CHALMERS
UNIVERSITY OF TECHNOLOGY

SSY191 - Model Based Development for Cyber-Physical System

Individual Assignment

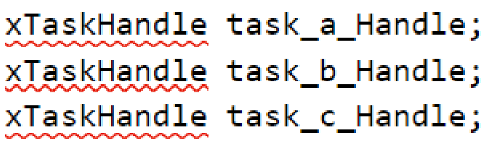
Individual Assignment - 2

Fikri Farhan Witjaksono fikrif@student.chalmers.se

May 16, 2021

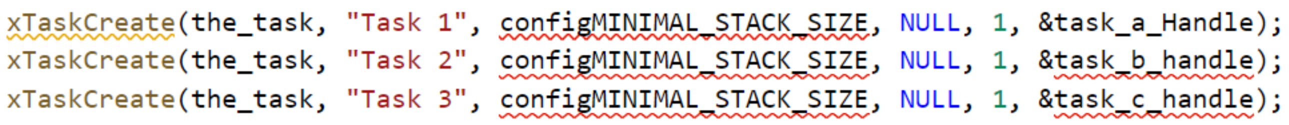
1 Problem 1a: Code and Petri-net model inspection for Deadlock Situation

In the C-code as shown in Figure 1, we could observe that the task handle was created separately for each task a, b and c. We know that each task cannot influence the priority of another task during its corresponding run. This is evident since the **pxCreatedTask** is set to **NULL** as shown in Figure 2. From there, we could also know that the size of the stack is the minimum possible as defined by FreeRTOS. From Figure 3, we could also know that the semaphores created were mutual exclusion (mutex) type which is to ensure that only one task is inside the critical section at the same time.



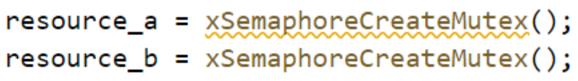
```
xTaskHandle task_a_Handle;
xTaskHandle task_b_Handle;
xTaskHandle task_c_Handle;
```

Figure 1: Task Handle Definition



```
xTaskCreate(the_task, "Task 1", configMINIMAL_STACK_SIZE, NULL, 1, &task_a_Handle);
xTaskCreate(the_task, "Task 2", configMINIMAL_STACK_SIZE, NULL, 1, &task_b_handle);
xTaskCreate(the_task, "Task 3", configMINIMAL_STACK_SIZE, NULL, 1, &task_c_handle);
```

Figure 2: Task Instances Creation



```
resource_a = xSemaphoreCreateMutex();
resource_b = xSemaphoreCreateMutex();
```

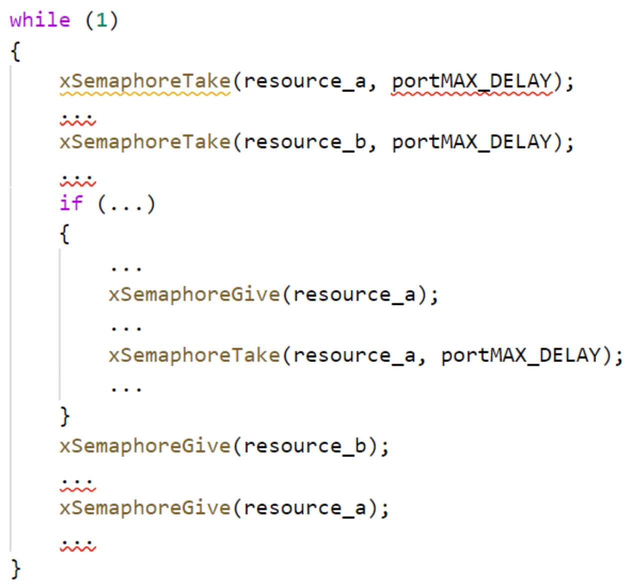
Figure 3: Mutual Exclusion (Mutex) for each resource

Looking at the original Petri-net model in Figure 4, we could see that we have 7 places nodes (i.e. $P \in \{p_0, p_1, p_2, p_3, p_4, p_5, p_6\}$), 8 transitions and 2 resources ($R \in \{r_a, r_b\}$). In the initial place denoted by p_0 marking, we have 3 different tokens. These tokens in p_0 are a form of representation for 3 corresponding processes/tasks that we have. By theorem, a transition is firing if the inputs have at least a token. Moreover, the transition is producing and consuming if a transition is getting an input token and outputting a token to the next place respectively. **Take** in the code refers to condition when a certain transition consume a token from the either resources (r_A/r_B) as well as a task token, whereas **Give** is referring to transition where token is produced back by transition to either resources (r_A/r_B) . The deadlock occurs if the following conditions are all satisfied namely :

- **Mutual exclusion**
- **Hold and Wait**
- **No Preemption**
- **Circular Wait**

However, the one deadlock scenario that is possible is when there is a token in p_1 from the second task while another token from first task is in p_4. The first condition is satisfied since we know that we are using mutex semaphores. The second condition is known from the fact that

Task 1 is requesting one resource from r_A at p_4 while that resource from r_A is currently hold by other Task 2 at p_1. Thirdly, we know that from figure 2 that we cannot preempt priorities of another task. Lastly, the circular wait happens since Task 3 is also requesting a resource from r_A which is currently held by Task 2 while Task 1 is also requesting the same resource held by Task 2 in order to be able to move the token to p_5.



```
while (1)
{
    xSemaphoreTake(resource_a, portMAX_DELAY);
    ...
    xSemaphoreTake(resource_b, portMAX_DELAY);
    ...
    if (...)
    {
        ...
        xSemaphoreGive(resource_a);
        ...
        xSemaphoreTake(resource_a, portMAX_DELAY);
        ...
    }
    xSemaphoreGive(resource_b);
    ...
    xSemaphoreGive(resource_a);
    ...
}
```

Figure 4: While loop definition for Tasks

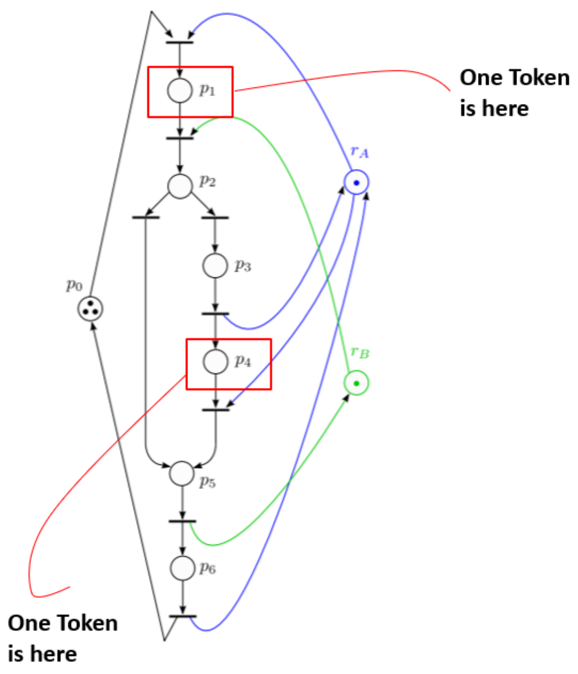


Figure 5: Original Petri-net model deadlock condition

2 Problem 1b: Deadlock Resource Allocation Graph (RAG)

The deadlock condition could be well illustrated as in Figure 6 below. In order for deadlock to occur, by theorem, a graph needs to have a cycle in itself. This is caused by the fact that for a deadlock to occur, **hold and wait** and **Circular Wait** condition must be satisfied, and this is only possible if the graph includes a cycle of waiting for resources in the processes involved. However, only having a cycle does not guarantee that we will have deadlock. Thus, it is a necessary but not sufficient condition for deadlock. However, in our case, since we are having a single instance type resource while having a cycle in the RAG, therefore we are guaranteed to have a deadlock due to reasons explained in problem 1a.

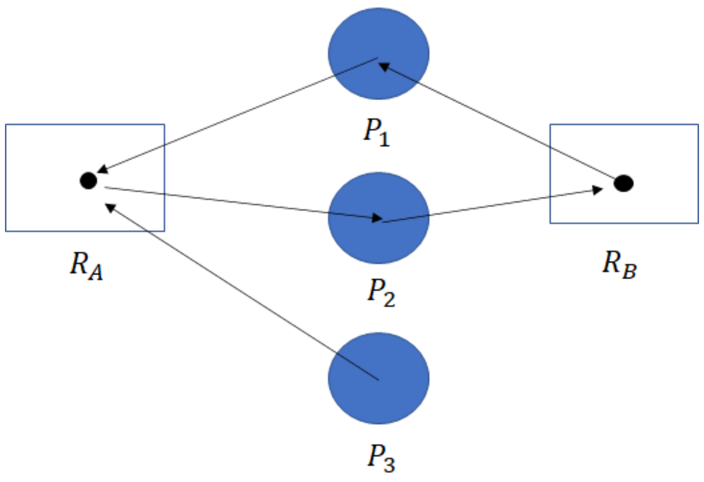


Figure 6: Resource Allocation Graph (RAG) Deadlock

3 Problem 1c: Deadlock-free Modified Code and Petri-net model

By adding an additional lock with a single instance type resource r_C, we could add **Take** command which makes the transition between place node p_0 and p_1 to consume an additional token from resource C, in addition to the transition between place node p_3 and p_4. Moreover, we could also add **Give** command which makes the transition between place node p_4 and p_5 as well as between p_1 and p_2, to produce token to the resource place node r_C. This modification will enable us to avoid the deadlock situation as explained in Problem 1b since we could avoid the **hold and wait** as well as **circular wait** when the place node p_1 is holding the Task 2 token and place node p_4 is holding the Task 1 token. The proposed additional resource is as shown in Figure 7 and Figure 8 in the next page.

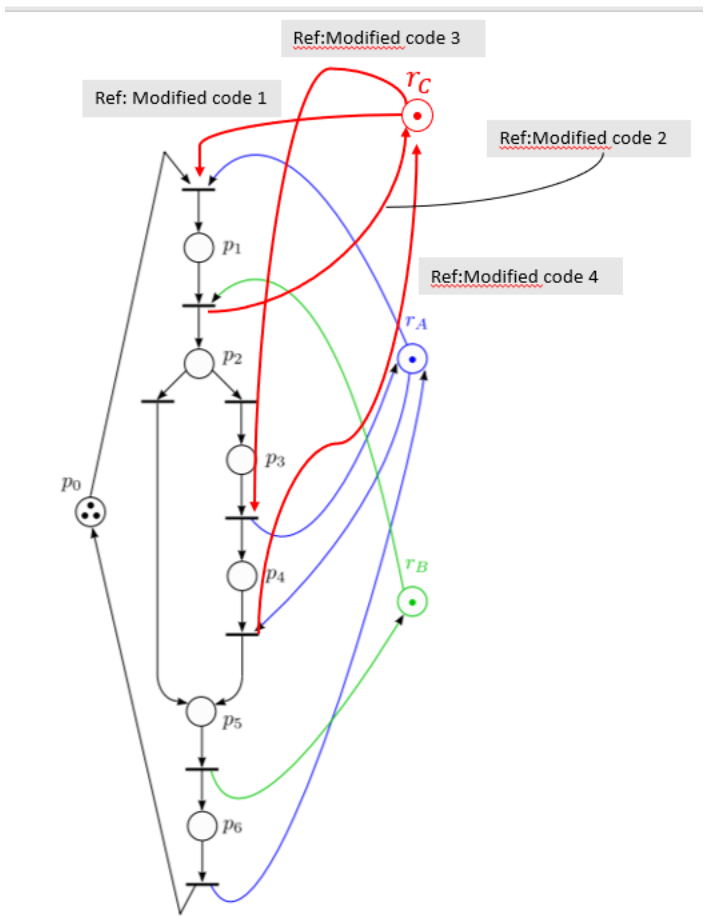
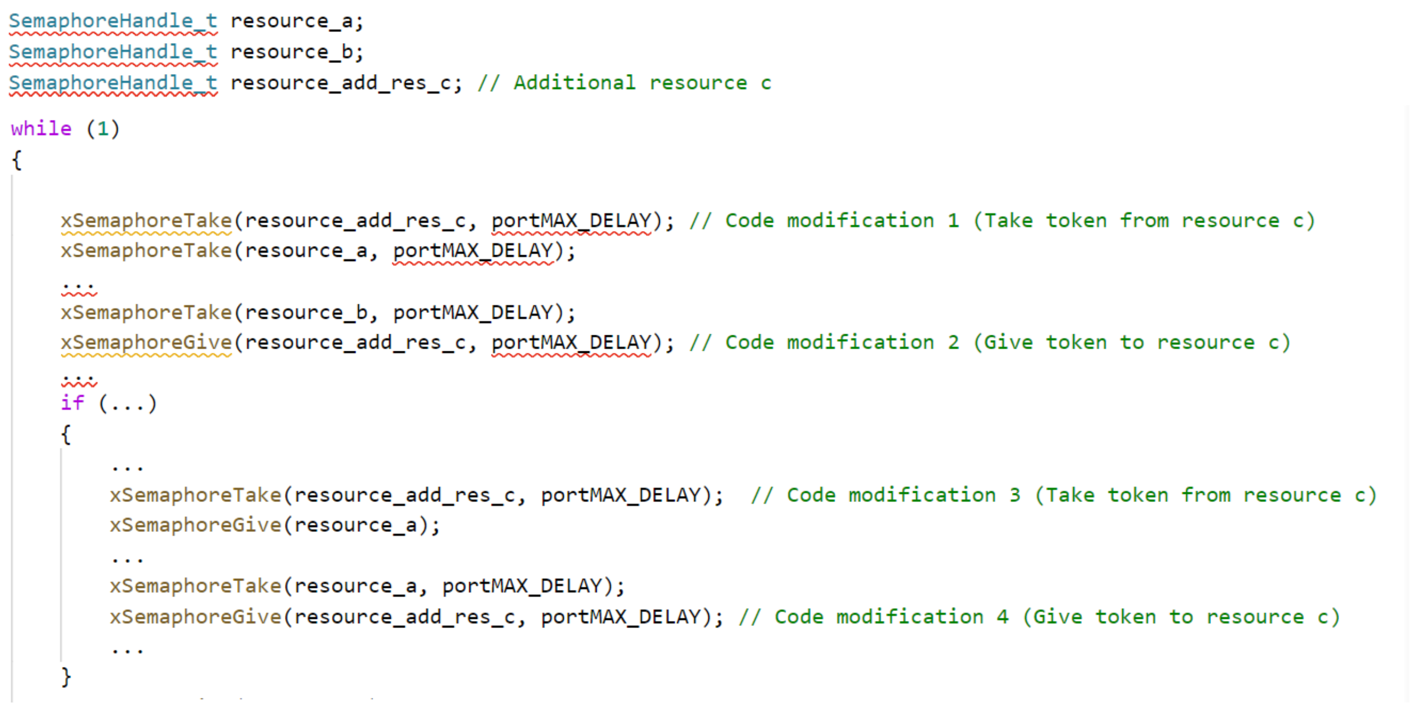


Figure 7: Deadlock-free Modified Petri-net



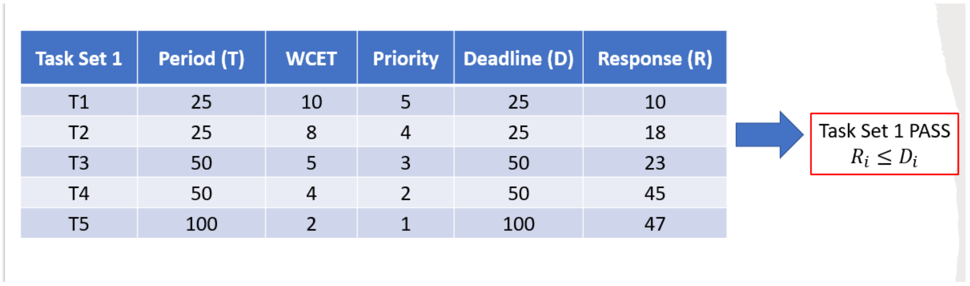
```
SemaphoreHandle_t resource_a;
SemaphoreHandle_t resource_b;
SemaphoreHandle_t resource_add_res_c; // Additional resource c

while (1)
{

    xSemaphoreTake(resource_add_res_c, portMAX_DELAY); // Code modification 1 (Take token from resource c)
    xSemaphoreTake(resource_a, portMAX_DELAY);
    ...
    xSemaphoreTake(resource_b, portMAX_DELAY);
    xSemaphoreGive(resource_add_res_c, portMAX_DELAY); // Code modification 2 (Give token to resource c)
    ...
    if (...)
    {
        ...
        xSemaphoreTake(resource_add_res_c, portMAX_DELAY);  // Code modification 3 (Take token from resource c)
        xSemaphoreGive(resource_a);
        ...
        xSemaphoreTake(resource_a, portMAX_DELAY);
        xSemaphoreGive(resource_add_res_c, portMAX_DELAY); // Code modification 4 (Give token to resource c)
        ...
    }
```

Figure 8: Modified code as reflected in Figure 7

4 Problem 2b: Schedulability and Response time analysis result

If we analyze the schedulability according to two different criteria, that is the *Liu-Layland criteria (e.g. Utilisation based Analysis)*, we will not know analytically whether each task set is schedulable or not since the criteria is only a necessary condition for schedulability, but not sufficient. Hence, satisfying the criteria will not guarantee that we will meet all the deadline (e.g. **schedulability is unknown**).

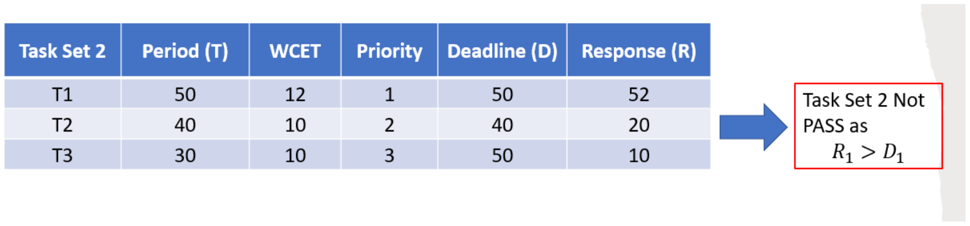
On the other hand, using *Response Time Analysis (RTA)*, we will exactly know whether we have a schedulable Task Set or not as shown in **Figure 9-13** below due to the fact that it is a necessary and sufficient condition for schedulability. It is shown that only **Task Set 1,4,5** are **passing** the test, hence **schedulable**. The rest of the task sets (e.g. **Task Set 2,3**) are **not schedulable**. The implementation of the correct analysis result to the C-Code is as shown in Figure 14. However, as shown in Figure 15, running the functions **main()** function will not give us the correct result that is illustrated by Figure 9-13 due to the fact these functions are defined as **SCHED_UNKNOWN** in its respective function definition despite the fact that we have modified **check_tests** function. Therefore, we will have to modify **schedulable_Liu_Layland** and **schedulable_response_time_analysis**, in order for **check_tests** function to be able to call the correct **check_schedulable** function.



Task Set 1	Period (T)	WCET	Priority	Deadline (D)	Response (R)
T1	25	10	5	25	10
T2	25	8	4	25	18
T3	50	5	3	50	23
T4	50	4	2	50	45
T5	100	2	1	100	47

Task Set 1 PASS $R_i \leq D_i$

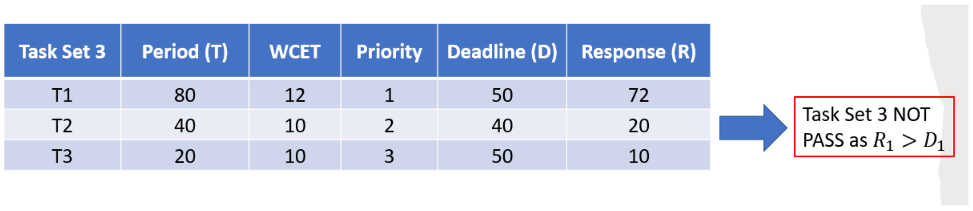
Figure 9: Task Set 1 RTA Correct Result



Task Set 2	Period (T)	WCET	Priority	Deadline (D)	Response (R)
T1	50	12	1	50	52
T2	40	10	2	40	20
T3	30	10	3	50	10

Task Set 2 Not PASS as $R_1 > D_1$

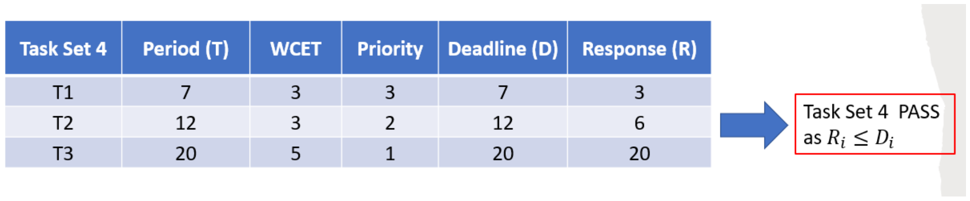
Figure 10: Task Set 2 RTA Correct Result



Task Set 3	Period (T)	WCET	Priority	Deadline (D)	Response (R)
T1	80	12	1	50	72
T2	40	10	2	40	20
T3	20	10	3	50	10

Task Set 3 NOT PASS as $R_1 > D_1$

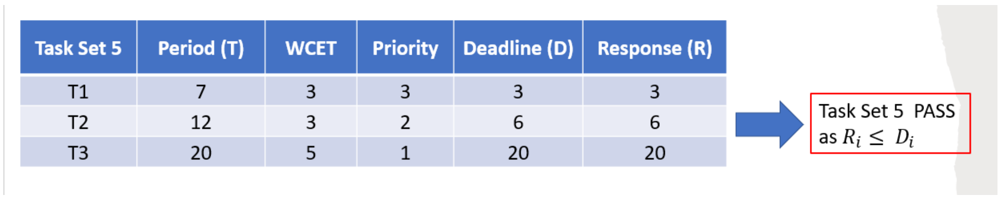
Figure 11: Task Set 3 RTA Correct Result



Task Set 4	Period (T)	WCET	Priority	Deadline (D)	Response (R)
T1	7	3	3	7	3
T2	12	3	2	12	6
T3	20	5	1	20	20

Task Set 4 PASS as $R_i \leq D_i$

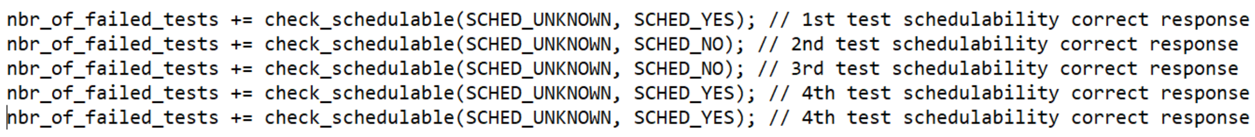
Figure 12: Task Set 4 RTA Correct Result



Task Set 5	Period (T)	WCET	Priority	Deadline (D)	Response (R)
T1	7	3	3	3	3
T2	12	3	2	6	6
T3	20	5	1	20	20

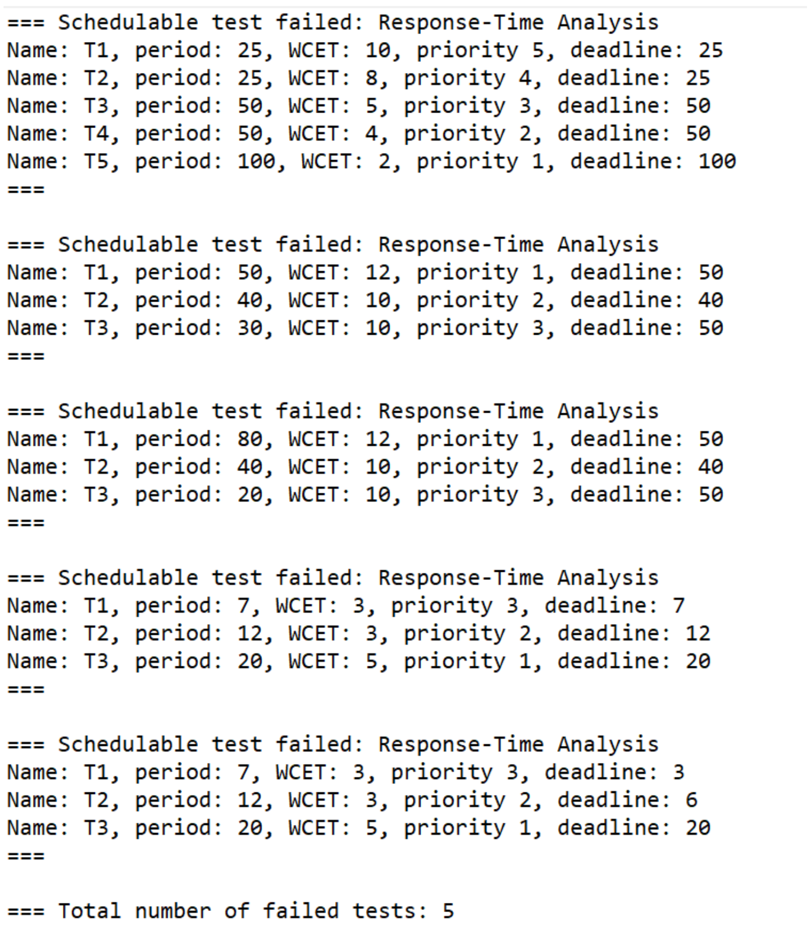
Task Set 5 PASS as $R_i \leq D_i$

Figure 13: Task Set 5 RTA Correct Result



```
nbr_of_failed_tests += check_schedulable(SCHED_UNKNOWN, SCHED_YES); // 1st test schedulability correct response
nbr_of_failed_tests += check_schedulable(SCHED_UNKNOWN, SCHED_NO); // 2nd test schedulability correct response
nbr_of_failed_tests += check_schedulable(SCHED_UNKNOWN, SCHED_NO); // 3rd test schedulability correct response
nbr_of_failed_tests += check_schedulable(SCHED_UNKNOWN, SCHED_YES); // 4th test schedulability correct response
nbr_of_failed_tests += check_schedulable(SCHED_UNKNOWN, SCHED_YES); // 4th test schedulability correct response
```

Figure 14: Correct Response Analysis Implementation



```
=== Schedulable test failed: Response-Time Analysis
Name: T1, period: 25, WCET: 10, priority 5, deadline: 25
Name: T2, period: 25, WCET: 8, priority 4, deadline: 25
Name: T3, period: 50, WCET: 5, priority 3, deadline: 50
Name: T4, period: 50, WCET: 4, priority 2, deadline: 50
Name: T5, period: 100, WCET: 2, priority 1, deadline: 100
===

=== Schedulable test failed: Response-Time Analysis
Name: T1, period: 50, WCET: 12, priority 1, deadline: 50
Name: T2, period: 40, WCET: 10, priority 2, deadline: 40
Name: T3, period: 30, WCET: 10, priority 3, deadline: 50
===

=== Schedulable test failed: Response-Time Analysis
Name: T1, period: 80, WCET: 12, priority 1, deadline: 50
Name: T2, period: 40, WCET: 10, priority 2, deadline: 40
Name: T3, period: 20, WCET: 10, priority 3, deadline: 50
===

=== Schedulable test failed: Response-Time Analysis
Name: T1, period: 7, WCET: 3, priority 3, deadline: 7
Name: T2, period: 12, WCET: 3, priority 2, deadline: 12
Name: T3, period: 20, WCET: 5, priority 1, deadline: 20
===

=== Schedulable test failed: Response-Time Analysis
Name: T1, period: 7, WCET: 3, priority 3, deadline: 3
Name: T2, period: 12, WCET: 3, priority 2, deadline: 6
Name: T3, period: 20, WCET: 5, priority 1, deadline: 20
===

=== Total number of failed tests: 5
```

Figure 15: Result of calling **main()** function