Linux temps réel avec Xenomai

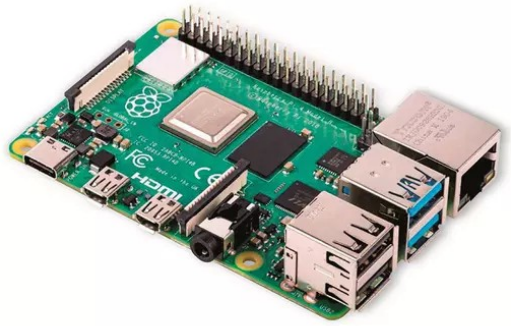
MI11 P22

Guillaume Sanahuja

gsanahuj@hds.utc.fr

Linux temps réél avec Xenomai

- un cours, trois TP sur la Raspberry Pi 4/ Joy-Pi Note :



  - TP 1 :
    - Prise en main Xenomai (tâches, synchro, latence)
  - TP 2 :
    - Pathfinder
  - TP 3 :
    - à définir

Préambule

- Conception temps réel, analyse du besoin en regard des ressources disponibles (économiques, techniques et humaines)
  - Besoin réel de performances temps réel ? NON → utiliser un OS standard.
  - Est-ce réalisable en bas niveau ? OUI → circuit électronique ou microcontrôleur + asm/C
  - Sinon, choix de la carte/processeur puis choix du système logiciel (Linux embarqué + Xenomai ou autre)
  - Attention à bien utiliser les tâches temps réel pour répondre aux fonctionnalités temps réel uniquement (ex : lecture capteurs, contrôle actionneurs, …)

Exemple : véhicule APAChE

- véhicule Autonome Partenaire de Conduite
- Véhicule robotisé

- Application des exigences :
  - Besoin réel de performances temps réel ? **OUI en partie**, on doit garantir un déterminisme pour assurer la sécurité de la conduite.
  - Est-ce réalisable en bas niveau ?
    - **OUI**, pour le contrôle commande, les actionneurs sont contrôlés par les cartes électroniques de série (Embedded Control Unit) et le management est assuré par une microautobox de dSPACE (calculateur programmé en C sans OS “lourd”)
    - **NON**, pour la perception, localisation et navigation. Choix d'un OS Linux standard car ici pas de contraintes temps réel. Un mécanisme de watchdog est en place entre la partie temps réel et non temps réel, la fonction de repli est assurée par le conducteur.

Exemple : Drones

- Application des exigences :
  - Besoin réel de performances temps réel ? **OUI en partie**, on doit garantir un déterminisme pour assurer la stabilité du vol.
  - Est-ce réalisable en bas niveau ?
    - **OUI**, pour les contrôleurs de moteurs (triphasés synchrones) réalisés avec des microcontrôleurs.
    - **NON**, pour les lois de commandes, le filtrage capteurs et la fusion de données qui peuvent demander beaucoup de ressources (CPU+RAM): utilisation d'un OS temps réel
    - **NON**, pour la perception, localisation et navigation. Choix d'un OS Linux standard car ici pas de contraintes temps réel.

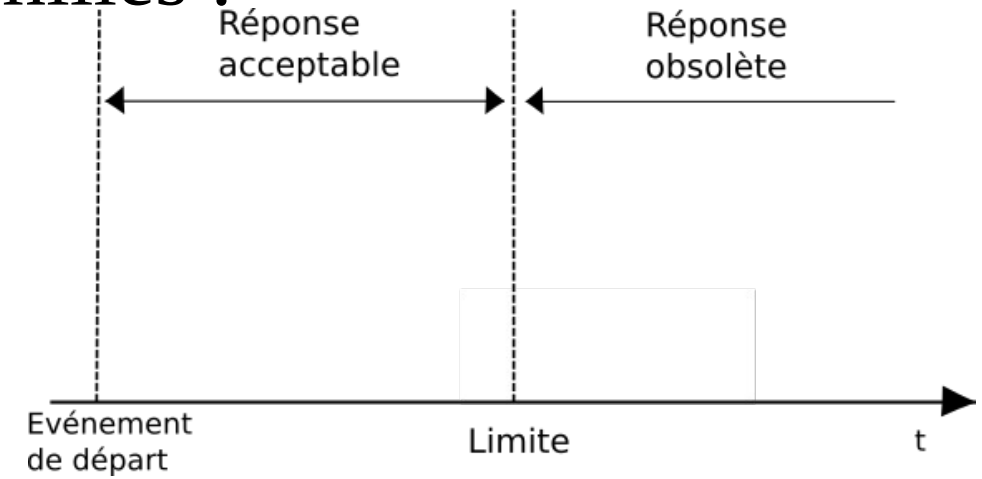
Sommaire

1. Introduction temps réel

1. Introduction temps réel

Définitions

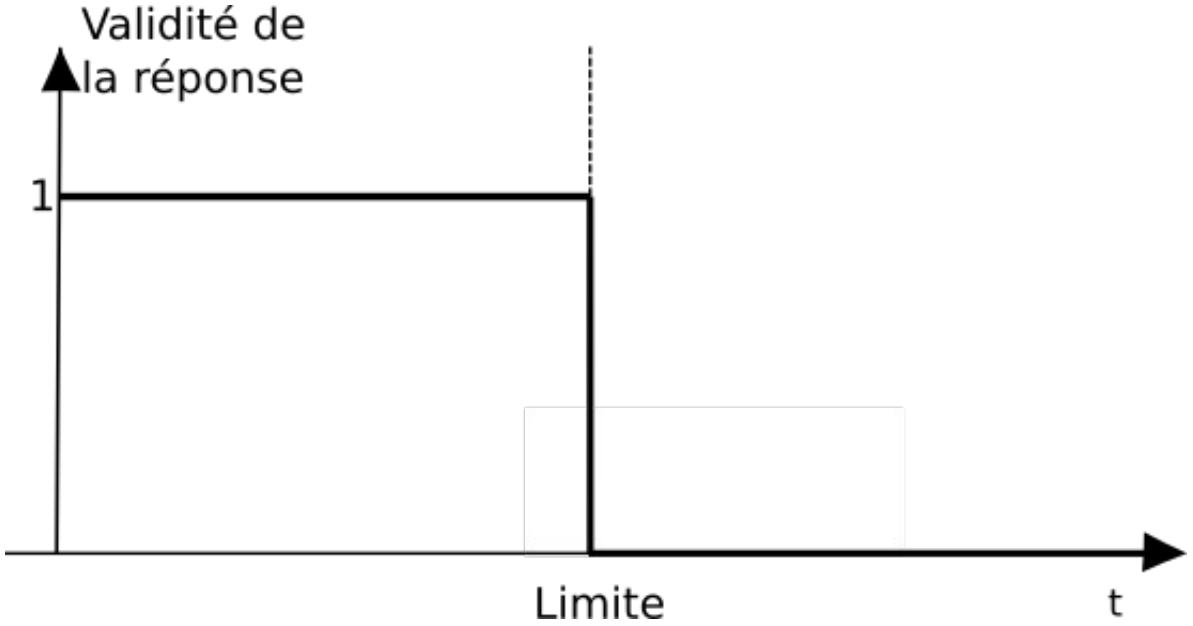
- Un système temps réel est un système qui fournit un résultat dans un temps garanti prévu au préalable
- Un système informatique est soumis à des contraintes temps réel si l'instant auquel il parvient au terme d'une opération entre en considération dans la validité du résultat de cette opération
- On distingue deux grandes familles :
  - Temps réel dur (strict)
  - Temps réel mou (souple)



1. Introduction temps réel

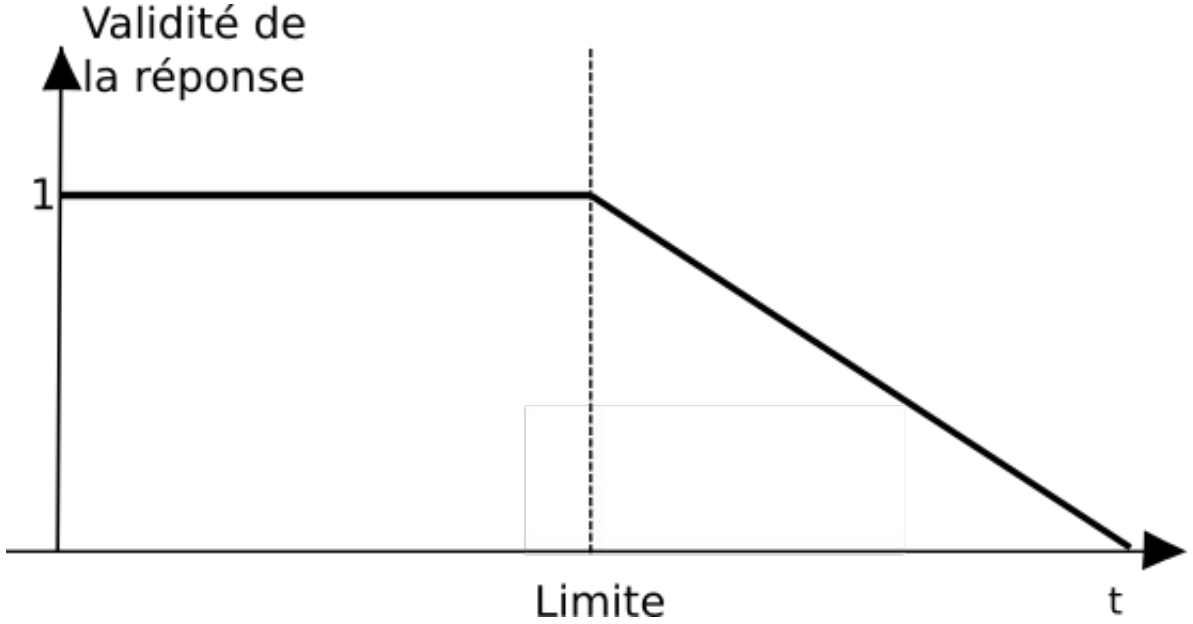
Temps réel dur

- En temps réel dur, si la limite de validité de la réponse est dépassée, le système aboutit sur un comportement catastrophique (événement redouté)
  - Ex : cas d'un système de contrôle commande (atterrissage d'un avion)



Temps réel mou

- En temps réel mou, on définit une limite admissible pour la fourniture du résultat. Passé ce délai, l'intérêt de la réponse décroît Pas de catastrophe.
  - Ex : système temps réel ne présentant pas de risques graves en cas de non réponse (streaming vidéo).



1. Introduction temps réel

Solutions “dures” libres

- RTLinux
  - Pionnier, Université du Nouveau Mexique
    - Dépôt de brevet n°59957450 en 1999 par FSMLabs
  - Racheté par WindRiver en 2007
- RTAI
  - Évolution de RTLinux
  - Développé à Politecnico di Milano
- Xenomai
  - Projet démarré en 2001 par Philippe Gerum

1. Introduction temps réel

Solutions “dures” propriétaires

- Montavista
  - Hard hat linux
- Windriver
  - Utilise la technologie RTLinux
- LynuxWorks
  - Blue cat linux
  - Compatibilité binaire avec LynxOS (OS certifié aéronautique)

Problèmes classiques

- Problèmes rencontrés dans le développement temps réel :
  - Démarrage en Round Robin
  - Inversion de priorité

Démarrage en Round Robin

- Ordonnancement Round Robin : tranches de temps fixes allouées à chaque processus (quantum).
- Illustration du problème :
  - Un programme lance 5 threads en série qui effectuent une boucle active pendant quelques secondes puis se terminent.
  - Résultat :

```
#./threads_rr
thread  start_time  end_time
[0]     1212033956  1212033959
[1]     1212033959  1212033962
[2]     1212033962  1212033965
[3]     1212033965  1212033968
[4]     1212033968  1212033971
#
```

Cela ressemble-t-il à un ordonnancement ”Round Robin” ?

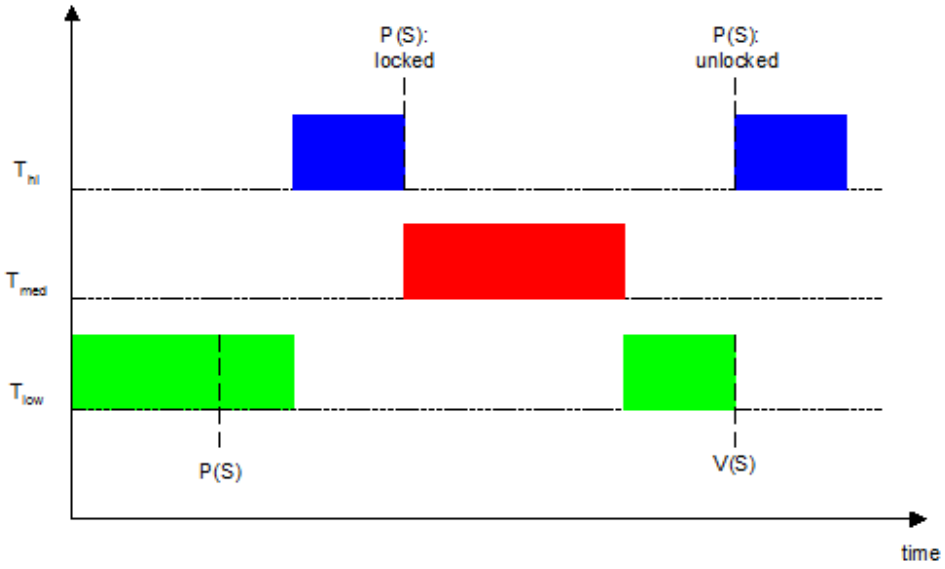
Démarrage en Round Robin

- Réponse : Non pas vraiment, cela ressemble à un ordonnancement FIFO où les tâches s'exécutent dans leur ordre d'arrivée.
- Cause : le *main* est ordonnancé en temps partagé et est donc préempté dès que la première tâche temps réel se lance.
- Solution : utilisation d'une barrière (ex : `pthread_barrier_t`) initialisée avec le nombre de thread à synchroniser.

```
#./threads_rr
thread  start_time  end_time
[0]     1212034000  1212034003
[1]     1212034000  1212034003
[2]     1212034000  1212034003
[3]     1212034000  1212034003
[4]     1212034000  1212034003
#
```

Inversion de priorité

- Problématique :
  - La tâche C de basse priorité démarre et prend la ressource partagée
  - La tâche A de haute priorité démarre, préempte C, et est mise en attente de la ressource partagée
  - La tâche B de plus basse priorité que A prend donc le CPU
  - **La tâche A dépend de C ce qui est normal (partage de ressource) mais devient également dépendante de la fin de B alors que fonctionnellement A et B ne sont pas liées**

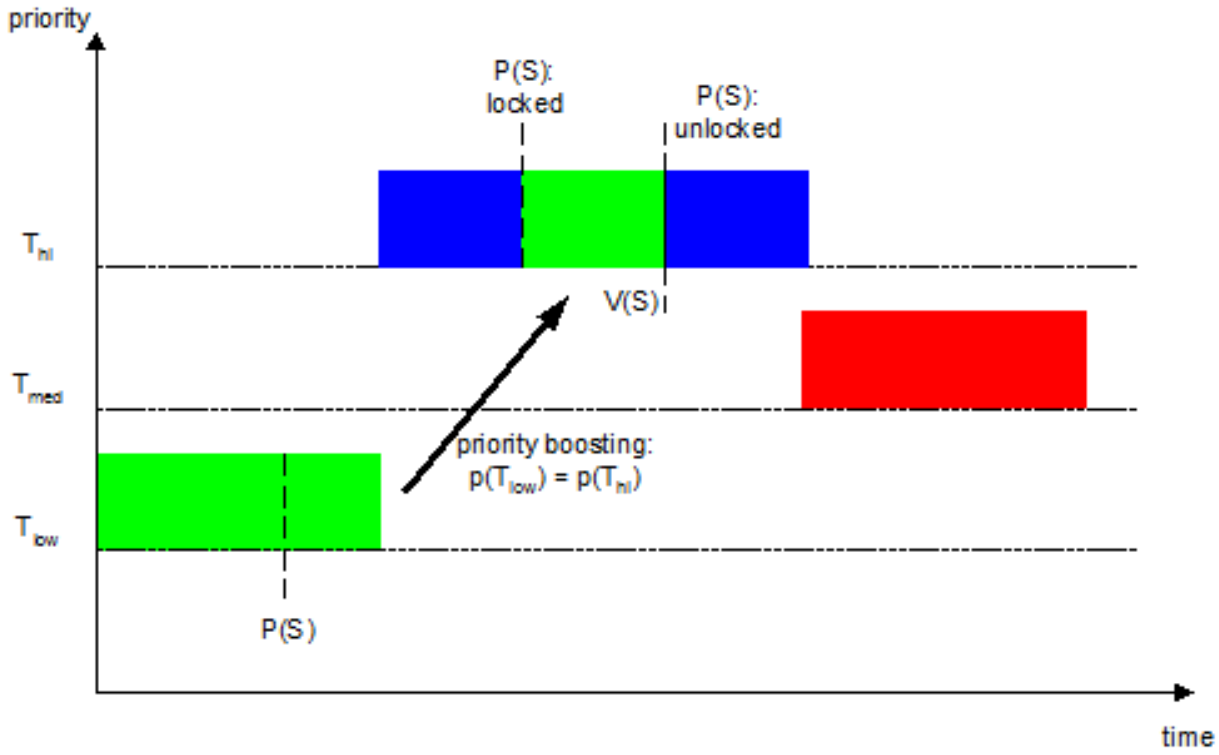


http://dchabal.developpez.com

Inversion de priorité

- Solution : héritage de priorité (2 règles)
  - Un mutex hérite de la priorité la plus élevée parmi celles de tous les threads qui le demandent
  - Un thread s'exécute avec la priorité la plus élevée parmi celles de tous les mutex qu'il tient

Inversion de priorité



http://dchabal.developpez.com

A retenir :

- Une tâche de très haute priorité est bloquée par une ressource partagée et la tâche de plus basse priorité s'exécute.
- Souvent dû à un problème de conception.

Éléments de synchronisation

- Protection d'accès à une ressource partagée
  - Évite la corruption de données lors d'accès simultanés en lecture / écriture
  - Objets de synchronisation : mutex, sémaphores
- Héritage de priorité
  - Fonctionnalité offerte par les mutex
  - Utilisation de sémaphore pour l'accès à une ressource partagée vous expose au problème d'inversion de priorité
  - Pourquoi ?
    - Un mutex est tenu par un processus
    - Un sémaphore est un “compteur” et n'est pas directement associé à un processus

2. Temps réel sous Linux?

2. Temps réel sous Linux?

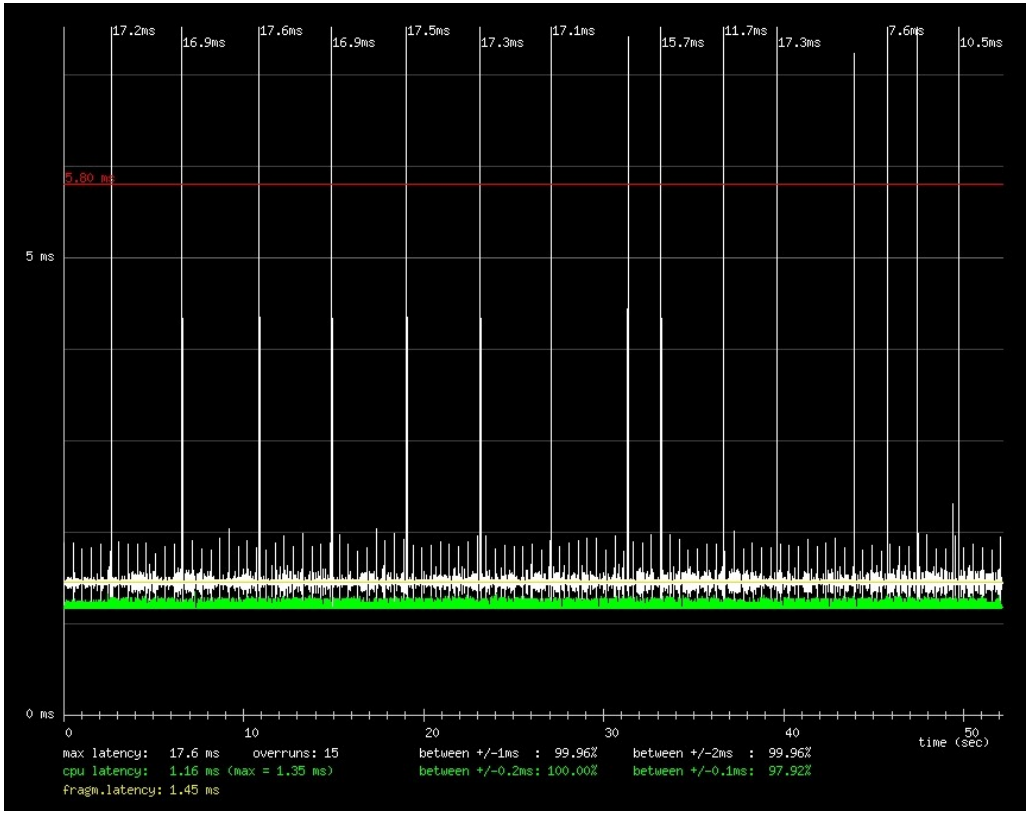
Noyau standard

- Le noyau Linux standard n'est pas temps réel
- Le noyau peut être non préemptif
- Le code noyau en général et celui des pilotes en particulier peut masquer les interruptions pendant des durées indéfinies
- L'accès à la RAM se fait par pagination à la demande
- L'algorithme d'ordonnancement standard est de répartir le CPU équitablement entre tous les processus
  - ordonnancement à temps partagé

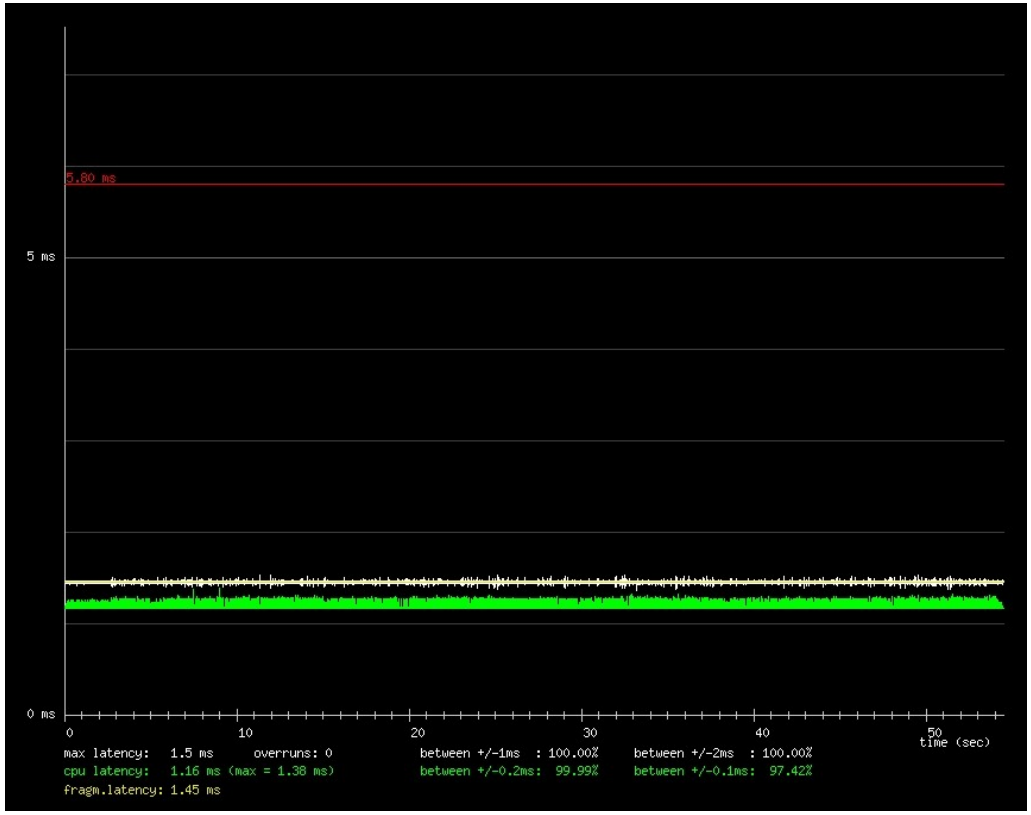
2. Temps réel sous Linux?

Noyau standard

Exemple: décodage audio



Noyau non préemptif



Noyau préemptif

Source: linuxjournal.com

2. Temps réel sous Linux?

Noyau standard

- Les performances temps réel des noyaux ont été grandement améliorées
  - Interruptions gérées par des threads
  - Timers haute résolution
  - Amélioration de la préemptivité du noyau (préemptif/non préemptif/points de préemptions)
  - Patch RT Preempt
  - Problème de l'inversion de priorité
    - Une tâche de basse priorité occupe le temps CPU à cause d'accès concurrents à une ressource partagée par des tâches de plus haute priorité
    - Une solution : héritage de priorité

2. Temps réel sous Linux?

Ordonnancement

- Priorité temps réel dans l'intervalle [1,99], les tâches de priorité 0 sont dans le contexte de temps partagé (l'ordonnancement est géré sur le principe de “nice” [-20,19]).
  - Nice faible => plus prioritaire
  - Pour le temps réel, priorité 99 le plus prioritaire
- Préemption automatique d'une tâche en cours d'exécution par une tâche de priorité plus élevée
- Politiques d'ordonnancement temps réel normalisées POSIX :
  - FIFO (premier arrivé premier servi): tâche la plus élevée lancée en premier. Tâches de même priorités par ordre d’arrivée. La tâche utilise le CPU jusqu’à ce qu’elle se termine/rende la main
  - Round Robin: lorsqu'une tâche est activée un délai lui est accordé à l'issue duquel elle laissera la main aux tâches de priorité équivalente

3. Présentation de Xenomai

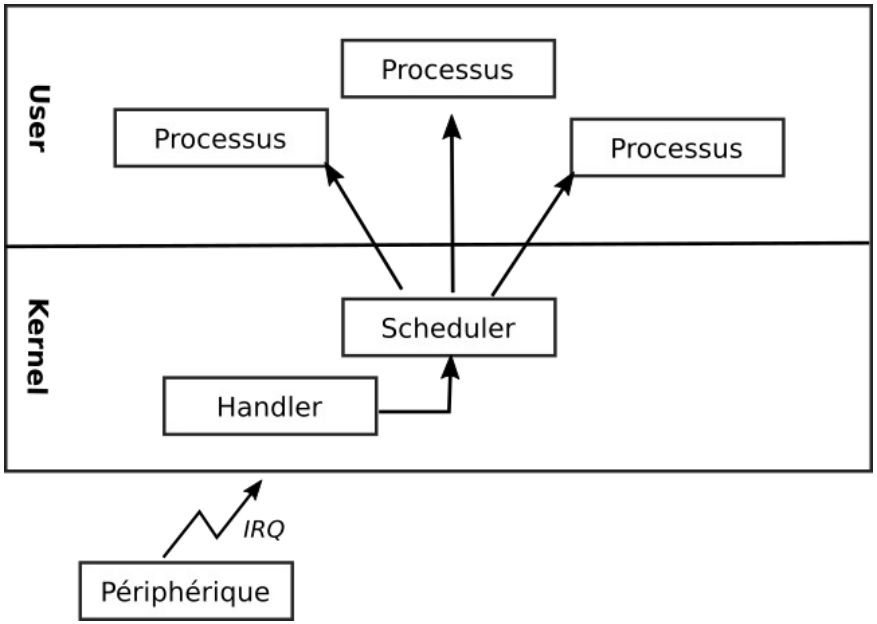
3. Présentation de Xenomai

Historique: RTLinux

- Premier projet : RTLinux
  - Développé par Victor Yodaiken et Michael Barabanov.
  - Devenu un produit commercial supporté par la société FSMLabs
  - Breveté en 1999 : “système d'exploitation temps réel exécutant des tâches temps réel et exécutant un système d'exploitation généraliste comme tâche de fond”
  - Racheté par WindRiver en 2007

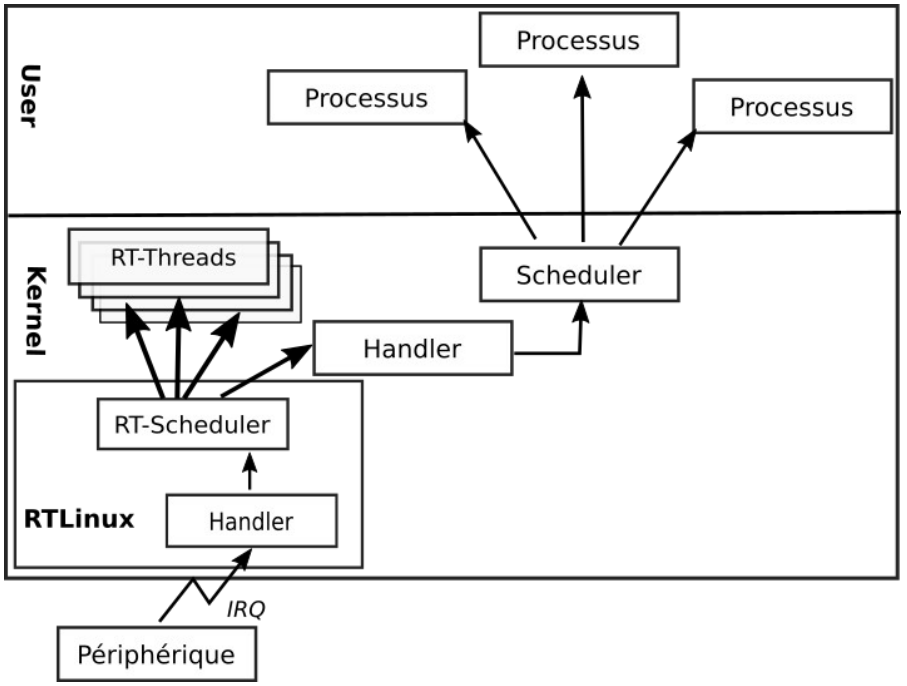
3. Présentation de Xenomai

Historique: RTLinux



- Ordonnancement Linux standard

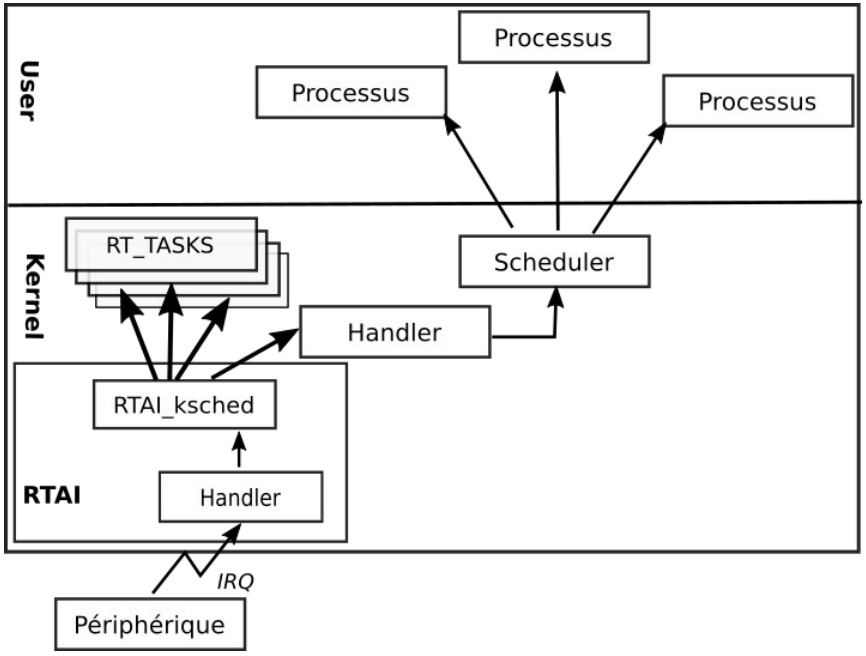
- Ordonnancement RT Linux



3. Présentation de Xenomai

Historique: RTAI

- Real Time Application Interface
- Développé par Paolo Mantegazza de l'université polytechnique de Milan dans les années 1990
- Sous licence LGPL initialement
- Passe sous licence GNU GPL en mars 2002, contraint par le brevet de FSMLabs :
  - Licence couverte par la licence libre de Open RTLinux
  - Ferme la porte à bon nombre d'applications commerciales



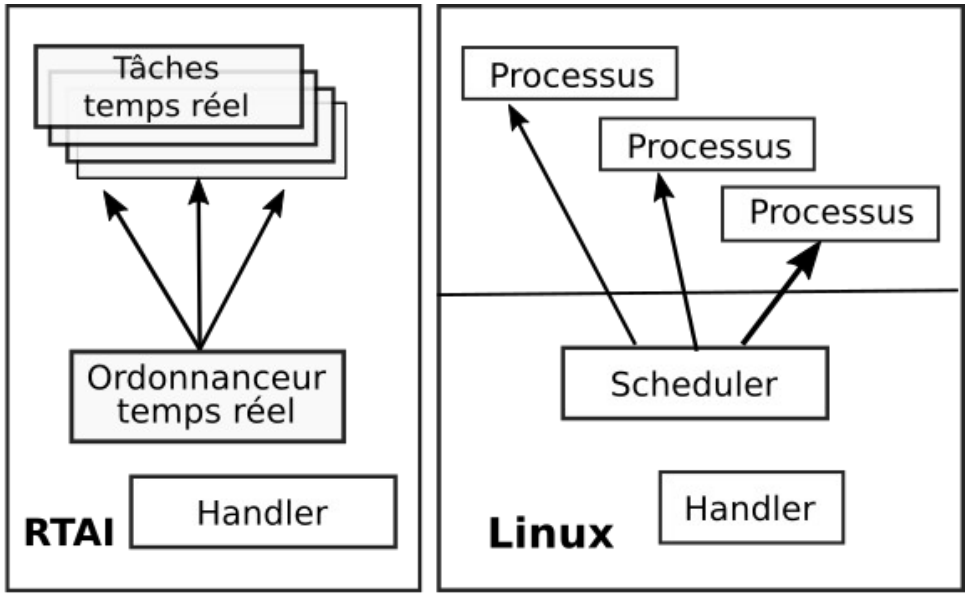
3. Présentation de Xenomai

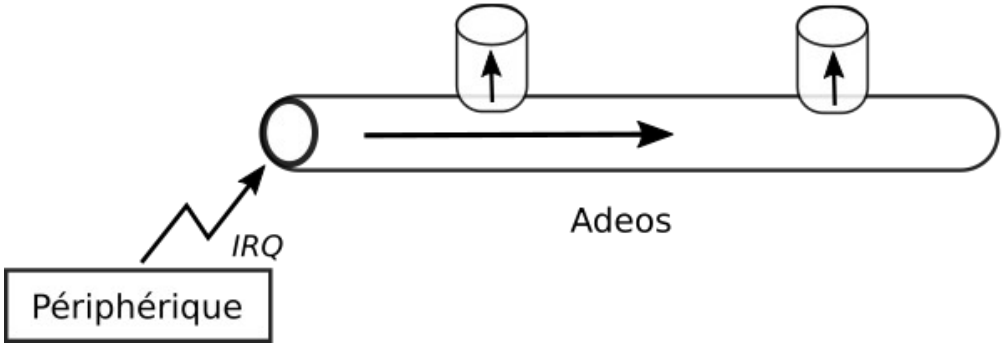
Historique: ADEOS

- En 2001, Karim Yaghmour publie un article sur un mécanisme de gestion d'interruptions nommé ADEOS (“Adaptative Domain Environment for Operating Systems”).
- Principe :
  - Couche logicielle basse en charge de capturer les interruptions matérielles et de les envoyer dans un pipeline (*i-pipe*)
  - Similaire au concept actuel de virtualisation
- Intérêt :
  - Fonctionnalités similaires à RTLinux
  - Pas de référence à un ordonnanceur temps réel, ni à un OS généraliste en tâche de fond ⇒ donc hors de portée du brevet FSMLabs
- Implémentation par Philippe Gerum, développeur français, comme couche basse de RTAI

3. Présentation de Xenomai

Historique: ADEOS/RTAI

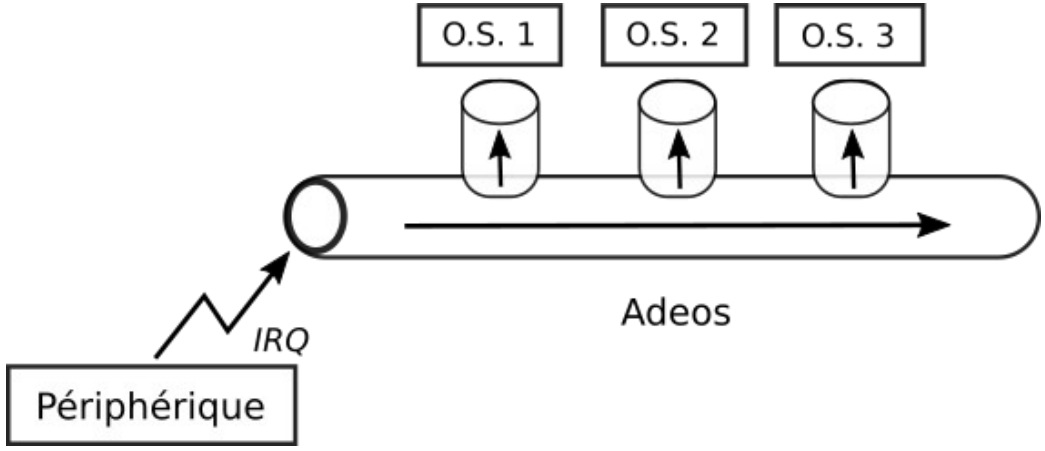




3. Présentation de Xenomai

Xenomai, concepts de base

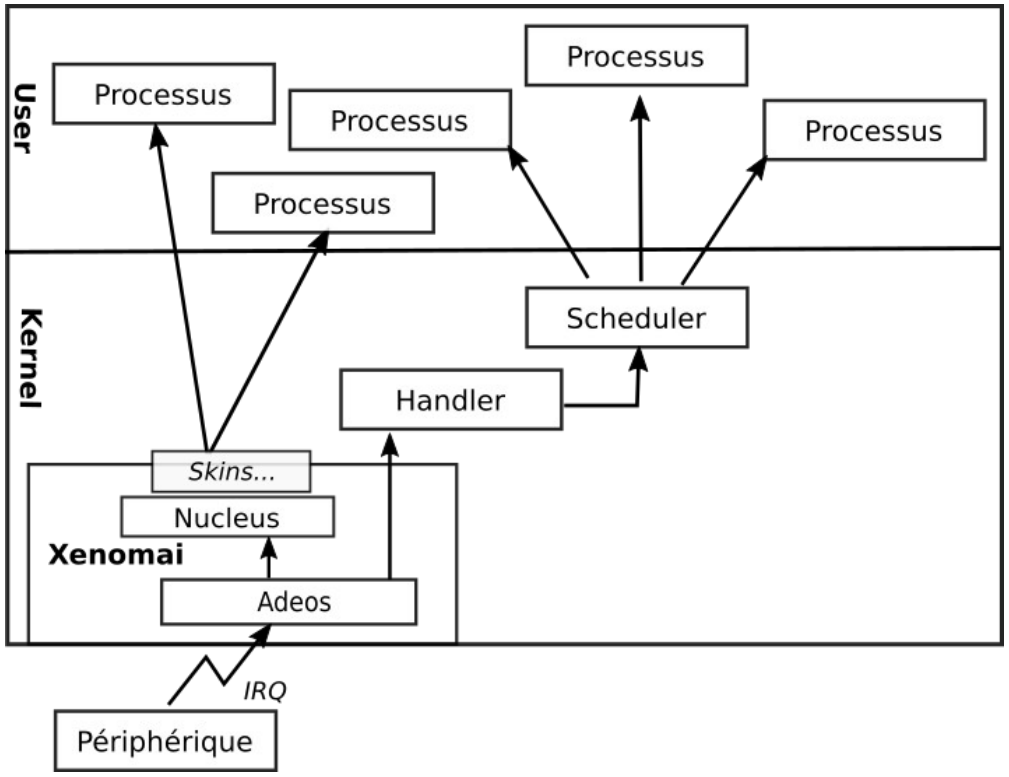
- Créé par Philippe Gerum en 2001, initialement intégré dans RTAI sous le nom de RTAI/Fusion puis devenu indépendant en 2005
- Utilisation d'ADEOS pour la virtualisation de l'interface OS/Matériel
  - Partage de ressources matérielles à travers d'OS
- ADEOS crée des "domaines" :
  - Utilisation de Linux comme "hôte" de démarrage (domaine principal)
  - Implantation d'un cœur d'exécutif temps réel (domaine prioritaire)



3. Présentation de Xenomai

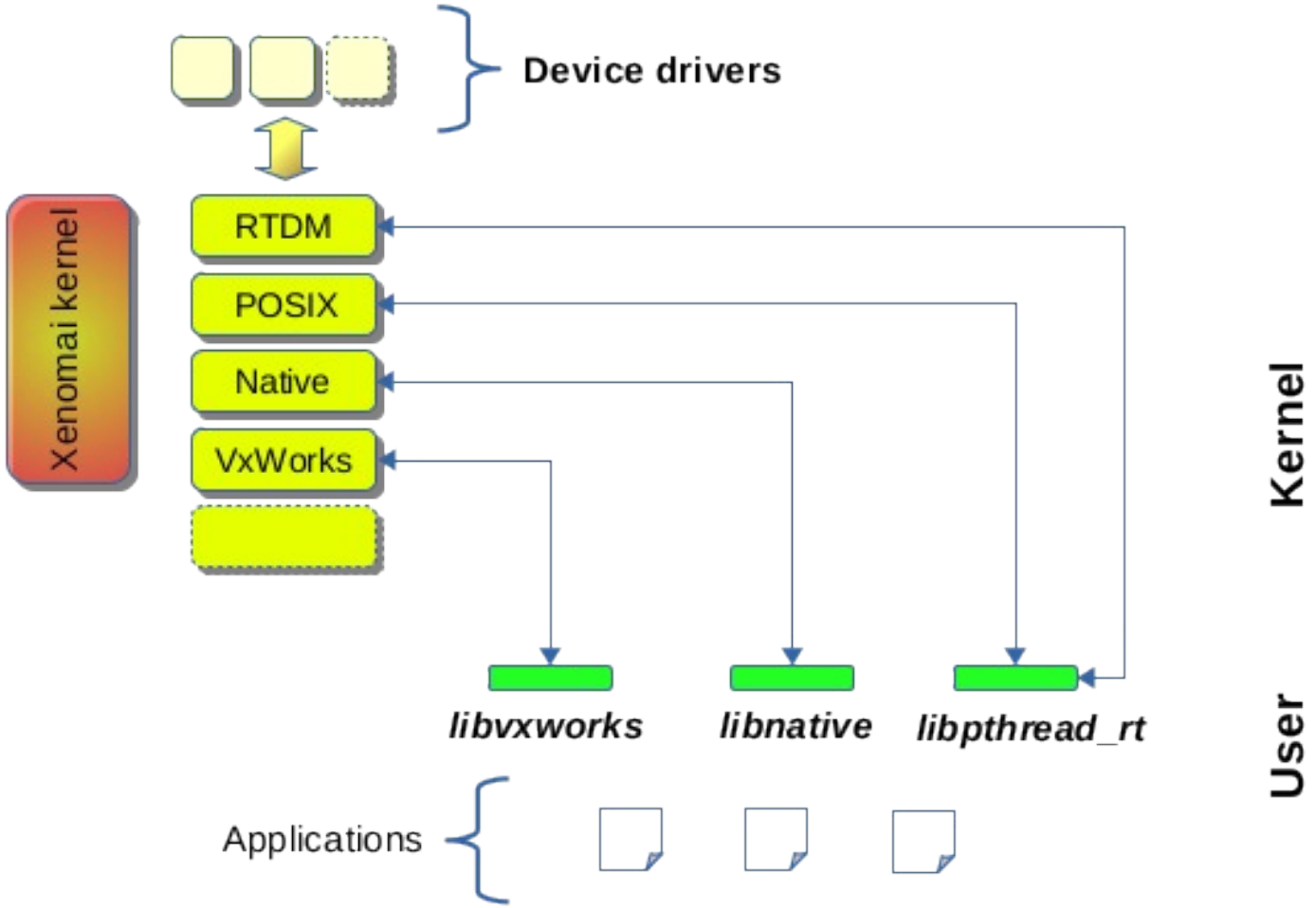
ADEOS

- Nucleus : ordonnanceur simple et déterministe installé dans le premier domaine d'ADEOS
- Le noyau Linux aura accès au CPU uniquement lorsque tous les threads Nucleus seront endormis.
- Pipeline d'interruptions, interruptions remontées au noyau Linux une fois traitées par Nucleus



3. Présentation de Xenomai

Architecture Xenomai 2



https://source.denx.de/Xenomai/xenomai/-/wikis/home

3. Présentation de Xenomai

Xenomai 2

- On peut créer des “personnalités” (skins) temps-réel
  - Facilité d'intégration de codes provenant d'autres OS temps réel dont les API spécifiques sont mappées sur l'API native Xenomai.
  - Principales skins disponibles : Native, POSIX PSE51, RTLinux, RTAI, VxWorks (Wind River Systems), PsoS (Wind Rivers, abandonné), VRTX (Mentor Graphics), uITRON (micro-ITRON)…

- Choix entre mode utilisateur et mode noyau
  - Le programme temps réel peut être développé au choix en mode noyau ou en mode utilisateur avec la même API
  - Dans le cas de RTLinux et RTAI cela est possible mais avec une API différente

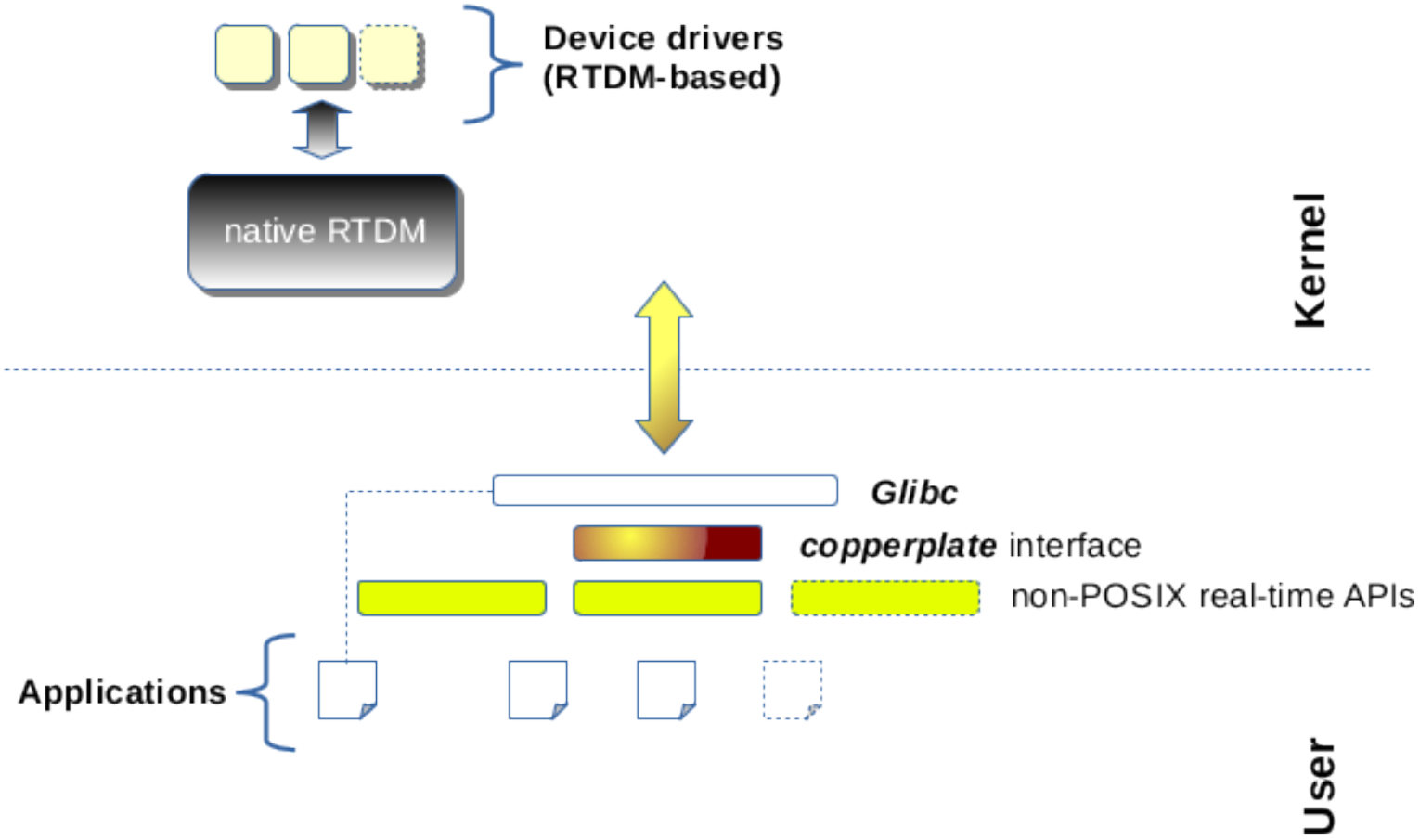
3. Présentation de Xenomai

Xenomai 3

- Evolution en 2015, suite aux améliorations des performances RT du noyau Linux
  - Mode Mercury: utilisation d'un noyau standard (ou mieux une version PREEMPT_RT)
  - Mode Cobalt: utilisation de l'*i-pipe*
  - API identique dans les 2 modes
  - Moins de skins disponibles que dans Xenomai 2
  - L'API Native devient Alchemy (très semblable)

3. Présentation de Xenomai

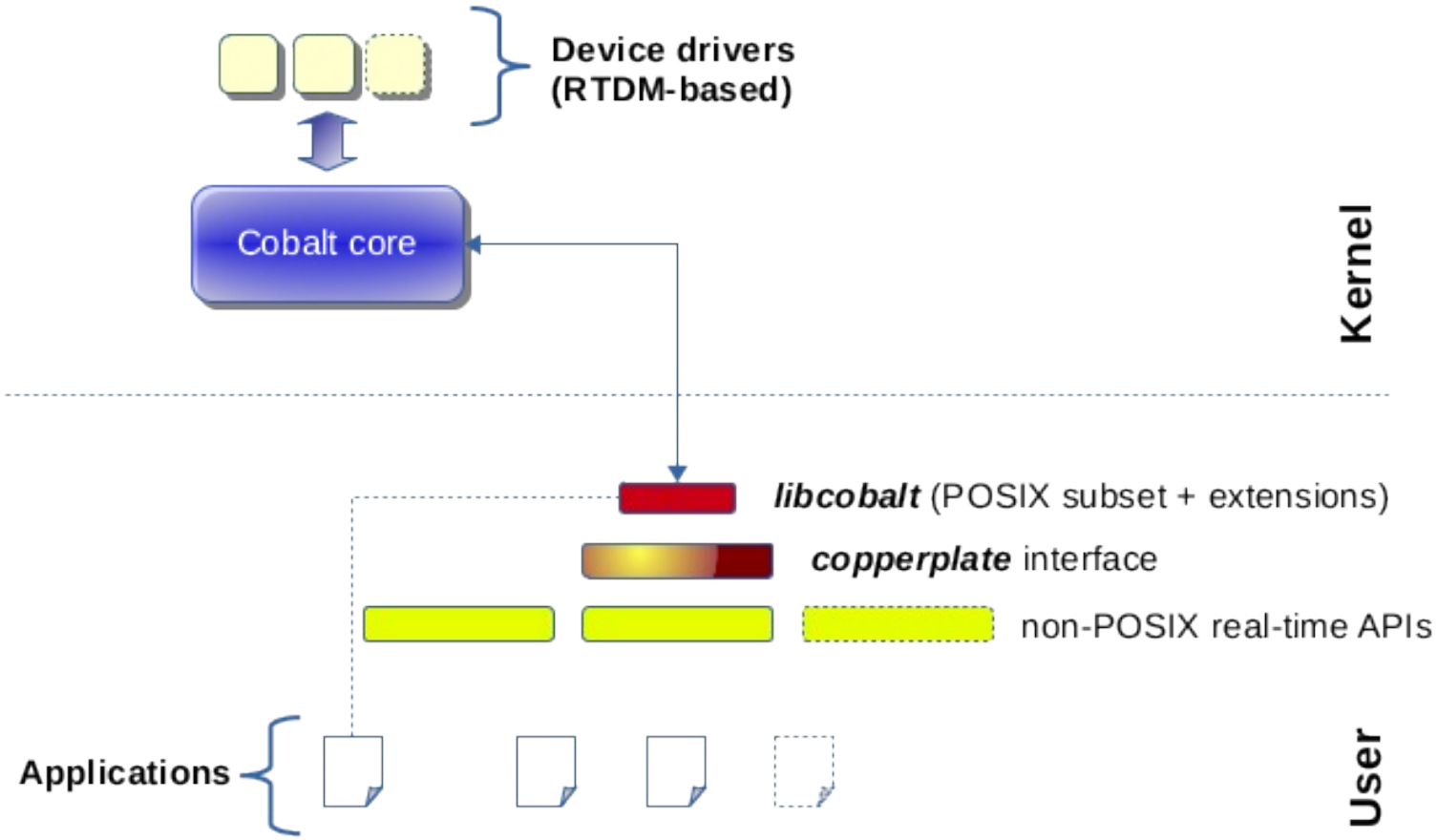
Architecture Mercury Xenomai 3



https://source.denx.de/Xenomai/xenomai/-/wikis/home

3. Présentation de Xenomai

Architecture Cobalt Xenomai 3



https://source.denx.de/Xenomai/xenomai/-/wikis/home

3. Présentation de Xenomai

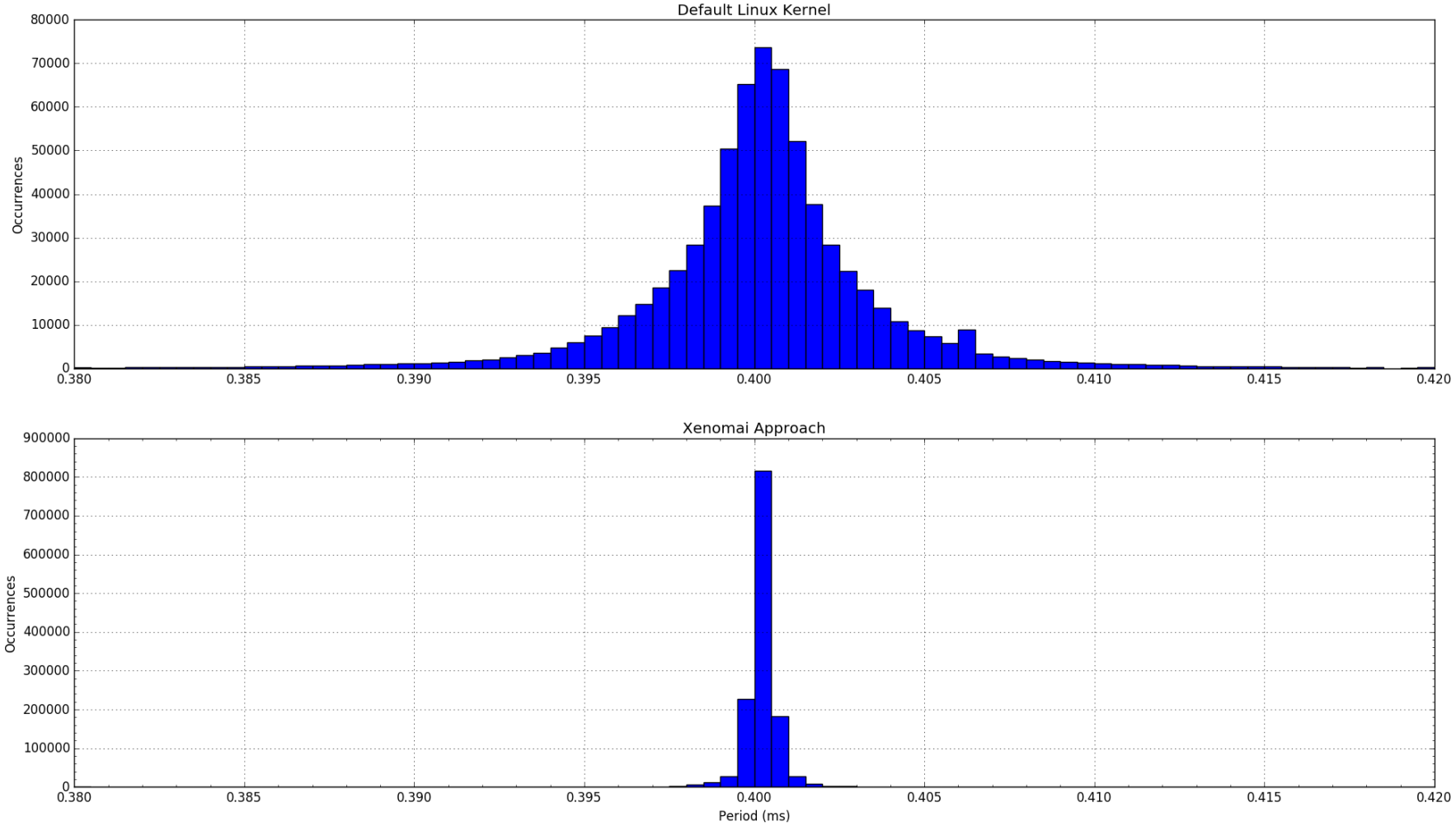
Avantages

- Très bonnes performances
  - Réécriture complète d'un micro noyau “abstrait”
  - Indépendance par rapport à Linux
- Compatibilité
  - API Posix standard
  - Exploitation de codes tiers écrit pour d'autres RTOS, possibilité d'utiliser les APIs RTAI, VxWorks,...

Risques

- Un thread temps réel fait un appel système Linux, ce qui provoque l'arrêt de l'ordonnancement Nucleus (inversion de priorité)
- Xenomai définit 2 modes :
  - Primaire : thread sous le contrôle de Nucleus
  - Secondaire : thread sous le contrôle du noyau Linux

Tests de performances



Source: cnx-software.com

- Timer 2.5 kHz (400us)

4. Installation de Xenomai Cobalt

4. Installation de Xenomai Cobalt

Sources

- Télécharger Xenomai
  - https://source.denx.de/Xenomai/xenomai/-/tags
  - `tar -xf xenomai-<version>.tar.bz2`
- Xenomai est séparé en 2 parties distinctes
  - La partie du code noyau dans `kernel`
  - La partie du code utilisateur dans `lib` (librairies)

4. Installation de Xenomai Cobalt

Patch du noyau Linux

- Télécharger un patch ipipe :
  - `https://xenomai.org/downloads/ipipe`

- Télécharger le noyau correspondant :
  - https://kernel.org/
  - Prendre même version que le patch !

4. Installation de Xenomai Cobalt

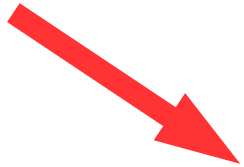
Patch du noyau Linux

- Appliquer le patch
  - 3 options :
    - `--linux` : l'emplacement des sources du noyau à patcher
    - `--ipipe` : le patch à appliquer
    - `--arch` : architecture cible
  - `cd xenomai-<version>`
  - `./scripts/prepare-kernel.sh --linux=../linux-<version du noyau> --ipipe=../ipipe-core-<version du noyau>-arm-<version du patch>.patch --arch=arm`

4. Installation de Xenomai Cobalt

Compilation du noyau Linux

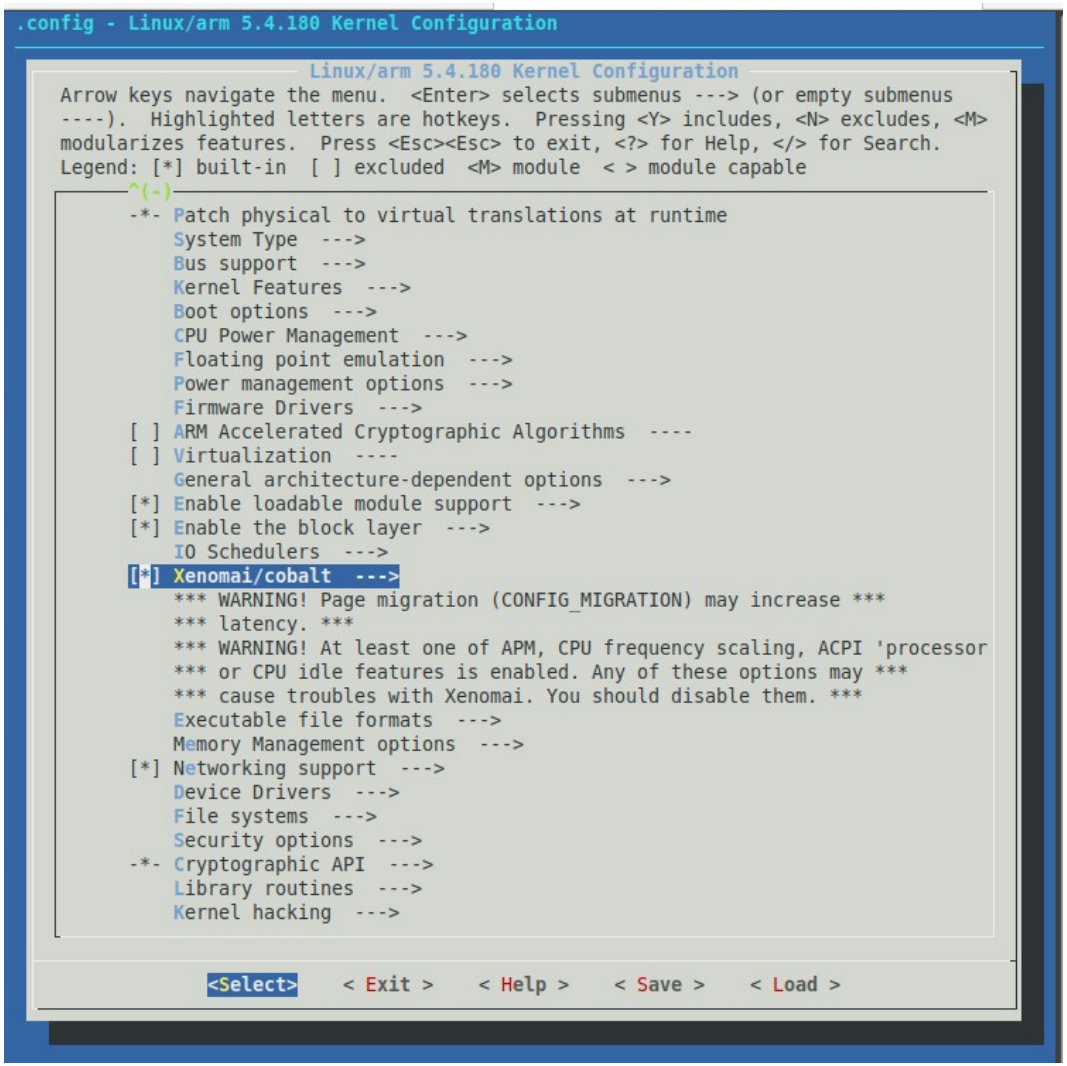
- Modifier la configuration
  - `make ARCH=<arm> menuconfig`
  - Les options pour le temps réel sont regroupées dans la partie “Xenomai/cobalt”
  - Certaines options standard sont incompatibles avec Xenomai
    - CPU frequency scaling
    - Power management
- Compiler le noyau

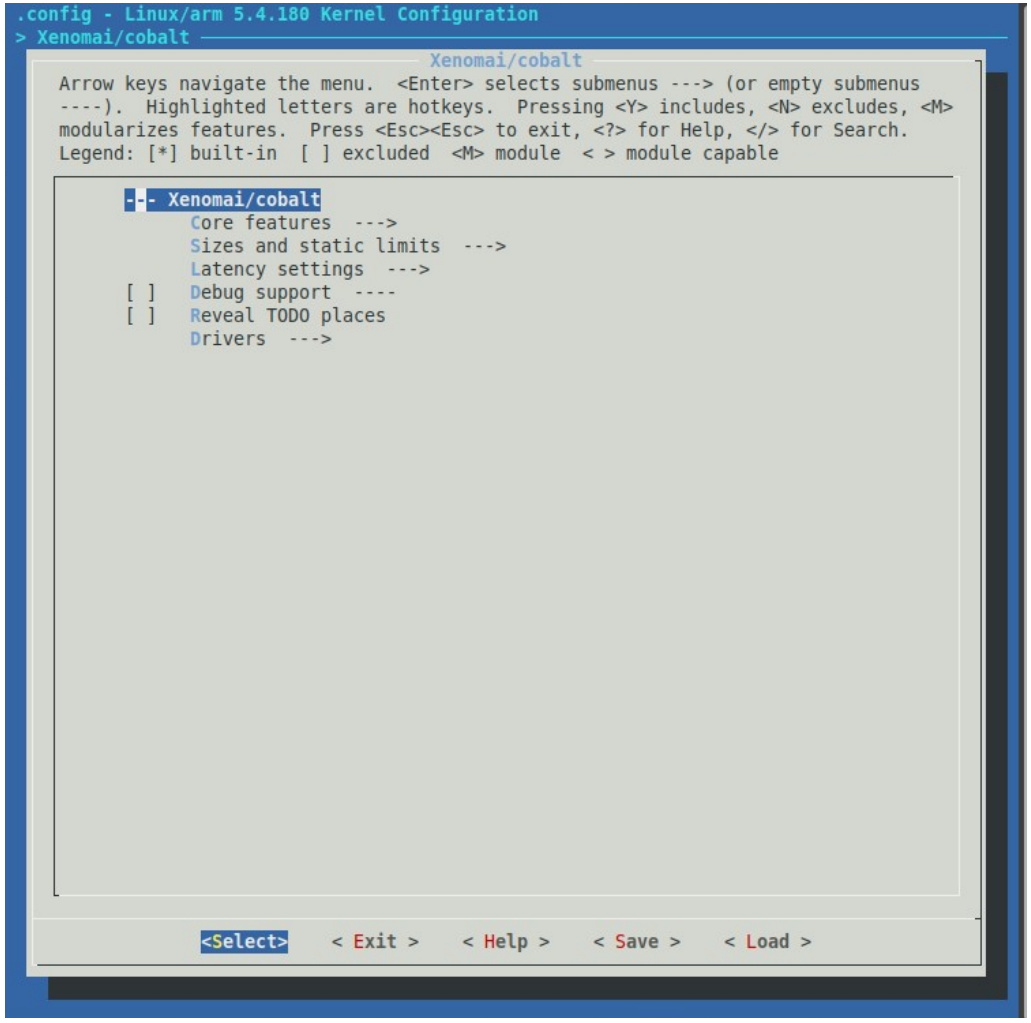


Attention, analyse préalable de l'intérêt d'un OS temps réel !

4. Installation de Xenomai Cobalt

Compilation du noyau Linux





4. Installation de Xenomai Cobalt

Compilation des librairies

- Création du script de configuration et du Makefile
  - `./scripts/bootstrap`
- Option de configuration disponibles
  - `./configure --help`

- Exemple
  - `./configure --build=i686-pc-linux-gnu --host=arm-linux --with-core=cobalt`
  - Une chaîne de compilation croisée doit être disponible

4. Installation de Xenomai Cobalt

Installation des librairies

- Installation
  - `make DESTDIR=<répertoire de travail> install`
- Répertoire de travail
  - Le répertoire de travail est le répertoire qui contient le système de fichier de la cible en cours de conception
  - Peut correspondre au répertoire /tftpboot/rootfs

4. Installation de Xenomai Cobalt

Compilation assistée?

- Compilation/installation faisable avec Yocto!
- Nécessite les recettes Xenomai pour Bitbake
- Recettes intégrées dans *meta-mi11* (cf TP1 Linux embarqué)
  - Noyau et système de fichiers pour les TP Xenomai faits avec Yocto
  - Machine “joypinote-xenomai”

4. Installation de Xenomai Cobalt

Test

- `/usr/bin/latency`
  - à lancer sur la cible
  - mesure le temps s'écoulant entre le déclenchement d'un timer matériel et l'activation de la routine de traitement
  - pour charger le système on peut utiliser
    - `dd if=/dev/zero of=/dev/null`
    - Commande linux : `stress`

```
root@joypinote-xenomai:~# latency -T 10
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      7.221|      7.479|      9.925|       0|     0|      7.221|      9.925
RTD|      5.795|      7.471|     13.555|       0|     0|      5.795|     13.555
RTD|      7.350|      7.509|     12.239|       0|     0|      5.795|     13.555
RTD|      7.220|      7.545|     11.887|       0|     0|      5.795|     13.555
RTD|      7.275|      7.542|     12.071|       0|     0|      5.795|     13.555
RTD|      7.219|      7.490|     11.626|       0|     0|      5.795|     13.555
RTD|      7.440|      7.564|     11.922|       0|     0|      5.795|     13.555
RTD|      7.199|      7.551|     11.477|       0|     0|      5.795|     13.555
RTD|      7.346|      7.499|     12.773|       0|     0|      5.795|     13.555
---|-----------|-----------|-----------|--------|------|-------------------------
RTS|      5.795|      7.516|     13.555|       0|     0|    00:00:10/00:00:10
```

```
root@joypinote-xenomai:~# stress -c 100 -t 50 -q &
[2] 492
root@joypinote-xenomai:~# latency -T 10
== Sampling period: 1000 us
== Test mode: periodic user-mode task
== All results in microseconds
warming up...
RTT|  00:00:01  (periodic user-mode task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|      7.518|      8.041|     13.703|       0|     0|      7.518|     13.703
RTD|      7.591|      8.344|     14.332|       0|     0|      7.518|     14.332
RTD|      7.572|      8.326|     15.091|       0|     0|      7.518|     15.091
RTD|      7.460|      8.106|     15.498|       0|     0|      7.460|     15.498
RTD|      7.552|      8.044|     13.571|       0|     0|      7.460|     15.498
RTD|      7.533|      8.143|     13.256|       0|     0|      7.460|     15.498
RTD|      7.477|      8.100|     13.866|       0|     0|      7.460|     15.498
RTD|      7.662|      8.105|     14.625|       0|     0|      7.460|     15.498
RTD|      7.532|      8.127|     14.754|       0|     0|      7.460|     15.498
---|-----------|-----------|-----------|--------|------|-------------------------
RTS|      7.460|      8.148|     15.498|       0|     0|    00:00:10/00:00:10
```

4. Installation de Xenomai Cobalt

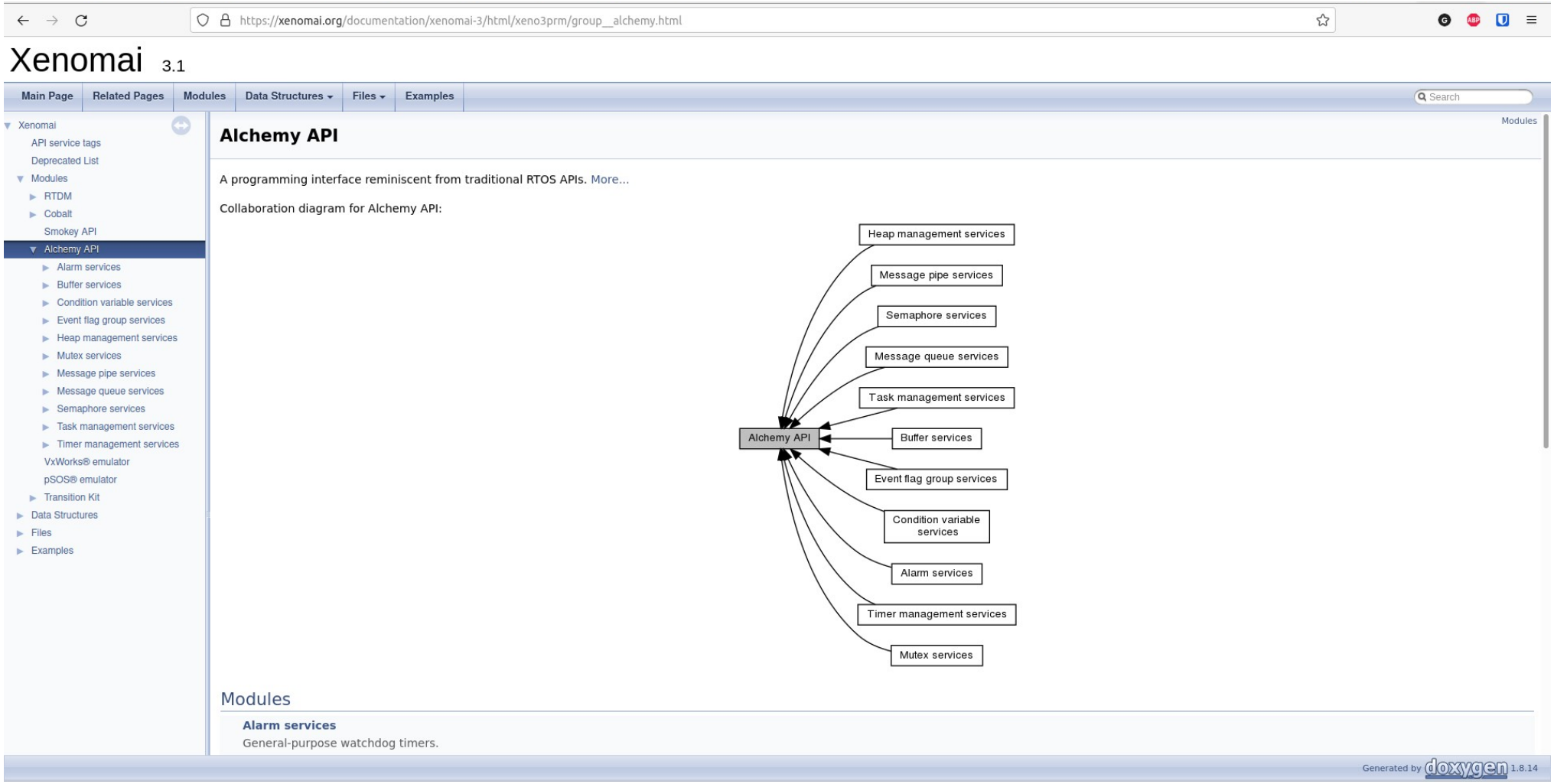
Dossier /proc/xenomai

- `heap`: statistiques d'utilisation de la mémoire par Xenomai et par les objets de l'API (sémaphores, piles, ...)
- `irq`: liste des interruptions gérées par Xenomai
- `latency`: anticipation sur les déclenchements timers
- `sched/stat et sched/threads`: liste des tâches temps réel avec leur état (MSW, CSW)
- `timer`: infos sur le timer disponible pour Xenomai
- `version`: n° de version de Xenomai

5. l'API alchemy Xenomai

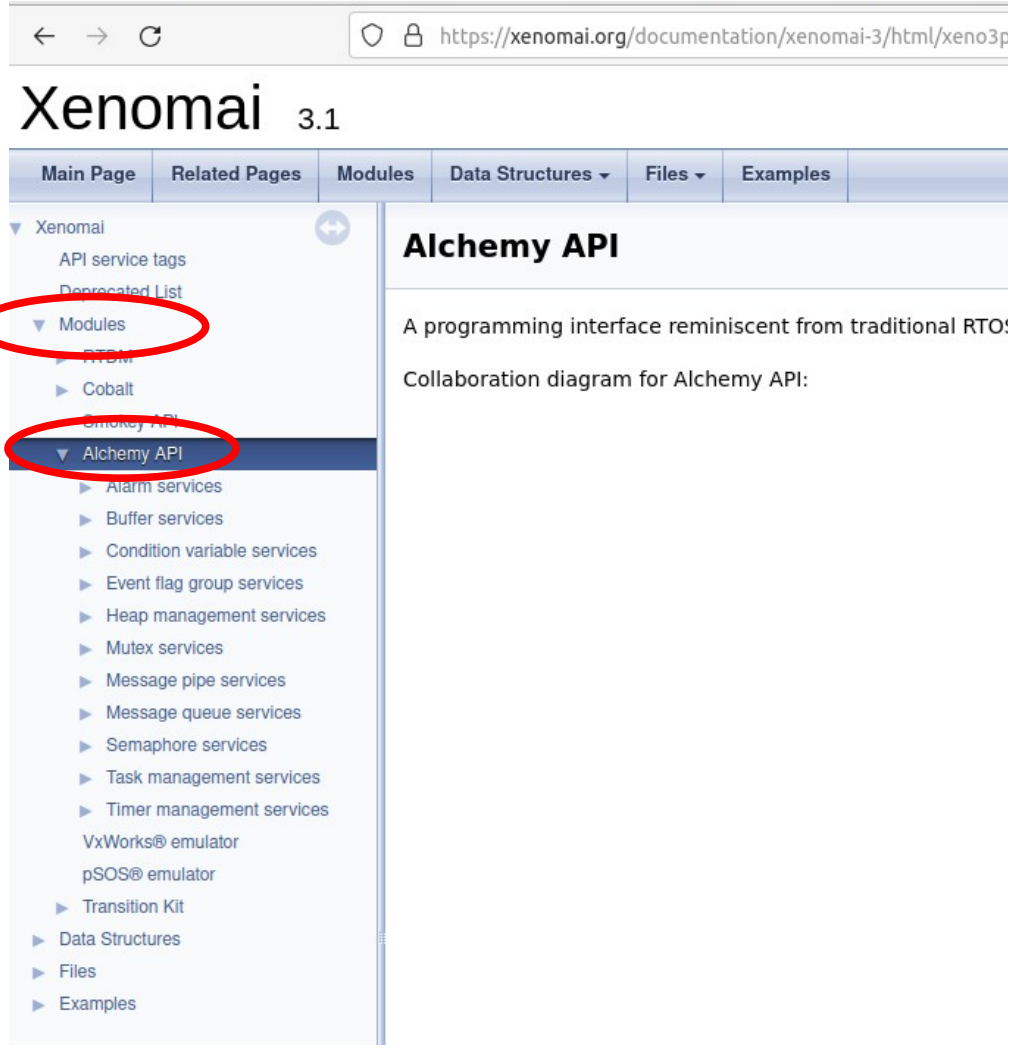
5. l'API alchemy Xenomai

https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__alchemy.html



5. l'API alchemy Xenomai

https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__alchemy.html



Sélectionner:

- Modules
- Alchemy API

A lire pour les TPs !!!

5. l'API alchemy Xenomai

Task management services (alchemy/task.h)

- rt_task_create: Create a new real-time task
- rt_task_start: Start a real-time task
- rt_task_suspend: Suspend a real-time task
- rt_task_resume: Resume a real-time task
- rt_task_delete: Delete a real-time task
- rt_task_yield: Manual round-robin.
- rt_task_set_periodic: Make a real-time task periodic
- rt_task_wait_period: Wait for the next periodic release point
- rt_task_set_priority: Change the base priority of a real-time task
- rt_task_sleep: Delay the calling task (relative)
- rt_task_sleep_until: Delay the calling task (absolute)
- rt_task_unblock: Unblock a real-time task
- rt_task_inquire: Inquire about a real-time task

5. l'API alchemy Xenomai

Task management services (alchemy/task.h)

- rt_task_set_mode: Change task mode bits
- rt_task_self: Retrieve the current task
- rt_task_slice: Set a task's round-robin quantum
- rt_task_send: Send a message to a task
- rt_task_receive: Receive a message from a task
- rt_task_reply: Reply to a task
- rt_task_spawn: Spawn a new real-time task
- rt_task_shadow: Turns the current Linux task into a native Xenomai task
- rt_task_bind: Bind to a real-time task.
- rt_task_unbind: Unbind from a real-time task
- rt_task_join: Wait on the termination of a real-time task
- rt_task_same: Compare two task descriptors

Task management services (alchemy/task.h)

int rt_task_create (RT_TASK *task,const char *name,int stksize,int prio,int mode)

Paramètres :

- task: The address of a task descriptor Xenomai will use to store the task-related data.
- name: An ASCII string standing for the symbolic name of the task.
- stksize: The size of the stack (in bytes) for the new task. If zero is passed, a reasonable pre-defined size will be substituted.
- prio: The base priority of the new task. This value must range from [0 .. 99] (inclusive) where 0 is the lowest effective priority. *Priorité 0 n'est pas temps réel*
- mode: The task creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new task:

  T_LOCK , T_WARNSW ,

  T_JOINABLE : allows another task to wait on the termination of the new task.

Task management services (alchemy/task.h)

int rt_task_start (RT_TASK *task,void(*)(void *cookie) entry,void *arg)

Paramètres :

- task: The descriptor address of the affected task which must have been previously created by the rt_task_create() service.
- entry: The address of the task's body routine. In other words, it is the task entry point.
- arg: A user-defined opaque argument entry will receive.

int rt_task_join (RT_TASK *task)

Paramètre :

- task: The address of a task descriptor to join.

Task management services (alchemy/task.h)

int rt_task_sleep (RTIME delay)

Paramètre :

- delay: The number of nano seconds to wait before resuming the task. Passing zero causes the task to return immediately with no delay.

int rt_task_delete (RT_TASK *task)

Paramètre :

- task: The descriptor address of the affected task.

5. l'API alchemy Xenomai

Semaphore services (alchemy/sem.h)

- rt_sem_create: Create a counting semaphore.
- rt_sem_delete: Delete a semaphore.
- rt_sem_p: Pend on a semaphore.
- rt_sem_p_until: Pend on a semaphore (with absolute timeout date).
- rt_sem_v: Signal a semaphore.
- rt_sem_broadcast: Broadcast a semaphore.
- rt_sem_inquire: Inquire about a semaphore.
- rt_sem_bind: Bind to a semaphore.
- rt_sem_unbind: Unbind from a semaphore.

Semaphore services (alchemy/sem.h)

int rt_sem_create(RT_SEM *sem,const char *name,unsigned long icount,int mode)

Paramètres :

- sem: The address of a semaphore descriptor Xenomai will use to store the semaphore-related data.
- name: An ASCII string standing for the symbolic name of the semaphore.
- icount:The initial value of the semaphore count.
- mode: The semaphore creation mode. The following flags can be OR'ed into this bitmask, each of them affecting the new semaphore:

  S_FIFO makes tasks pend in FIFO order on the semaphore.

  S_PRIO makes tasks pend in priority order on the semaphore.

Semaphore services (alchemy/sem.h)

int rt_sem_p(RT_SEM *sem,RTIME timeout)

Paramètres :

- sem: The descriptor address of the affected semaphore.
- Timeout: The number of nanoseconds to wait for a semaphore unit to be available. Passing TM_INFINITE causes the caller to block indefinitely until a unit is available. Passing TM_NONBLOCK causes the service to return immediately without waiting if no unit is available.

int rt_sem_v(RT_SEM *sem)

Paramètre :

- sem: The descriptor address of the affected semaphore.

Semaphore services (alchemy/sem.h)

int rt_sem_delete(RT_SEM *sem)

Paramètre :

- sem: The descriptor address of the affected semaphore.

5. l'API alchemy Xenomai

Mutex services (alchemy/mutex.h)

- rt_mutex_create: Create a mutex.
- rt_mutex_delete: Delete a mutex
- rt_mutex_acquire: Acquire a mutex.
- rt_mutex_acquire_until: Acquire a mutex (with absolute timeout date).
- rt_mutex_release: Unlock mutex.
- rt_mutex_inquire: Inquire about a mutex.
- rt_mutex_bind: Bind to a mutex.
- rt_mutex_unbind: Unbind from a mutex.

Mutex services (alchemy/mutex.h)

int rt_mutex_create(RT_MUTEX *mutex,const char *name)

Paramètres :

- mutex: The address of a mutex descriptor Xenomai will use to store the mutex-related data.
- name: An ASCII string standing for the symbolic name of the mutex.

int rt_mutex_delete(RT_MUTEX *mutex)

Paramètre :

- mutex: The descriptor address of the affected mutex.

Mutex services (alchemy/mutex.h)

int rt_mutex_acquire(RT_MUTEX *mutex,,RTIME timeout)

Paramètres :

- mutex: The descriptor address of the mutex to acquire.
- timeout: The number of nanoseconds to wait for the mutex to be available to the calling task. Passing TM_INFINITE causes the caller to block indefinitely until the mutex is available. Passing TM_NONBLOCK causes the service to return immediately without waiting if the mutex is still locked by another task.

int rt_mutex_release(RT_MUTEX *mutex)

Paramètre :

- mutex: The descriptor address of the released mutex.

6. Exemples avec l'API alchemy Xenomai

6. Exemples avec l'API alchemy Xenomai

Makefile et xeno-config

```
XENO_CONFIG := /path-to/xeno-config
CFLAGS := $(shell $(XENO_CONFIG) --alchemy --cflags)
LDFLAGS := $(shell $(XENO_CONFIG) --alchemy --ldflags)
CC := $(shell $(XENO_CONFIG) --cc)

EXECUTABLE := rt_program

all: $(EXECUTABLE)

%: %.c
   $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS)
```

6. Exemples avec l'API alchemy Xenomai

Création de thread

```
#include <stdio.h>
#include <alchemy/task.h>

#define TASK_PRIO 99 //highest RT priority
#define TASK_MODE 0 //no flags
#define TASK_STKSZ 0 //use default one

void task_body() {
   while(1) {};//appels systèmes Linux
}

int main(int argc, char *argv[]) {
   int err;
   RT_TASK task_desc;

   err=rt_task_create(&task_desc,"task",TASK_STKSZ,TASK_PRIO,TASK_MODE);
   if(err!=0) {
      printf("error rt_task_create\n");
      return EXIT_FAILURE;
   }

   rt_task_start(&task_desc,&task_body,NULL);

   rt_task_delete(&task_desc);
}
```

6. Exemples avec l'API alchemy Xenomai

Analyse système

- Fichier `/proc/xenomai/sched/stat`
- MSW : nombre de changements de modes
- CSW : nombre de changements de contextes

```
root@joypinote-xenomai:~# cat /proc/xenomai/sched/stat
CPU  PID    MSW        CSW        XSC        PF    STAT       %CPU  NAME
  0  0      0          224        0          0     00018000  100.0  [ROOT]
  0  346    8          10         54         0     000680c0    0.0  main
  0  348    5          10         14         0     00048044    0.0  task
  0  0      0          4856       0          0     00000000    0.0  [IRQ18:[timer]]
root@joypinote-xenomai:~# cat /proc/xenomai/sched/stat
CPU  PID    MSW        CSW        XSC        PF    STAT       %CPU  NAME
  0  0      0          230        0          0     00018000  100.0  [ROOT]
  0  346    8          10         54         0     000680c0    0.0  main
  0  348    8          16         17         0     00048044    0.0  task
  0  0      0          4895       0          0     00000000    0.0  [IRQ18: [timer]]
```

6. Exemples avec l'API alchemy Xenomai

Utilisation *correcte* de thread RT

```
#include <stdio.h>
#include <alchemy/task.h>

#define TASK_PRIO 99 //highest RT priority
#define TASK_MODE 0 //no flags
#define TASK_STKSZ 0 //use default one

void task_body() {
   while(1) rt_printf("Hello world!\n");
}

int main(int argc, char *argv[]) {
   int err;
   RT_TASK task_desc;

   err=rt_task_create(&task_desc,"hello",TASK_STKSZ,TASK_PRIO,TASK_MODE);
   if(err!=0) {
      printf("error rt_task_create\n");
      return EXIT_FAILURE;
   }

   rt_task_start(&task_desc,&task_body,NULL);
   rt_task_delete(&task_desc);
}
```

```
root@joypinote-xenomai:~# cat /proc/xenomai/sched/stat
CPU  PID    MSW        CSW        XSC        STAT       %CPU  NAME
  0  0      0          272        0          00018000  100.0  [ROOT]
  0  352    8          10         54         000680c0    0.0  main
  0  354    2          10         17         00048044    0.0  hello
  0  0      0          7530       0          00000000    0.0  [IRQ18]
root@joypinote-xenomai:~# cat /proc/xenomai/sched/stat
CPU  PID    MSW        CSW        XSC        STAT       %CPU  NAME
  0  0      0          276        0          00018000  100.0  [ROOT]
  0  352    8          10         54         000680c0    0.0  main
  0  354    2          14         21         00048044    0.0  hello
  0  0      0          7582       0          00000000    0.0  [IRQ18]
```

6. Exemples avec l'API alchemy Xenomai

Remarques

- Les codes précédents ne sont pas complets!
- *Auto-shadowing* du *main* avec une priorité 0!
- Le *main* est en concurrence avec les autres threads (RT ou non)
- A la fin du *main* le programme s'arrête!!!

6. Exemples avec l'API alchemy Xenomai

Exemple multi tasks

```
#include <alchemy/task.h>

#define TASK_PRIO 99 //highest RT priority
#define TASK_MODE 0 //no flags
#define TASK_STKSZ 0 //use default one

void task_body_1() {
    //real time task 1 stuffs
}

void task_body_2() {
    //real time task 2 stuffs
}

int main(int argc, char *argv[]) {
   RT_TASK task_desc_1,task_desc_2;

   rt_task_create(&task_desc_1,"task_1",TASK_STKSZ,TASK_PRIO,TASK_MODE);
   rt_task_start(&task_desc_1,&task_body_1,NULL);

   rt_task_create(&task_desc_2,"task_2",TASK_STKSZ,TASK_PRIO,TASK_MODE);
   rt_task_start(&task_desc_2,&task_body_2,NULL);

   rt_task_delete(&task_desc_1);
   rt_task_delete(&task_desc_2);
}
```

6. Exemples avec l'API alchemy Xenomai

Exemple multi tasks

```
rt_task_create(&task_desc_1,"task_1",TASK_STKSZ,TASK_PRIO,TASK_MODE);
rt_task_start(&task_desc_1,&task_body_1,NULL);
```

- `main` et `task_body_1` en concurrence
- `task_body_1` se lance (+ haute priorité)
- si `task_body_1` ne rend jamais la main
  - pas de retour au `main`
  - `task_body_2` ne se lancera jamais!
- si `task_body_1` rend la main
  - retour au `main` (pas d'autre thread actif)
- si `task_body_1` change de mode
  - `main` et `task_body_1` en concurrence
  - ordonnanceur donne la main au `main`

6. Exemples avec l'API alchemy Xenomai

Exemple multi tasks

```
rt_task_create(&task_desc_2,"task_2",TASK_STKSZ,TASK_PRIO,TASK_MODE);
rt_task_start(&task_desc_2,&task_body_2,NULL);
```

- `main, task_body_1` et `task_body_2` en concurrence
  - on suppose `task_body_1` en attente
- `task_body_2` se lance (+ haute priorité)
- si `task_body_2` ne rend jamais la main
  - pas de retour au `main`
  - `task_body_1` ne continuera jamais!
- si `task_body_2` rend la main
  - reprise de `task_body_1` si attente finie
  - sinon retour au `main`
- on suppose retour au `main`

6. Exemples avec l'API alchemy Xenomai

Exemple multi tasks

```
rt_task_delete(&task_desc_1);
rt_task_delete(&task_desc_2);
```

- destruction des 2 tâches même si elles ne sont pas *terminées*
- fin du programme!

- il peut être souhaitable de bloquer le `main`
  - `getc()`
  - `getchar()`
  - `rt_task_join()`
  - etc

7. Développement de drivers avec RTDM

7. Développement de drivers avec RTDM

Rappels drivers et interruptions

- Pour interagir avec les périphériques, il est nécessaire de développer des drivers
  - Fichiers .ko
  - Sous linux, les périphériques sont accessibles via le dossier des fichiers spéciaux `/dev`
- Lors du fonctionnement normal d'un processus, une tâche est dans un état suspendu ou actif.
- Le réveil d'une tâche se fait via une interruption :
  - Logicielle : déclenchement d'un timer, libération d'un objet de synchronisation
  - Matérielle : donnée entrante sur un périphérique

7. Développement de drivers avec RTDM

Périphériques RTDM

- Les périphériques gérés par RTDM sont de deux types :
  - Périphériques nommés : accessibles par les appels `open()`, `close()`, `read()`, `write()`, `ioctl()`, …
  - Périphériques protocoles : accessibles par les appels `socket()`, `close()`, `sendto()`, `recvfrom()`, …
- Très similaire aux fonctions classiques de gestion des périphériques Linux

Exemple code module RTDM

```
#include <linux/module.h>
#include <rtdm/driver.h>

static ssize_t foo_driver_read_rt(struct rtdm_fd *fd,void __user *buf,
size_t len) {
        rtdm_printk("foo_driver_read_rt %i\n",len);
        return len;
}

static ssize_t foo_driver_write_rt(struct rtdm_fd *fd,const void
__user *buf, size_t len) {
        rtdm_printk("foo_driver_write_rt %i\n",len);
        return len;
}

int foo_driver_open(struct rtdm_fd *fd, int oflags) {
        rtdm_printk("foo_driver_open\n");
        return 0;
}

static void foo_driver_close(struct rtdm_fd *fd) {
        rtdm_printk("foo_driver_close\n");
}

static void __exit foo_driver_exit(void) {
        rtdm_printk("foo_driver_exit\n");
        rtdm_dev_unregister(&foo_device);
}
module_exit(foo_driver_exit);
```

```
static struct rtdm_driver foo_driver = {
        .profile_info=   RTDM_PROFILE_INFO(foo, //name
                          RTDM_CLASS_EXPERIMENTAL, //major
                          1, //minor
                          1), //version
        .device_flags=   RTDM_NAMED_DEVICE|,
        .device_count=   1,
        .ops = {
                .open=       foo_driver_open,
                .close=      foo_driver_close,
                .read_rt=    foo_driver_read_rt,
                .write_rt=   foo_driver_write_rt,
        },
};

static struct rtdm_device foo_device = {
        .driver = &foo_driver,
        .label = "foo",
};

static int __init foo_driver_init(void) {
        if (!rtdm_available()) return -ENOSYS;

        rtdm_printk("foo_driver_init\n");
        rtdm_dev_register(&foo_device);
        return 0;
}
module_init(foo_driver_init);

MODULE_LICENSE("GPL");
```

Exemple Makefile module RTDM

```
### List of modules to be build
MODULES =squelette_module_rtdm

### Note: to set the kernel source path, use "make KSRC=..."
KSRC ?= /usr/src/linux

ifneq ($(MODULES),)

OBJS     := ${patsubst %, %.o, $(MODULES)}
CLEANMOD := ${patsubst %, .%*, $(MODULES)}
PWD      := $(shell if [ "$$PWD" != "" ]; then echo $$PWD; else pwd; fi)


obj-m       := $(OBJS)
EXTRA_CFLAGS := -I$(KSRC)/include/xenomai $(ADD_CFLAGS)

all::
	$(MAKE) -C $(KSRC) SUBDIRS=$(PWD) modules


modules:
	@echo "$(CFLAGS)"

clean::
	$(RM) $(CLEANMOD) *.o *.ko *.mod.c Module*.symvers Module.markers modules.order
	$(RM) -R .tmp*

endif
```

Exemple code userspace RTDM

- Les périphériques gérés par RTDM sont visibles dans `/dev/rtdm/`
- Code réalisant la commande `cat` sur un device RTDM

```
#include <alchemy/task.h>

#define LG_BUFFER 80

void task() {
    int i,fd;
    char buffer[LG_BUFFER];

    fd = open("/dev/rtdm/foo", O_RDONLY);

    if (fd < 0) {
        rt_printf("cannot open device\n");
        return;
    }

    while((i=read(fd,buffer,LG_BUFFER)) >0) {
        buffer[i]='\0';
        rt_printf("%s\n",buffer);
    }

    close(fd);
}

int main(int argc, char* argv[]) {
…
}
```

7. Développement de drivers avec RTDM

Exemple Makefile userspace RTDM

```
XENO_CONFIG := /path-to/xeno-config
CFLAGS := $(shell $(XENO_CONFIG) --alchemy --rtdm --cflags)
LDFLAGS := $(shell $(XENO_CONFIG) --alchemy --rtdm --ldflags)
CC := $(shell $(XENO_CONFIG) --cc)

EXECUTABLE := rt_program

all: $(EXECUTABLE)

%: %.c
   $(CC) -o $@ $< $(CFLAGS) $(LDFLAGS)
```

- ajouter `--rtdm` aux flags de compilation et de link
- les appels `open(), close(), read(), write(), ioctl()` sont interceptés et gérés par libcobalt, pour assurer le temps réel

Questions ?