Bordeaux INP
ENSEIRB
MATMECA

Rapport Projet

Raspberry Pi 3 : Solutions temps réel / Xenomai

Réalisé par : **HARCHAOUI TAWFIK**
AIT MOUSSA Ilyass
EL KATI Mohamed-Islam
EL ALLAM Oussama

Supervisé par : **Mr. Patrice KADIONIK**

SE 2019/2020
ENSEIRB-MATMECA

LE 27/01/2020 à Talence

Sommaire

INTRODUCTION

L'objectif de ce projet est de comprendre plus en détail ce que propose un noyau Temps Réel dur. En effet, ce type de noyau est créé pour un traitement Temps Réel dur vu que cela impacte réellement les performances globales du système, et ne garantit que les temps de latence. Dans ce projet, on étudiera **Xenomai** qui comprend un noyau Temps Réel dur.

On utilise comme support la carte **Raspberry PI3**. Et on réalisera dans la suite dans un premier temps des mesures de latence afin de comparer entre le Linux standard et Xenomai, ensuite on réalisera quelques programmes en C pour tester l'API NATIVE et l'API POSIX.

1- Xenomai - temps réel

Les systèmes temps réel sont caractérisés en s'appuyant sur deux catégories : **temps réel dur** et **temps réel souple.** Le noyau Linux standard n'est pas capable de supporter des programmes en temps réel dur, néanmoins l'adjonction d'une extension comme Xenomai lui permet de rentrer dans cette catégorie. [1]

L'intérêt de l'utilisation de la RaspBerry PI 3 comme support réside dans le fait qu'elle regroupe bon nombre d'éléments habituellement disponibles que sur des plateformes plus lourdes (puissance de calcul, périphériques entrée/sortie, environnement logiciel libre) sur un format facilement embarquante et à un prix défiant toute concurrence.

On commence d'abord par l'installation de Xenomai en appliquant un patch :

D'abord, on télécharge puis on décompresse l'archive xenomai (la dernière version xenomai-3.0.2). Ensuite, on regarde pour quelles versions du kernel le patch *ipipe* est disponible. Après quelques tentatives, on a enfin réussi cette étape.

La prochaine étape sera les mesures de temps de latence.

2- Mesures des temps de latence

Les mesures du temps de latence consistent à créer un processus Temps Réel périodique de période T. Ce temps de latence peut être mesuré avec les utilitaires **cyclictest** et **latency** fournis par Xenomai. cyclictest est un outil standard développé pour le projet PREEMPT-RT qui a été repris pour Xenomai. A titre indicatif, on donnera les mesures de temps de latence dans le cas où le noyau est non stressé et dans le cas où le noyau est stressé. Pour stresser le noyau, on utilisera l'utilitaire *stress.*
On commence par dévalider le *throttling :*

```
# echo 1 > /proc/sys/kernel/sched_rt_runtime_us
```

Puis on dévalide l'anticipation sur la latence minimale de Xenomai :

```
# echo 0 > /proc/xenomai/latency
```

❖ Cas 1 : Linux Standard

On commence par le cas où le noyau n'est pas stressé, et on lance *cyclictest* pour générer un processus Temps Réel périodique dans l'espace utilisateur de plus forte priorité 99 et de période 5000 µs :

```
# cyclictest -n -p 99 -i 5000
```

Ensuite on stresse le noyau via *stress* et lance cyclictest à nouveau :

```
# stress -c 50 -i 50 &
```

```
# cyclictest –n –p 99 –i 5000
```

On obtient les résultats suivants :

	Linux standard non stressé	Linux standard stressé
cyclictest	115us	230us

❖ Cas 2 : Noyau Xenomai non stressé:

On commence nos calculs d'abord via cyclictest :

```
# cyclictest –n –p 99 –i 5000
```

Après, on utilise l'outil Xenomai *latency* dans 3 modes différents :

On utilisera *latency* pour générer un processus Temps Réel périodique dans l'espace utilisateur de plus forte priorité 99 et de période 5000 µs par la commande :

```
# /usr/xenomai/bin/latency –t0 –p 5000
```

Puis, on utilisera *latency* pour générer un processus Temps Réel périodique dans l'espace noyau de plus forte priorité 99 et de période 5000 µs par la commande :

```
# /usr/xenomai/bin/latency –t1 –p 5000
```

Après, on passe au troisième mode de *latency* (le mode timer IRQ) via la commande :

```
# /usr/xenomai/bin/latency –t2 –p 5000
```

❖ Cas 3 : Noyau Xenomai stressé :

On stresse d'abord le noyau par la commande :

```
# stress –c 50 –i 50 &
```

On commence nos calculs via cyclictest :

```
# cyclictest –n –p 99 –i 5000
```

Après, on utilise l'outil Xenomai *latency* dans 3 modes différents :

```
# /usr/xenomai/bin/latency –t0 –p 5000
# /usr/xenomai/bin/latency –t1 –p 5000
# /usr/xenomai/bin/latency –t2 –p 5000
```

Le tableau ci-dessous résume les résultats obtenus :

	Noyau Xenomai non stressé	Noyau Xenomai non stressé
cyclictest	15us	17us
Latency-t0	14.983us	15.824us
Latency-t1	6.008us	8.447us
Latency-t2	3.976us	4.154us

3- L'API Native

Xenomai fournit une API très complète avec les fonctionnalités temps réel suivantes :

- Tâches avec 99 niveaux de priorité, round robin optionnel…
- Files de messages ;
- Allocation dynamique de mémoire spécifique RT ;
- Sémaphores;
- watchdogs.
- Timers.
- Mutexes.
- …

Cette API, dite « native », est complétée par des skins. Ce sont des API d'autres RTOS (pSos, VxWorks…) qui encapsulent des appels natifs.

Les skins sont destinés au portage d'applications existantes ou à du prototypage. [2]

Dans la suite on crée des programmes en C, en utilisant les bibliothèques suivantes :

```
#include <alchemy/task.h>
#include <alchemy/timer.h>
#include <alchemy/sem.h>
#include  <trank/rtdk.h>
```

- **Chenillard :**

  Dans cette partie, on crée le programme classique chenillard qui est un mouvement lumineux qui se produit en allumant et éteignant successivement une série de LED connectées à la carte Raspberry Pi3. On sert des fichiers '**bcm2835.h**' et '**bcm2835.c**' afin de créer nos fonctions qui permettent de modifier l'état des LEDs et du bouton poussoir connectés à la carte.

  On commence par créer les fonctions suivantes :

```
void BSP_init_Led() ;
```

  ⇨ Cette fonction permet d'initialiser les 6 LEDs.

```
void BSP_set_LED(int i) ;
```

  ⇨ Cette fonction permet d'allumer la LED dont son numéro est donné en paramètre.

```
void BSP_clr_LED(int i) ;
```

  ⇨ Cette fonction éteint la LED dont son numéro est donné en paramètre.

  Enfin, on combine ces fonctions pour créer notre fonction chenillard :

```
void Chenillard_LED()
{
    int i;
    for(i = 1; i < 7; i++)
    {
        BSP_set_LED(i);
```

```
            bcm2835_delay(500);
            BSP_clr_LED(i);
      }
}
```

Après dans le main, on initialise les LEDs et dans une boucle infinie on appelle notre fonction.
Ensuite dans un terminal, on compile notre application chenillard grâce au shell script go :

```
Host% ./go
```

Puis, grâce au shell script goinstall on copie notre application dans le **tftpboot.**

```
Host% ./goinstall
```

On vérifie le bon fonctionnement du programme.

❖ **Bouton poussoir :**

Dans cette partie, on crée le programme Bouton poussoir qui permet d'allumer la LED2. Dans ce programme on va modifier le fichier précédent car on va s'en servir des fonctions qui permettent d'initialiser, allumer et éteindre les LEDs. La fonction qu'on va ajouter est la suivante :

```
      void Bouton()
{
  //init bouton
  bcm2835_gpio_fsel(Bp , BCM2835_GPIO_FSEL_INPT);
  uint8_t value = bcm2835_gpio_lev(Bp);
  if(value == LOW)
    {
      BSP_set_LED(2);
    }else{
     BSP_clr_LED(2);
    }
}
```

Et de même, on appelle cette fonction dans une boucle infinie dans le main.

Après, on vérifie le bon fonctionnement du programme.

❖ Concurrence :

Le but de cette partie est de gérer l'accès exclusif à une ressource partagée, qui sera représentée par la led 1. Dans la tâche principale on crée un sémaphore binaire sem_desc et on lance les deux taches task1 et task2, qui sont bloquées sur le sémaphore sem_desc et on libère le sémaphore à la fin. Sachant que les deux tâches sont bloquées sur le sémaphore, et à leur libération on change l'état de la led1.

Une tâche qui souhaite acquérir une ressource doit faire appel à la fonction ***rt_sem_p(&sem_desc,TM_INFINITE)*** qui joue le rôle d'un 'wait'. Si le sémaphore est disponible (sa valeur est supérieure à 0), sa valeur est définie sur 0, et la tâche continue son exécution. Si la valeur du sémaphore est 0, la tâche effectuant une attente sur le sémaphore est placée dans une liste d'attente.

Une tâche libère un sémaphore en effectuant un appel à la fonction ***rt_sem_v(&sem_desc).*** Si aucune tâche n'attend le sémaphore, la valeur du sémaphore est simplement définie sur 1.

On déclare au début notre sémaphore via **RT_SEM** et les tâches via **RT_TASK.** Dans le main on crée notre sémaphore grâce à :
rt_sem_create(&sem_desc,"MySemaphore",SEM_INIT,SEM_MODE);

Puis on crée les tâches via :

```
     if (rt_task_spawn( &task_desc0, /* task descriptor */
                    "my task 0", /* name */
                     0           /* 0 = default stack size */,
                     99          /* priority */,
                     T_JOINABLE , /* needed to call rt_task_join after */
                     &task,      /* entry point (function pointer) */
                     NULL        /* function argument */ )!=0)
{
  printf("rt_task_spawn error\n");
  //return 1;
}

if (rt_task_spawn( &task_desc1, /* task descriptor */
                    "my task 1", /* name */
                     0           /* 0 = default stack size */,
            99          /* priority */,
                T_JOINABLE , /* needed to call rt_task_join after */
                &task1,      /* entry point (function pointer) */
```

```
            NULL        /* function argument */ )!=0)
  {
    printf("rt_task_spawn error\n");
    //return 1;
  }
```

Et enfin, on appelle les deux fonctions suivantes :

```
  rt_task_join(&task_desc0);
  rt_task_join(&task_desc1);
```

On vérifie le bon fonctionnement du programme.

❖ **RDV :**

Le but de ce TP est de synchroniser 2 tâches à un instant donné dans l'exécution de leur code. On lance les deux taches task1 et task2. Task1 est bloquée le sémaphore binaire sem_desc, et task2 donne un rdv à la tâche task1 par libération du sémaphore binaire.

```
  static void task(void *arg)
{
       // Now, wait for a semaphore unit...
         rt_sem_p(&sem_desc,TM_INFINITE);
           printf("Task1\n");
         /* then release it. */
         rt_sem_v(&sem_desc);
}
static void task1(void *arg)
{
       rt_task_sleep(1000000000);
     /* Now, wait for a semaphore unit... */
       rt_sem_v(&sem_desc);
         //rt_sem_p(&sem_desc,TM_INFINITE);
         printf("Task2 RDV avec Task1\n");
}
```

Dans task1, la fonction ***rt_sem_p(&sem_desc,TM_INFINITE)*** nous permet de la bloquer sur la sémaphore binaire sem. Puis dans task2, on reste en attente grâce à ***rt_task_sleep(1000000000)*** ensuite on libère le sémaphore via la fonction ***rt_sem_v(&sem_desc)*** et on donne RDV à la tâche task1.

On vérifie le bon fonctionnement du programme.

- **Gestion du temps :**

Le but de ce TP est de réaliser un dispatching d'un travail par une tache. On va créer trois tâches, puis va lancer les trois tâches *task1*, *task2* et *task3*, *et* chacune va écrire un numéro (1 pour task1, 2 pour task2 et 3 pour task3).

e but de ce TP est de synchroniser 2 tâches à un instant donné dans l'exécution de leur code. On lance les deux taches task1 et task2. Task1 est bloquée le sémaphore binaire sem_desc, et task2 donne un rdv à la tâche task1 par libération du sémaphore binaire.

```
  void task1(void *arg)
{
  while(1)
    {
      // bcm2835_gpio_fsel(LED2, BCM2835_GPIO_FSEL_OUTP);
       rt_task_sleep(1000000000);
        printf("task 1\n");
    }
}
 void task2(void *arg)
{
  // nsleep(1000);
 while(1)
    {
      // bcm2835_gpio_fsel(LED2, BCM2835_GPIO_FSEL_OUTP);
       rt_task_sleep(5000000000);
        printf("task 2\n");
   }
}
 void task3(void *arg)
{
  // nsleep(1000);
 while(1)
    {
      // bcm2835_gpio_fsel(LED2, BCM2835_GPIO_FSEL_OUTP);
       rt_task_sleep(10000000000);
        printf("task 3\n");
    }
}
```

❖ Problème des philosophes :

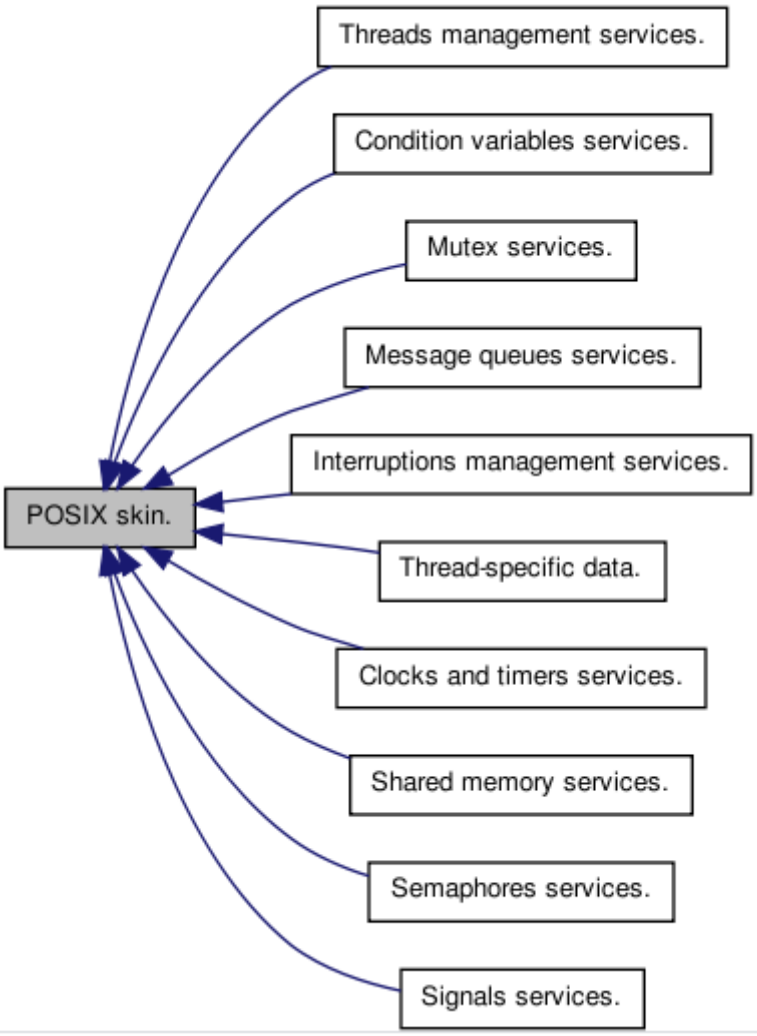
Le but de ce TP est de traiter le problème des philosophes. C'est un problème classique sur le partage de ressources en informatique et particulièrement pour les questions d'ordonnancement des processus. On considère cinq philosophes qui se trouvent autour d'une table, et chacun des philosophes a devant lui un plat de spaghettis, et à gauche de chaque assiette se trouve une fourchette. Sachant qu'un philosophe n'a que deux états possibles : penser pendant un temps déterminé et manger. Il faut prendre en considération des contraintes extérieurs qui s'imposent à cette situation : Pour manger, un philosophe a besoin de deux fourchettes (celle qui se trouve à sa gauche et celle qui se trouve à la gauche de son voisin de droite), et si un philosophe n'arrive pas à s'emparer d'une fourchette, il se met à penser pendant une durée déterminée en attendant de renouveler sa tentative.

On utilisera les sémaphores binaires pour résoudre ce problème. Les cinq première LEDs de la carte représentent les cinq philosophes, et lorsqu'un philosophe mange la LED correspondante va s'allumer et elle va s'éteindre lorsqu'il va réfléchir.

On crée les cinq sémaphores, puis on crée les cinq tâches. Chaque tâche va gérer les deux états possibles du philosophe correspondant. Et chaque sémaphore gère la disponibilité d'une fourchette.

4- L'API POSIX

Le skin Xenomai POSIX est une implémentation d'un petit sous-ensemble de la spécification Single Unix sur le noyau RTOS générique Xenomai.



Dans cette partie, on essaiera d'utiliser l'API Posix au lieu de l'API Native et réaliser les mêmes programmes qu'on a déjà réalisé avec l'API NATIVE.
La différence se manifeste dans les bibliothèques qu'on va inclure dans nos programmes, donc aussi les fonctions qui permettront de gérer les threads et les sémaphores seront différents.
Les bibliothèques qu'on va inclure sont :

```
#include <pthread.h>
#include <semaphore.h>
```

La bibliothèque **pthread.h** nous permet de gérer les threads, donc dans nos programmes précédents on doit modifier les fonctions permettant de gérer les taches :

- **Int pthread_create (pthread_t *ptid_r, const pthread_attr_t *attr, void *(*start)(void *), void *arg) ;**

  ⇨ Cette fonction nous permet de créer un thread. Par exemple dans le programme du RDV :

```
pthread_t task_desc0,task_desc1;
pthread_create(&task_desc0, NULL,func[0],NULL);
pthread_create(&task_desc1, NULL,func[1],NULL);
```

- **Int pthread_join (pthread_t thread, void **retval) ;**

  ⇨ Cette fonction attend la fin d'un thread spécifique.

```
pthread_join(task_desc0,NULL);
pthread_join(task_desc1,NULL);
```

La bibliothèque **semaphore.h** nous permet d'utiliser les sémaphores, et de même on doit modifier dans nos programmes les fonctions permettant de gérer les sémaphores.

- **Int sem_init(sem_t *sem, int pshared, unsigned int value) ;**

  ⇨ Cette fonction nous permet d'initialiser nos sémaphores.

```
sem_t sem_desc;
sem_init(&sem_desc, 0,0);
```

- **Int sem_post(sem_t *sem) ;**

  ⇨ Cette fonction nous permet de libérer le sémaphore.

```
sem_post(&sem_desc);
```

- **Int sem_wait(sem_t *sem) ;**

  ⇨ Cette fonction nous permet d'attendre la libération du sémaphore.

```
sem_wait(&sem_desc);
```

Ensuite on a vérifié le bon fonctionnement de tous nos programmes.

5- Conclusion

Ce projet nous a permis de comprendre la notion du temps réel dur, et l'utilité du noyau Xenomai. Et on a pu créer nos programmes avec les deux API Native et Posix.
Aussi ce projet nous permet de voir l'importance du travail en groupe.

6- Réferences

[1] : Solutions temps réel sous Linux – Christophe Blaess – 2ème édition.

[2] : https://dchabal.developpez.com/tutoriels/linux/xenomai/?page=page_5#LV-E