Université Polytechnique Hauts-de-France

INSA Hauts-de-France

Année 2024 - 2025

5A ESE - O9MA332

Architecture Temps Réel Pour L'embarqué

TD n°2 : Exemples d'applications temps réel avec Xenomai dans l'espace utilisateur

Marwane AYAIDA (marwane.ayaida@uphf.fr)

Comme tous les systèmes d'exploitation multi-tâches, *Xenomai* permet à plusieurs tâches de s'exécuter à tour de rôle. Ces tâches s'occupent chacune d'une activité bien précise avec des contraintes temps réel. Ces dernières peuvent être divisées en plusieurs *threads* indépendants. Les threads ont besoin souvent de communiquer entre eux et d'échanger des données pour le bon fonctionnement du système. Dans ce TD, nous allons voir comment on peut créer un processus sur *Xenomai* dans le domaine secondaire (*Linux*) et comment on peut faire pour communiquer et synchroniser des tâches.
Dans ce TD, nous allons tout d'abord commencer par mettre en oeuvre des applications simples dans l'espace utilisateur et puis on fera de même dans le TD suivant dans l'espace noyau.

1 Préparation de l'environnement

Nous avons précédemment installé Xenomai et exécuté quelques programmes de tests qui nous permettaient d'avoir un premier aperçu des performances de ce système. Nous allons à présent commencer à créer nos propres applications en utilisant l'API de *Xenomai*.
Avant de commencer à compiler et exécuter des applications sur Xenomai, il faudrait toujours passer en mode super-utilisateur (root) sur le Raspberry :

```
1 ~$ sudo su
```

Maintenant que vous êtes en mode super-utilisateur (root), ouvrez le fichier "*.bashrc*" du root et ajouter cette ligne à la fin :

```
1 # nano /root/.bashrc
2 # Ajoutez cette ligne a la fin du fichier
3 export LD_LIBRARY_PATH=/usr/xenomai/lib
```

Il faut maintenant que vous rouvrez de nouveau le terminal en mode supe-utilisateur ou que vous chargiez le fichier "*.bashrc*" pour prendre en compte cette ligne en plus :

```
1 # source /root/.bashrc
```

Pour vérifier que cela a été bien pris en compte, lancez cette commande :

```
1 # echo $LD_LIBRARY_PATH
2 /usr/xenomai/lib/
```

Dorénavant, toutes les commandes doivent être exécutées en mode super-utilisateur (root).
Passons maintenant à nos premiers exemples au niveau Linux.

2 Implémentation d'applications simples avec Xenomai dans l'espace utilisateur

Comme nous l'avons évoqué auparavant, le développement sous Xenomai s'effectue en utilisant des processus tout à fait classiques de l'espace utilisateur, au sein desquels on démarre des threads temps réel qui sont ordonnancés par Xenomai, sauf lorsqu'ils exécutent des appels système durant lesquels c'est l'ordonnanceur Linux qui reprend le contrôle, ce que l'on nomme « mode secondaire ».
Les deux difficultés que peut rencontrer le développeur Xenomai sont :

- la connaissance de l'API temps réel : si Xenomai propose des skins permettant d'incorporer facilement du code écrit pour d'autres systèmes d'exploitation, il est néanmoins conseillé d'utiliser, pour les développements spécifiques, l'API ***Alchemy*** dans *Xenomai 3*, qui a remplacé l'API *Native* dans *Xenomai 2* car elle est en principe la plus efficace ;
- la recherche des commutations du mode primaire au mode secondaire qui se produisent lors d'appels système, parfois dissimulés dans des fonctions de bibliothèque *C*.

À priori, le point le plus compliqué est le second, car il entraîne un passage d'un ordonnancement temps réel strict Xenomai (plus prioritaire que les traitements d'interruptions de Linux) à un temps réel souple Linux. Les conséquences ne sont pas directement visibles, mais la prédictibilité des temps de réponse est amoindrie. En outre, la détection de ces basculements est difficile du fait que certaines fonctions de bibliothèques n'invoquent pas immédiatement les appels système sous-jacents mais les diffèrent pour optimiser les temps de traitement en regroupant plusieurs opérations. On peut penser à *malloc()*, mais aussi à *fprintf()*, par exemple. Heureusement, il existe des outils de mise au point intégrés dans Xenomai qui nous permettront de détecter ces basculements en mode secondaire.
La documentation de l'API ***Alchemy*** de *Xenomai* est disponible sur le site wiki du projet (*https ://gitlab.denx.de/Xenomai/xenomai/wikis/home*) accessible depuis le site principal *https ://xenomai.org*, mais également dans le répertoire d'installation : */usr/xenomai/share/doc/xenomai/html/api/*. Pour peu que l'option « *–enable-doc-build* » ait été fournie à la commande "*./configure*" lors de la compilation des bibliothèques et outils de *Xenomai*. Il s'agit d'une documentation générée à l'aide de l'outil *Doxygen*. Je ne présenterai ici que les fonctions essentielles à l'écriture de tâches, en vous laissant vous reporter à ces sources pour avoir plus de précisions sur les routines utilisées.
L'écriture d'application avec Xenomai nécessite l'inclusion de certains des fichiers

d'en-tête résumés dans le tableau suivant 1.

TABLE 1 – Fichiers d'en-tête de l'API Alchemy

Fichiers	***Rôles***
<alchemy/alarm.h>	Programmation d'alarme pour activer un thread temps réel de manière différée et/ou périodique.
<alchemy/buffer.h>	Communication *Fifo* par blocs mémoire entre threads ou processus.
<alchemy/cond.h>	Synchronisation sur des variables conditions proches des *pthread_cond_t*.
<alchemy/event.h>	Notification d'événements en utilisant des flags programmables.
<alchemy/heap.h>	Allocation de mémoire par tas, éventuellement partagés entre processus.
<alchemy/intr.h>	Gestion des interruptions depuis l'espace utilisateur.
<alchemy/misc.h>	Autorisation d'accès aux ports d'entrées-sorties.
<alchemy/mutex.h>	Synchronisation sur des mutex proches des *pthread_mutex_t*.
<alchemy/pipe.h>	Communication par tubes nommés avec les processus de l'espace Linux.
<alchemy/queue.h>	Transmission de messages sans copie (buffers partagés).
<alchemy/sem.h>	Synchronisation par sémaphores.
<alchemy/task.h>	Fonctions et types permettant de gérer les tâches : création, configuration, destruction, mise en sommeil, envoi de message, etc.
<alchemy/timer.h>	Configuration, lecture et conversion de l'heure système.

L'utilisation de la fonction *mlockall()* permet de verrouiller en mémoire physique les pages utilisées par le processus, évitant ainsi tous les comportements imprévisibles liés à la gestion de la mémoire virtuelle. Avec *Xenomai*, cet appel est indispensable, car le déclenchement d'une faute de page (lors d'un accès à une zone qui n'est pas encore disponible) entraînerait un retour en mode secondaire. Ceci se produirait de manière totalement imprévisible puisque par défaut, c'est à l'instant de la première utilisation d'une page mémoire qu'elle est attribuée par le noyau, et non lors de son allocation initiale.
Tout programme *Xenomai* devra donc commencer – avant de lancer ses tâches temps réel – par une invocation :

```
1 mlockall(MCL_CURRENT | MCL_FUTURE);
```

Si le programme temps réel doit afficher des messages sur sa sortie standard, sa sortie d'erreur ou dans un fichier de traces, il peut utiliser l'une des fonctions de la bibliothèque *RTDK* (intégrée dans *Xenomai*) :

```
1  int rt_printf (const char * format...);
2  int rt_vprintf (const char * format, va_list arguments);
3
4  int rt_fprintf (FILE * stream, const char * format...);
5  int rt_vfprintf (FILE * stream, const char * format, va_list arguments);
6
7  int rt_puchar(int c);
8  int rt_fputc(int c, FILE *stream);
9
10 int rt_puts (const char * string);
11
12 int rt_fputs(const char *string, FILE *stream);
13
14 int rt_fwrite(const void *buffer, size_t size, size_t nb, FILE *stream);
15
16 void rt_syslog (int priority, const char *format...);
17 void rt_vsyslog (int priority, const char *format, va_list args);
```

À l'usage, ces fonctions sont très proches de leurs équivalents de la *libC* standard. Leur fonctionnement interne est toutefois assez différent, car pour garantir un déroulement déterministe, il est hors de question de s'appuyer directement sur les appels système de Linux comme *write()*. Une bibliothèque créée par *Jan Kiszka* propose donc de mémoriser les messages dans des buffers circulaires spécifiques à chaque thread. Un thread non temps réel est chargé d'effectuer l'affichage des données périodiquement (100 ms).

La création d'une tâche peut être réalisée avec *rt_task_create()*, qui prépare toutes les structures de données nécessaires au fonctionnement d'un nouveau thread, suivi de *rt_task_start()*, qui démarre véritablement le thread suspendu. On peut également regrouper les deux opérations en une seule avec *rt_task_spawn()*, qui crée une tâche et la laisse démarrer immédiatement.

```
1  int rt_task_create (RT_TASK * task,
2                      const char * name,
3                      int stack_size,
4                      int priority,
5                      int mode);
```

```
6 int rt_task_start (RT_TASK * task,
7                    void (* function) (void *arg),
8                    void * arg);
9 int rt_task_spawn (RT_TASK * task, const char * name,
10                   int stack_size,
11                   int priority,
12                   int mode,
13                   void (* function) (void * arg),
14                   void * arg);
```

Arrêtons-nous un instant sur les paramètres de ces fonctions présentés dans le tableau 2.

TABLE 2 – Paramètres de la création des tâches

Paramètres	***Signification***
task	Pointeur sur une structure qui sera initialisée durant l'appel à *rt_task_create()* pour représenter la tâche temps réel.
name	Nom de la tâche tel qu'il apparaît dans */proc/xenomai/sched*.
stack_size	Taille de la pile. Si la valeur est nulle, une taille par défaut est affectée permettant de stocker 1 024 entiers.
priority	La priorité va de 1 à 99 dans l'espace des tâches *Xenomai* de manière similaire aux threads temps réel souple de Linux.
mode	Configuration du thread temps réel.
function	La fonction que le thread doit exécuter.
arg	Argument que la fonction du thread temps réel reçoit en paramètre à son démarrage.

Le paramètre *mode* est une combinaison par OU binaire entre les éléments suivants (tableau 3).

TABLE 3 – Paramètres du champs mode

Paramètres	***Signification***
T_CPU(*numero*)	La tâche s'exécutera sur le CPU indiqué. Le *numero* doit être strictement inférieur à *RTHAL_NR_CPUS*.
T_FPU	La tâche temps réel utilisera les ressources de la *FPU* (*Floating Point Unit*). Il s'agit d'un attribut fixé automatiquement pour les tâches créées dans l'espace utilisateur. Pour celles créées dans le noyau, ceci permet d'indiquer à *Xenomai* s'il faut sauvegarder ou non les registres de la *FPU* lors des commutations vers cette tâche.
T_JOINABLE	Cet attribut indique qu'il est possible d'attendre la fin de l'exécution de la tâche. Nous allons l'utiliser ci-après.
T_LOCK	Avant d'activer cette tâche, l'ordonnanceur sera verrouillé (aucune autre tâche ne pourra s'exécuter sur ce *CPU*) et il faudra le déverrouiller explicitement en utilisant *rt_task_set_mode()*.
T_WARNSW	Si le thread passe du mode primaire au mode secondaire, il recevra un signal *SIGDEBUG*. Par défaut, ceci tue le processus, mais on peut le capturer pour analyser le problème. Nous le détaillerons ci-après.

2.1 Tâches basiques (Hello World)

Dans notre premier exemple, nous allons également utiliser les appels système :

```
1 int rt_task_join (RT_TASK * task);
```

qui permet d'attendre la fin d'un thread temps réel, et :

```
1 int rt_task_sleep (RTIME duration);
```

pour endormir la tâche courante pendant une durée comptée en nanosecondes (nous reviendrons sur cet argument ultérieurement).
L'application suivante (*exemple-hello-01.c*) est définit dans l'espace utilisateur :

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
```

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

#include <alchemy/task.h>

void say_hello_world (void * unused)
{
        while (1) {
                rt_printf("Hello from Xenomai Realtime Space\n");
                rt_task_sleep(1000000000LL); // 1 milliard ns = 1 s.
        }
}
int main(void)
{
        int err;
        RT_TASK task;

        mlockall(MCL_CURRENT|MCL_FUTURE);

        if ((err = rt_task_spawn(& task, "Hello_01",
                                 0, 50, T_JOINABLE,
                                 say_hello_world, NULL)) != 0) {
                fprintf(stderr, "rt_task_spawn: %s\n", strerror(-err));
                exit(EXIT_FAILURE);
        }
        rt_task_join(& task);
        return 0;
}
```

Les fonctions de l'API *Xenomai* renvoient généralement zéro si tout s'est bien passé et une valeur d'erreur en cas d'échec. Elles ne remplissent pas la variable globale errno comme peuvent le faire d'autres fonctions de la bibliothèque *C*, aussi ne doit-on pas appeler directement *perror()*, mais enregistrer le code d'erreur négatif et le transmettre à *strerror()* avant de l'afficher.
La compilation d'un programme avec l'API native de *Xenomai* va faire appel à un fichier *Makefile* dans lequel on invoquera le script *xeno-config* (installé dans */usr/xenomai/bin/*) pour obtenir les paramètres de compilation à transmettre au compilateur *gcc*.

En voici un exemple minimal :

```
APPLICATIONS = exemple-hello-01

SKIN = alchemy

XENOCONFIG=/usr/xenomai/bin/xeno-config
CC=$(shell $(XENOCONFIG) --cc)
CFLAGS=$(shell $(XENOCONFIG) --skin=$(SKIN) --cflags)
LDFLAGS=$(shell $(XENOCONFIG) --skin=$(SKIN) --ldflags)
LIBDIR=$(shell $(XENOCONFIG) --skin=$(SKIN) --libdir)

all:: $(APPLICATIONS)

clean::
        $(RM) -f $(APPLICATIONS) *.o *~
```

Compilons le programme :

```
# make
# ./exemple-hello-01
Hello from Xenomai Realtime Space
Hello from Xenomai Realtime Space
Hello from Xenomai Realtime Space
(Controle-C)
#
```

On démarre le processus normalement. Pour qu'il puisse trouver les bibliothèques dont il a besoin, il ne faut pas oublier de configurer la variable *LD_LIBRARY_PATH* avant de lancer l'application (*bashrc*). Sinon, on risque d'avoir ce type d'erreur :

```
# ./exemple-hello-01
./exemple-hello-01: error while loading shared libraries: libalchemy.so.0:
        cannot open shared object file: No such file or directory
```

Tout naturellement, il est nécessaire d'avoir les privilèges root pour exécuter une application avec des threads de hautes priorités.
Pendant le déroulement du thread, il est possible de l'observer ainsi :

```
# cat /proc/xenomai/sched/threads
# cat /proc/xenomai/shed/stat
```

Nous observons la présence du noyau *Linux* (*ROOT*) sur quatre *CPU* et du handler de l'interruption timer pour *Xenomai*. Vous pourrez observer combien de fois

notre tâche a été activée (colonne *Context Switches CSW*) et il y a eu combien de commutations du mode primaire vers le secondaire (colonne *Mode Switches MSW*). Il est possible de basculer un thread déjà existant dans le domaine *Linux* en un thread temps réel sans en créer de nouveau. Pour cela, on invoque la fonction :

```
1 int rt_task_shadow (RT_TASK * task,s
2                     const char * name,
3                     int priority,
4                     int mode);
```

Les seuls modes autorisés sont *T_LOCK* et *T_WARNSW*.
En voici un exemple très simple :

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/mman.h>
5
6  #include <alchemy/task.h>
7  int main(void)
8  {
9          int err;
10
11         mlockall(MCL_CURRENT|MCL_FUTURE);
12
13         if ((err = rt_task_shadow(NULL, "Hello␣World␣02",
14                                   50, 0)) != 0) {
15                 fprintf(stderr, "rt_task_shadow:␣%s\n", strerror(-err));
16                 exit(EXIT_FAILURE);
17         }
18         while (1) {
19                 rt_printf("Hello␣World␣02\n");
20                 rt_task_sleep(1000000000LL); // 1 sec.
21         }
22         return 0;
23 }
```

Dans cet exemple, nous n'avons pas besoin de récupérer l'identifiant de la nouvelle tâche, ainsi on fourni un pointeur *NULL* en premier argument. L'exécution donne un résultat comparable au précédent :

```
1 # ./exemple-hello-02
```

```
Hello World 02
Hello World 02
Hello World 02
Hello World 02
Hello World 02
(Controle-C)
#
```

2.2 Tâches périodiques

Il est souvent nécessaire dans les applications temps réel de différer un traitement, afin qu'il soit réalisé plus tard, ou de programmer une action à effectuer périodiquement. Pour obtenir ce comportement, *Xenomai* nous propose deux méthodes complémentaires : les réveils périodiques et les alarmes. Chaque tâche peut être régulièrement activée par un réveil personnel. Elle peut aussi recevoir les signaux d'une ou plusieurs alarmes (qui peuvent elles-mêmes être partagées entre plusieurs tâches).
Le principe est simple : nous pouvons rendre un thread périodique, après sa création, grâce à la fonction :

```
int rt_task_set_periodic (RT_TASK * tache,
                          RTIME start,
                          RTIME period);
```

Le premier paramètre est l'identifiant de la tâche. Dans le cas où une tâche veut se rendre périodique elle-même, elle peut employer *NULL* ou bien *rt_task_self()* qui lui renvoie toujours son propre identifiant :

```
RT_TASK * rt_task_self (void);
```

Les deux autres paramètres s'expriment en unités d'horloge *Xenomai*. On peut configurer cette valeur, qui correspond également à la résolution de tous les paramètres temporels de l'API *Alchemy*, en ajoutant l'option "*–alchemy-clock-resolution=n*" sur la ligne de commande du processus :

- si la valeur est zéro (ou si elle n'est pas indiquée sur la ligne de commande), les heures de déclenchement et les durées sont mesurées en nanosecondes ;
- si la valeur n'est pas nulle, elle correspond à la durée (en microsecondes) d'un tick, et toutes les autres valeurs de temps sont exprimées en multiples de ce tick.

Dans la plupart des cas, on préfère laisser la période de base de Xenomai à zéro, et

exprimer les valeurs temporelles en nanosecondes. C'est ce que nous utiliserons dans la suite.
Le deuxième paramètre *start* indique l'heure de la première activation de la tâche. Si la valeur est *TM_NOW*, l'activation est immédiate. On peut également utiliser la valeur *rt_timer_read()* qui nous renvoie la valeur du timer système, à laquelle on ajoute la durée désirée avant le premier déclenchement.

```
1 RTIME rt_timer_read (void);
```

Le troisième paramètre est la période de réveil du thread. La valeur *TM_INFINITE* permet d'arrêter l'activation régulière de la tâche. Comme *Xenomai* anticipe les déclenchements des timers pour corriger leur latence, il est indispensable que la valeur soit supérieure à celle se trouvant dans */proc/xenomai/latency* (10 156 nanosecondes, soient 10,156 microsecondes sur mon *Raspberry Pi 3*).
Pour être réveillée, celle-ci doit préalablement être en attente sur l'appel *rt_task_wait_period()* :

```
1 int rt_task_wait_period (unsigned long * overruns);
```

Cette fonction bloque la tâche appelante (sauf si la période n'a pas encore été programmée) jusqu'à la prochaine activation. Elle renvoie normalement zéro. En cas de problème (période non programmée, signal reçu, etc.), *rt_task_wait_period()* renvoie un code d'erreur négatif.
Le comportement global de la tâche périodique ressemblera donc à :

```
1 while (rt_task_wait_period (NULL) == 0) {
2         // traitement
3 }
```

Si toutefois le traitement dure plus longtemps qu'une période, ou si une autre tâche plus prioritaire préempte le thread appelant, un dépassement pourra se produire. Si le paramètre de *rt_task_wait_period()* n'est pas *NULL*, le pointeur sera renseigné avec le nombre d'activations manquées. Attention, si aucun dépassement n'a lieu, la valeur n'est pas modifiée, il est donc important d'initialiser la variable à zéro avant d'appeler la fonction.
Ainsi, nous pouvons effectuer une mesure de précision des tâches périodiques avec le programme suivant (*exemple-periodique.c*), auquel on fournit une période en microsecondes et qui nous affiche sur sa sortie standard les durées effectives entre activations.

```
1 #include <fcntl.h>
2 #include <stdio.h>
```

```
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/mman.h>
6
7  #include <alchemy/task.h>
8  #include <alchemy/timer.h>
9
10 void periodic_function (void * arg)
11 {
12         RTIME previous = 0;
13         RTIME now;
14         RTIME period;
15         RTIME duration;
16
17         period = * (RTIME *) arg;
18         period = period * 1000; // en ns
19         rt_task_set_periodic(NULL, TM_NOW, period);
20
21         while (1) {
22                 rt_task_wait_period(NULL);
23                 now = rt_timer_read();
24                 if (previous != 0) {
25                         duration = now - previous;
26                         rt_printf("%llu\n", duration/1000);
27                 }
28                 previous = now;
29         }
30 }
31
32 int main(int argc, char * argv[])
33 {
34         RT_TASK task;
35         RTIME period;
36         int err;
37
38         mlockall(MCL_CURRENT|MCL_FUTURE);
39
40         if ((argc != 2) || (sscanf(argv[1], "%llu", & period) != 1)) {
41                 fprintf(stderr, "usage: %s period_us\n", argv[0]);
```

```
42                  exit(EXIT_FAILURE);
43          }
44
45          if ((err = rt_task_spawn(& task, NULL, 0, 50,
46                  T_JOINABLE, periodic_function, &period)) != 0) {
47                  fprintf(stderr, "rt_task_spaw:␣%s\n", strerror(-err));
48                  exit(EXIT_FAILURE);
49          }
50
51          rt_task_join(& task);
52          return 0;
53  }
```

Effectuons un premier test avec une période d'une milliseconde :

```
1  # ./exemple-periodique 1000
2  1000
3  1000
4  999
5  1001
6  999
7  999
8  1000
9  999
10 1000
11 1000
12 999
13 1001
14 999
15 999
16 1000
17 999
18 999
19 1000
20 (Controle-C)
21 #
```

Faites le test et observez l'écart entre votre demande et ce qui est obtenu.
Faisons à présent fonctionner le même programme avec de fortes perturbations engendrées par le script *dohell*. En effet, *Xenomai* est fourni avec un script nommé *dohell* qui place le système dans des conditions infernales... En exécutant de nom-

breuses commandes en parallèle, il provoque une activité importante des processus et le déclenchement de fréquentes interruptions (réseau, disque...). Il est configurable grâce à différentes options (tableau 4) :

TABLE 4 – Paramètres du programme dohell

Options	***Signification***
-b hackbench	Chemin d'accès au programme de tests *hackbench*. Si l'option n'est pas fournie, ce test n'est pas lancé.
-s serveur -p port	Cible vers qui envoyer des paquets réseau avec *nc* (ou *netcat*). Par défaut, le port est 9. Si *-s* n'est pas précisé, cet outil est ignoré.
-m repertoire	Répertoire (point de montage, éventuellement) dans lequel on crée en boucle un fichier de 100 Mo remplis de zéros.
-l repertoire	Répertoire contenant les test *LTP* (*Linux Test Packages*) à exécuter.
duree	Durée en secondes d'exécution de *dohell*.

Utilisé par l'option *-b*, le programme *hackbench* peut être trouvé sur le site d'Ingo Molnár chez Redhat :
http ://people.redhat.com/mingo/cfs-scheduler/tools/.
Quant aux tests de *LTP*, on peut les obtenir à l'adresse suivante :
http ://ltp.sourceforge.net/.
Outre les commandes précédentes, le script *dohell* exécute en boucle les actions suivantes en parallèle :

- lire */proc/interrupts* ;
- exécuter *ps* ;
- copier */dev/zero* dans */dev/null* ;
- descendre l'arborescence depuis la racine.

Voici un exemple d'exécution de *dohell* :

```
1 sudo /usr/xenomai/bin/dohell -m /mnt/ 30
```

Re-exécutons le programme avec les perturbations :

```
1 # ./exemple-periodique 1000
```

Remarquez la différence avec la première exécution. Vous pourrez aussi refaire le test en diminuant la période.
N'oubliez pas d'arrêter le script *dohell*. Pour cela, cherchez le PID de ce processus :

```
# ps -edf | grep "dohell"
```

Par la suite, tuez ce ou ces processus :

```
# kill -9 XXXX
```

2.3 Synchronisation des tâches dans Xenomai

Naturellement, *Xenomai* offre une panoplie complète de mécanismes classiques de synchronisation – *sémaphores*, *mutex*, *variables conditions*, *attente d'événements*. Nous allons observer quelques exemples simples en rappelant qu'une description plus complète de l'API est disponible dans la documentation en ligne ou dans le répertoire d'installation de *Xenomai*.

2.3.1 Sémaphores

Xenomai propose une implémentation des sémaphores classiques de *Dijkstra*, ainsi que quelques extensions. Un sémaphore est un objet de synchronisation courant dans la programmation temps réel. Il est associé à un compteur initialisé avec une valeur N positive. Chaque fois qu'une tâche a besoin d'accéder à la ressource protégée par le sémaphore, elle appelle l'opération *P()*. Si le compteur est strictement supérieur à zéro, il est décrémenté et l'opération se termine immédiatement, sinon la tâche est endormie jusqu'à ce que le compteur redevienne positif (ou que le délai maximal d'attente soit atteint). Lorsqu'une tâche a terminé son accès à la ressource concernée, elle invoque l'opération *V()*, qui incrémente le compteur en réveillant éventuellement une tâche en attente. Nous sommes ainsi sûrs qu'il n'y aura jamais plus de N tâches tenant le sémaphore à un instant donné (si N est initialisé à *1*, le comportement est approximativement celui d'un mutex).
Pour information, *P()* et *V()* viennent du hollandais Proberen (tester) et Verhogen (augmenter), langue maternelle de *Dijkstra*, qui a décrit en premier cette structure de synchronisation.
Les fonctions essentielles pour manipuler les sémaphores de *Xenomai* sont les suivantes :

```
int rt_sem_create (RT_SEM * semaphore, const char * name,
                   unsigned long count, int mode);
int rt_sem_delete (RT_SEM * semaphore);
int rt_sem_p (RT_SEM * semaphore, RTIME timeout);
int rt_sem_v (RT_SEM * semaphore);
```

Le dernier paramètre *mode* du *rt_sem_create()* peut prendre l'une des trois valeurs suivantes (tableau 5), ce qui représente une extension par rapport aux sémaphores usuels.

TABLE 5 – Valuers du paramètre *mode* du *rt_sem_create()*

Valeurs	***Signification***
S_PRIO	Lors d'une opération *V()*, le compteur est incrémenté et on réveille la tâche la plus prioritaire endormie dans une opération *P()* pour qu'elle puisse tester et décrémenter le compteur.
S_FIFO	Lors d'une opération *V()*, le compteur est incrémenté, puis on réveille la première tâche endormie, quelle que soit sa priorité.
S_PULSE	Lors d'une opération *V()*, on réveille la tâche la plus prioritaire. Si aucune tâche n'est en attente, le compteur n'est pas augmenté. Ainsi, le sémaphore joue un rôle de point de blocage pour chaque *P()* et de libération à chaque *V()*, ce qui rappelle le principe des variables conditions. Dans ce mode, le compteur doit obligatoirement être initialisé avec une valeur nulle (et il la conserve).

Nous allons construire un petit exemple (*exemple-semaphore.c*) dans lequel six tâches temps réel vont tenter simultanément d'accéder à un sémaphore dont le compteur est initialisé – avec le troisième argument de *rt_sem_create()* – à la valeur *4*. Chaque tâche qui obtient l'accès au sémaphore le conserve durant deux secondes, avant de le restituer et de se remettre en attente, ceci à quatre reprises. À chaque fois qu'un thread obtient ou restitue le sémaphore, il l'indique dans une table globale qu'une septième tâche vient consulter régulièrement pour afficher l'état des threads du processus.

```
1  #include <errno.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/mman.h>
6
7  #include <alchemy/sem.h>
8  #include <alchemy/task.h>
9  #include <alchemy/timer.h>
10
```

```
11 #define NB_TASKS 6
12 static RT_SEM sem;
13 static int with_semaphore[NB_TASKS];
14
15 void thread_function (void * arg)
16 {
17         int i;
18         int num = (int) arg;
19
20         for (i = 0; i < 4; i ++) {
21                 with_semaphore[num] = 0;
22                 rt_sem_p(& sem, TM_INFINITE);
23                 with_semaphore[num] = 1;
24                 sleep(2);
25                 rt_sem_v(& sem);
26         }
27         with_semaphore[num] = -1;
28 }
29
30
31 void observer(void * unused)
32 {
33         int i;
34         rt_printf("With␣semaphore␣␣␣␣Without␣semaphore\n");
35         while (1) {
36                 for (i = 0; i < NB_TASKS; i ++)
37                         if (with_semaphore[i] == 1)
38                                 rt_printf("%d␣", i);
39                 rt_printf ("␣␣␣␣␣␣␣␣␣␣");
40                 for (i = 0; i < NB_TASKS; i ++)
41                         if (with_semaphore[i] == 0)
42                                 rt_printf("%d␣", i);
43                 rt_printf("\n");
44                 rt_task_sleep(1000000000);
45         }
46 }
47
48
49 int main(int argc, char * argv[])
```

```
50 {
51         int i;
52         int err;
53         RT_TASK task[NB_TASKS];
54         RT_TASK observ;
55
56         mlockall(MCL_CURRENT|MCL_FUTURE);
57
58         err = rt_sem_create(& sem, "Sem-01",
59                             NB_TASKS-2, S_PRIO);
60         if (err != 0) {
61                 fprintf(stderr, "rt_sem_create:%s\n",
62                         strerror(-err));
63                 exit(EXIT_FAILURE);
64         }
65
66         for (i = 0; i < NB_TASKS; i ++) {
67                 err = rt_task_spawn(& task[i], NULL,
68                                     0, 99, T_JOINABLE,
69                                     thread_function, (void *) i);
70                 if (err != 0) {
71                         fprintf(stderr, "rt_task_spawn:%s\n",
72                                 strerror(-err));
73                         exit(EXIT_FAILURE);
74                 }
75         }
76
77         err = rt_task_spawn(& observ, NULL, 0, 99,
78                             T_JOINABLE, observer, NULL);
79         if (err != 0) {
80                 fprintf(stderr, "rt_task_spawn:%s\n",
81                         strerror(-err));
82                 exit(EXIT_FAILURE);
83         }
84
85         for (i = 0; i < NB_TASKS; i ++)
86                 rt_task_join(& task[i]);
87
88         rt_task_delete(& observ);
```

```
89          rt_sem_delete(& sem);
90
91          return EXIT_SUCCESS;
92  }
```

Ce programme présente un défaut important qui est l'accès aux cases du tableau *with_semaphore[]* qui n'est pas protégé. Il y a donc un risque que la consultation faite par le thread observateur se produise au moment exact d'une écriture par l'un des autres threads et obtienne une valeur incohérente. Dans une application réelle, il conviendrait d'utiliser un autre mécanisme de verrouillage pour protéger l'accès au tableau, comme un mutex que nous verrons plus loin. Lors de l'exécution, nous voyons bien que seules quatre tâches peuvent obtenir simultanément le sémaphore, les deux autres restant en attente.

```
1  # ./exemple-semaphore
2  With semaphore Without semaphore
3  0 1 2 3   4 5
4  0 1 2 3   4 5
5  0 1 4 5   2 3
6  0 1 4 5   2 3
7  2 3 4 5   0 1
8  2 3 4 5   0 1
9  0 1 2 3   4 5
10 0 1 2 3   4 5
11 0 1 4 5   2 3
12 0 1 4 5   2 3
13 2 3 4 5
14 2 3 4 5
15 #
```

2.3.2 Mutex

Les *Mutex* dans *Xenomai* sont principalement utilisés à travers les quatre fonctions suivantes :

```
1  int rt_mutex_create (RT_MUTEX * mutex, const char * name);
2  int rt_mutex_delete (RT_MUTEX * mutex);
3  int rt_mutex_acquire (RT_MUTEX * mutex, RTIME timeout);
```

```
4 int rt_mutex_release (RT_MUTEX * mutex);
```

Le verrouillage du mutex est réalisé par *rt_mutex_acquire()* et sa libération avec *rt_mutex_ release()*.
Les *Mutex* dans *Xenomai* disposent automatiquement du mécanisme d'héritage de priorité. Ceci permet d'éviter les situations d'inversions de priorités que l'on peut rencontrer si une tâche de haute priorité est bloquée en attente d'un mutex tenu par un thread de faible priorité préempté par une tâche de priorité intermédiaire.
Nous allons mettre ceci en évidence dans l'exemple suivant *exemple-mutex.c*. La tâche *T1* (priorité *10*) prend le *Mutex* et démarre la tâche *T3* (priorité *30*) qui se met en attente sur le même *Mutex*. Ensuite, *T1* démarre une tâche *T2* (priorité *20*) qui consomme du temps *CPU* avant de se terminer.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/mman.h>
5
6  #include <alchemy/task.h>
7  #include <alchemy/mutex.h>
8  #include <alchemy/timer.h>
9
10 static RT_MUTEX mutex;
11
12 void thread_1(void *unused);
13 void thread_2(void * unused);
14 void thread_3(void * unused);
15
16 void thread_1(void *unused)
17 {
18         int err;
19         RT_TASK task_2, task_3;
20         rt_fprintf(stderr, "T1 starts\n");
21
22         rt_fprintf(stderr, "T1 waits for the mutex\n");
23         rt_mutex_acquire(& mutex, TM_INFINITE);
24         rt_fprintf(stderr, "T1 holds the mutex\n");
25
26         rt_fprintf(stderr, "T1 starts T3\n");
27         if ((err = rt_task_spawn(& task_3, NULL, 0, 30,
```

```
28                              T_JOINABLE, thread_3, NULL) != 0)) {
29                  fprintf(stderr, "rt_task_spawn:%s\n",
30                          strerror(-err));
31                  exit(EXIT_FAILURE);
32          }
33
34          rt_fprintf(stderr, "T1 starts T2\n");
35          if ((err = rt_task_spawn(& task_2, NULL, 0, 20,
36                              T_JOINABLE, thread_2, NULL) != 0)) {
37                  fprintf(stderr, "rt_task_spawn:%s\n",
38                          strerror(-err));
39                  exit(EXIT_FAILURE);
40          }
41
42          rt_fprintf(stderr, "T1 releases the mutex\n");
43          rt_mutex_release(& mutex);
44
45          rt_fprintf(stderr, "T1 waits for T2 & T3\n");
46          rt_task_join(& task_2);
47          rt_task_join(& task_3);
48
49          rt_fprintf(stderr, "T1 quits\n");
50  }
51
52  void thread_2(void * unused)
53  {
54          rt_fprintf(stderr, "    T2 starts\n");
55          rt_fprintf(stderr, "    T2 works\n");
56          rt_timer_spin(1500000000LLU);
57          rt_fprintf(stderr, "    T2 quits\n");
58  }
59
60  void thread_3(void * unused)
61  {
62          rt_fprintf(stderr, "        T3 starts\n");
63
64          rt_fprintf(stderr, "        T3 waits for the mutex\n");
65          rt_mutex_acquire(& mutex, TM_INFINITE);
66          rt_fprintf(stderr, "        T3 holds the mutex\n");
```

```
67
68          rt_fprintf(stderr, "␣␣␣␣␣␣␣␣T3␣works\n");
69          rt_timer_spin(1500000000LLU);
70
71          rt_fprintf(stderr, "␣␣␣␣␣␣␣␣T3␣releases␣the␣mutex\n");
72          rt_mutex_release(& mutex);
73          rt_fprintf(stderr, "␣␣␣␣␣␣␣␣T3␣quits\n");
74 }
75
76 int main(int argc, char * argv [])
77 {
78          int err;
79          RT_TASK task;
80
81          mlockall(MCL_CURRENT|MCL_FUTURE);
82
83          if ((err = rt_mutex_create(& mutex, "Mutex-exemple")) != 0) {
84                  fprintf(stderr, "rt_mutex_create:%s\n",
85                                  strerror(-err));
86                  exit(EXIT_FAILURE);
87          }
88
89          if ((err = rt_task_spawn(& task, NULL, 0, 10,
90                                  T_JOINABLE, thread_1, NULL) != 0)) {
91                  fprintf(stderr, "rt_task_spawn:%s\n",
92                                  strerror(-err));
93                  exit(EXIT_FAILURE);
94          }
95
96          rt_task_join(& task);
97          rt_mutex_delete(& mutex);
98          return 0;
99 }
```

Exécutons l'exemple :

```
1  # ./exemple-mutex
2  T1 starts
3  T1 waits for the mutex
4  T1 holds the mutex
5  T1 starts T3
6                       T3 starts
7                       T3 waits for the mutex
8  T1 starts T2
9  T1 releases the mutex
10                      T3 holds the mutex
11                      T3 works
12                      T3 releases the mutex
13                      T3 quits
14            T2 starts
15            T2 works
16            T2 quits
17 T1 waits for T2 & T3
18 T1 quits
```

Dans une situation d'inversion de priorité, *T2* pourrait s'exécuter en entier avant que *T1* ne puisse redémarrer, libérer le mutex et laisser *T3* se dérouler.
Nous voyons qu'en réalité, il n'en est rien. L'héritage de priorité propulse *T1* temporairement à la priorité 30 où elle n'est pas préemptée par *T2*.

3 Conclusion sur la programmation Xenomai dans l'espace utilisateur

Nous avons vu quelques primitives de base proposées par *Xenomai* pour la gestion des tâches et leur synchronisation. L'API complète est naturellement bien plus riche, et l'on trouvera de nombreux détails et exemples dans la documentation en ligne.
Nous ne nous sommes intéressés pour le moment qu'à la programmation *Xenomai* et *Linux* en mode utilisateur. Il est pourtant parfois nécessaire de développer des modules de code qui viendront se loger directement dans le kernel. Ceci est utile pour gérer et traiter efficacement les interruptions, par exemple, mais également les entrées-sorties vers les périphériques. Nous allons en étudier un aperçu de dans la section suivante.