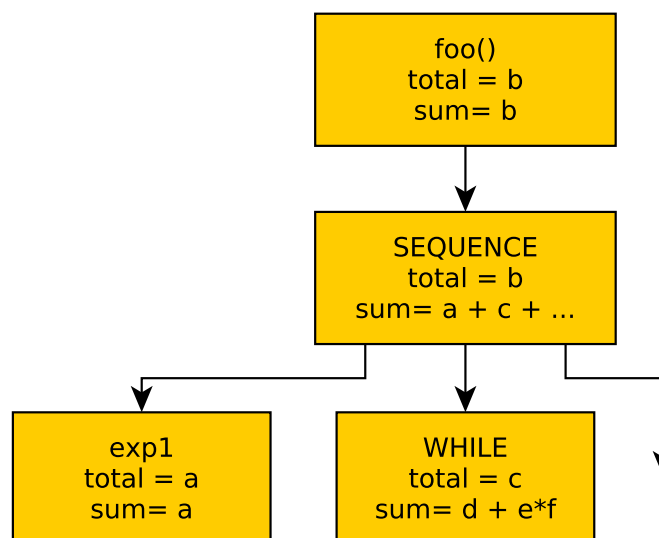


Figure 5: Example illustration of an annotated tree.

3 Scheduling Anomalies

[9 points]

- a) [3 points] Consider a multiprocessor system with two processors (P_1 and P_2). Given in Table 4, a task set that contains 6 tasks (A to F) with their corresponding computation times. Suppose a non-preemptive priority based system with priorities decreasing alphabetically (Task A has the highest and F has the lowest priorities).

Table 4: Task Set 1

Task	Computation Time	Task	Computation Time
A	3	D	4
B	6	E	4
C	4	F	4

Given below are the precedence constraints of the Task set and Figure 6 shows the precedence relations in the form of a directed acyclic graph.

$$A \prec B, D \prec E, B \prec F, D \prec F \quad (1)$$

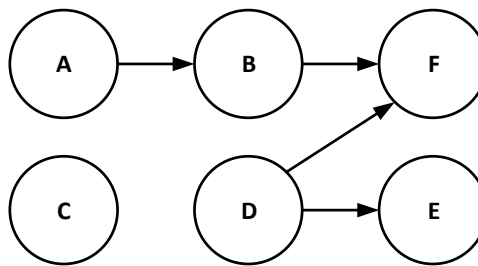


Figure 6: Precedence relations

With the given parameters, complete the timing diagram given below in Figure 7 that illustrates which task is allocated the processor at each time unit. If no task can be scheduled at a time unit then indicate it by X. Also state the global completion time for the given task set.

CPU Time	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Processors															
P1	A	A	A												
P2															

Figure 7: Timing Diagram - Solution Space for 3a

CPU Time \ Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P1	A	A	A												
P2															
P3															

Figure 8: Timing Diagram - Solution Space for 3b

- b) [3 points] In this part, the number of processors is increased from **two** to **three**. Complete the timing diagram given below in Figure 8 and state in this case what happens to the global completion time.
- c) [3 points] In this part, consider the same system as in the first part, except that the precedence relation between task *A* and *B* is relaxed as shown in Figure 9. Complete the timing diagram given below in Figure 10 and state in this case what happens to the global completion time.

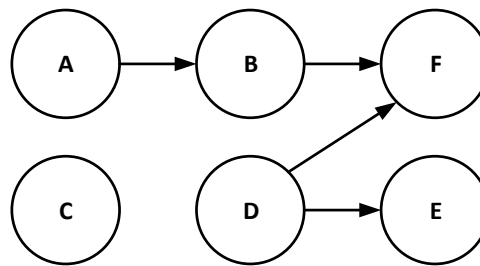


Figure 9: New Precedence Relations

CPU Time \ Processors	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P1	A	A	A												
P2															

Figure 10: Timing Diagram - Solution Space for 3c

4 Hands-on Task FreeRTOS: Getting Started [9 points]

In this part, you are required to create simple task instances in a simulated real time environment with the help of FreeRTOS. You are required to use C Programming for your implementation.

Before getting started, you should ensure to configure the initial setup. The following steps might be helpful.

- Install Microsoft Visual Studio. You can download the latest version for free using your student account from this link:
<http://e5.onthehub.com/d.ashx?s=h5783esqbf>
- Download the FreeRTOS package from this link: <https://www.freertos.org/>. Alternatively you can obtain the source files directly from SVN from this link:
<https://sourceforge.net/p/freertos/code/HEAD/tree/>.

To validate your setup you should be able to run the existing demo project out-of-the-box.

If you use Visual studio you can load the demo project via the `WIN32.sln` file (File → Open → Project/Solution) which is located in

`FreeRTOSv10.1.1/FreeRTOS/Demo/WIN32-MSVC/` in the extracted FreeRTOS package. This opens a FreeRTOS demo project with all the necessary configurations and ready for execution. There is a `main.c` file in the project which is the entry point and a `main_blinky.c` as well as a `main_full.c` which contains the actual application code.

Find the definition of the macro `mainCREATE_SIMPLE_BLINKY_DEMO_ONLY` and set it to 1 which when run executes only a simple blink demo application. This code is implemented as a sender-receiver task where the sender task sends to the receiver continuously and the receiver task prints the message “Message received from task” every time it receives the signal from the sender. Once built and run you can see the message “Message received from task” getting printed continuously.

It is also recommended you to set the macro `mainCREATE_SIMPLE_BLINKY_DEMO_ONLY` to 0 and try to execute. This would run a comprehensive test and demo application.

Having checked that your tool chain is up and running, open the project from the provided .zip-file (`chatterbox.zip`). The provided .zip-file contains the folder structure given in Figure 11, and should be familiar from the FreeRTOS distribution. Instead of the demo code, the provided project contains a Chatterbox-App.

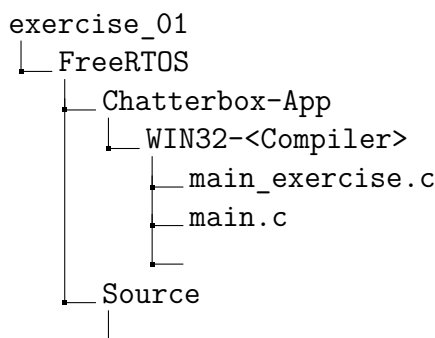


Figure 11: Folder structure for Chatterbox-Application.

The `main()`-function in the file `main.c` is the entry-point to the application. In this exercise, this `main()`-function initializes the heap and calls the `main_exercise()`-function from the file `main_exercise.c`. Your task is to implement the missing parts of the “chatterbox”-application in the provided `main_exercise.c`-file. The “chatterbox”-application basically consists of 3 instances of the “chatterbox”-task. Every time the chatterbox-task is executed, it outputs a string (in this exercise the string is simply printed to the console output). The behavior of a task instance is defined at its creation by two values: a) the output string and an integer flag which can have either value 0 (infinite task instance executions) or 1 (task instance shall only be executed 5 times).

To implement the chatterbox-app, extend the `main_exercise.c`-file as follows:

- a) [3 points] Think of a suitable data structure which can be used to pass the output string and the integer flag to the C function which implements the task instances (see part b.). Initialize the necessary data structures for three task instances in the `main_exercise()`-function as follows: The output string of each task instance is `TaskX` where `X` is the number of the task instance (ranging from 1 to 3). Task instance number 1 and 2 shall be repeated indefinitely, whereas task instance with number 3 shall only be executed 5 times.
- b) [3 points] Extend the `main_exercise.c`-file with the C function which implements the behavior of the chatterbox task instances. Chatterbox task instances shall print their output string every `mainTASK_CHATTERBOX_OUTPUT_FREQUENCY_MS` ticks. Use the `vTaskDelayUntil()`-function, to block a task instance until the time it has to generate the next output. After a *Task instance number 3* has printed its output string for the final time it shall additionally print a string indicating its termination and delete its task instance.
- c) [3 points] In the `main_exercise()`-function create the three task instance using the code from the previous subquestions. Give task instance number 3 higher priority than task instance number 1 and 2. Start the scheduler, so that the three task instance are executed concurrently.