Compte-rendu TP "Linux Xenomai - Partie 2"

Mewen Michel et Sander Ricou - MI11 UTC

Pathfinder

Question 1 *Expliquez le principe des fonctions `create_and_start_rt_task`, `rt_task` ainsi que de la structure `task_descriptor`.*

- `create_and_start_rt_task` :
    - création d'une tâche temps réel
    - passage de la tâche simple en tâche périodique
    - lancement de la tâche
- `rt_task` : travail d'une tâche
    - attente de la prochaine période,
    - occupation de la ressource éventuelle,
    - calcul,
    - libération de la ressource
- `task_descriptor` :

```
typedef struct task_descriptor{
    RT_TASK task;                   // tâche temps réel
    void (*task_function)(void*);   // pointeur sur la fonction de la tâche à
exécuter
    RTIME period;                   // période d'exécution
    RTIME duration;                 // temps de calcul à chaque période
    int priority;                   // priorité de la tâche
    bool use_resource;              // défini si la tâche a besoin d'une
ressource
} task_descriptor;
```

Question 2 *Expliquez le principe de la fonction `rt_task_name`. Quelles autres informations sont stockées dans la structure `RT_TASK_INFO`?*

La fonction `rt_task_name` récupère la tâche courante (par `rt_task_inquire(NULL,&info);`) afin de retourner son nom.

La structure `RT_TASK_INFO` contient :

```
typedef struct rt_task_info {
    int         bprio           // Base priority.
    int         cprio           // Current priority.
    unsigned    status          // Task's status.
    RTIME       relpoint        // Time of next release.
    char        name [XNOBJECT_NAME_LEN]    // Symbolic name assigned at
creation.
    RTIME       exectime        // Execution time in primary mode in
nanoseconds.
    int         modeswitches    // Number of primary->secondary mode switches.
    int         ctxswitches     // Number of context switches.
    int         pagefaults      // Number of triggered page faults.
} RT_TASK_INFO;
```

Question 3 *Décrivez comment vous avez réalisé la fonction `busy_wait`.*

```
void busy_wait(RTIME time) {
    static RT_TASK_INFO info;
    rt_task_inquire(NULL,&info);

    // calcule le temps d'exécution de la fonction auquel on arrêtera
d'attendre
    RTIME time_to_end = info.exectime + time;

    do {
        rt_task_inquire(NULL,&info);        // récupère les informations
actualisées de la tâche
    } while (time_to_end > info.exectime);  // tant qu'on est avant la fin de
l'attente, on boucle
}
```

Question 4 *Quel est le résultat ? Le timming est-il correct ?*

Notre fonction `rt_task` instrumentée avec `time_since_start()` est :

```
void rt_task(void *cookie) {
    struct task_descriptor* params=(struct task_descriptor*)cookie;

    rt_printf("started task %s, period %ims, duration %ims, use resource
%i\n",rt_task_name(),(int)(params->period/1000000),(int)(params-
>duration/1000000),params->use_resource);
    unsigned long brouette;

    while(1) {
        rt_task_wait_period(&brouette);
        if(params->use_resource) acquire_resource();

        rt_printf("Period wait %d\n", time_since_start());
        busy_wait(params->duration);
        rt_printf("Wait done %d\n", time_since_start());

        if(params->use_resource) release_resource();
    }
}
```

Cela nous donne le résultat suivant :

```
./pathfinder
started task ORDO_BUS, period 125ms, duration 25ms, use resource 0
Period done 125
Wait done 150
Period done 250
Wait done 275
Period done 375
Wait done 400
Period done 500
Wait done 525
...
```

Ce qui semble correct !

Question 5 *Expliquez comment le sémaphore doit être initialisé. Commentez l'enchaînement des tâches pour les cas extrêmes du temps d'exécution de METEO.*

Afin de gérer l'accès à la ressource du bus 1553, il faut initialiser le sémaphore à 1 : il ne peut y avoir qu'une tâche à y accéder à chaque instant.

Selon que la tâche METEO prenne 40 ms ou 60 ms à s'exécuter, nous obtenons le résultat suivant :

- 40ms :

```
...
doing METEO : 5226
doing ORDO_BUS : 5250
doing ORDO_BUS ok : 5275
doing RADIO : 5275
doing RADIO ok : 5300
doing CAMERA : 5300
doing CAMERA ok : 5325
doing METEO ok : 5325
doing DISTRIB_DONNEES : 5325
doing DISTRIB_DONNEES ok : 5350
doing PILOTAGE : 5350
doing ORDO_BUS : 5375
doing ORDO_BUS ok : 5400
doing PILOTAGE ok : 5400
doing DISTRIB_DONNEES : 5401
doing DISTRIB_DONNEES ok : 5426
doing ORDO_BUS : 5500
doing ORDO_BUS ok : 5525
...
```

- 60ms :

```
...
doing METEO : 5226
doing ORDO_BUS : 5250
doing ORDO_BUS ok : 5275
doing RADIO : 5275
doing RADIO ok : 5300
doing CAMERA : 5300
doing CAMERA ok : 5325
doing METEO ok : 5361
doing DISTRIB_DONNEES : 5361
doing ORDO_BUS : 5375
doing ORDO_BUS ok : 5400
doing DISTRIB_DONNEES ok : 5411
doing PILOTAGE : 5411
doing PILOTAGE ok : 5436
doing DISTRIB_DONNEES : 5436
doing DISTRIB_DONNEES ok : 5461
doing ORDO_BUS : 5500
doing ORDO_BUS ok : 5525
doing DISTRIB_DONNEES : 5525
doing DISTRIB_DONNEES ok : 5550
doing PILOTAGE : 5550
doing PILOTAGE ok : 5575
...
```

Question 6 *Expliquez le fonctionnement de la solution retenue.*

On utilise un second sémaphore pour observer si la tâche DISTRIB_DONNEES est en cours d'exécution. Si c'est le cas, ORDO_BUS va observer qu'il ne reste plus de jeton pour ce sémaphore :

```
RT_SEM_INFO sem_info;

rt_sem_inquire(&distrib_sem, &sem_info);
if(sem_info.count == 0){
    // DISTRIB_DONNEES est en cours de fonctionnement
}
```

Question 7 *Testez votre programme pour les cas extrêmes du temps d'exécution de METEO. Qu'observez vous? Expliquez ce phénomène à l'aide de chronogrammes.*

- Avec 40ms :

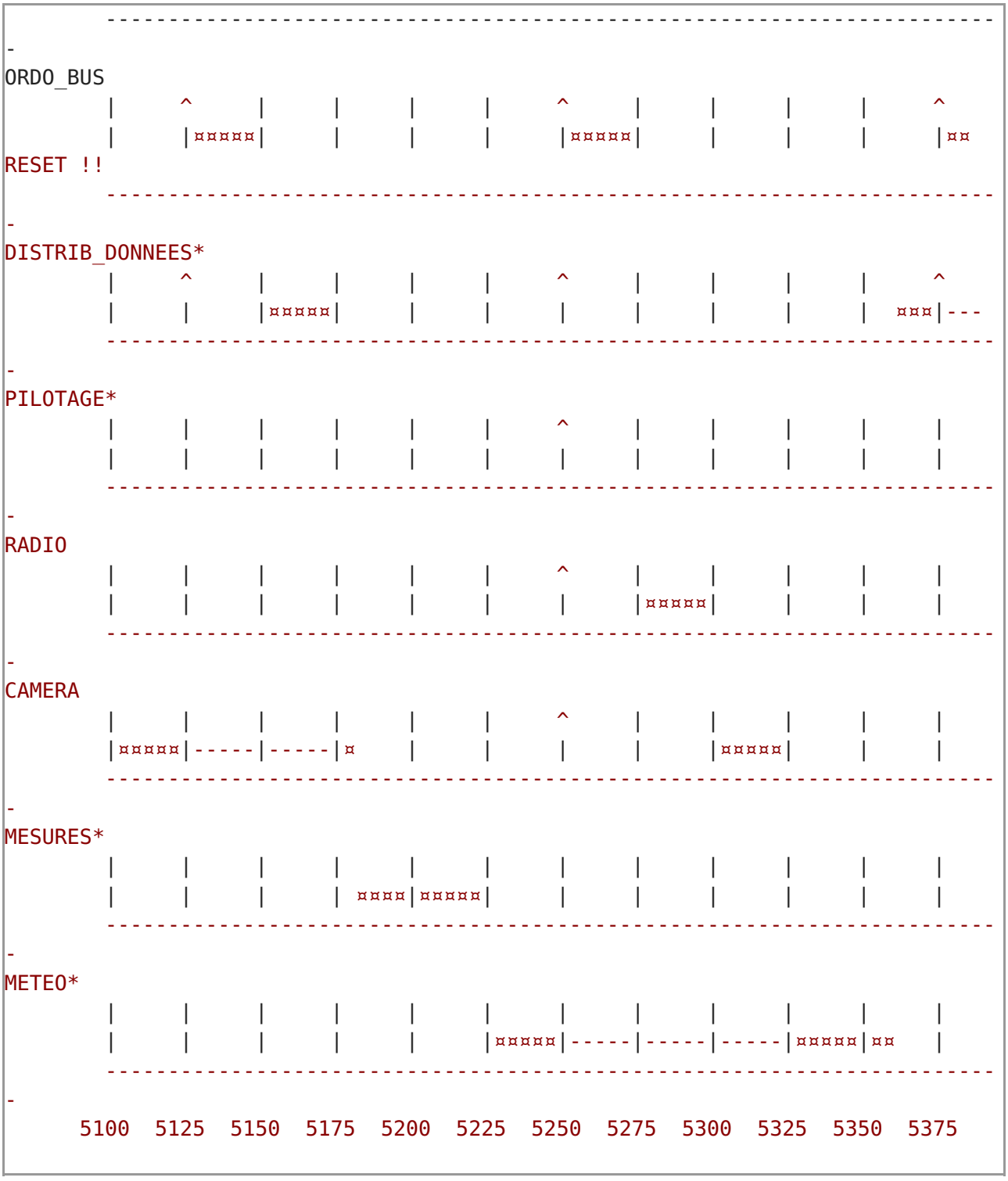
```
doing ORDO_BUS : 5000
doing ORDO_BUS ok : 5025
doing DISTRIB_DONNEES : 5025
doing DISTRIB_DONNEES ok : 5050
doing PILOTAGE : 5050
doing PILOTAGE ok : 5075
doing RADIO : 5076
doing RADIO ok : 5101
doing CAMERA : 5101
doing ORDO_BUS : 5125
doing ORDO_BUS ok : 5150
doing DISTRIB_DONNEES : 5150
doing DISTRIB_DONNEES ok : 5175
doing CAMERA ok : 5176
doing MESURES : 5176
doing MESURES ok : 5226
doing METEO : 5226
doing ORDO_BUS : 5250
doing ORDO_BUS ok : 5275
...
```

- 60ms :

```
doing ORDO_BUS : 5001
doing ORDO_BUS ok : 5026
doing DISTRIB_DONNEES : 5026
doing DISTRIB_DONNEES ok : 5051
doing PILOTAGE : 5051
doing PILOTAGE ok : 5076
doing RADIO : 5076
doing RADIO ok : 5101
doing CAMERA : 5101
doing ORDO_BUS : 5125
doing ORDO_BUS ok : 5151
doing DISTRIB_DONNEES : 5151
doing DISTRIB_DONNEES ok : 5176
doing CAMERA ok : 5176
doing MESURES : 5176
doing MESURES ok : 5226
doing METEO : 5226
doing ORDO_BUS : 5250
doing ORDO_BUS ok : 5276
doing RADIO : 5276
doing RADIO ok : 5301
doing CAMERA : 5301
doing CAMERA ok : 5326
doing METEO ok : 5361
doing DISTRIB_DONNEES : 5361
doing ORDO_BUS : 5375
reset
doing DISTRIB_DONNEES ok : 5386
doing PILOTAGE : 5386
doing PILOTAGE ok : 5411
doing DISTRIB_DONNEES : 5411
doing DISTRIB_DONNEES ok : 5436
doing DISTRIB_DONNEES : 5501
doing DISTRIB_DONNEES ok : 5526
doing PILOTAGE : 5526
doing PILOTAGE ok : 5551
doing RADIO : 5551
doing RADIO ok : 5576
doing CAMERA : 5576
doing CAMERA ok : 5601
```

On voit que `METEO` (de priorité la plus basse) retarde `DISTRIBDONNEES` (de priorité supérieure). On a donc un cas d'**inversion de priorité** due à l'accès concurent au bus 1553, notre ressource.

Voici le chronogramme schématisant ce problème :



Question 8 *Quelle solution proposez vous pour résoudre le problème ?*

Pour résoudre l'inversion de priorité, il faut élever temporairement la priorité de METEO au niveau de DISTRIBDONNEES. Il faut donc utiliser un mécanisme de synchronisation qui fait de l'héritage de priorité : le mutex. Celui-ci est utile pour les ressources partagées car il gère les priorités. Le sémaphore, lui, sert plutôt pour la synchronisation des tâches.

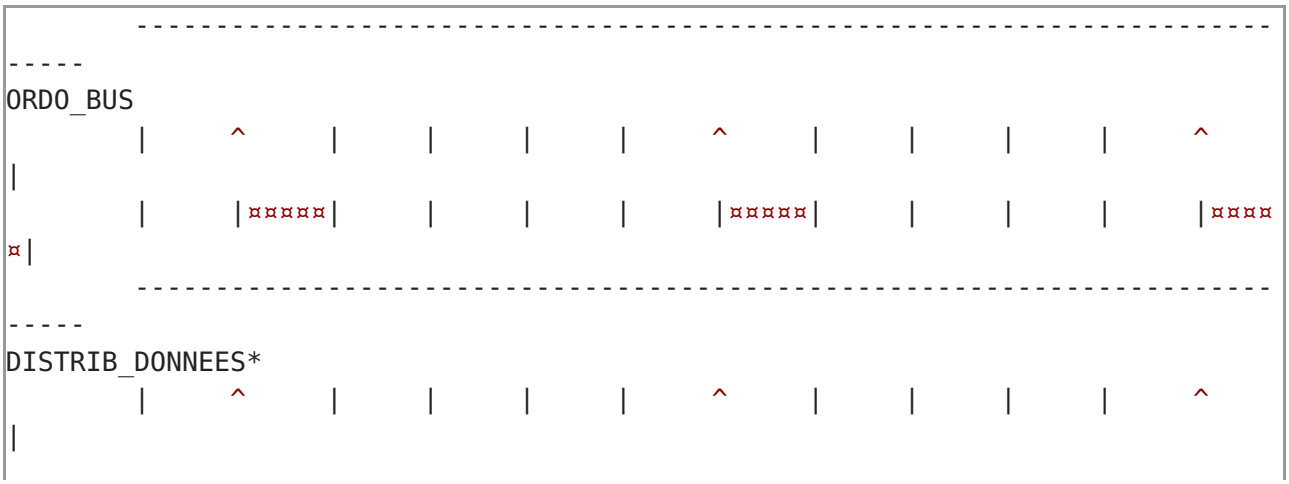
Question 9 *Testez et commentez le résultat.*

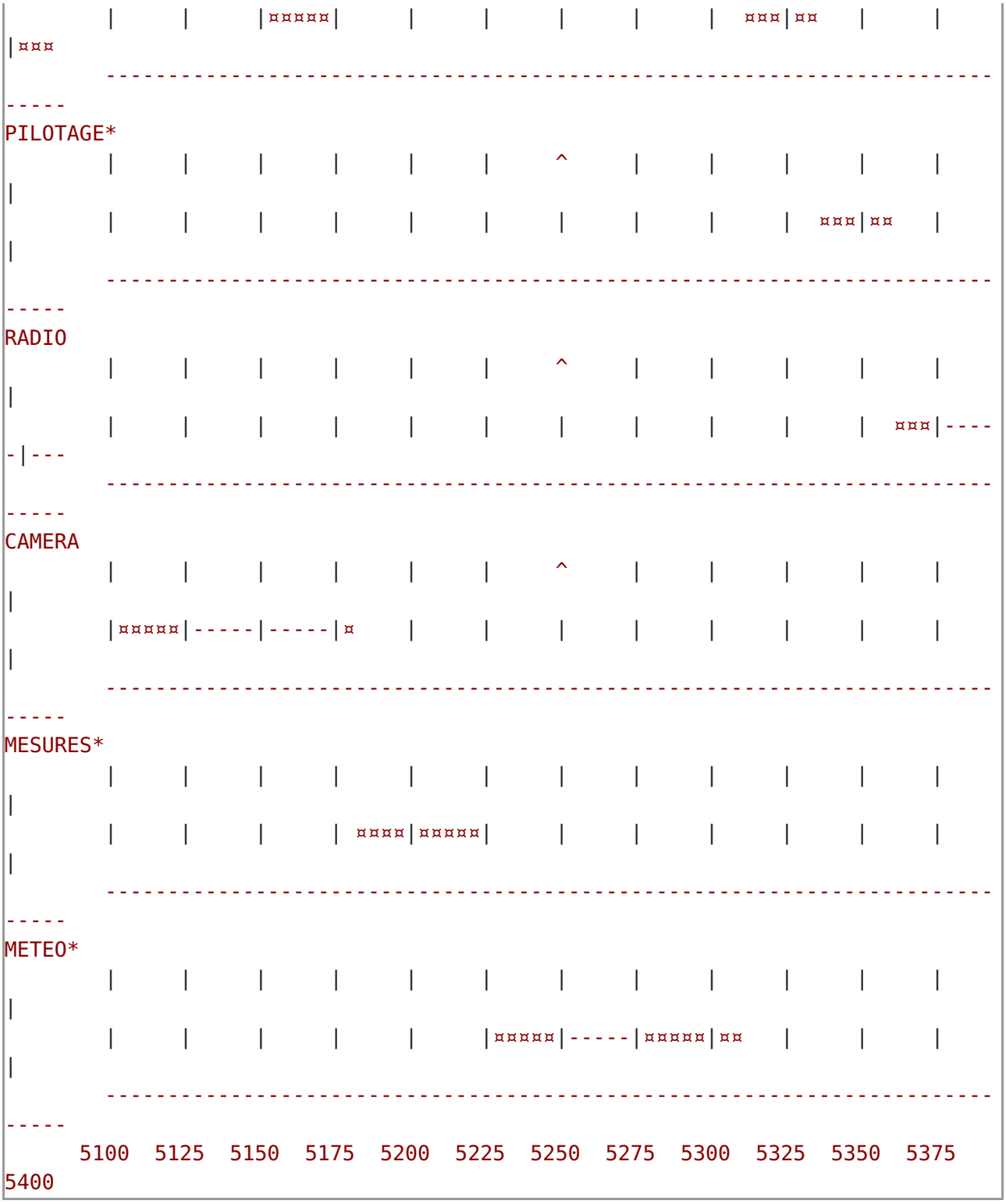
L'exécution de nos tâches avec un mutex sur la ressource et un temps de calcul de 60ms pour METEO donne le résultat suivant :

```
...
doing ORDO_BUS : 5000
doing ORDO_BUS ok : 5025
doing DISTRIB_DONNEES : 5025
doing DISTRIB_DONNEES ok : 5050p
doing PILOTAGE : 5051
doing PILOTAGE ok : 5076 || bprio: 5, cprio: 5
doing RADIO : 5076
doing RADIO ok : 5101 || bprio: 4, cprio: 4
doing CAMERA : 5101
doing ORDO_BUS : 5125
doing ORDO_BUS ok : 5150
doing DISTRIB_DONNEES : 5150
doing DISTRIB_DONNEES ok : 5175
doing CAMERA ok : 5176 || bprio: 3, cprio: 3
doing MESURES : 5176
doing MESURES ok : 5226 || bprio: 2, cprio: 2
doing METEO : 5226
doing ORDO_BUS : 5250
doing ORDO_BUS ok : 5275
doing METEO ok : 5311 || bprio: 1, cprio: 6
doing DISTRIB_DONNEES : 5311
doing DISTRIB_DONNEES ok : 5336
doing PILOTAGE : 5336
doing PILOTAGE ok : 5361 || bprio: 5, cprio: 5
doing RADIO : 5361
doing ORDO_BUS : 5375
doing ORDO_BUS ok : 5400
doing DISTRIB_DONNEES : 5400
doing DISTRIB_DONNEES ok : 5425
...
```

On observe qu'il n'y a plus de reset de la part de ORDO_BUS mais on voit également que le mutex augmente la priorité de METEO à 6 pour qu'elle ne monopolise pas la ressource, finisse rapidement son exécution et permette à DISTRIB_DONNEES de s'exécuter. Normalement, DISTRIB_DONNEES devrait ici s'exécuter à 5275 (juste après ORDO_BUS) mais on termine plutôt l'exécution de METEO (commencée à 5226).

Voici le chronogramme schématisant cette exécution :





On voit bien que la priorité de METEO est augmentée pour libérer rapidement le bus nécessité par les tâches de priorité plus importante (*héritage de priorité*).

Question 10 *Fournissez le code complet du programme, en prenant soin de le commenter.*

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/mman.h>
```

```c
#include <native/task.h>
#include <native/sem.h>
#include <native/mutex.h>
#include <native/timer.h>

#include <rtdk.h>

#define TASK_MODE T_JOINABLE
#define TASK_STKSZ 0

//#define SEM                       // compilation avec mutex, décommenter pour
compiler avec sémaphore

static RT_SEM sem;
static RT_MUTEX mutex;
static RT_SEM distrib_sem;
RTIME init_time;

typedef struct task_descriptor{
    RT_TASK task;                   // tâche temps réel
    void (*task_function)(void*);   // pointeur sur la fonction de la tâche à
exécuter
    RTIME period;                   // période d'exécution
    RTIME duration;                 // temps de calcul à chaque période
    int priority;                   // priorité de la tâche
    bool use_resource;              // défini si la tâche a besoin d'une
ressource
} task_descriptor;

////////////////////////////////////////////////////////
char* rt_task_name(void) {
    static RT_TASK_INFO info;
    rt_task_inquire(NULL,&info);

    return info.name;
}

////////////////////////////////////////////////////////
int time_since_start(void) {
    return (rt_timer_read()-init_time)/1000000;
}

////////////////////////////////////////////////////////
// Acquisition de la ressource par un mutex ou un sémaphore (selon SEM)
void acquire_resource(void) {
#ifdef SEM
    rt_sem_p(&sem, TM_INFINITE);
#else
    rt_mutex_acquire(&mutex, TM_INFINITE);
#endif
}
```

```
////////////////////////////////////////////////////////
void release_resource(void) {
#ifdef SEM
    rt_sem_v(&sem);
#else
    rt_mutex_release(&mutex);
#endif
}

////////////////////////////////////////////////////////
// Attente active sur un certain TEMPS D'EXECUTION de la tache
void busy_wait(RTIME time) {
    static RT_TASK_INFO info;
    rt_task_inquire(NULL,&info);

    // moment du compteur de temps d'exécution de la tâche courrante
    // auquel on va arrêter l'attente.
    // Insensible à l'éventuelle préamption de la tâche entre temps.
    RTIME time_to_end = info.exectime + time;
    do {
        rt_task_inquire(NULL,&info);
    } while (time_to_end > info.exectime);
}

////////////////////////////////////////////////////////
// Tâche de surveillance du fonctionnement du bus de données
void rt_task_ordobus(void *cookie) {
    struct task_descriptor* params=(struct task_descriptor*)cookie;

    rt_printf("started task %s, period %ims, duration %ims, use resource
%i\n",rt_task_name(),(int)(params->period/1000000),(int)(params-
>duration/1000000),params->use_resource);
    unsigned long brouette;

    RT_SEM_INFO sem_info;


    while(1) {
        rt_task_wait_period(&brouette);
        rt_sem_inquire(&distrib_sem, &sem_info);
        if(sem_info.count == 0){
            // Valeur actuelle du sémaphore = 0 -----> distribdonnees est en
cours de fonctionnement
            rt_printf("reset\n");
            exit(1);
        }
        rt_printf("doing %s : %d\n",rt_task_name(), time_since_start());
        busy_wait(params->duration);
        rt_printf("doing %s ok : %d\n",rt_task_name(), time_since_start());
    }
}
```

```
////////////////////////////////////////////////////////
void rt_task_distribdonnes(void *cookie) {
    struct task_descriptor* params=(struct task_descriptor*)cookie;

    rt_printf("started task %s, period %ims, duration %ims, use resource
%i\n",rt_task_name(),(int)(params->period/1000000),(int)(params-
>duration/1000000),params->use_resource);
    unsigned long brouette;

    while(1) {
        rt_task_wait_period(&brouette);
        if(params->use_resource) acquire_resource();
        rt_sem_p(&distrib_sem, TM_INFINITE);
        rt_printf("doing %s : %d\n",rt_task_name(), time_since_start());
        busy_wait(params->duration);
        rt_printf("doing %s ok : %d\n",rt_task_name(), time_since_start());
        rt_sem_v(&distrib_sem);
        if(params->use_resource) release_resource();
    }
}

////////////////////////////////////////////////////////
void rt_task(void *cookie) {
    struct task_descriptor* params=(struct task_descriptor*)cookie;

    rt_printf("started task %s, period %ims, duration %ims, use resource
%i\n",rt_task_name(),(int)(params->period/1000000),(int)(params-
>duration/1000000),params->use_resource);
    unsigned long brouette;

    while(1) {
        rt_task_wait_period(&brouette);
        if(params->use_resource) acquire_resource();
        rt_printf("doing %s : %d\n",rt_task_name(), time_since_start());
        busy_wait(params->duration);
        static RT_TASK_INFO info;
        rt_task_inquire(NULL,&info);
        rt_printf("doing %s ok : %d || bprio: %d, cprio: %d \n",rt_task_name(),
time_since_start(), info.bprio, info.cprio);
        if(params->use_resource) release_resource();
    }
}

////////////////////////////////////////////////////////
int create_and_start_rt_task(struct task_descriptor* desc,char* name) {
    int status=rt_task_create(&desc->task,name,TASK_STKSZ,desc-
>priority,TASK_MODE);
        if(status!=0) {
                printf("error creating task %s\n",name);
                return status;
        }
```

```
    status=rt_task_set_periodic(&desc->task,TM_NOW,desc->period);

    if(status!=0) {
                printf("error setting period on task %s\n",name);
                return status;
        }

        status=rt_task_start(&desc->task,desc->task_function,desc);
    if(status!=0) {
                printf("error starting task %s\n",name);
        }
    return status;
}


////////////////////////////////////////////////////////
int main(void) {

    /* Avoids memory swapping for this program */
    mlockall(MCL_CURRENT|MCL_FUTURE);

    rt_print_auto_init(1);

    init_time=rt_timer_read();

    // Création du sémaphore d'utilisation du bus 1553
    if (rt_sem_create(&sem, "wait port 1553", 1, S_PRIO) == -1){
            printf("rt_sem_create: failed: %s\n", strerror(errno));
    }

    // Création du sémaphore d'exécution de la tâche distribdonnees
    if (rt_sem_create(&distrib_sem, "distribdonnees running", 1, S_PRIO) == -1)
{
            printf("rt_sem_create: failed: %s\n", strerror(errno));
    }

    // Création du mutex d'utilisation du bus 1553
    if (rt_mutex_create(&mutex, "mutex 1553") == -1){
            printf("rt_mutex_create: failed: %s\n", strerror(errno));
    }

    // Création des tâches définies dans le sujet
    struct task_descriptor desc_ordo_bus;
    desc_ordo_bus.period = (RTIME)125000000;
    desc_ordo_bus.duration = (RTIME)25000000;
    desc_ordo_bus.priority = 7;
    desc_ordo_bus.use_resource = false;
    desc_ordo_bus.task_function = rt_task_ordobus;
    char* name_ordo_bus = "ORDO_BUS";
    create_and_start_rt_task(&desc_ordo_bus, name_ordo_bus);

    struct task_descriptor desc_distrib_donnees;
```

```
desc_distrib_donnees.period = (RTIME)125000000;
desc_distrib_donnees.duration = (RTIME)25000000;
desc_distrib_donnees.priority = 6;
desc_distrib_donnees.use_resource = true;
desc_distrib_donnees.task_function = rt_task_distribdonnes;
char* name_distrib_donnees = "DISTRIB_DONNEES";
create_and_start_rt_task(&desc_distrib_donnees, name_distrib_donnees);

struct task_descriptor desc_pilotage;
desc_pilotage.period = (RTIME)250000000;
desc_pilotage.duration = (RTIME)25000000;
desc_pilotage.priority = 5;
desc_pilotage.use_resource = true;
desc_pilotage.task_function = rt_task;
char* name_pilotage = "PILOTAGE";
create_and_start_rt_task(&desc_pilotage, name_pilotage);

struct task_descriptor desc_radio;
desc_radio.period = (RTIME)250000000;
desc_radio.duration = (RTIME)25000000;
desc_radio.priority = 4;
desc_radio.use_resource = false;
desc_radio.task_function = rt_task;
char* name_radio = "RADIO";
create_and_start_rt_task(&desc_radio, name_radio);

struct task_descriptor desc_camera;
desc_camera.period = (RTIME)250000000;
desc_camera.duration = (RTIME)25000000;
desc_camera.priority = 3;
desc_camera.use_resource = false;
desc_camera.task_function = rt_task;
char* name_camera = "CAMERA";
create_and_start_rt_task(&desc_camera, name_camera);

struct task_descriptor desc_mesures;
desc_mesures.period = (RTIME)5000000000;
desc_mesures.duration = (RTIME)50000000;
desc_mesures.priority = 2;
desc_mesures.use_resource = true;
desc_mesures.task_function = rt_task;
char* name_mesures = "MESURES";
create_and_start_rt_task(&desc_mesures, name_mesures);

struct task_descriptor desc_meteo;
desc_meteo.period = (RTIME)5000000000;
desc_meteo.duration = (RTIME)60000000;//40 à 60 ms
desc_meteo.priority = 1;
desc_meteo.use_resource = true;
desc_meteo.task_function = rt_task;
char* name_meteo = "METEO";
create_and_start_rt_task(&desc_meteo, name_meteo);
```

```
    getchar();
    return  EXIT_SUCCESS;
}
```