Temps réel sous LINUX (reloaded)

Pierre Ficheux (pierre.ficheux@openwide.fr)

Janvier 2006

Résumé

Cet article est une mise à jour du dossier *Temps réel sous Linux* paru en juillet 2003 dans le numéro 52 de Linux Magazine. Après une définition des concepts liés au temps réel, nous nous attacherons à décrire les solutions Linux disponibles en insistant particulièrement sur le composant XENOMAI 2. La lecture et la mise en application des exemples décrits nécessite une bonne connaissance « système » de Linux en particulier sur la compilation du noyau. Les codes source des programmes présentés sont disponibles sur http://pficheux.free.fr/articles/lmf/realtime_reloaded.

Définition d'un système temps réel

Temps partagé et temps réel

La gestion du temps est l'un des problèmes majeurs des systèmes d'exploitation. La raison est simple : les système d'exploitation modernes sont tous *multitâche,* or ils utilisent du matériel basé sur des processeurs qui ne le sont pas, ce qui oblige le système à partager le temps du processeur entre les différentes tâches. Cette notion de partage implique une gestion du passage d'une tâche à l'autre qui est effectuée par un ensemble d'algorithmes appelé *ordonnanceur* (*scheduler* en anglais).

Un système d'exploitation classique comme Unix, Linux ou Windows utilise la notion de *temps partagé,* par opposition *au temps réel.* Dans ce type de système, le but de l'ordonnanceur est de donner à l'utilisateur une impression de confort d'utilisation tout en assurant que toutes les tâches demandées sont finalement exécutées. Ce type d'approche entraîne une grande complexité dans la structure même de l'ordonnanceur qui doit tenir compte de notions comme la régulation de la charge du système ou la date depuis laquelle une tâche donnée est en cours d'exécution. De ce fait, on peut noter plusieurs limitations par rapport à la gestion du temps.
Tout d'abord, la notion de priorité entre les tâches est peu prise en compte, car l'ordonnanceur a pour but premier le partage équitable du temps entre les différentes tâches du système (on parle de quantum de temps ou *tick* en anglais). Notez que sur les différentes versions d'UNIX dont Linux, la commande `nice` permet de modifier la priorité de la tâche au lancement.

Ensuite, les différentes tâches doivent accéder à des ressources dites *partagées,* ce qui entraîne des incertitudes temporelles. Si une des tâches effectue une écriture sur le disque dur, celui-ce n'est plus disponible aux autres tâches à un instant donné et le délai de disponibilité du périphérique n'est pas prévisible.

En outre, la gestion des entrées/sorties peut générer des temps morts, car une tâche peut être bloquée en attente d'accès à un élément d'entrée/sortie. La gestion des *interruptions* reçues par une tâche n'est pas optimisée. Le temps de latence - soit le temps écoulé entre la réception de l'interruption et son traitement - n'est pas garanti par le système.

Enfin, l'utilisation du mécanisme de *mémoire virtuelle* peut entraîner des fluctuations dans les temps d'exécution des tâches.

Notion de temps réel

Le cas des systèmes temps réel est différent. Il existe un grande nombre de définition d'un système dit *temps réel* mais une définition simple d'un tel système pourra être la suivante :

Un système temps réel est une association logiciel/matériel où le logiciel permet, entre autre, une gestion adéquate des ressources matérielles en vue de remplir certaines tâches ou fonctions dans des limites temporelles bien précises.

Un autre définition pourrait être :

"Un système est dit temps réel lorsque l'information après acquisition et traitement reste encore pertinente".

Ce qui signifie que dans le cas d'une information arrivant de façon périodique (sous forme d'une interruption périodique du système), les temps d'acquisition et de traitement doivent rester inférieurs à la période de rafraîchissement de cette information.

Il est évident que la structure de ce système dépendra de ces fameuses contraintes. On pourra diviser les systèmes en deux catégories :

1. Les systèmes dits à contraintes *souples* ou *molles* (*soft real time*). Ces systèmes acceptent des variations dans le traitement des données de l'ordre de la demi-seconde (ou 500 ms) ou la seconde. On peut citer l'exemple des systèmes multimédia : si quelques images ne sont pas affichées, cela ne met pas en péril le fonctionnement correct de l'ensemble du système. Ces systèmes se rapprochent fortement des systèmes d'exploitation classiques à temps partagé. Ils garantissent un temps moyen d'exécution pour chaque tâche. On a ici une répartition **égalitaire** du temps CPU aux processus.
2. Les systèmes dits à contraintes *dures* (*hard real time*) pour lesquels une gestion stricte du temps est nécessaire pour conserver l'intégrité du service rendu. On citera comme exemples les contrôles de processus industriels sensibles comme la régulation des centrales nucléaires ou les systèmes embarqués utilisés dans l'aéronautique. Ces systèmes garantissent un temps maximum d'exécution pour chaque tâche. On a ici une répartition **totalitaire** du temps CPU aux tâches.

Les systèmes à contraintes dures doivent répondre à trois critères fondamentaux :

1. Le déterminisme *logique* : les mêmes entrées appliquées au système doivent produire les mêmes effets.
2. Le déterminisme *temporel* : un tâche donnée doit obligatoirement être exécutée dans les délais impartis, on parle d'*échéance*.
3. La *fiabilité* : le système doit être disponible. Cette contrainte est très forte dans le cas d'un système embarqué car les interventions d'un opérateur sont très difficiles ou même impossibles. Cette contrainte est indépendante de la notion de temps réel mais la fiabilité du système sera d'autant plus mise à l'épreuve dans le cas de contraintes dures.

Un système temps réel n'est pas forcément *plus rapide* qu'un système à temps partagé. Il devra par contre satisfaire à des contraintes temporelles strictes, prévues à l'avance et imposées par le processus extérieur à contrôler. Une confusion classique est de mélanger temps réel et rapidité de calcul du système donc puissance du processeur (microprocesseur, micro-contrôleur, DSP). On entend souvent :

« Être temps réel, c'est avoir beaucoup de puissance: des MIPS voire des MFLOPS”

Ce n'est pas toujours vrai. En fait, être temps réel dans l'exemple donné précédemment, c'est être capable d'acquitter l'interruption périodique (moyennant un temps de latence d'acquittement d'interruption imposé par le matériel), traiter l'information et le signaler au niveau utilisateur (réveil d'une tâche ou libération d'un sémaphore) dans un temps inférieur au temps entre deux interruptions périodiques consécutives. On est donc lié à la contrainte durée entre deux interruptions générées par le processus extérieur à contrôler.
Si cette durée est de l'ordre de la seconde (pour le contrôle d'une réaction chimique par exemple), il ne sert à rien d'avoir un système à base de Pentium IV ! Un simple processeur 8 bits du type micro-contrôleur Motorola 68HC11 ou Microchip PIC ou même un processeur 4 bits fera amplement l'affaire, ce qui permettra de minimiser les coûts sur des forts volumes de production.

Si ce temps est maintenant de quelques dizaines de microsecondes (pour le traitement des données issues de l'observation d'une réaction nucléaire par exemple), il convient de choisir un processeur nettement plus performant comme un processeur 32 bits Intel x86, StrongARM ou Motorola ColdFire.

L'exemple donné est malheureusement idyllique (quoique fréquent dans le domaine des télécommunications et réseaux) puisque notre monde interagit plutôt avec un système électronique de façon apériodique.

Il convient donc avant de concevoir ledit système de connaître la durée minimale entre 2 interruptions, ce qui est assez difficile à estimer voire même impossible. C'est pour cela que l'on a tendance à concevoir dans ce cas des systèmes performants (en terme de puissance de calcul CPU et rapidité de traitement d'une interruption) et souvent sur-dimensionnés pour respecter des contraintes temps réel mal cernées à priori. Ceci induit en cas de sur-dimensionnement un sur-coût non négligeable.

En résumé, on peut dire qu'un système temps réel doit être prévisible (*predictible* en anglais*), l*es contraintes temporelles pouvant s'échelonner entre quelques micro-secondes (µs) et quelques secondes.

Une petite expérience

L'expérience décrite sur la figure ci-dessous met en évidence la différence entre un système classique et un système temps réel. Elle est extraite d'un mémoire sur le temps réel réalisé par William Blachier à l'ENSIMAG en 2000.

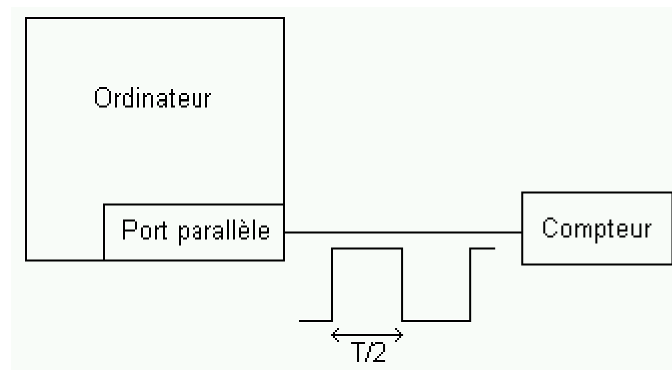

Figure 1. Test comparatif temps réel/temps partagé

Le but de l'expérience est de générer un signal périodique sortant du port parallèle du PC. Le temps qui sépare deux émissions du signal sera mesuré à l'aide d'un compteur. Le but est de

visualiser l'évolution de ce délai en fonction de la charge du système. La fréquence initiale du signal est de 25 Hz (Hertz) ce qui donne une demi-période T/2 de 20 ms.
Sur un système classique, cette demi-période varie de 17 à 23 ms, ce qui donne une variation de fréquence entre 22 Hz et 29 Hz. La figure ci-dessous donne la représentation graphique de la mesure sur un système non chargé puis à pleine charge :

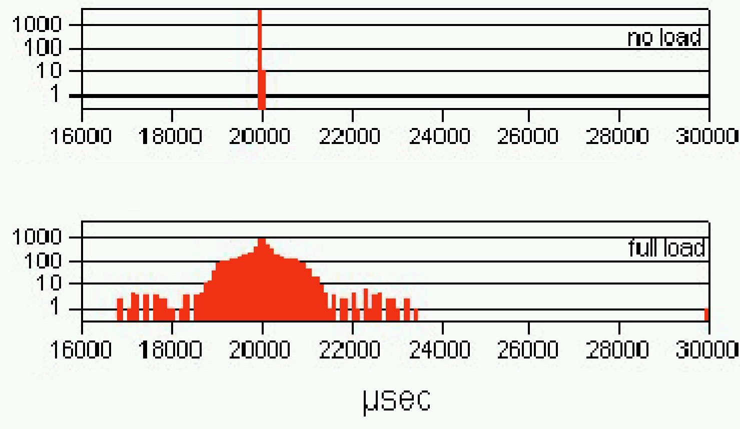

Figure 2. Représentation sur un système classique

Sur un système temps réel, la demi-période varie entre 19,990 ms et 20,015 ms, ce qui donne une variation de fréquence de 24,98 Hz à 25,01 Hz. La variation est donc beaucoup plus faible. La figure donne la représentation graphique de la mesure:

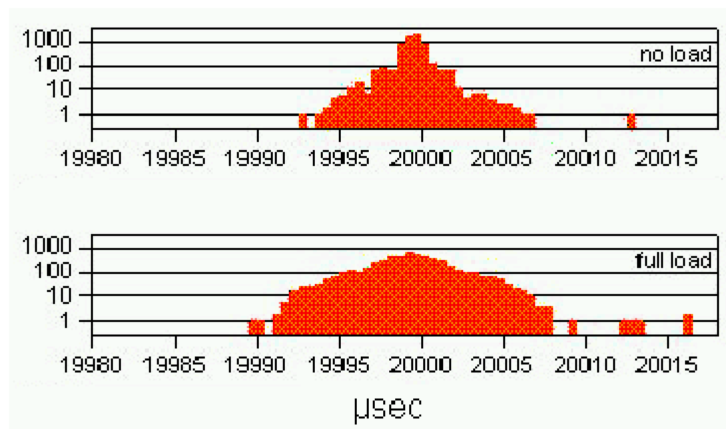

Figure 3. Représentation sur un système temps réel

Lorsque la charge est maximale, le système temps réel assure donc une variation de +/- 0,2 Hz alors que la variation est de +/- 4 Hz dans le cas d'un système classique.

Ce phénomène peut être reproduit à l'aide du programme `square.c` écrit en langage C.

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <asm/io.h>

#define LPT 0x378
```

```
int ioperm();

int main(int argc, char **argv)
{
    setuid(0);

    if (ioperm(LPT, 1, 1) < 0) {
          perror("ioperm()");
          exit(-1);
    }

    while(1) {
          outb(0x01, LPT);
          usleep(50000);

          outb(0x00, LPT);
          usleep(50000);
    }
    return(0);
}
```

Le signal généré sur la broche 2 (bit D0) du port parallèle est théoriquement un signal périodique carré de demi-période T/2 de 50 ms. On observe à l'oscilloscope le signal suivant sur un système non chargé (Intel Celeron 1.2Ghz, 128 Mo de RAM). Le système utilise une distribution *Fedora Core 4* équipée d'un noyau 2.6.14 compilé à partir des sources (ou *vanilla kernel*).

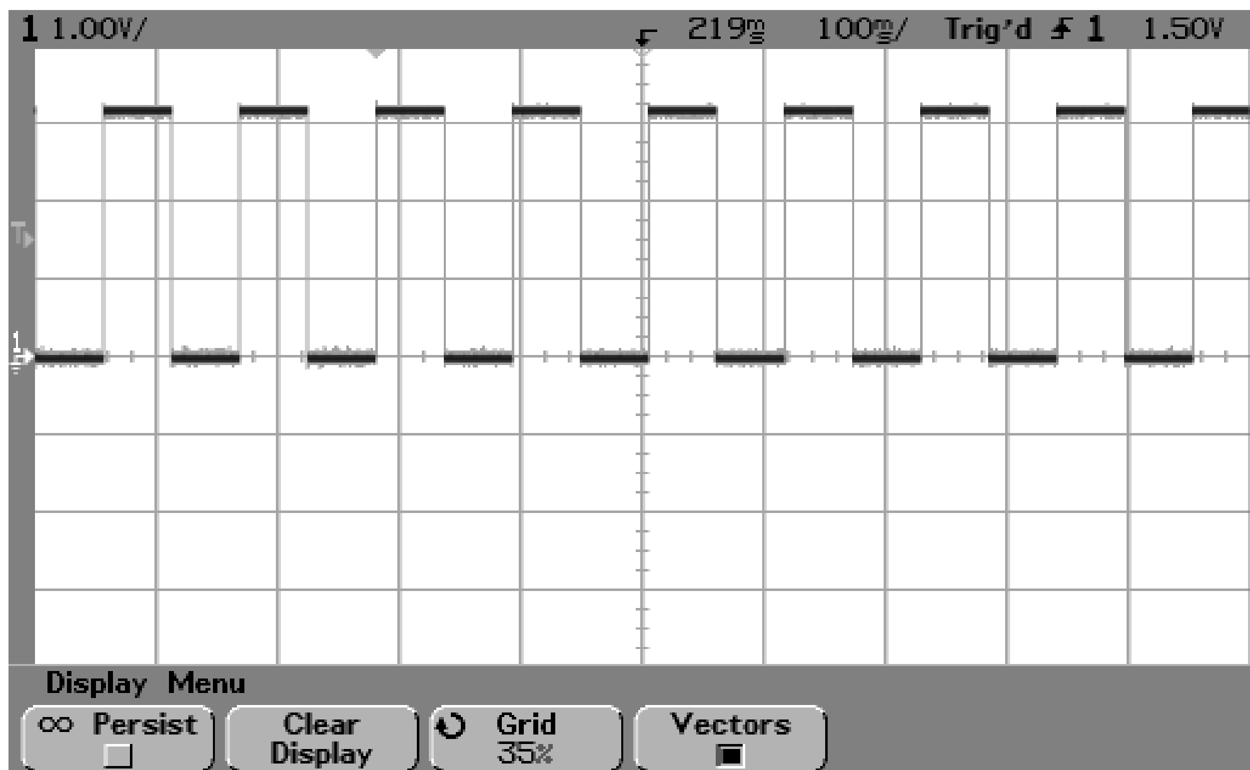

Figure 4. Génération d'un signal carré sous Linux 2.6.14 non chargé

Dès que l'on stresse le système (écriture répétée sur disque de fichiers de 50 Mo en utilisant la commande `dd`), on observe le signal suivant :

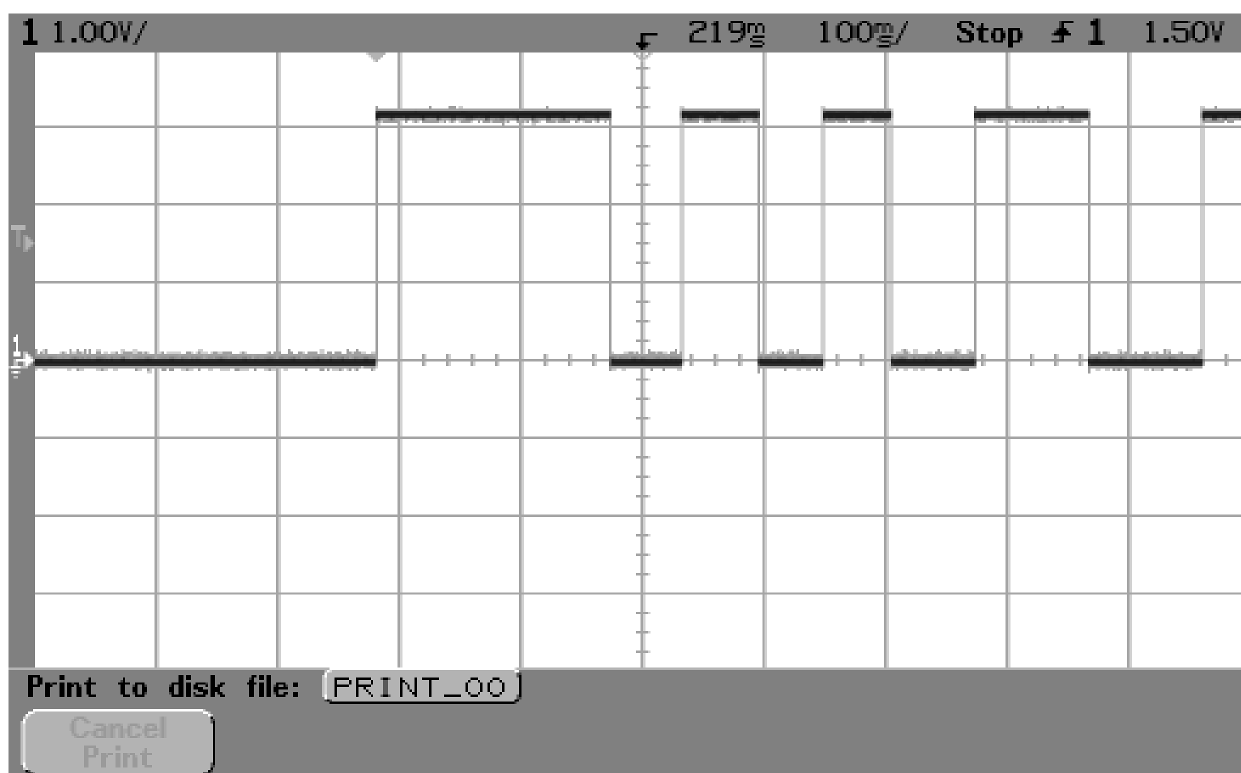

Figure 5. Génération d'un signal carré sous Linux 2.6.14 chargé

La forme du signal varie maintenant au cours du temps, il n'est pas de forme carrée mais rectangulaire. Linux n'est donc plus capable de générer correctement ce signal. Cette expérience met donc en évidence l'importance des systèmes temps réel pour certaines applications critiques.

Préemption et commutation de contexte

Le noyau (*kernel*) est le composant principal d'un système d'exploitation multitâche moderne. Dans un tel système, chaque tâche (ou processus) est décomposée en *threads* (*processus léger* ou *tâche légère*) qui sont des éléments de programmes coopératifs capables d'exécuter chacun une portion de code dans un même espace d'adressage. Chaque thread est caractérisé par un *contexte* local contenant la *priorité* du thread, ses variables locales ou l'état de ses registres. Le passage d'un thread à un autre est appelé changement de contexte (*context switch).* Ce changement de contexte sera plus rapide sur un thread que sur un processus car les threads d'un processus évoluent dans le même espace d'adressage ce qui permet le partage des données entre les threads d'un même processus. Dans certains cas, un processus ne sera composé que d'un seul thread et le changement de contexte s'effectuera sur le processus lui-même.

Dans le cas d'un système temps réel, le noyau est dit *préemptif,* c'est à dire qu'un thread peut être interrompu par l'ordonnanceur en fonction du niveau de priorité et ce afin de permettre l'exécution d'un thread de plus haut niveau de priorité prêt à être exécuté. Ceci permet d'affecter les plus hauts niveaux de priorité à des tâches dites *critiques* par rapport à l'environnement réel contrôlé par le système. La vérification des contextes à commuter est réalisée de manière régulière par l'ordonnanceur en fonction de l'horloge logicielle interne du système, ou *tick timer* système.

Dans le cas d'un noyau non préemptif - comme le noyau Linux - un thread sera interrompu uniquement dans le cas d'un appel au noyau ou d'une une interruption externe. La notion de priorité étant peu utilisée, c'est le noyau qui décide ou non de commuter le thread actif en

fonction d'un algorithme complexe.

Les extensions POSIX

La complexité des systèmes et l'interopérabilité omniprésente nécessitent une standardisation de plus en plus grande tant au niveau des protocoles utilisés que du code-source des applications. Même si elle n'est pas obligatoire, l'utilisation de systèmes conformes à *POSIX* est de plus en plus fréquente.

POSIX est l'acronyme de *Portable Operating System Interface* ou interface portable pour les systèmes d'exploitation. Cette norme a été développée par l'IEEE (*Institute of Electrical and Electronic Engineering*) et standardisée par l'ANSI (*American National Standards Institute*) et l'ISO (*International Standards Organisation*).

Le but de POSIX est d'obtenir la portabilité des logiciels au niveau de leur code source. Le terme de *portabilité* est un anglicisme dérivé de *portability,* terme lui-même réservé au jargon informatique. Un programme qui est destiné à un système d'exploitation qui respecte POSIX doit pouvoir être adapté à moindre frais sous n'importe quel autre système POSIX. En théorie, le portage d'une application d'un système POSIX vers un autre doit se résumer à une compilation des sources du programme.

POSIX a initialement été mis en place pour les systèmes de type UNIX mais d'autres systèmes d'exploitation comme *Windows NT* sont aujourd'hui conformes à POSIX. Le standard POSIX est divisé en plusieurs sous-standards dont les principaux sont les suivants:

- IEEE 1003.1-1990 : POSIX Partie 1 : Interface de programmation (API) système. Définition d'interfaces de programmation standards pour les systèmes de type UNIX, connu également sous l'appellation ISO 9945-1. Ce standard contient la définition de ces fonctions (*bindings*) en langage C.
- IEEE 1003.2-1992 : Interface applicative pour le s*hell* et applications annexes. Définit les fonctionnalités du shell et commandes annexes pour les systèmes de type UNIX.
- IEEE 1003.1b-1993 : Interface de programmation (API) temps réel. Ajout du support de programmation temps réel au standard précédent. On parle également de POSIX.4.
- IEEE 1003.1c-1995 : Interface de programmation (API) pour le multithreading.

Pour le sujet qui nous intéresse, la plupart des systèmes d'exploitation temps réel sont conformes partiellement ou totalement au standard POSIX. C'est le cas en particulier des systèmes temps réel LynxOS (http ://www.lynuxworks.com) et QNX (http ://www.qnx.com). Quant à Linux, sa conformité par rapport à POSIX 1003.1b (temps réel) est partielle dans sa version standard et il nécessite l'application de modification (ou *patch)* sur les sources du noyau.

Tour d'horizon des principaux systèmes temps réel

Le but de cette section est d'effectuer un rapide tour d'horizon des principaux systèmes d'exploitation utilisés dans les environnement embarqués. Ce tour d'horizon n'inclut pas les systèmes à base de Linux qui seront bien entendu décrits en détails plus loin dans l'article.

Il convient avant tout de préciser les différences entre noyau, exécutif et système d'exploitation temps réel :

- Un noyau temps réel est le minimum logiciel pour pouvoir faire du temps réel : ordonnanceur, gestion de tâches, communications inter-tâches, autant dire un système plutôt limité mais performant.
- Un exécutif temps réel possède un noyau temps réel complété de modules/bibliothèques pour

faciliter la conception de l'application temps réel : gestion mémoire, gestion des E/S, gestion de timers, gestion d'accès réseau, gestion de fichiers. Lors de la génération de l'exécutif, on choisit à la carte les bibliothèques en fonction des besoins de l'application temps réel. Pour le développement, on a besoin d'une machine hôte (*host*) et de son environnement de développement croisé (compilateur C croisé, utilitaires, débogueur) ainsi que du système cible (*target*) dans lequel on va télé-charger (par liaison série ou par le réseau) l'application temps réel avec l'exécutif.

- Un système d'exploitation temps réel est le cas particulier où l'on a confusion entre le système hôte et le système cible qui ne font plus qu'un. On a donc ici un environnement de développement natif.

VxWorks et pSOS

VxWorks est aujourd'hui l'exécutif temps réel le plus utilisé dans l'industrie. Il est développé par la société *Wind River* (http ://www.windriver.com) qui a également racheté récemment les droits du noyau temps réel *pSOS*, un peu ancien mais également largement utilisé. VxWorks est fiable, à faible empreinte mémoire, totalement configurable et porté sur un nombre important de processeurs (PowerPC, 68K, CPU32, ColdFire, MCORE, 80x86, Pentium, i960, ARM, StrongARM, MIPS, SH, SPARC, NECV8xx, M32 R/D, RAD6000, ST 20, TriCore). Un point fort de VxWorks a été le support réseau (sockets, commandes, NFS, RPC, etc.) disponible dès 1989 au développeur bien avant tous ses concurrents. VxWorks est également conforme à POSIX 1003.1b.

QNX

Développé par la société canadienne QNX Software (http ://www.qnx.com), QNX est un système temps réel de type UNIX. Il est conforme à POSIX, permet de développer directement sur la plate-forme cible et intègre l'environnement graphique *Photon*, proche de *X Window System*.

µC/OS (micro-C OS) et µC/OS II

µC/OS, développé par le Canadien Jean J. Labrosse, est un exécutif temps réel destiné à des environnements de très petite taille construits autour de micro-contrôleurs. Il est maintenant disponible sur un grand nombre de processeurs et peut intégrer des protocoles standards comme TCP/IP (µC/IP) pour assurer une connectivité IP sur une liaison série par PPP. Il est utilisable gratuitement pour l'enseignement (voir http ://www.ucos-ii.com).

Windows CE

Annoncé avec fracas par Microsoft comme le système d'exploitation embarqué *qui tue,* Windows CE et ses cousins comme Embedded Windows NT n'ont pour l'instant pas détrôné les systèmes embarqués traditionnels. Victime d'une réputation de fiabilité approximative, Windows CE est pour l'instant cantonné à l'équipement de nombreux assistants personnels ou *PDA*.

LynxOS

LynxOS est développé par la société LynuxWorks (http ://www.lynuxworks.com) qui a récemment modifié son nom de part son virage vers Linux avec le développement de Blue Cat. Ce système temps réel est conforme à la norme POSIX.

Nucleus

Nucleus est développé par la société Accelerated Technology Inc. (http ://www.acceleratedtechnology.com). Il est livré avec les sources et il n'y pas de *royalties* à payer pour la redistribution.

eCOS

Acronyme pour *Embeddable Configurable Operating System*, eCOS fut initialement développé par la société Cygnus*,* figure emblématique et précurseur de l'open source professionnel, aujourd'hui rattachée à la société Red Hat Software. Ce système est adapté aux solutions à très faible empreinte mémoire et profondément enfouies. Son environnement de développement est basé sur Linux et la chaîne de compilation GNU avec conformité au standard POSIX.

Les contraintes des systèmes propriétaires

La majorité des systèmes propriétaires décrits à la section précédente souffrent cependant de quelques défauts forts contraignants.

Les systèmes sont souvent réalisés par des sociétés de taille moyenne qui ont du mal à suivre l'évolution technologique : le matériel évolue très vite - la durée de vie d'un processeur de la famille x86 est par exemple de l'ordre de 12 à 24 mois - les standards logiciels font de même et de plus en plus d'équipements nécessitent l'intégration de composants que l'on doit importer du monde des systèmes informatiques classiques ou du multimédia.

De ce fait, les coûts de licence et les droits de redistribution des systèmes (ou *royalties*) sont parfois très élevés car l'éditeur travaille sur un segment de marché très spécialisé, une *niche* dans laquelle les produits commercialisés le sont pour leur fonction finale et non pour la valeur du logiciel lui-même. Contrairement au monde de la bureautique où la pression commerciale peut inciter l'utilisateur à faire évoluer son logiciel fréquemment - et donc à payer un complément de licence - le logiciel embarqué est considéré comme un mal nécessaire souvent destiné à durer plusieurs années, en conservant une compatibilité avec du matériel et des processeurs obsolètes.

Le coût de développement d'applications autour de systèmes propriétaires est souvent plus élevé car les outils de développement sont mal connus de la majorité des développeurs disponibles sur le marché du travail car non étudiés à l'université. Il est donc nécessaire de recruter du personnel très spécialisé donc rare. Les formations autour de ces outils sont également onéreuses car très spécialisées ce qui oblige l'éditeur à pratiquer des coûts élevés pour compenser le manque d'effet de masse.

Tout cela implique un ensemble de spécificités contraignantes pour la gestion globale des outils informatiques de l'entreprise.

Linux comme système temps réel

Forts des arguments concernant l'open source, il est normal d'être tenté par l'utilisation de Linux comme système temps réel. Outre les avantages inhérents à l'open source, la fiabilité légendaire de Linux en fait un candidat idéal. Malheureusement, Linux n'est pas nativement un système temps réel. Le noyau Linux fut en effet conçu dans le but d'en faire un système généraliste donc basé sur la notion de temps partagé et non de temps réel.

La communauté Linux étant très active, plusieurs solutions techniques sont cependant disponibles dans le but d'améliorer le comportement du noyau afin qu'il soit compatible avec les contraintes d'un système temps réel comme décrit au début de l'article. Concrètement, les solutions techniques disponibles sont divisées en deux familles :

1. L'option « préemptive » du noyau Linux 2.6. Les modifications sont dérivées des patch préemptifs utilisées dans les noyaux 2.4 et introduits entre autres par la société Montavista (http://www.mvista.com). Nous parlerons assez peu de cette solution car elle s'applique uniquement dans le cas de temps-réel « mou » et on parle plus d'amélioration de qualité de service (réduction de latence) que de temps réel.

2. Le noyau temps réel auxiliaire. Les promoteurs de cette technologie considèrent que le noyau Linux ne sera jamais véritablement temps réel et ajoute donc à ce noyau un autre noyau disposant d'un véritable ordonnanceur temps réel à priorités fixes. Ce noyau auxiliaire traite directement les tâches temps réel et délègue les autres tâches au noyau Linux, considéré comme la tâche de fond de plus faible priorité. Cette technique permet de mettre en place des systèmes temps réel « durs ». Cette technologie utilisée par RTLinux et les projets européens RTAI et XENOMAI (http://www.xenomai.org). RTLinux est supporté commercialement par FSMLabs (http://www.fsmlabs.com) qui a des relations chaotiques avec la communauté open source à cause du brevet logiciel qui couvre l'utilisation de ce noyau auxiliaire. Dans cette article nous décrirons en détail l'utilisation de la technologie XENOMAI qui permet de créer de tâches temps réel dans l'espace utilisateur et non plus uniquement dans l'espace noyau comme auparavant avec RTLinux ou RTAI.

Le noyau 2.6 préemptif

Le mode dit « préemptif » est sélectionné dans les sources du noyau 2.6 dans le menu *Processor type and features/Preemption model*. Il faut choisir l'option *Preemptible kernel* puis recompiler le noyau.

Test du programme square

Nous avons également effectué un test à l'oscilloscope lors de l'utilisation du programme `square` décrit précédemment. Les résultats sur un système stressé sont assez peu probants et l'on peut noter plusieurs points largement en dehors de l'épure.

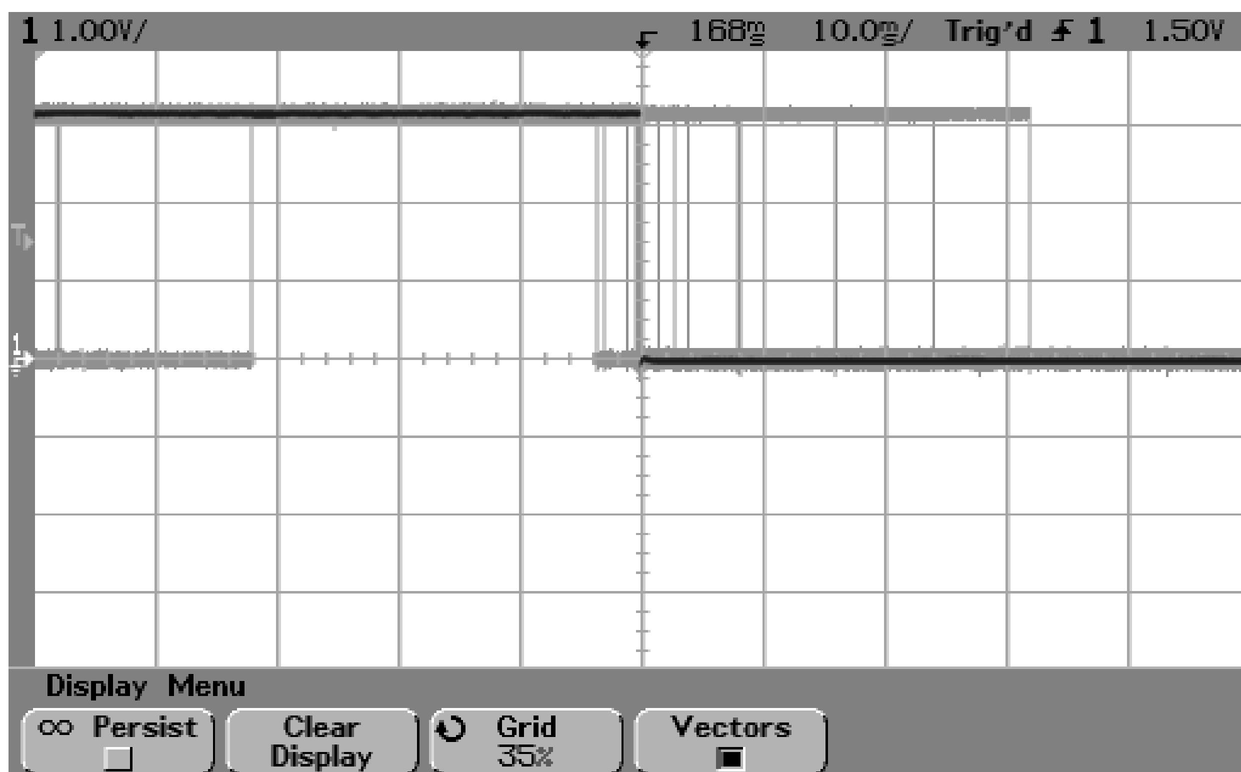

Figure 6. Exécution de square sur un noyau 2.6 préemptif stressé

Indépendamment des résultats, on touche du doigt le problème principal d'une solution à base de noyau Linux modifié : on ne peut garantir un respect fiable des échéances temporelles dans 100 % des cas et les résultats constatés sont basés sur un comportement moyen ce qui provoque quelques points en dehors des limites autorisées. Dans le cas de certains systèmes, ce type de comportement n'est pas acceptable.

XENOMAI 2: temps réel dur sous Linux 2.6

Le projet Xenomai a été fondé en 2001, dans le but de créer un système d'émulation de RTOS traditionnels dans l'environnement GNU/Linux. Entre 2003 et 2005, Xenomai a étroitement collaboré avec le projet RTAI, afin de construire une plate-forme d'aide au développement d'applications temps-réel de qualité industrielle, fortement intégrée au système Linux. De cette collaboration naîtra la branche spéciale de développements dite *RTAI/Fusion*, fondée sur le coeur de technologie du projet Xenomai. Depuis Octobre 2005, Xenomai poursuit seul cet objectif, avec l'aide renouvelée des contributeurs de tous horizons qui ont participé à cet effort depuis son lancement.

La structure schématique de Xenomai basée sur ADEOS est décrit sur la figure suivante.

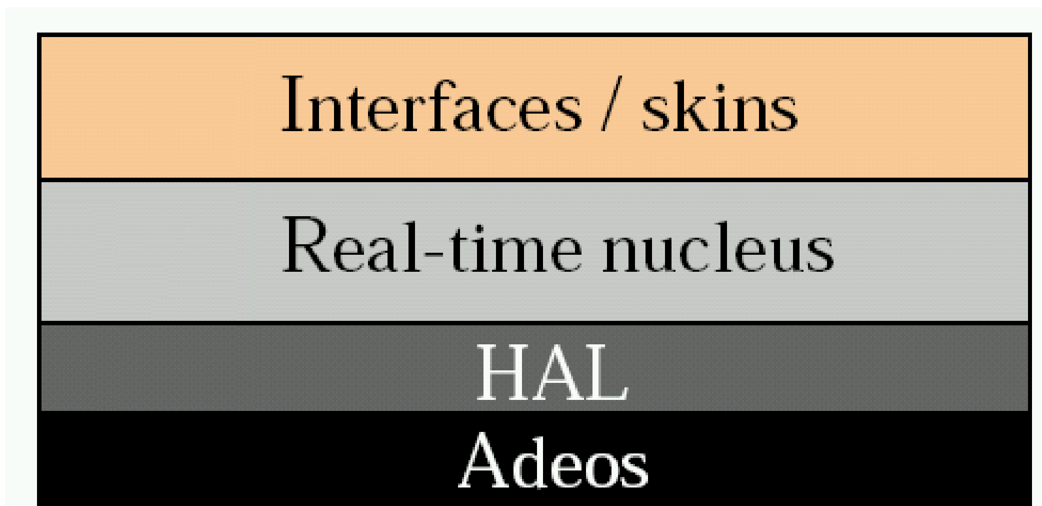

Figure 7: Structure de Xenomai

Utilisation d'ADEOS

ADEOS est une couche de virtualisation des ressources développée par *Philippe Gerum*, sur une idée originale de *Karim Yaghmour* publiée dans un article technique, datant de 2001. 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 obtenu par les FSMLabs, concernant le principe de virtualisation des interruptions, dont dépend tout sous-système temps réel hôte d'un système temps-partagé comme Linux.
Le but d'ADEOS est ainsi de permettre la cohabitation sur la même machine physique d'un noyau Linux et d'un système temps-réel. Pour ce faire, ADEOS crée des entités multiples appelées domaines. En première approximation, on peut considérer qu'un domaine correspond à un OS (exemple: Linux et un RTOS). Les différents domaines sont concurrents vis à vis des évènements externes comme les interruptions. Il réagissent en fonction du niveau de priorité accordé à chacun d'eux. ADEOS constituant l'interface bas niveau de Xenomai, le portage de Xenomai sur une nouvelle architecture correspond en grande partie au portage d'ADEOS sur cette même architecture.

Exécution des tâches temps réel dans l'espace utilisateur

Historiquement, les extensions temps-réel classiques comme RTAI et RTLinux sont basées sur la notion de « double noyau »:

- Un noyau temps réel à priorités fixes chargé de gérer les tâches temps-réel dur. Le noyau et les tâches associées sont des modules du noyau Linux.
- Le noyau Linux considéré comme une tâche de plus faible priorité par le noyau temps-réel et gérant les autres tâches situées dans l'espace utilisateur.

De ce fait, les tâches temps-réel étaient jusqu'à présent programmées dans l'espace du noyau (kernel-space) ce qui pose des problèmes à différents niveaux :

1. Au niveau légal, car en toute rigueur, *seule la GPL est applicable dans l'espace du noyau*. Le code source des tâches temps-réel *devait donc être diffusé sous GPL*.

2. Au niveau technique car l'écriture de modules dans l'espace du noyau est complexe et rend plus difficile la mise au point et la maintenance des modules. De plus, il est nécessaire d'utiliser des objets de communication (FIFO ou mémoire partagée) entre les tâches temps-réel et le reste du programme qui lui réside dans l'espace utilisateur (user-space).

Une première étape franchie par RTAI a permis de disposer d'un environnement de programmation temps-réel plus abordable, ressemblant globalement au contexte d'un module noyau dont l'exécution aurait cependant lieu en espace utilisateur (extension LXRT).
Xenomai a fortement amélioré l'approche introduite dans RTAI-3.x (LXRT) afin de permettre une exécution simple et performante des tâches temps-réel dans l'espace utilisateur, tout en conservant la cohérence des priorités entre l'espace temps-réel Xenomai et la classe temps-réel Linux, avec une possibilité de migration transparente des tâches entre ces espaces. Au-delà de cet aspect, c'est le niveau d'intégration du sous-système temps-réel avec le coeur standard Linux qui a été fortement augmenté. Xenomai permet ainsi d'exécuter un programme temps-réel dans un processus utilisateur Linux multi-threads, tout en gardant l'opportunité de le mettre au point avec un débogueur classique comme GDB.
En résumé, nous pouvons dire que, du point de vue de l'utilisateur, Xenomai rend obsolète la notion de « double programmation » inhérente aux approches antérieures. Cependant, la programmation de tâches temps réel dans l'espace du noyau reste toujours possible.

Un coeur de RTOS générique

L'un des objectifs de Xenomai est de permettre la migration aisée d'applications issues d'un environnement RTOS traditionnel vers un système GNU/Linux. L'architecture de Xenomai repose ainsi sur un nano-kernel proposant une API générique « neutre » utilisant les similarités sémantiques entre les différentes API des RTOS les plus répandus (fonctions de création et gestion de thread, sémaphores, etc.).
Xenomai peut donc émuler des interfaces de programmation propriétaires en factorisant de manière optimale leurs aspects communs fondamentaux, en particulier dans le cas de RTOS spécifiques (ou « maison ») très répandus dans l'industrie.

Les API de Xenomai

Xenomai intègre les APIs de RTOS traditionnels suivantes :

- VxWorks

- pSOS+
- uITron
- VRTX

Ces interfaces sont utilisables soit dans le contexte d'un module noyau Linux, soit sous la forme d'une machine virtuelle fonctionnant dans le contexte d'un processus Linux. L'application et l'interface temps-réel sont alors associées dans un espace utilisateur protégé, et leur fonctionnement est isolé du reste des processus Linux (*sandboxing*).
De plus, une interface POSIX temps-réel complète est également disponible. Enfin, Xenomai définit sa propre interface native exploitant au mieux les capacités du système temps-réel qu'il introduit, dont les principales classes de service sont :

- Gestion de tâches
- Objets de synchronisation
- Messages et communication
- Entrées/sorties
- Allocation mémoire
- Gestion du temps et des alarmes
- Interruptions

Installation de Xenomai 2.01

Les sources de la distribution sont disponibles sur le site http://www.xenomai.org. Après extraction des sources, il est aisé de créer rapidement une version utilisable de Xenomai.

On doit tout d'abord créer un noyau intégrant l'extension ADEOS. Pour ce faire, on doit « patcher » le noyau 2.6 utilisé avec un des fichiers choisi dans le répertoire `arch/i386/patches`. Pour les noyaux récents, nous conseillons d'utiliser les fichiers `adeos-ipipe-*`. On utilise donc la séquence de commandes suivante.

```
# cd /usr/src/kernels/linux-2.6.14
# patch -p1 < adeos-ipipe-2.6.14-i386-1.0-10.patch
```

On recompile ensuite le noyau, on l'installe puis on redémarre le système sur ce nouveau noyau.

Une fois le système Linux correctement redémarré, on doit compiler les modules Xenomai. A partir du répertoire des sources, on utilise les commandes suivantes.

```
# mkdir build
# cd build
# make -f ../makefile
```

Ce qui conduit à l'affichage de la fenêtre de configuration.

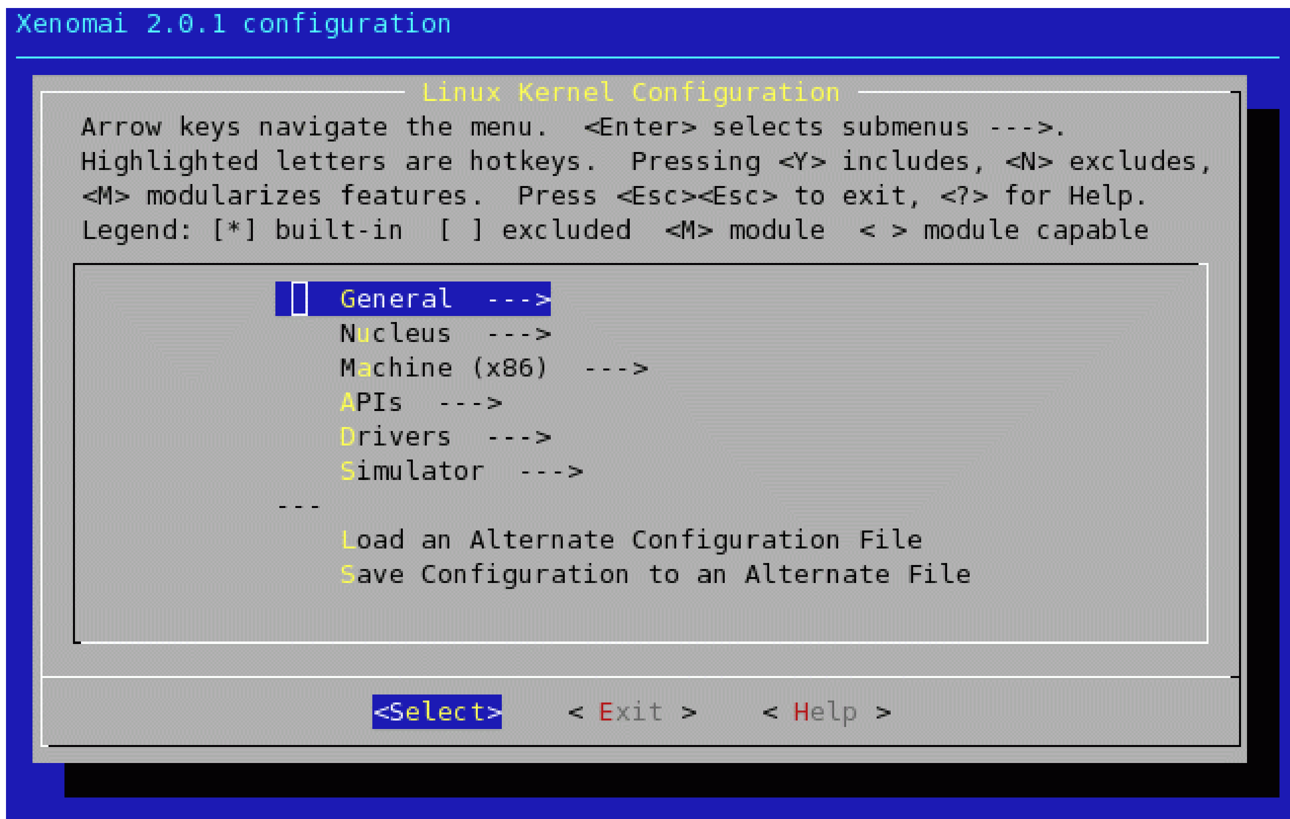

Figure 8: Configuration de Xenomai

Pour un premier test on peut laisser les options par défaut mais il est conseillé aux utilisateurs de chipset Intel d'activer le *SMI workaround* (SMI pour *System Management Interrupt*) dans le menu *Machine (x86)/SMI workaround*. Lorsque l'on sort de l'outil de configuration, la compilation se poursuit automatiquement.

On doit ensuite installer la distribution binaire par la commande `make install`. Les fichiers sont installés sur le répertoire `/usr/realtime`. Un premier test peut être effectué en utilisant les programmes de démonstration disponibles sur le répertoire `testsuite` de la distribution binaire.

On peut effectuer un test de latence en mode utilisateur en faisant.

```
# cd /usr/realtime/testsuite/latency
# ./run
*
*
* Type ^C to stop this application.
*
*
== Sampling period: 100 us
RTT|  00:00:02
RTH|-----lat min|-----lat avg|-----lat max|-overrun|----lat best|---lat worst
RTD|        -866|        -773|        1295|       0|        -866|        1295
RTD|        -900|        -644|        5884|       0|        -900|        5884
RTD|        -887|        -774|         435|       0|        -900|        5884
RTD|        -895|        -655|        5032|       0|        -900|        5884
RTD|        -929|        -771|        1825|       0|        -929|        5884
RTD|        -925|        -772|        1406|       0|        -929|        5884
RTD|        -909|        -652|        5718|       0|        -929|        5884
RTD|        -881|        -768|        1450|       0|        -929|        5884
```

Dans le cas présent, cela indique que la latence maximale est de 5884 ns en appliquant le même principe de stress qu'au début de l'article.

Le même test est disponible dans l'espace noyau.

```
# cd ../klatency
# ./run
*
*
* Type ^C to stop this application.
*
*
RTT|  00:00:01
RTH|----klat min|----klat avg|----klat max| overrun|---klat best|--klat worst
RTD|       -1984|       -1845|         301|       0|       -1984|         301
RTD|       -1976|       -1897|         137|       0|       -1984|         301
RTD|       -1979|       -1833|        3814|       0|       -1984|        3814
RTD|       -1963|       -1899|        -758|       0|       -1984|        3814
RTD|       -1980|       -1835|        3845|       0|       -1984|        3845
RTD|       -1983|       -1897|         418|       0|       -1984|        3845
RTD|       -1975|       -1897|         156|       0|       -1984|        3845
RTD|       -1971|       -1833|        3373|       0|       -1984|        3845
RTD|       -1962|       -1898|        -726|       0|       -1984|        3845
```

Sur ce test rapide, on s'aperçoit que le résultat est meilleur (3845 ns au maximum) puisque la tâche temps réel tourne directement dans l'espace du noyau.

Exemple du programme de test

Le programme de test `square` a été porté vers l'API de Xenomai 2.01 en version espace utilisateur et noyau. Nous utilisons l'API native Xenomai. Dans le cas de l'espace utilisateur, le code source est donné ci-dessous. On notera que la programmation est très proche de celle d'un programme Linux standard en mode utilisateur, mise à part les appels à l'API Xenomai native.

```
#include <sys/mman.h>
#include <sys/time.h>
#include <sys/io.h>
#include <unistd.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <getopt.h>
#include <time.h>
#include <native/task.h>
#include <native/timer.h>
#include <native/sem.h>

RT_TASK square_task;

#define LPT             0x378
#define PERIOD          50000000LL /* 50 ms = 50 x 1000000 ns */
int nibl = 0x01;

long long period_ns = 0;
int loop_prt = 100;             /* print every 100 loops: 5 s */
int test_loops = 0;             /* outer loop count */

/*
 * Corps de la tâche temps réel. C'est une tâche périodique sur 50 ms
```

```
 */
void square (void *cookie)
{
    int err;
    unsigned long old_time=0LL, current_time=0LL;

    if ((err = rt_timer_start(TM_ONESHOT))) {
      fprintf(stderr,"square: cannot start timer, code %d\n",err);
      return;
    }

    if ((err = rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(period_ns)))
{
      fprintf(stderr,"square: failed to set periodic, code %d\n",err);
      return;
    }

    for (;;)
        {
          long overrun = 0;
          test_loops++;

          if ((err = rt_task_wait_period()))
            {
              if (err != -ETIMEDOUT)
                rt_task_delete(NULL); /* Timer stopped. */

              overrun++;
            }

          /* Change le niveau de la sortie */
          outb(nibl, LPT);
          nibl = ~nibl;

          /*
           * On note la date et on affice éventuellement 1 fois toutes
           * les N boucles
           */

          old_time = current_time;
          current_time = (long long)rt_timer_read();

          if ((test_loops % loop_prt) == 0)
            printf ("Loop= %d dt= %ld ns\n", test_loops, current_time -
old_time);
          }
}

void cleanup_upon_sig(int sig __attribute__((unused)))
{
    rt_timer_stop();
    exit(0);
}

int main (int argc, char **argv)
{
    int c, err;

    while ((c = getopt(argc,argv,"p:v:")) != EOF)
        switch (c)
            {
            case 'p':
                period_ns = atoll(optarg) * 1000LL;
                break;
```

```
            case 'v':
                loop_prt = atoi(optarg);
                break;

            default:
                fprintf(stderr, "usage: square  [-p <period_us>] [-v
<loop_print_cnt>]\n");
                exit(2);
            }


    if (period_ns == 0)
      period_ns = PERIOD; /* ns */

    signal(SIGINT, cleanup_upon_sig);
    signal(SIGTERM, cleanup_upon_sig);
    signal(SIGHUP, cleanup_upon_sig);
    signal(SIGALRM, cleanup_upon_sig);

    printf("== Period: %Ld us\n",period_ns / 1000);

    /* Pour accèder au port parallèle en mode utilisateur */
    if (ioperm (LPT, 1, 1) < 0) {
      perror("ioperm");
      exit (1);
    }

    /* Cré ation de la tâche temps réel */
    if ((err = rt_task_create(&square_task,"sampling",0,99,T_FPU)) {
      fprintf(stderr,"square: failed to create square task, code %d\n",err);
      return 0;
    }

    /* Démarrage de la tâche */
    if ((err = rt_task_start(&square_task,&square,NULL)) {
      fprintf(stderr,"square: failed to start square task, code %d\n",err);
      return 0;
    }

    pause();

    return 0;
}
```

Pour compiler le programme, il est nécessaire de mettre en place le fichier `Makefile` suivant. Il faut également ajouter le chemin d'accès `/usr/realtime/bin` à la variable `PATH` de l'utilisateur. De même, il est nécessaire d'ajouter le chemin d'accès aux bibliothèques Xenomai `/usr/realtime/lib` au fichier `/etc/ld.so.conf` puis utiliser la commande `ldconfig` pour valider la modification.

```
# Allow overriding xeno-config on make command line
XENO_CONFIG=xeno-config
prefix := $(shell $(XENO_CONFIG) --prefix)

ifeq ($(prefix),)
$(error Please add <xenomai-install-path>/bin to your PATH variable)
endif

STD_CFLAGS  := $(shell $(XENO_CONFIG) --xeno-cflags)
STD_LDFLAGS := $(shell $(XENO_CONFIG) --xeno-ldflags) -lnative
STD_TARGETS := xenomai_square
```

```
all: $(STD_TARGETS)

$(STD_TARGETS): $(STD_TARGETS:%=%.c)
          $(CC) -o $@ $< $(STD_CFLAGS) $(STD_LDFLAGS)

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

La compilation du programme s'effectue par la commande `make`.

Le fichier `.runinfo` contient les paramètres nécessaires au lancement du programme.

```
xenomai_square:native:!./xenomai_square;popall:control_c
```

Pour exécuter le programme, on utilise la ligne suivante.

```
# xeno-load xenomai_square
```

La trace sur l'oscilloscope est la suivante.

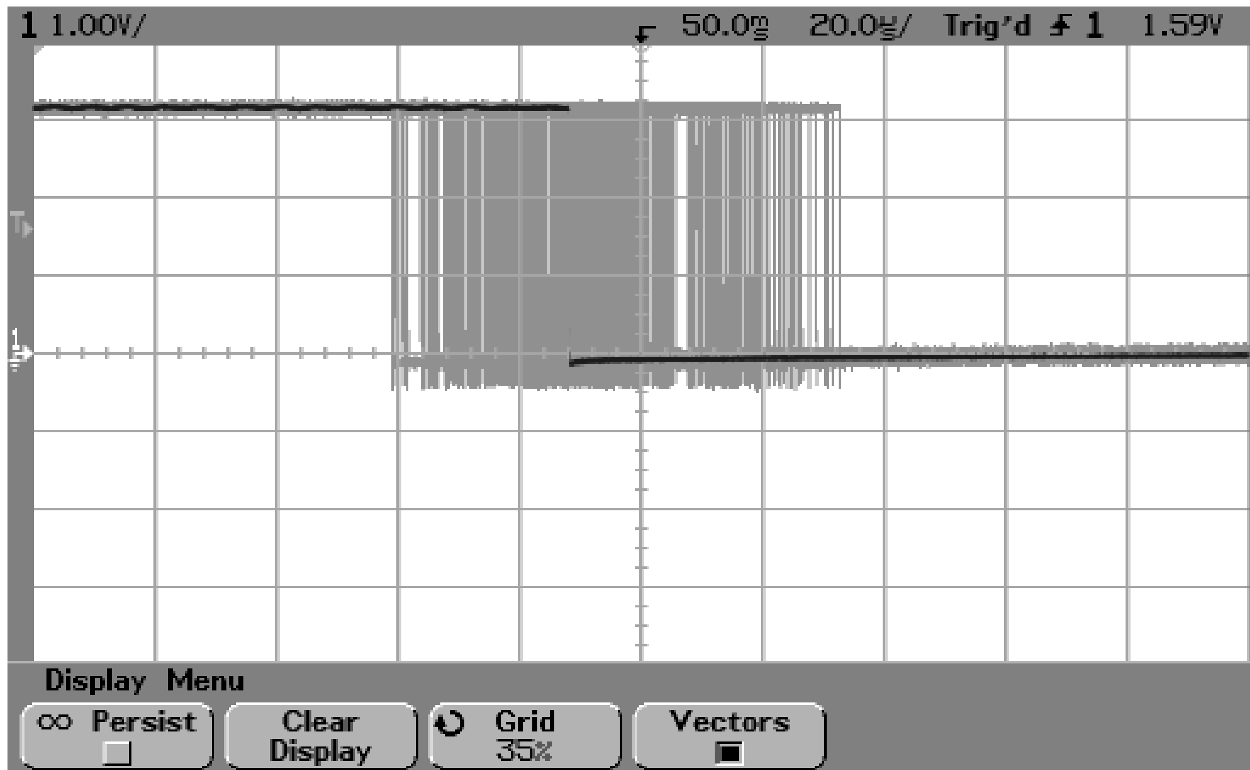

Figure 9: Exécution de xenomai_square en mode utilisateur

Par rapport aux tests effectués avec le noyau 2.6 standard, on notera qu'il n'y a plus de point à l'extérieur d'une limite donnée. Dans le cas présent, on mesure une latence maximale de 40 µs. Toute contrainte externe supérieure (?) à cette valeur pourra donc être scrupuleusement respectée.

Nous avons également testé une version du programme exécutable dans l'espace noyau. Au niveau Linux cela correspond à un module classique. Le code source est donné ci-dessous et l'on notera la forte similitude avec la version développée pour l'espace utilisateur.

```
#include <native/task.h>
#include <native/timer.h>
#include <native/sem.h>
#include <xeno_config.h>
#include <nucleus/types.h>

RT_TASK square_task;

#define LPT             0x378
#define PERIOD          50000000LL /* 50 ms = 50 x 1000000 ns */
int nibl = 0x01;

long long period_ns = PERIOD;
int loop_prt = 100;             /* print every 100 loops: 5 s */
int test_loops = 1;             /* outer loop count */

/* Corps de la tâche temps réel */
void square (void *cookie)
{
    int err;
    unsigned long old_time=0LL, current_time=0LL;

    printk (KERN_INFO "entering square\n");

    if ((err = rt_task_set_periodic(NULL, TM_NOW, rt_timer_ns2ticks(period_ns)))
{
      xnarch_logerr("square: failed to set periodic, code %d\n",err);
      return;
    }

    for (;;)
        {
             long overrun = 0;
             test_loops++;

             if ((err = rt_task_wait_period()))
               {
                 xnarch_logerr("square: rt_task_wait_period err %d\n",err);
                 if (err != -ETIMEDOUT)
                      rt_task_delete(NULL);

                 overrun++;
               }

             outb(nibl, LPT);
             nibl = ~nibl;

             old_time = current_time;
             current_time = (long long)rt_timer_read();

             if ((test_loops % loop_prt) == 0)
               printk (KERN_INFO "Loop= %d dt= %ld ns\n", test_loops,
current_time - old_time);
           }
}

/* Chargement de la tâche (insmod) */
int __square_init (void)
{
    int err;

    printk (KERN_INFO "square_init\n");

    if ((err = rt_timer_start(TM_ONESHOT)))
```

```
    {
        xnarch_logerr("square: cannot start timer, code %d\n",err);
        return 1;
    }

  if ((err = rt_task_create(&square_task,"ksampling",0,99,0)))
    {
        xnarch_logerr("square: failed to create square task, code %d\n",err);
        return 2;
    }


  if ((err = rt_task_start(&square_task,&square,NULL)))
    {
        xnarch_logerr("square: failed to start square task, code %d\n",err);
        return 4;
    }

  return 0;
}

/* Fin de la tâche (rmmod) */
void __square_exit (void)
{
  printk (KERN_INFO "square_exit\n");
  rt_task_delete (&square_task);
  rt_timer_stop ();
}

/* Points d'entrée API modules Linux */
module_init(__square_init);
module_exit(__square_exit);
```

Le fichier `Makefile` à utiliser est très similaire à celui d'un module 2.6 classique. Dans notre cas, cela conduit à la génération du module `xenomai_square_module.ko`.

```
XENO_CONFIG=xeno-config
prefix := $(shell $(XENO_CONFIG) --prefix)

ifeq ($(prefix),)
$(error Please add <xenomai-install-path>/bin to your PATH variable)
endif

MODULE_NAME = xenomai_square_module

$(MODULE_NAME)-objs = xenomai_square-module.o

JUNK      = *~ *.bak DEADJOE

# First pass, kernel Makefile reads module objects
ifneq ($(KERNELRELEASE),)
obj-m     := $(MODULE_NAME).o

EXTRA_CFLAGS += -I$(PWD)/../include
EXTRA_CFLAGS += $(shell $(XENO_CONFIG) --module-cflags)

# Second pass, the actual build.
else
KDIR      := /lib/modules/$(shell uname -r)/build
PWD       := $(shell pwd)

all:
```

```
            $(MAKE) -C $(KDIR) M=$(PWD) modules

clean:
            rm -f *~
            $(MAKE) -C $(KDIR) M=$(PWD) clean

distclean: clean
            $(RM) $(JUNK) $(OBJS)

help:
            $(MAKE) -C $(KDIR) M=$(PWD) help


# Indents the kernel source the way linux/Documentation/CodingStyle.txt
# wants it to be.
indent:
            indent -kr -i8 $($(MODULE_NAME)-objs:.o=.c)

install:
            $(MAKE) -C $(KDIR) M=$(PWD) modules_install

endif
```

L'exécution du programme correspond au chargement des modules Xenomai puis du module généré soit.

```
# insmod  /usr/realtime/modules/xeno_hal.ko
# insmod  /usr/realtime/modules/xeno_nucleus.ko
# insmod  /usr/realtime/modules/xeno_native.ko
# insmod xenomai_square_module.ko
```

On obtient l'affichage suivant sur l'oscilloscope. On note le même type de comportement mis à part que la valeur maximale de la latence est inférieure, soit 30 µs.

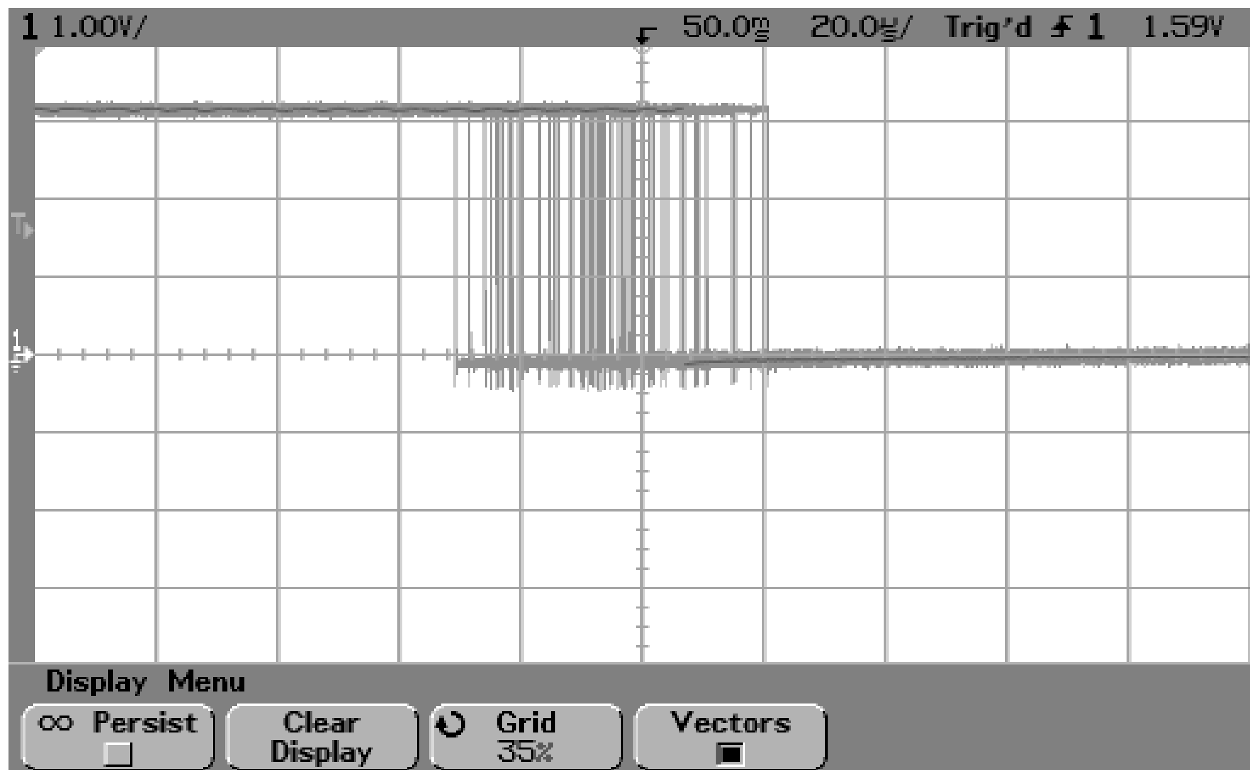

Figure 10: Exécution de xenomai_square_module en mode noyau

Conclusions

Avec l'apparition de solutions comme Xenomai 2, le développement d'applications temps réel dur sous Linux est grandement facilité. L'utilisation de l'espace utilisateur permet à la fois de respecter les contraintes des licences (GPL/LGPL) mais aussi de disposer des outils classiques de la programmation sous Linux. De nombreux efforts sont en cours pour permettre à cette solution d'atteindre une maturité industrielle encore plus grande mais plusieurs références prestigieuses sont déjà disponibles (Thales, Thomson, Xerox, etc.) sur diverses architectures (IA32, IA64, PowerPC, PowerPC 64, ARM).

Remerciements

Merci à Philippe Gerum (rpm@xenomai.org) pour les informations concernant Xenomai et à Patrice Kadionik (kadionik@enseirb.fr) pour la relecture.

Bibliographie

- Dossier « Temps réel sous Linux » de Pierre Ficheux et Patrice Kadionik (juillet 2003) sur http://pficheux.free.fr/articles/lmf/realtime
- Page « *Systèmes embarqués* » de Patrice Kadionik sur http://www.enseirb.fr/~kadionik/embedded/embedded.html
- Site du projet XENOMAI sur http://www.xenomai.org
- Documentation et API XENOMAI sur http://snail.fsffrance.org/www.xenomai.org/documentation/branches/v2.0.x/html/api/index.html
- Site de FSMlabs/RTLinux sur http://www.fsmlabs.com
- Site du projet RTAI sur http://www.rtai.org

- Site du projet ADEOS sur
  http://home.gna.org/adeos/