Temps Réel

Jérôme Pouiller <j.pouiller@sysmic.org>

Sixième partie VI

Architectures des OS temps réels

sysmic

Architectures multi-coeurs

Il est possible d'utiliser des architecture multi-coeurs pour des systèmes temps réels.
On distingue alors 2 types d'architectures :

- Les systèmes symétriques (SMP) où les tâches ne sont pas affectées à un CPU particulier
- Les systèmes asymétriques où on affecte manuellement les tâches à un CPU

Systèmes asymétriques

Les systèmes asymétriques sont assez simples à architecturer :

- On effectue une étude séparées pour chaque CPU
- On considère les échanges entre les CPU comme des entrées/sorties

Sous Linux, il est possible d'associer une tâche avec un CPU grâce à :

- la fonction *CPU affinity*
- la commande `taskset`
- aux fonctions *cgroup* et *cpuset*

Systèmes symétriques

- Beaucoup plus complexe à architecturer
- La plupart de nos algorithme ne sont plus prouvé dans une architecture SMP
- La technologie est relativement récente (fin des années 90)
- Pourquoi si tard ?
  - Principalement problèmes matériels
  - Problèmes de barrières mémoires (hard ou soft)
  - Algorithme beaucoup plus ardu (inversion de priorité par exemple)

Limites des système classiques

Les systèmes classiques s'appuient sur un système d'exploitation en général mal adapté pour le temps réel :

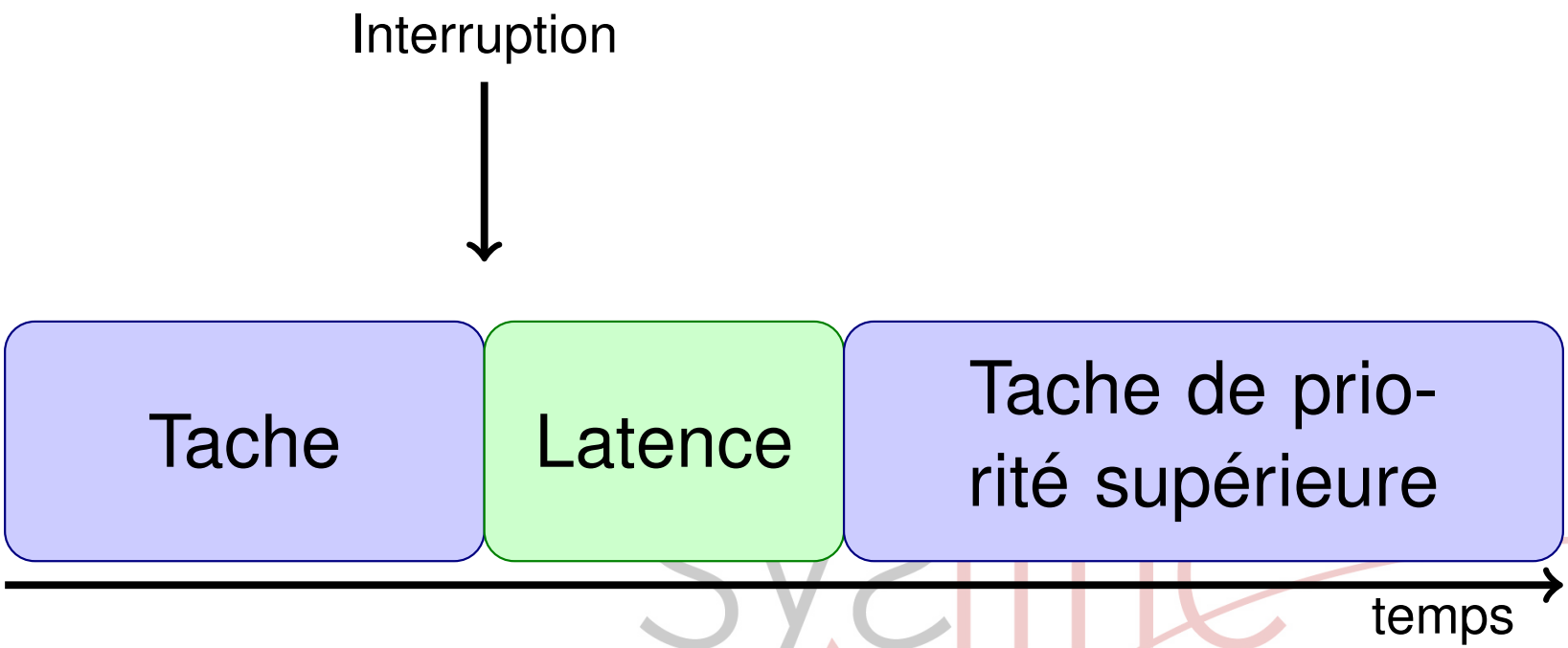
- Politique d'ordonnancement visant à équilibrer équitablement le temps alloué à chaque tâche
- La gestion de la mémoire virtuelle, des caches engendrent des fluctuations temporelles
- La gestion des temporisateurs qui servent à la manipulation des temps pas assez fin
- Mécanismes d'accès aux ressources partagées et de synchronisation comportent des incertitudes temporelles
- Gestion des interruptions non-optimisées
- Systèmes non-certifiés
- API mal adaptées au systèmes temps réels

Mécanisme basiques

- Possibilité d'exécuter une tache avec `SCHED_RT` ou `SCHED_FIFO`
- Possibilité de marquer les pages de mémoire avec `mlock`
- L'accès à des horloges haute précision est un élément clef d'un système temps réel.
  - Le framework *hpet* (*High Precision Timers*) est assez récent dans le noyau Linux (2001) (à titre de comparaison, l'équivalent de *hpet* est apparu sous Windows à partir de Vista).
  - Autrefois, l'horloge était utilisée pour l'ordonnanceur. Le temps retourné par les différents services était celui calculé par l'ordonnanceur. Avec *hpet*, les horloges sont des périphériques à part entière

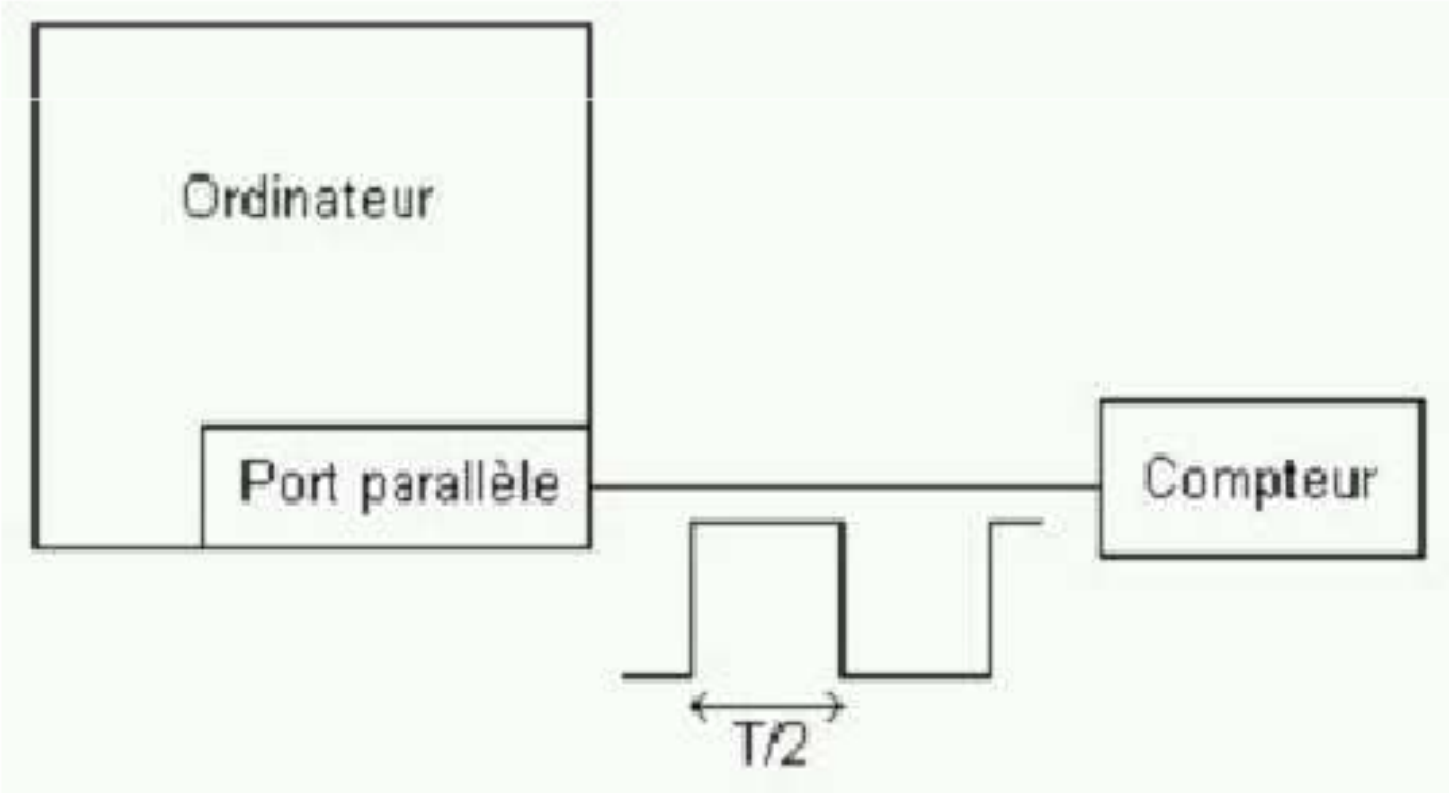
Latence aux évènements

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



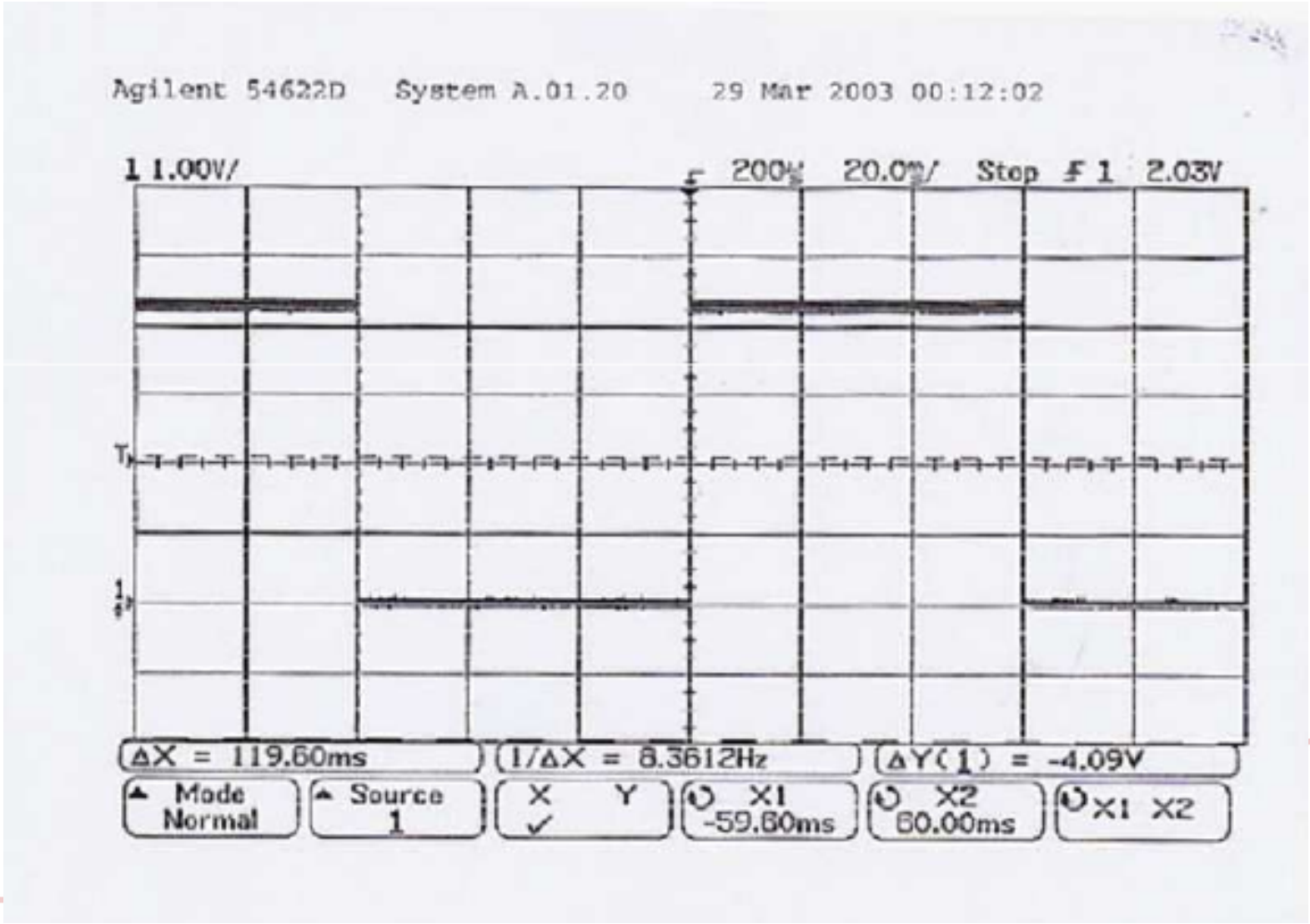
Latence aux évènements
Exemple concret

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



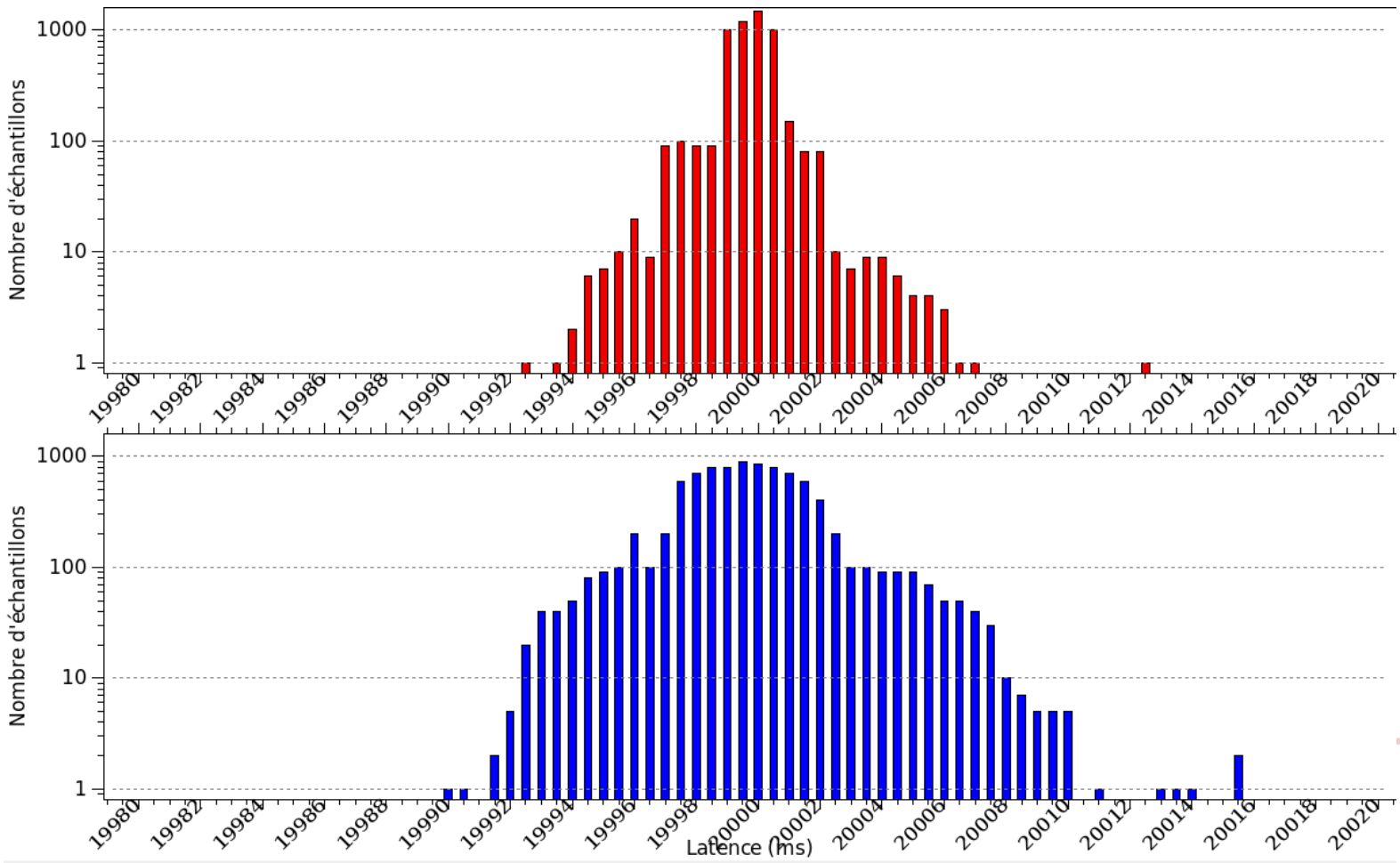
Latence aux évènements

Système temps réel



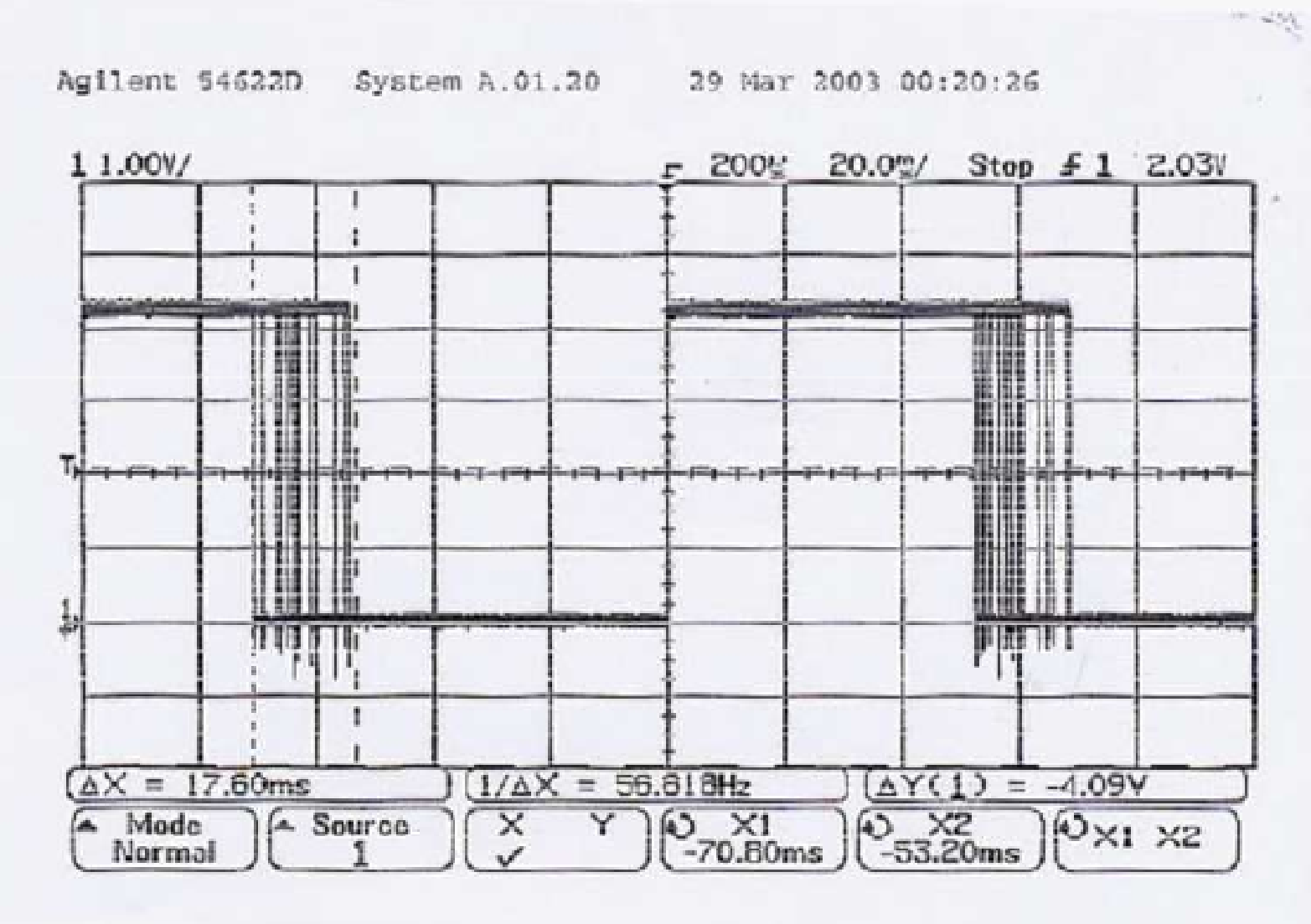
Latence aux évènements

Système temps réels



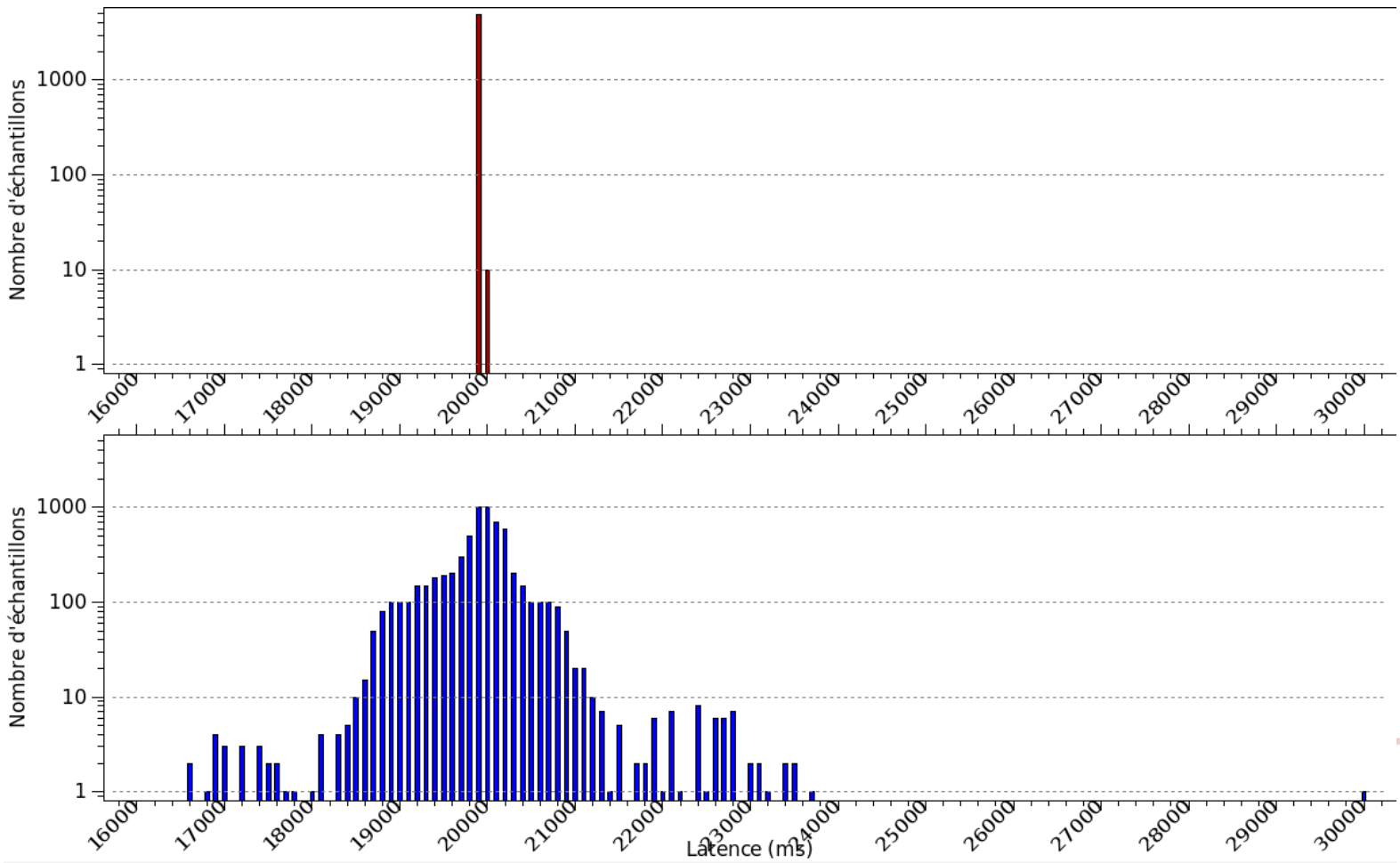
Latence aux évènements

Système classique



Latence aux évènements

Système classique



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

Noyau low latency
Problème du noyau normal

Dans un noyau Linux classique, il y a 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* (ordonnanceur désactivé)
- Équivalent d'une ressource partagée par tout les processus

→ Latence

Noyau low latency
Implémentation

- Difficile de gérer les différents contextes noyau

→ Noyau réentrant (= thread-safe)

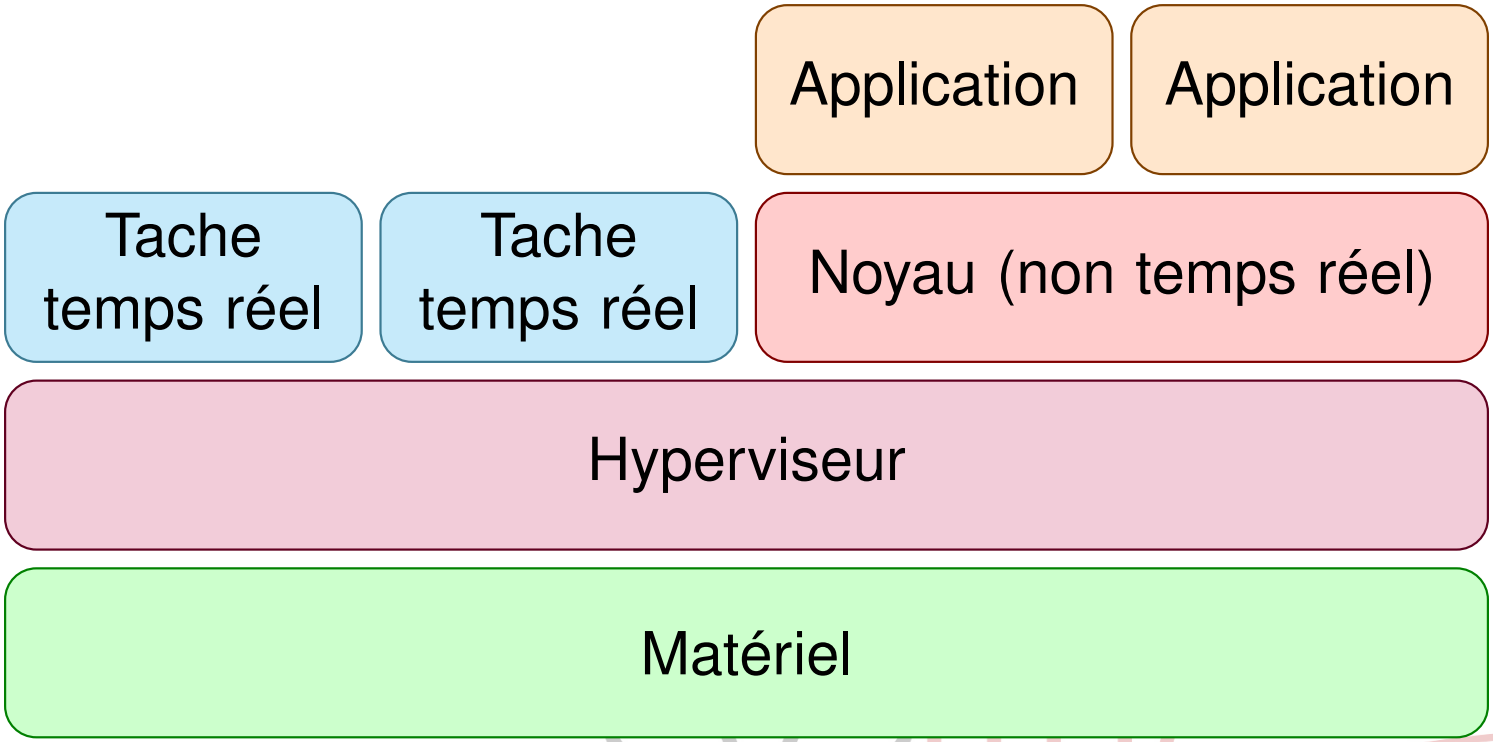
- Gestion des interruption assez complexe
- Overhead assez important

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


Application
Application
Tache
temps réel
Tache
temps réel
Noyau (non temps réel)
Hyperviseur
Matériel

Hyperviseur

- Possibilité de préempter le noyau sans patch low-latency
- Possibilité de différer 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 réel

→ 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)
  - http://www.xenomai.org/documentation/xenomai-head/html/api

RT-preempt

Gestion des interruption dans des threads

- Permet de préempter les interruptions

→ Moins de latence des taches

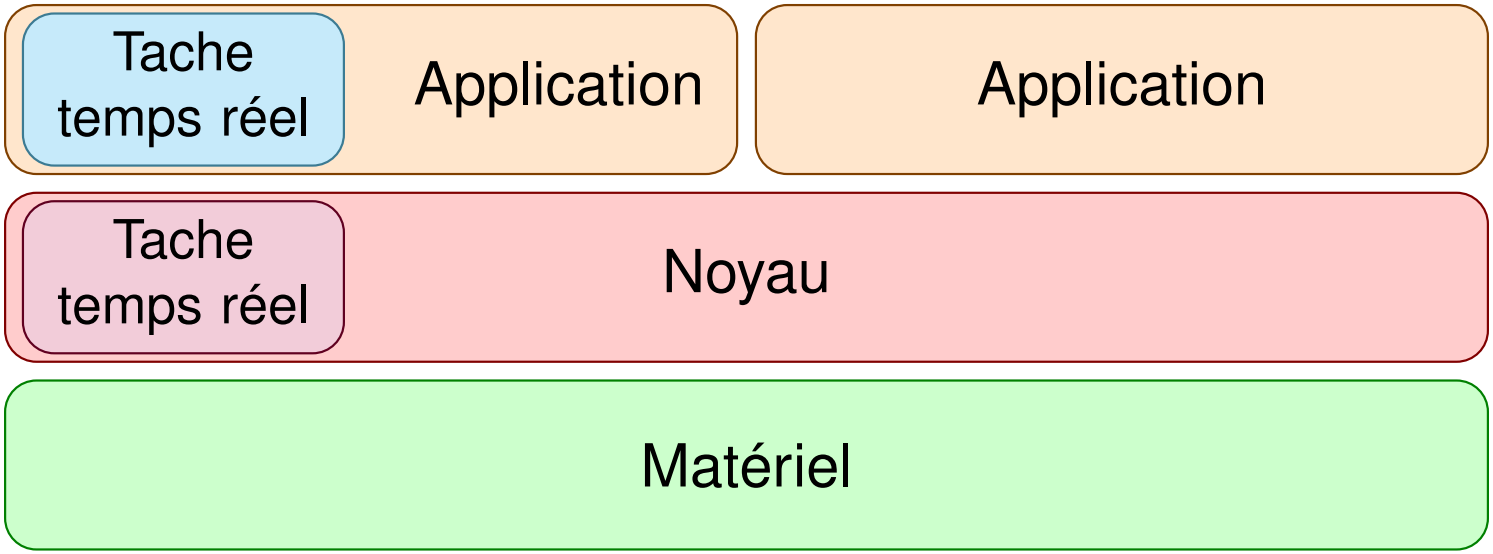
- Permet d'ordonnancer les interruption

→ 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/linux-2.6-rt.git)
- Implémentation assez complexe

→ Non portable

- Performance très bonnes (20μs)
- Intégré dans le noyau 3.0