

- PROJET RTEL4I -



Spécifications fonctionnelles des extensions temps réel de Linux (L2.2-a)





Spécifications
fonctionnelles des
extensions temps réel

Version: 1.0
Auteur: Pierre Ficheux

MODIFICATIONS

| <i>VERSION</i> | <i>DATE</i> | <i>AUTEUR(S)</i> | <i>DESCRIPTION</i> |
|-----------------------|--------------------|-------------------------|---------------------------|
| 1.0 | 15/01/2009 | P. Ficheux | Création |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

Table des matières

| | |
|--|----|
| 1.Contexte du document..... | 5 |
| 2.Architectures utilisées dans l'embarqué..... | 5 |
| 2.1.L'architecture x86..... | 5 |
| 2.2.L'architecture ARM..... | 6 |
| 2.3.L'architecture PowerPC..... | 7 |
| 2.4.Notion de MMU..... | 7 |
| 2.5.Les autres architectures..... | 7 |
| 2.6.Le co-design et les FPGA..... | 8 |
| Figure 1. Carte Armadeus APF27..... | 8 |
| 2.7.Conclusions sur les choix d'architecture..... | 8 |
| 3.Linux et le temps réel..... | 9 |
| 3.1.Description de l'expérience..... | 10 |
| 3.2.Développement du programme de test..... | 10 |
| 3.3.Configuration des options de préemption..... | 14 |
| 3.4.Modification de la politique d'ordonnancement..... | 15 |
| 3.5.Modification du noyau Linux..... | 16 |
| 3.5.1.Les contraintes fonctionnelles..... | 17 |
| 3.5.2.Tableau récapitulatif des critères..... | 18 |
| 3.5.3.Le patch PREEMPT-RT..... | 19 |
| 3.5.4.Les solutions à base de co-noyau..... | 20 |
| 4.Conclusions générales..... | 30 |

Index des figures

| | |
|--|----|
| Figure 1. Carte Armadeus APF27..... | 8 |
| Figure 2. Test temps réel sur port parallèle..... | 9 |
| Figure 3. Affichage de la courbe avec un système non chargé..... | 13 |
| Figure 4. Courbe désynchronisée avec un système chargé..... | 14 |
| Figure 5. Les deux espaces de mémoire de Linux..... | 16 |
| Figure 6. Architecture RTLinux (et RTAI)..... | 21 |
| Figure 7. Architecture simplifiée de Xenomai..... | 23 |
| Figure 8. Architecture détaillée de Xenomai..... | 24 |
| Figure 9. Principe du pipe-line ADEOS..... | 25 |
| Figure 10. Répartition de l'application sur 2 domaines..... | 26 |
| Figure 11. Lien de l'application avec les noyaux (cas de POSIX et VxWorks). 26 | |
| Figure 12. Migration de domaine..... | 27 |

1. Contexte du document

Ce document a pour but de décrire les architectures sur lesquelles les extensions temps réels utilisées par RTE4I seront portées. Le choix des extensions sera également décrit et justifié dans ce même document.

2. Architectures utilisées dans l'embarqué

Dans ce paragraphe nous décrirons les principales architectures utilisées pour les applications embarquées. Il est important de noter que certains éléments pris en compte lors de la préparation du projet (2008) ont pas mal évolués. Le principal élément notoire est l'entrée en force d'Intel dans les applications embarquées grâce à sa gamme Atom.

2.1. L'architecture x86

Le monde de l'informatique est dominé par l'architecture PC/x86, héritée des systèmes personnels initialement développés par IBM au début des années 80. En dépit des nombreux détracteurs et autres analystes qui prévoient toujours sa fin proche, l'architecture PC/x86 a su s'adapter à l'évolution technologique, en balayant sur son passage bon nombre de concurrents pourtant présentés comme révolutionnaires. Dans la dernière édition de l'ouvrage, nous citons Apple comme seule exception à l'utilisation massive du x86. Ce n'est plus le cas aujourd'hui, tous les ordinateurs Apple (à l'exception des produits mobiles) étant désormais fournis avec un processeur Intel. Les anciennes versions PowerPC ne sont plus compatibles avec la nouvelle mouture de Mac OS X (*Snow Leopard*).

Le PC/x86 est depuis le début une architecture *ouverte*, ce qui a permis très rapidement de développer des machines compatibles à bas prix, et donc d'imposer l'architecture sur un grand nombre de marchés, du très bas de gamme aux systèmes professionnels. Par opposition, d'autres concurrents ont jalousement gardé leurs secrets et interdit la réalisation de clones, ce qui a limité la diffusion à des marchés plutôt haut de gamme. Ce qui est vrai pour l'architecture de base (la carte mère) l'est encore plus pour les périphériques comme les cartes d'extension, vendues souvent à des prix prohibitifs sur les architectures non dérivées des standards x86. Toutes les architectures ont désormais adopté des standards du monde x86, comme le bus PCI (pour *Peripheral Component Interconnect*), permettant la disponibilité de périphériques très compétitifs.

Le PC/x86 a longtemps été associé – exclusivement ou presque – aux systèmes d'exploitation de Microsoft. Ces derniers sont depuis longtemps les plus répandus, et ils existent presque exclusivement sur architecture x86 (sauf pour les systèmes embarqués comme Windows CE). De même, les systèmes d'exploitation libres comme Linux ont également comme référence l'architecture x86.

Enfin, le PC est depuis toujours associé au processeur Intel ou compatible. Intel s'est longtemps comporté en « Microsoft du processeur », agissant en quasi-hégémonie sur ce marché. Il est aujourd'hui talonné par d'autres fondeurs comme AMD, mais ces derniers se doivent de fournir des pro-

cesseurs les plus compatibles possibles tout en restant plus performants que ceux d'Intel. Désormais, on peut considérer que la messe est dite et qu'Intel a pour longtemps gagné la bataille, surtout avec l'arrivée du processeur Atom équipant les NetPC qui, dès la première année, avaient acquis près de 15 % du marché des PC.

REMARQUE: Des éléments récents nous prouvent que le processeur Atom ne se limite plus au marché du PC grand public, de nombreux équipements étant désormais basés sur cette architecture (exemple: la nouvelle set-top box de Free).

2.2.L'architecture ARM

L'architecture x86 n'est pas forcément la meilleure pour toutes les applications, loin s'en faut. Dans le cas de la production d'une série de cartes mères assez importante (plusieurs milliers d'exemplaires) ou d'un besoin particulier (faible consommation, intégration), d'autres processeurs sont plus avantageux et moins complexes à mettre en œuvre. L'architecture ARM est originale car la société ARM Ltd (<http://www.arm.com>) ne commercialise pas de processeur mais vend uniquement des licences à des fabricants célèbres (Atmel, Freescale, ST et même Intel). Le principal avantage de cette architecture est le rapport performances/consommation qui est un point essentiel, surtout pour les applications embarquées.

Actuellement, l'architecture ARM est la plus utilisée pour les applications embarquées, suivie par x86 et PowerPC. D'autres architectures comme MIPS ou SH4 ont des utilisations plus dédiées et sont souvent intégrées dans des *chipsets* fournissant d'autres fonctions matérielles. Citons en exemple les produits Broadcom (<http://www.broadcom.com>), spécialisés dans les applications de communication (IAD pour *Internet Access Device*), ou STMicroelectronics (<http://www.st.com>), qui fournit des solutions pour les *set-top boxes* basées sur des processeurs SH4. L'offre de STMicroelectronics est cependant en cours d'évolution vers ARM.

Les processeurs ARM ont un bon niveau d'intégration et une faible consommation et permettent donc la conception de cartes mères plus simples et moins coûteuses que les x86. L'architecture se décline en ARM7 (sans MMU), ARM9 et plus récemment ARM11. L'architecture ARM9 est peu coûteuse et très utilisée sous diverses marques. Citons :

- les produits AT91RM9200 (http://www.atmel.com/dyn/products/product_card.asp?part_id=2983) et plus récemment SAM926x de chez Atmel (<http://www.at91.com/component/resource/article/products/9-sam9/557-sam9263.html>)
- les produits i.MX de chez Freescale (http://www.freescale.com/webapp/sps/site/homepage.jsp?code=IMX_HOME), également fournisseur de processeurs PowerPC
- le processeur S3C24xx de chez SAMSUNG (http://www.samsung.com/global/business/semiconductor/productInfo.do?fmly_id=836&partnum=S3C2440)
- les processeurs à base de cœur Cortex-A8 ou A9. Ces derniers représentent la dernière gamme disponible et sont proposés par des fabricants comme TI (gamme OMAP). Ils sont en général très bien supportés par Linux (exemple: <http://gumstix.org>).

2.3.L'architecture PowerPC

En ce qui concerne des architectures de même niveau de complexité, la gamme PowerPC co-développée par IBM et Motorola est encore très utilisée dans le monde industriel, en raison de son très bon rapport consommation/performances et de la relative simplicité de son architecture par rapport au x86. En effet, quasiment toutes les cartes industrielles utilisées dans les transports (aéronautique, ferroviaire) sont basées sur l'architecture PowerPC. Cependant l'architecture semble quelque peu délaissée par son principal constructeur Freescale (ex-Motorola) qui s'est fortement impliquée sur l'offre ARM avec la gamme i.MX.

2.4.Notion de MMU

Le noyau Linux a été initialement développé sur la base du mécanisme de protection de mémoire du processeur Intel 386/486. Ce mécanisme repose sur un composant matériel appelé MMU (pour *Memory Management Unit*, c'est-à-dire unité de gestion mémoire) et permet à un processus de ne jamais écraser l'espace mémoire d'un autre processus. La MMU autorise la conversion entre les adresses physiques – adresses effectivement utilisées dans la machine – et les adresses virtuelles – adresses vues par le processus et allouées par le système d'exploitation. Si un processus tente de sortir par erreur de l'espace mémoire qui lui est accordé, la MMU détecte l'erreur et stoppe le programme en générant une erreur de « violation de segmentation » (le fameux *segmentation violation* associé au signal `SIGSEGV`). De ce fait, un programme tournant sous Linux dans l'espace dit « utilisateur » – par opposition à l'espace « noyau » – ne peut jamais corrompre le système.

Les versions courantes du noyau Linux sont prévues pour fonctionner sur des processeurs avec MMU, ce qui concerne la majorité des processeurs utilisés dans l'informatique classique, et aussi dans un bon nombre d'applications embarquées. En revanche, ces processeurs sont en général plus onéreux et plus gourmands en ressources matérielles, et certaines applications dites « profondément enfouies » (ou *deeply embedded*) ne pourront utiliser que des processeurs dépourvus de MMU.

Il existe des systèmes d'exploitation dédiés à ces micro-contrôleurs – μ C/OS en est un très bon exemple – mais ceux-ci sont en général bien plus limités que Linux au niveau des protocoles standards et de l'interopérabilité avec le monde extérieur. Autre point, la gestion de la MMU est complexe, et de ce fait la majorité des RTOS propriétaires comme VxWorks ne l'utilisent pas par défaut pour des raisons de performances. De même, la mise au point d'un système est plus simple dans le cas où les programmes et le noyau se partagent le même espace de mémoire.

2.5.Les autres architectures

Nous avons évoqué les architectures SH4 et MIPS. Celles-ci sont dans une moindre mesure assez fréquemment utilisées pour des applications embarquées.

2.6. Le co-design et les FPGA

Ce domaine était déjà présent en 2008 mais il a connu une très forte croissance depuis. On peut dire que la majorité des projets industriels utilisent des architectures de ce type car cela permet une grande souplesse dans le développement à la fois logiciel mais aussi matériel car les fonctionnalités de la carte peuvent être modifiées aussi aisément que le logiciel embarqué exécuté. Les deux principaux fabricants de FPGA (Altera et Xilinx) fournissent un support Linux de très bonne qualité. De ce fait, certaines parties de l'application peuvent être migrées vers la partie FPGA pour des raisons de performances ou de propriété intellectuelle. Les livrables L2.5b décrit une application vidéo basée sur une telle architecture et utilisant un processeur NIOS2.

Certains constructeurs de cartes embarquées fournissent désormais un FPGA en plus du processeur ARM (en général), citons l'exemple des cartes Armadeus, de conception française et développées nativement pour Linux (voir <http://www.armadeus.com>). Ces cartes intègrent un FPGA Spartan-3 de chez Xilinx.



Figure 1. Carte Armadeus APF27

Du fait de la souplesse de la conception, ces architectures permettent de fonctionner avec ou sans MMU même si de nos jours la majorité des applications sont basés sur des processeurs avec MMU.

REMARQUE: La partie temps réel n'était pas supportée de manière correcte par aucun des deux constructeurs. Un des points importants fut le développement des extensions pour ces architectures, voir les livrables L2.3.

2.7. Conclusions sur les choix d'architecture

Les différents éléments nous ont permis de choisir les travaux les plus judicieux ainsi que le planning de réalisation. Les architectures retenues sont donc les suivantes :

- Intel Atom (x86)
- Xilinx Microblaze

- Altera NIOS2
- ARM (divers constructeurs dont Atmel et SAMSUNG)
- ST Microelectronics SH4

Bien entendu la complexité peut varier suivant les architectures. Dans certains cas il s'agit d'adapter des composants logiciels existants (x86, ARM). Dans d'autres cas, il est nécessaire de développer presque entièrement le support de l'architecture (Xilinx, Altera).

Le cas de ARM est assez particulier, car plusieurs constructeurs peuvent utiliser le même cœur (exemple: ARM920 pour Atmel et SAMSUNG) mais une adaptation est nécessaire pour chaque fabricant car le fonctionnement dépend également des périphériques du processeur (exemple: contrôleur d'interruption) qui lui peut varier suivant le constructeur.

3.Linux et le temps réel

Dans ce paragraphe nous allons rappeler quelques éléments concernant l'utilisation d'un système Linux pour des applications temps réel embarquées. Linux est un système POSIX, il dispose donc des API nécessaires pour programmer des applications temps réel.

- une bibliothèque de gestion de processus léger ou *thread* (norme 1003.1c-1995) ; dans le cas d'un système POSIX, la notion de processus léger correspond à celle de tâche élémentaire, un processus (au sens UNIX) pouvant être composé de plusieurs *threads* ;
- la prise en compte des API de programmation temps réel : ordonnancement, compteurs, sémaphores, etc. (norme 1003.1b-1993).

L'exemple présenté permet de mettre en place une tâche périodique qui écrit régulièrement sur le port parallèle d'un PC x86. Le port parallèle a un léger parfum de suranné, mais c'est un excellent périphérique de test, peu coûteux et encore présent sur bon nombre de PC de laboratoire, même s'il a effectivement disparu des PC multimédia au profit des interfaces USB. Le schéma ci-dessous décrit l'expérience mise en place.

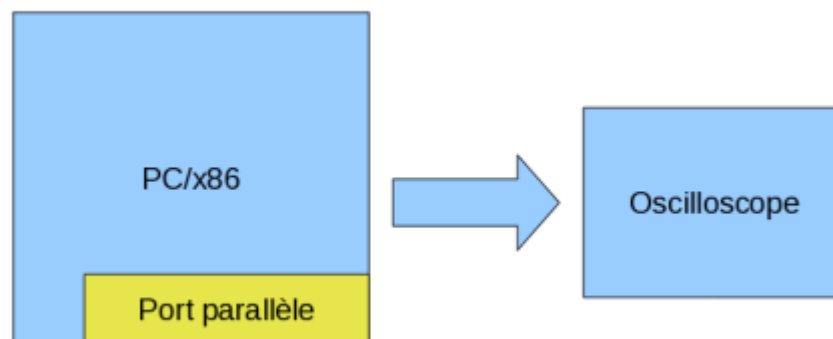


Figure 2. Test temps réel sur port parallèle

3.1. Description de l'expérience

Le principe de l'expérience est simple :

1. Un programme tournant sous Linux sollicite le port parallèle du PC en écrivant périodiquement les valeurs 0 puis 0xFF dans le registre de données, situé en général à l'adresse 0x378. Cela a pour effet de faire varier le niveau des signaux D0 à D7 du port entre les niveaux logiques 0 et 1. L'adresse du registre de données peut être vérifiée à l'aide du contenu du fichier virtuel `/proc/ioports`.
2. Le résultat est affiché sur un oscilloscope relié au signal D0 et à la masse du connecteur. En utilisant un oscilloscope disposant d'un affichage persistant, on peut visualiser les performances des différents programmes de test développés.

Si le système respecte scrupuleusement la périodicité, l'affichage persistant montrera une seule courbe. Dans le cas contraire, on pourra observer un délai entre la période théorique de la courbe et la période réelle. On parle alors de *gigue* (ou *jitter*) ou bien de « temps de latence », abusivement nommé « latence » ou *latency* en anglais.

3.2. Développement du programme de test

Le programme de test à écrire est une tâche périodique et l'on doit donc programmer un compteur. Dès que ce compteur arrive à échéance, on procède à l'écriture de la valeur sur le port, puis à l'inversion logique de la valeur à écrire (0 ou 0xFF). Pour cela, le plus simple est d'utiliser l'API POSIX V.4 proposée par Linux.

- On crée le compteur avec la fonction `timer_create`.
- La période du compteur est initialisée par `time_settime`.
- À l'échéance du compteur, on reçoit un signal `SIGALRM`. La fonction de traitement de ce signal nous permet d'écrire sur le registre de données et de modifier la valeur pour la prochaine écriture.

Pour écrire sur le registre de données, il y a deux solutions.

- La solution la plus portable consiste à utiliser un pilote de périphérique. Le noyau Linux fournit un pilote du port parallèle, associé au fichier spécial `/dev/parport0`.
- Une autre solution est d'accéder directement au registre en utilisant la fonction `outb`, même si cette solution n'est pas portable, car liée à l'architecture x86. Notons que dans ce cas, il est nécessaire d'exécuter le programme de test en tant que super-utilisateur et d'utiliser l'appel système `ioperm` pour obtenir le droit d'accès au registre de données. Nous reproduisons ci-après le code source du programme.

Déclaration des en-têtes et paramètres globaux

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // Nécessaire pour timer_create/timer_settime
```

```
#include <signal.h>
#include <sys/io.h> // Nécessaire pour outb

// Adresse du port parallèle
#define LPT          0x378

// Déclaration du compteur
timer_t my_timer;

// Valeur à écrire sur le registre de données
int nibl;
```

Les fonctions suivantes sont associées aux signaux **SIGALRM** et **SIGINT**. Ce dernier est lié à l'interruption du programme par l'utilisateur (**Ctrl-C**) car le test est une boucle infinie.

Fonctions de traitement des signaux

```
// Prise en compte de l'arrêt par l'utilisateur (Ctrl-C)
void got_sigint (int sig)
{
    timer_delete (my_timer);
    exit (0);
}

// Fonction exécutée à l'échéance du compteur
void got_sigalrm (int sig)
{
    // Écriture de la valeur sur le registre de données
    outb (nibl, LPT);

    // Changement d'état de la valeur 0 <--> 0xFF
    nibl = ~nibl;
}
}
```

La partie principale se charge de l'initialisation des fonctions de traitement des signaux et du compteur. Elle se termine par une boucle infinie d'attente de signal (**SIGINT** ou **SIGALRM**).

Partie principale du programme

```
main (int ac, char **av)
{
    struct itimerspec new, old;

    // Affectation des "handlers" de signaux
    signal (SIGALRM, got_sigalrm);
    signal (SIGINT, got_sigint);

    // Demande l'accès au registre de données du port parallèle
    ioperm (LPT, 1, 1);

    // Création du compteur
    timer_create (CLOCK_REALTIME, NULL, &my_timer)

    // Programme une période de 5 ms pour le compteur (it_interval) la
    // valeur étant programmé en microsecondes
    // ATTENTION : ne pas mettre 0 dans it_value car cela correspond à
    // l'arrêt du compteur
    new.it_value.tv_sec = 0;
```

```
new.it_value.tv_nsec = 5000000; /* Démarrage dans 5 ms */
new.it_interval.tv_sec = 0;
new.it_interval.tv_nsec = 5000000; /* Période = 5 ms */

// Démarrage du compteur
timer_settime (my_timer, 0, &new, &old);

// Attente des signaux
while (1)
    pause ();
}
```

Le programme doit être compilé en utilisant la bibliothèque `librt`. Nous rappelons qu'il doit être exécuté en tant que super-utilisateur à cause de l'accès direct au registre de données.

Exécution et test du programme

```
$ gcc -O2 -o rt_square rt_square.o -lrt
# ./rt_square
```

Si l'on observe le signal à l'aide de l'oscilloscope et que le système n'est pas chargé, on constate comme prévu que la courbe est (relativement) régulière, avec une périodicité de 5 ms. Cette valeur est relativement peu contraignante dans le cas d'une application temps réel ou les ordres de grandeur sont plutôt de l'ordre de la centaine, voire dizaine de microsecondes.

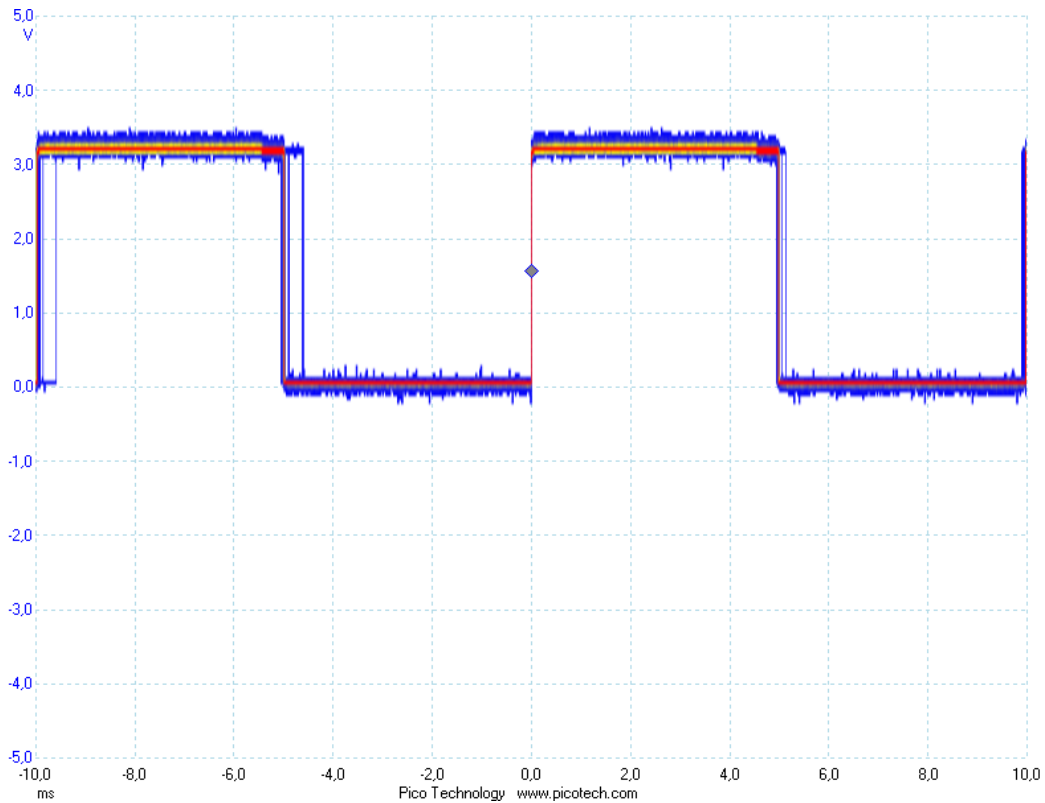


Figure 3. Affichage de la courbe avec un système non chargé

Par contre, si l'on charge le système, on observe des temps de latence qui dépendent de la charge et, bien entendu, de la puissance du système. Sur un système peu puissant, on peut arriver à une courbe totalement irrégulière. Sur un système plus puissant, on pourra observer la trace de la latence en mettant l'oscilloscope en mode persistant et en le synchronisant sur le front montant de la courbe.

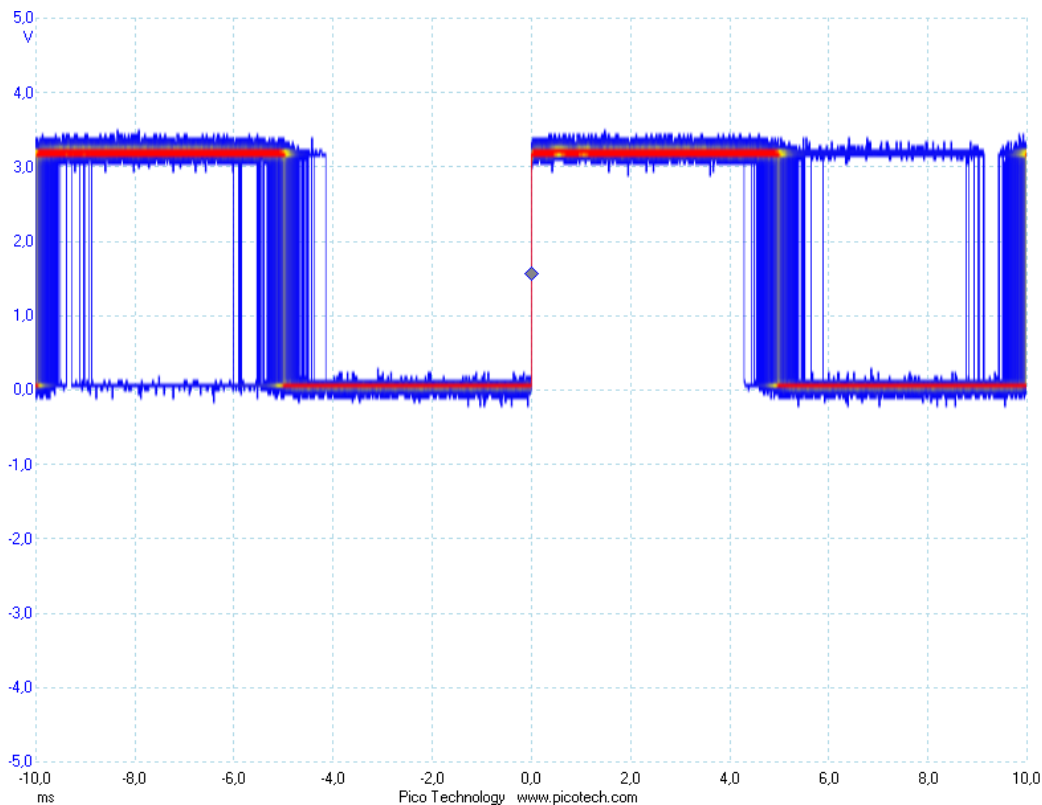


Figure 4. Courbe désynchronisée avec un système chargé

3.3. Configuration des options de préemption

Le noyau Linux 2.6 inclut des options de *préemption* héritées des patches publiés à l'époque du noyau 2.4.

- le patch Preempt-Kernel de Robert M. Love (`CONFIG_PREEMPT_VOLUNTARY`)
- le patch Low-Latency d'Andrew Morton (`CONFIG_PREEMPT`)

Les options sont accessibles dans la configuration du noyau Linux, dans le menu **Processor type and features>Preemption Model**.

La principale différence entre les deux options est l'utilisation de points de préemption « explicites » dans le cas de `CONFIG_PREEMPT_VOLUNTARY`. Ces points sont placés à des endroits stratégiques du noyau afin de réduire le temps de latence de l'ordonnanceur (ou *rescheduling latency*). Dans le cas de `CONFIG_PREEMPT`, l'approche est plus globale en rendant le noyau plus préemptif, à l'exception des sections critiques. Cependant, la valeur de latence moyenne annoncée est de l'ordre de la milliseconde, ce qui est finalement peu différent du comportement du noyau standard. Concrètement, ces

options réduisent les valeurs maximales de latence et améliorent de manière statistique le temps de réponse du noyau, mais ne rendent pas le système entièrement préemptif, les résultats obtenus étant très proches de ceux d'un Linux standard.

3.4. Modification de la politique d'ordonnancement

On peut également améliorer les performances en modifiant le comportement de l'ordonnanceur pour le programme de test. Par défaut, un processus sous Linux est traité avec un politique de « temps partagé » (ou *time-sharing*), ce qui signifie que toutes les tâches ont visiblement la même priorité, et surtout que celle-ci est dynamique. Cela explique le résultat observé, car en cas de charge du système, le programme de test n'est pas plus prioritaire qu'un accès disque ou réseau, ou bien qu'un affichage graphique. Ce mode par défaut correspond à la politique `SCHED_OTHER` de l'ordonnanceur et la valeur de priorité 0.

Il existe pour le noyau Linux deux autres politiques d'ordonnancement, proches du temps réel. Ces modes utilisent des priorités fixes comprises entre 1 et 99.

- La politique `SCHED_FIFO` utilise un système à priorité fixe basé sur le principe du « premier arrivé, premier servi ». Le processus sera exécuté jusqu'à ce qu'il soit bloqué par une entrée/sortie ou préempté par un processus de priorité supérieure.
- La politique `SCHED_RR` utilise un système à priorité fixe basé sur un algorithme de tourniquet (ou *round-robin*). C'est une amélioration du cas précédent et qui affecte une tranche temporelle limitée à chaque processus.

On peut donc améliorer le comportement de notre programme de test en indiquant qu'il fonctionne en `SCHED_FIFO` ou `SCHED_RR`, et en lui affectant la priorité maximale de 99. Pour cela, on peut utiliser la fonction ci-dessous.

Modification de la politique en `SCHED_FIFO`

```
int set_realtime_priority(void)
{
    struct sched_param schp;

    memset(&schp, 0, sizeof(schp));

    // Lecture de la priorité maximale disponible (en général 99)
    schp.sched_priority = sched_get_priority_max(SCHED_FIFO);

    // Affectation de la politique SCHED_FIFO + priorité
    if (sched_setscheduler(0, SCHED_FIFO, &schp) != 0) {
        perror("sched_setscheduler");
        exit(1);
    }

    return 0;
}
```

Si l'on ajoute l'appel à cette fonction au début de programme de test, on peut constater que l'affichage est stable malgré la charge du système. Cependant, cela ne change pas le fait que le noyau Linux n'est pas préemptif. En l'occurrence, on ne pourra pas gérer correctement un grand nombre de

tâches concurrentes en utilisant `SCHED_FIFO` ou `SCHED_RR`. Notons également que notre exemple n'est pas si contraignant pour le système, sachant que la base de temps logicielle du noyau Linux, définie par la constante `HZ`, varie entre 1 et 10 ms pour le noyau 2.6 et ce en fonction des architectures matérielles.

3.5.Modification du noyau Linux

L'expérience réalisée sur un noyau Linux standard démontre qu'il n'est pas utilisable dans un véritable contexte temps réel dur. Dans le cadre du document L2.1 (cas d'utilisation et exigences applicables à Linux temps réel), il est donc nécessaire d'apporter des modifications plus importantes au noyau afin d'obtenir des performances temps réel « dures ».

Les options de « préemption » décrites au paragraphe précédent ne sont pas suffisante car elles ont des limitations importantes suivant la configuration du système (absence de préemption dans le cas du multi-processeur ou dans une routine d'interruption).

Dans le cas d'un environnement Linux, l'espace de mémoire est divisé en deux (utilisateur et noyau), ce qui rend plus complexe la mise en place d'une solution temps réel, sachant que le passage d'un espace à l'autre a forcément des conséquences sur les performances. Cette séparation n'existe pas dans le cas des autres RTOS.

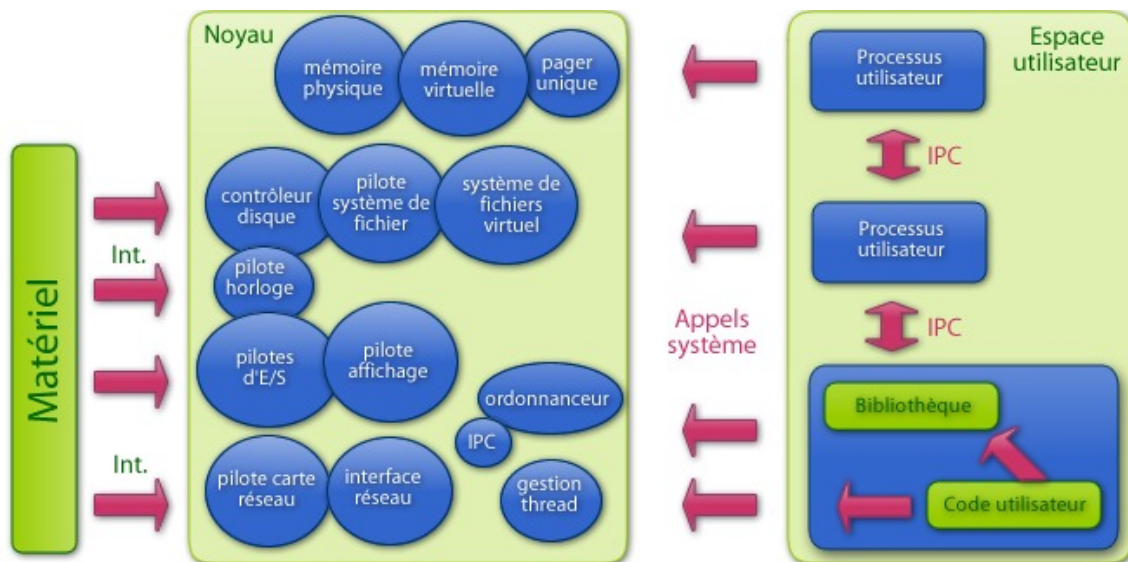


Figure 5. Les deux espaces de mémoire de Linux

Il existe deux manières d'améliorer les performances temps réel du système :

- Modifier le fonctionnement du noyau Linux lui-même, ce qui est (partiellement) accompli par les options de préemption décrites précédemment. Cette solution est utilisée par le patch PREEMPT-RT.

- Utiliser le service d'un « co-noyau » temps réel. Cette solution est utilisée par RTLinux, RTAI et Xenomai. RTLinux étant désormais propriétaire (édité par Wind River), les solutions envisageables sont RTAI et Xenomai.

Dans la suite du document, nous décrirons les caractéristiques des extensions envisageables en fonction des contraintes fonctionnelles.

3.5.1. Les contraintes fonctionnelles

Les principales contraintes fonctionnelles sont liées à différents que nous explicitons ci-après. Nous avons rassemblé des notes sur 5 dans un tableau récapitulatif d'évaluer les meilleures extensions parmi les principaux choix disponibles. Les 9 contraintes fonctionnelles sont numérotées CF1 à CF9.

Simplicité d'utilisation (CF1)

Suivant les techniques utilisées, les extensions temps réel nécessiteront ou non un apprentissage supplémentaire pour l'installation et/ou le développement d'application.

Portabilité des extensions temps réel (CF2)

Ce point concerne la liste des architectures effectivement supportées en fonction des architectures matérielles sélectionnées. En effet, les systèmes embarqués ont une longue durée de vie (plusieurs années) et il est important de pouvoir adapter facilement une application existante sur une nouvelle architecture.

Compatibilité de l'extension avec le noyau standard (CF3)

Ce point concerne la disponibilité des différents patch en fonction des évolutions des versions de noyau. Il est difficile d'assurer une parfaite synchronisation des évolutions mais il est important que le patch ne soit pas lié à une version spéciale du noyau (exemple: une version fournie par un fabricant de carte ou de processeur).

Performances (CF4)

Ce point correspond aux performances accessibles en valeur de « latence » et de « jitter ». La latence est une valeur prévisible de prise en compte d'un événement. Le « jitter » correspond à la variation (souvent imprévisible) de la latence.

Interfaces de développement (CF5)

Ce point concerne les interfaces de développement disponibles. La présence d'une API POSIX est indispensable. Il est également important de disposer d'une API de développement clairement définie pour des pilotes temps réel.

Nous avons déjà évoqué la durée de vie assez longue du logiciel embarqué. Il est parfois important de pouvoir importer du code existant (exemple: du code VxWorks) dans une nouvelle version de l'application en utilisant des interfaces de développement d'adaptation « wrapper ».

Outils de mise au point (CF6)

La mise au point d'un programme temps réel est complexe. Un critère de choix est la disponibilité d'outils de mise au point mais aussi la possibilité de développer de nouveau outil ou bien d'adapter des outils existants à l'extension temps réel.

Documentation disponible (CF7)

De nombreux projets libres disposent d'une documentation assez pauvre ou bien pas forcément à jour. L'existence d'une documentation complète est un critère de choix important pour une utilisation dans un environnement industriel.

Compatibilité des licences (CF8)

La prise en compte des licences – et des contraintes juridiques en général - est un point fondamental dans le cas d'une utilisation industrielle. En particulier, la licence GPL impose la distribution du code source des composants fournis. Cette approche n'est pas forcément acceptable pour un industriel pour des raisons de propriété intellectuelle. Un critère de choix est donc l'influence de la prise en compte des licences sur l'architecture du système. Par exemple, si une tâche temps réel est développée en espace noyau, son code source doit être en théorie diffusé sous GPL.


Dynamisme de la communauté (CF9)

C'est un point essentiel dans le cas de l'utilisation d'un composant libre. En effet, si il est envisageable d'adapter une composant à une architecture donnée, il est également que cette adaptation soit à terme intégrée au projet ou a défaut que cette communauté puisse fournir un support réactif pour l'évolution du portage réalisé.

N'oublions pas que le but de RTEL4I final n'est pas de se substituer au communautés de développement mais plutôt de fédérer les travaux de plusieurs communautés (Eclipse, Xenomai, PREEMPT-RT, etc.) afin de faciliter de travail de l'utilisateur final en réalisant – entre autres - des « connecteurs » entre les composants.

3.5.2. Tableau récapitulatif des critères

| Critère | PREEMPT-RT | RTAI | Xenomai |
|--------------------------|------------|------|---------|
| Simplicité d'utilisation | 5 | 2 | 3 |
| Portabilité | 3 | 1 | 3 |

| | | |
|---|---|--|
|  | Spécifications fonctionnelles des extensions temps réel | Version: 1.0 Auteur: Pierre Ficheux |
|---|---|--|

| | | | |
|--------------------------------------|-----------|-----------|-----------|
| Compatibilité avec le noyau standard | 2 | 2 | 3 |
| Performances | 2 | 4 | 4 |
| Interfaces de développement | 4 | 2 | 3 |
| Outils de mis au point | 4 | 2 | 3 |
| Documentation | 4 | 2 | 4 |
| Compatibilité des licences | 4 | 3 | 4 |
| Dynamisme de la communauté | 4 | 2 | 5 |
| Total | 32 | 20 | 33 |

3.5.3. Le patch PREEMPT-RT

Contrairement à ce que l'on peut penser, l'extension PREEMPT-RT n'a rien à voir avec le patch Preempt-Kernel hérité du noyau 2.4. PREEMPT-RT est un patch expérimental (donc non inclus au noyau officiel) qui permet d'utiliser le noyau Linux 2.6 dans le cadre d'applications temps réel *dures*.

Le développement fut démarré initialement par Ingo Molnar et Thomas Gleixner. La principale source d'informations concernant PREEMPT-RT est le site RT Wiki sur https://rt.wiki.kernel.org/index.php/Main_Page.

Nous pouvons résumer ci-après les modifications effectuées sur le noyau Linux. Grâce à ces améliorations, on obtient au final un noyau Linux préemptif dans sa quasi-totalité.

- prise en compte des interruptions par des *threads* en espace noyau (on parle de *threaded interrupts model*), ce qui autorise la préemption des routines de traitement d'interruption (ISR) qui n'est pas possible dans le noyau standard ;
- prévention des inversions de priorité, en mettant en place un héritage de priorité basé sur des « *mutex* » ; ce point est fondamental pour un système temps réel dur, car le problème de l'inversion de priorité conduit fréquemment à des *dead locks* ;
- remplacement des *spinlocks* par des *mutex* ;
- mise en place de compteurs à haute précision, permettant d'exprimer les délais et les échéances en microsecondes.

Le principal intérêt de cette solution est de conserver la notion de noyau Linux « unique » et donc la disponibilité des API standards (POSIX) au travers – entre autres - de la Glibc. De même, la procédure de développement des pilotes de périphérique sera strictement identique à celle du noyau standard. Enfin, l'installation du patch est simple puisqu'elle concerne uniquement le noyau (application du patch puis re-compilation du noyau statique et des modules).

Par contre ce patch a quelques inconvénients :

- Il modifie de manière fondamentale le noyau Linux, ce qui peut le rendre incompatible avec d'autres patch comme ceux fournis par des constructeurs de cartes embarquées.
- Les tâches temps réel restent en concurrence avec les tâche Linux standards car elle sont gérées par le même ordonnanceur (même si les politiques sont différentes). Dans la plupart des cas de figure les performances seront donc inférieures aux résultats obtenus avec une architecture à co-noyau.
- Il est surtout développé et testé sur x86. Pour une utilisation sur une autre architecture, il est nécessaire de maintenir ce patch à l'extérieur de l'arbre du noyau « mainline ».

3.5.4. Les solutions à base de co-noyau

Malgré les efforts de la communauté PREEMPT-RT, la solution à co-noyau est certainement le meilleur choix pour la mise en place d'un véritable RTOS à base de noyau Linux. En effet, une bonne partie des applications ne nécessitent pas forcément de respecter scrupuleusement les contraintes d'un RTOS et dans ce cas le choix de PREEMPT-RT offre un meilleur rapport complexité/performances.

L'existence de deux noyaux n'a cependant pas que des avantages. Du fait de la cohabitation des tâches temps réel (gérées par le co-noyau) avec des tâches temps partagé gérées par Linux, il est nécessaire de « *virtualiser* » les interruptions matérielles, afin de savoir par quel noyau elles sont traitées. Le schéma ci-dessous donne l'architecture générale d'une telle solution dans le cas de RTLinux et par extension de RTAI.

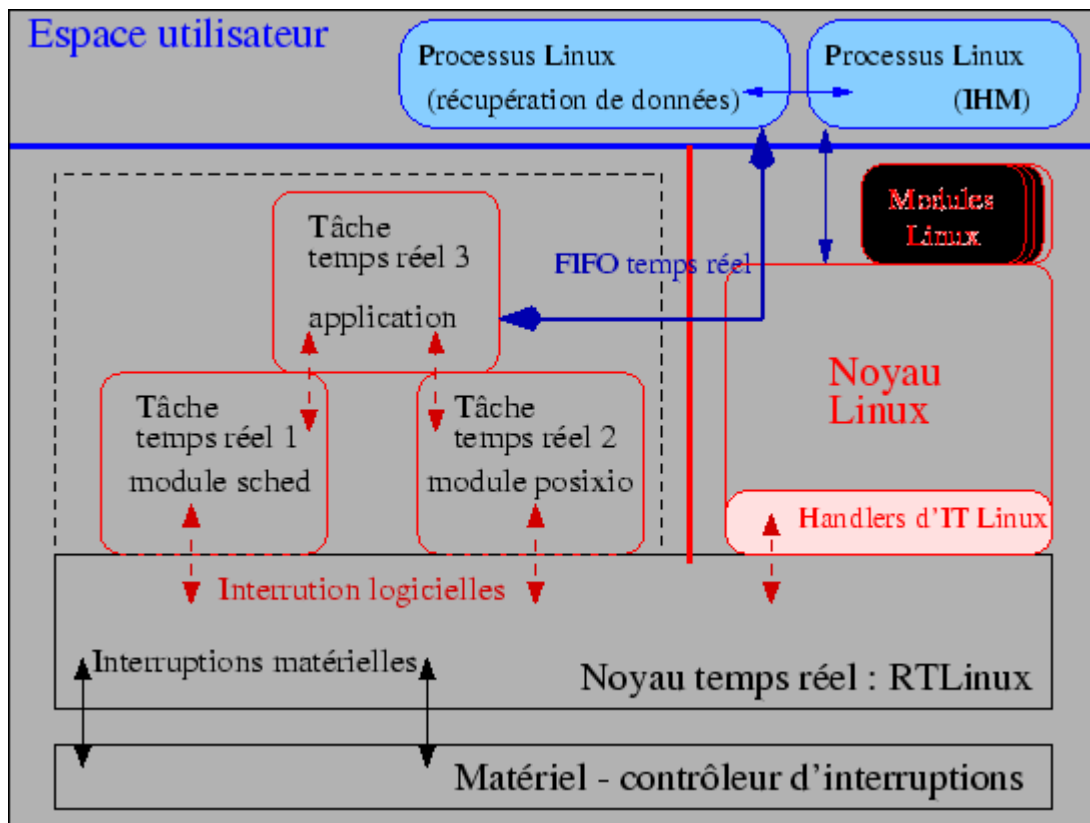


Figure 6. Architecture RTLinux (et RTAI)

Nous remarquons que les tâches temps réel se situent dans l'espace noyau ce qui induit de fortes limitations dans le cas d'applications complexes.

- La programmation en espace noyau est complexe, et peu de fonctions sont disponibles
- Dans le cas (très fréquent) d'une interface homme-machine, il est nécessaire de mettre en place des éléments de communication entre les deux espaces (FIFO ou mémoire partagée) ce qui rend la solution complexe et pénalise les performances.

Le projet RTAI, qui est une émanation ou « fork » du projet RTLinux initiale avait cependant ajouté une interface nommée LXRT permettant de développer des tâches temps réel dans l'espace utilisateur. Cette fonctionnalité est native dans Xenomai pour lequel les tâches temps réel sont systématiquement développées dans l'espace utilisateur, même si il est toujours possible – mais non recommandé - de les développer en espace noyau.

Le projet RTAI

Comme nous l'avons dit, RTAI (<http://www.rtai.org>) est dérivé des versions initiales (libres) de RT-Linux. Ce projet a avant tout un but universitaire (développé au DIAPM, Ecole Polytechnique de Milan, Italie) mais il est parfois utilisé dans des projets industriels.

Le site annonce le support de plusieurs architectures (x86, PowerPC, ARM, m68k) mais ils est surtout utilisé pour des plate-formes x86. Les performances sont similaires à celle des autres systèmes co-noyau comme Xenomai. Les principaux freins à l'utilisation intensive de RTAI nous semblent être les suivants :

- La portabilité est limitée
- La communauté est beaucoup moins active que pour Xenomai
- Le système a un lourd héritage de développement des tâches en espace noyau, même si LXRT est disponible
- L'architecture du code source est moins bien étudiée que celle de Xenomai

Pour toutes ces raisons, nous avons placé les développements et tests sur RTAI dans un niveau de priorité inférieur par rapport aux deux autres systèmes. Une configuration de test pour Intel Atom est cependant intégrée à la chaîne de production.

Le projet Xenomai

La version actuelle de Xenomai (2.5) est liée au projet RTAI. Xenomai a été développé au départ (2001) dans le but d'émuler les API de programmation de RTOS propriétaires (VxWorks, VRTX, pSOS, ...). Le non initial de Xenomai était « Xenodaptor ».

Le but n'était pas de créer un nouveau système temps réel, mais plutôt un outil facilitant la migration d'un RTOS propriétaire vers un environnement Linux. L'annonce initiale du projet par son auteur Philippe Gerum est disponible sur <http://www.mail-archive.com/rtl@fslmlabs.com/msg01156.html>. En janvier 2003, le projet Xenomai fusionne avec le projet RTAI. En avril 2004, la première version RTAI/Fusion est publiée. Elle est le résultat d'une année de travail pour l'amélioration du projet RTAI concernant la disponibilité sur le noyau 2.6 et l'utilisation d'ADEOS afin de s'éloigner de la technologie brevetée de RTLinux.

En 2005, le projet Xenomai reprend son indépendance, et la version 2.0 est publiée sur la base de RTAI/Fusion 0.9.1. À cela s'ajoute le portage sur de nombreuses architectures comme ARM, PowerPC 32 et 64 bits, Blackfin, IA64, AMD 64 et plus récemment Altera NIOS2 en 2009. En plus des sources du projet, le site <http://www.xenomai.org> rassemble un grand nombre de documents, et la documentation en ligne de l'API de développement Xenomai est disponible sur <http://www.xenomai.org/documentation/xenomai-2.5/html/api>.

Architecture de Xenomai

Comme nous l'avons dit dans l'introduction, Xenomai est avant tout un outil de migration. Sa particularité est la présence d'interfaces (appelées également « personnalités » ou *skins*) permettant d'adapter du code source provenant d'un autre RTOS à l'environnement Linux étendu avec Xenomai. L'autre particularité très intéressante est la possibilité de développer des applications temps réel dans l'espace utilisateur, et non plus seulement sous forme de modules en espace noyau. Pour Xenomai, le développement en espace noyau est désormais réservé aux pilotes temps réel RTDM (*Real Time Driver Model*) que nous évoquerons en fin de document. Il est toujours possible de développer des tâches en espace noyau, mais n'oublions pas que cela amène des limitations techniques (peu de fonctions disponibles, mise au point difficile, etc.) ainsi que des problèmes juridiques (utilisation obligatoire de la licence GPL dans l'espace noyau). À ce jour, les « personnalités » disponibles sont les suivantes :

- Native (Xenomai / Nucleus) ;
- POSIX ;
- pSOS+ ;
- RTAI (espace noyau uniquement) ;
- uITRON ;
- VRTX ;
- VxWorks ;
- RTDM (espace noyau uniquement).

L'architecture générale de Xenomai est visible sur le schéma ci-dessous.

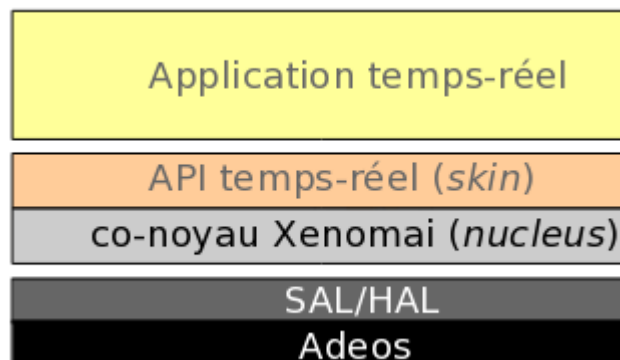


Figure 7. Architecture simplifiée de Xenomai

Les interfaces s'appuient sur le noyau temps réel de Xenomai, nommé *Nucleus*, mais il n'a rien à voir avec le RTOS Nucleus développé par Mentor Graphics (http://www.mentor.com/products/embedded_software/nucleus_rtos). Les couches SAL (pour *System Abstraction Layer*) et HAL (pour *Hardware Abs-*

traction Layer) fournissent l'abstraction de l'architecture hôte (la machine sur laquelle Xenomai est exécuté).

L'adaptation avec le matériel est réalisée par la couche ADEOS, le portage de Xenomai sur une nouvelle architecture correspond donc peu ou prou au portage d'ADEOS sur ce matériel. Le principe de fonctionnement est très proche de celui de RTLinux, comme nous le constatons sur le schéma ci-dessous.

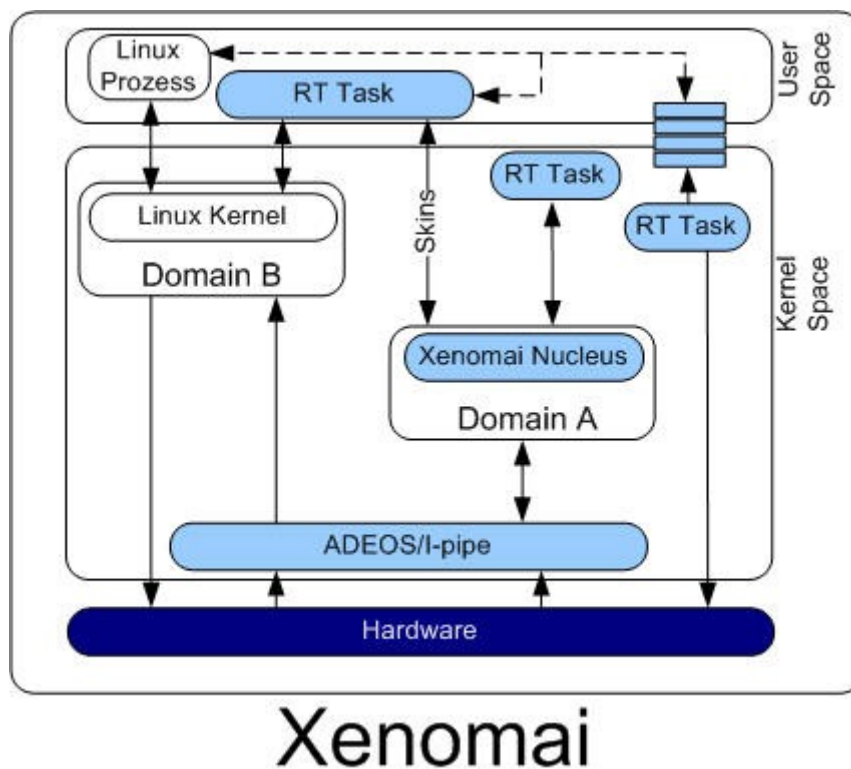


Figure 8. Architecture détaillée de Xenomai

ADEOS

ADEOS (pour *Adaptive Domain Environment for Operating Systems*, voir <http://home.gna.org/adeos>) est une couche de virtualisation des ressources matérielles développée par Philippe Gerum, sur une idée originale de Karim Yaghmour publiée dans un article technique datant de 2001 (voir <http://www.opersys.com/adeos>). Initialement développé pour le projet Xenomai en 2002, ADEOS fut introduit dans l'architecture RTAI dès 2003, afin de se libérer des incertitudes liées au brevet logiciel déposé par FSMLabs et concernant le principe de virtualisation des interruptions.

Le but d'ADEOS est de permettre la cohabitation sur la même machine physique d'un noyau Linux et d'un noyau temps réel. Pour ce faire, ADEOS crée des entités multiples appelées « domaines ».

En première approximation, on peut considérer qu'un domaine correspond à un noyau de système d'exploitation (exemple : Linux et un RTOS). Les différents domaines sont concurrents vis-à-vis des événements externes comme les interruptions. Il réagissent en fonction des niveaux de priorité accordés à chacun d'eux.

La virtualisation des ressources affecte les interruptions ainsi que les exceptions processeur ou les appels système provenant des processus Linux. Les événements circulent sur un *pipeline* virtuel, les notifications étant effectuées en fonction de la priorité. Dans le cas présent, le domaine déterministe (Xenomai) est toujours prioritaire par rapport au domaine racine (Linux). On dit que Linux est une tâche de plus faible priorité (ou *idle task*) pour Xenomai.

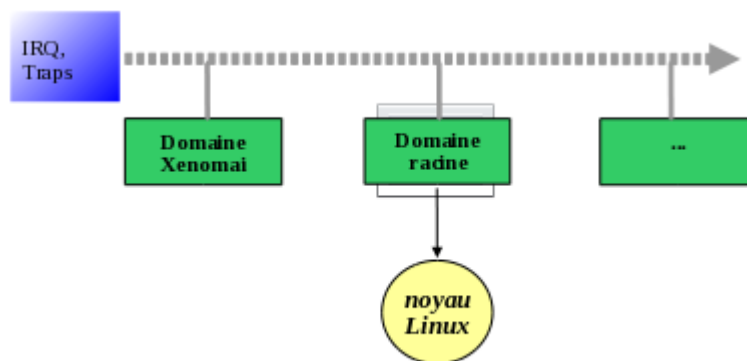


Figure 9. Principe du pipe-line ADEOS

Une description détaillée des fonctionnalités actuelles d'Adeos est disponible à l'adresse <http://www.xenomai.org/documentation/trunk/pdf/Life-with-Adeos-rev-B.pdf>.

Structure et fonctionnement d'une application Xenomai

Nous avons vu qu'une application Xenomai était développée dans l'espace utilisateur de Linux. Vu du système, c'est donc un exécutable utilisant à la fois les bibliothèques standards comme la Glibc et les bibliothèques fournies par Xenomai. L'application est en fait répartie sur les deux domaines.

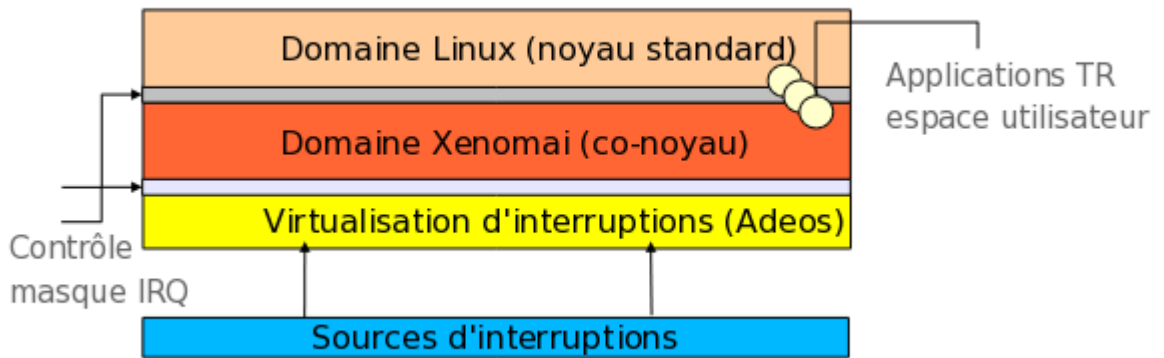


Figure 10. Répartition de l'application sur 2 domaines

Le schéma suivant décrit de manière plus détaillée la structure d'une application. À droite, nous avons le cas d'une application utilisant la personnalité POSIX, et à gauche, une application basée sur la personnalité VxWorks.

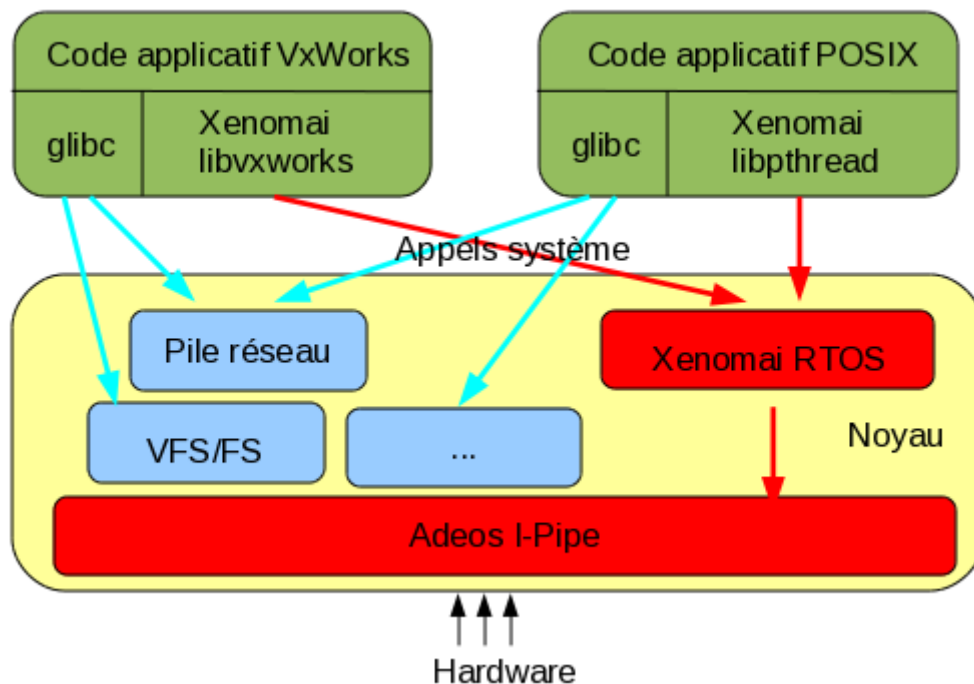


Figure 11. Lien de l'application avec les noyaux (cas de POSIX et VxWorks)

D'après le schéma, on constate que les appels système non pris en compte par les bibliothèques Xenomai le seront par la Glibc. Il y a donc cohabitation au sein d'une même application entre des fonctions temps réel (déterministes, gérées par le noyau Xenomai) et les fonctions Linux issues de la Glibc (non déterministes, gérées par le noyau Linux). Dans le cas de la personnalité POSIX, les tâches temps réel sont créées par la fonction `pthread_create` fournie par Xenomai.

Plus généralement, si un appel système est fourni par les deux API, la version Xenomai sera utilisée en priorité à la compilation de l'application. Si l'on désire utiliser la version Glibc de l'appel système, on devra préfixer le nom par `__real_`, exemple: `__real_pthread_create`.

L'API de développement noyau (RTDM)

Dans le cas général, on aura fréquemment à utiliser une carte fournie avec un pilote Linux (exemple : une carte contrôleur de bus ou une carte d'acquisition). Une « erreur » fréquente est d'utiliser ce pilote avec une application développée sous Xenomai car chaque appel système vers le périphérique (`read`, `write`, ...) correspondra à une « migration de domaine » comme décrit sur la figure ci-dessous.

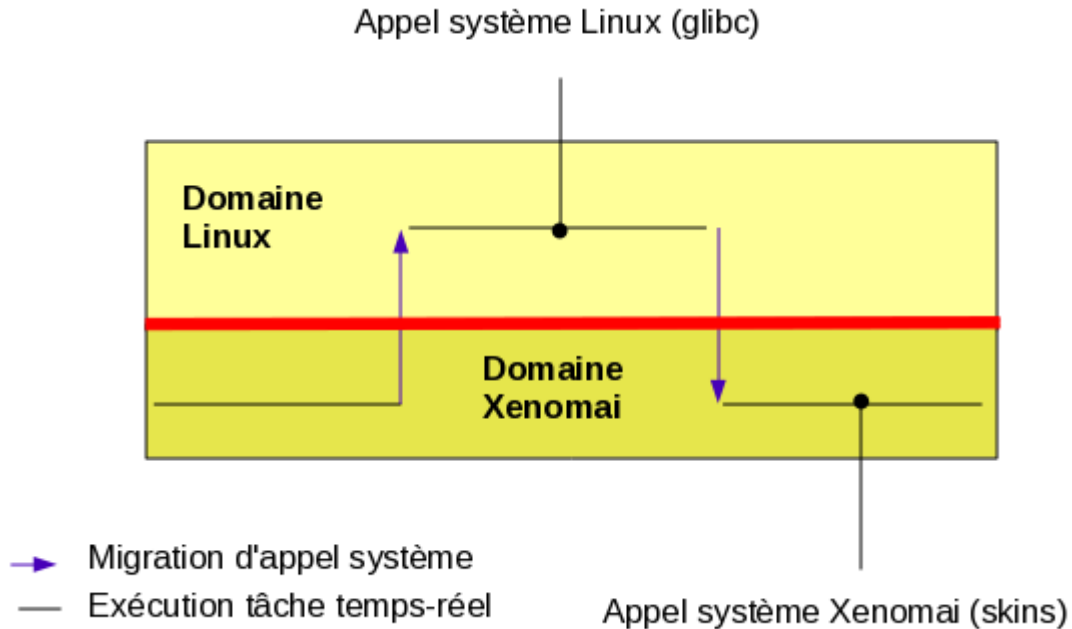


Figure 12. Migration de domaine

Ce problème est potentiellement une source de perturbation au nouveau du respect des contraintes temps réel. Il est donc nécessaire de disposer d'une API permettant de développer des pilotes temps

réel dans l'espace noyau (voir contrainte fonctionnelle CF5). Ce problème n'existe pas dans le cas de PREEMPT-RT (noyau unique) car l'API de programmation des pilotes est celle du noyau Linux. Nous donnons ci-dessous un petit exemple de programme basé sur l'utilisation d'un pilote Linux dans le cas d'une programme POSIX compilable sur Linux et Xenomai.

```
int main (int ac, char **av)
{
    ...
    // Accès direct au registre de données
    fd = open ("/dev/parport0", O_WRONLY);
    ...
    // Creation du thread, en passant le descripteur de fichier à la
    // associée
    pthread_create (&thid_square, &thattr_square, &thread_square, (void*)fd);

    pause();

    return 0;
}
```

Dans la fonction liée au *thread*, on utilise l'appel système `write` pour écrire sur le port de données. La valeur du descripteur de fichier est obtenue dans l'argument abstrait passé à la fonction en dernier argument de `pthread_create`.

```
void *thread_square (void *arg)
{
    int fd = (int)arg;
    ...
    // Boucle infinie
    for (;;) {
        ...
        // Ecriture sur le registre de données
        write (fd, &nibl, 1);
        nibl = ~nibl;
    }
}
```

Xenomai fournit l'API RTDM (Real Time Driver Module) qui est très proche de celle utilisée pour l'écriture de pilotes Linux, car elle fournit les mêmes types de fonctions, ces dernières étant exécutées en majorité dans le domaine Xenomai. Notons également que RTDM est utilisable (en théorie) sous RTAI.

RTDM est une API très proche de celle des pilotes Linux, dont la description dépasse largement les limites de cet ouvrage. Les principes généraux sont les suivants.

Un pilote est avant tout un module, autrement dit un programme exécuté en espace noyau. Dans le cas de Linux, la compilation d'un module produit un fichier `.ko` (pour *Kernel Object*) qui peut être chargé dynamiquement par les commandes `insmod` ou `modprobe` et déchargé par `rmmmod`. Un module a obligatoirement deux points d'entrée, respectivement nommés `module_init` et `module_exit`. Le premier est appelé lors du chargement du module, alors que le deuxième est appelé lors du déchargement.

Un pilote contient en plus divers points d'entrée appelés lors de l'invocation des appels système dans l'application qui utilise le pilote. Si l'application utilise un module `helloworld.ko` contenant les points d'entrées `hello_open`, `hello_read`, `hello_write`, `hello_release`, un appel système `open("/dev/hello", O_RDONLY)` exécutera la fonction `hello_open` du pilote. Le comportement sera le même pour l'appel système `close`, qui exécutera `hello_release`, et ainsi de suite.

Les différents types de pilote RTDM

L'API RTDM est destinée à mettre en place dans le domaine Xenomai les mêmes services noyau que dans le domaine Linux.

- pilote de périphérique matériel (exemple : carte électronique) ; on parle alors d'un *named device*. Le pilote contiendra des entrées comme `open`, `read`, `write`, `ioctl`, etc. ;
- implémentation de protocole (exemple : protocole réseau) ; on parle alors d'un *protocol device*. Le pilote contiendra des entrées de type `socket`, `listen`, `send`, `recv`, etc.

Chaque pilote fait partie d'une classe (`SERIAL`, `CAN`, `EXPERIMENTAL`, ...) et d'une sous-classe (`PP`, `HELLOWORLD`, ...). On peut avoir plusieurs instances d'un même pilote chargées à un même instant.

Les fonctions `_rt` et `_nrt`

Ce point est très spécifique à l'API RTDM. En effet, contrairement au cas d'un pilote Linux, le code d'un pilote RTDM peut être exécuté dans le domaine Xenomai (déterministe ou `_rt`), mais aussi dans le domaine Linux (non déterministe ou `_nrt`). L'API prévoit d'implémenter les fonctions du pilote pour chaque domaine. On pourra donc avoir une fonction `hello_ioctl_rt` et une fonction `hello_ioctl_nrt` suivant le contexte d'exécution. Cependant, on connaît dans la pratique les fonctions typiquement non déterministes (`open` et `close`). Pour les autres fonctions, on devra théoriquement concevoir le système pour utiliser les versions `_rt`, sachant que le but est de développer un pilote déterministe.

Modification de l'application

Lorsque le pilote RTDM est développé, une modification mineure de l'application est nécessaire car les points d'entrée du pilote sont légèrement différents de ceux du pilote Linux.

On modifie la partie principale en utilisant `rt_dev_open` (et non plus `open`) pour ouvrir le périphérique. On utilise ensuite `rt_dev_write` (et non plus `write`) dans le *thread* temps réel pour écrire sur le périphérique

Ouverture du périphérique RTDM

```
// Open RTDM driver
if ((fd = rt_dev_open("pp0", 0)) < 0 ) {
    perror("rt_open");
    exit(EXIT_FAILURE);
}
```

Écriture sur le périphérique

```
rt_dev_write(fd, (void *)&nibl, 1);
```

Conclusions sur Xenomai

L'environnement Xenomai est très riche, et permet de satisfaire toutes les contraintes fonctionnelles décrites au début de ce document. Par contre :

- L'adaptation à une nouvelle architecture est complexe car il faut virtualiser le périphérique à l'aide du micro-noyau ADEOS. Cette adaptation ne dépend pas uniquement du cœur de processeur (exemple: ARM926) mais aussi des périphériques choisis par le fabricant du processeur.
- L'utilisation des bibliothèques standards de Linux est une source potentielle de problèmes de latence du fait de la migration de domaine.

Xenomai sera donc réservé à des applications nécessitant des temps de réponses très faibles, similaires à ceux des RTOS classiques.

4. Conclusions générales

Ce document nous a permis de décrire les contraintes inhérentes à l'utilisation des extensions temps réel de Linux tout en évaluant les avantages et inconvénients des diverses solutions. Les solutions industrialisables et maintenues sont cependant peu nombreuses sachant que l'utilisation des multiples architecture matérielles rend le choix très complexe. Pour choisir une extension parmi celles sélectionnées dans ce document, il faudra donc prendre en considération des critères comme :

- La cible utilisée
- Le type d'application développée
- Les contrainte temps réel nécessaires
- La charge de développement assignée