Université de Technologie de Compiègne

MI11

Rapport du TP1 Xenomai

Clément BLANQUET et Rafik CHENNOUF

Juin 2017

Sommaire

Table des figures

Chapitre 1

TP1 Xenomai

Exercice 1 : tâches

1.1 Question 1.1

Un code "classique" ne s'exécute pas de façon temps réel. En effet, il n'apparaît pas dans le fichier /proc/xenomai/stats ce qui signifie qu'il n'est pas temps réel.

```
root@devkit8600-xenomai:~# cat /proc/xenomai/stat
CPU  PID    MSW        CSW        PF    STAT       %CPU  NAME
  0  0      0          0          0     00500080  100.0  ROOT
  0  0      0          9592       0     00000000    0.0  IRQ68: [timer]
```

FIGURE 1.1 – Statistiques Xenomai avec un programme non temps réel

1.2 Question 1.2

Voici le code qui permet de créer une tâche temps réel :

```
#include <stdio.h>
#include <native/task.h>
#include <analogy/os_facilities.h>
#include <unistd.h>
#include <sys/mman.h>

#define TASK_PRIO 99
#define TASK_MODE 0
#define TASK_STKSZ 0

RT_TASK task_printf;

void task_hello() // Notre future tache temps reel
{
  while(1)
  {
    sleep(rt_timer_ns2ticks(1000000000));
    printf("Hello World !\n");
  }
}

```

```
int main()
{
  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);
  int err;
  //Creation de la tache temps reel
  err = rt_task_create(&task_printf, "hello world", TASK_STKSZ,
   TASK_PRIO, TASK_MODE);
  if(!err)
    rt_task_start(&task_printf, &task_hello, NULL);

  getchar();

  return 0;
}
```

Le chemin des fichiers à inclure est :

```
/opt/poky/1.7.3/sysroots/armv7a-vfp-neon-poky-linux-gnueabi/usr/
    include/xenomai
```

Le chemin des librairies est :

```
/opt/poky/1.7.3/sysroots/armv7a-vfp-neon-poky-linux-gnueabi/usr/lib
```

Cela nous donne la ligne de commande (après avoir fait un *source*) :

```
$CC -o main main.c -I/opt/poky/1.7.3/sysroots/armv7a-vfp-neon-poky-
    linux-gnueabi/usr/include/xenomai -L/opt/poky/1.7.3/sysroots/
    armv7a-vfp-neon-poky-linux-gnueabi/usr/lib -lxenomai -lnative
```

1.3 Question 1.3

Cette application n'est toujours pas temps réel. En effet, malgré la création d'une tâche temps réel, les fonctions *sleep* et *printf* qui s'y trouvent ne sont pas temps réel.

Le fichier de statistiques Xenomai donne :

```
root@devkit8600-xenomai:~# cat /proc/xenomai/stat
CPU  PID    MSW        CSW        PF    STAT       %CPU  NAME
  0  0      0          335        0     00500080  100.0  ROOT
  0  0      0          11084      0     00000000    0.0  IRQ68: [timer]
```

FIGURE 1.2 – Statistiques Xenomai avec une tâche temps réel

1.4 Question 1.4

Voici la fonction task_hello après avoir remplacé la fonction sleep par son équivalent temps réel :

```
1 void task_hello()
{
3   while(1)
    {
5     rt_task_sleep(rt_timer_ns2ticks(1000000000));
      printf("Hello World !\n");
7   }
}
```

Le fichier de statistiques Xenomai donne :

```
root@devkit8600-xenomai:~# cat /proc/xenomai/stat
CPU  PID    MSW        CSW        PF    STAT       %CPU  NAME
  0  0      0          202        0     00500080  100.0  ROOT
  0  965    4          9          0     00300184    0.0  hello world
  0  0      0          7802       0     00000000    0.0  IRQ68: [timer]
```

FIGURE 1.3 – Statistiques Xenomai avec une tâche temps réel et la fonction rt_task_sleep

1.5 Question 1.5

Voici le code après avoir remplacé les fonctions sleep et printf par leurs équivalents temps réel :

```
void task_hello()
2 {
  while(1)
4 {
    rt_task_sleep(rt_timer_ns2ticks(1000000000));
6   rt_printf("Hello World !\n");
  }
8 }
```

Le fichier de statistiques Xenomai donne :

```
root@devkit8600-xenomai:~# cat /proc/xenomai/stat
CPU  PID    MSW        CSW        PF    STAT       %CPU  NAME
  0  0      0          46         0     00500080  100.0  ROOT
  0  957    0          46         0     00300184    0.0  hello world
  0  0      0          2860       0     00000000    0.0  IRQ68: [timer]
```

FIGURE 1.4 – Statistiques Xenomai avec une tâche temps réel et les fonctions rt_task_sleep et rt_printf

Interprétation des statistiques Xenomai : On constate que :

— Lorsqu'on crée une tâche temps réel qui n'utilise aucune fonction temps réel (dans notre exemple, les fonctions *sleep* et *printf*), elle n'apparaît pas dans le fichier de statistiques. Elle n'est en fait absolument pas temps réel.

— Lorsque, dans cette même tâche, on rend la fonction *sleep* temps réel (en la remplaçant par *rt_task_sleep*), elle apparaît dans le fichier de statistiques. On constate que son nombre de changements de contexte (9) est très inférieur au nombre de changements de contexte du ROOT (282), ce qui signifie que notre tâche temps-réel a bien la priorité sur a tâche ROOT (Linux). La tâche temps-réel est rarement interrompue. De plus, notre tâche a un nombre de changement de modes (MSW) égal à 4. Elle passe donc plusieurs fois du mode primaire, qui correspond à un mode temps-réel dur, à un mode secondaire qui correspond à un mode temps-réel mou et vise versa.

— Lorsque les fonctions *sleep* et *printf* ont été remplacées par leurs équivalents temps réel (*rt_task_sleep* et *rt_printf*), le nombre changements de contextes de la tâche est égal à celui de ROOT (46) mais nous remarquons surtout qu'il n'y a pas eu de changements de mode.

Exercice 2 : Synchronisation

2.1 Question 2.1

Voici le code du programme lançant deux tâches Xenomai qui afficheront chacune une partie du message "Hello World !" :

```
#include <stdio.h>
#include <native/task.h>
#include <analogy/os_facilities.h>
#include <unistd.h>
```

```c
#include <sys/mman.h>

#define TASK_PRIO 99
#define TASK_MODE 0
#define TASK_STKSZ 0

RT_TASK task_printf;
RT_TASK task_printf2;

void task_hello() //Premiere tache temps reel
{
  rt_printf("Hello\n");
}

void task_world() //Deuxieme tache temps reel
{
  rt_printf("World !\n");
  rt_task_sleep(rt_timer_ns2ticks(1000000000));
}


int main()
{
  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);
  int err1, err2;
  //Creation des deux taches temps reel
  err1 = rt_task_create(&task_printf, "hello", TASK_STKSZ, TASK_PRIO,
    TASK_MODE);
  err2 = rt_task_create(&task_printf2, "world", TASK_STKSZ, TASK_PRIO
    , TASK_MODE);
  if(!err1 && !err2)
  {
    rt_task_start(&task_printf, &task_hello, NULL); //Lancement tache
    rt_task_join(&task_printf); //Attente tache
    rt_task_start(&task_printf2, &task_world, NULL); //Lancement
   tache
    rt_task_join(&task_printf2); //Attente tache
  }

  getchar();

  return 0;
}
```

Le résultat est le suivant :

```
root@devkit8600-xenomai:~# ./synchro
Hello
```

```
World !
c
root@devkit8600-xenomai:~#
```

2.2 Question 2.2

Pour le moment, les priorités des tâches n'ont aucune influence. En effet, elles sont lancées dans l'ordre dans lequel elles se trouvent dans notre code : après le *rt_task_start*, rien ne "bloque" l'exécution de la tâche ; les tâches se lancent donc l'une après l'autre comme on l'a défini. Pour afficher les messages dans le désordre, il faut donc inverser les deux tâches dans le code. On peut aussi utiliser des sémaphores, comme vu dans les questions suivantes.

2.3 Question 2.3

Il faut initialiser le sémaphore à 0 de manière à bloquer les tâches.

2.4 Question 2.4

Le paramètre *mode* lors de la création du sémaphore nous sert à définir le mode d'ordonnancement à utiliser. Par exemple, si on choisit le mode *S_FIFO* alors les tâches seront ordonnancés en suivant la méthode FIFO (First In First Out), c'est à dire que les tâches attendront dans leur ordre d'arrivée que le sémaphore se libère. Un second exemple de *mode* utilisable est le mode *S_PRIO* qui permet d'ordonnancer les tâches par ordre de priorité, c'est à dire que les tâches avec la plus haute priorité auront accès au sémaphore avant les tâches de plus faible priorité.

2.5 Question 2.5

```
#include <stdio.h>
#include <native/task.h>
#include <analogy/os_facilities.h>
#include <unistd.h>
#include <sys/mman.h>
#include <native/sem.h>

#define TASK_PRIO_HELLO 98
#define TASK_PRIO_WORLD 99
//#define TASK_PRIO 99
#define TASK_MODE 0
```

```
#define TASK_STKSZ 0

RT_TASK task_printf;
RT_TASK task_printf2;
RT_SEM sem;


void task_hello()      // affichage de 'Hello'
{
  rt_sem_p (&sem, 0); // décrémentation du sémaphore
  rt_printf("Hello\n");
}

void task_world()      // affichage de 'World'
{
  rt_sem_p (&sem, 0); // décrémentation du sémaphore
  rt_printf("World !\n");
}


int main()
{
  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);
  int err1, err2;

  // création du sémaphore en mode FIFO
  rt_sem_create (&sem, "sem", 0, S_FIFO);

  // création des deux tâches
  err1 = rt_task_create(&task_printf, "hello", TASK_STKSZ,
   TASK_PRIO_HELLO, TASK_MODE);
  err2 = rt_task_create(&task_printf2, "world", TASK_STKSZ,
   TASK_PRIO_WORLD, TASK_MODE);


  if(!err1 && !err2)
  {
    // démarrage de la tâche qui affiche 'Hello'
    rt_task_start(&task_printf, &task_hello, NULL);

    // attente de sa terminaison
    rt_task_join(&task_printf);

    // démarrage de la tâche qui affiche 'World'
    rt_task_start(&task_printf2, &task_world, NULL);

    // attente de sa terminaison
    rt_task_join(&task_printf2);
```

```
59
        getchar();
61
        rt_sem_v (&sem);   // incrémentation du sémaphore
63      rt_sem_v (&sem);   // incrémentation du sémaphore
    }
65
    return 0;
67  }
```

Le programme précédent avec le mode *S_FIFO* pour le sémaphore nous donne le résultat suivant :

```
1  root@devkit8600-xenomai:~# ./synchro

3  Hello
   World !
```

Cependant, si nous utilisons le mode *S_PRIO* pour le sémaphore nous obtenons :

```
   root@devkit8600-xenomai:~# ./synchro
2
   World !
4  Hello
```

Ceci est cohérent puisque nous avons défini les priorités comme suit :

```
   #define TASK_PRIO_HELLO 98
2  #define TASK_PRIO_WORLD 99
```

La tâche qui s'occupe d'afficher le 'World !' est plus prioritaire que la tâche qui affiche le 'Hello'.

2.6 Question 2.6

```
   #include <stdio.h>
2  #include <native/task.h>
   #include <analogy/os_facilities.h>
4  #include <unistd.h>
   #include <sys/mman.h>
6  #include <native/sem.h>
```

```
8  #define TASK_PRIO_HELLO 98
   #define TASK_PRIO_WORLD 97
10 #define TASK_PRIO_METRO 99
   //#define TASK_PRIO 99
12 #define TASK_MODE 0
   #define TASK_STKSZ 0
14
   RT_TASK task_printf;
16 RT_TASK task_printf2;
   RT_TASK task_metronome;
18 RT_SEM sem;

20 void task_hello()
   {
22   while(1)
     {
24     rt_sem_p (&sem, 0); // décrémentation du sémaphore
       rt_printf("Hello\n");
26   }
   }
28
   void task_world()
30 {
     while(1)
32   {
       rt_sem_p (&sem, 0); // décrémentation du sémaphore
34     rt_printf("World !\n");
       rt_sem_v (&sem);    // incrémentation du sémaphore
36   }
   }
38
   void task_metro()   // tâche métronome
40 {
     while(1)
42   {
       rt_sem_v (&sem); // incrémentation du sémaphore
44     rt_sem_v (&sem); // incrémentation du sémaphore

46     rt_sem_p (&sem, 0); // décrémentation du sémaphore
       rt_task_sleep(rt_timer_ns2ticks(1000000000)); // attente 1 sec
48   }
   }
50

52 int main()
   {
54   mlockall(MCL_CURRENT|MCL_FUTURE);
     rt_print_auto_init(1);
```

```
56   int err1, err2, err3;

58   // création du sémaphore en mode PRIO
     rt_sem_create (&sem, "sem", 0, S_PRIO);
60
     err1 = rt_task_create(&task_printf, "hello", TASK_STKSZ,
      TASK_PRIO_HELLO, TASK_MODE);
62   err2 = rt_task_create(&task_printf2, "world", TASK_STKSZ,
      TASK_PRIO_WORLD, TASK_MODE);

64   // création de la tâche métronome
     err3= rt_task_create(&task_metronome, "metro", TASK_STKSZ,
      TASK_PRIO_METRO, TASK_MODE);
66
     if(!err1 && !err2 && !err3)
68   {
       rt_task_start(&task_printf, &task_hello, NULL);
70     rt_task_join(&task_printf);
       rt_task_start(&task_printf2, &task_world, NULL);
72     rt_task_join(&task_printf2);

74     // démarrage de la tâche métronome
       rt_task_start(&task_metronome, &task_metro, NULL);
76     // attente de sa terminaison
       rt_task_join(&task_metronome);
78
       getchar();
80
     }
82
     return 0;
84 }
```

Le programme ci-dessus nous donne le résultat suivant à l'infini :

```
root@devkit8600-xenomai:~# ./synchro
2
Hello
4 World !
Hello
6 World !
Hello
8 World !
```

La tâche métronome possède la plus haute priorité comme on peut le voir ci-dessous :

```
#define TASK_PRIO_HELLO 98
#define TASK_PRIO_WORLD 97
#define TASK_PRIO_METRO 99
```

Initialement, les tâches 'Hello' et 'World' sont bloquées. A chaque activation de la tâche métronome, le sémaphore est incrémenté de 2. A la première incrémentation, c'est la tâche 'Hello' qui prend la main puisque elle a une priorité supérieure à la tâche 'World'. La tâche 'Hello' décrémente le sémaphore et affiche 'Hello'. Ensuite, la tâche métronome reprend son exécution puisque la tache 'World' est bloquée et elle incrémente une seconde fois le sémaphore. La tâche 'World' est donc débloquée, elle décrémente le sémaphore, affiche 'World !' puis incrémente le sémaphore pour débloquer la tâche métronome qui va reprendre son exécution en décrémentant le sémaphore et en faisant une attente d'une seconde avant de relancer le même processus.

2.7 Question 2.7

Le fichier de statistiques de Xenomai lors du lancement de notre programme est le suivant :

```
root@devkit8600-xenomai:~# cat /proc/xenomai/stat
*CPU  PID    MSW        CSW        PF    STAT       %CPU  NAME
  0  0      0          513        0     00500080   99.8  ROOT
  0  1264   0          43         0     00300182    0.0  hello
  0  1265   0          85         0     00300182    0.0  world
  0  1266   0          84         0     00300184    0.0  metro
  0  0      0          53562      0     00000000    0.0  IRQ68: [timer]
```

FIGURE 1.5 – Fichier de statistiques de Xenomai

On retrouve bien nos 3 tâches 'hello', 'world' et 'metro'. On voit également que ces tâches ne subissent que très peu de changements de contexte (CSW) car ce sont des tâches temps-réel.

Pour le fichier du scheduler de Xenomai nous avons :

```
root@devkit8600-xenomai:~# cat /proc/xenomai/sched
CPU  PID    CLASS  PRI      TIMEOUT    TIMEBASE   STAT       NAME
  0  0      idle    -1      -          master     R          ROOT
  0  1264   rt      98      -          master     W          hello
  0  1265   rt      97      -          master     W          world
  0  1266   rt      99      625ms11us  master     D           metro
```

FIGURE 1.6 – Fichier du scheduler de Xenomai

Nous retrouvons bien les priorités de nos tâches que nous avons défini dans notre code. On voit également que nos trois tâches sont temps-réel via la colonne *CLASS* (rt). On voit également que la tâche 'ROOT' qui correspond à Linux est en mode 'idle', c'est à dire qu'elle a priorité la plus faible (-1). Dans la colonne *STAT*, on remarque que la tâche 'metro' est marquée de la lettre 'D' qui signifie 'Delayed', c'est à dire que la tâche est retardée sans aucune autre condition d'attente (wait d'une seconde). Les tâches 'hello' et 'metro' sont marquées de la lettre 'W' qui signifie que ces tâches sont en attente d'une ressource (sémaphore) et la tâche 'ROOT' est marquée de la lettre 'R' qui signifie 'Runnable', c'est à dire que la tâche est exécutable.

Exercice 3 : Latence

3.1 Question 3.1

```
1  #include <stdio.h>
   #include <native/task.h>
3  #include <analogy/os_facilities.h>
   #include <unistd.h>
5  #include <sys/mman.h>
   #include <native/sem.h>
7  #include <nucleus/timer.h>

9  #define TASK_PRIO 99
   #define TASK_MODE 0
11 #define TASK_STKSZ 0
   #define TAILLE 10000
13
   RT_TASK task_latence;
15
   void task_wait()
17 {
     int i;
19
```

```
  // boucle de 1 à 10000
  for(i = 0; i < TAILLE; i++)
  {
    // attente de 1ms
    rt_task_sleep(rt_timer_ns2ticks(1000000));
  }
}

int main()
{
  mlockall(MCL_CURRENT|MCL_FUTURE);
  rt_print_auto_init(1);
  int err1;

  err1 = rt_task_create(&task_latence, "wait", TASK_STKSZ, TASK_PRIO,
    TASK_MODE);

  if(!err1)
  {
    rt_task_start(&task_latence, &task_wait, NULL);
    rt_task_join(&task_latence);

    getchar();
  }


  return 0;
}
```

3.2 Question 3.2

```
#include <stdio.h>
#include <native/task.h>
#include <analogy/os_facilities.h>
#include <unistd.h>
#include <sys/mman.h>
#include <native/sem.h>
#include <nucleus/timer.h>


#define TASK_PRIO 99
#define TASK_MODE 0
#define TASK_STKSZ 0
#define TAILLE 10000

RT_TASK task_latence;
```

```
16
17 void task_wait()
18 {
19   int i;
20   RTIME moy = 0, min, max;
21   RTIME begin, end;
22
23   min = 9999999999999;
24   max = 0;
25
26
27   for(i = 0; i < TAILLE; i++)
28   {
29     begin = rt_timer_read();  // lecture du temps
30     rt_task_sleep(rt_timer_ns2ticks(1000000));
31     end = rt_timer_read();    // lecture du temps
32
33     if(min > end - begin)     // calcul du min
34       min = end - begin;
35
36     if(max < end - begin)     // calcul du max
37     {
38       max = end - begin;
39     }
40
41     moy += end - begin;  // calcul de la moyenne
42   }
43
44   moy = moy / TAILLE;
45
46   rt_printf("Moyenne :%llu \nMaximum : %llu \nMinimum : %llu\n", moy,
       max, min);
47
48 }
49
50
51 int main()
52 {
53   mlockall(MCL_CURRENT|MCL_FUTURE);
54   rt_print_auto_init(1);
55   int err1;
56
57   err1 = rt_task_create(&task_latence, "wait", TASK_STKSZ, TASK_PRIO,
       TASK_MODE);
58
59   if(!err1)
60   {
61     rt_task_start(&task_latence, &task_wait, NULL);
62     rt_task_join(&task_latence);
```

```
64      getchar();
    }
66
68  return 0;
}
```

Le programme précédent nous donne le résultat suivant :

```
root@devkit8600-xenomai:~# ./latence
Moyenne :1002756
Maximum : 1044920
Minimum : 1002440
```

FIGURE 1.7 – Résultat du programme *Latence* sans stress

3.3 Question 3.3

Ici, on charge le CPU via la commande *stress* avant de lancer notre programme :

```
^Croot@devkit8600-xenomai:~# ./latence
Moyenne :1005789
Maximum : 1030760
Minimum : 1003560
```

FIGURE 1.8 – Résultat du programme *Latence* avec stress

On remarque que les résultats avec charge CPU et sans charge CPU sont pratiquement les mêmes ce qui veut dire que la charge n'a aucune influence sur les tâches exécutées. Ceci est logique puisque le programme est temps-réel donc une charge CPU ne ralentira que les programmes non temps-réel qui sont toujours reliés à Linux (ROOT) comme on a pu le constater dans le TP précédent. Ici, les tâches temps-réel ont la plus grande priorité sur le CPU.