SSY191- Model Based development of cybewrphysical systems
Individual assignment 2

Harish Ravi , Chalmers ID (rharish)

1 Problem 1

(a) On inspecting the given C code and the Petri-net, it confirms that both show the same behavior. Three three tasks that is created in the main function of C code represents the three tokens at p_0 in the petri-net. The additional tokens in r_a and r_b are represented as semaphores **resources_a** and **resources_b**. The **Take** function , represents one transition in the petri net which needs one resource and one token to execute. Similarly **Give** function returns back the token to r_a and r_b. We can find a deadlock situation in the given system , which happens when first task is at p_4 and another task is at p_1, as shown in the figure below.

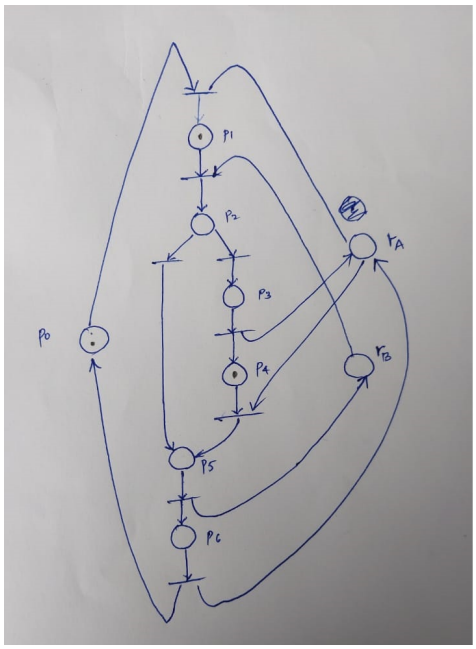

Figure 1.1: Deadlock situation

This situation is a deadlock as , Task 1 cannot proceed as it needs r_a , which is used by task2 to reach p_1 , at the same time even Task2 cannot proceed because it needs r_b, which Task 1 can only return if it can proceed! Also, Task3 cannot proceed as well , as it needs r_a , which now Task2 is using.

(b) The resource allocation graph(RAG) for the petri net or C code is given below..

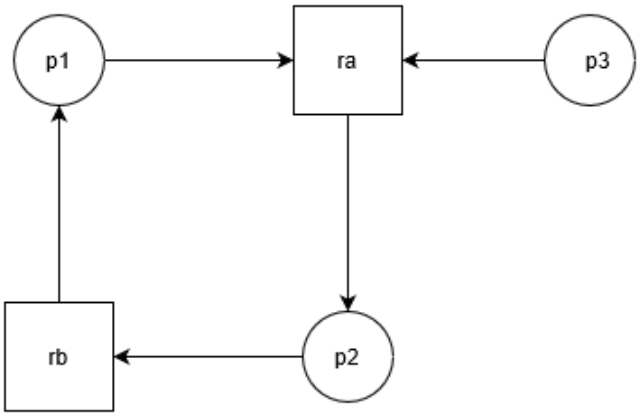

Figure 1.2: RAG

Deadlock occurs if and only if the circular wait condition is not resolvable. A circular wait condition is a scenario where a closed chain of processes exists, such that each of the process holds at least one resource needed by the next process in the chain. This along with three other conditions such as mutual exclusion , Hold and wait and No Preemption are the necessary and sufficient conditions for deadlock.
An RAG with a cycle , with only one instance per resource type then it results in a deadlock. This is because, this in turn creates a circular wait condition, which is a sufficient condition for deadlock.

(c) After identifying the deadlock situation in (a), the petrinet and C code can be modified as shown below. As discussed, we need to avoid task1 to go to p_4 state and another at p_1 at the same time. To avoid this situation, we can add a additional resource r_x, which will prevent this situation. This resource can be made to return the mutual resource that created the deadlock.

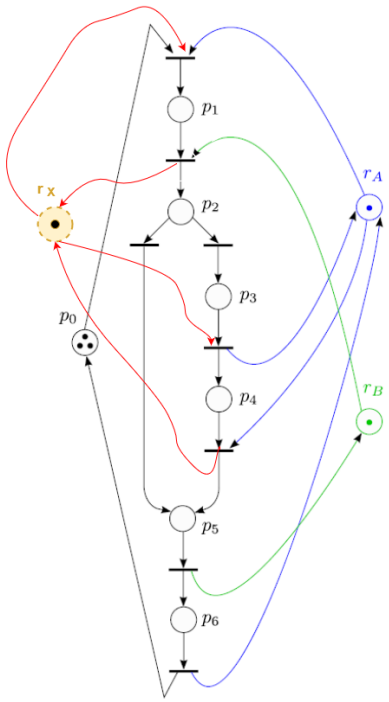

Figure 1.3: A modified PN, to avoid deadlock - addition of new resource r_x

```c
#include <stdio.h>
#include <stdlib.h>

#include "FreeRTOS.h"
#include "task.h"
#include "semphr.h"

xTaskHandle task_a_Handle;
xTaskHandle task_b_Handle;
xTaskHandle task_c_Handle;

SemaphoreHandle_t resource_a;
SemaphoreHandle_t resource_b;
SemaphoreHandle_t resource_x; // Added new semaphore to avoid deadlocks

void the_task(void *pvParameters)
{
  while (1)
  { // transition p0-p1
    xSemaphoreTake(resource_a, portMAX_DELAY);
    xSemaphoreTake(resource_x, portMAX_DELAY);
    ...
    // Transition p1-p2
    xSemaphoreTake(resource_b, portMAX_DELAY);
    xSemaphoreGive(resource_x, portMAX_DELAY);
    ...
    if (...) // two paths for p2
    {
      ...
      xSemaphoreTake(resource_x, portMAX_DELAY);
      xSemaphoreGive(resource_a);

      ...
      xSemaphoreTake(resource_a, portMAX_DELAY);
      xSemaphoreGive(resource_x);
      ...
    }
    xSemaphoreGive(resource_b);
    ...
    xSemaphoreGive(resource_a);
    ...
  }
}

int main(int argc, char **argv)
{
    resource_a = xSemaphoreCreateMutex();
    resource_b = xSemaphoreCreateMutex();
  resource_x = xSemaphoreCreateMutex(); // Added new semaphore to avoid deadlocks
    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)
    ;

    vTaskStartScheduler();
    for( ;; );
}
```

Listing 1: Modified C code to avoid Deadlocks

2 Problem 2

(a) The function **nbr_of_tasks** is implements as follows,

```c
int nbr_of_tasks()
{
    // Todo: Implement this function. You should not change anything outside this
    // function.
    task* curr_task = first_task;
    int count =0;
    while(curr_task!=NULL){

```

```
        count ++;
        curr_task = curr_task->next;
    }
  return count;
}
```

Listing 2: Function nbr_of_tasks

(b) After analysing the given test cases , the result is tabulated as follows, for scheduling results.

Test Case	**Liu-Layland Criteria**	**Response time analysis**
Test 1	SCHED_UNKNOWN	SCHED_YES
Test 2	SCHED_UNKNOWN	SCHED_NO
Test 3	SCHED_UNKNOWN	SCHED_NO
Test 4	SCHED_UNKNOWN	SCHED_YES
Test 5	SCHED_UNKNOWN	SCHED_YES

(c) The implementation of the functions are given below:

```

int schedulable_Liu_Layland()
{



    int schedulable = SCHED_UNKNOWN;
    double U =0;
    task* curr_task = first_task;

    while (curr_task != NULL) {
        U += (double) curr_task->WCET / curr_task->period;
        curr_task = curr_task->next;
        }
        int N = nbr_of_tasks();
    double upper_limit = N*(pow(2,1/N) - 1);
    int check = (U<=upper_limit);
    if(check){
      schedulable = SCHED_YES;
    }

    return schedulable;
}
```

Listing 3: Function schedulable_Liu_Layland

```
int schedulable_response_time_analysis()
{

    int schedulable = SCHED_YES;
    task* curr_task = first_task;
    int N = nbr_of_tasks();

    while (curr_task != NULL) {
        int w_prev = 0;
        int w_next = curr_task->WCET;
        while (w_prev!=w_next)
        {
            w_prev = w_next;
            w_next = curr_task->WCET;
            task* hp = first_task;
            while (hp!=NULL)
            {
                if(hp->priority > curr_task->priority)
                    {
                       w_next += ceil((double) w_prev/hp->period)* hp->WCET;
                    }
                    hp =hp->next;
            }

        }
        int R = w_next;
        if (R > curr_task->deadline)
        {
            schedulable = SCHED_NO;
        }
        curr_task = curr_task->next;
        }
    return schedulable;
}
```

Listing 4: Function schedulable_response_time_analysis

After implementing this function, I was able to pass all the test case with understanding in question (b).

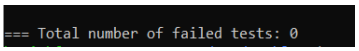

Figure 2.1: Output