Politecnico Di Torino

Department of CONTROL AND COMPUTER ENGINEERING (DAUIN)

Master Degree in Mechatronic Engineering

2024/2025

Operating Systems for Embedded Systems - Prof. Violante

Laboratory 03 – Scheduling, Priority Inversion, Priority Ceiling, Deadlock

Wednesday, November 13, 2024

Fornasier Francesca – 338688

Exercise 1

1.1) OIL file description

```
OIL_VERSION = "2.5";

CPU exercise1 {
    OS config {
        STATUS = STANDARD;
        BUILD = TRUE {
            TRAMPOLINE_BASE_PATH = "/Users/francescafornasier/github/Operating_Systems_for_Embedded_Systems/trampoline";
            APP_NAME = "lab03ex1";
            APP_SRC = "ex1.cpp";
            CPPCOMPILER = "avr-g++";
            COMPILER = "avr-gcc";
            LINKER = "avr-gcc";
            ASSEMBLER = "avr-gcc";
            COPIER = "avr-objcopy";
            SYSTEM = PYTHON;
            LIBRARY = serial;
        };
        SYSTEM_CALL = TRUE;
    };

    APPMODE stdAppmode {};
```

First part of the oil file, containing directives for the oil compiler.
We include the library “serial” so we can use the serial monitor of the ArduinoUno.

```
    ALARM alarmA_1000 {
        COUNTER = SystemCounter;
        ACTION = ACTIVATETASK { TASK = TaskA; };
        AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 1000; CYCLETIME = 1000; };
    };

    ALARM alarmB_1500 {
        COUNTER = SystemCounter;
        ACTION = ACTIVATETASK { TASK = TaskB; };
        AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 1500; CYCLETIME = 1500; };
    };

    ALARM alarmC_2800 {
        COUNTER = SystemCounter;
        ACTION = ACTIVATETASK { TASK = TaskC; };
        AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 2800; CYCLETIME = 2800; };
    };
```

We are asked to define three periodic tasks, so we set three alarms that when expire activate the associated task. Alarm time is qual to cycle time because all the tasks have property autostart equal to true that will put them ins the ready queue at the starting of the operating system.

```
    TASK TaskA {
        PRIORITY = 3;
        AUTOSTART = TRUE { APPMODE = stdAppmode; };
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };

    TASK TaskB {
        PRIORITY = 2;
        AUTOSTART = TRUE { APPMODE = stdAppmode; };
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };

    TASK TaskC {
        PRIORITY = 1;
        AUTOSTART = TRUE { APPMODE = stdAppmode; };
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };
};
```

Definition of the tasks with autostart set to true, activation equal to true so that at most one instance of each task can be activated in each period.
Property schedule equal to full so that the scheduler can perform preemption based on priority of the tasks.
Timeline scheduling is not feasible because the minor cycle is 100 ms, that is lower than the execution time of the tasks.
We use rate monotonic scheduling algorithms, priorities are static and have to be inversely proportional to the periods of the tasks. We assign 3 to TaskA (period 1000 ms), 2 to TaskB (period 1500 ms) and 1 to TaskC (period 2800 ms).
We can verify using the feasibility criteria:

$$\left(1+\frac{200}{1000}\right)\times\left(1+\frac{700}{1500}\right)\times\left(1+\frac{300}{2800}\right)\leq 2$$

We get 1.949 that is lower than 2, so the scheduling is feasible.

1.2) C++ file description

```
#include "tpl_os.h"
#include "tpl_com.h"
#include "Arduino.h"

DeclareAlarm(alarmA_1000);
DeclareAlarm(alarmB_1500);
DeclareAlarm(alarmC_2800);

int worstR_C = 0;

void setup(){
    Serial.begin(115200);
}
```

In the cpp file we define the behavior of our application.
In the first part we declare the alarms and we define a variable to store the value of the worst case response time.
In the setup function we initialise the communication over the serial port.

```
void do_things(int ms) {
    unsigned long mul = ms * 198UL;  // corrected to have expected duration of tasks
    unsigned long i;
    for(i=0; i<mul; i++)
        millis();
}
```

We implement the do_things function that simulate a workload of a certain number of milliseconds (provided as parameter by the caller)

```
TASK(TaskA) {
    unsigned long finishTime;
    int periodA = 1024;
    static int newInstA_time = 0;
    int respTime;

    Serial.println("A start");

    do_things(200);

    finishTime = millis();

    Serial.println("A end");

    // calculate response time
    respTime = finishTime - newInstA_time;
    Serial.print("A resTime: ");
    Serial.println(respTime);

    newInstA_time += periodA;

    TerminateTask();
}
```

TaskA: we have a variable newInstA_time initialised with 0 and incremented by the value of the period every time we execute the task, that indicates the time instant in which the instance of the task has been activated. We call the function do_things to simulate the execution time of 200 ms, then we compute the response time comparing the time when the do_things function terminates and the activation time and print it.
As last operation we terminate the task.

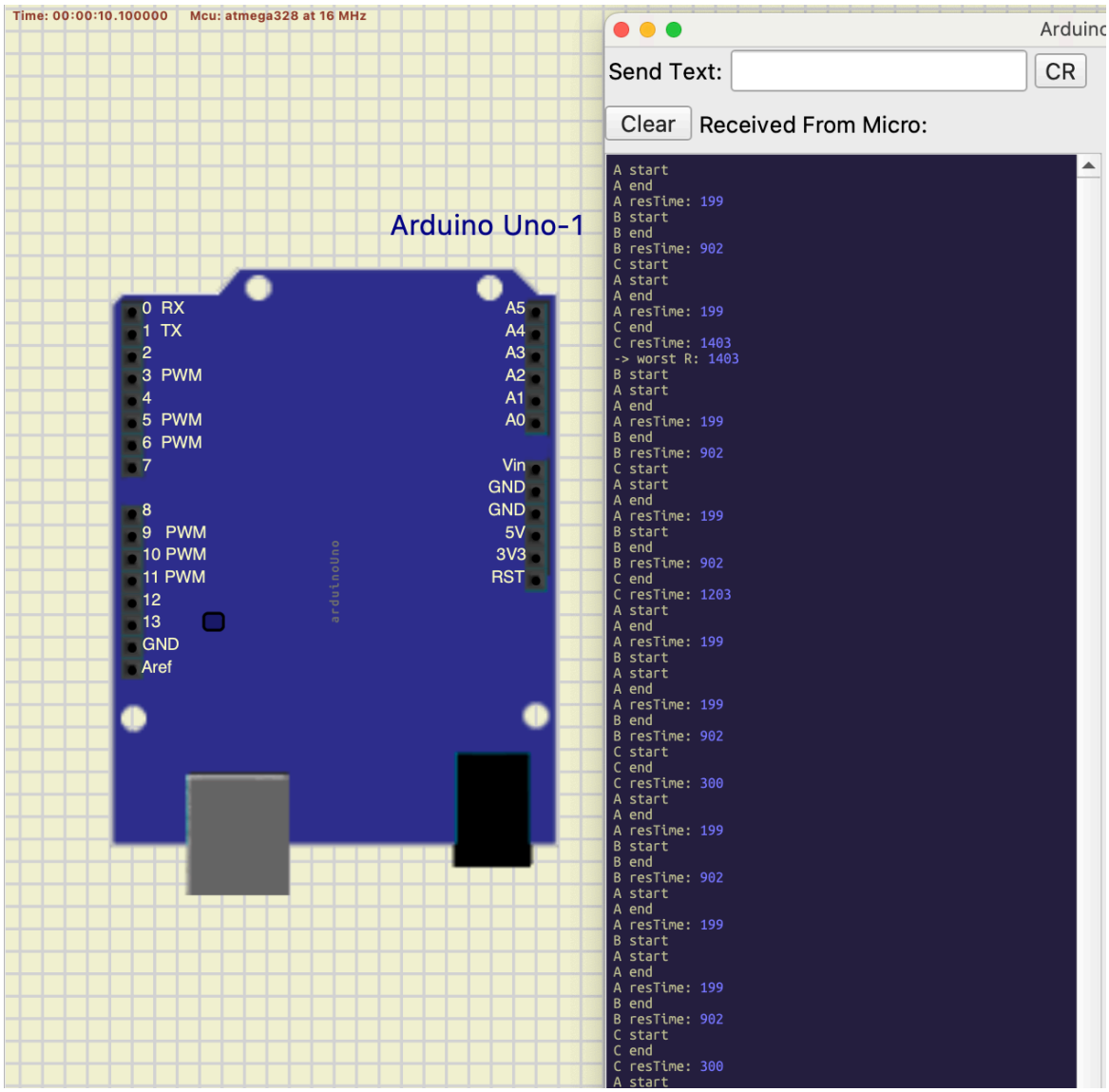
```
TASK(TaskB) {
    unsigned long finishTime;
    int periodB = 1536;
    static int newInstB_time = 0;
    int respTime;

    Serial.println("B start");

    do_things(700);

    finishTime = millis();

    Serial.println("B end");

    // calculate response time
    respTime = finishTime - newInstB_time;
    Serial.print("B resTime: ");
    Serial.println(respTime);

    newInstB_time += periodB;

    TerminateTask();
}
```

TaskB does the same thing. We have variable newInstB_time to store the time instant of the last activation of the task initialised to 0, we simulate execution time of 700 ms then we get the current time with millis().
We get and print the response time calculating current time - activation time.
We increment the activation time variable with the period duration and then terminate the task.

```
TASK(TaskC) {
    unsigned long finishTime;
    int periodC = 2867;
    static int newInstC_time = 0;
    int respTime;

    Serial.println("C start");

    do_things(300);

    finishTime = millis();

    Serial.println("C end");

    // calculate response time
    respTime = finishTime - newInstC_time;
    Serial.print("C resTime: ");
    Serial.println(respTime);

    newInstC_time += periodC;

    if (respTime > worstR_C){
        worstR_C = respTime;
        Serial.print("-> worst R: ");
        Serial.println(worstR_C);
    }

    TerminateTask();
}
```

Also for task C we simulate the execution time (here it is 300 ms) we calculate the response time of the task by getting the current time and comparing it with the time instant in which the task was activated and we print it.
We update the newInstC_time variable.
We are requested to measure the worst case response of the task so every time we calculate the response time we compare it with the worst value we obtained so far, if the new one is greater we update the worst case response time and print it out on the serial monitor.

1.3) Output analysis

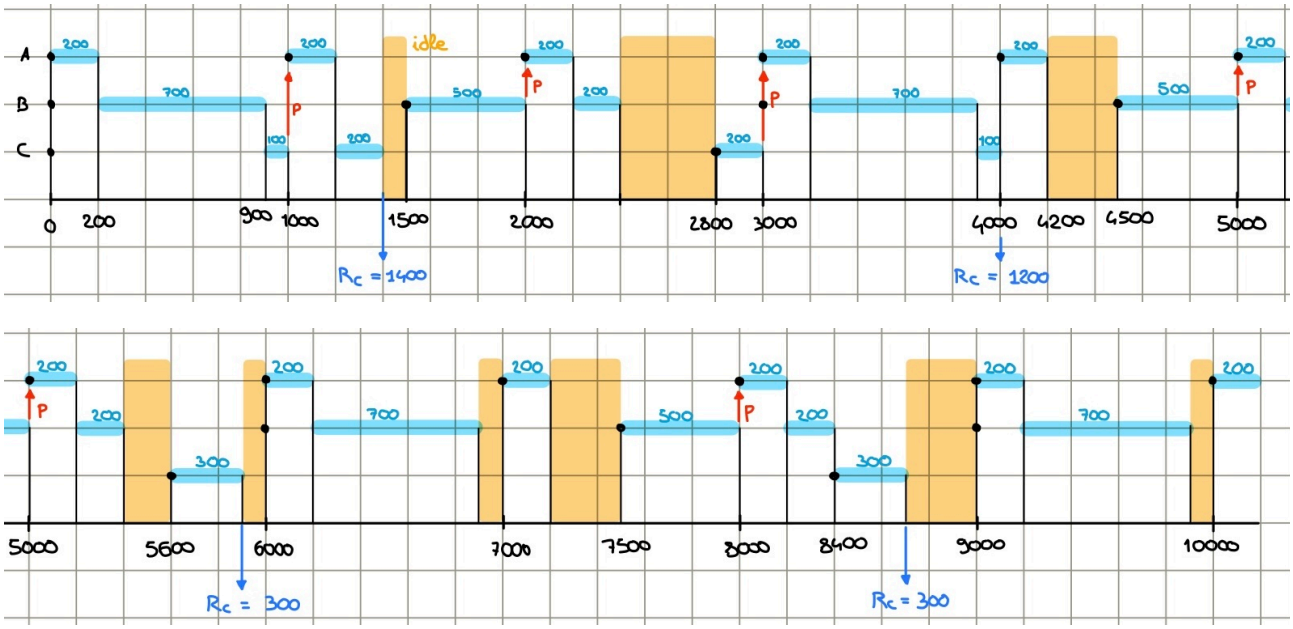


At time 0 all the tasks are activated and set as ready. First task to be execute is A, that has highest priority so is starts and ends without being interrupted. After A terminates B can execute without interruptions.
C starts execution but, since it has lowest priority and a new instance of A is activated during the execution, it is blocked and A starts executing again. A terminates and C can conclude its execution too.

THEORETICAL WORST CASE RESPONSE TIME

Since we are interested in the lowest priority task, the worst case happens when all the tasks are activated at the same time, so at time 0.
In this case, C has to wait for A0 and B0 to be executed and also a second instance of task A that starts during the execution of C and preempts it.

Comparing the theoretical result with our simulation, we can notice some differences because the time the tasks require is not exactly the one we expect. We also do not take into account the time required for printing over the serial port, that adds to the required time, and also the amount of time used by the scheduler to do the context switch (that we assumed to be negligible in the theoretical analysis).



1.4) Side question: why would delay(ms) lead to an incorrect behavior

Delay function does not lead to the desired behavior because it measures time based on the system clock, so it will continue “counting" even though the task is not running because it has been preempted by a higher priority task.

Exercise 2

2.1) OIL file description

```
    RESOURCE Sem{
        RESOURCEPROPERTY = STANDARD;
    };

    TASK TaskA {
        PRIORITY = 3;
        AUTOSTART = TRUE { APPMODE = stdAppmode; };
        ACTIVATION = 1;
        SCHEDULE = FULL;
        RESOURCE = Sem;
    };

    TASK TaskB {
        PRIORITY = 2;
        AUTOSTART = TRUE { APPMODE = stdAppmode; };
        ACTIVATION = 1;
        SCHEDULE = FULL;
    };

    TASK TaskC {
        PRIORITY = 1;
        AUTOSTART = TRUE { APPMODE = stdAppmode; };
        ACTIVATION = 1;
        SCHEDULE = FULL;
        RESOURCE = Sem;
    };
};
```

The oil file is the same as the one of the previous exercise to which we added a shared exclusive resource Sem, needed to manage the tasks accessing to the critical section.
In task A and C we specify the resource they are using.

2.2) C++ file description

```
#include "tpl_os.h"
#include "tpl_com.h"
#include "Arduino.h"

DeclareAlarm(alarmA_1000);
DeclareAlarm(alarmB_1500);
DeclareAlarm(alarmC_2800);
DeclareResource(Sem);

void setup(){
    Serial.begin(115200);
    Serial.println("Output: start/finish time of tasks and CS, response time of every task");
}

void do_things(int ms) {
    unsigned long mul = ms * 198UL;  // corrected to have expected duration of tasks
    unsigned long i;
    for(i=0; i<mul; i++)
        millis();
}
```

Here we define the behavior of the application.
In the initial part we include the libraries we need to work with Arduino and we declare the alarms and the shared resource.
The setup function initialise the communication over the serial port and prints what are the informations that we will print during the execution.

The do_things function, like in exercise 1, is what the tasks use to simulate a workload of the specified number of milliseconds.

```
55  TASK(TaskB) {
56      unsigned long time = millis();
57
58      Serial.print("B start: ");
59      Serial.println(time);
60
61      do_things(700);
62
63      time = millis();
64      Serial.print("B end: ");
65      Serial.println(time);
66
67      TerminateTask();
68  }
```

TaskB is the only one that does not have a critical section so it does not have to wait for the resource to become available. Task B prints the time when it starts the execution, executes for 700 ms (do_things with 700 given as a parameter) then gets the time when it ends the execution and prints it.
Finally the task terminates.

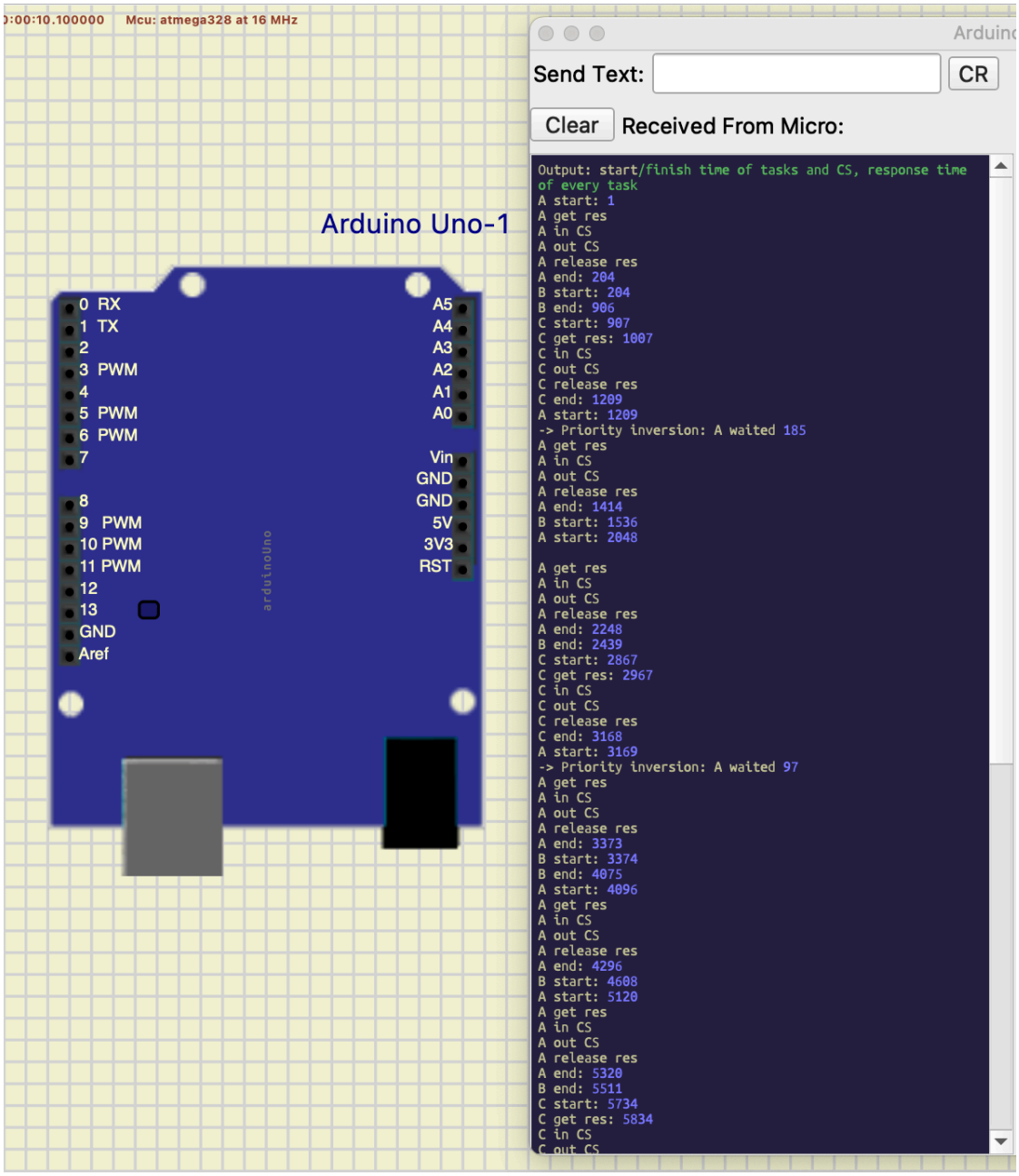
```
70  TASK(TaskC) {
71      unsigned long time = millis();
72
73      Serial.print("C start: ");
74      Serial.println(time);
75
76      do_things(100);
77
78      GetResource(Sem);
79      Serial.println("C get res");
80
81      Serial.println("C in CS");
82
83      do_things(200);
84
85      Serial.println("C out CS");
86
87      time = millis();
88      Serial.print("C release res\nC end: ");
89      Serial.println(time);
90
91      ReleaseResource(Sem);
92
93      TerminateTask();
94  }
```

Task C is the least important task. It prints the starting time, runs for 100 ms and then performs the GetResource to enter the critical section.
When the resource becomes available the task gets it, prints an alert on the monitor saying that he got the resource and starts the critical section and executes for 200 ms. At the end of the critical section it prints again an alert and releases the resource so that other tasks could enter the critical section.

```
22  TASK(TaskA) {
23      unsigned long time=millis();
24      int deltaT;
25      int periodA = 1024;
26      static int newInstA_time = 0;
27
28      Serial.print("A start: ");
29      Serial.println(time);
30
31      if((deltaT = time-newInstA_time) > 1){  // > 0 but i wanted to have some margin
32          Serial.print("-> Priority inversion: A waited ");
33          Serial.println(deltaT);
34      }
35
36      Serial.println("A get res");
37      GetResource(Sem);
38
39      Serial.println("A in CS");
40      do_things(200);
41
42      newInstA_time += periodA;
43
44      Serial.println("A out CS");
45
46      time = millis();
47      Serial.print("A release res\nA end: ");
48      Serial.println(time);
49
50      ReleaseResource(Sem);
51
52      TerminateTask();
53  }
```

Task A starts executing, saves the time instant in a variable and prints it.
It checks if priority inversion happened comparing the starting time with the time when the task was activated (the value is obtained through a static variable initialised with 0 and to which we add the value of the period every time we execute the task): if it is greater than 0 task A had to wait to be executed even though it has highest priority, so we can affirm priority inversion occurred.
The task gets the resource, executes the critical section then prints the termination time and releases the resource. Finally, if terminates.

2.3) Output analysis



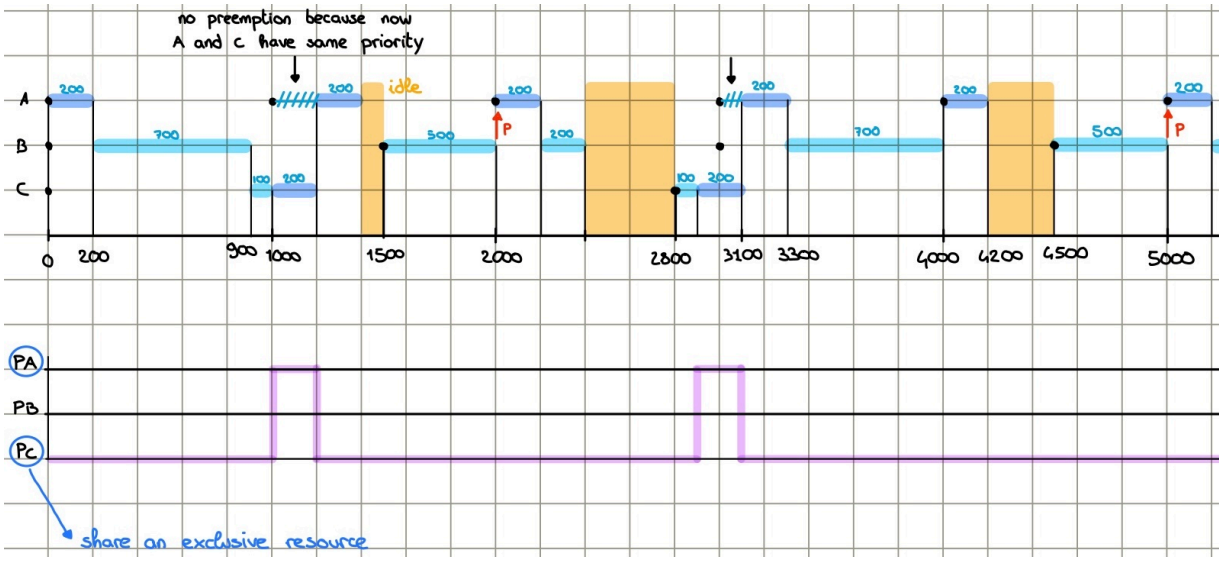
From the output we can see at time 0, A starts the execution and gets the resource because it has highest priority. It executes the critical section then releases the resource and terminates at time 204. At time 204 B is able to start and terminate its execution since it has the highest priority among the active tasks. At time 907 also C can start the execution of the non-critical section. After 100 ms, at time 1007, it can get the resource and continue with the execution of the critical section, terminating at time 1209.

Now we have a second activation of task A: we can see that it last activation happened 185 ms before it can actually start executing.

This happens because the instance of A was activated during the execution to the critical section of task C, that could not be interrupted. OSEK uses Immediate Priority Ceiling mechanism to deal manage the priority inversion, so the priority of the task running the critical section is temporarily raised to the maximum priority among the tasks that share the exclusive resource.

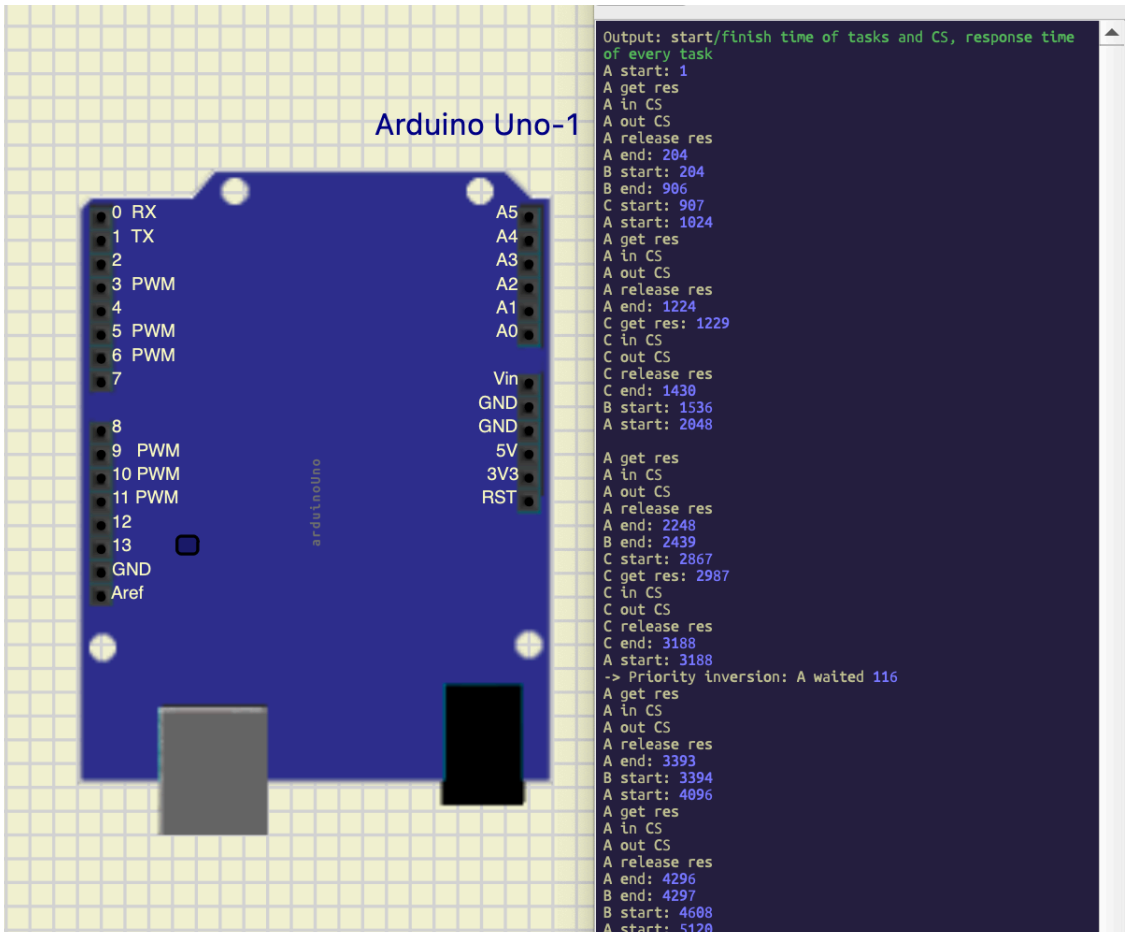
According to this, task C assumed the same priority of A, disabling preemption for both tasks (B has now priority lower than C).

This result is consistent with the theoretical result we expected form the timeline, exception made for some variations in the duration of the tasks.



But, since the behavior of the real system is necessarily different from the theoretical expectations, some small changes in the timing may lead to results that are different from whats we expected

For example, we can see from the timeline that at time 1000 ms task A is activated and at the same time task C requests to enter the critical section. If the non-critical section of task C takes a little longer than the nominal value of execution time, task A is able to preempt it before priority ceiling is performed, so it can execute first and no priority inversion occurs.



Exercise 3

3.1) OIL file description

The oil file is almost the same of the previous exercise but, instead of defining a resource, we want to use messages to implement the critical section; we define tho message objects, one for sending the messages and the other to receive them.

```
41  MESSAGE sent {
42      MESSAGEPROPERTY = SEND_STATIC_INTERNAL{
43          CDATATYPE = "int";
44      };
45  };
46
47  MESSAGE received {
48      MESSAGEPROPERTY = RECEIVE_QUEUED_INTERNAL {
49          SENDINGMESSAGE = sent;
50          QUEUESIZE = 1;
51      };
52  };
```

Messages are static internal because we need to send them inside the same processor, the receiving message is queued because we want the message to be removed when one task gets to read it. Queue size is set to 1 since the message is used as a token to get in the critical section, there will never be more than one message sent before a task reads it.

```
54  TASK Init {
55      PRIORITY = 4;
56      AUTOSTART = TRUE { APPMODE = stdAppmode; };
57      ACTIVATION = 1;
58      SCHEDULE = FULL;
59      MESSAGE = sent;
60  };
```

We have an Initialisation task that is only meant to send the first message. For this reason it is aperiodic, with autostart equal to true and highest priority since it has to be the first one to start executing.

```
62  TASK TaskA {
63      PRIORITY = 3;
64      AUTOSTART = TRUE { APPMODE = stdAppmode; };
65      ACTIVATION = 1;
66      SCHEDULE = FULL;
67      MESSAGE = sent;
68      MESSAGE = received;
69  };
71  TASK TaskB {
72      PRIORITY = 2;
73      AUTOSTART = TRUE { APPMODE = stdAppmode; };
74      ACTIVATION = 1;
75      SCHEDULE = FULL;
76  };
77
78  TASK TaskC {
79      PRIORITY = 1;
80      AUTOSTART = TRUE { APPMODE = stdAppmode; };
81      ACTIVATION = 1;
82      SCHEDULE = FULL;
83      MESSAGE = sent;
84      MESSAGE = received;
85  };
```

Task A, B, C are defined as in the previous exercise but we also specify the messages every task is going to use. B does not have a critical section, it does not need messages; A and C have to be able to both receive and send messages.

3.2) C++ file description

```
#include "tpl_os.h"
#include "tpl_com.h"
#include "Arduino.h"

DeclareAlarm(alarmA_1000);
DeclareAlarm(alarmB_1500);
DeclareAlarm(alarmC_2800);
DeclareMessage(sent);
DeclareMessage(received);
int worstR_C = 0;

void setup(){
    Serial.begin(115200);
}

void do_things(int ms) {
    unsigned long mul = ms * 199UL;  // corretto in modo da avere esecuzione di task A pari a 200 ms
    unsigned long i;
    for(i=0; i<mul; i++)
        millis();
}
```

In the fist part we added the declaration of the message objects, the setup and do_things functions are the same as exercise 1 and 2.

```
TASK(Init) {
    int x = 1;
    if( SendMessage(sent, &x) != E_OK )
        Serial.println("Error sending message");

    TerminateTask();
}
```

The Init task has a variable with the payload of the message, it sends the message and terminates

```
TASK(TaskA) {
    unsigned long time=millis();
    int msg_R, msg_S=1;

    Serial.print("A start: ");
    Serial.println(time);

    while( ReceiveMessage(received, &msg_R) != E_OK );  // do nothing, locked in busy waiting

    Serial.println("A received message");

    Serial.println("A in CS");
    do_things(200);

    Serial.println("A out CS");

    time = millis();
    Serial.print("A send message\nA end: ");
    Serial.println(time);

    if( SendMessage(sent, &msg_S) != E_OK)
        Serial.println("Error sending message");

    TerminateTask();
}
```

Task A prints its start time, then is locked in busy waiting checking for new messages.
When the operation is successful, it prints a message and enters the critical section. After that it prints the termination time and send a message back before terminating.

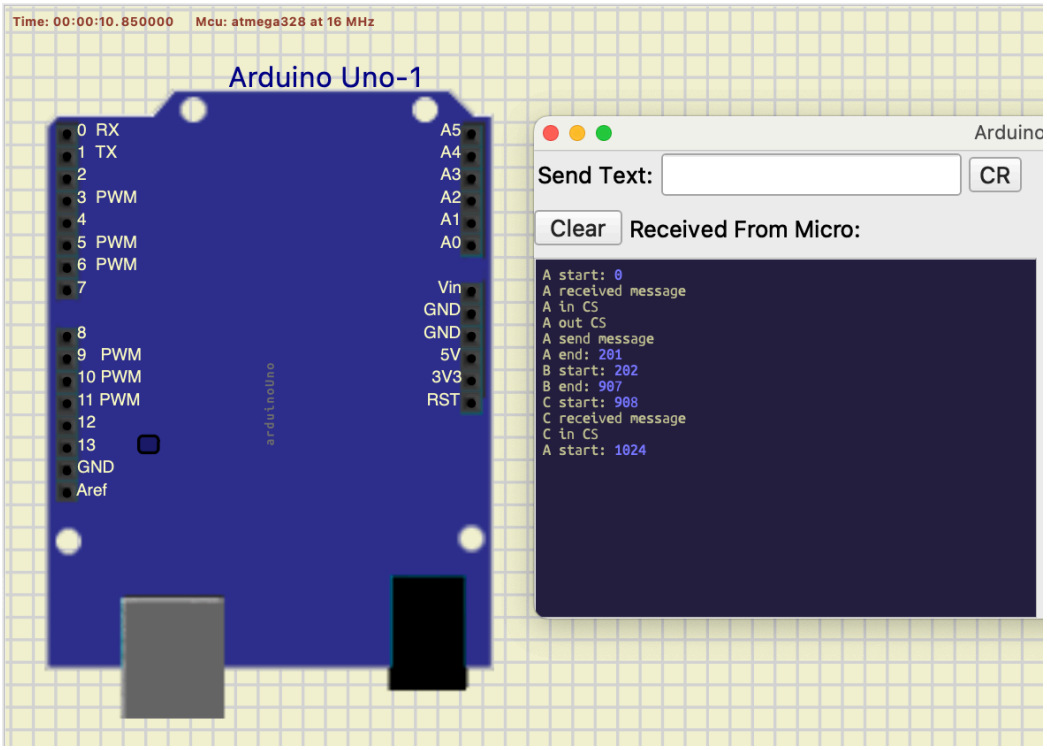
```
58  TASK(TaskB) {
59      unsigned long time = millis();
60
61      Serial.print("B start: ");
62      Serial.println(time);
63
64      do_things(700);
65
66      time = millis();
67      Serial.print("B end: ");
68      Serial.println(time);
69
70      TerminateTask();
71  }
```

Task B is the same as before, just simulates the execution for 700 ms and prints start and termination time.

```
73  TASK(TaskC) {
74      unsigned long time = millis();
75      int msg_R, msg_S=1;
76
77      Serial.print("C start: ");
78      Serial.println(time);
79
80      do_things(100);
81
82      while( ReceiveMessage(received, &msg_R) != E_OK );  // do nothing, locked in busy waiting
83
84      Serial.println("C received message");
85
86      Serial.println("C in CS");
87
88      do_things(200);
89
90      Serial.println("C out CS");
91
92      time = millis();
93      Serial.print("C send message\nC end: ");
94      Serial.println(time);
95
96      SendMessage(sent, &msg_S);
97
98      TerminateTask();
99  }
```

Task C prints the start time, executes for 100 ms and then is in busy waiting until it gets a message. When it receives a massage it executes the critical section for 200 ms and before terminating sends a message to unlock another task that was waiting.

3.3) Output analysis



At time 0 task A with highest priority starts running and receives the message from Init; it can enter the critical section and at the end it sends a message and terminates at time 201.
B executes next (from 201 to 907 ms) because it has intermediate priority.
At time 908 C starts executing its non-critical section, then receives the message from A and starts also the critical section.
As we can see here, a new instance of A is activated during the execution of critical section of C and can preempt it due to its higher priority.
At this moment C is blocked because it was preempted before sending the message and A is blocked in the while loop checking for messages that no task is able to send.
So this solution leads to a state of deadlock.