Claude BARON

Introduction

Xenomai est une solution « temps réel dur » libre adossée à Linux.
Ce document a pour but une présentation par la pratique des principales fonctionnalités de l'API Native de Xenomai. La section d'introduction décrit quelques éléments de base utiles sur Xenomai, les autres chapitres définissent de façon pragmatique les outils offerts par Xenomai pour la programmation d'application temps réel.

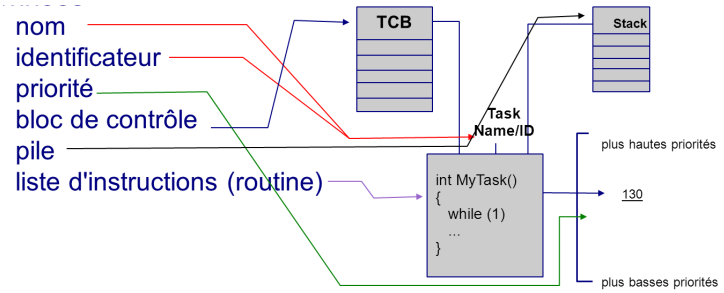
1. Quelques points de rappel sur Linux

Cf [Premiers pas avec Xenomai, David Chabal dchabal ClaudeLELOUP, 2012]

Tâche, processus et thread

Quelle différence y a-t-il entre un processus et un thread sous Linux ?
Un développeur répondra probablement que ça n'a rien à voir, qu'un processus peut comporter plusieurs threads, que l'espace d'adressage mémoire est commun... Un architecte d'application temps réel répondra qu'il n'y a pas de différence fondamentale car les deux sont traités de la même manière dans le scheduler. C'est le point de vue qui est également adopté dans ce document. Dans un contexte d'ordonnancement et de priorité, les termes *tâche*, *processus* et *thread* se confondent.

Une tâche est un simple programme qui s'exécute comme s'il était le seul à utiliser le CPU. Le processus de conception pour des applications temps-réel implique de diviser le travail à effectuer entre différentes tâches qui sont chacune responsable d'une partie du problème. Chaque tâche possède une priorité, du code à exécuter, une partie de la mémoire et une zone de pile (stack).



Dans les appplication temps réel, une tâche est typiquement une boucle infinie qui peut être dans un des états suivants :

- Exécutable : la tâche est prête à être exécutée mais n'a pas à ce moment l'usage du CPU.
- Active : la tâche est celle dont les instructions sont actuellement exécutées par le CPU.
- Bloquée : la tâche attend un signal ou une ressource afin de poursuivre son exécution.
- Interrompue : la tâche possédait l'usage du processeur lorsqu'elle a été suspendue par une interruption.
- ….

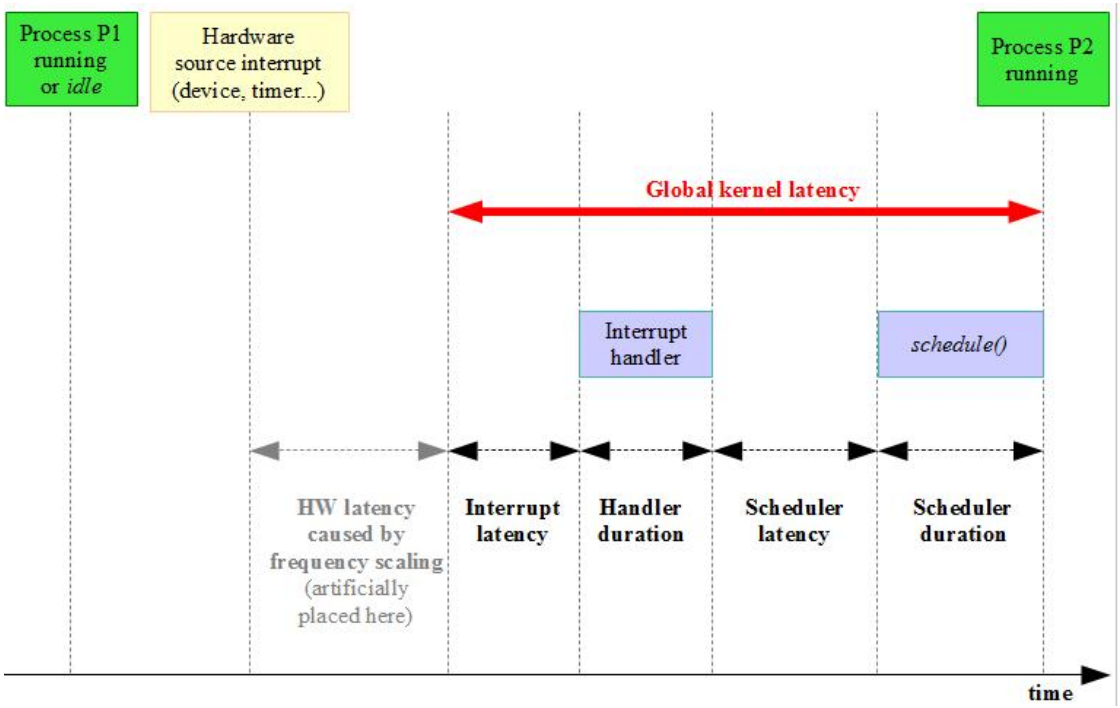
Latence

Le principal problème pour qu'un OS satisfasse à une utilisation temps réel est la latence.
La **latence** est le délai entre le moment où un traitement est nécessaire et l'instant où il est effectif (c'est par exemple le temps qui sépare la génération d'une interruption de son traitement).
Dans un système temps réel, la latence est bornée afin de **garantir le déterminisme temporel**. La valeur de latence retenue pour décrire un système d'exploitation temps réel représente toujours le cas le plus défavorable. Plus la latence est faible, plus on considère le système comme étant performant.

Dans un OS temps réel, la latence est classiquement de l'ordre de quelques microsecondes.

Note : accumulation des latences dans la gestion d'une interruption matérielle

Analysons comment est traitée une interruption.
Dans cet exemple, le matériel génère une interruption (signalant des données disponibles, l'expiration d'un timer, etc.) et un processus doit prendre en compte cet événement.



La latence de l'interruption est le temps entre la génération de l'interruption et l'appel du handler.
Elle est augmentée par :

- le masquage des interruptions à l'intérieur du noyau : le traitement est alors reporté au démasquage ;
- le partage d'une ligne d'interruption : tous les handlers installés sur cette interruption sont alors appelés, ces appels multiples sont forcément coûteux ;
- l'arrivée d'une interruption plus prioritaire : matériellement, si deux interruptions arrivent simultanément, l'arbitrage est fait par le contrôleur d'interruptions (APIC).

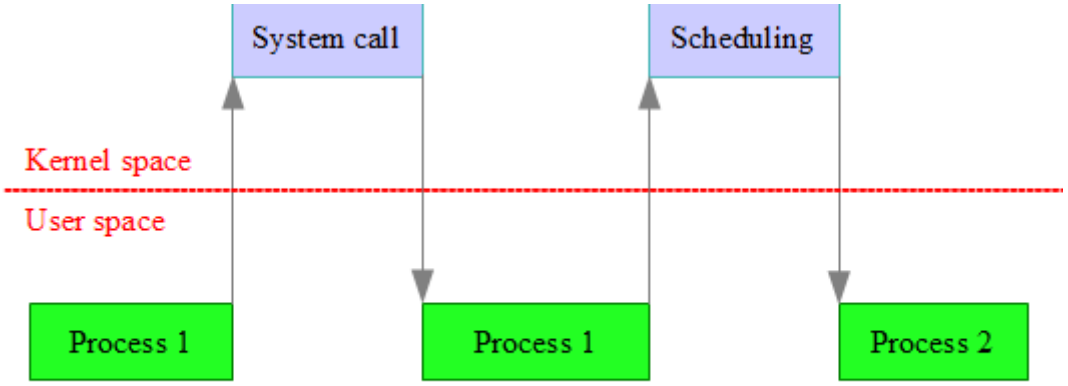
Généralement, les handlers d'interruption ne sont pas réentrants (la **réentrance** est la propriété pour une fonction d'être utilisable simultanément par plusieurs tâches utilisatrices). Par conséquent, afin de prévenir toute autre interruption qui serait alors imbriquée, la ligne est temporairement masquée.
Si des interruptions sont masquées pendant l'exécution du handler, cela signifie que toutes celles arrivant vont être en attente (pending), augmentant ainsi leur latence...
Remarque : Indiquée en gris sur le schéma, la latence induite par les économies d'énergie est matérielle. Elle est un peu particulière car elle n'est pas réellement ajoutée juste après la génération de l'interruption mais elle correspond à une moindre performance du processeur. Afin d'économiser de l'énergie, celui-ci diminue sa vitesse lorsqu'il n'a rien à faire et met un certain temps à retrouver ses pleines capacités.

Préemption

Le remplacement involontaire du processus s'exécutant par un autre est appelé **préemption**. Le terme involontaire signifie que le processus en cours ne s'est pas, par exemple, mis en sommeil ou en attente de réception de données.

Note : préemption en mode noyau

Lors de son exécution, un processus fait des allers-retours entre le mode utilisateur, dans lequel se déroulent les algorithmes codés par les développeurs, et le mode noyau pour les appels systèmes (lecture d'une socket, consultation de la date système, etc.).
Dans cet exemple, P1 et P2 ont la même priorité et s'exécutent en round-robin :



Le problème du noyau Linux est que, par défaut, il n'y a pas de préemption à l'intérieur du noyau.
Pour faire du temps réel, cela n'est pas tolérable pour au moins deux raisons :

- un processus de basse priorité en mode noyau ne peut pas être interrompu par un processus de haute priorité, on se situe alors dans un cas d'inversion de priorité ;
- le temps passé à l'intérieur du noyau peut être très long.

(Nous verrons par la suite qu'il est possible de préempter les tâches à l'intérieur du mode noyau.)

Remarque sur le noyau Linux et ses évolutions

Le problème de base est de rendre les processus en mode noyau préemptibles. Concrètement, cela signifie que si un processus P1 fait un appel système et que, un autre, P2 - plus prioritaire - devient éligible, P1 suspendra son exécution, P2 deviendra alors le processus courant et pourra faire à son tour des appels système.
Conclusion : le noyau doit être réentrant, et ce n'est pas forcément simple à appliquer sur un code déjà existant de plusieurs centaines de milliers de lignes... Le développement réentrant implique la mise en œuvre d'exclusion mutuelle dès qu'une donnée est partagée.

Attention : ne pas confondre préemption et réentrance : La réentrance est nécessaire pour la préemption

Dans la version 2.2, il existait un verrou qui protégeait les sections critiques du noyau. Ce n'est finalement qu'à partir de la version 2.6 que l'exécution d'applications temps réel « mou » est possible grâce à la préemption en mode kernel, via l'ajout de points de descheduling dans les appels système. Il faut cependant noter que cette préemption n'est que partielle puisqu'elle exclut les sections critiques et des handlers d'interruption. Parmi les améliorations importantes, on trouve également dans cette version un nouveau scheduler qui choisit la tâche à exécuter parmi celles éligibles en un temps quasi constant, et ce, quel que soit le nombre de tâches.

Mécanismes de synchronisation

Dans le noyau, il existe plusieurs implémentations des sections critiques (spinlocks, RCU, sémaphores). Le choix est guidé par les performances : temps d'exécution de la section (bref ou long), nombre et type d'accès (lecture ou écriture) à une variable partagée, donnée accessible par plusieurs CPU, etc.

2. Linux et le temps réel

Xenomai est une des réponses à la question « Comment faire du temps réel dur avec Linux ? » mais pas la seule…

Comment faire du temps réel avec Linux : les approches possibles

- ***Aucune préemption forcée***

Par défaut, le code exécuté en mode noyau n'est pas préemptible. Dans cette configuration, les changements de contexte sont réduits : le CPU est disponible pour les calculs et le cache est efficace.
La latence généralement constatée (non garantie) est de l'ordre de la dizaine de millisecondes.
L'activation de la préemption à l'intérieur du noyau dégrade les performances globales du système (overhead).

- ***Une première possibilité : l'option Voluntary Kernel Preemption***

Cette option insère des points de descheduling supplémentaires. Concrètement, les appels système relancent volontairement le scheduler dans leur déroulement.

- ***Pour du temps réel « mou » : Preemptible Kernel (Low-Latency Desktop)***

Ici tout le noyau (hors sections critiques) devient préemptible. Dans le code, il est alors nécessaire d'identifier les sections critiques afin de désactiver la préemption à l'intérieur de celles-ci.
Les versions standard 2.6 peuvent être une solution, rendant ainsi Linux immédiatement opérationnel. La latence maximale est alors de l'ordre de la milliseconde.

- ***Le temps réel avec le patch PREEMPT_RT***

Le principe est de rendre « totalement » préemptible le code du noyau, la latence est ainsi abaissée à moins de 50 µs.
Ce qui n'était pas encore préemptible (les sections critiques (au nombre de 11 000 !) et les handlers d'interruption) le sont désormais et les sémaphores supportent l'héritage de priorité.
Les développements d'applications RT reposent sur l'API POSIX classique.
Limitations : Le patch atteint presque les 60 000 lignes(19), ce qui modifie par conséquent sensiblement le code du noyau.

- ***Le temps réel dur avec les co-noyaux***

Une autre approche consiste à se reposer sur un micronoyau pour satisfaire les contraintes temps réel. Ce micronoyau gère alors le scheduling des tâches temps réel, les timers, les interruptions et la communication entre processus. Linux reste utilisé pour les services non temps réel qu'il procure (connectivité réseau, USB…) et devient une simple tâche du micronoyau.
Xenomai repose sur ce principe.

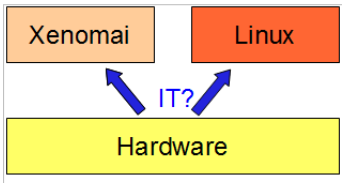
3. Qu'est-ce que Xenomai ?

Xenomai, comment ça marche ?

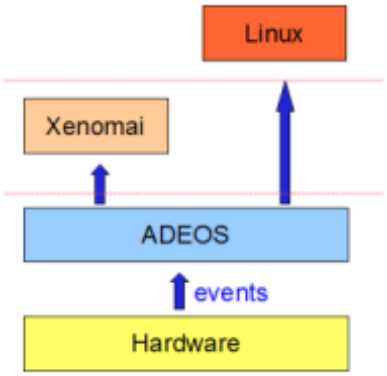
Xenomai est un système d'exploitation temps réel qui a Linux pour tâche de fond. Linux est alors préempté comme une simple tâche. Les tâches gérées par Xenomai apportent ainsi une garantie d'exécution temps réel dur.
Qui dit deux systèmes d'exploitation, dit deux ordonnanceurs… et (au moins) deux problèmes : comment partager le matériel, et comment faire interagir les tâches Linux et Xenomai entre-elles.

1. Le routage des interruptions avec ADEOS

Comment alors faire cohabiter ces deux OS sur le même matériel ? Pour l'accès au CPU et à la mémoire, il n'y a pas de problème car Linux est une tâche. Mais si une interruption (timer, carte…) arrive, qui va la traiter ?



La solution consiste à intercaler une couche entre le matériel et les systèmes d'exploitation. Cette couche de virtualisation (ou hyperviseur, ou encore couche d'abstraction matérielle) s'appelle ADEOS.
ADEOS est un nanokernel qui va capturer les évènements pour ensuite les dispatcher vers Xenomai ou Linux, mais pas dans n'importe quel ordre ! Le but étant tout de même d'avoir un OS temps réel, il est impératif que Xenomai soit le premier à les traiter. Si elles ne sont pas pour lui, elles sont alors « passées » à Linux.



Très concrètement, ADEOS se présente sous la forme d'un patch du noyau Linux.

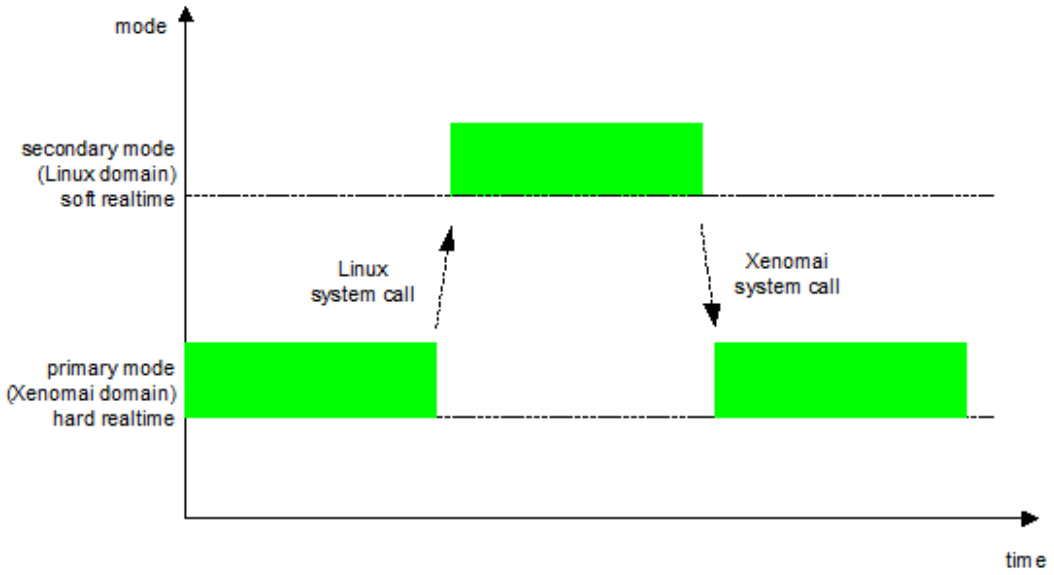
2. Coexistence Xenomai/Linux et partage des ressources : le co-scheduling

En théorie, la virtualisation peut faire cohabiter en toute indépendance plusieurs OS sur le même matériel. Or, que veut-on ? Réponse : faire du temps réel avec Linux afin de profiter de tout l'environnement non temps réel. Intuitivement, on sent bien qu'il faut faire un lien entre les deux. Oui, mais comment ? Par les ressources utilisées !
Pour l'instant, chacun est tranquillement chez soi : les tâches Xenomai dans leur scheduler, idem pour les tâches Linux. Imaginons, un instant qu'une tâche Xenomai cherche à obtenir la date. La ressource « date » appartient au noyau Linux et on ne peut pas aller la chercher directement depuis Xenomai, il se peut en effet qu'elle soit en cours de mise à jour et donc invalide ! Il faut par conséquent passer par un appel système Linux.
Comment traiter les appels système Linux depuis les tâches Xenomai ? Les appels système regroupent les accès à des ressources partagées (terminal, couche réseau…) dont la concurrence d'accès est gérée par Linux. Or, Linux ne connaît pas les tâches Xenomai !!!
La solution consiste à déplacer les tâches d'un scheduler à l'autre en fonction des ressources utilisées. Une tâche créée via l'API Xenomai pourra s'exécuter sous Linux, ce mécanisme totalement transparent est appelé shadow threading.

Une tâche schedulée par Xenomai est dite en primary mode.
Une tâche Xenomai schedulée par Linux est dite en secondary mode.

Le passage d'un mode à l'autre est fonction des ressources accédées et donc des appels système Xenomai ou Linux.

Les termes primary mode et secondary mode sont hérités de la terminologie ADEOS qui appelle les systèmes d'exploitation reposant sur lui des domaines (domains). Ces domaines ont une priorité dans la réception des évènements. Dans notre cas, Xenomai les réceptionne en premier (primary domain, déterministe) et Linux en second (secondary domain, non déterministe).
Une fois que la tâche Xenomai a fait son appel système Linux et qu'elle se trouve en secondary mode, elle y reste tant qu'elle ne fait pas d'appel système Xenomai.



Linux devient une tâche de Xenomai.
Mais quelle priorité lui affecter ? A priori, la plus faible possible (tâche idle pour Xenomai).
Comment alors considérer la priorité des tâches Xenomai passées dans le scheduler Linux ? Faible comme Linux ou bien constante ?
Dans Xenomai, la deuxième implémentation a été retenue.
La priorité d'une tâche Xenomai est constante, quel que soit le scheduler dans lequel elle se trouve.
Cette implémentation n'est possible qu'à une condition : que la priorité de la tâche Linux dans l'ordonnanceur Xenomai évolue dynamiquement, sinon on se retrouverait dans un cas d'inversion de priorité.

La priorité de la tâche Linux dans l'ordonnanceur Xenomai est déterminée de la manière suivante :

- 0 s'il n'y a aucune tâche en secondary mode (idle Xenomai) ;
- la priorité maximale des tâches en secondary mode.

3. La gestion des priorités dans les modes primary et secondary

Nous voici face à un nouveau problème : une fois dans l'ordonnanceur Linux, la tâche Xenomai est maintenant une tâche comme les autres et on revient au point de départ, à savoir, comment faire du temps réel avec Linux ?
Rappelons que les dérives temporelles sont introduites par :

- les interruptions (handlersLinux) : nous avons maintenant ADEOS pour les masquer (Interrupt Shield[1]) ;

[1] Mécanisme qui protège les tâches en secondary mode (de l'augmentation de leur temps d'exécution due aux handlers d'interruption Linux) en filtrant les interruptions. Le temps d'exécution de la tâche est alors borné.

- la préemption par des tâches de plus forte priorité : il suffit de s'exécuter dans la classe SCHED_FIFO ;
- l'absence de préemption à l'intérieur du noyau : les options natives (« low latency desktop ») la résolvent partiellement.

Une tâche en secondary mode est du temps réel mou et son temps d'exécution est borné (Interrupt Shield).
Pour les tâches à contrainte temps réel dur, il faut absolument rester en primary mode.
Les tâches Xenomai ont 99 niveaux de priorité possibles, exactement comme la classe SCHED_FIFO/RR.
Ceci est normal puisque les priorités ne changent pas d'un scheduler à l'autre, la correspondance est donc immédiate.

Les priorités Xenomai sont constantes à travers les schedulers : Le noyau Linux est une tâche de fond dont la priorité évolue dynamiquement en fonction de la tâche Xenomai de plus haute priorité éligible en mode secondaire. Parmi un ensemble de tâches Xenomai éligibles, c'est toujours celle de plus haute priorité qui sera exécutée, quel que soit son mode.

Attention : une tâche Linux peut préempter une tâche Xenomai.
Il faut être attentif à deux choses pour les tâches en secondary mode afin d'éviter des comportements non souhaités :

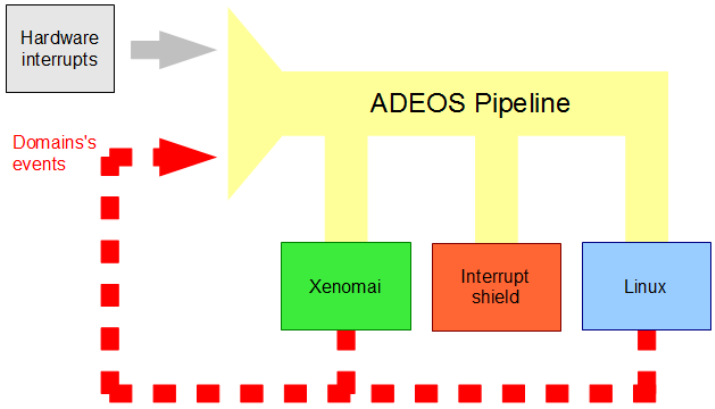
- leurs priorités doivent être inférieures à celles en primary mode ;
- leurs priorités doivent être supérieures aux tâches Linux s'exécutant dans la classe SCHED_FIFO (dépend de la distribution).

4. ADEOS ou comment modifier Linux proprement

Revenons que la notion d'« événements ». Peuvent provoquer un rescheduling :

- les interruptions matérielles : timer pour les tâches périodiques, équipement prêt...
- les signaux logiciels : fin d'une tâche suite à une faute...
- les créations/destructions de tâche : fork, exit...
- les changements de caractéristiques d'une tâche : priorité, policy ;
- les deschedulings volontaires.

L'idée est d'avoir un comportement uniforme quels que soient le type d'événements et le domaine d'origine. On réapplique ce que nous avions vu pour les interruptions matérielles : tous les événements sont envoyés dans un tuyau et les domaines (OS) viennent y puiser les informations qui les intéressent.



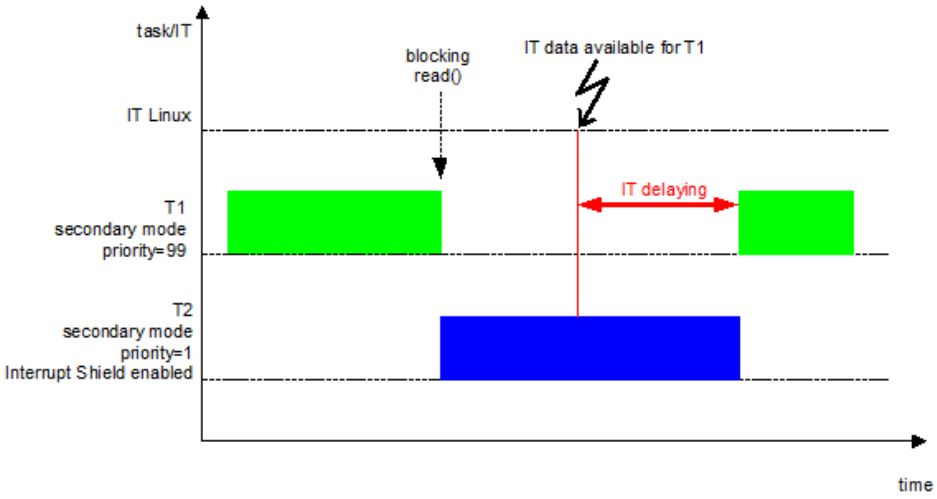
Le patch ADEOS est « respectueux » du code initial du noyau Linux car il ne le modifie que très peu.

4. Interrupt shield

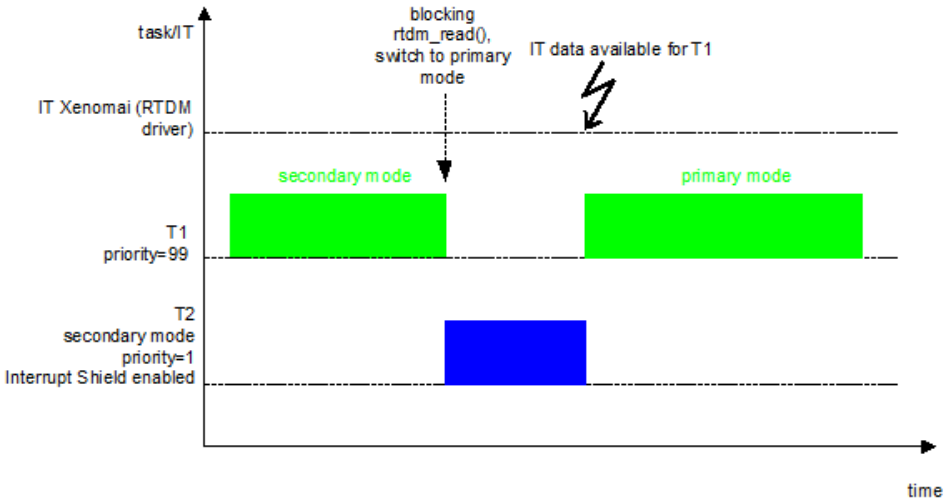
Devinette : nous sommes en secondary mode, nous maîtrisons les priorités des autres processus Linux, quelle est la seule chose qui peut augmenter notre temps d'exécution ?

Réponse : les handlers d'interruption Linux (top-half) et les tasklets hors de contrôle du scheduler des tâches (bottom-half).
Afin de s'en prémunir, un mécanisme appelé *Interrupt shield* protège les tâches en secondary mode en filtrant les interruptions. Le temps d'exécution de la tâche est alors borné.
Mais alors pourquoi l'IS n'est-il pas activé par défaut lors de l'installation ? Simplement, car il y a un risque d'inversion de priorité.

Prenons l'exemple suivant, soit deux tâches T1 et T2 en secondary mode. T2 active l'InterruptShield. T1 a la main (normal car elle est prioritaire) puis se place en attente de données. Elle est alors deschedulée, T2 entre alors en jeu. Une interruption arrive. T1 est censée s'exécuter, or elle est schedulée par Linux, et celui-ci n'a pas encore été informé que l'interruption est arrivée (IS). T2 termine alors son traitement, l'interruption est relâchée, T1 est débloquée.



La solution serait alors que T2 soit protégée des interruptions Linux tout en laissant T1 s'exécuter si des données arrivent.



Ceci revient à dire qu'il faut que l'interruption pour T1 ne soit pas traitée par Linux mais directement par Xenomai. Si ce handler est géré par Xenomai, le read doit l'être également.

Les modes d'exécution

Les tâches Xenomai « courantes » s'exécutent en mode utilisateur.

Attention à ne pas confondre :

- primary et secondary désignent les schedulers ;
- user et kernel désignent le mode d'exécution et donc les ressources accessibles.

L'API native et les skins

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 destinées au portage d'applications existantes ou à du prototypage.

5. Quelques conseils pratiques pour la conception et la programmtion

9.1 Méthodologie de développement

Prototypage de l'architecture dynamique du logiciel

L'architecture dynamique comprend les tâches, les structures qu'elles utilisent pour communiquer (files de messages, variables partagées...) et la manière dont elles sont activées (périodiquement, sémaphore...), comment les composants interagissent au cours du temps.
On débute par l'identification des interfaces du logiciel : monde extérieur (réseau...), matériel (carte série...), autres applications Linux, etc.

Prototypage de l'architecture statique du logiciel

Ensuite, on identifie les tâches et on les caractérise, par exemple :

- asynchrone (sur quel événement ? interruption, action utilisateur) ou synchrone (sur quel événement ? libération d'un sémaphore...) ;
- si synchrone, la fréquence d'activation ;
- la priorité et sa justification ;
- les variables partagées avec les autres tâches ;
- le mode d'exécution attendu : primary ou secondary (dépend de la contrainte temporelle et des interfaces) ;
- le point d'entrée dans l'architecture statique.

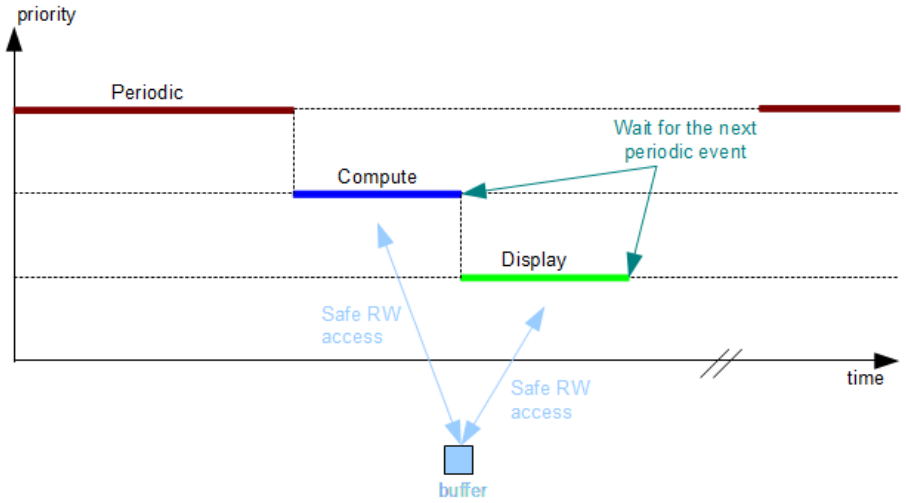
Par la suite, on s'attaque aux variables partagées, ou plus exactement à leur protection. Elles peuvent être protégées par les priorités (surtout si l'architecture est très synchrone, c'est-à-dire que l'on maîtrise totalement l'ordre d'activation des tâches) ou utiliser les mécanismes de Xenomai (mutexes, files de messages...).

Exemple : soient trois tâches :

- Periodic, priorité = 30, périodique ;
- Compute, priorité = 20, synchrone sur Periodic ;
- Display, priorité = 10, synchrone sur Periodic.

Compute et Display partagent une variable appelée buffer.

L'ordre d'activation est alors le suivant :



La variable buffer est alors protégée par conception grâce aux priorités.
Attention: la protection des variables par les priorités n'est valable que dans les systèmes monoCPU.

Vérification de la faisabilité, évaluation du respect des délais

La tenue des contraintes temporelles est également vérifiée de manière préliminaire à cette étape. Des benchmarks sont réalisés pour les traitements « lourds », en tenant compte des hypothèses les plus défavorables (pire cas).
La validation du prototype ne doit pas s'arrêter au cas nominal, mais doit également prendre en compte les cas d'erreur. C'est d'ailleurs eux qui compliquent l'application puisqu'il faut terminer proprement l'application alors que tous les objets RT n'ont pas été créés, que des tâches sont bloquées derrière des sémaphores qui ne seront jamais débloqués, etc.

9.2 Pratiques de programmation

Les mauvaises…

Attention, contrairement aux tâches Linux, la taille de la pile des tâches (définie par rt_task_create à la création de la tâche) n'est pas extensible dynamiquement, l'utilisation de la récursivité est donc à éviter.

… et les bonnes

- Initialisez les variables locales lors de la déclaration, cela dépend des compilateurs et des options, mais elles peuvent alors avoir une valeur indéterminée (dépend du contenu de la pile) et surtout totalement différente d'une exécution sur l'autre (non déterminisme).
- Notez symboliquement les priorités (#define) et regroupez ces définitions (dans un même fichier par exemple), cela permet de comprendre plus facilement le logiciel sans forcément avoir la documentation à côté.
- Priorités des tâches :
  - Lors de l'affectation initiale des priorités, prévoyez large. Plutôt que 1, 2, 3, 4... préférez 10, 20, 30, 40... en cas d'évolution de la spécification (rajout d'une tâche ou changement de priorité), l'impact sera moins lourd.
  - Généralement les tâches logicielles émulant un comportement matériel (timer, génération de signal...) sont les plus prioritaires de l'architecture. Les tâches identifiées comme « lentes » sont souvent celles qui enregistrent ou affichent des traces.
- Pour le type d'activation, on appelera ‘synchrone’ une tâche dont l'exécution est déterminée par un événement (sémaphore libéré toutes les secondes par exemple).

- Terminaison : il faut toujours veiller à ce que les tâches se terminent proprement. Des problèmes de terminaison indiquent parfois des erreurs de codage.
- Les variables globales partagées peuvent être protégées de deux manières :
  - par des sémaphores d'exclusion mutuelle ;
  - par les priorités, à condition que les tâches soient synchrones, c'est-à-dire que l'on maîtrise toujours leur date d'exécution.
- Files de messages :
  - Les files de messages sont utilisées pour que des tâches de priorités différentes puissent communiquer.
  - Dans la mesure du possible, le dimensionnement de chaque file doit être justifié. Cependant, dans la pratique, on considère souvent une taille maximale qui ne sera jamais atteinte.
  - Il est toujours intéressant de savoir si on a perdu des messages et combien lors du stockage (cela peut aider à la mise au point en indiquant un dysfonctionnement ou une sous-évaluation du volume des messages). Si l'écriture dans une file échoue, incrémentez un compteur de messages et affichez-le en fin d'exécution.
- API : si Xenomai propose des skins permettant d'incorporer facilement du code écrit pour d'autres systèmes d'exploitation, il est conseillé d'utiliser l'API 'native' pour les développements spécifiques, car elle est en principe plus efficace (c'est l'API considérée dans ce document)
- L'écriture d'application avec Xenomai nécessite l'inclusion de certains des fichiers d'en-tête suivants :

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

9.3 Aide à la mise au point

Recupérer les codes d'éerreur systématiquement retournés par les fonctions.

Contrôlez toujours le code de retour des appels système, qu'ils soient Xenomai ou Linux.

Exemple :

```
if (rt_task_spawn(   &task_desc,  /* task descriptor */
                     "my task",   /* 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;
```

Ajout de traces

Bien que très pratique, le printf n'est pas toujours accessible (intérieur du noyau) ou impacte les performances temporelles (changement de mode).
Dans les applications hors du noyau, on peut utiliser une tâche dédiée à l'affichage des traces. Les tâches appellent la fonction log_write qui écrit un message dans une file. Cette file est scrutée par la tâche log_task qui a la priorité la plus faible et qui ensuite effectue un printf.
Cette solution présente plusieurs avantages :

- on peut insérer des informations en plus du message brut comme le nom de la tâche et la date ;
- à la manière de printk, on peut associer un niveau de criticité du message que l'on filtrera lors de l'affichage au fur et à mesure du développement (syslog).

Parfois, l'ajout de traces est nécessaire à la validation. Dans la version de production, ces traces doivent seulement être inhibées, les performances temporelles sont ainsi identiques.

Chapitre 1

Les processus et l'ordonnancement

Les systèmes modernes en temps réel sont basés sur les concepts complémentaires de multitâche et de communication inter-tâche. Un environnement multitâche permet à des applications en temps réel d'être réalisées sous la forme d'un ensemble de tâches indépendantes, chacune avec un flux d'exécution séparé et avec son propre ensemble de ressources du système. Les moyens de communication inter-tâches permettent à ces tâches de synchroniser et de coordonner leurs activités.

Le multitâche donne l'illusion de l'existence de nombreuses tâches s'exécutant simultanément alors que, en fait, le noyau entrelace les exécutions en utilisant un algorithme d'ordonnancement.

L'ordonnanceur (scheduler) est le composant du noyau responsable de déterminer quelle tâche s'éxécute à un moment donné. Il gère l'état des tâches et répartit l'usage du processeur entre celles-ci. Un ordonnanceur temps-réel est basé sur la priorité, il donne une priorité unique à chaque tâche. Chaque tâche possèdent une priorité (fixe ou variable au cours de son exécution) basée sur son importance L'ordonnanceur attribue toujours le processeur à la tâche exécutable (non dormante et non bloquée) de plus haute priorité.

Chaque tâche a son propre contexte, qui représente l'environnement du CPU et les ressources système que la tâche voit chaque fois qu'elle est programmée pour s'exécuter par le noyau. Quand l'ordonnanceur doit transférer l'usage du processeur d'une tâche à une autre, il opère un changement de contexte.

Le contexte représente l'état du processeur à un moment donné. Afin que la tâche suspendue puisse continuer son exécution sans être affectée, la première opération à effectuer lors d'un changement de contexte est la sauvegarde de l'état du processeur au moment de la suspension. Le noyau possède pour chaque tâche non dormante un espace mémoire réservé à cet effet. Les données temporaires utilisées par la tâche suspendue doivent être préservées lors des opérations de la nouvelle tâche active.

Durant un changement de contexte, le contexte d'une tâche est enregistré dans son bloc de contrôle (TCB). Il inclut :

- a thread of execution, that is, the task's program counter
- the CPU registers and floating-point registers if necessary
- a stack of dynamic variables and return addresses of function calls
- I/O assignments for standard input, output, error
- a delay timer
- a timeslice timer
- kernel control structures
- signal handlers
- debugging and performance monitoring values.

Le changement de contexte ajoute de l'overhead à l'application. Plus le processeur possède de registres, plus il y a d'overhead. Le temps requit par un changement de contexte est déterminé par le nombre de registres à sauver et à restaurer par le CPU.

Le scheduler multitâche de Xenomai utilise un ordonnancement préemptif des tâches dirigé par les interruptions basé sur la priorité des tâches. Il offre des temps de commutation de contexte rapides et une faible latence d'interruption.

Les tâches peuvent être lancées en mode :

- 'one-shot' (exécution unique) : c'est le mode par défaut sous Xenomai ;
- 'périodique' (exécutions répétitives à chaque période) : la création est la même mais il faut rendre ensuite l'éxécution de la tâche périodique.

Remarque : l'appellation 'tâche' fait reference ici à une tâche temps reel dans l'espace user de Xenomai

- Les différents états d'une tâche : active, prête, endormie, suspendue, bloquée, ….

Exemples :

#define	T_BLOCKED	Sleep-wait for a resource
#define	T_DELAYED	Delayed
#define	T_READY	Linked to the ready queue
#define	T_DORMANT	Not started yet or killed
#define	T_STARTED	Thread has been started
#define	T_LOCK	Holds the scheduler lock (i.e. not preemptible)

1. Créer une tâche

Une tâche est constituée:

- d'un descripteur (qui contient toutes les infos de la tache à un moment donnée),
- d'une pile d'execution,
- d'une priorité,
- d'un mode correspondant au mode de la tache à sa création.

Le descripteur doit être créé avant la tache, il est type RT_TASK :

```
RT_TASK tasc_desc
```

Ensuite, pour créer et activer une tâche avec Xenomai, il y a deux possibilités :

1. Les tâches peuvent être d'abord créées (rt_task_create) puis activées (rt_task_start) :
   a. la fonction rt_task_create() crée une nouvelle tâche temps-réel. Cette tâche s'exécutera soit en kernel mode soit en user-space, cela dépendra du contexte d'appel. On retrouve les paramètres classiques pour les tâches d'un OS temps réel (point d'entrée, priorité, round-robin...). Par défaut une taille 'raisonnable' est prédéfinie (4 Ko sur x86). Si cette taille n'est pas suffisante, on obtiendra immédiatement un segmentation fault. Attention, contrairement

aux tâches Linux, la taille de la pile elle n'est pas extensible dynamiquement, l'utilisation de la récursivité est donc à éviter.

Si la tâche est créée avec succès, la fonction retourne 0 sinon une erreur. (NB : il en sera de même pour toutes les futures fonctions présentées, on retourne en général 0 pour succès et une autre valeur sinon dont il faudra aller voir la valeur dans la doc pour identifier l'erreur).

b. La fonction rt_task_start ().Cette fonction prend en paramètre le task descriptor, l'adresse de la fonction à exécuter et éventuellement des paramètres via cookie.

2. Les tâches sont créées et activées simultanément (rt_task_spawn) :
   La fonction rt_task_spawn() crée et lance immédiatement la tâche. Elle prend exactement les mêmes paramètres que rt_task_create() et rt_task_start().

Primitive	Description
`int rt_task_create ( RT_TASK * task, const char * name, int stksize, int prio, int mode );`	Créer une tâche
`int rt_task_start ( RT_TASK * task, void(*)(void *cookie) entry, void * cookie );`	Lancer une tâche
`int rt_task_spawn ( RT_TASK * task, const char * name, int stksize,int prio, int mode, void(*)(void *cookie) entry, void * cookie );`	Créer et lancer une tâche
`int rt_task_delete (RT_TASK *task)`	Détruire une tâche

Paramètres d'entrée des fonctions :

task	Pointeur d'adresse de la tâche (adresse du task descriptor Xenomai qui sera utilisé pour stocker les données liées à la tâche (identifiant pour cette tâche), de type RT_TASK)
name	Nom de la tâche (chaîne de caractères)
stksize	Taille de la pile où est située la tâche (0 pour utiliser la valeur de la taille par défaut) NB : si la tâche fait un appel système Linux, la priorité sera la même dans le scheduler Linux que dans celui de Xenomai.
prio	Priorité de la tâche (de 0 à 99, 0 étant la priorité la plus basse)
mode	Mode de création de la tâche Les principaux flags possibles sont : — T_SUSP : la tâche démarre en mode suspendu, son exécution n'aura lieu que si un thread fait appel à la fonction rt_task_resume(). — T_JOINABLE : autorise une autre tâche d'attendre la teminaison de la tâche qu'on crée (utilisation de rt_task_join()). — T_FPU : autorise à utilise l'unité de calcul en virgule flottante (Floating Point Unit) quand elle est présente sur la plateforme. — T_CPU (cpuid) : affinité d'une tâche à un processeur. Passer T_FPU|T_CPU(1) va donc créer une tâche avec un support FPU activé et qui aura pour affinité le CPU #1. 0 : no_flags
entry	Each real-time task is associated with a C function that is called when the task is scheduled to run. Linux is written in the C programming language (with some assembly language), as is Xenomai, so C is the preferred language for writing RT Linux programs.In C, a program's "entry point" where execution begins is a function. Entry is the address of the task's body routine.

2. Détruire une tâche

Pour mettre fin à une tâche, on utilise la fonction rt_task_delete (). Elle prend en argument le task descriptor. Attention, si on lui passe comme paramètre NULL, c'est la tâche courante qui est supprimée !

Traditionnellement, pour détruire une tâche dès la fin de son exécution, on la créée dans le programme principal avec le mode T_JOINABLE et on attend dans le programme principal que la tâche ait terminé son exécution avec la fonction rt_task_join()(si le pgme ppal se terminait, alors toutes les tâches qu'il a créées se termineraient aussi, même si elles n'avaient pas terminé leur exécution).

3. Mettre une tâche en sommeil et la réveiller

Il existe deux possibilités pour créer une attente :

- rt_task_sleep() : passage d'un temps relatif. Elle permet de retarder la tâche appelante en lui donnant une durée du blocage en nombre de clock ticks. La valeur donnée d'un tick est soit celle donnée par CONFIG_XENO_OPT_NATIVE_PERIOD si le skin natif est lié à une base de temps périodique, sinon c'est en nanosecondes2.

```
int rt_task_sleep ( RTIME delay );
```

La valeur retournée par la fonction est soit 0 si succès, soit -EINTR si rt_task_unblock() a été appelé pour la tâche dormante avant que la durée ne se soit écoulée, soit une autre valeur pour une erreur.

- rt_task_sleep_until() : passage d'une date absolu. Elle retarde l'exécution de la tâche jusqu'à ce que la date donnée soit atteinte. Cette date absolue est donnée en clock ticks. Il est rare qu'on ait besoin de donner un retard de cette manière, on préfère plutôt utiliser rt_task_sleep() pour cela. Par contre, il est possible de lui passer la macro TM_INFINITE qui met l'appelant dans un état d'attente infini jusqu'à ce que rt_task_unblock() soit appelé.

```
int rt_task_sleep_until ( RTIME date );
```

Les valeurs de retour sont à peu prêt les mêmes que pour rt_task_sleep().

Pour débloquer une tâche en sommeil, on fait appel à rt_task_unblock().

```
int rt_task_unblock ( RT_TASK* task );
```

2 In Xenomai, time is given by the system clock. When using the native skin, the system clock can be configured with the function rt_timer_set_mode(ns_tick). When ns_tick is 0 the clock is configured to give the time in nanoseconds (which is the default), otherwise the time is given in jiffies where each jiffie is ns_tick nanoseconds long.
The system clock can be read by the function rt_timer_read() : RTIME now; now = rt_timer_read();

4. Modifier la priorité d'une tâche

Pour modifier la priorité d'une tâche, on peut utiliser la fonction rt_task_set_priority(). Elle prend en argument le task descriptor et la nouvelle priorité à affecter à la tâche (valeur entre 0 et 99).

```
int rt_task_set_priority ( RT_TASK * task, int prio);
```

Ce service appelle la procédure de rescheduling.

NB : Assigner la même priorité à une tâche courante ou à une tâche prête à être exécutée la renvoie à la fin de son groupe de priorité, causant un manual round-robin.

5. Obtenir des informations sur les tâches

Récuperer le descripteur de la tâche courante

Pour récupérer le descripteur de la tâche courante, il suffit d'appeler RT_TASK* rt_task_self (void). Cette fonction retourne un pointeur non NULL si succès. Le descripteur de la tâche courante peut s'avérer utile pour beaucoup d'autre opération comme l'annulation, la suspension ou l'arrêt de cette tâche.

Récupérer des informations sur une tâche

Grâce à la fonction rt_task_inquire(), on peut récupérer des informations sur une tâche particulière.

```
int rt_task_inquire ( RT_TASK* task, RT_TASK_INFO* info );
```

L'ensemble des informations sont retournées dans la stucture RT_TASK_INFO qui possède les champs suivant :

- int bprio : La priorité de base affectée à cette tâche.
- int cprio : La priorité courante. Elle a pu être modifiée en cours de route par la fonction rt_task_set_priority() et du coup, elle sera différente de la priorité de base.
- unsigned status : le status de la tâche. Les principaux status possibles sont :
- T_BLOCKED : attente passive pour une ressource.
- T_DELAYED : retardé.
- T_READY : ajouté à la ready queue.
- T_DORMANT : pas encore commencé ou tué.
- T_STARTED : démarré
- ...

Comparer deux tâches

On peut parfois avoir besoin de comparer deux descripteurs pour savoir s'ils réfèrent à la même tâche. On utilise la fonction rt_task_same. Si les deux descripteurs réfèrent à la même tâche, alors une valeur di fférente de 0 est retournée.

```
int rt_task_same ( RT_TASK* task1, RT_TASK* task2);
```

6. Suspendre une tâche

Pour suspendre une tâche, on utilise la primitive rt_task_suspend(). Pour lui faire reprendre son exécution, on utilise rt_task_resume().

7. Créer et activer une tâche périodique

Une tâche périodique est une tâche dont l'exécution est activée de façon régulière à chaque période. Elle est démarrée sous Xenomai comme les autres tâches ('rt_task_start). On a juste besoin de faire appel à deux fonctions supplémentaires, rt_task_set_periodic() afin d'indiquer à la tâche sa période et rt_task_wait_period() pour descheduler la tâche en attendant le prochain cycle (quand la tâche périodique a terminé son traitement, elle libère le processeur et commence à attendre la prochaine période)

Primitive	Description
int rt_task_set_periodic (RT_TASK * task, RTIME start_time, RTIME period)	Permet d'indiquer la période de la tâche
int rt_timer_set_mode (RTIME nstick)	Régler la vitesse d'horloge du système
int rt_task_wait_period (unsigned long * overruns_r)	Faire attendre la tâche jusqu'au prochain cycle
int rt_timer_inquire (RT_TIMER_INFO * info)	Informations sur le timer

Paramètres des fonctions :

task	descriptor of the affected task. This task is immediately delayed until the first periodic release point is reached. If *task* is NULL, the current task is set periodic
start_time	absolute time, expressed in clock ticks (nanoseconds or jiffies), when the task should begin execution. If 'start_time' is equal to TM_NOW, the current system date is used, and no initial delay takes place.
Period	task's period, in clock ticks, which will be rounded to the nearest multiple of the period of the system timer in periodic mode. Note: for newer Xenomai periodic mode is emulated by a software driver (see the note above) which uses one-shot mode programming of the timer chip instead. This leads to an accurracy of the task's period in the order of the baseperiod of the timer chip
nstick	Temps entre deux tops d'horloge
info	Information du timer sous forme d'une structure

Chapitre 2
La communication inter-tâches

Dans un système multi-tâches, les tâches sont en concurrence mais doivent coopérer afin que le fonctionenment global de l'application soit bien celui désiré et soit cohérent. Elles échangent donc ou partage des informations et synchronisent entre elles leurs traitements et l'accès aux informations.

On distingue deux modèles de communication :

- soit le système est dit 'centralisé' : il prévilégie la communication et la synchronisation par mémoire commune ;
- soit le système est dit ''distribué' : il prévilégie la communication et la synchronisation par messages.

Pour faire communiquer deux tâches dans une application, on peut donc soit utiliser des structures de données partagées (files, listes, tableaux, structures, …) en mémoire commune (variable gloable), soit utiliser des files de message, ou des pipes.

1. Communication par mémoire partagée

Une ressource partagée est une ressource (LCD, port série, variable partagée, etc) qui peut être utilisée par plus d'une tâche. Chaque tâche doit gagner l'accès exclusif à la ressource partagée pour éviter la corruption des données, on appele ce mécanisme l'exclusion mutuelle qui est facilement implémentée grâce au sémaphore.

Il est nécessaire d'utiliser des sections critiques lorsqu'il y a accès à des ressources partagées par plusieurs threads. En programmation concurrente, une section critique est une portion de code dans laquelle il doit être garanti qu'il n'y aura jamais plus d'un thread simultanément. C'est donc une partie de code qui doit être traitée indivisiblement. Dès qu'une section critique commence son exécution, elle ne peut pas être interrompue. Pour assurer cela, les interruptions sont désactivées avant de rentrer dans le code de la section critique et réactivées quand le code est terminé. Une section critique peut être protégée par un mutex, un sémaphore ou d'autres primitives de programmation concurrente.

Puisqu'à un moment donné, jamais plus d'un thread ne peut être actif dans une section critique, le thread la détenant doit la libérer le plus vite possible pour éviter qu'elle ne devienne un goulot d'étranglement. Libérer la section critique permet d'améliorer les performances en permettant aux threads en attente d'y accéder.

Protection des données partagées par un Mutex

Afin de prévenir tout conflit, elle sera protégée par un MUTEX (MUTual EXclusion object). Un mutex permet de synchroniser différentes tâches en protégeant les structures de données partagées des modifications concurrentes et en implémentant des sections critiques. Le but d'un tel objet est d'éviter tout

risque d'accès simultané aux données partagées par plusieurs processus. Le mutex gère automatiquement son accès par les processus appelants, en fonction de leur priorité notamment.

C'est un sémaphore « amélioré », il gère directement la propriété, les timeout, protège de la destruction (pour éviter des verrous pris et jamais rendu), possède un protocole pour gérer les inversions de priorité. Un mutex peut être soit libre soit pris par une tâche. Il ne peut jamais être pris par deux tâches simultanément. Une tâche tentant de prendre un mutex déjà pris est bloquée jusqu'à ce que la première relâche le mutex.
Un mutex est composé d'une file d'attente, d'une valeur (0 si mutex verrouillé et 1 mutex libre) et d'un pointeur qui pointe sur leur adresse. Il a 2 états possibles : unlocked (appartient à aucune tâche) et locked (possédé par une tâche).

Primitive	Description
int rt_mutex_create (RT_MUTEX *mutex, const char *name)	Créer un Mutex
int rt_mutex_acquire (RT_MUTEX *mutex, RTIME timeout)	Vérrouiller un mutex
int rt_mutex_release (RT_MUTEX *mutex)	Déverrouiller un mutex
int rt_mutex_delete (RT_MUTEX *mutex)	Supprimer un mutex
int rt_mutex_inquire (RT_MUTEX *mutex, RT_MUTEX_INFO *info)	Informations sur le mutex

Paramètres d'entrée des fonctions :

mutex	Pointeur d'adresse du mutex.
name	Pointeur qui permet de nommer le mutex avec une chaine de caractère.
timeout	"TM_INFINITE" pour verrouiller la variable globale. Lorsque le temps infini est spécifié, le mutex est verrouillé.
info	Information de l'alarme comme l'état du mutex(verrouillé ou déverrouillé), le nombre de taches en file d'attente ou leur noms, sous forme d'une structure.

2. Communication par file de messages

Pour transmettre des informations entre des processus ou des périphériques, Xenomai fournit un service de file de messages. Le service « Message Queue» est une méthode par laquelle les tâches temps-réel peuvent échanger ou transmettre des données par le biais d'une file de messages gérée par Xenomai. Les messages sont de taille variable et peuvent être de différents types. Une file de messages peut être créée par une tâche et utilisée par plusieurs tâches qui se transmettent des informations.

Primitive	Description
int rt_queue_create (RT_QUEUE *q, const char *name, size_t poolsize, size_t qlimit, int mode)	Créer une file de messages
int rt_queue_delete (RT_QUEUE *q)	Supprimer une file de messages
void * rt_queue_alloc (RT_QUEUE *q, size_t size)	Allouer une mémoire tampon pour une file de messages
int rt_queue_free (RT_QUEUE *q, void *buf)	Libérer un tampon de file de messages

int rt_queue_send (RT_QUEUE *q, void *mbuf, size_t size, int mode)	Envoyer un message à une file de messages
ssize_t rt_queue_receive (RT_QUEUE *q, void **bufp, RTIME timeout)	Recevoir un message depuis la file de messages
int rt_queue_flush (RT_QUEUE *q) ;	Supprimer tous les messages non lus d'une file de messages
Int rt_queue_inquire (RT_QUEUE *q, RT_QUEUE_INFO *info)	Information sur la file de messages

Paramètres d'entrée des fonctions :

q	Adresse de la file d'attente
Size	Taille en octets de la mémoire tampon pour la file de messages
name	Nom de la file de messages
poolsize	Taille maximale en octets pour chaque message (pré-allocation de la mémoire tampon)
qlimit	Nombre maximale de messages que la file peut supporter
mode	Mode de création de la file de messages
timeout	Temps pour exécuter/attendre
bufp	Pointeur vers un emplacement de mémoire pour recevoir un message
mbuf	L'adresse de la mémoire tampon pour envoyer un message
buf	L'adresse des données du message qui doit être écrit dans la file de messages
info	Information de la file de message sous forme d'une structure

Chapitre 4
La synchronisation

Les tâches temps-réel doivent souvent se synchroniser et se coordonner lors de leur exécution : une tâche doit attendre qu'un traitement se termine, ou qu'une ressource se libère, ou doit s'exécuter dans un ordre précis (notion de précédence).

La synchronisation entre les tâches nécessite un objet particulier qui permet de gérer cette synchronisation.

Dans le cas de l'accès concurrent à des ressources partagées, il existe les sémaphores mutex. Dans les autres cas, il existe les sémaphores (à compteur) ; proposés par Dijkstra en 1965, ils sont une généralisation des verrous et comprennent une variable entière que l'on incrémente/décrémente plutôt qu'un booléen.

Il existe également d'autres moyens comme les drapeaux événements (event flag, associé à un événement unique, il permet de signaler aux tâches la présence d'un événement) ou encore les variables condition (variable qui peut suspendre une tâche jusqu'à ce qu'une condition (prédicat) devienne vraie).

1. Les event flags

Le drapeau événement (event flag) permet simplement de signaler aux tâches la présence d'un événement. L'événement peut être quelconque, défini par le développeur. Par exemple, une ISR est activée lors d'une interruption en provenance d'un périphérique, et peut ainsi informer une tâche en attente que le périphérique est prêt pour livrer des données.

Un drapeau événement est associé à un événement unique Les événements (présents ou non) peuvent être représentés à l'aide d'un tableau de bits (bitmap). Le bit correspondant à un événement particulier est mis à 1 pour "dire" que l'événement s'est produit, ou à 0 dans le cas contraire.

Les tâches peuvent être réveillées en présence d'un événement parmi plusieurs (disjonction), ou seulement si tous les événements sont présents (conjonction). Le mode de réveil est toujours le mode broadcast. La remise à zéro d'un événement se fait de manière explicite.

2. Les variables condition

On appelle variable condition une variable qui peut suspendre une tâche jusqu'à ce qu'une condition (prédicat) devienne vraie.
La suspension de la tâche s'effectue à l'aide de l'opération rt_cond_wait(var), alors qu'une autre tâche utilisera l'opération rt_cond_signal(var)pour réveiller la tâche en attente.
Une variable condition n'est modifiable qu'avec l'utilisation des primitives rt_cond_wait()et rt_cond_signal().

- rt_cond_wait(x)bloque l'exécution de la tâche sur la condition x.
    - La tâche reprendra l'exécution seulement si une autre exécute rt_cond_signal(x).

    - rt_cond_wait(x) est toujours bloquant.
- rt_cond_signal(x) reprend l'exécution d'une tâche bloquée sur la condition x.
    - S'il existe une tâche en attente, elle sera débloquée (passage à READY).
        - S'il existe plusieurs tâches en attente, c'est la politique de la file d'attente associée à la variable condition qui "dira" quelle tâche doit être exécutée; les autres tâches restent dans l'état WAITING.
        - L'opération n'est valable que pour une seule tâche (mode de réveil normal).
        - Il existe une autre fonction (rt_cond_broadcast(x)) pour le mode broadcast.
    - S'il n'en existe pas : ne rien faire, l'événement est considéré comme perdu (pas de mémorisation du signal de l'événement).

L'appel à rt_cond_wait() est forcément bloquant

3. Les sémaphores

Le sémaphore est un objet utilisé par l'ensemble des tâches concernées par la synchronisation; cet objet est capable de "réveiller" une ou plusieurs tâches en présence d'un événement (généré par une tâche ou une ISR). Le mode de réveil des tâches peut consister à réveiller une seule tâche (mode normal), ou à réveiller l'ensemble des tâches en attente (mode broadcast).

A la différence d'un mutex, un sémaphore n'est pas détenu par une tâche (en effet, plusieurs tâches peuvent entrer dans la section critique, en utilisant le même sémaphore).

Chaque sémaphore dispose de sa propre file d'attente. Plusieurs tâches peuvent utiliser le même sémaphore. Chaque sémaphore dispose de sa propre file d'attente (FIFO ou "par priorité"). La politique de la file d'attente peut être celle de l'ordonnanceur.

Trois opérations de base :

- rt_sem_create(&sema, .., size, ...) : création d'un sémaphore d'identificateur sema avec size entrées
- rt_sem_p(&sema, ...) : attendre sur le sémaphore sema (Puis-je?)
- rt_sem_v(&sema) : signaler le sémaphore sema (Vas-y!)

L'appel à rt_sem_p() est potentiellement bloquant.

Les primitives associées aux sémaphores sont :

Primitive	Description
int rt_sem_create (RT_SEM * sem, const char * name, unsigned long icount, int mode)	Créer un sémaphore
int rt_sem_delete (RT_SEM * sem)	Supprimer un sémaphore
int rt_sem_p (RT_SEM * sem , RTIME timeout) Lorsque cette fonction est réalisée, la valeur du compteur du sémaphore se décrémente de 1	Verrouiller un sémaphore
int rt_sem_v (RT_SEM * sem) Lorsque cette fonction est réalisée, la valeur du compteur du sémaphore	Déverrouiller un sémaphore

s'incrémente de 1.	
int rt_sem_inquire (RT_SEM * sem , RT_SEM_INFO * info)	Informations sur le sémaphore

Paramètres d'entrée des fonctions :

sem	Sémaphore de type RT_SEM
name	Nom que l'on veut donner au sémaphore
icount	Valeur initiale du compteur
mode	Mode dans lequel le sémaphore va gérer les threads (2 modes principaux : mode = S_FIFO pour se mettre en FIFO, mode = S_PRIO pour que le sémaphore accorde l'accès selon la priorité de la tâche à exécuter)
timeout	Temps pour exécuter/attendre : mettre à la valeur TM_INFINITE ou TM_NONBLOCK
info	Information du sémaphore sous forme d'une structure

Chapitre 5
Management des timers

Les alarmes sont des timers généraux. N'importe quelle tâche Xenomai peut créer des alarmes et les utiliser avec un handler défini par l'utilisateur. Ce handler est alors exécuté après que le délai initial spécifié s'est écoulé. Ces alarmes peuvent être soit one shot ou périodique. Dans le cas où une alarme est périodique, le noyau temps-réel la reprogramme automatiquement pour la prochaine exécution.

Les watchdog timers n'appellent pas un handler lorsqu'ils expirent, mais débloquent une tâche en attente.

Tableau des primitives lié au watchdog :

Primitive	Description
int rt_alarm_create (RT_ALARM * alarm, const char * name)	Créer une alarme
int rt_alarm_start (RT_ALARM * alarm, RTIME value, RTIME interval)	Démarrer une alarme
int rt_alarm_stop (RT_ALARM * alarm)	Désarmer une alarme
int rt_alarm_wait (RT_ALARM * alarm)	Attente du prochain coup d'alarme
int rt_alarm_inquire (RT_ALARM * alarm, RT_ALARM_INFO * info)	Informations sur l'alarme
int rt_alarm_delete (RT_ALARM * alarm)	Suppression d'une alarme

Paramètres d'entrée des fonctions :

alarm	Adresse de l'alarme
name	Nom de l'alarme
value	Date de l'alarme initiale
interval	Période de rechargement de l'alarme
info	Information de l'alarme sous forme d'une structure

Bibliographie

Travaux tutorés des 4AE et 4IR 2012

David Chabal et Claude LELOUP, Premiers pas avec Xenomai, février 2012

http://www.xenomai.org/documentation/xenomai-2.4/pdf/native-api.pdf