Formation à Linux Embarqué

Jérôme Pouiller <j.pouiller@sysmic.org>

Cinquième partie V

Temps réel

Contexte

Garantie qu'une action est éxecutée en moins d'un temps donné.

Implique :

- Une garantie de qualité (0 bug)
- Souvent des processus garantissant la qualité dans les différentes phases : conception, développement, validation
- Une connaissances importante de l'environnement extérieur (fréquence maximum des interruption, etc...)

→ On se limitera à l'architecture logicielle

Contexte

Ordonnancement statique :

- Charge des taches temps réelles $\ll 100\%$

$\rightarrow$ Ordonnancement avec des priorités statiques suffisante

$\rightarrow$ Problématique des algorithmes d'ordonnancement à priorité dynamique (EDF, LST, etc...) secondaire

Contexte

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

Pour les allergiques : le temps de réponse d'une tâche est égal à la somme du temps d'exécution de la tache et de toutes les tâche de priorité supérieure qui s'exécutent en même temps.

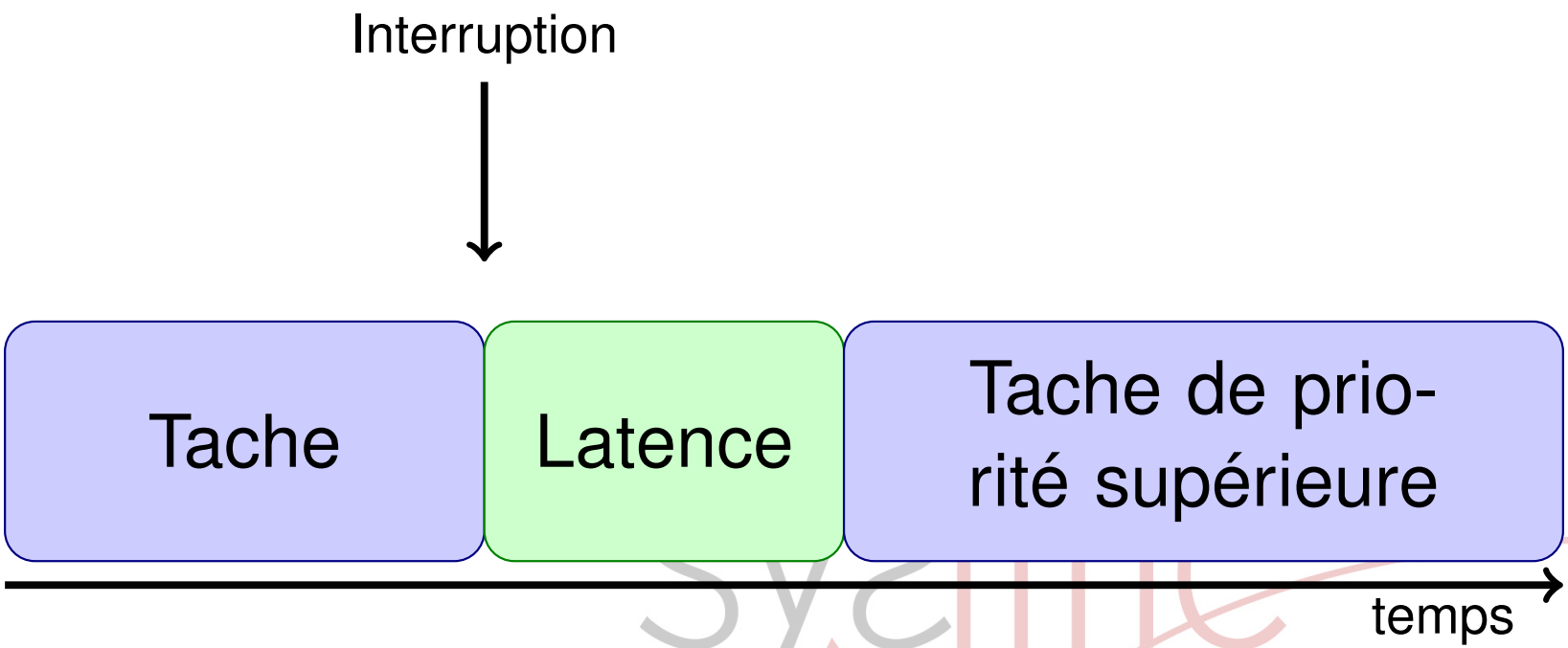
Objectif : Rendre cette formule assez simple pour être prédictible.

Pas si simple :

- Interruptions
- Changement de contexte
- Sections critiques (ordonnanceur désactivé)
- Caches, Swap

Latence aux évènements

- Coeur du problème
- Si la latence était nul (ou au moins constante), on calculerait simplement le temps de réponses de nos tâches.

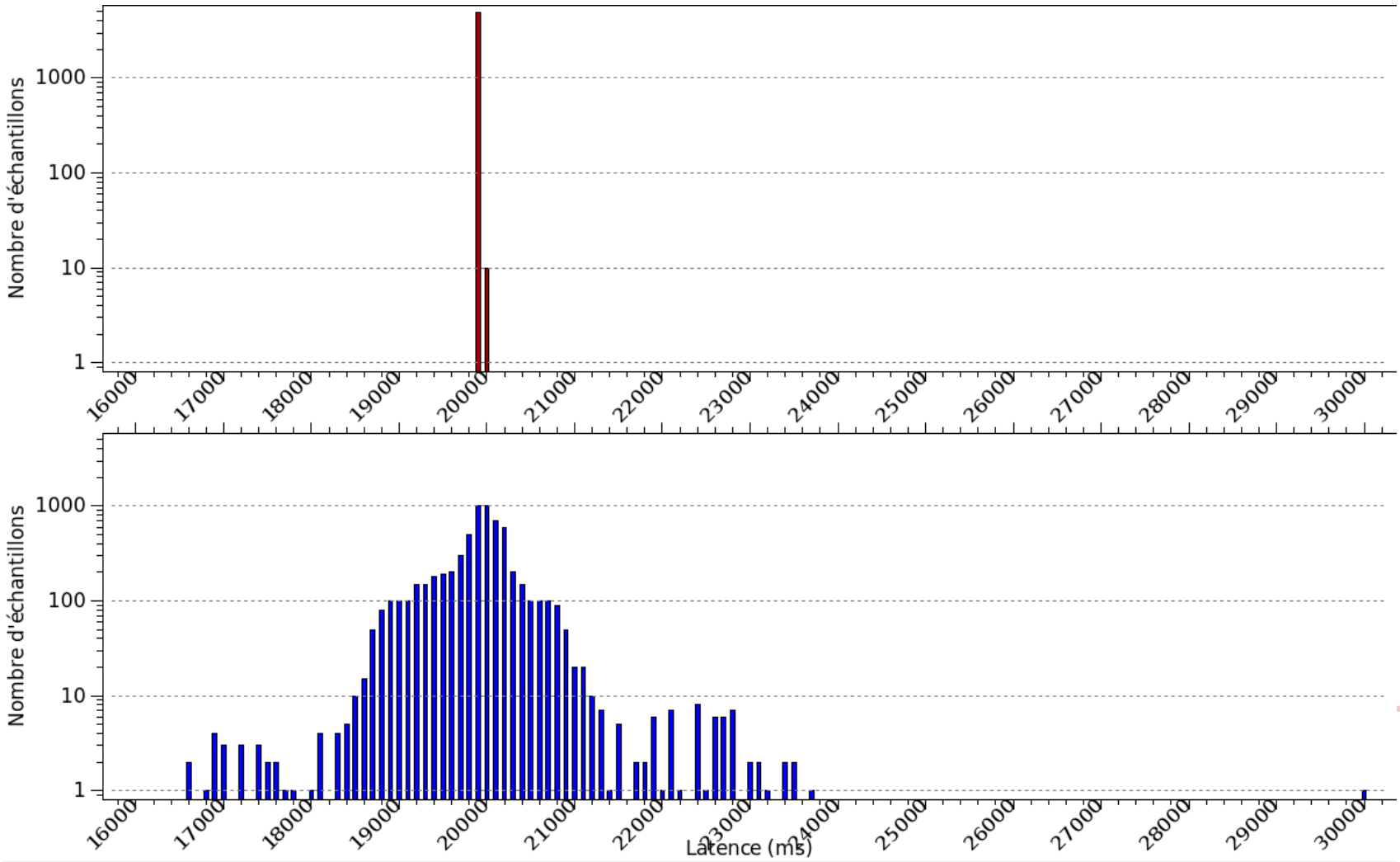


Latence aux évènements
Exemple concret

- On paramètre un timer à 50Hz
- On mesure le temps effectif de chaque période

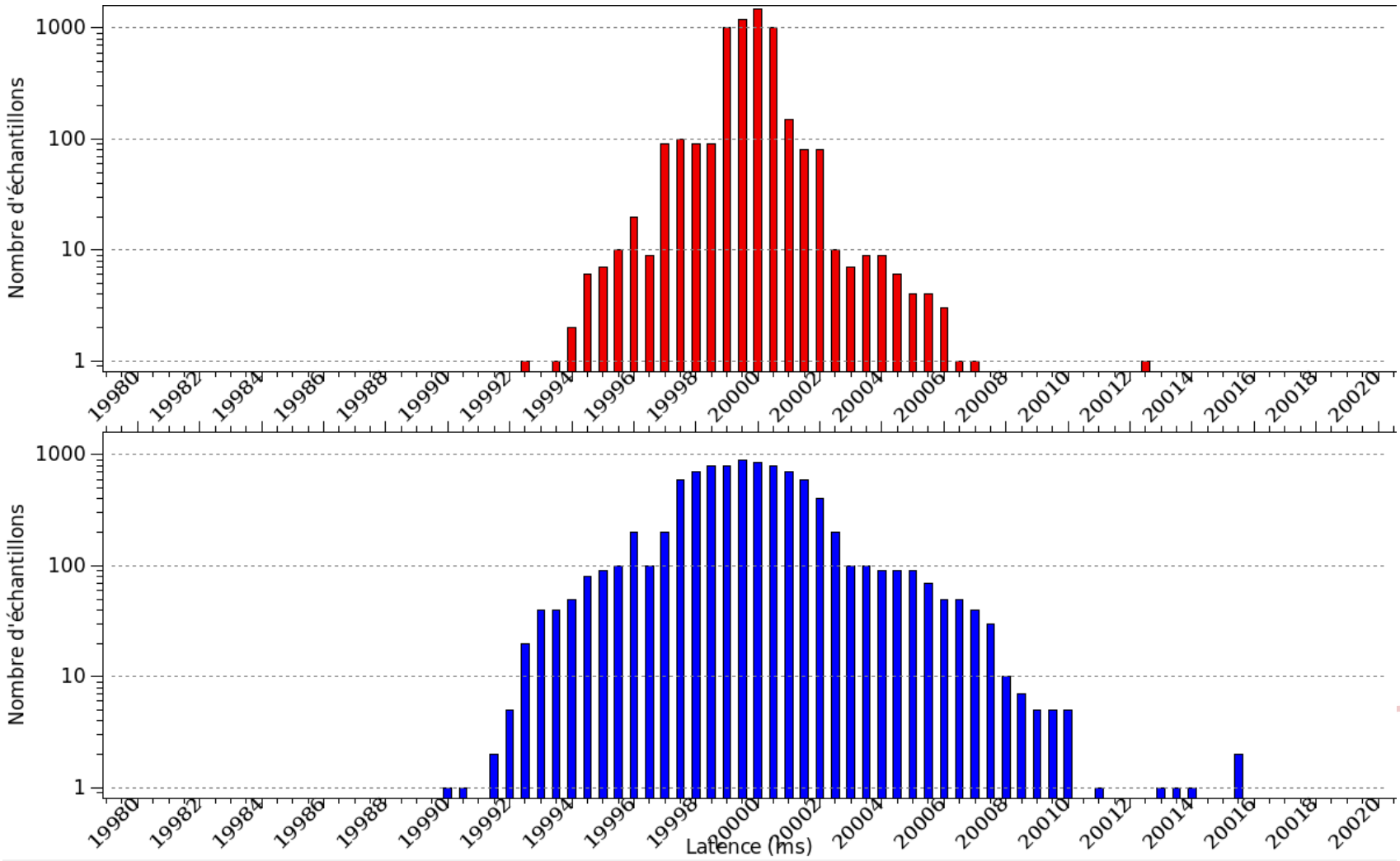
Latence aux évènements

Système classique

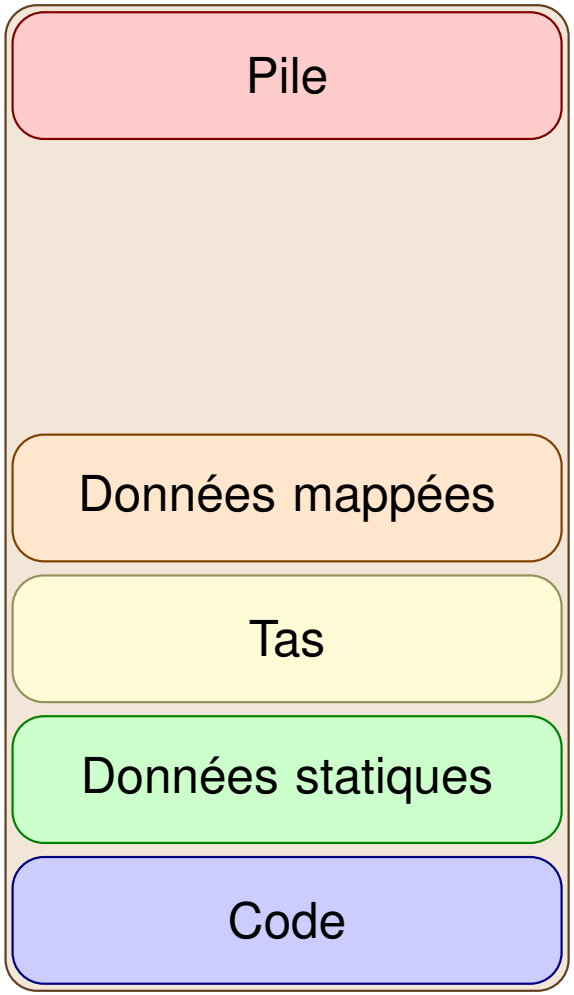


Latence aux évènements

Système temps réels



Tâches Posix
Processus



- Utilisation de la MMU

→ Permet de cloisonnement des processus (*Multitâche*)

→ Permet de partage de pages (r–/rw-/r-x)

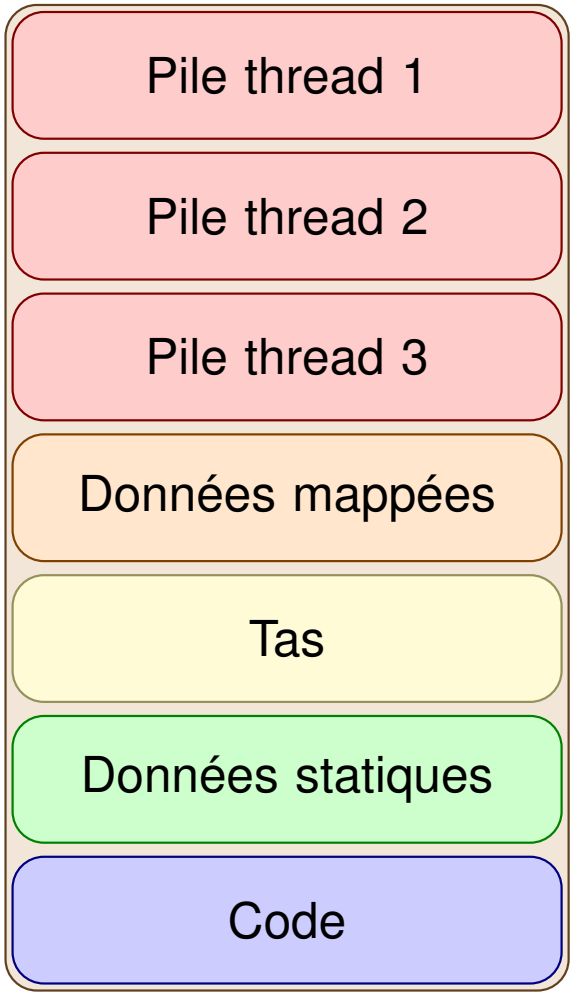
- Communication inter-processus (IPC) relativement complexe

→ Noyau

→ Pages partagées il faut mapper des morceau de memoire dans deux processus) ou passer par le noyau comme arbitre

- Le noyau possède un contexte
- Le noyau est le seul à pouvoir modifier la MMU

Tâches Posix
Threads



- Même contextes

→ Partage de la mémoire

→ Échange de données facilité

→ Posix offre pas mal de services

→ Moins de sécurité

- Optimisations possible pour l'ordonnancement de threads

→ Grouper l'ordonnancement des threads pour réduire les changement de contexte

→ Placer les threads d'un même processus sur le même CPU pour réduire les erreurs de cache

Tâches Posix
Interruptions

- Interruption matérielle :
  - Interrompt la tache en cours
  - Gère l'interruption dans le contexte noyau (+/- suivant les architectures et les types d'interruption)
  - Le driver approprié fait son travail
- Appel système (syscall) :
  - Appel à une fonction du noyau avec le contexte du noyau (exemple : envoi d'un packet réseau, ou sleep)
  - Techniquement, on déclenche une interruption logicielle (historiquement, maintenant, ca s'optimise)

Le monotache
Par scrutation

- Une tache
- Difficulté d'extention
- Difficulté d'implémentation si les fréquences de vérifications des capteurs sont différentes
- CPU utilisé de manière sous-optimale
- Latence peu controlée

Le monotâche
Interruptions synchrones

- Une tache
- ... + des interruptions
- Modèle MS-DOS
- Latence dans le cas de traitement dans l'interruption

Le monotâche
Interruptions asynchrones

- Traitement dans la tache principale
- Modèle de la plupart des logiciels des petits microcontrolleurs
- Pas de prioritisation des actions
- Partage d'informations entre les interruption et la tache

→ Sections critiques

→ Inhibition les interruptions

- Latence peu controlée

Evolution du multitache

Multitache non-préemptif

- Capacité d'avoir plusieurs processus en mémoire
- Switch de tache lors des appels aux syscalls ou par interruptions
- Windows 3.1, μcOS-II
- Pas de Round-Robin
- Partage d'informations entre les taches
- Partage d'informations entre les interruption et la tache

→ Sections critiques

→ Inhibition les interruptions

→ ... en local seulement dans le cas des multi-coeurs

→ Nécéssite l'utilisation de spin-lock

Evolution du multitache

Multitache préemptif

- Programmation d'une interruption d'horloge
- 10Hz $\rightarrow$ Permet au système de récuperer la main toute les 100ms
- Permet de faire un ordonnancement préemptif
- Linux jusqu'à 2.4
- Partage d'informations entre les taches
- Partage d'informations entre les interruption et la tache

Noyau low latency

Problème du noyau normal

Un seul contexte noyau pour tout le monde

- Pas possible de préempter le système
- Personne ne peut prendre la main lorsqu'un processus est dans un syscall
- Équivalent d'une ressource partagée par tout les processus
- Possible de créer une gigantesque inversion de priorité

Noyau low latency
Implémentation

- Difficile de gérer les différents contextes noyau

→ Noyau réentrant (= thread-safe)

- Gestion des interruption assez complexe
- Overhead assez important

Noyau low latency

Gestion des interruptions

- Partage de donnée dans le noyau : mutex
- Que ce passe-t-il si on met un mutex dans une interruption ?

→ Ca se passe mal

→ Il faut désactiver les interruption

→ Doit être de courte durée !

- Et en multicore ?

→ Il faut garder un mutex

- Mutex = réordonnancement

→ Inadapté dans notre cas

→ Utilisation de spin_lock (Attente active)

- Désactivation des interruption implique une latence

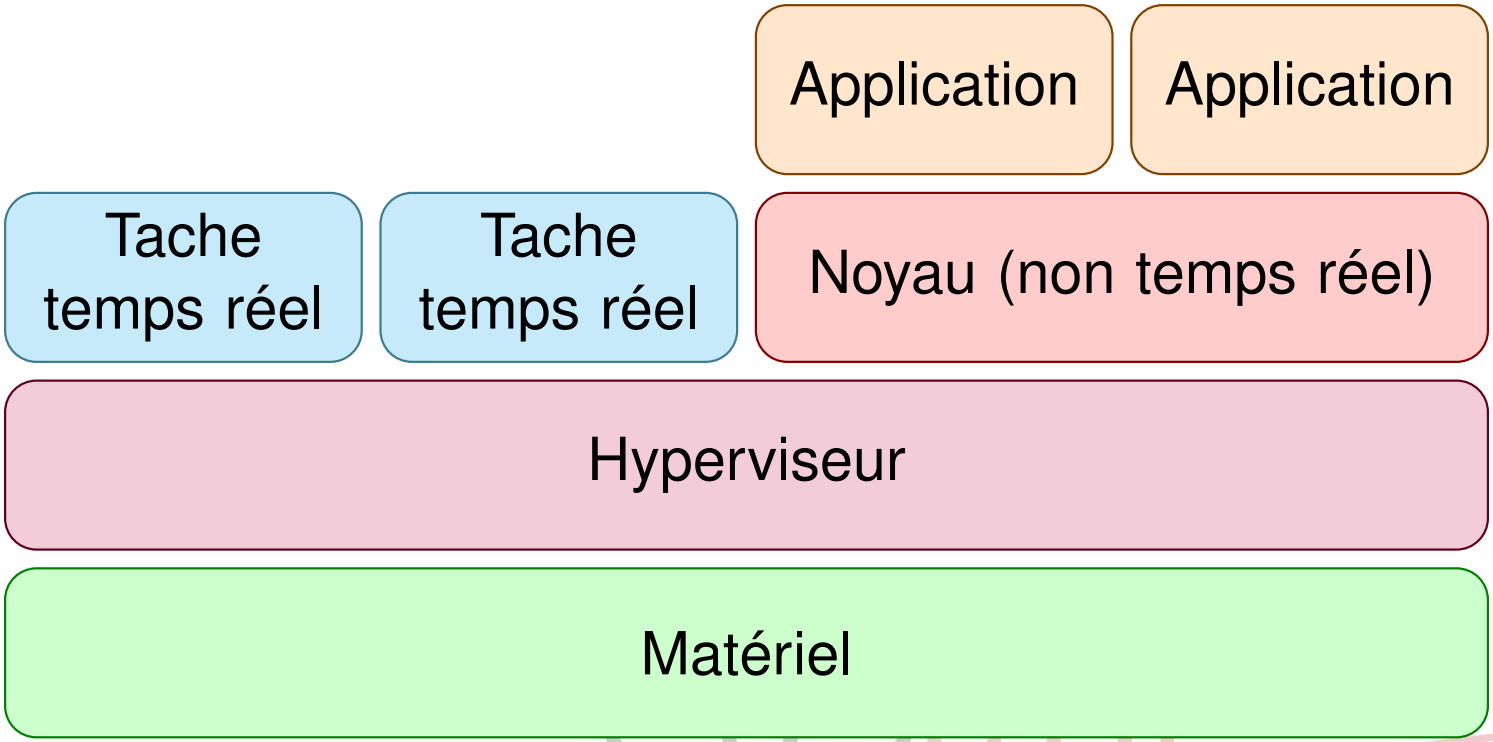
Noyau low latency
Résultats

- Patch low-latency mergé dans le mainstream avec le noyau 2.6 (`CONFIG_PREEMPT`)
- Latences maximum de l'ordre de 300μs (chiffres de l'époque)

Hyperviseur

- Noyau low latency encore problèmatique pour les interruptions
- Beaucoup d'interruptions $\rightarrow$ latence
- Hyperviseur

Temps réel
Nano Kernel



Hyperviseur

- Possibilité de préempter le noyau sans patch low-latency
- Possibilité de differer les interruptions
- Possibilité d'ignorer les interruptions dans les sections temps réelles
- Performances excellentes ($< 20\mu s$)
- Technique non spécifique à Linux
- Peu de code en mode temps réelles

→ Certifiable

- Fonctionne au dessus du noyau
- Comportement des interruptions à modifier
- Communication entre les tâches temps réelles et le reste

→ Nécessite de patcher le noyau

- RTLinux, RTAI et Adeos (`http://download.gna.org/adeos/patches/`, `git://git.xenomai.org/ipipe-gch.git`) (Xenomai)

Hyperviseur
Adeos

- Fork de RTAI
- API utilisateur assurée par Xenomai
  - Skins
  - → Possibilité de faire fonctionner une application développée pour vxWorks
  - Skin native consistante
  - Beaucoup mieux que Posix (de plus, il existe une skin Posix)
  - `www.xenomai.org/documentation/xenomai-head/html/api`

Patch Adeos
Xenomai

Il est possible de télécharger les patchs sur http://download.gna.org/adeos/patches. Néanmoins, utiliser les patchs distribués avec Xenomai nous garanti la compatibilité entre Xenomai et Adeos. Cela permet de plus d'utiliser certaines facilitées de compilation de Xenomai.

Utiliser une version du noyau non officiellement supporté par Adeos demande un effort de développement assez important.

```
$ wget http://download.gna.org/xenomai/stable/xenomai-2.5.5.tar.bz2
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.33.7.tar.bz2
$ tar xvf xenomai-2.5.5.tar.bz2
$ tar xvf linux-2.6.33.7.tar.bz2
$ mv linux-2.6.33.7 linux-2.6.33.7-ipipe
$ mkdir linux-2.6.33.7-ipipe/build xenomai-2.5.5/build
$ cd xenomai-2.5.5
$ ./scripts/prepare-kernel.sh --linux=/home/user/linux-2.6.33.7-ipipe --arch=arm
$ cd ../linux-2.6.33.7-ipipe
$ make O=build CROSS_COMPILE=arm-linux- ARCH=arm usb-a9260_defconfig
$ make O=build CROSS_COMPILE=arm-linux- ARCH=arm menuconfig
```

RT-preempt

Gestion des interruption dans des threads

- Permet de préempter les interruptions

→ Moins de latence des taches

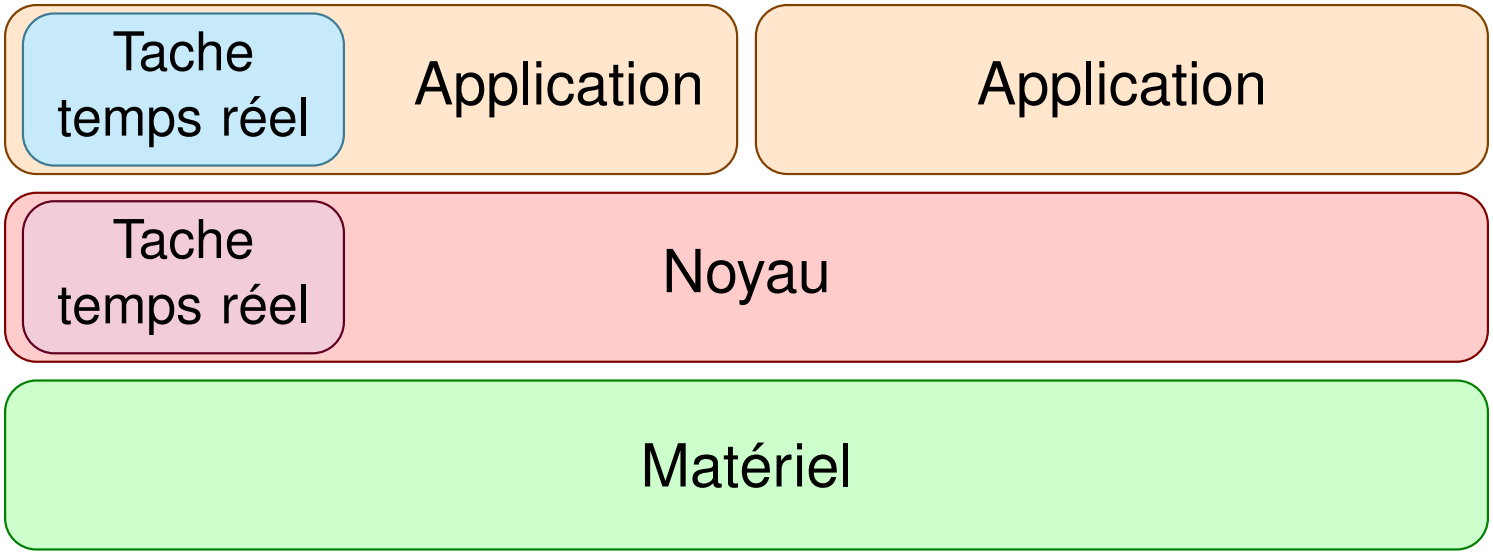
- Permet d'ordonnancer les interruption

→ Permet de se passer de la désactivation des interruptions

→ Permet de remplacer les spin lock par des mutex

→ Moins de latence des interruptions

RT-preempt


Tache
temps réel
Application
Application
Tache
temps réel
Noyau
Matériel

RT-preempt

- Patch RT (Patchs : `http://www.kernel.org/pub/linux/kernel/projects/rt/`, Git : `git://git.kernel.org/pub/scm/linux/kernel/git/rostedt`
- Implémentation assez complexe

→ Non portable

- Performance très bonnes (20μs)

Patch RT-preempt

Fonctionnement :

```
$ wget
http://www.kernel.org/pub/linux/kernel/projects/rt/
    patch-2.6.33.7.2-rt30.bz2
$ wget http://www.kernel.org/pub/linux/kernel/v2.6/
    linux-2.6.33.7.tar.bz2
$ tar xvf linux-2.6.33.7.tar.bz2
$ mv linux-2.6.33.7 linux-2.6.33.7-rt30
$ cd linux-2.6.33.7-rt30
$ bzcat ../patch-2.6.33.7.2-rt30.bz2 | patch -p1
$ mkdir build
$ make O=build CROSS_COMPILE=arm-linux- ARCH=arm
    usb-a9260_defconfig
$ make O=build CROSS_COMPILE=arm-linux- ARCH=arm
    uImage
% cp build/arch/arm/boot/uImage /srv/tftp/uImage
    -2.3.33.7-rt30
```

API

- Attention au swaping : mlockall + allocation de la pile
- Attention aux l'inversions de priorité

Ordonnanceur Linux

- Trois domaines : Other, Fifo, RR
- Ordonnanceur en temps constant (o(1))
- Permet d'ordonnancer efficacement un nombre illimité de processus
- Possible grace à des priorités statiques
- Difficulté d'implémenter les ordonnanceurs à priorité dynamiques
- Rate Monotonic si priorité vaut 1/fréquence de la tache

Ordonnanceur Linux
Priorités dynamiques

- Implémentation du serveur sporadique récent
- Pas d'EDF
- Serveur Sporadique en test

Ordonnanceur Linux
Stabilité

- Pas certifiable !
- Hpet : 2004
- PI : 2006 (pas forcement performante en multi-coeurs)
- Serveur Sporadique : (patch proposé en Aout 2008)
- EDF : En cours de discutions (patch proposé en septembre 2009)