****

**Department of CONTROL AND COMPUTER ENGINEERING (DAUIN)**

**Master Degree in Mechatronic Engineering**

**2021/2022**

**Operating systems for embedded systems – Prof. Violante**

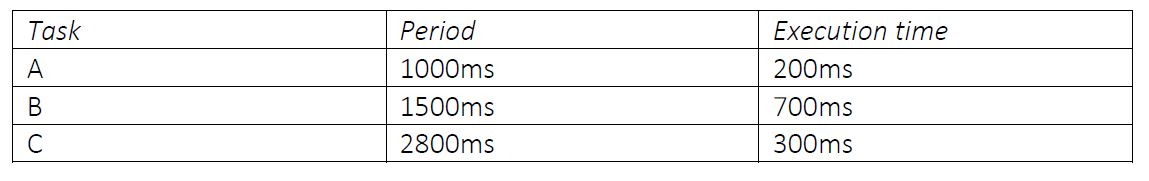
**Laboratory 03**

Andrea Catania s280665

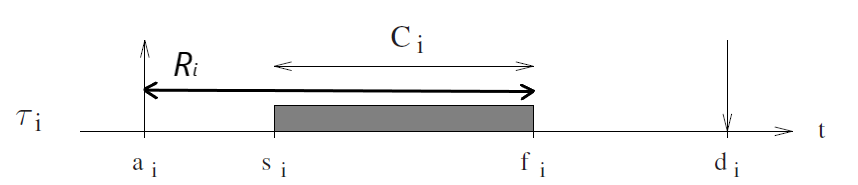
Andrea Usai s292759

**EXERCISE 1**

The application will be composed by 3 periodic tasks with the following characteristics:



The purpose of this exercise is to find a way to measure the response time of a task defined as the elapsed time between the activation ()and the finishing() time of the task as described in the figure below:



Since all the tasks are periodic we assigned the priorities of the tasks using the Rate Monotonic scheduling policy. In particular we assigned the following priorities(as it is possible to observe in the task definition in the OIL file):

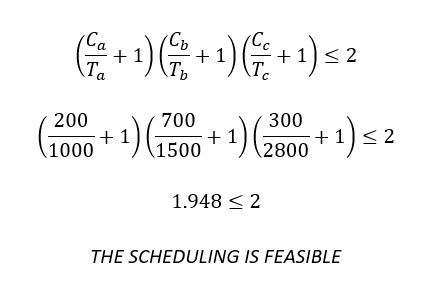
-Priority 3 ->Task A ()

-Priority 2 -> Task B ()

-Priority 1 -> Task C ()

**FEASIBILITY**

Before starting to write the code we ensured that the rate monotonic scheduling was feasible and it was computed as follows:

****

**FILE.OIL**

Basically, the duration of a major cycle was considered as observation time. It was computed as the Least Common Multiple among all periods of the task in order to be able to consider a good number of instances for each task. It follows that:

OIL\_VERSION = "2.5";

IMPLEMENTATION trampoline {

/\* This fix the default STACKSIZE of tasks \*/

TASK {

UINT32 STACKSIZE = 32768 ;

} ;

/\* This fix the default STACKSIZE of ISRs \*/

ISR {

UINT32 STACKSIZE = 32768 ;

} ;

};

CPU default {

OS config {

STATUS = EXTENDED;

TRACE = FALSE;

BUILD = TRUE {

APP\_SRC = "exercise1.c";

TRAMPOLINE\_BASE\_PATH = "../../..";

CFLAGS="-O0 -ggdb";

APP\_NAME = "exercise1\_exe";

LINKER = "gcc";

SYSTEM = PYTHON;

};

};

APPMODE stdAppmode {};

ALARM A {

COUNTER = SystemCounter;

ACTION = ACTIVATETASK { TASK = tau\_A; };

AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 100; CYCLETIME = 100; };

};

ALARM B {

COUNTER = SystemCounter;

ACTION = ACTIVATETASK { TASK = tau\_B; };

AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 150; CYCLETIME = 150; };

};

ALARM C {

COUNTER = SystemCounter;

ACTION = ACTIVATETASK { TASK = tau\_C; };

AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 280; CYCLETIME = 280; };

};

ALARM stopper {

COUNTER = SystemCounter;

ACTION = ACTIVATETASK { TASK = stop; };

AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 4200; CYCLETIME = 0; };

};

TASK tau\_A {

PRIORITY = 3;

AUTOSTART = TRUE { APPMODE = stdAppmode;};

ACTIVATION = 1;

SCHEDULE = FULL;

};

TASK tau\_B {

PRIORITY = 2;

AUTOSTART = TRUE { APPMODE = stdAppmode;};

ACTIVATION = 1;

SCHEDULE = FULL;

};

TASK tau\_C {

PRIORITY = 1;

AUTOSTART = TRUE { APPMODE = stdAppmode;};

ACTIVATION = 1;

SCHEDULE = FULL;

};

TASK stop {

PRIORITY = 99;

AUTOSTART = FALSE;

ACTIVATION = 1;

SCHEDULE = FULL;

};

};

**FILE .C**

In order to measure the response time of the Task C (and determine the worst-case response time), it was necessary to compute two intervals:

1. The execution time.
2. The dispatch time.

In order to compute the first interval we used the system call *gettimeofday()* calling it just before and after the invocation of the *do\_other\_things()* function, storing the value respectively in two *timeval* variables called *start* and *end* and subtracting them in such a way to be able to convert the result in milliseconds. The execution time in this contest is the time elapsed between the *starting time*  and the *finishing time* .

For the second interval we used the system call *GetAlarm(ID alarm)* which allows us to obtain the remaining number of ticks before the next activation of the corrispective task. Considering that each tick of the specified target (Posix/Linux) is 10ms, the dispatch time was computed as follows:

Thus, the response time can be computed as:

**#include <time.h>**

**#include <sys/time.h>**

**#include "tpl\_os.h"**

**#include <stdio.h>**

**long sf = 1;**

**/\* Keep the caller busy for howmany\_us microseconds \*/**

**void do\_other\_things(int howmany\_us)**

**{**

**long mul = howmany\_us \* sf;**

**long i;**

**for(i=0; i<mul; i++) time(NULL);**

**}**

**#define CAL\_US 1000000**

**#define CAL\_T 5**

**void calibrate(void)**

**{**

**struct timeval now;**

**long nsf = 0;**

**printf("Calibrating delay... \r\n");**

**while(nsf == 0 || nsf - sf > CAL\_T || sf - nsf > CAL\_T)**

**{**

**struct timeval old, new;**

**long l;**

**gettimeofday(&old, NULL);**

**do\_other\_things(CAL\_US);**

**gettimeofday(&new, NULL);**

**l = (long)((new.tv\_sec - old.tv\_sec) \* 1000000 + (new.tv\_usec - old.tv\_usec));**

**nsf = CAL\_US \* sf / l;**

**sf = (sf + nsf)/2;**

**printf("... l= %ld sf= %ld\r\n", l, sf);**

**}**

**}**

**int main(void)**

**{**

**calibrate();**

**StartOS(OSDEFAULTAPPMODE);**

**return 0;**

**}**

**DeclareAlarm(A);**

**DeclareAlarm(B);**

**DeclareAlarm(C);**

**DeclareTask(tau\_A);**

**DeclareTask(tau\_B);**

**DeclareTask(tau\_C);**

**TASK(tau\_A)**

**{**

**static int C\_A = 200000; //200 msec = 200 000 us**

**long time;**

**struct timeval start;**

**struct timeval end;**

**gettimeofday(&start, NULL);**

**do\_other\_things(C\_A);**

**gettimeofday(&end, NULL);**

**// time = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000);**

**// printf(" task A eseguita.... tempo impiegato=%ld \r\n\n", time);**

**TerminateTask();**

**}**

**TASK(tau\_B)**

**{**

**static int C\_B = 700000; //700 msec = 700 000 us**

**long time;**

**struct timeval start;**

**struct timeval end;**

**gettimeofday(&start, NULL);**

**do\_other\_things(C\_B);**

**gettimeofday(&end, NULL);**

**// time = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000);**

**// printf(" task B eseguita.... tempo impiegato=%ld \r\n\n", time);**

**TerminateTask();**

**}**

**TASK(tau\_C)**

**{**

**static int C\_C = 300000; //300 msec = 300 000 us**

**long time;**

**long tic;**

**long dispatch = 0;**

**struct timeval start;**

**struct timeval end;**

**GetAlarm(C,&tic);**

**gettimeofday(&start, NULL);**

**do\_other\_things(C\_C);**

**gettimeofday(&end, NULL);**

**dispatch = 2800-tic\*10;**

**time = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000);**

**printf("#Activation Task C\t|execution time=%ld\t|response\_time = %ld\r\n\n", time,dispatch+time);**

**//printf("n. tics = %ld response\_time = %ld\r\n\n",tic,dispatch+time);**

**TerminateTask();**

**}**

**TASK(stop)**

**{**

**CancelAlarm(A);**

**CancelAlarm(B);**

**CancelAlarm(C);**

**printf("Shutdown\r\n");**

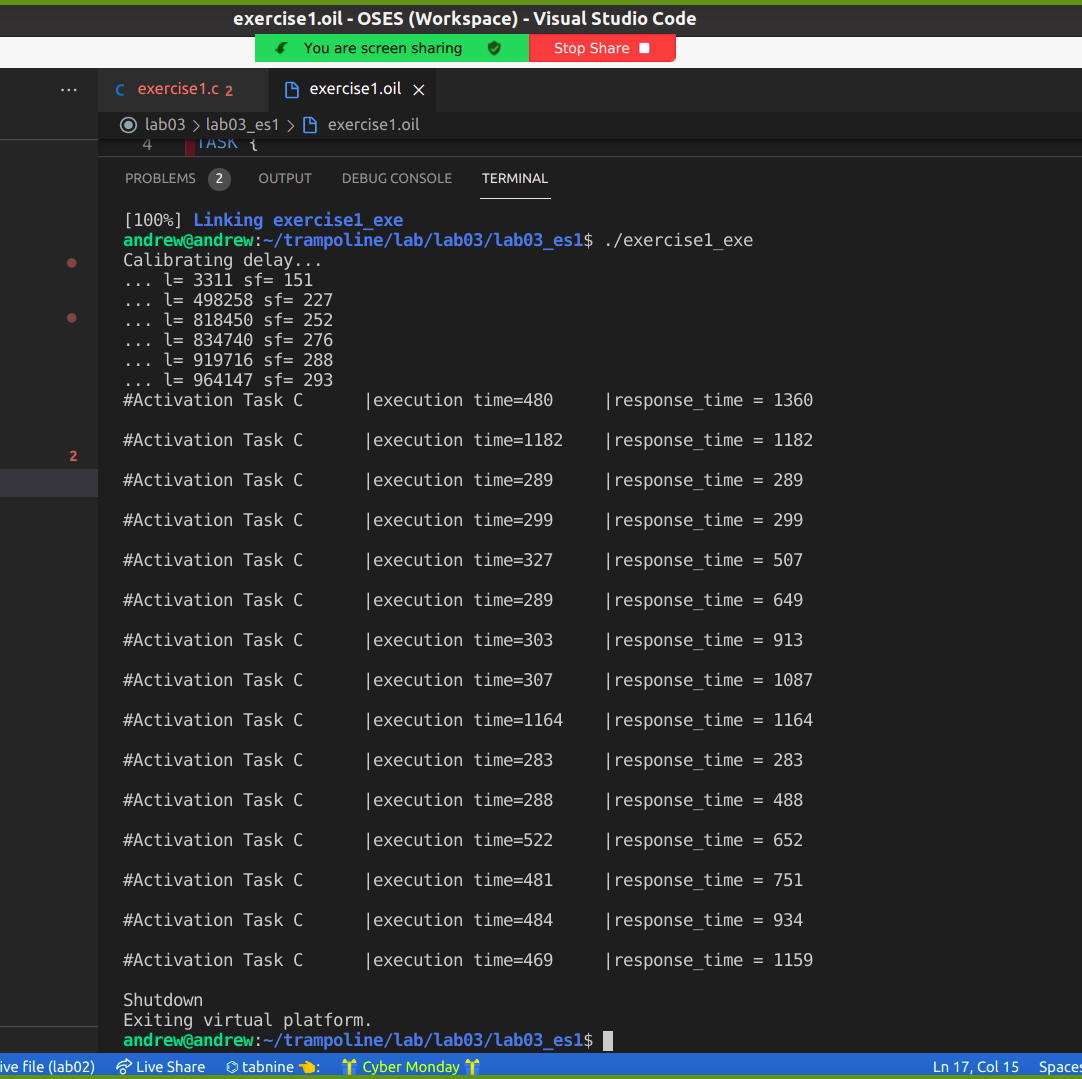
**ShutdownOS(E\_OK);**

**TerminateTask();**

**}**

**RESULTS EXERCISE 1**

Results of application execution are the following:

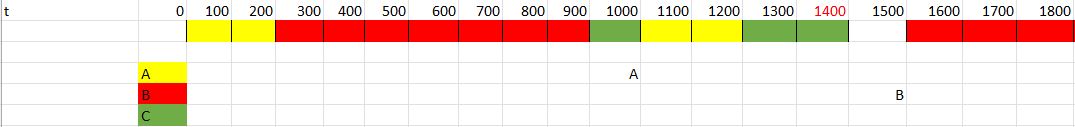


As it is possible to see from the above picture, the execution time of all tasks is not the same as the theoretical one(where we have considered that all the tasks run exactly for its own WCET). This is due to the real execution of the application and in particular on the target system we are using, Ubuntu 20.04 Linux distribution, which is not a real-time operating system.

In the execution of the code we can see an oscillation(sometimes also huge) of the execution time due to the preemption of this task in favour of the higher priority tasks. The Worst-Case Execution Time as it was defined, doesn’t take into account the interferences due to the preemption while the execution time we printed did.

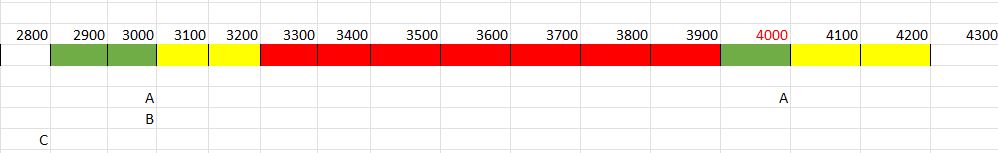
It is worth noticing that the WCET in the theoretical case corresponds to the first instance of task C(in this particular execution was 1360ms). This is the expected result as it is possible to see from the timeline we depicted below.

The obtained results reflect our expectations for all the duration of the execution that, as said before, lasts 42000msec.

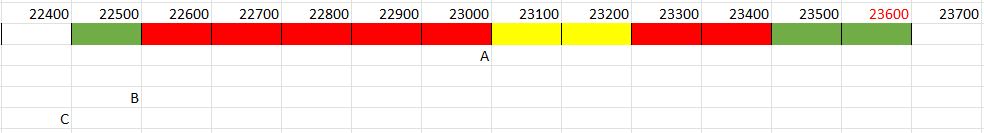


As we can see from the timeline, the extension of the response time of task C is related to the second activation of task A, which having higher priority, performs the preemption on task C.

In the same way, we can compare the second instance (real value of response time 1182 msec)



and the ninth instance of task C (real value of response time 1164 msec):



For the sake of completeness we attach the complete timeline we build.

**EXERCISE 2**

**FILE .OIL**

This time we have that task A and task C need to share a shared exclusive resource (for the sake of simplicity we decide to use a simple int variable called *shared\_variable*). In particular this shared resource will be occupied by task A for its entire execution time (200msec) while task C will retain the shared resource for 200msec only after having executed for 100 msec.

In OSEK the Ceiling Priority mechanism is implemented in order to avoid the so-called *“Priority Inversion”* problem. This problem arises when we have, for instance, tasks that want to share a mutual exclusive resource that have priority at the opposite (the highest and lower) and there are other tasks with intermediate priority. In particular, let's imagine that the low priority task is in its critical section and the task with higher priority is activated. The latter remains in the blocked state since the resource is occupied by the lower priority task. In this scenario, if an intermediate priority task is activated, since they do not use the shared resource, preempts the lower priority task. The worst case is if all the intermediate tasks are activated when the low priority task enters in its critical section. In this context the task with higher priority remains blocked for all the duration of the critical section of the lower task plus all the execution time of the intermediate tasks. With the Priority Ceiling we have that the priority of the task that enters in the critical section is promoted to the highest priority among all tasks that want to use this critical section.

In our application we have exactly this condition, in fact we have Task A with higher priority and task C with the lower that share the exclusive resource and task B has a priority in the between task A and C but does not use the exclusive shared resource.

In particular we can see the Priority Ceiling mechanism in action around time instant 1000msec and 3000msec as described in the section *“results exercise 2”* .

**OIL\_VERSION = "2.5";**

**IMPLEMENTATION trampoline {**

**/\* This fix the default STACKSIZE of tasks \*/**

**TASK {**

**UINT32 STACKSIZE = 32768 ;**

**} ;**

**/\* This fix the default STACKSIZE of ISRs \*/**

**ISR {**

**UINT32 STACKSIZE = 32768 ;**

**} ;**

**};**

**CPU default {**

**OS config {**

**STATUS = EXTENDED;**

**TRACE = FALSE;**

**BUILD = TRUE {**

**APP\_SRC = "exercise2.c";**

**TRAMPOLINE\_BASE\_PATH = "../../..";**

**CFLAGS="-O0 -ggdb";**

**APP\_NAME = "exercise2\_exe";**

**LINKER = "gcc";**

**SYSTEM = PYTHON;**

**};**

**};**

**APPMODE stdAppmode {};**

**RESOURCE shared\_variable{**

**RESOURCEPROPERTY = STANDARD;**

**};**

**ALARM A {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = tau\_A; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 100; CYCLETIME = 100; };**

**};**

**ALARM B {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = tau\_B; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 150; CYCLETIME = 150; };**

**};**

**ALARM C {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = tau\_C; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 280; CYCLETIME = 280; };**

**};**

**ALARM stopper {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = stop; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 4200; CYCLETIME = 0; };**

**};**

**TASK tau\_A {**

**PRIORITY = 3;**

**AUTOSTART = TRUE { APPMODE = stdAppmode;};**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**RESOURCE = shared\_variable;**

**};**

**TASK tau\_B {**

**PRIORITY = 2;**

**AUTOSTART = TRUE { APPMODE = stdAppmode;};**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**};**

**TASK tau\_C {**

**PRIORITY = 1;**

**AUTOSTART = TRUE { APPMODE = stdAppmode;};**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**RESOURCE = shared\_variable;**

**};**

**TASK stop {**

**PRIORITY = 99;**

**AUTOSTART = FALSE;**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**};**

**};**

**FILE .C**

La gestione della risorsa è permessa tramite l’utilizzo delle system calls *GetResource()* e *ReleaseResource(),* le quali, poste rispettivamente prima e dopo la funzione *do\_other\_things()* permettono di definire la critical section. All’interno di tale critical section si è deciso di impostare il valore di una variabile intera per simulare l’utilizzo della risorsa. In particolare, il valore di tale counter sarà impostato a 1 nel caso in cui la task A entri nella critical section e 3 nel caso in cui sia la task C ad entrare nella critical section.

**#include <time.h>**

**#include <sys/time.h>**

**#include "tpl\_os.h"**

**#include <stdio.h>**

**long sf = 1;**

**/\* Keep the caller busy for howmany\_us microseconds \*/**

**void do\_other\_things(int howmany\_us)**

**{**

**long mul = howmany\_us \* sf;**

**long i;**

**for(i=0; i<mul; i++);**

**}**

**#define CAL\_US 1000000**

**#define CAL\_T 5**

**void calibrate(void)**

**{**

**struct timeval now;**

**long nsf = 0;**

**printf("Calibrating delay... \r\n");**

**while(nsf == 0 || nsf - sf > CAL\_T || sf - nsf > CAL\_T)**

**{**

**struct timeval old, new;**

**long l;**

**gettimeofday(&old, NULL);**

**do\_other\_things(CAL\_US);**

**gettimeofday(&new, NULL);**

**l = (long)((new.tv\_sec - old.tv\_sec) \* 1000000 + (new.tv\_usec - old.tv\_usec));**

**nsf = CAL\_US \* sf / l;**

**sf = (sf + nsf)/2;**

**printf("... l= %ld sf= %ld\r\n", l, sf);**

**}**

**}**

**DeclareAlarm(A);**

**DeclareAlarm(B);**

**DeclareAlarm(C);**

**DeclareTask(tau\_A);**

**DeclareTask(tau\_B);**

**DeclareTask(tau\_C);**

**DeclareResource(shared\_variable);**

**int counter;**

**int main(void)**

**{**

**calibrate();**

**StartOS(OSDEFAULTAPPMODE);**

**return 0;**

**}**

**TASK(tau\_A)**

**{**

**static int C\_A = 200000; //200 msec = 200 000 us**

**long response;**

**long tic;**

**long dispatch;**

**struct timeval start;**

**struct timeval end;**

**GetAlarm(A, &tic);**

**dispatch=1000-tic\*10;**

**gettimeofday(&start, NULL);**

**GetResource(shared\_variable);**

**printf("Task A enters in the critical section\r\n\n");**

**counter=1;**

**do\_other\_things(C\_A);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**printf("Task A exits from critical section. Task A completed. Response time A=%ld \r\n\n", response);**

**ReleaseResource(shared\_variable);**

**TerminateTask();**

**}**

**TASK(tau\_B)**

**{**

**static int C\_B = 700000; //700 msec = 700 000 us**

**long response;**

**long tic;**

**long dispatch;**

**struct timeval start;**

**struct timeval end;**

**GetAlarm(B, &tic);**

**dispatch=1500-tic\*10;**

**gettimeofday(&start, NULL);**

**do\_other\_things(C\_B);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**printf("Task B completed. Response time B=%ld \r\n\n", response);**

**TerminateTask();**

**}**

**TASK(tau\_C)**

**{**

**static int C\_C = 300000; //300 msec = 300 000 us**

**long response;**

**long tic;**

**long dispatch;**

**struct timeval start;**

**struct timeval end;**

**GetAlarm(C, &tic);**

**dispatch=2800-tic\*10;**

**gettimeofday(&start, NULL);**

**do\_other\_things(100000);**

**printf("Task C enters in the critical section\r\n\n");**

**GetResource(shared\_variable);**

**counter=3;**

**do\_other\_things(C\_C-100000);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**printf("Task C exits from critical section. Task C completed. Response time=%ld \r\n\n", response);**

**ReleaseResource(shared\_variable);**

**TerminateTask();**

**}**

**TASK(stop)**

**{**

**CancelAlarm(A);**

**CancelAlarm(B);**

**CancelAlarm(C);**

**printf("Shutdown\r\n");**

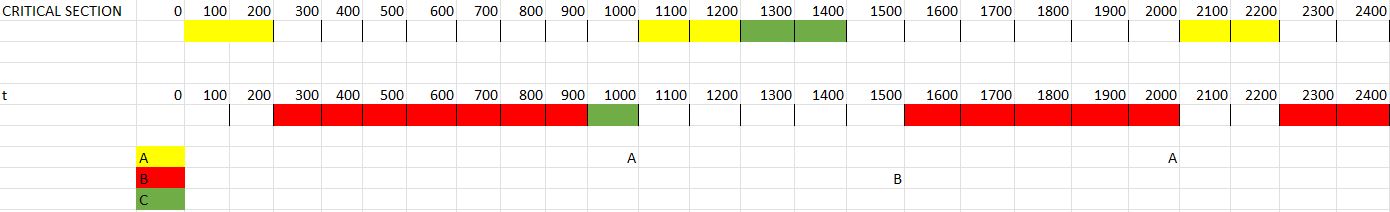
**ShutdownOS(E\_OK);**

**TerminateTask();**

**}**

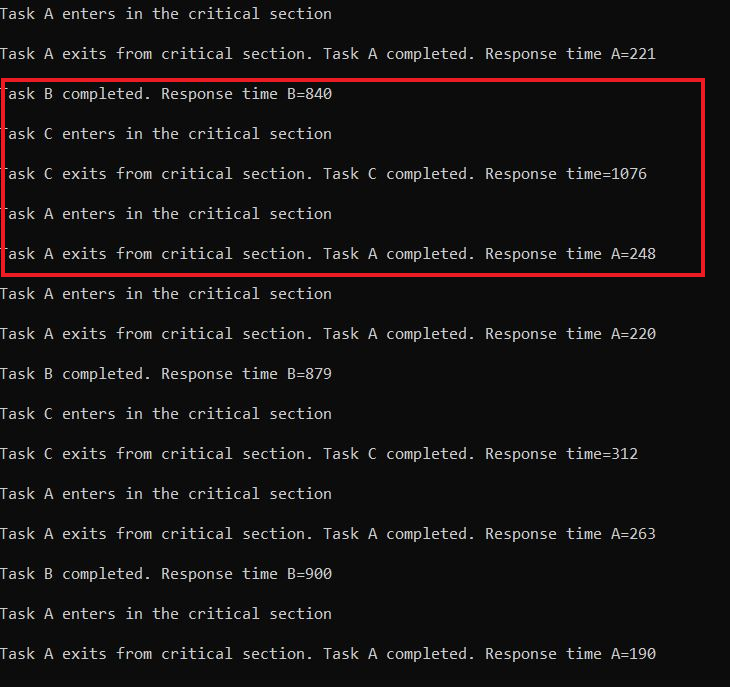
**RESULTS EXERCISE 2**

To simplify the comparison, we report just some significant part of the timeline (where we can notice priority ceiling effect during the entering in the critical section):



We can notice that, from the theoretical point of view, at the time instant t=1000msec there is the activation of task A. At the same time, there is also the entering of task C in the critical section.

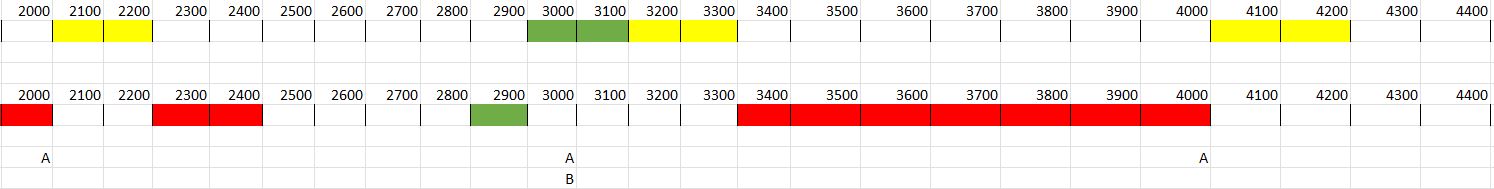
In this case, we have a request for simultaneous entry in the critical section. However, task A will enter inside it due to a higher priority.



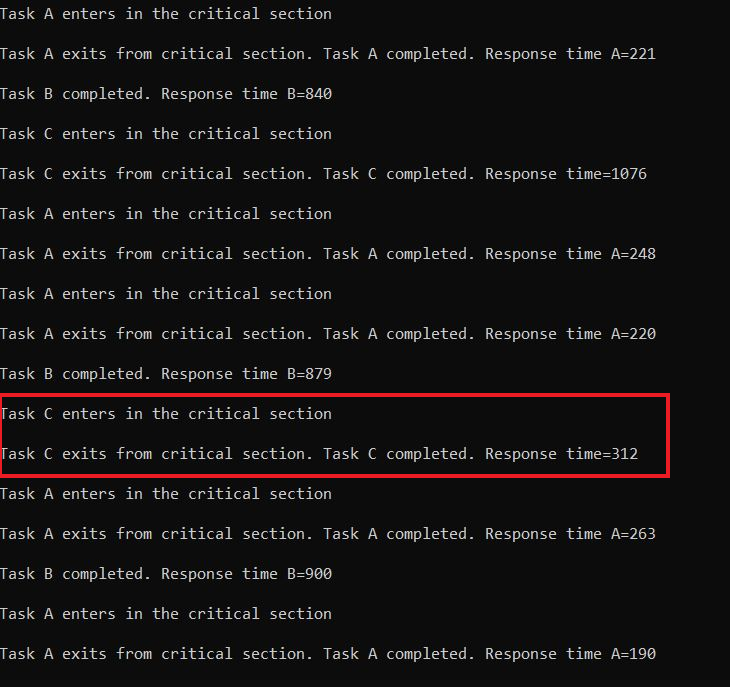
In this case, we have a different case as compared to the theoretical expectations. In particular, we can notice that task C begins the execution after 840msec from its activation (immediately after termination of task B)

Essa entrerà quindi nella critical section prima che la task A venga attivata all’istante t=1000msec (ovvero a circa 840+100=940 msec). Successivamente, una volta che la task C esce dalla critical section e termina la sua esecuzione, inizierà ad essere eseguita ed entrerà nella critical section la task A.

Another relevant case is the following:



The second instance of task C highlights the priority ceiling protocol. In particular, we can notice that at the time instant t=3000msec there is the activation of task A and B. However, task C enters in the critical section at time t=2900msec increasing its priority level to the same value of task A (because it shares the resource with it) preventing preemption (that can happen only if Priority A > Priority C or Priority B > Priority C ). After the termination of task C, it will be executed task A and then task B. The response time of task C in this case will simply be 300msec.



**EXERCISE 3**

**FILE .OIL**

**OIL\_VERSION = "2.5";**

**IMPLEMENTATION trampoline {**

**/\* This fix the default STACKSIZE of tasks \*/**

**TASK {**

**UINT32 STACKSIZE = 32768 ;**

**} ;**

**/\* This fix the default STACKSIZE of ISRs \*/**

**ISR {**

**UINT32 STACKSIZE = 32768 ;**

**} ;**

**};**

**CPU default {**

**OS config {**

**STATUS = EXTENDED;**

**TRACE = FALSE;**

**BUILD = TRUE {**

**APP\_SRC = "exercise3.c";**

**TRAMPOLINE\_BASE\_PATH = "../../..";**

**CFLAGS="-O0 -ggdb";**

**APP\_NAME = "exercise3\_exe";**

**LINKER = "gcc";**

**SYSTEM = PYTHON;**

**};**

**};**

**APPMODE stdAppmode {};**

**MESSAGE message {**

**MESSAGEPROPERTY = SEND\_STATIC\_INTERNAL {**

**CDATATYPE = "uint32";**

**};**

**};**

**MESSAGE object {**

**MESSAGEPROPERTY = RECEIVE\_UNQUEUED\_INTERNAL {**

**SENDINGMESSAGE = message;**

**FILTER = NEWISDIFFERENT;**

**INITIALVALUE = 0x00 ;**

**};**

**};**

**ALARM A {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = tau\_A; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 100; CYCLETIME = 100; };**

**};**

**ALARM B {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = tau\_B; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 150; CYCLETIME = 150; };**

**};**

**ALARM C {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = tau\_C; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 280; CYCLETIME = 280; };**

**};**

**ALARM initialization\_alarm {**

**COUNTER = SystemCounter;**

**ACTION =ACTIVATETASK { TASK = initialization\_task; };**

**AUTOSTART = TRUE {APPMODE = stdAppmode; ALARMTIME = 0; CYCLETIME = 0; };**

**};**

**ALARM stopper {**

**COUNTER = SystemCounter;**

**ACTION = ACTIVATETASK { TASK = stop; };**

**AUTOSTART = TRUE { APPMODE = stdAppmode; ALARMTIME = 4200; CYCLETIME = 0; };**

**};**

**TASK tau\_A {**

**PRIORITY = 3;**

**AUTOSTART = TRUE { APPMODE = stdAppmode;};**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**MESSAGE = message;**

**MESSAGE = object;**

**};**

**TASK tau\_B {**

**PRIORITY = 2;**

**AUTOSTART = TRUE { APPMODE = stdAppmode;};**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**};**

**TASK tau\_C {**

**PRIORITY = 1;**

**AUTOSTART = TRUE { APPMODE = stdAppmode;};**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**MESSAGE = message;**

**MESSAGE = object;**

**};**

**TASK initialization\_task {**

**PRIORITY = 4;**

**AUTOSTART = TRUE { APPMODE =stdAppmode;};**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**MESSAGE = message;**

**};**

**TASK stop {**

**PRIORITY = 99;**

**AUTOSTART = FALSE;**

**ACTIVATION = 1;**

**SCHEDULE = FULL;**

**};**

**};**

**FILE .C**

**#include <time.h>**

**#include <sys/time.h>**

**#include "tpl\_os.h"**

**#include <stdio.h>**

**long sf = 1;**

**/\* Keep the caller busy for howmany\_us microseconds \*/**

**void do\_other\_things(int howmany\_us)**

**{**

**long mul = howmany\_us \* sf;**

**long i;**

**for(i=0; i<mul; i++);**

**}**

**#define CAL\_US 1000000**

**#define CAL\_T 5**

**void calibrate(void)**

**{**

**struct timeval now;**

**long nsf = 0;**

**printf("Calibrating delay... \r\n");**

**while(nsf == 0 || nsf - sf > CAL\_T || sf - nsf > CAL\_T)**

**{**

**struct timeval old, new;**

**long l;**

**gettimeofday(&old, NULL);**

**do\_other\_things(CAL\_US);**

**gettimeofday(&new, NULL);**

**l = (long)((new.tv\_sec - old.tv\_sec) \* 1000000 + (new.tv\_usec - old.tv\_usec));**

**nsf = CAL\_US \* sf / l;**

**sf = (sf + nsf)/2;**

**printf("... l= %ld sf= %ld\r\n", l, sf);**

**}**

**}**

**DeclareAlarm(A);**

**DeclareAlarm(B);**

**DeclareAlarm(C);**

**DeclareAlarm(initialization\_alarm);**

**DeclareTask(tau\_A);**

**DeclareTask(tau\_B);**

**DeclareTask(tau\_C);**

**DeclareTask(initialization\_task);**

**DeclareMessage(message);**

**DeclareMessage(object);**

**unsigned int critical\_section;**

**/\***

**If 0 no tasks in critical section**

**If 1 task A is in its critical section**

**If 3 task C is in its critical section**

**\*/**

**int main(void)**

**{**

**calibrate();**

**StartOS(OSDEFAULTAPPMODE);**

**return 0;**

**}**

**TASK(initialization\_task)**

**{**

**static unsigned int critical\_section=0;**

**SendMessage(message, &critical\_section);**

**printf("tsk init... counter=%d\r\n\n",critical\_section);**

**TerminateTask();**

**}**

**TASK(tau\_A)**

**{**

**static unsigned int critical\_section;**

**static int C\_A = 200000; //200 msec = 200 000 us**

**long response;**

**long tic;**

**long dispatch;**

**struct timeval start;**

**struct timeval end;**

**printf("#Activation task A\r\n\n");**

**GetAlarm(A, &tic);**

**dispatch=1000-tic\*10;**

**ReceiveMessage(object, &critical\_section);**

**if(critical\_section==0){**

**critical\_section=1;**

**SendMessage(message, &critical\_section);**

**printf("COUNTER=%d Task A enters in the 'critical section'. \r\n\n",critical\_section);**

**gettimeofday(&start, NULL);**

**do\_other\_things(C\_A);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**critical\_section=0;**

**printf("COUNTER=%d Task A exits from critical section.",critical\_section);**

**printf(" Response time=%ld \r\n\n", response);**

**SendMessage(message, &critical\_section);**

**}**

**else {**

**while(critical\_section == 3) ReceiveMessage(object,&critical\_section);**

**if(critical\_section==0)**

**{**

**critical\_section=1;**

**SendMessage(message, &critical\_section);**

**printf("COUNTER=%d Task A enters in the 'critical section'. \r\n\n",critical\_section);**

**gettimeofday(&start, NULL);**

**do\_other\_things(C\_A);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**critical\_section=0;**

**printf("COUNTER=%d Task A exits from critical section.",critical\_section);**

**printf(" Response time=%ld \r\n\n", response);**

**SendMessage(message, &critical\_section);**

**}**

**}**

**TerminateTask();**

**}**

**TASK(tau\_B)**

**{**

**static int C\_B = 700000; //700 msec = 700 000 us**

**long response;**

**long tic;**

**long dispatch;**

**struct timeval start;**

**struct timeval end;**

**GetAlarm(B, &tic);**

**dispatch=1500-tic\*10;**

**gettimeofday(&start, NULL);**

**printf("#Activation task B\r\n\n");**

**do\_other\_things(C\_B);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**printf("Task B completed. Response time=%ld \r\n\n", response);**

**TerminateTask();**

**}**

**TASK(tau\_C)**

**{**

**static unsigned int critical\_section;**

**static int C\_C = 300000; //300 msec = 300 000 us**

**long response;**

**long tic;**

**long dispatch;**

**struct timeval start;**

**struct timeval end;**

**GetAlarm(C, &tic);**

**dispatch=2800-tic\*10;**

**printf("#Activation Task C\r\n\n");**

**gettimeofday(&start, NULL);**

**do\_other\_things(100000);**

**ReceiveMessage(object, &critical\_section);**

**if(critical\_section==0){**

**critical\_section=3;**

**SendMessage(message, &critical\_section);**

**printf("COUNTER=%d Task C enters in the 'critical section'. \r\n\n",critical\_section);**

**do\_other\_things(C\_C-100000);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**critical\_section=0;**

**printf("COUNTER=%d Task C exits from 'critical section'. ",critical\_section);**

**printf(" Response time=%ld \r\n\n", response);**

**SendMessage(message, &critical\_section);**

**}**

**else {**

**while(critical\_section == 1) ReceiveMessage(object,&critical\_section);**

**if(critical\_section==0)**

**{**

**critical\_section=3;**

**SendMessage(message, &critical\_section);**

**printf("COUNTER=%d Task C enters in the 'critical section'. \r\n\n",critical\_section);**

**do\_other\_things(C\_C-100000);**

**gettimeofday(&end, NULL);**

**response = (long)((end.tv\_sec - start.tv\_sec) \* 1000 + (end.tv\_usec - start.tv\_usec)/1000 + dispatch);**

**critical\_section=0;**

**printf("COUNTER=%d Task C exits from 'critical section'. ",critical\_section);**

**printf(" Response time=%ld \r\n\n", response);**

**SendMessage(message, &critical\_section);**

**}**

**}**

**TerminateTask();**

**}**

**TASK(stop)**

**{**

**CancelAlarm(A);**

**CancelAlarm(B);**

**CancelAlarm(C);**

**printf("Shutdown\r\n");**

**ShutdownOS(E\_OK);**

**TerminateTask();**

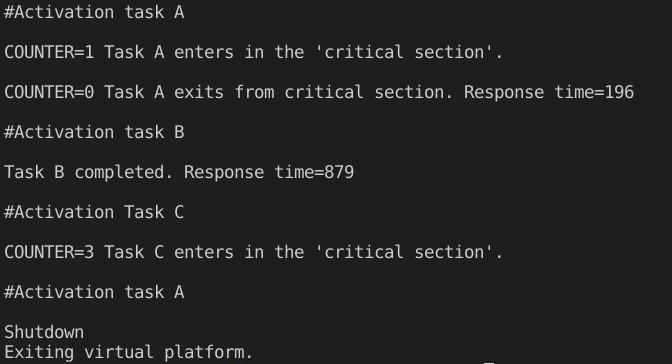
**}**

**RESULTS EXERCISE 3**

E’ possibile capire se l’idea di Mr. C.Lever sia fattibile o meno andando ad analizzare due casi fondamentali riguardanti la task A la quale ha la priorità maggiore.

In particolare se A si attiva:

1. dopo che C è entrata nella critical section, allora il counter sarà = 3, per cui avendo la preeemption il programma si blocca all’interno del ciclo while di A (perchè il counter non verrà più modificato);

****

Per completezza vengono riportati gli stati delle task a i vari istanti di tempo (in rosso sono segnati gli stati che evidenziano tale fenomeno di blocco.

[ 0] proc idle change to state READY\_AND\_NEW

[ 0] proc tau\_C change to state READY\_AND\_NEW

[ 0] proc tau\_B change to state READY\_AND\_NEW

[ 0] proc tau\_A change to state READY\_AND\_NEW

[ 0] proc initialization\_task change to state READY\_AND\_NEW

[ 0] proc initialization\_task change to state RUNNING

[ 0] proc initialization\_task change to state SUSPENDED

[ 0] proc tau\_A change to state RUNNING

[ 19] proc tau\_A change to state SUSPENDED

[ 19] proc tau\_B change to state RUNNING

[ 89] proc tau\_B change to state SUSPENDED

[ 89] proc tau\_C change to state RUNNING

[ 100] time object expired: A

[ 100] proc tau\_A change to state READY\_AND\_NEW

[ 100] proc tau\_C change to state READY

[ 100] proc tau\_A change to state RUNNING

[ 150] time object expired: B

[ 150] proc tau\_B change to state READY\_AND\_NEW

[ 200] time object expired: A

[ 280] time object expired: C

[ 300] time object expired: B

[ 300] time object expired: A

[ 400] time object expired: A

[ 450] time object expired: B

[ 500] time object expired: A

[ 560] time object expired: C

[ 600] time object expired: B

[ 600] time object expired: A

[ 700] time object expired: A

[ 750] time object expired: B

[ 800] time object expired: A

[ 840] time object expired: C

[ 900] time object expired: B

[ 900] time object expired: A

[ 1000] time object expired: A

[ 1050] time object expired: B

[ 1100] time object expired: A

[ 1120] time object expired: C

[ 1200] time object expired: B

[ 1200] time object expired: A

[ 1300] time object expired: A

[ 1350] time object expired: B

[ 1400] time object expired: C

[ 1400] time object expired: A

[ 1500] time object expired: B

[ 1500] time object expired: A

[ 1600] time object expired: A

[ 1650] time object expired: B

[ 1680] time object expired: C

[ 1700] time object expired: A

[ 1800] time object expired: B

[ 1800] time object expired: A

[ 1900] time object expired: A

[ 1950] time object expired: B

[ 1960] time object expired: C

[ 2000] time object expired: A

[ 2100] time object expired: B

[ 2100] time object expired: A

[ 2200] time object expired: A

[ 2240] time object expired: C

[ 2250] time object expired: B

[ 2300] time object expired: A

[ 2400] time object expired: B

[ 2400] time object expired: A

[ 2500] time object expired: A

[ 2520] time object expired: C

[ 2550] time object expired: B

[ 2600] time object expired: A

[ 2700] time object expired: B

[ 2700] time object expired: A

[ 2800] time object expired: C

[ 2800] time object expired: A

[ 2850] time object expired: B

[ 2900] time object expired: A

[ 3000] time object expired: B

[ 3000] time object expired: A

[ 3080] time object expired: C

[ 3100] time object expired: A

[ 3150] time object expired: B

[ 3200] time object expired: A

[ 3300] time object expired: B

[ 3300] time object expired: A

[ 3360] time object expired: C

[ 3400] time object expired: A

[ 3450] time object expired: B

[ 3500] time object expired: A

[ 3600] time object expired: B

[ 3600] time object expired: A

[ 3640] time object expired: C

[ 3700] time object expired: A

[ 3750] time object expired: B

[ 3800] time object expired: A

[ 3900] time object expired: B

[ 3900] time object expired: A

[ 3920] time object expired: C

[ 4000] time object expired: A

[ 4050] time object expired: B

[ 4100] time object expired: A

[ 4200] time object expired: stopper

[ 4200] proc stop change to state READY\_AND\_NEW

[ 4200] time object " stopper" change to state SUSPENDED

[ 4200] time object expired: C

[ 4200] time object expired: B

[ 4200] time object expired: A

[ 4200] proc tau\_A change to state READY

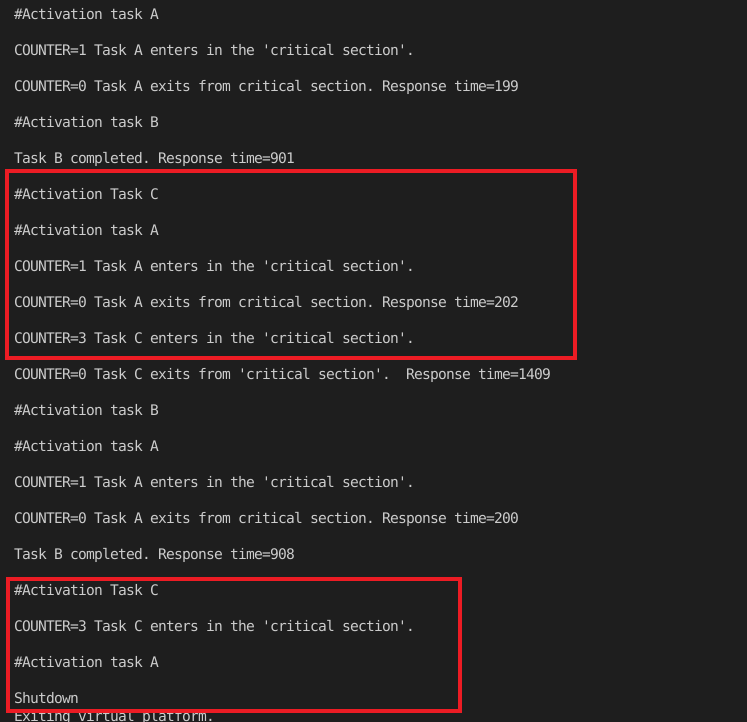
[ 4200] proc stop change to state RUNNING

[ 4200] time object " A" change to state SUSPENDED

[ 4200] time object " B" change to state SUSPENDED

[ 4200] time object " C" change to state SUSPENDED

1. prima che C entri nella critical section, allora il counter è ancora = 0, per cui si ha preemption ma il programma continua;



notiamo però che alla seconda attivazione di C si ha il problema descritto nel caso precedente.

[ 0] proc idle change to state READY\_AND\_NEW

[ 0] proc tau\_C change to state READY\_AND\_NEW

[ 0] proc tau\_B change to state READY\_AND\_NEW

[ 0] proc tau\_A change to state READY\_AND\_NEW

[ 0] proc initialization\_task change to state READY\_AND\_NEW

[ 0] proc initialization\_task change to state RUNNING

[ 0] proc initialization\_task change to state SUSPENDED

[ 0] proc tau\_A change to state RUNNING

[ 19] proc tau\_A change to state SUSPENDED

[ 19] proc tau\_B change to state RUNNING

[ 90] proc tau\_B change to state SUSPENDED

[ 90] proc tau\_C change to state RUNNING

[ 100] time object expired: A

[ 100] proc tau\_A change to state READY\_AND\_NEW

[ 100] proc tau\_C change to state READY

[ 100] proc tau\_A change to state RUNNING

[ 120] proc tau\_A change to state SUSPENDED

[ 120] proc tau\_C change to state RUNNING

[ 140] proc tau\_C change to state SUSPENDED

[ 140] proc idle change to state RUNNING

[ 150] time object expired: B

[ 150] proc tau\_B change to state READY\_AND\_NEW

[ 150] proc idle change to state READY

[ 150] proc tau\_B change to state RUNNING

[ 200] time object expired: A

[ 200] proc tau\_A change to state READY\_AND\_NEW

[ 200] proc tau\_B change to state READY

[ 200] proc tau\_A change to state RUNNING

[ 219] proc tau\_A change to state SUSPENDED

[ 219] proc tau\_B change to state RUNNING

[ 240] proc tau\_B change to state SUSPENDED

[ 240] proc idle change to state RUNNING

[ 280] time object expired: C

[ 280] proc tau\_C change to state READY\_AND\_NEW

[ 280] proc idle change to state READY

[ 280] proc tau\_C change to state RUNNING

[ 300] time object expired: B

[ 300] proc tau\_B change to state READY\_AND\_NEW

[ 300] time object expired: A

[ 300] proc tau\_A change to state READY\_AND\_NEW

[ 300] proc tau\_C change to state READY

[ 300] proc tau\_A change to state RUNNING

[ 400] time object expired: A

[ 450] time object expired: B

[ 500] time object expired: A

[ 560] time object expired: C

[ 600] time object expired: B

[ 600] time object expired: A

[ 700] time object expired: A

[ 750] time object expired: B

[ 800] time object expired: A

[ 840] time object expired: C

[ 900] time object expired: B

[ 900] time object expired: A

[ 1000] time object expired: A

[ 1050] time object expired: B

[ 1100] time object expired: A

[ 1120] time object expired: C

[ 1200] time object expired: B

[ 1200] time object expired: A

[ 1300] time object expired: A

[ 1350] time object expired: B

[ 1400] time object expired: C

[ 1400] time object expired: A

[ 1500] time object expired: B

[ 1500] time object expired: A

[ 1600] time object expired: A

[ 1650] time object expired: B

[ 1680] time object expired: C

[ 1700] time object expired: A

[ 1800] time object expired: B

[ 1800] time object expired: A

[ 1900] time object expired: A

[ 1950] time object expired: B

[ 1960] time object expired: C

[ 2000] time object expired: A

[ 2100] time object expired: B

[ 2100] time object expired: A

[ 2200] time object expired: A

[ 2240] time object expired: C

[ 2250] time object expired: B

[ 2300] time object expired: A

[ 2400] time object expired: B

[ 2400] time object expired: A

[ 2500] time object expired: A

[ 2520] time object expired: C

[ 2550] time object expired: B

[ 2600] time object expired: A

[ 2700] time object expired: B

[ 2700] time object expired: A

[ 2800] time object expired: C

[ 2800] time object expired: A

[ 2850] time object expired: B

[ 2900] time object expired: A

[ 3000] time object expired: B

[ 3000] time object expired: A

[ 3080] time object expired: C

[ 3100] time object expired: A

[ 3150] time object expired: B

[ 3200] time object expired: A

[ 3300] time object expired: B

[ 3300] time object expired: A

[ 3360] time object expired: C

[ 3400] time object expired: A

[ 3450] time object expired: B

[ 3500] time object expired: A

[ 3600] time object expired: B

[ 3600] time object expired: A

[ 3640] time object expired: C

[ 3700] time object expired: A

[ 3750] time object expired: B

[ 3800] time object expired: A

[ 3900] time object expired: B

[ 3900] time object expired: A

[ 3920] time object expired: C

[ 4000] time object expired: A

[ 4050] time object expired: B

[ 4100] time object expired: A

[ 4200] time object expired: stopper

[ 4200] proc stop change to state READY\_AND\_NEW

[ 4200] time object " stopper" change to state SUSPENDED

[ 4200] time object expired: C

[ 4200] time object expired: B

[ 4200] time object expired: A

[ 4200] proc tau\_A change to state READY

[ 4200] proc stop change to state RUNNING

[ 4200] time object " A" change to state SUSPENDED

[ 4200] time object " B" change to state SUSPENDED

[ 4200] time object " C" change to state SUSPENDED

Tali problemi si possono evidenziare immediatamente alla prima e alla seconda attivazione. In particolare alla seconda si avrà sempre il blocco dell’applicazione causata dalla preemption della task C quando questa è nella sua “critical region” e quindi il valore del counter è già stato modificato. In tal modo quindi A entrerà in un ciclo senza fine in cui non è possibile uscire perchè la critical section di C non è stata completata.