

UNIVERSITAT POLITÈCNICA DE CATALUNYA

SISTEMES ENCASTATS I UBICS

Entregues laboratoris

Laboratori 5 - RTOS

Carlota Catot
Miguel Antunez

Quatrimestre primavera 2020-2021

Índice

1. Pregunta 0	2
1.1. Codi	2
2. Pregunta 1	3
2.1. Codi	3
2.2. Intervals de temps	5
2.2.1. Gràfiques prioritat normal	5
2.2.2. Gràfiques prioritat alta	7
2.3. Comportament paral·lelisme threads	7
2.4. Cost/Overhead	8
3. Pregunta 2	9
3.1. Codi que provoca una inversió de prioritat	9
3.2. Solució	10
3.2.1. Codi	10
4. Pregunta 3	11

1. Pregunta 0

Un senzill programa que realitzi en paral·lel una intermitència de 0.3 segons del LED 1 i una alternança entre el LED 2 i el LED 3 d'un segon

1.1. Codi

```
#include "mbed.h"
#include "rtos.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);
Thread thread;

void led2_thread() {

    led2 = !led2;
    while (true) {
        led2=!led2;
        led3=!led3;
        Thread::wait(1000);
    }
}

int main() {
    thread.start(led2_thread);
    while (true) {
        led1 = !led1;
        Thread::wait(300);
    }
}
```

Per tal de resoldre el primer problema s'han definit els 3 leds. Al inici de la funció main s'ha generat un thread per controlar els leds 2 i 3 de manera que a l'inici d'aquesta funció es posa el led 2 a 1 per tal de que els dos leds es comportin de manera oposada, i amb la funció wait del thread fem que canviï el seu estat cada segon. Mentrestant a la funció principal el led 1 canviarà d'estat cada 0.3 segons aplicant el mateix mètode.

2. Pregunta 1

Implementeu un programa de dos threads productors i dos consumidors, gestionat mitjançant una cua de missatges (o mail). Un dels productors generarà un nombres parells i l'altre d'imparells. Els threads consumidors senzillament imprimiran el número obtingut de la cua per la línia sèrie

2.1. Codi

```
#include "mbed.h"

typedef struct {
    int value;
} message_t;

Mutex mutex;
const int queue_size = 10;
ConditionVariable cond(mutex);
MemoryPool<message_t, queue_size> mpool;
Queue<message_t, queue_size> queue;

Thread thread_producer_even;
Thread thread_producer_odd;
Thread thread_consumer_1;
Thread thread_consumer_2;

Serial pc(USBTX, USBRX, 115200);

int pointer = 0;
int countPar = 1;
int countImpar = 1;

void producer_even(){
    while (true) {
        mutex.lock();
        int par = countPar%2;

        if(par==0){
            message_t *message = mpool.alloc();
            message->value = countPar;
            queue.put(message);
            printf("\nProducer even value %d \n", message->value);
            pointer++;
        }

        countPar++;

        while (pointer == queue_size){
            printf("\nThe queue is full, wait a consumer\n");
            cond.wait();
        }
        cond.notify_all();
        mutex.unlock();
    }
}
```

```

void producer_odd(){
    while (true) {
        mutex.lock();
        int impar = countImpar%2;

        if(impar!=0){
            message_t *message = mpool.alloc();
            message->value = countImpar;
            queue.put(message);
            printf("\nProducer odd value %d \n", message->value);
            pointer++;
        }

        countImpar++;

        while (pointer == queue_size){
            printf("\nThe queue is full, wait a consumer\n");
            cond.wait();
        }
        cond.notify_all();
        mutex.unlock();
    }
}

void consumer_1(){
    while (true) {
        mutex.lock();

        while (pointer == 0){
            printf("\nThe queue is empty, wait for a producer.\n");
            cond.wait();
        }

        osEvent evt = queue.get();
        pointer--;

        if (evt.status == osEventMessage) {
            message_t *message = (message_t*)evt.value.p;
            printf("\nConsumer_1: Got value %.d .\n", message->value);
            mpool.free(message);
        }

        cond.notify_all();
        mutex.unlock();
    }
}

void consumer_2(){

    while (true) {
        mutex.lock();
        while (pointer == 0){
            printf("\nThe queue is empty, wait for a producer.\n");
            cond.wait();
        }
    }
}

```

```

osEvent evt = queue.get();
pointer--;

if (evt.status == osEventMessage) {
    message_t *message = (message_t*)evt.value.p;
    printf("\nConsumer_2: Got value %.d .\n" , message->value);
    mpool.free(message);
}

cond.notify_all();
mutex.unlock();
}

}

int main () {
    thread_producer_even.start(&producer_even);
    thread_producer_odd.start(&producer_odd);
    thread_consumer_1.start(&consumer_1);
    thread_consumer_2.start(&consumer_2);
}

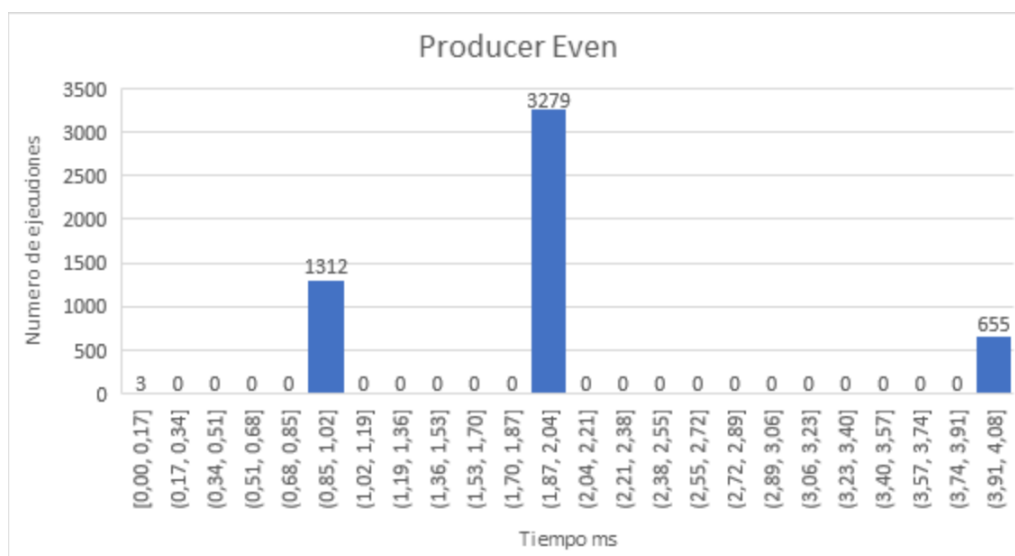
```

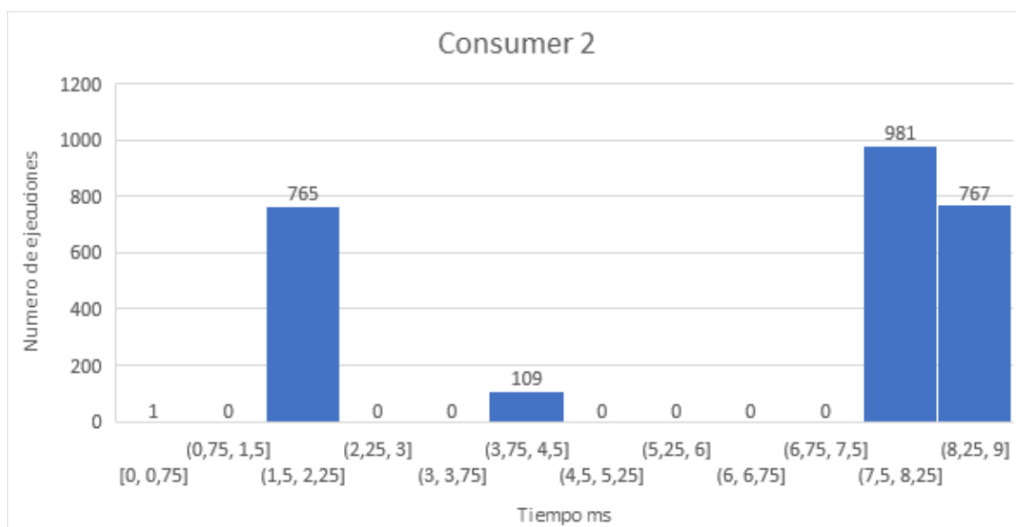
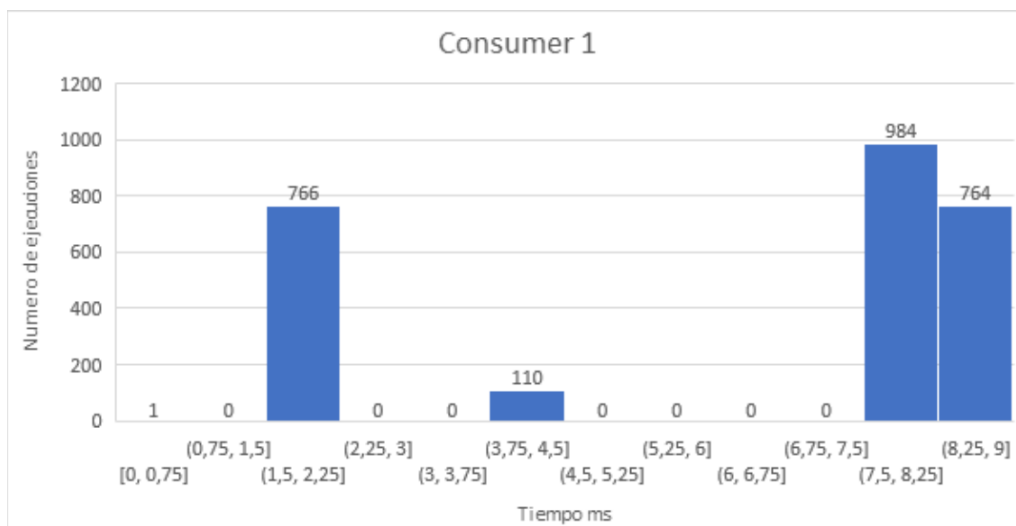
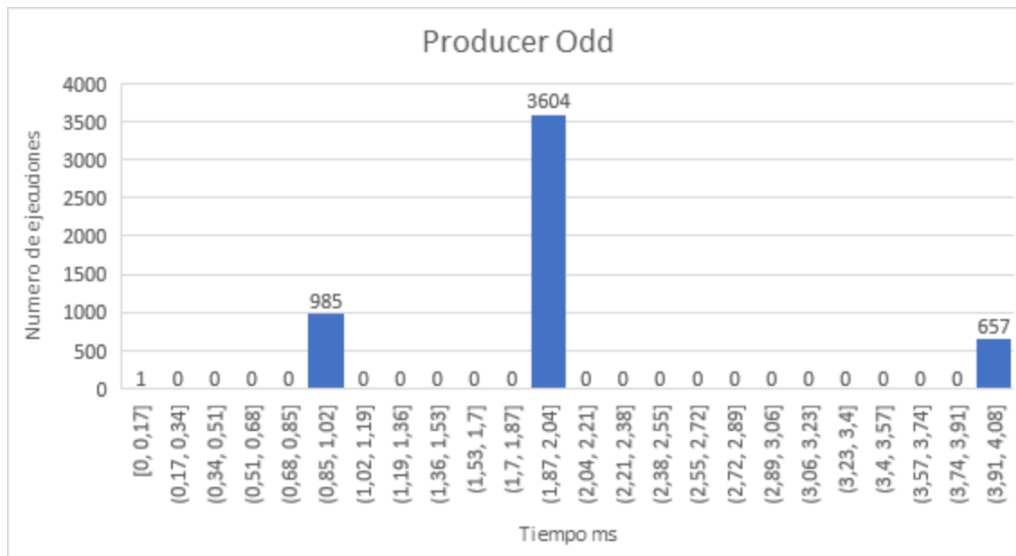
2.2. Intervals de temps

Mesureu l'interval de temps (time slot) que dedica el RTOS per executar cada thread amb prioritat normal i en alta prioritat

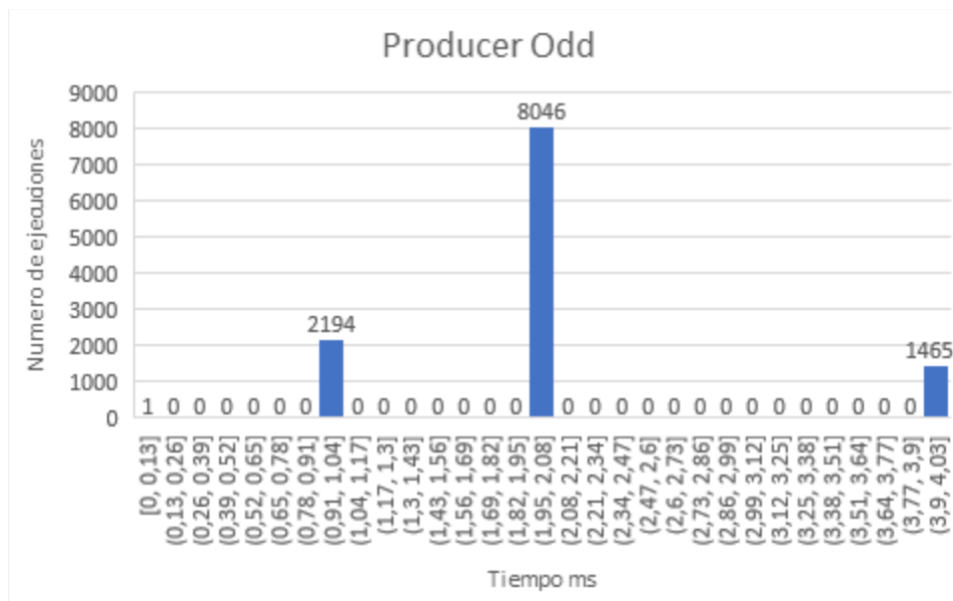
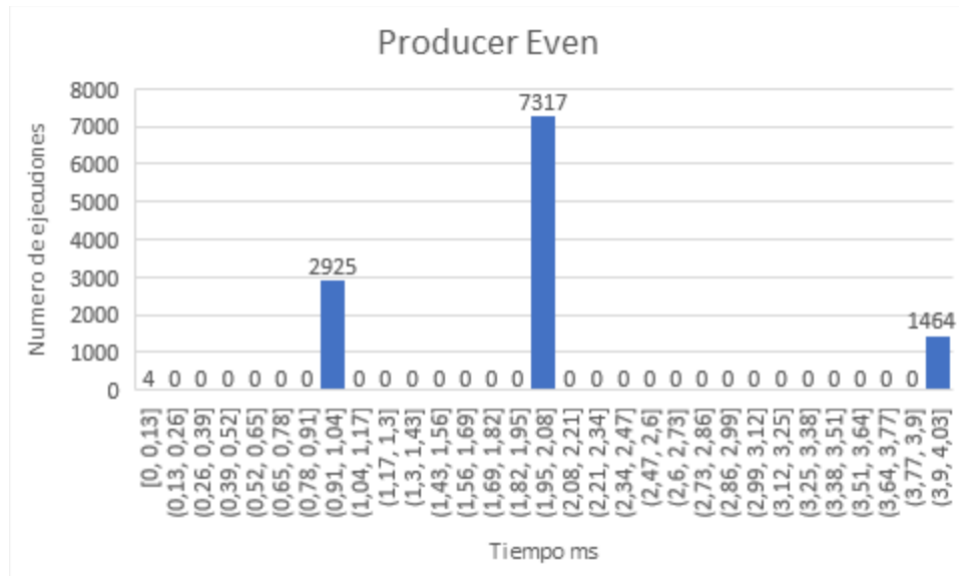
Per tal de mesurar el temps hem executat molts cops tots els threads i hem obtingut les següents gràfiques de temps utilitzant la classe Timer, podem observar que els temps varien, però els threads que executen codi molt similar o igual, repeteixen el patró de temps, quan s'assigna major prioritat a un thread com és el cas del *Producer Even* s'incrementa el nombre d'execucions en el menor temps del time slot.

2.2.1. Gràfiques prioritat normal





2.2.2. Gràfiques prioritat alta



2.3. Comportament paral·lelisme threads

Comproveu si la funció printf s'executa correctament quan es cridada des de diversos threads d'igual prioritat. Quin comportament s'observa?

En utilitzar mutex.lock() i mutex.unlock() no s'observa cap comportament estrany en la funció, les línies s'imprimeixen a la consola de manera normal.

2.4. Cost/Overhead

Mesureu el temps que triga el RTOS per canviar de thread (scheduling cost/overhead).

Per tal de mesurar el temps hem generat una gràfica per poder-ho veure de manera més visual:



3. Pregunta 2

Implementeu un codi que provoqui intencionadament un bloqueig per inversió de prioritat (priority inversion) entre tres threads de prioritat alta, normal i baixa. Implementeu una solució per resoldre aquesta situació. Per canviar la prioritat del programa principal (main) podeu utilitzar la funció *osThreadSetPriority(osThreadGetId(), osPriorityLow)*;

Una inversió de prioritat no limitada es produeix quan un procés de prioritat immediata endarrerix l'execució d'un altre de prioritat superior perquè aquest últim està bloquejat a l'espera d'un recurs controlat per un tercer procés de baixa prioritat.

3.1. Codi que provoca una inversió de prioritat

```
#include "mbed.h"
#include "rtos.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);

Thread threadNormal;
Thread threadHigh;

Mutex mutex;

void led(DigitalOut led, uint32_t t) {
    mutex.lock();
    Thread::wait(t);
    led=!led;
    mutex.unlock();
}

void led2_thread() {
    while (true) {
        led(led2,10000);
    }
}

void led3_thread() {
    while (true) {
        led(led3,1000);
    }
}

int main() {
    threadHigh.set_priority(osPriorityHigh);
    threadHigh.start(led3_thread);
    threadNormal.start(led2_thread);

    osThreadSetPriority(osThreadGetId(), osPriorityLow);
    while (true) {
        led(led1,10);
    }
}
```

3.2. Solució

Herència de prioritat: La solució denominada herència de prioritat proposa eliminar la inversió de prioritats elevant la prioritat d'un procés amb un lock en un recurs compartit al màxim de les prioritats dels processos que estiguin esperant el recurs. La idea d'aquest protocol consisteix que quan un procés bloqueja indirectament a processos de més alta prioritat, la prioritat original és ignorada i executa la secció crítica corresponent amb la prioritat més alta dels processos que està bloquejant.

3.2.1. Codi

```
#include "mbed.h"
#include "rtos.h"

DigitalOut led1(LED1);
DigitalOut led2(LED2);
DigitalOut led3(LED3);

Thread threadNormal;
Thread threadHigh;

Mutex mutex;

void led(DigitalOut led, uint32_t t) {
    mutex.lock();
    Thread::wait(t);
    led=!led;
    mutex.unlock();
}

void led2_thread() {
    while (true) {
        led(led2,10000);
    }
}

void led3_thread() {
    while (true) {
        led(led3,1000);
    }
}

int main() {
    threadHigh.set_priority(osPriorityHigh);
    threadHigh.start(led3_thread);
    threadNormal.start(led2_thread);

    osThreadSetPriority(osThreadGetId(), osPriorityLow);

    while (true) {
        osThreadSetPriority(osThreadGetId(), osPriorityHigh);
        led(led1,10);
        osThreadSetPriority(osThreadGetId(), osPriorityLow);
    }
}
```

4. Pregunta 3

Planifiqueu i executeu les tasques següents definides per (I = Init time, C= Cost, D = Deadline) pels casos següents:

```
#include "mbed.h"
#include "mbed_events.h"

Timer timer;
Timer t1;
Timer t2;
Timer t3;

EventQueue queue;
Serial pc(USBTX, USBRX, 115200);

void task1(int c, int d){
    t1.reset();
    t1.start();
    printf("task1: DeadLine %d s\r\n", d);
    wait_ms(c);
    float tex = t1.read_ms()/1000;

    if(tex > d){
        printf("task1: No se cumple el DeadLine %f s\r\n", tex);
    } else {
        printf("task1: Se cumple el DeadLine - Coste: %f s\r\n", tex);
    }

    t1.reset();
}

void task2(int c, int d){
    t2.reset();
    t2.start();
    printf("task2: DeadLine %d s\r\n", d);
    wait_ms(c);
    float tex = t2.read_ms()/1000;

    if(tex > d){
        printf("task2: No se cumple el DeadLine %f s\r\n", tex);
    } else {
        printf("task2: Se cumple el DeadLine - Coste: %f s\r\n", tex);
    }

    t2.reset();
}

void task3(int c, int d){
    t3.reset();
    t3.start();
    printf("task3: DeadLine %d s\r\n", d);
    wait_ms(c);
    float tex = t3.read_ms()/1000;
```

```

        if(tex > d){
            printf("task3: No se cumple el DeadLine %f s\r\n", tex);
        } else {
            printf("task3: Se cumple el DeadLine - Coste: %f s\r\n", tex);
        }

        t3.reset();
    }

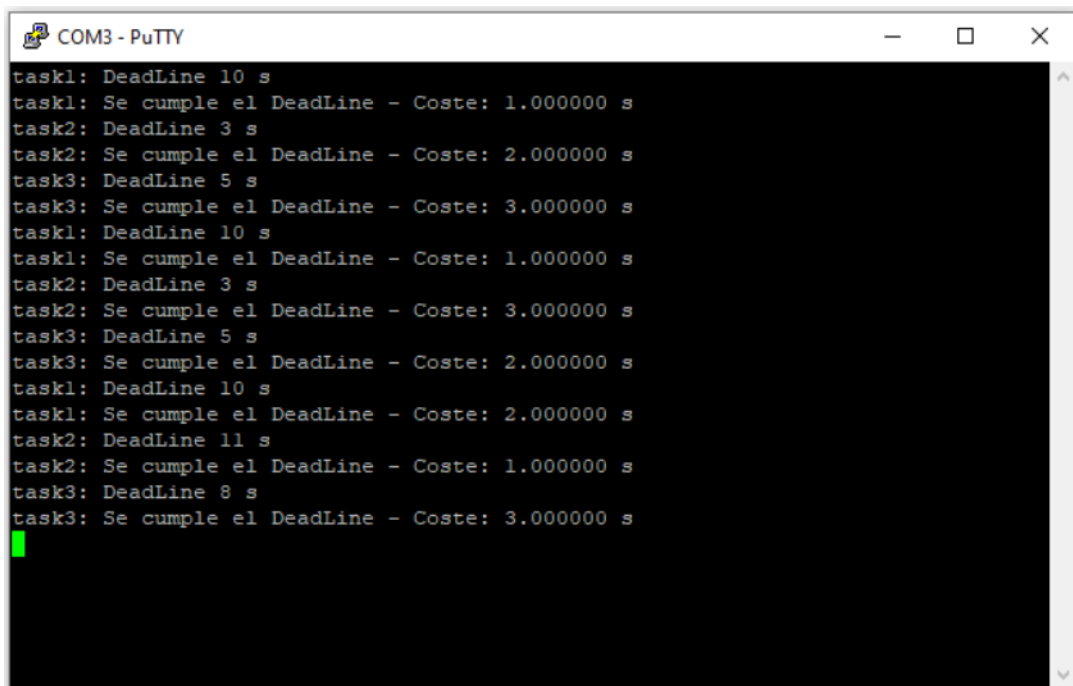
int main() {
    queue.call(task1, 100, 10);
    queue.call(task2, 300, 3);
    queue.call(task3, 400, 5);

    queue.call(task1, 200, 10);
    queue.call(task2, 400, 3);
    queue.call(task3, 300, 5);

    queue.call(task1, 300, 10);
    queue.call(task2, 200, 11);
    queue.call_in(1000, task3, 400, 8);

    queue.dispatch();
}

```



```

COM3 - PuTTY
task1: DeadLine 10 s
task1: Se cumple el DeadLine - Coste: 1.000000 s
task2: DeadLine 3 s
task2: Se cumple el DeadLine - Coste: 2.000000 s
task3: DeadLine 5 s
task3: Se cumple el DeadLine - Coste: 3.000000 s
task1: DeadLine 10 s
task1: Se cumple el DeadLine - Coste: 1.000000 s
task2: DeadLine 3 s
task2: Se cumple el DeadLine - Coste: 3.000000 s
task3: DeadLine 5 s
task3: Se cumple el DeadLine - Coste: 2.000000 s
task1: DeadLine 10 s
task1: Se cumple el DeadLine - Coste: 2.000000 s
task2: DeadLine 11 s
task2: Se cumple el DeadLine - Coste: 1.000000 s
task3: DeadLine 8 s
task3: Se cumple el DeadLine - Coste: 3.000000 s

```