Programmation Temps Réel

OS Temps réel

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d'Ingénierie et de Gestion du Canton de Vaud

Octobre 2017

Ordonnanceur historique

- Jusqu'à la version 2.6 de Linux:
- Calcul le poids de toutes les tâches
- Elit la tâche de plus fort poids
- Complexité: $O(n)$
- Calcul du poids:
  - Tâche déjà activée sur un autre CPU: - 1000
  - Tâche temps réel: + 1000
  - Tâche temps partagé: poids initialisé au nombre de ticks non consommés
  - Tâche précédemment sur même CPU: + 15
  - Si la configuration de la tâche est déjà en MMU: + 1
  - + priorité définie par l'utilisateur (1-40)

Noyau 2.6

- 2 tableaux: Active et Expired
- Chaque liste contient 40 listes de tâches
- Elit la tâche de plus haute priorité dans le tableau Active
- Complexité: $O(1)$
- Lorsqu'une tâche a épuisé son quota elle passe dans la table Expired
- Lorsque la table Active est vide on intervertit les 2 tables

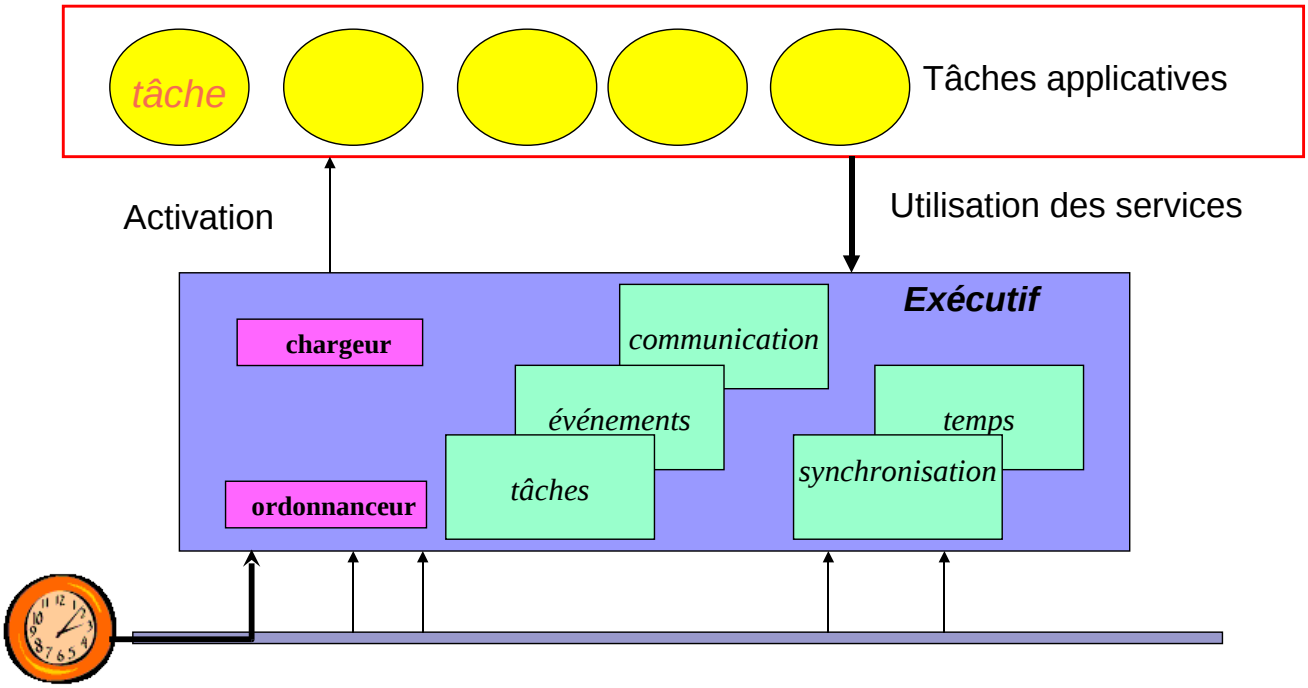
Noyau 2.6.23: Ordonnanceur CSF

- CSF: Completely Faire Scheduler
- Organisation des tâches dans un *red-black tree*
- Complexité: $O(n)$
- Exploite une valeur de *virtual runtime* des tâches (temps CPU consommé)
- Augmenter la priorité d'une tâche de 1 multiplie le temps disponible par 1.1

OS temps réel

- Un système d'exploitation temps-réel est une catégorie de sys. d'exploitation permettant la programmation et l'exécution de tâches temps-réel.
  - Il ne diffère pas fondamentalement d'un OS standard "non" temps-réel.
  - Il est généralement plus simple, plus petit (car destiné à être embarqué) et offre moins de confort au niveau du développement.
- Un OS temps-réel permet de gérer le temps de manière "stricte" (temps-réel strict) ou non (temps-réel souple).
  - Ordonnancement spécifique des tâches temps-réel
  - Structures de données "temps-réel"
  - Accès direct au matériel

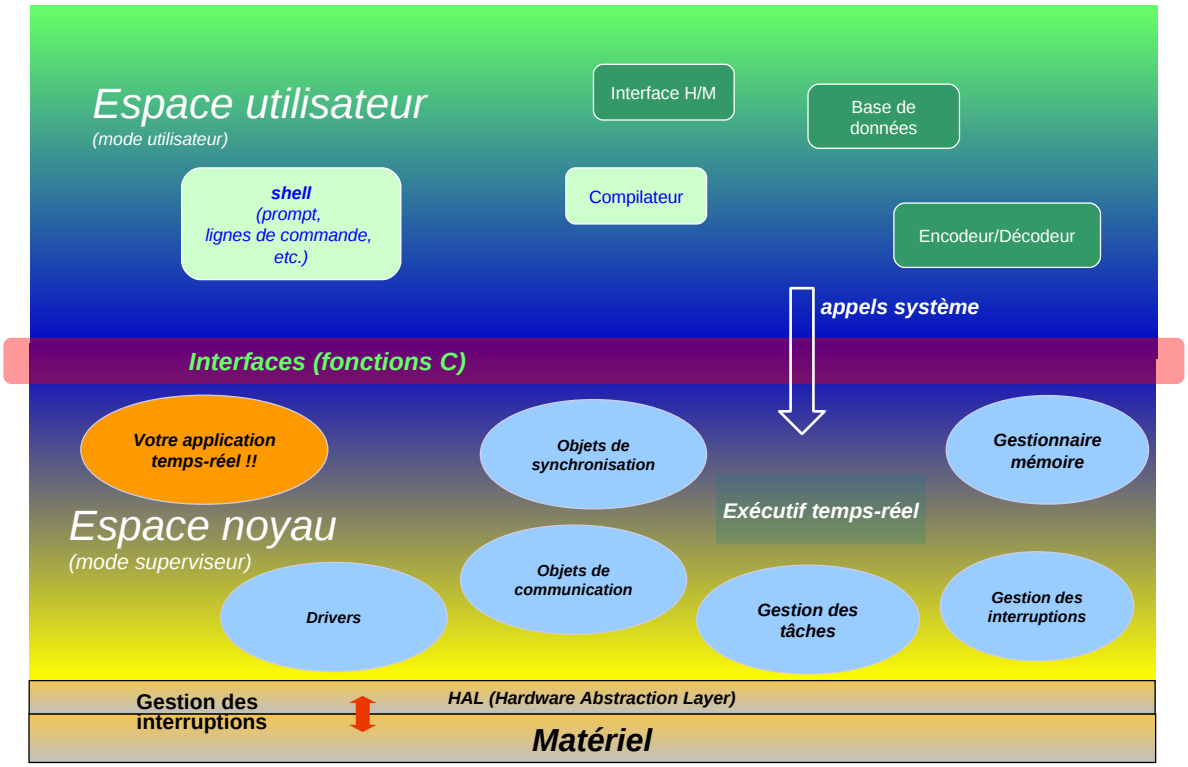
Exécutif temps réel - Structure



Espace utilisateur / noyau (1)

- Les processeurs offrent différents modes d'exécution
- Mode utilisateur
  - Permet l'exécution de plusieurs processus en assurant leur protection.
  - Pas d'accès aux instructions privilégiées.
  - Avec l'aide de la MMU (Memory Management Unit), le gestionnaire mémoire peut différencier les espaces d'adressage.
  - C'est le mode dans lequel les applications vont s'exécuter.
- Mode superviseur (il peut y en avoir plusieurs)
  - Accès à toutes les instructions, y compris aux instructions privilégiées (instructions I/O, accès mémoire physique, etc.).
  - Plus de protection garantie.
  - Accès à toute la mémoire physique

Espace utilisateur / noyau (2)



Espace utilisateur / noyau (3)

- L'espace utilisateur correspond à l'environnement d'exécution lorsque le processeur est en mode utilisateur.
  - Les applications d'un ordinateur conventionnel (PC/Laptop/etc.) s'exécutent dans l'espace utilisateur.
- L'espace noyau correspond à l'environnement d'exécution lorsque le processeur est en mode noyau.
  - Les services de l'OS s'exécutent dans l'espace noyau.
  - La plupart des drivers.
  - Les applications temps-réels !
- La barrière entre l'espace utilisateur et l'espace noyau constitue l'interface entre l'application et le système d'exploitation.
  - L'application fait appel aux services du système d'exploitation à l'aide d'appels système.

Exemple d'appels système

- Un appel système est généralement une fonction C.
  - Un appel système déclenche une interruption logicielle; il s'agit d'une interruption synchrone, c-à-d que le processeur exécute une instruction particulières (INT, SWI, etc.).
  - Il peut y avoir plusieurs interruptions logicielles (overflow, division par zéro, etc.) - Une des interruptions logicielles est réservée aux appels systèmes.
- Exemple d'appels systèmes:
  - open(), write(), read(), close(), etc.
- Les appels système ne sont pas visibles dans l'espace noyau!!
  - Par conséquent, une application temps-réel ne se programme pas de la même manière qu'une application dans l'espace utilisateur

OS temps réel

- OS temps-réel commercial

OS	site web
VxWorks	http://www.windriver.com
pSOS+	http://www.windriver.com
ThreadX	http://www.ghs.com/products/rtos/threadx.html
LynxOS	http://www.lynuxworks.com/rtos
Qnx	http://www.qnx.com
RTLinux	http://www.fsmlabs.com/
Jaluna	http://www.jaluna.com

OS temps réel

- OS temps-réel en logiciel libre (Open Source, GNU, etc.)

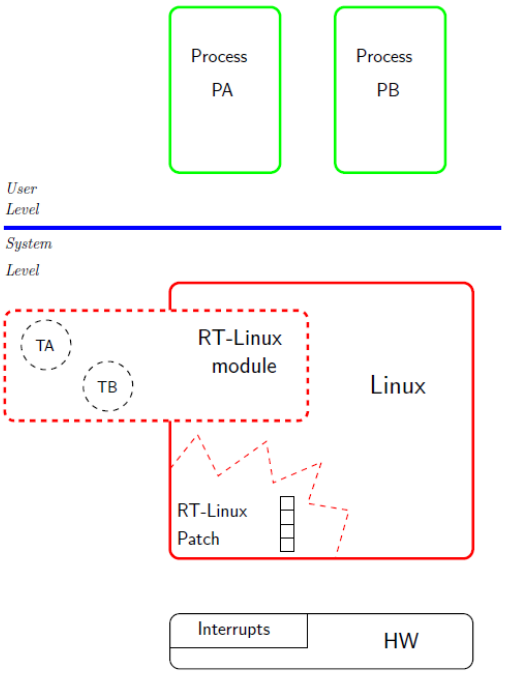
OS	site web
RTAI	http://www.rtai.org
XENOMAI	http://www.xenomai.org
ucLinux	http://www.uclinux.org (patch Linux 2.4 / intégré à Linux 2.6)
eCos	http://sources.redhat.com/ecos
MaRTE OS	http://marte.unican.es
ORK	http://polaris.dit.upm.es/ ork (Ada)
S.Ha.R.K.	http://shark.sssup.it
FIASCO	http://os.inf.tu-dresden.de/fiasc
DROPS	http://os.inf.tu-dresden.de/drops

- http://free-electrons.com/docs/realtime/
- http://www.dailymotion.com/video/xkwf6n_ubuntu-party-10-10-toulouse-linux-et-le-temps-reel-par-tech

RT-Linux

- RT-Linux a été la première approche pour rendre Linux plus temps réel
  - Consiste en un patch pour Linux, ainsi qu'un module du noyau chargeable dynamiquement
- Le patch:
  - Modifie le sous-système de gestion des interruptions de Linux, afin d'intercepter et servir les interruptions temps réel
- Le module contient:
  - Le nano-noyau (ordonnanceur + gestionnaire d'interruptions + librairies)
  - Le code applicatif
- Le code de l'application s'exécute avec les privilèges systèmes

Architecture RT-Linux



- Lorsqu'une interruption est levée
  - Si elle est pour le sous-système RT, elle est redirigée vers le nano-noyau et vers l'application RT
  - Si elle est pour Linux, elle est marquée comme étant prête
  - Et sera servie lorsque toutes les tâches RT auront terminé leur exécution
  - Linux doit être exécuté avec les interruptions activées

Priorités

- Les activités temps réel (tâches et handlers d'interruption) ont toujours la priorité sur les activités de Linux (tâches et handlers d'interruption)
- Grâce à deux mécanismes:
  - Interception des interruptions
  - Instructions cli/sti virtuelles

Interruption dans Linux

- Il faut éviter que Linux ne désactive les interruption trop longtemps
  - Sinon les activités temps réel pourraient être trop retardées
  - Une instruction de type CLI est substituée par une fonction qui marque les interruptions comme désactivées pour Linux seulement
  - Les interruptions peuvent encore être interceptées par le sous-système RT
  - Si l'interruption est pour Linux, elle est mise en attente
  - Si elle est pour le RT, elle est servie immédiatement
  - Quand Linux veut exécuter une instruction STI, une fonction est appelée
  - Elle parcours les interruptions en attente et les sert toutes

Avantages / Désavantages

- Cette approche à un grand avantage
  - Latence minimum: les interruption RT ne peuvent être bloquées par Linux, seulement par le sous-système RT
- Mais souffre de quelques désavantages
  - Les applications temps réel s'exécutent dans l'espace noyau, donc elles peuvent facilement faire crasher le système
  - La communication entre le sous-système RT et Linux est forcément non temps réel
  - Linux est ordonnancé avec une moindre priorité, donc risque de famine pour les tâches Linux
  - Il n'est pas possible d'utiliser les device drivers Linux pour le temps réel
  - Les développeurs RT doivent réécrire les device drivers pour le sous-système RT

Historique de RT-Linux

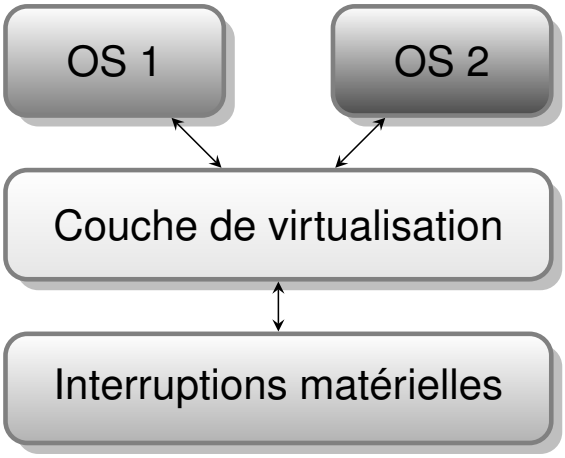
- La technique utilisée par RT-Linux a été introduite par Yodaiken e Barabanov (Université de New Mexico)
- Distribuée sous forme de logiciel libre (licence GPL)
- Le système d'interception des interruptions a ensuite été brevetée
- FSM-labs a été fondé, pour vendre une version professionnelle de RT-Linux
- FSM-labs possède le domaine http://www.rtlinux.org
- Une version libre existe encore, mais n'est plus réellement supportée

RTAI et Adeos

- Paolo Mantegazza (Polytechnique de Milan), a participé au développement de RT-Linux
- Suite à des divergences d'opinion, il quitte le développement de RT-Linux
- Développe *RTAI* (Real-Time Application Interface)
- Après des menaces de poursuites par Yodaiken, ils substituent tout le code RT-Linux par *Adeos*
- Et réécrivent le nano-noyau temps réel
- Récemment, *Xenomai* s'est détaché de RTAI

Adeos

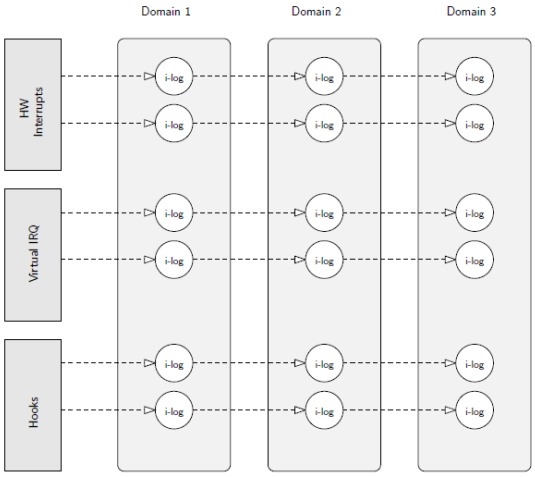
- Adeos est une couche logicielle qui permet de virtualiser les interruptions de manière générale et flexible
- Il généralise le concept de base utilisé dans RT-Linux
- Technique décrite dans un papier publié avant le brevet de RT-Linux
- Dès lors, pas de problème avec le brevet (en théorie)



Adeos

- Adeos manipule des *domaines*
- Un domaine contient une entité capable de gérer les interruptions
- Les interruptions sont ici autant matérielles que logicielles

Structure d'Adeos

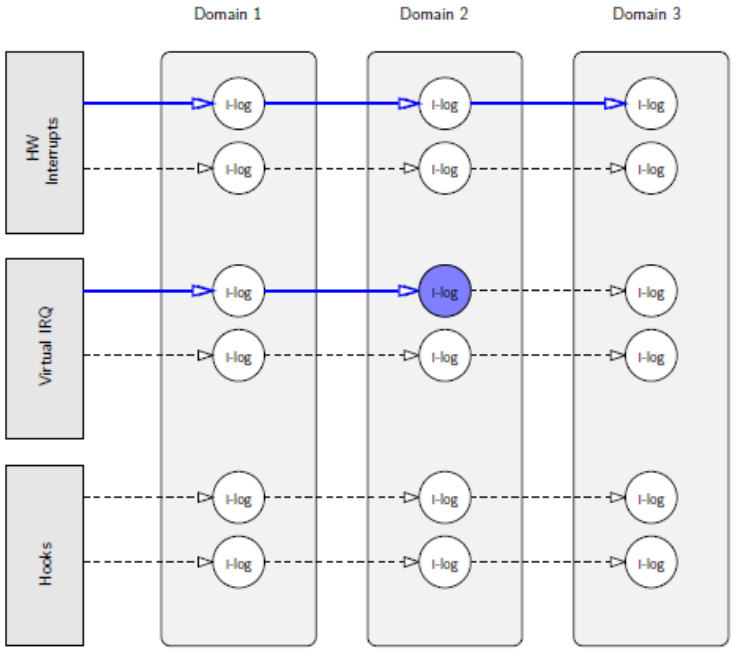


- Les domaines sont identifiés par un numéro unique
- Les domaines à numéro petit sont prioritaires pour la gestion des événements
- La propagation des événements est donc pipelinée

Règles pour la propagation des événements

- Chaque événement va du premier domaine au dernier
  - Ceci indépendamment de qui l'a généré
- Un domaine peut transmettre l'événement ou le stopper
- Un domaine peut suspendre un événement
  - équivalent à la désactivation des interruptions pour le domaines suivants
  - Les événements peuvent être "désuspendus" plus tard, et transmis aux domaines suivants
  - Les domaines précédents ne sont pas affectés par une suspension
  - Il est possible de sélectionner quels événements suspendre

Adeos - Exemple



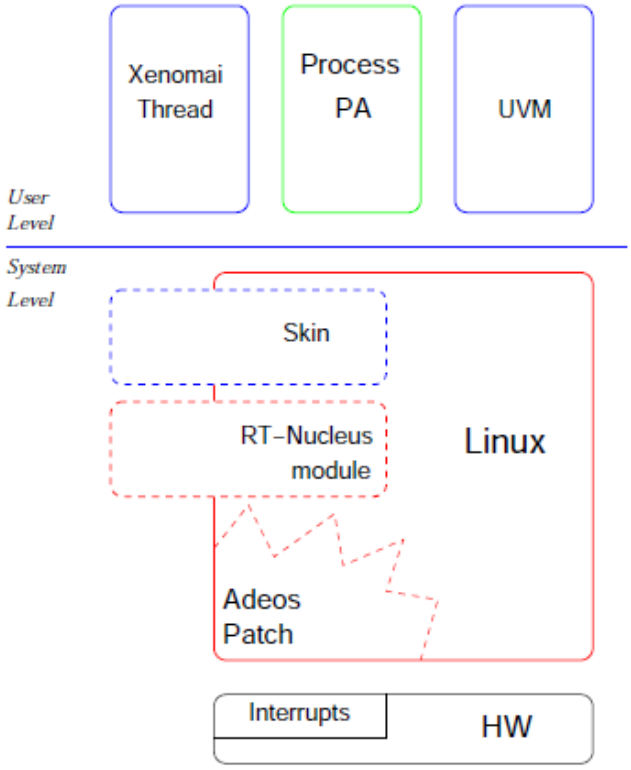
Comment utiliser Adeos

- Idée de base:
  - Linux est dans le domaine 3
  - L'OS temps réel est dans le domaine 1
  - Le domaine 2 est utilisé pour bloquer les événements
  - Les opérations Linux pour autoriser/masquer les interruptions sont modifiés en
    - Blocage/déblocage des interruptions dans le domaine 2

Historique de Xenomai

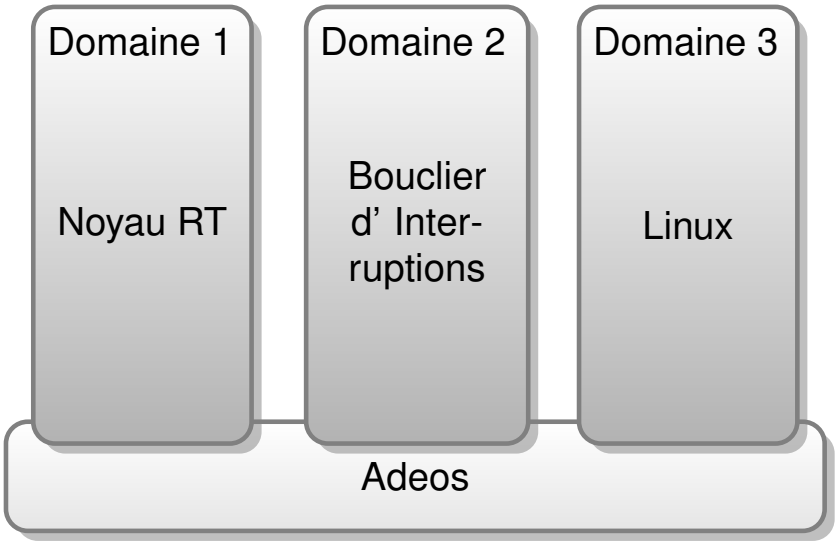
- Xenomai est un nouveau projet (2005)
- Branche de RTAI
- Le responsable, Philippe Gerum, était un des responsables de la maintenance de RTAI, puis a créé sa propre branche
- Excellent point: beaucoup de documentation

Architecture de Xenomai



- Structure similaire à RTAI
- Xenomai offre une meilleure intégration avec Linux
- Nouveau:
  - Threads Xenomai
  - Skins
  - UVMs

Domaines

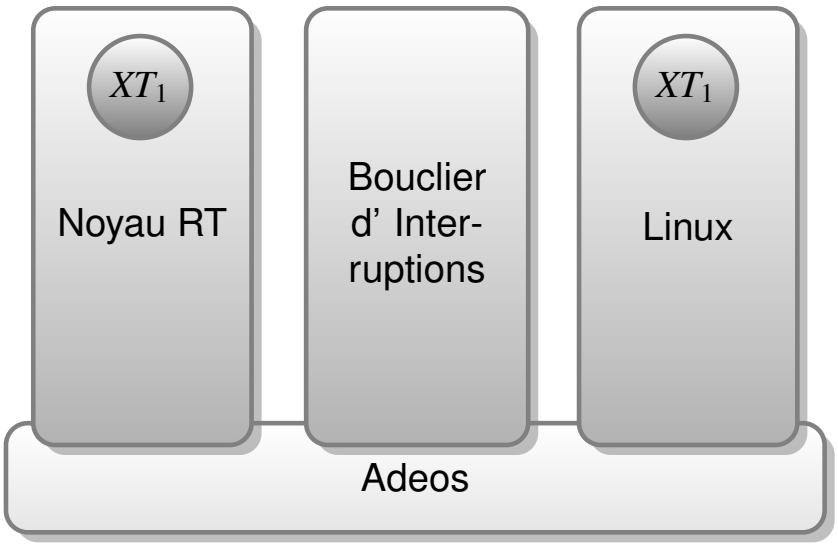


- Un domaine primaire exécute le noyau temps réel (1)
- Un domaine secondaire exécute Linux (3)
- Un domaine intermédiaire peut bloquer les interruptions (2)

Threads Xenomai

- Un thread temps réel peut s'exécuter
  - Toujours dans le domaine primaire
    - Très proche de RT-Linux ou de rt-thread RTAI
    - Espace mémoire identique à celui du noyau
    - Temps de réponse très court
- Dans le domaine secondaire et primaire
  - Appelés *Threads Xenomai*
  - Mémoire dans l'espace utilisateur
  - Mémoire protégée des autres processus
  - Ne peut pas crasher le noyau
  - Peuvent être exécutés dans le domaine primaire ou secondaire

Changement de mode



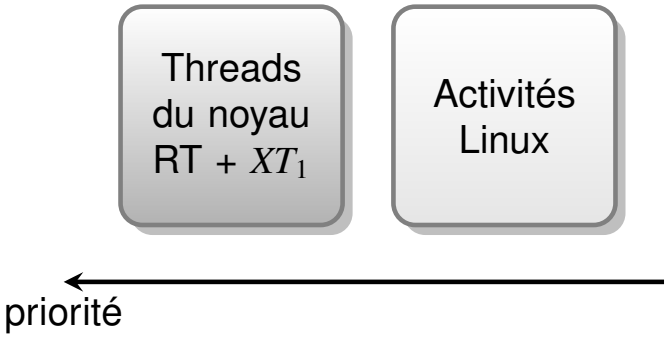
- Un thread Xenomai démarre dans le mode primaire
- Lorsqu'il invoque un appel système non RT, il saute dans le mode secondaire

Threads Xenomai

- Les priorités du domaine primaire sont compatibles avec celles du domaine secondaire
- Le noyau RT fournit 100 priorités, comme Linux
- Un thread qui passe d'un domaine à l'autre garde sa priorité
- Donc, un thread de priorité 96 dans le domaine temps réel sera ordonnancé par Linux à la même priorité s'il passe dans le domaine Linux

Threads Xenomai en mode primaire

- Un thread Xenomai démarre en général en mode primaire
- Quand le thread est en mode primaire:
  - Il est supprimé de la liste des threads prêts de Linux
  - Il est servi par l'ordonnanceur du noyau temps réel
  - Il a priorité sur les threads Linux, même sur les threads Linux de priorité supérieure
- En mode primaire, il est en concurrence avec les autres threads RT



Saut dans le mode secondaire

- Un thread Xenomai reste dans le domaine primaire jusqu'à ce qu'il appelle une primitive non RT
  - Exemple: `printf()`, ou `write()` sur un fichier
  - Dans ce cas, le thread passe dans le mode secondaire
- Quand un thread est déplacé dans le mode secondaire:
  - Le noyau TR insert le thread dans la liste Linux
  - Le noyau TR invoque l'ordonnanceur Linux
  - Tout Linux est ordonnancé avec la priorité du thread Xenomai

Priorités

- Lorsque
  - Au moins un thread Xenomai (XT_1) s'exécute en mode secondaire
  - Et XT_1 a la plus haute priorité parmis les threads RT
- Alors, la priorité est ainsi:

Priorité en mode secondaire

- Si un processus Linux avec une plus haute priorité que le thread Xenomai arrive, il est exécuté par Linux
- Si un thread RT avec une priorité plus basse que le thread Xenomai arrive, il n'y pas de préemption
- Si un thread RT avec une priorité plus haute que le thread Xenomai arrive, il y a préemption

Mélange de processus RT et de threads Xenomai

- Dangereux
- Les processus RT Linux pourraient préempter les threads Xenomai alors qu'ils sont en mode secondaire
- Suggestion: ne pas mélanger les deux

Interruptions Linux

- Pour réduire la latence:
  - Si une interruption arrive alors qu'un thread Xenomai s'exécute en mode secondaire, elle n'est pas transmise à Linux
  - Le mécanisme fonctionne ainsi:
    - Quand un thread Xenomai est exécuté en mode secondaire, le domaine de bouclier d'interruption est activé
    - Toutes les interruptions Linux sont bloquées, jusqu'à ce que le thread Xenomai termine sont exécution
    - Les interruptions sont ensuite servies lorsque Linux reprend sont exécution

Latence

- Le pire cas de latence est observé lorsque:
  - Un thread Xenomai passe en mode secondaire
  - L'ordonnanceur Linux est invoqué
  - Cependant, Linux était en train de gérer une interruption, et la préemption n'est pas autorisée
  - Il faut attendre que Linux réautorise les interruptions
- Les progrès de Linux au niveau de la réduction de la latence des interruptions devrait aider

Interfaces Xenomai

- Xenomai fournit différents API au développeur
- L'API interne (core) est utilisée en interne par le noyau RT
  - Ne devrait pas être utilisé directement
- Plusieurs *skins* construits sur cette interface
- Un de ces skins est l'interface native Xenomai
- Un skin est un module noyau chargeable

Interfaces disponibles

- native
- POSIX
- PSOS
- RTAI
- μ-Itron
- VRTX
- VxWorks
- RTDM (rt driver model)

Idée sous-jacente

- Tous les RTOS ont un comportement similaire, surtout par rapport à l'API proposé
- Les implémentations internes divergent
- Les différences d'API ne sont pas essentielles
  - Ordre des priorités
  - Files d'attente des sémaphores
  - etc.
- Les tâches sont très semblables

Tâches

- Après analyse, les développeurs de Xenomai ont trouvé que:
  - En supportant les états POSIX
  - Et les états de μ-Itron
- Les états des tâches de toutes les interfaces étaient gérés
- L'interface interne:
  - Est essentielle
  - Tout appel système peut être implémenté comme une séquence d'appels internes

Skin RTDM

- En utilisant le mécanisme des threads Xenomai (passage d'un mode à l'autre)
- Il est possible d'écrire un device driver dans l'espace utilisateur, avec un bon temps de réponse
- L'idée est qu'un thread peut attendre une interruption
- Le handler d'interruption est embarqué dans un thread Xenomai
- Le thread attend sur une ligne d'interruption (est bloqué)
- Quand l'interruption est levée, le noyau RT débloque le thread

Conclusion

- Xenomai permet des temps de réponse temps réel dans l'espace utilisateur, avec:
  - Protection mémoire
  - Facilité de débugging
- Il fournit une couche d'émulation (UVM) pour faciliter l'observation de l'application (GDB)
- Il fournit des skins, et donc accepte plusieurs interfaces RTOS

Application Xenomai

Include

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <sys/mman.h>

#include <native/task.h>
#include <native/timer.h>
```

Application Xenomai

Tâche

```
RT_TASK demo_task;

void demo(void *arg)
{
    RTIME now, previous;

    rt_task_set_periodic(NULL, TM_NOW, 1000000000);
    previous = rt_timer_read();

    while (1) {
        rt_task_wait_period(NULL);
        now = rt_timer_read();

        printf("Time since last turn: %ld.%06ld ms\n",
               (long)(now - previous) / 1000000,
               (long)(now - previous) % 1000000);
        previous = now;
    }
}
```


Application Xenomai

Main (tâche principale)

```
void catch_signal(int sig)
{
    rt_task_delete(&demo_task);
}

int main(int argc, char* argv[])
{
    signal(SIGTERM, catch_signal);
    signal(SIGINT, catch_signal);

    /* Avoids memory swapping for this program */
    mlockall(MCL_CURRENT|MCL_FUTURE);

    rt_task_spawn(&demo_task, "trivial", 0, 99, 0, demo, 0);

    pause();
}
```

Application vs. module: Interruption

Module

L'interruption cause l'exécution de l'ISR

```
// Création d'une interruption depuis l'espace noyau
int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq,
                    rt_isr_t isr, rt_iack_t iack, int mode)
```

Application

L'interruption doit être gérée par une tâche différée

```
// Création d'une interruption depuis l'espace utilisateur
int rt_intr_create (RT_INTR *intr, const char *name, unsigned irq,
                    int mode)
int rt_intr_wait(RT_INTR *intr, RTIME timeout)
```


Application vs. module: Alarme

Module

L'alarme cause l'exécution du handler d'alarme

```
// Création d'une alarme depuis l'espace noyau
int rt_alarm_create (RT_ALARM *alarm, const char *name,
                     rt_alarm_t handler, void *cookie)
```

Application

L'alarme doit être gérée par une tâche différée

```
// Création d'une interruption depuis l'espace utilisateur
int rt_alarm_create (RT_ALARM *alarm, const char *name)
int rt_alarm_wait(RT_ALARM *alarm)
```