Université de Technologie de Compiègne

MI11

Rapport du TP2 Xenomai

Clément BLANQUET et Rafik CHENNOUF

Juin 2017

Sommaire

Table des figures

Chapitre 1

TP2 Xenomai

Exercice : Pathfinder

1.1 Question 1

La fonction *create_and_start_rt_task* sert, comme son nom l'indique, à créer et lancer une tâche. Elle commence par créer la tâche (avec rt_task_create), puis lui assigne une période (avec rt_task_set_periodic) et enfin la lance (rt_task_start).

La fonction *rt_task* permet de simuler l'exécution d'une tâche temps réel en effectuant une attente active d'une durée d'une période grâce à la fonction busy_wait.

Elle permet d'obtenir les informations suivantes :

— à chaque nouvelle période, le nom de la tâche s'affiche
— la priorité de base et la priorité actuelle de la tâche
— à la fin de la période, le nom de la tâche s'affiche avec un "ok" (qui montre la fin de la période)

La structure *task_descriptor* permet d'obtenir des informations de la tâche en cours, c'est à dire :

— la tâche elle même (de type RT_TASK)
— le pointeur vers la fonction associée
— la période de la tâche
— la durée d'exécution la tâche
— la priorité de la tâche
— si la tâche utilise une ressource ou non

1.2 Question 2

La fonction *rt_task_name* sert à obtenir le nom de la tâche en cours grâce à la structure RT_TASK_INFO et son champ *name*. Cette structure a d'autres champs :

— bprio : priorité de base (ne change pas au cours du temps)
— cprio : priorité actuelle (peut changer au cours du temps)
— status : statut de la tâche
— relpoint : temps restant avant la prochaine exécution
— exectime : temps d'exécution de la tâche depuis son lancement

— modeswitches : nombre de changements de mode primaire / secondaire
— ctxswitches : nombre de changements de contextes
— pagefaults : nombre de défauts de page

1.3 Question 3

Voici notre fonction *busy_wait* :

```
void busy_wait(RTIME time)
{
    static RT_TASK_INFO info; // info sur la tâche
    // initialisation des infos dont exectime
    rt_task_inquire(NULL,&info);
    // recuperation du temps d'execution depuis le debut
    RTIME begin = info.exectime;

    //Tant que le temps d'execution n'est pas terminé
    while(info.exectime - begin < time)
        rt_task_inquire(NULL,&info); // mise à jour des infos
```

On commence par acquérir les informations relatives à la tâche grâce à une structure RT_TASK_INFO et à la fonction rt_task_inquire. La champ *exectime* de cette structure nous permet d'obtenir le temps d'exécution de la tâche depuis son lancement (donc il y a certainement eu plusieurs occurrences). On appelle ce temps *begin*. Ce qu'on veut, c'est simuler le temps d'**une** exécution de la tâche.

Pour cela, on va, dans une boucle, vérifier la différence entre le temps d'exécution actuel de la tâche (avec le champ *exectime* de RT_TASK_INFO mis à jour) et le temps d'exécution de la tâche initial (qu'on a appelé *begin*). Dès que cette différence atteint la durée voulue (durée d'exécution de la tâche donnée en paramètre), on peut sortir de la boucle et de la fonction. De cette façon, on a réalisé une attente active pendant la durée recherchée.

1.4 Question 4

On se sert de la fonction time_since_start dans la fonction rt_task pour avoir un point de repère temporel :

```
rt_printf("doing %s time : %d\n",rt_task_name(), time_since_start());
```

Voici le résultat de l'exécution du programme :

```
root@devkit8600-xenomai:~# ./pathfinder
started task ORDO_BUS, period 125ms, duration 25ms, use resource 0
doing ORDO_BUS    time : 125
doing ORDO_BUS ok    time : 150
doing ORDO_BUS    time : 250
doing ORDO_BUS ok    time : 275
doing ORDO_BUS    time : 375
doing ORDO_BUS ok    time : 400
doing ORDO_BUS    time : 500
doing ORDO_BUS ok    time : 525
doing ORDO_BUS    time : 625
doing ORDO_BUS ok    time : 650
doing ORDO_BUS    time : 750
doing ORDO_BUS ok    time : 775
doing ORDO_BUS    time : 875
doing ORDO_BUS ok    time : 900
doing ORDO_BUS    time : 1000
doing ORDO_BUS ok    time : 1025
doing ORDO_BUS    time : 1125
doing ORDO_BUS ok    time : 1150
doing ORDO_BUS    time : 1250
doing ORDO_BUS ok    time : 1275
```

FIGURE 1.1 – Résultat avec ORDO_BUS

Le timing est bon : 25ms d'exécution et 125ms de période.

1.5 Question 5

Pour une bonne coordination des tâches, le sémaphore doit être utilisé comme suit :

- Faire un sem_p (-1) dans acquire_resource et un sem_v (+1) dans release_resource sur le sémaphore
- Initialisation du sémaphore à 0 (bloque tout)
- Initialisation de toutes les tâches (et donc toutes les tâches sont bloquées)
- Faire un sem_v (+1) sur le sémaphore juste après ces initialisations (ce qui débloquera la tâche la plus prioritaire)

De cette façon, les tâches s'enchaîneront de la bonne manière selon leurs priorités.

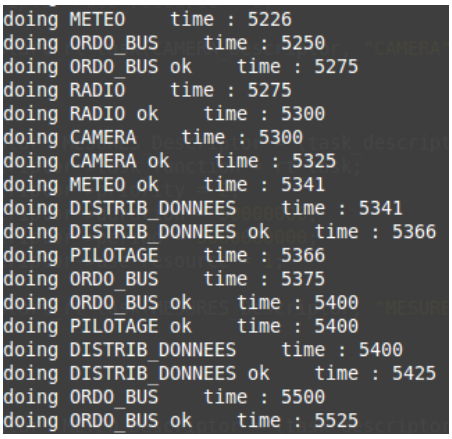

FIGURE 1.2 – Enchaînement avec meilleur cas pour METEO (40ms)

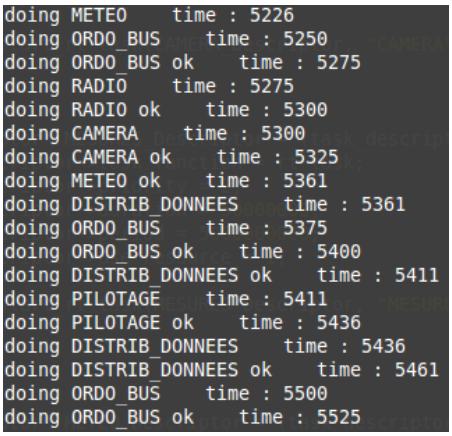

FIGURE 1.3 – Enchaînement avec pire cas pour METEO (60ms)

On se rend compte que, lors du meilleur cas pour la tâche METEO (soit 40ms), ORDO_BUS n'a pas encore terminé sa période et n'est donc pas prêt pour exécution avant le début de la tâche PILOTAGE ou la fin de DISTRIB_DONNEES. Dans le pire cas pour la tâche METEO (soit 60ms), ORDO_BUS a atteint sa période et est donc exécuté avant la tâche PILOTAGE et s'intercale entre le début et le fin de l'exécution de DISTRIB_DONNEES.

1.6 Question 6

Pour être sur qu'entre deux exécutions de ORDO_BUS, DISTRIB_DONNEES s'est exécutée entièrement, nous pouvons utiliser une seconde sémaphore en mode S_PRIO (comme la première) qui vaut 1 avant la création des tâches pour que la tâche ORDO_BUS puisse s'exécuter au début. Nous créons ensuite deux nouvelles fonctions **rt_task_ordo_bus** et **rt_task_distrib_donnees** pour remplacer la fonction de base **rt_task**. Dans la fonction **rt_task_distrib_donnees**, lors de l'exécution de la tâche DISTRIB_DONNEES, il faut incrémenter la sémaphore pour prévenir la tâche ORDO_BUS que la tâche DISTRIB_DONNEES est en cours d'exécution. Si la tâche DISTRIB_DONNEES est en cours d'exécution et qu'on est en train d'exécuter la tâche ORDO_BUS alors il faut lancer un reset et terminer le programme. Pour cela, dans la fonction **rt_task_ordo_bus**, il faut vérifier si une unité du sémaphore est disponible, si c'est le cas alors on exécute la tâche normalement sinon on lance un reset et on termine le programme. Il faut faire attention de ne pas bloquer la tâche ORDO_BUS sur le sémaphore car la suite du code doit pouvoir s'exécuter.

1.7 Question 7

Nous avons testé notre programme pour le cas extrême du temps d'exécution de METEO (60ms) sans terminer notre programme après le reset afin de mieux voir ce qu'il se passe. Voici le résultat obtenu :

```
doing METEO    time : 5226
doing ORDO_BUS    time : 5250
doing ORDO_BUS ok    time : 5275
doing RADIO    time : 5275
doing RADIO ok    time : 5300
doing CAMERA    time : 5300
doing CAMERA ok    time : 5325
doing METEO ok    time : 5361
doing DISTRIB_DONNEES    time : 5361
RESET

doing ORDO_BUS    time : 5375
doing ORDO_BUS ok    time : 5400
doing DISTRIB_DONNEES ok    time : 5411
doing PILOTAGE    time : 5411
```

FIGURE 1.4 – Reset avec le cas extrême pour METEO (60ms)

On voit que le reset se place entre le début de la tâche DISTRIB_DONNEES et le début de la tâche ORDO_BUS ce qui valide le fonctionnement de la solution.

Cependant, on remarque que lorsque la tâche METEO démarre elle s'empare du sémaphore puis elle est préemptée par la tâche ORDO_BUS de plus haute priorité. A la fin de la tâche ORDO_BUS, la tâche DISTRIB_DONNEES démarre mais elle est bloquée par le sémaphore car c'est la tâche METEO qui le possède. Ensuite, après les exécutions des tâches RADIO et CAMERA, la tâche METEO reprend la main et se termine en libérant le sémaphore ce qui permet à la tâche DISTRIB_DONNEES de reprendre son exécution avant le Reset. Ceci est problématique car une tâche de plus haute priorité (DISTRIB_DONNEES) ne peut pas avoir accès au processeur car une tâche de plus faible priorité (METEO) l'utilise. C'est ce qu'on appelle une inversion de priorité. Le chronogramme suivant illustre ce qu'il se passe :

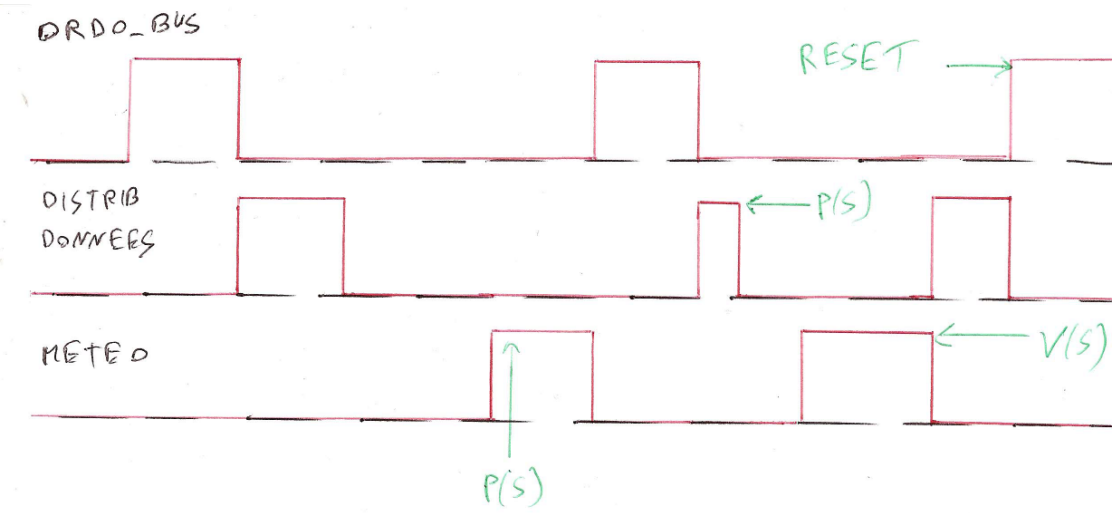

FIGURE 1.5 – Chronogramme illustrant l'inversion de priorité

1.8 Question 8

Afin de résoudre ce problème, nous pouvons utiliser un mutex à la place de notre premier sémaphore. En effet, un mutex hérite de la priorité la plus élevée parmi celles de toutes les tâches qui le demandent et une tâche s'exécute avec la priorité la plus élevée parmi celles de tous les mutex qu'il tient.

1.9 Question 9

Après lancement du programme, nous obtenons le résultat suivant :

```
Name : ORDO_BUS  Base prio : 7  Current prio : 7
doing ORDO_BUS ok    time : 10275
Name : METEO  Base prio : 1  Current prio : 6
doing METEO ok    time : 10275
doing DISTRIB_DONNEES    time : 10275
Name : DISTRIB_DONNEES  Base prio : 6  Current prio : 6
doing DISTRIB_DONNEES ok    time : 10300
```

FIGURE 1.6 – Résultat avec un mutex

On voit bien que la tâche METEO a hérité de la priorité de la tâche DISTRIB_DONNEES (6) ce qui lui permet de reprendre son exécution normalement et de se terminer. Ensuite la tâche DISTRIB_DONNEES prend la main et s'exécute. Il n'y a plus du tout d'inversion de priorité.

1.10 Question 10

```
1  #include <stdio.h>
   #include <stdlib.h>
3  #include <stdbool.h>
   #include <sys/mman.h>
5
   #include <native/task.h>
7  #include <native/timer.h>
   #include <native/sem.h>
9  #include <native/mutex.h>
   #include <rtdk.h>
11
   #define TASK_MODE T_JOINABLE
13 #define TASK_STKSZ 0

15 RTIME init_time;
   RT_SEM sem;
17 RT_SEM semReset;
   RT_MUTEX mutex;
19


21 typedef struct task_descriptor{
     RT_TASK task;
23   void (*task_function)(void*);
```

```
  RTIME period;
  RTIME duration;
  int priority;
  bool use_resource;
} 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;
}

//////////////////////////////////////////////////////////
void acquire_resource(void) {

  // Version non fonctionnelle avec sémaphore
  // rt_sem_p (&sem, TM_INFINITE );

  // Version fonctionnelle avec mutex
  rt_mutex_acquire (&mutex, TM_INFINITE);
}

//////////////////////////////////////////////////////////
void release_resource(void) {

  // Version non fonctionnelle avec sémaphore
  //  rt_sem_v (&sem);

  // Version fonctionnelle avec mutex
  rt_mutex_release(&mutex);
}

//////////////////////////////////////////////////////////
void busy_wait(RTIME time)
{
    static RT_TASK_INFO info;      // Info sur la tâche

    // Initialisation des infos dont exectime
    rt_task_inquire(NULL,&info);

    // Recuperation du temps d'execution initial (> 0)
    RTIME begin = info.exectime;
```

```
73
75      while(info.exectime - begin < time)
        {
77          // Mise à jour des infos dont exectime
            rt_task_inquire(NULL,&info);
79      }
}
81

83  void rt_task(void *cookie) {
      struct task_descriptor* params=(struct task_descriptor*)cookie;
85
      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);
87
      while(1) {
89      rt_task_wait_period(NULL);
        if(params->use_resource) acquire_resource();
91      rt_printf("doing %s    time : %d\n",rt_task_name(),
        time_since_start());
        busy_wait(params->duration);
93
        static RT_TASK_INFO info;       // Info sur la tâche
95      rt_task_inquire(NULL,&info);     // Initialisation des infos

97    // Test du mutex avec priorité de base et priorité courrante
        rt_printf("Name : %s  Base prio : %d  Current prio : %d\n", info.
        name, info.bprio, info.cprio);
99
        rt_printf("doing %s ok    time : %d\n",rt_task_name(),
        time_since_start());
101     if(params->use_resource) release_resource();
      }
103 }

105 // Fonction de la tâche ORDO_BUS implémentant le RESET
    void rt_task_ordo_bus(void *cookie) {
107
      struct task_descriptor* params=(struct task_descriptor*)cookie;
109
      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);
111
      while(1) {
113     rt_task_wait_period(NULL);
        if(params->use_resource) acquire_resource();
```

```
115
// Si la décrémentation n'est pas possible, on ne bloque pas la tâche
117     if(rt_sem_p (&semReset, TM_NONBLOCK ) == -EWOULDBLOCK )
  {
119      rt_printf("RESET\n\n\n\n\n\n\n\n\n\n\n\n");  // Affichage du RESET
      exit(-1);                    // Exit du programme
121   }

123     rt_printf("doing %s     time : %d\n",rt_task_name(),
    time_since_start());
     busy_wait(params->duration);
125
     static RT_TASK_INFO info;
127     rt_task_inquire(NULL,&info);

129   // Test du mutex avec priorité de base et priorité courrante
     rt_printf("Name : %s  Base prio : %d  Current prio : %d\n", info.
    name, info.bprio, info.cprio);
131
     rt_printf("doing %s ok     time : %d\n",rt_task_name(),
    time_since_start());
133     if(params->use_resource) release_resource();
  }
135 }

137
// Fonction de la tâche DISTRIB_DONNEES implémentant le RESET
139 void rt_task_distrib_donnees(void *cookie) {

141   struct task_descriptor* params=(struct task_descriptor*)cookie;

143   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);

145   while(1) {
     rt_task_wait_period(NULL);
147     if(params->use_resource) acquire_resource();
     rt_printf("doing %s     time : %d\n",rt_task_name(),
    time_since_start());
149     busy_wait(params->duration);

151     static RT_TASK_INFO info;
     rt_task_inquire(NULL,&info);
153
     // Test du mutex avec priorité de base et priorité courante
155     rt_printf("Name : %s  Base prio : %d  Current prio : %d\n", info.
    name, info.bprio, info.cprio);
```

```
157     rt_printf("doing %s ok     time : %d\n",rt_task_name(),
        time_since_start());

159     // Incrémentation du sémaphore pour le RESET
        rt_sem_v (&semReset);
161
        if(params->use_resource) release_resource();
163

165   }

167 }

169
int create_and_start_rt_task(struct task_descriptor* desc,char* name)
      {
171   int status=rt_task_create(&desc->task,name,TASK_STKSZ,desc->
       priority,TASK_MODE);
      if(status!=0) {
173     printf("error creating task %s\n",name);
        return status;
175   }

177   status=rt_task_set_periodic(&desc->task,TM_NOW,desc->period);
      if(status!=0) {
179     printf("error setting period on task %s\n",name);
        return status;
181   }

183   status=rt_task_start(&desc->task,desc->task_function,desc);
      if(status!=0) {
185     printf("error starting task %s\n",name);
      }
187   return status;
}
189


191
int main(void) {
193
      //Avoids memory swapping for this program
195   mlockall(MCL_CURRENT|MCL_FUTURE);

197   rt_print_auto_init(1);

199   init_time=rt_timer_read();

201   // Version non fonctionnelle avec sémaphore
      //rt_sem_create (&sem, "sem", 0, S_PRIO);
```

```
203
    // Version fonctionnelle avec mutex
205 rt_mutex_create (&mutex, "mutex");

207 // Sémaphore pour le RESET
    rt_sem_create (&semReset, "semReset", 1, S_PRIO);
209
// Initialisation, création et lancement de la tâche ORDO_BUS
211 task_descriptor* ORDO_BUS_Descriptor = (task_descriptor*)malloc(
      sizeof(task_descriptor));
    ORDO_BUS_Descriptor->task_function = rt_task_ordo_bus;
213 ORDO_BUS_Descriptor->priority = 7;
    ORDO_BUS_Descriptor->duration = 25000000;
215 ORDO_BUS_Descriptor->period = 125000000;
    ORDO_BUS_Descriptor->use_resource = 0;
217
    create_and_start_rt_task(ORDO_BUS_Descriptor, "ORDO_BUS");
219

221 // Initialisation, création et lancement de la tâche
      DISTRIB_DONNEES
    task_descriptor* DISTRIB_DONNEES_Descriptor = (task_descriptor*)
      malloc(sizeof(task_descriptor));
223 DISTRIB_DONNEES_Descriptor->task_function = rt_task_distrib_donnees
      ;
    DISTRIB_DONNEES_Descriptor->priority = 6;
225 DISTRIB_DONNEES_Descriptor->duration = 25000000;
    DISTRIB_DONNEES_Descriptor->period = 125000000;
227 DISTRIB_DONNEES_Descriptor->use_resource = 1;

229 create_and_start_rt_task(DISTRIB_DONNEES_Descriptor, "
      DISTRIB_DONNEES");

231
    // Initialisation, création et lancement de la tâche PILOTAGE
233 task_descriptor* PILOTAGE_Descriptor = (task_descriptor*)malloc(
      sizeof(task_descriptor));
    PILOTAGE_Descriptor->task_function = rt_task;
235 PILOTAGE_Descriptor->priority = 5;
    PILOTAGE_Descriptor->duration = 25000000;
237 PILOTAGE_Descriptor->period = 250000000;
    PILOTAGE_Descriptor->use_resource = 1;
239
    create_and_start_rt_task(PILOTAGE_Descriptor, "PILOTAGE");
241

243 // Initialisation, création et lancement de la tâche RADIO
    task_descriptor* RADIO_Descriptor = (task_descriptor*)malloc(sizeof
      (task_descriptor));
```

```
245   RADIO_Descriptor->task_function = rt_task;
      RADIO_Descriptor->priority = 4;
247   RADIO_Descriptor->duration = 25000000;
      RADIO_Descriptor->period = 250000000;
249   RADIO_Descriptor->use_resource = 0;

251   create_and_start_rt_task(RADIO_Descriptor, "RADIO");


253
      // Initialisation, création et lancement de la tâche CAMERA
255   task_descriptor* CAMERA_Descriptor = (task_descriptor*)malloc(
        sizeof(task_descriptor));
      CAMERA_Descriptor->task_function = rt_task;
257   CAMERA_Descriptor->priority = 3;
      CAMERA_Descriptor->duration = 25000000;
259   CAMERA_Descriptor->period = 250000000;
      CAMERA_Descriptor->use_resource = 0;
261
      create_and_start_rt_task(CAMERA_Descriptor, "CAMERA");
263


265   // Initialisation, création et lancement de la tâche MESURES
      task_descriptor* MESURES_Descriptor = (task_descriptor*)malloc(
        sizeof(task_descriptor));
267   MESURES_Descriptor->task_function = rt_task;
      MESURES_Descriptor->priority = 2;
269   MESURES_Descriptor->duration = 50000000;
      MESURES_Descriptor->period = 5000000000;
271   MESURES_Descriptor->use_resource = 1;

273   create_and_start_rt_task(MESURES_Descriptor, "MESURES");


275
      // Initialisation, création et lancement de la tâche METEO
277   task_descriptor* METEO_Descriptor = (task_descriptor*)malloc(sizeof
        (task_descriptor));
      METEO_Descriptor->task_function = rt_task;
279   METEO_Descriptor->priority = 1;
      METEO_Descriptor->duration = 60000000; // 40000000
281   METEO_Descriptor->period = 5000000000;
      METEO_Descriptor->use_resource = 1;
283
      create_and_start_rt_task(METEO_Descriptor, "METEO");
285
      // Version non fonctionnelle avec sémaphore
287   // rt_sem_v (&sem); // Incrémentation du sémaphore principal

289   getchar();
```

```
291     return   EXIT_SUCCESS;
}
```