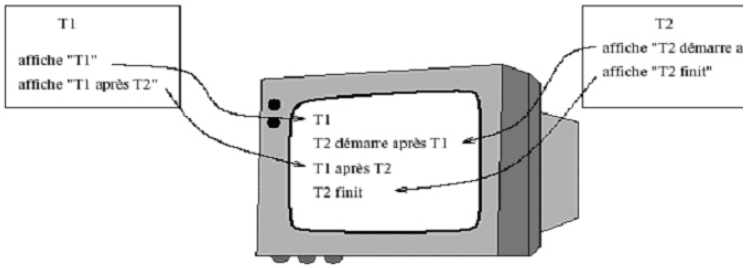
Systemes temps réel
IN412 - TP 2

David Rajendra
Schapira Boris

Exercice 1

Cahier des charges : Développer 2 tâches qui assurent l'ordre d'éxécution suivant :



On met en place 2 sémaphores, contenus dans un tableau `static SEM semaphores[2]`. Ces derniers vont permettre d'éxécuter les deux tâches dans l'ordre.

Les deux sémaphores sont initialisés à 0. La deuxième tâche `TASK_2` est bloqué par le deuxième sémaphore `semaphore[1]` ainsi, la première tâche `TASK_1` peut s'éxécuter en premier et afficher `[TASK 1] Affiche T1` .
Puis, il libère `semaphore[1]` et attend le premier sémaphore `semaphore[0]`.
`TASK_2` affiche `[TASK 2] T2 demarre après T1` et permet l'éxécution de `TASK_2` par la libération de `semaphore[0]`.
`TASK_1` affiche `[TASK 1] T1 après T2` et libère `semaphore[1]`, ce qui permet à `TASK_2` d'afficher `[TASK 2] T2 finit`.
Nous avons au final :

```
[TASK 1] Affiche T1
[TASK 2] T2 demarre après T1
[TASK 1] T1 après T2
[TASK 2] T2 finit
```

Nous avons respecté le cahier des charges.

Voici le code source :

```
#include <linux/module.h>
MODULE_LICENSE("GPL");
#include <asm/io.h>

#include <asm/rtai.h>
#include <rtai_sched.h>
#include <rtai_sem.h>

#define PRIORITE    1
#define STACK_SIZE  2000
#define PERIODE     1000000000    //  1 s
#define TICK_PERIOD 1000000    //  1 ms
#define N_BOUCLE    10

#define TASK_1 1
#define TASK_2 2

static RT_TASK mes_taches[2];
static SEM semaphores[2];

void my_task(int arg)
{
if(arg==TASK_1)
{
printk("[TASK %d] Affiche T1\n",arg);
rt_sem_signal(&semaphores[1]);
rt_sem_wait(&semaphores[0]);
printk("[TASK %d] T1 après T2\n",arg);
rt_sem_signal(&semaphores[1]);
}else
{
rt_sem_wait(&semaphores[1]);
printk("[TASK %d] T2 demarre après T1\n",arg);
rt_sem_signal(&semaphores[0]);
rt_sem_wait(&semaphores[1]);
printk("[TASK %d] T2 finit\n",arg);
}
}
```

```
int init_module(void) {

int ierr;

rt_set_oneshot_mode();

rt_sem_init(&semaphores[0],0);
rt_sem_init(&semaphores[1],0);

ierr = rt_task_init(&mes_taches[0], my_task, TASK_1, STACK_SIZE,
PRIORITE, 0, 0);
printk("[tache %d] cree code retour %d par programme %s\n",TASK_1,
ierr,__FILE__);
ierr |= rt_task_init(&mes_taches[1], my_task, TASK_2, STACK_SIZE,
PRIORITE, 0, 0);
printk("[tache %d] cree code retour %d par programme %s\n",TASK_2,
ierr,__FILE__);

if (!ierr)
{
start_rt_timer(nano2count(TICK_PERIOD));
rt_task_resume(&mes_taches[0]);
rt_task_resume(&mes_taches[1]);
}
return ierr;
}

void cleanup_module(void) {

stop_rt_timer();

rt_task_delete(&mes_taches[0]);
rt_task_delete(&mes_taches[1]);

rt_sem_delete(&semaphores[0]);
rt_sem_delete(&semaphores[1]);
}
```

Exercice 2

Cahier des charges : mettre en place une tâche de type chien de garde `my_watchdog` qui vérifie qu'un compteur `i`,simulé par la tâche `my_compteur` est régulièrement incrémenté.

Nous avons implémenté le problème de la manière suivante :

Il y a un sémaphore `semaphore` qui va permettre à `my_watchdog` de contrôler si `my_compteur` s'est bien éxécutée. On utilise un `rt_sem_wait_timed` afin que `my_watchdog` ne soit pas bloquée si le compteur manque à son devoir.
En effet, `my_compteur` est une tâche périodique qui contient une boucle infinie. Cette dernière a pour but d'incrémenter le compteur `i` ainsi que la valeur de `semaphore` à chaque période grâce à `rt_task_wait_period()`.
`my_watchdog` prend d'abord le temps(`before=rt_get_time_ns()`), teste si `my_compteur` s'est éxécutée puis reprend le temps afin de calculer la différence.
Si cette différence est supérieur à 1.3s (`ERR_TIME_1`) ou 1s (`ERR_TIME_2`), il le signale par des messages d'erreur.
Dans le cas où `my_compteur` est trop long (si `semaphore` est à 0 plus de `2*ERR_TIME_1`) ou s'est bloquée, `rt_sem_wait_timed` permet à `my_watchdog` de ne pas attendre indéfiniment et éxécute la suite du code. Il y aura bien sûr génération d'un message d'erreur.

Voici le code source :

```
#include <linux/module.h>
MODULE_LICENSE("GPL");
#include <asm/io.h>

#include <asm/rtai.h>
#include <rtai_sched.h>
#include <rtai_sem.h>

#define PRIORITE    1
#define STACK_SIZE  2000
#define PERIODE     1500000000    //  1.5s
#define TICK_PERIOD 1000000    //  1 ms
#define N_BOUCLE    10

#define ERR_TIME_1 1300000000 // 1.3s
#define ERR_TIME_2 1000000000 // 1s

static RT_TASK mes_taches[2];
static SEM semaphore;

void my_watchdog(int arg)
{
RTIME before,diff,after;

printk("[watchdog] Lancement Chien de Garde\n");
while(1)
{
before=rt_get_time_ns();

rt_sem_wait_timed(&semaphore,2*nano2count(ERR_TIME_1));

after=rt_get_time_ns();
diff=after-before;
printk("[watchdog] diff = %llu\n",diff);

if(diff>ERR_TIME_1)
printk("[watchdog] ERREUR : (1) Compteur non incremente
 depuis %llu\n",diff);
```

```
diff=rt_get_time_ns()-after;
if(diff>ERR_TIME_2)
printk("[watchdog] ERREUR : (2) non respect echeance
 (%llu ns)\n",diff);
}
}
void my_compteur(int arg)
{
int i=0;
printk("[compteur] Lancement du compteur\n");
while(1)
{
printk("[compteur] cpt=%d\n",++i);

rt_sem_signal(&semaphore);
rt_task_wait_period();
}
}

int init_module(void)
{

int ierr;

RTIME now;

rt_set_oneshot_mode();

rt_sem_init(&semaphore,1);

ierr = rt_task_init(&mes_taches[0], my_watchdog, 0,
STACK_SIZE, PRIORITE+20, 0, 0);
printk("[watchdog] cree code retour %d par programme
%s\n",ierr,__FILE__);
ierr = rt_task_init(&mes_taches[1], my_compteur, 0,
STACK_SIZE, PRIORITE, 0, 0);
printk("[compteur] cree code retour %d par programme
%s\n",ierr,__FILE__);

if (!ierr)
```

```
{
start_rt_timer(nano2count(TICK_PERIOD));
rt_task_resume(&mes_taches[0]);

now = rt_get_time();

rt_task_make_periodic(&mes_taches[1], now+nano2count(PERIODE),
nano2count(PERIODE));
}
return ierr;
}

void cleanup_module(void)
{

stop_rt_timer();

rt_task_delete(&mes_taches[0]);
rt_task_delete(&mes_taches[1]);

rt_sem_delete(&semaphore);
}
```

Exercice 3

Cahier des charges : Reprendre l'exercice 2 en remplaçant l'affichage console par un affichage sur shell standard grâce à un porcessus utlisateur relié aux tâches temps réel par une fifo.

On remplace tous les `printk` par un appel à la procédure `rtf_put` qui va charger le message dans une fifo. On visionnera cette dernière dans un shell standard par la commande `cat /dev/rtf0`.
On met en place une nouvelle structure :

```
typedef struct _TX_STRUCT
{
unsigned char msg;
long long unsigned data;
}tx_struct;
```

ainsi que qu'une convention de message :

```
#define FIFO_MSG_CPT_LAUNCH 0
#define FIFO_MSG_WDT_LAUNCH 1
#define FIFO_MSG_CPT_ERROR 2
#define FIFO_MSG_WDT_ERROR 3
#define FIFO_MSG_EXIT 255
```

On crée un `tx_struct data` et la fifo est créée dans `init_module` par `rtf_create`. Dès qu'un message doit être affiché, on met l'information dans `data.msg` et le temps de réponse dans `data.data` puis on envoie `data` dans la fifo par `rtf_put(1,&data,sizeof(tx_struct))`
Pour une utilisation plus simple, nous avons implémenté un petit programme pour que l'utilisateur ait un message plus explicite qui s'affiche. Ce programme inteprète ce qu'il y a dans la fifo tant qu'il n'y a pas le message de fin.

Voici le code source :

```
#include <linux/module.h>
MODULE_LICENSE("GPL");
#include <asm/io.h>

#include <asm/rtai.h>
#include <rtai_sched.h>
#include <rtai_sem.h>
#include <rtai_fifos.h>

#define PRIORITE    1
#define STACK_SIZE  2000
#define PERIODE     1500000000    //  1.5s
#define TICK_PERIOD 1000000    //  1 ms
#define N_BOUCLE    10

#define ERR_TIME_1 1300000000 // 1.3s
#define ERR_TIME_2 1000000000 // 1s

#define FIFO_MSG_CPT_LAUNCH 0
#define FIFO_MSG_WDT_LAUNCH 1
#define FIFO_MSG_CPT_ERROR 2
#define FIFO_MSG_WDT_ERROR 3
#define FIFO_MSG_EXIT 255

typedef struct _TX_STRUCT
{
unsigned char msg;
long long unsigned data;
}tx_struct;

static RT_TASK mes_taches[2];
static SEM semaphore;
static tx_struct data;

void my_watchdog(int arg)
{
RTIME before,diff,after;

data.msg=FIFO_MSG_WDT_LAUNCH;
```

```
data.data=0;
rtf_put(1,&data,sizeof(tx_struct));
while(1)
{
before=rt_get_time_ns();

rt_sem_wait_timed(&semaphore,2*nano2count(ERR_TIME_1));

after=rt_get_time_ns();
diff=after-before;

if(diff>ERR_TIME_1)
{
data.msg=FIFO_MSG_CPT_ERROR;
data.data=diff;
rtf_put(1,&data,sizeof(tx_struct));
}

diff=rt_get_time_ns()-after;
if(diff>ERR_TIME_2)
{
data.msg=FIFO_MSG_WDT_ERROR;
data.data=diff;
rtf_put(1,&data,sizeof(tx_struct));
}
}
}
void my_compteur(int arg)
{
int i=0;
while(1)
{
i++;
rt_sem_signal(&semaphore);
rt_task_wait_period();
}
}

int init_module(void)
{
```

```
int ierr;

RTIME now;

rt_set_oneshot_mode();

rt_sem_init(&semaphore,1);
rtf_create(1,200);

ierr = rt_task_init(&mes_taches[0], my_watchdog, 0,
STACK_SIZE, PRIORITE+20, 0, 0);
ierr = rt_task_init(&mes_taches[1], my_compteur, 0,
STACK_SIZE, PRIORITE, 0, 0);

if (!ierr)
{
start_rt_timer(nano2count(TICK_PERIOD));
rt_task_resume(&mes_taches[0]);

now = rt_get_time();

rt_task_make_periodic(&mes_taches[1], now+nano2count(PERIODE),
nano2count(PERIODE));
}
return ierr;
}

void cleanup_module(void)
{

stop_rt_timer();

data.msg=FIFO_MSG_EXIT;
data.data=0;
rtf_put(1,&data,sizeof(tx_struct));

rt_task_delete(&mes_taches[0]);
rt_task_delete(&mes_taches[1]);
```

```
rt_sem_delete(&semaphore);
rtf_destroy(1);
}
```

Voici le code source du programme utilisateur :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define FIFO_MSG_CPT_LAUNCH 0
#define FIFO_MSG_WDT_LAUNCH 1
#define FIFO_MSG_CPT_ERROR 2
#define FIFO_MSG_WDT_ERROR 3
#define FIFO_MSG_EXIT 255

typedef struct _TX_STRUCT
{
unsigned char msg;
long long unsigned data;
}tx_struct;

int main()
{
int fh;
tx_struct data;

printf("Appuyer sur ENTREE une fois la fifo RTAI cree.\n");
getchar();
fh=open("/dev/rtf1",O_RDONLY);

do
{
read(fh,&data,sizeof(tx_struct));
switch(data.msg)
{
case FIFO_MSG_CPT_LAUNCH:
printf("Création du compteur\n");
break;
case FIFO_MSG_WDT_LAUNCH:
```

```
printf("Création du Chien de Garde\n");
break;
case FIFO_MSG_CPT_ERROR:
printf("[ERREUR] Le compteur n'a plus compté depuis %llu ns\n",
data.data);
break;
case FIFO_MSG_WDT_ERROR:
printf("[ERREUR] Le chien de garde à prit %llu ns\n",data.data);
break;
default:
break;
}
}while(data.msg!=FIFO_MSG_EXIT);

close(fh);
return 0;
}
```