- PROJET RTEL4I -

Spécifications des moyens de validation des extensions temps réel de Linux
(L2.2-b)

MODIFICATIONS

VERSION	***DATE***	***AUTEUR(S)***	***DESCRIPTION***
1.0	15/10/2010	P. Ficheux	Création

Table des matières

1.Contexte du document....5
2.Gestion du temps dans les systèmes d’exploitation....5
  2.1.Système à temps partagé....5
  2.2.Système temps réel....6
3.Test d'un système temps réel....7
4.Moyens et logiciels mis en œuvre....9
  4.1.Cas d’un noyau Linux modifié (PREEMPT-RT)....10
  4.2.Cas d'un système à double noyau (Xenomai)....12
  4.3.Autres outils de test (monitoring)....13
    4.3.1.Ftrace....14
    4.3.2.Cas de Xenomai (xenoscope)....19

Index des figures

Figure 1.Calcul de latence...7
Figure 2.Affichage de la courbe avec un système Linux non chargé................8
Figure 3.Affichage de la courbe avec un système Linux chargé.......................9
Figure 4. Activation de Ftrace dans la configuration du noyau.......................14
Figure 5. Présentation graphique des résultats...19

1.Contexte du document

Ce document a pour but de décrire les méthodes et les moyens utilisés pour la validation des extensions temps réel de Linux décrites dans le document L2.2a « Spécifications fonctionnelles des extensions temps réel de Linux ». Nous décrirons ci-après les logiciels et matériels utilisés et nous fournirons également plusieurs exemples concrets.

2.Gestion du temps dans les systèmes d'exploitation

La gestion du temps est l'un des problèmes majeurs des systèmes d'exploitation. En effet, les systèmes d'exploitation modernes sont tous multitâche. Or, ils utilisent du matériel basé sur un nombre fini de processeurs, 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).

2.1.Système à temps partagé

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 d'assurer à tous les utilisateurs un temps de réponse moyen « acceptable ». 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 mal prise en compte, car l'ordonnanceur a pour but premier le partage équitable du temps entre les différentes tâches du système et, de ce fait, la priorité de la tâche peut varier au cours du temps. On parle d'ailleurs de *priorité dynamique*, ce qui est le comportement par défaut de l'ordonnanceur du noyau Linux (pour lequel toutes les tâche ont la même priorité égale à 0), et qui est à l'opposé du fonctionnement d'un ordonnanceur temps réel, pour lequel les tâches ont des priorités fixes (1 à 99 dans le cas d'un système POSIX).

Ensuite, les différentes tâches doivent accéder à des ressources partagées, ce qui entraîne des incertitudes temporelles. Si une des tâches effectue une écriture sur le disque, celui-ci 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, et le temps de *latence* – soit le temps écoulé entre la réception de l'interruption et son traitement – n'est pas garanti par le système. On dit alors que le système n'est pas « préemptif ». Notons également que l'utilisation du mécanisme de *mémoire virtuelle* peut entraîner des fluctuations importantes dans les temps d'exécution des tâches.

2.2.Système temps réel

Il existe un grande nombre de définitions d'un système de ce type, mais une version simple pourrait être :

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

Une autre définition pourrait être :

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

Cela signifie que dans le cas d'une information arrivant de façon périodique (sous forme d'une interruption), 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, et l'on pourra diviser les systèmes en deux catégories :

- Les systèmes dits à contraintes *souples* ou *molles* (*soft real time*). Ces systèmes acceptent en général des variations de l'ordre de quelques millisecondes (ms) dans le traitement des données. 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 correct pour chaque tâche. On a ici une répartition égalitaire du temps processeur aux tâches.
- 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 exemple les contrôles de processus industriels sensibles, comme la régulation ou les systèmes embarqués utilisés dans le transport. Ces systèmes garantissent un temps maximum d'exécution pour chaque tâche. On a ici une répartition totalitaire du temps processeur aux tâches. Dans le cas d'un système temps réel, le comportement est dit « préemptif », c'est-à-dire qu'une tâche peut être interrompue à tout moment par l'ordonnanceur en fonction du niveau de priorité (fixe), afin de permettre l'exécution d'une tâche de plus haut niveau de priorité.

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 : une tâche donnée doit obligatoirement être exécutée dans les délais impartis, on parle d' « échéance temporelle » *;*
3. la fiabilité : le système doit être disponible ; 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 doit 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. Être temps réel, 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 de durée entre deux interruptions générées par le processus extérieur à contrôler.

3.Test d'un système temps réel

Nous avons vu au paragraphe précédent les trois critère de bon fonctionnement d'un système temps réel « dur ». Dans la suite du document, nous décrirons en détail le critère de déterminisme temporel car c'est de loin le plus complexe à mettre en œuvre et le plus spécifique à une comportement temps réel.

Le principal point à vérifier est le respect de l'échéance temporelle quelle que soit la charge du système. En effet la caractéristique d'un système « temps partagé » comme Linux est l'apparition rapide de perturbations dans le fonctionnement d'une tâche périodique en cas de charge du système.

- Augmentation de la « latence » (temps de réponse)
- Variations imprévues de la latence (apparition de « jitter »).

A titre d'exemple, si l'on considère une tâche endormie (waiting task) La latence peut être identifiée comme la somme des temps d'attente des différentes phases entre la réception de l'interruption matérielle et le réveil de la tâche.

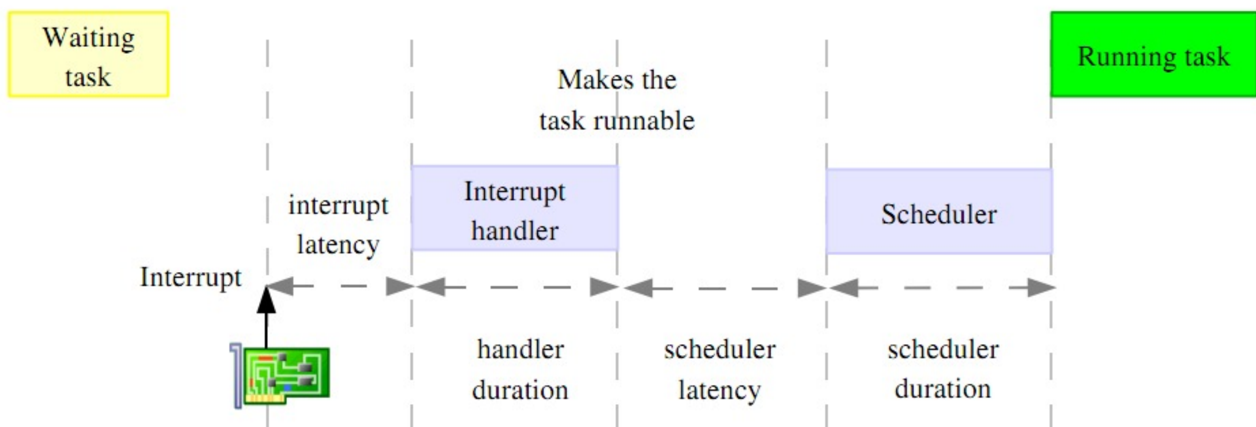

Figure 1.Calcul de latence

REMARQUE: par abus de langage on utilise parfois le terme *latence* au lieu de terme *jitter* alors qu'il s'agit bien d'une variation.

De même nous pouvons reprendre les figures suivantes qui décrivent un tâche périodique sous Linux avec ou sans charge. Ces figures proviennent d'une mesure réelle effectuée avec un oscilloscope synchronisé sur le front montant au centre de la figure. Dans le cas de la deuxième figure, on peut constater l'apparition de « jitter ».

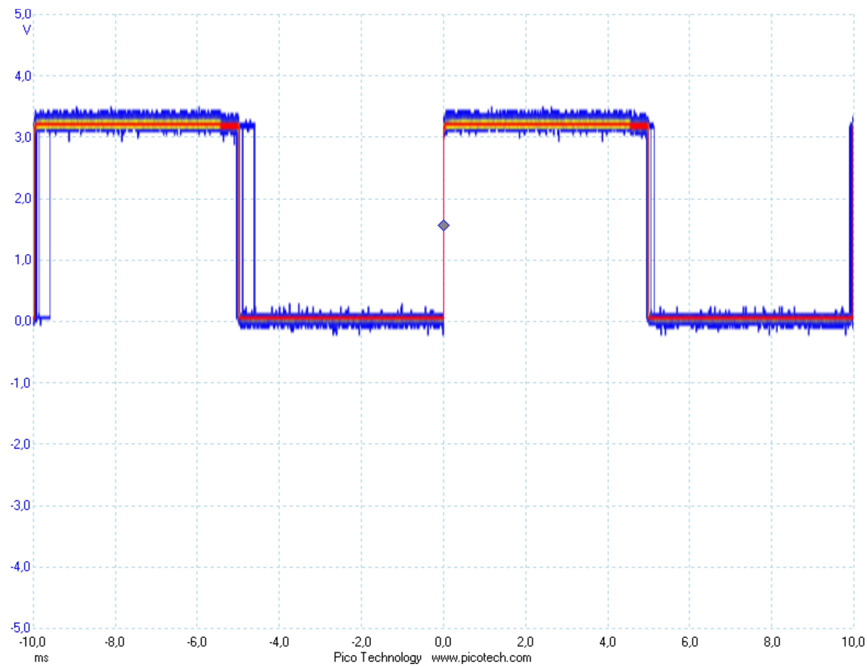

Figure 2.Affichage de la courbe avec un système Linux non chargé

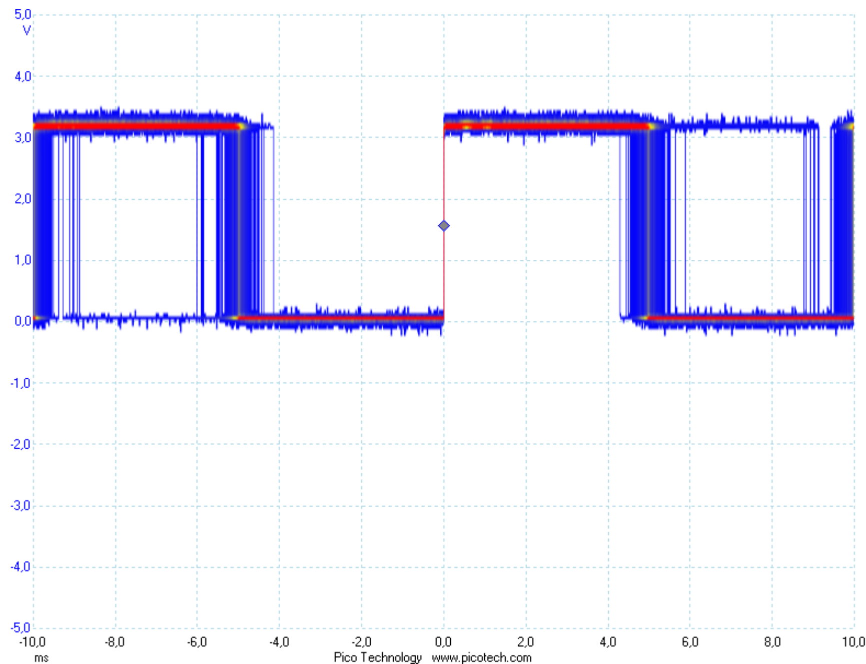

Figure 3.Affichage de la courbe avec un système Linux chargé

4.Moyens et logiciels mis en œuvre

La validation des performances passe par plusieurs types d’outils :

1. Un ou plusieurs outils produisant des tâches périodiques
2. Un ou plusieurs outils générant la charge du système à plusieurs niveaux (entrées/sorties, mémoire, appels systèmes, réseau, etc.)
3. Des outils permettant d’effectuer du monitoring. L’avantage est de pouvoir « enregistrer » les événements afin de réaliser une analyse *a posteriori*. L’intérêt est de ne pas perturber les résultats par une charge induite par des outils de traitement ou de visualisation utilisés en temps réel.

Suivant la technologie utilisée, on devra choisir un outil de sollicitation adaptée. Dans le cas d’une extension PREEMPT-RT, la génération de tâche périodique passe par les bibliothèques standards. Dans le cas d’un système à double noyau (Xenomai) on devra utiliser des outils basés sur les API spécifiques fournies par l’extension. Dans le cas présent, nous allons décrire des procédures de test basées sur des outils ou des bibliothèques standards.

4.1. Cas d'un noyau Linux modifié (PREEMPT-RT)

Le principal outil de génération de tâches périodiques utilisé est `cyclictest`. Il fait partie d'une suite d'outils disponibles sur le wiki de PREEMPT-RT soit https://rt.wiki.kernel.org/index.php/Cyclictest.

Le principe de l'outil est de démarrer un certain nombre de thread en leur affectant une priorité statique donnée (option `-p`). Si la priorité du premier thread est P, celle du second sera P-1 et ainsi de suite. Les nombreuses options disponibles permettent également de spécifier – entre autres - le type d'horloge utilisé.

```
$ ./cyclictest --help
cyclictest V 0.72
Usage:
cyclictest <options>

-a [NUM] --affinity        run thread #N on processor #N, if possible
                           with NUM pin all threads to the processor NUM
-b USEC  --breaktrace=USEC send break trace command when latency > USEC
-B       --preemptirqs     both preempt and irqsoff tracing (used with -b)
-c CLOCK --clock=CLOCK     select clock
                           0 = CLOCK_MONOTONIC (default)
                           1 = CLOCK_REALTIME
-C       --context         context switch tracing (used with -b)
-d DIST  --distance=DIST   distance of thread intervals in us default=500
-D       --duration=t      specify a length for the test run
                           default is in seconds, but 'm', 'h', or 'd' maybe added
                           to modify value to minutes, hours or days
-E       --event           event tracing (used with -b)
-f       --ftrace          function trace (when -b is active)
-h       --histogram=US    dump a latency histogram to stdout after the run
                           (with same priority about many threads)
                           US is the max time to be be tracked in microseconds
-i INTV  --interval=INTV   base interval of thread in us default=1000
-I       --irqsoff         Irqsoff tracing (used with -b)
-l LOOPS --loops=LOOPS     number of loops: default=0(endless)
-m       --mlockall        lock current and future memory allocations
-M       --refresh_on_max  delay updating the screen until a new max latency is hit
-n       --nanosleep       use clock_nanosleep
-N       --nsecs           print results in ns instead of us (default us)
-o RED   --oscope=RED      oscilloscope mode, reduce verbose output by RED
-O TOPT  --traceopt=TOPT   trace option
-p PRIO  --prio=PRIO       priority of highest prio thread
-P       --preemptoff      Preempt off tracing (used with -b)
-q       --quiet           print only a summary on exit
-r       --relative        use relative timer instead of absolute
-s       --system          use sys_nanosleep and sys_setitimer
-t       --threads         one thread per available processor
-t [NUM] --threads=NUM     number of threads:
                           without NUM, threads = max_cpus
                           without -t default = 1
-T TRACE --tracer=TRACER   set tracing function
    configured tracers: unavailable (debugfs not mounted)
-u       --unbuffered      force unbuffered output for live processing
-v       --verbose         output values on stdout for statistics
                           format: n:c:v n=tasknum c=count v=value in us
-w       --wakeup          task wakeup tracing (used with -b)
```

```
-W       --wakeuprt         rt task wakeup tracing (used with -b)
-y POLI  --policy=POLI      policy of realtime thread, POLI may be fifo(default) or rr
                            format: --policy=fifo(default) or --policy=rr
-S       --smp              Standard SMP testing: options -a -t -n and
                            same priority of all threads
-U       --numa             Standard NUMA testing (similar to SMP option)
                            thread data structures allocated from local node
```

La ligne suivante donne un exemple d'utilisation de la commande avec 5 threads.

Ligne de commande utilisée pour cyclictest

```
# cyclictest -p 99 -t5 -n -q
```

Les *threads* démarrés par `cyclictest` ont des priorités hautes, entre 95 et 99. Les valeurs maximales de « jitter » sont sur la droite et exprimées en microsecondes.

Système Linux standard non chargé

```
T: 0 (  504) P:99 I:1000 C:  14259 Min:  7 Act:   11 Avg:   11 Max:  39
T: 1 (  505) P:98 I:1500 C:   9506 Min:  6 Act:   12 Avg:   11 Max:  34
T: 2 (  506) P:97 I:2000 C:   7130 Min:  6 Act:   14 Avg:   10 Max:  32
T: 3 (  507) P:96 I:2500 C:   5704 Min:  6 Act:   11 Avg:   11 Max:  33
T: 4 (  508) P:95 I:3000 C:   4754 Min:  6 Act:   15 Avg:   10 Max:  44
```

On peut augmenter graduellement la charge du système en augmentant le nombre d'appels systèmes. Nous rappelons qu'un appel système est une fonction utilisé en un programme en espace utilisateur mais exécuté dans l'espace du noyau. Certains appels systèmes sont liés à des périphériques matériels (exemples: les fonctions `open`, `close`, `read`, `write`) mais d'autres sont internes au fonctionnement du noyau (exemples: `fork`, `getpid`). L'utilisation intensive d'appels systèmes provoque une augmentation de charge significative pour le système.

Dans le cas présent, nous pouvons utiliser la commande `top` qui par défaut obtient régulièrement auprès du noyau l'état de différents processus du système. La période par défaut est 3 secondes mais si nous forçons la période à 0 secondes, il s'en suit un grand nombre d'appels systèmes qui ont pour effet une augmentation de la charge. Si l'on cumule 3 appels à la commande `top -d0`, on constate une augmentation significative de la valeur du jitter (rapport 10 à 20).

Exécution de 3 commandes « top -d 0 » simultanées

```
T: 0 (16538) P:99 I:1000 C:  12075 Min:  7 Act:   13 Avg:   29 Max: 476
T: 1 (16539) P:98 I:1500 C:   8050 Min:  7 Act:   10 Avg:   23 Max: 464
T: 2 (16540) P:97 I:2000 C:   6038 Min:  6 Act:   11 Avg:   21 Max: 492
T: 3 (16541) P:96 I:2500 C:   4831 Min:  7 Act:   19 Avg:   23 Max: 664
T: 4 (16542) P:95 I:3000 C:   4026 Min:  7 Act:   10 Avg:   24 Max: 741
```

Un autre niveau de la charge du système, l'outil `hackbench` permet d'obtenir une charger beaucoup plus élevée en sollicitant la « scalabilité » du système et particulièrement de l'ordonnanceur. Le principe est de créer un grand nom de processus clients et serveurs qui dialoguent à la manière d'une *chat room* en utilisant des *tubes* UNIX (ou *pipe*).

```
$ hackbench -h
Usage: hackbench [-p|--pipe] [-s|--datasize <bytes>] [-l|--loops <num loops>]
```

```
        [-g|--groups <num groups] [-f|--fds <num fds>]
        [-T|--threads] [-P|--process] [--help]
```

Dans l'exemple suivant nous créons 200 processus ce qui a pour effet d'augmenter le jitter de manière très significative (plus de 16 ms).

Exécution de la commande « `hackbench -p -g 200` »

```
T: 0 (  485) P:99 I:1000 C:  23649 Min:  7 Act:   15 Avg: 3762 Max: 16170
T: 1 (  486) P:98 I:1500 C:  15766 Min:  6 Act:   14 Avg:   24 Max: 16654
T: 2 (  487) P:97 I:2000 C:  11825 Min:  7 Act:   16 Avg:   20 Max: 16133
T: 3 (  488) P:96 I:2500 C:   9460 Min:  7 Act:   16 Avg:   21 Max: 16601
T: 4 (  489) P:95 I:3000 C:   7884 Min:  7 Act:   14 Avg:   19 Max: 15072
```

Dans le cas d'un noyau modifié avec le patch PREEMPT-RT, on obtient les résultats suivants, quasiment indépendants de la charge.

Système Linux/PREEMPT-RT non chargé

```
T: 0 (  504) P:99 I:1000 C:  14259 Min:  7 Act:   11 Avg:   11 Max:   39
T: 1 (  505) P:98 I:1500 C:   9506 Min:  6 Act:   12 Avg:   11 Max:   34
T: 2 (  506) P:97 I:2000 C:   7130 Min:  6 Act:   14 Avg:   10 Max:   32
T: 3 (  507) P:96 I:2500 C:   5704 Min:  6 Act:   11 Avg:   11 Max:   33
T: 4 (  508) P:95 I:3000 C:   4754 Min:  6 Act:   15 Avg:   10 Max:   44
```

Exécution de 3 commandes « top -d 0 » simultanées

```
T: 0 (  514) P:99 I:1000 C:  16052 Min:  6 Act:   12 Avg:   12 Max:   67
T: 1 (  515) P:98 I:1500 C:  10702 Min:  6 Act:   13 Avg:   11 Max:   27
T: 2 (  516) P:97 I:2000 C:   8027 Min:  5 Act:   12 Avg:   10 Max:   33
T: 3 (  517) P:96 I:2500 C:   6422 Min:  7 Act:   13 Avg:   11 Max:   28
T: 4 (  518) P:95 I:3000 C:   5352 Min:  7 Act:   13 Avg:   10 Max:   33
```

Exécution de la commande « `hackbench -p -g 200` »

```
T: 0 ( 8524) P:99 I:1000 C:  13733 Min:  6 Act:   14 Avg:   13 Max:   45
T: 1 ( 8525) P:98 I:1500 C:   9156 Min:  6 Act:   17 Avg:   12 Max:   35
T: 2 ( 8526) P:97 I:2000 C:   6867 Min:  6 Act:   12 Avg:   11 Max:   41
T: 3 ( 8527) P:96 I:2500 C:   5494 Min:  7 Act:   15 Avg:   12 Max:   29
T: 4 ( 8528) P:95 I:3000 C:   4579 Min:  7 Act:   16 Avg:   10 Max:   25
```

4.2.Cas d'un système à double noyau (Xenomai)

Dans le cas d'un système à double noyau, certains outils ne sont pas utilisables directement car on modifier développer avec les bibliothèques fournies avec l'extension (exemple: `libpthread_rt` au lieu de `libpthread` dans le cas de Xenomai). En particulier, il n'est pas possible d'utiliser l'outil `cyclictest` décrit précédemment.

Le projet Xenomai étant assez récent dans sa version actuelle, il existe encore assez peu d'outils de mesures hormis ceux fournis avec la distribution Xenomai. Le plus utilisé est le programme `latency` qui permet de tester une tâche périodique, la période étant ajustable et par défaut à 100 µs. L'intérêt

de l'utilisation de cette commande est son universalité sur toutes les plate-formes (x86, ARM, NIOS2, SH4, ...) supportant Xenomai.

Lors des mesures, nous constatons que les valeurs de latence n'ont rien à voir avec celles obtenues dans le cas d'un noyau standard, même en utilisant la commande `hackbench -p -g 200`.

Test du programme « latency »

```
# /usr/xenomai/bin/latency
== Sampling period: 100 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 100 us period, priority 99)
RTH|--lat min|--lat avg|--lat max|-overrun|---msw|---lat best|--lat worst
RTD|  2.380|       2.445|    3.380|       0|     0|       2.380|    3.380
RTD|  2.395|       2.473|    3.839|       0|     0|       2.380|    3.839
RTD|  2.403|       2.489|    5.110|       0|     0|       2.380|    5.110
RTD|  2.403|       2.475|    3.651|       0|     0|       2.380|    5.110
...
RTD|  1.583|       2.489|    4.659|       0|     0|       1.583|    5.110
RTD|  2.380|       2.494|    4.944|       0|     0|       1.583|    5.110
```

On démarre alors le programme `hackbench -p -g 200`, la latence maximale atteint seulement 11 µs, alors que l'on peut constater visuellement que les temps de réponse du domaine Linux sont dégradés. Un test avec `cyclictest` confirme cela en affichant des latences jusqu'à plus de 18 ms (et non pas µs) pour le domaine Linux, soit plus de 1600 fois plus que pour le domaine Xenomai.

Latences après le démarrage de « hackbench -p -g 200 »

```
RTD|  2.418|       5.234|   10.674|       0|     0|       1.583|   10.674
RTD|  3.027|       5.828|   10.704|       0|     0|       1.583|   10.704
RTD|  1.989|       5.594|   10.817|       0|     0|       1.583|   10.817
...
RTD|  2.922|       3.732|   11.080|       0|     0|       1.583|   11.080
RTD|  2.801|       3.763|    8.132|       0|     0|       1.583|   11.080
RTD|  2.914|       3.765|    9.283|       0|     0|       1.583|   11.080
```

Résultat de « cyclictest » pour le domaine Linux

```
T: 0 ( 8517) P:99 I:1000 C:  17613 Min:  10 Act:  26 Avg:   77 Max: 18119
T: 1 ( 8518) P:98 I:1500 C:  11742 Min:   9 Act:  17 Avg:   35 Max: 18509
T: 2 ( 8519) P:97 I:2000 C:   8807 Min:  10 Act:  16 Avg:   29 Max: 17941
T: 3 ( 8520) P:96 I:2500 C:   7046 Min:  10 Act:  18 Avg:   31 Max: 16943
T: 4 ( 8521) P:95 I:3000 C:   5872 Min:  10 Act:  17 Avg:   28 Max: 16869
```

4.3.Autres outils de test (monitoring)

D'autres outils comme Ftrace, OProfile ou LTTng peuvent être utiles pour la validation du système. En effet la validation *temporelle* d'un logiciel est fondamentale dans le cas d'un système temps réel et il est donc nécessaire de pouvoir étudier la synchronisation entre les tâches. A titre d'exemple, on peut imaginer une erreur de gestion d'un sémaphore qui entraine la lecture des données erronées lors

du traitement concurrents par plusieurs tâches. La difficulté de mise au point vient le plus souvent de la résolution temporelle utilisée. En effet, il est quasiment impossible d'utiliser des méthodes classiques (exemple: débogueur GDB) pour résoudre ce type de problème sur un programme donc la granularité est de quelques dizaines de µs. De même, la classique méthode des traces (`printf` ou `syslog` n'est également pas applicable).

4.3.1.Ftrace

L’outil Ftrace (littéralement *Function Tracer*) a été introduit dans le noyau 2.6.27 dans le but d’unifier les outils de trace du noyau. C’est donc un composant standard utilisable pour toutes les versions du noyau et pour toutes les architectures officiellement supportées. Cet outil est compatibles avec la majorité des architectures Linux dans le cas d'un noyau standard ou bien PREEMPT-RT. Il permet de tracer la quasi-totalité des événements du noyau car il est intégré aux sources du noyau « mainline ». Par contre, l'intégration à Xenomai n'est pas totalement validée.

Ftrace est capable de produire des information concernant les latences, le traitement des interruptions, les changements de contexte (ou *context switch*). Il permet également de générer des graphes d’appel de fonctions. Notons que l’on peut bien entendu étendre les fonctionnalités de Ftrace en y ajoutant des greffons.

Activation dans le noyau

L’activation de Ftrace s’effectue dans le menu **Kernel hacking** du la configuration du noyau accessible par la commande `make menuconfig` (ou bien `xconfig`, `gconfig`). Pour cela on doit activer l’option **Tracers** puis sélectionner les différentes traceurs dans le sous-menu associé.

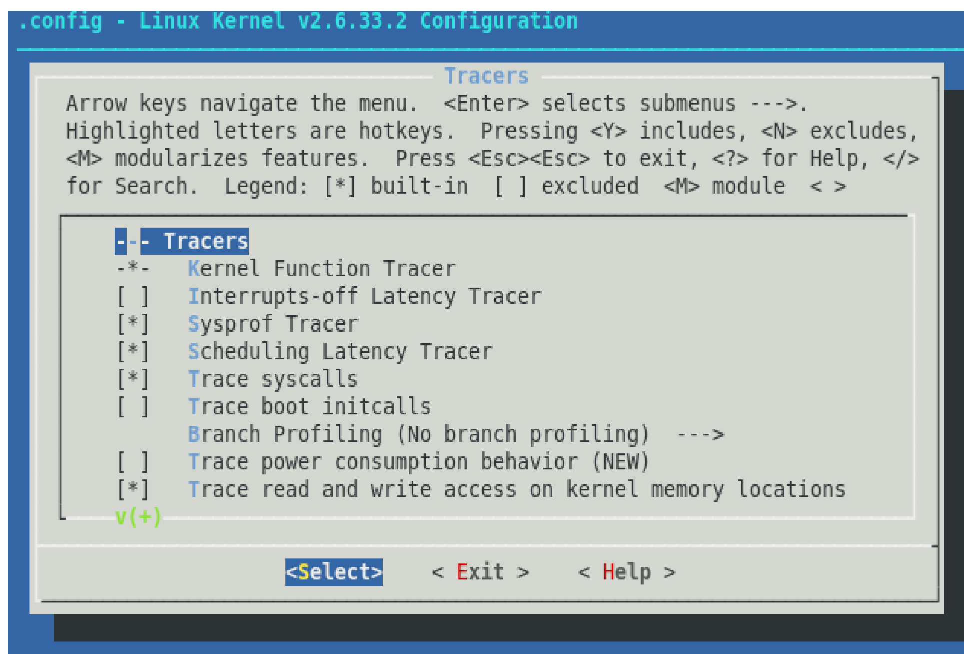

Figure 4. Activation de Ftrace dans la configuration du noyau

Une documentation d'utilisation très complète est disponible dans le fichier `Documentation/trace/ftrace.txt`.

Les bases de Ftrace

L'utilisation de Ftrace passe par les étapes suivantes :

1. Sélectionner un *traceur* qui indique quelle fonctionnalités on doit tracer dans le noyau
2. Définir un *filtre* pour limiter les traces produites (exemple: le nom de la fonction du pilote à tracer)
3. Activer la trace en effectuant la commande :

```
# echo 1 > /proc/sys/kernel/ftrace_enabled
```

4. Après enregistrement, désactiver la trace afin de limiter la taille des données par :

```
# echo 0 > /proc/sys/kernel/ftrace_enabled
```

5. Exploiter les résultats

L'utilisation de Ftrace nécessite d'activer la fonctionnalité *DebugFS* qui est un système de fichiers dédié aux outils de mise au point. Tout comme `/proc` ou `/sys`, DebugFS permet de manipuler des données du noyau comme des fichiers standards. Dans le cas de Ftrace, il permet de configurer les traces mais également d'obtenir les résultats dans un fichier virtuel. Notons que d'autres outils comme LTTng utilisent également DebugFS.

Ce système de fichier est en général monté sur `/debug` ou bien `/sys/kernel/debug`. Si ce n'est pas fait automatiquement, on peut effectuer le montage par la commande :

```
# mount -t debugfs nodev /sys/kernel/debug
```

On dispose alors du répertoire `/sys/kernel/debug/tracing` contenant entre-autres le fichier `trace` qui contient les résultats de l'instrumentation. Les principaux fichiers de ce répertoire sont les suivants :

- `available_tracers` contient les différents traceurs disponibles, soit `nop`, `function`, `function_graph`, ...
- `current_tracer` contient le traceur courant, donc une valeur de la liste précédente.
- `trace` contient les résultats lisibles de l'instrumentation.
- `tracing_on` permet d'activer/désactiver les traces en écrivant les valeur 1 ou 0.
- `available_events` contient les évènements traçables comme `sched_wakeup`. Ces évènements correspondent à des points de trace (ou `Tracepoints`) *statiques* ajoutés au noyau Linux, voir `Documentation/trace/events.txt`.
- `set_ftrace_pid` permet de tracer un processus donné par son PID.

Un premier exemple simple

Dans ce premier exemple, nous allons tracer l'utilisation d'un pilote de test correspondant au module `mydriver1.ko`. Les fonctions `open`, `release`, `read`, `write` du pilote se limitent à une simple fonction `printk`. La configuration de l'instrumentation s'effectue simplement par les commandes décrites ci-après. Dans le cas présent nous limitons l'instrumentation aux fonctions du pilote testé (`mydriver1`) en utilisant un filtre.

```
# modprobe mydriver1.ko
# cd /sys/kernel/debug/tracing
# echo function > current_tracer
# echo 'mydriver1_*' > set_trace_filter
# cat set_strace_filter
mydriver1_release
mydriver1_open
mydriver1_write
mydriver1_read
# echo 1 > tracing_on
```

On effectue ensuite un accès au pilote au travers du fichier spécial `/dev/mydriver1`, en effectuant les appels systèmes `open`, `write`, `close` par une simple commande `echo` puis on visualise les traces. L'affichage est assez clair, précisons simplement que la colonne de droite correspond à la fonction appelante (exemple: `chrdev_open`).

```
# echo salut > /dev/mydriver1

# cat trace
# tracer: function
#
#        TASK-PID    CPU#    TIMESTAMP  FUNCTION
#           | |       |          |         |
         bash-12960 [001] 3711754.199386: mydriver1_open <-chrdev_open
         bash-12960 [001] 3711754.199413: mydriver1_write <-vfs_write
         bash-12960 [001] 3711754.199419: mydriver1_release <-__fput
```

Analyse de performances avec PREEMPT-RT

Le but de l'exemple est de comparer les performances du noyau Linux standard par rapport au même noyau modifié par application du *patch* PREEMPT-RT. La méthode de test est basée sur un programme en espace utilisateur qui est réveillé avec une période de 5 ms. Lors du réveil, il utilise un pilote simple pour écrire un octet sur le port parallèle d'un PC/x86 via l'appel système `write`.

```
 for (i = 0; i < 100; i++)
    {
      /* 5ms */
      usleep (5000);

     /* Write 1 octet to the device */
      if (write(fd, &one, 1) < 0)
      {
        perror("Can't write to device.");
        exit(-6);
```

```
        }
    }
```

Dans le cadre de notre test, nous avons comparé 3 configurations :

1. Configuration Linux standard avec ordonnancement à priorité dynamique (SCHED_OTHER), ce qui constitue le cas général de l'exécution d'un programme sous Linux.
2. Configuration Linux standard avec ordonnancement à priorité fixe (SCHED_FIFO)
3. Configuration Linux PREEMPT-RT avec ordonnancement à priorité fixe (SCHED_FIFO)

Dans les deux derniers cas, nous avons également désactivé la pagination en utilisant la fonction mlockall. L'extrait de code est donné ci-après :

```
 /* Declare ourself as a real time task */
  param.sched_priority = MY_PRIORITY;
  if(sched_setscheduler(0, SCHED_FIFO, &param) == -1)
  {
     perror("sched_setscheduler failed");
     exit(-1);
  }

  /* Lock memory */
  if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1)
  {
     perror("mlockall failed");
     exit(-2);
  }
```

Bien entendu, nous avons effectué pour chaque configuration une mesure pour un système au repos (charge proche de 0) puis une mesure pour un système fortement chargé (proche de 100%) grâce à l'utilitaire hackbench également disponible depuis le site PREEMPT-RT. La configuration de Ftrace utilise le traceur nop et trace les événements sched_wakeup.

Dans le cas du programme standard sans le patch PREEMPT-RT, on obtient les traces suivantes :

```
kswapd0-35      0d.h.. 771644849us!: sched_wakeup: comm=a.out pid=26218 prio=120 success=1
hackbenc-7508   0d.h2. 771653245us!: sched_wakeup: comm=a.out pid=26218 prio=120 success=1
hackbenc-7508   0d.h1. 771658319us!: sched_wakeup: comm=a.out pid=26218 prio=120 success=1
hackbenc-26297 0d.h1. 771671639us+: sched_wakeup: comm=a.out pid=26218 prio=120 success=1
hackbenc-26299 0dNh.. 771681179us!: sched_wakeup: comm=a.out pid=26218 prio=120 success=1
hackbenc-26335 0dNh.. 771690824us!: sched_wakeup: comm=a.out pid=26218 prio=120 success=1
...
```

Dans le cas de la configuration temps réel avec le patch PREEMPT-RT, on obtient les résultats suivants :

```
<...>-19074    0d.h3. 2478610554us!: sched_wakeup: comm=a.out pid=22410 prio=50 success=1
<...>-19073    0d.h3. 2478615571us!: sched_wakeup: comm=a.out pid=22410 prio=50 success=1
<...>-19067    0d.h3. 2478620584us!: sched_wakeup: comm=a.out pid=22410 prio=50 success=1
<...>-19967    0d.h3. 2478625596us+: sched_wakeup: comm=a.out pid=22410 prio=50 success=1
<...>-14718    0d.h3. 2478630611us+: sched_wakeup: comm=a.out pid=22410 prio=50 success=1
<...>-15344    0d.h3. 2478635625us+: sched_wakeup: comm=a.out pid=22410 prio=50 success=1
<...>-14691    0d.h3. 2478640634us+: sched_wakeup: comm=a.out pid=22410 prio=50 success=1
```

. . .

On en déduit les deux tableaux suivants, avec ou sans charge pour les 3 types de configuration. On remarque immédiatement la supériorité du noyau avec patch PREEMPT-RT pour lequel la variation de période est très faible, et ce indépendamment de la charge du système.

délai entre deux réveil de la tâche (100 écritures)	**2.6.33.7-nort test standard idle**	**2.6.33.7-nort test rt idle**	**2.6.33.7-rt test rt idle**
Délai moyen (µs)	5075	5107	5004
Délai médian (µs)	5052	5002	5003
Délai maximum (µs)	7109	11584	5029
Délai minimum (µs)	5030	5001	5002
Ecart type	207	763	3,29

Tableau 1. Résultats sans charge

délai entre deux réveil de la tâche (100 écritures)	**2.6.33.7-nort test standard chargé**	**2.6.33.7-nort test rt chargé**	**2.6.33.7-rt test rt chargé**
Délai moyen (µs)	10602.89	5698.6	5014.35
Délai médian (µs)	9733	5015	5014
Délai maximum (µs)	19497	17132	5021
Délai minimum (µs)	5074	5001	5004
Ecart type	3251	2119	2,63

Tableau 2. Résultats avec charge 100%

Une représentation graphique produite sur la base des résultats de Ftrace est encore plus significative. Nous présentons ici les 3 colonnes de résultats (noyau standard / `SCHED_OTHER`, noyau standard / `SCHED_FIFO`, noyau PREEMPT_RT / `SCHED_FIFO`). On constate que seule la troisième colonne donne des résultats inférieurs à 100 µs de latence.

	5051	5957		5002	5002		5004	5014
	5052	10215		5001	5005		5003	5013
	5051	9940		5002	5017		5003	5013
	5052	9810		5003	5015		5003	5015
	5051	10021		5001	5007		5004	5015
	5052	18174		5002	5017		5003	5015
	5052	9424		11584	7409		5004	5016
	5051	18528		5002	5018		5004	5018
	5052	9740		5002	5018		5003	5019
	5051	9869		5002	5007		5003	5012
	5052	9991		5001	5007		5003	5013
	5051	9717		5002	5017		5004	5014
	5052	9800		5002	5482		5002	5015
	5052	9694		5001	5014		5004	5014
	5051	9840		5002	5015		5003	5016
	5052	9782		5001	5007		5003	5020
	5053	9729		5003	5013		5005	5019
	5052	19497		5002	17132		5004	5014
	5052	12273		5002	5036		5004	5015
moyenne	5075.28	10602.89		5107.34	5698.6		5003.92	5014.35
medianne	5052	9733		5002	5015		5003	5014
max	7109	19497		11584	17132		5029	5021
min	5030	5074		5001	5001		5002	5004
écart-type	207.65	3251.39		763.11	2119.53		3.29	2.63
légende	5000µs -> 5100µs	5100µs -> 6000µs		6000µs -> 10000µs	> 10000µs			

Figure 5. Présentation graphique des résultats

4.3.2.Cas de Xenomai (xenoscope)

Dans le cas de Xenomai, les événements temps réel ne sont pas gérés par le noyau Linux mais par le noyau temps réel de Xenomai nommé *nucleus*. Dans le cadre du projet RTEL4I, il s'agit donc de mettre en place un outil permettant d'enregistrement les évènements temps réel au niveau de nucleus puis de traiter ces évènements soit en « live » soit après enregistrement.

Cet outil doit permettre d'enregistrer une session complète afin d'analyser les problèmes éventuels de synchronisation (principe du « flight recorder »).

Les principales fonctionnalités demandées sont les suivantes :

- Modèle client/serveur avec enregistrement des événements nucleus au niveau de la cible et transmission au poste de développement x86 pour affichage en temps réel ou enregistrement dans un fichier.
- L'outil de lecture doit permet un affichage graphique
- La durée d'enregistrement doit pouvoir être spécifiée par un compteur que l'on peut arrêter/relance
- On doit pouvoir poser des « points d'arrêt » en fonctions de conditions (exemple: valeur de « trigger » atteint)