Systèmes d'exploitation Temps Réel

email : kadionik@enseirb-matmeca.fr
web : kadionik.enseirb-matmeca.fr

Patrice KADIONIK
ENSEIRB-MATMECA

HISTORIQUE

- V 1.0 09/19 : Création du document. Récupération partie Temps Réel IT363.
- V 1.1 04/20 : Mise en forme. Ajout Xenomai 3, API *Alchemy* et *Cobalt*. Ecriture pendant le confinement.
- V 1.2 11/20 : MAJ et remise en forme.
- V 1.3 01/25 : MAJ mise en œuvre PREEMPT-RT et Xenomai 3.
- To do : WCET, modélisation.

DEDICACE

Je dédicace ce cours à mes 2 mentors que j'ai pu avoir dans ma vie professionnelle :

- Bernard Humbert du Centre de Recherche Nucléaire de Strasbourg. Ingénieur de Recherche à l'IN2P3, j'ai rencontré Bernard dans les années 1990-1991 quand j'étais doctorant. J'ai pu apprendre avec lui l'électronique numérique et l'informatique embarquée.
- Jean Louis Pédroza du Centre d'Etudes Nucléaires de Bordeaux-Gradignan. Ingénieur de Recherche à l'IN2P3, j'ai rencontré Jean Louis dans les années 1989-1992 quand j'étais doctorant. On était voisin de bureau ce qui était propice aux discussions. J'ai pu apprendre avec lui l'électronique numérique et l'informatique embarquée Temps Réel. Jean Louis n'est pas un anonyme pour l'option Systèmes Embarqués car il assuré ce présent cours durant de longues années de 2004 à 2018 même quand il est parti en retraite en 2017.

  Je remercie vivement Bernard et Jean Louis pour tout ce qu'ils m'ont donné.

Je n'oublie pas non plus mon ancien collègue Patrice Nouel, maître de conférences à l'ENSEIRB-MATMECA et parti en retraite en 2007. C'est le roi de l'électronique numérique et du VHDL. J'ai eu plaisir à travailler avec lui. Merci Pat !

CHAPITRE 0 :
INTRODUCTION

INTRODUCTION

- Ce module a pour but de présenter les éléments techniques pour appréhender au mieux le monde des systèmes Temps Réel :
  - Introduction aux systèmes Temps Réel.
  - Introduction au Temps Réel sous Linux. Les offres Linux Temps Réel.
  - Présentation de PREEMPT-RT et mesures de temps de latence sur cible x86.
  - Présentation de Xenomai et mesures de temps de latence sur cibles x86 et ARM (Raspberry Pi).
  - Compléments techniques sur Linux : ordonnancement, gestion du temps, programmation concurrente, norme POSIX et *threads*.
  - Mise en œuvre des API Xenomai *Mercury* et *Cobalt*.

POSIX=*Portable Operating System Interface eXchange*

API=*Application Programming Interface*

CHAPITRE 1 :
INTRODUCTION AU TEMPS REEL

LES SYSTEMES EMBARQUES ET LE TEMPS REEL

- Le Temps Réel est intimement lié à l'existence des systèmes embarqués.

- Quand on a un système embarqué, on se pose généralement des questions sur les temps de réaction et la question fatidique :

*« Est-ce que mon système réagit aux sollicitations **extérieures** dans un temps borné ? »*

LES SYSTEMES EMBARQUES ET LE TEMPS REEL

- Pour répondre à cette question, le système embarqué doit respecter des contraintes temporelles.

- On y trouve enfoui un système d'exploitation ou un noyau Temps Réel (dur) ou RTOS pour répondre à cette question.

- Le Temps Réel est un concept un peu vague. On pourrait le définir comme :

Un système est dit Temps Réel (dur) lorsque l'information après acquisition et traitement reste encore pertinente.

RTOS=*Real Time Operating System*

LES SYSTEMES EMBARQUES ET LE TEMPS REEL

- Cela veut dire par exemple que dans le cas d'une information arrivant par exemple de façon périodique (sous forme d'une interruption périodique du système), les temps d'acquisition et de traitement doivent rester inférieurs à la période de rafraîchissement de cette information pour qu'elle garde son sens.

- Pour cela, il faut que le noyau ou le système Temps Réel soit *déterministe* et *préemptif* (condition nécessaire mais pas suffisante).

LES SYSTEMES EMBARQUES ET LE TEMPS REEL

- Une confusion classique est de mélanger Temps Réel et rapidité de calcul du système donc puissance du processeur.

- On entend souvent :

Être temps Réel, c'est avoir beaucoup de puissance : des MIPS, des MFLOPS….

NON. Il faut répondre au plus juste dans un temps maximum garanti a priori !

LES SYSTEMES EMBARQUES ET LE TEMPS REEL

- Fonctionnement en Temps Réel :
  - Réactivité : des opérations de calcul doivent être faites en réponse à un événement extérieur (interruption matérielle).
  - La validité d'un résultat (et sa pertinence) dépend du moment où il est délivré.

- Rater une échéance va causer une erreur de fonctionnement : :
  - Temps Réel dur (*hard Real Time*) : plantage **irréversible**, destruction.
  - Temps Réel mou (*soft Real Time*) : dégradation temporaire **réversible** non dramatique des performances ou du fonctionnement du système.

TEMPS REEL DUR

- On parle de **Temps Réel dur** (*hard Real Time*) quand les événements traités trop tardivement ou perdus provoquent des conséquences catastrophiques pour la bonne marche du système (perte d'informations cruciales, plantage…).

- Les systèmes à contraintes dures ne tolèrent qu'une gestion stricte du temps afin de conserver l'intégrité du service rendu.

- On citera comme exemples les contrôles de processus industriels sensibles comme la régulation des centrales nucléaires ou les systèmes embarqués utilisés dans l'aéronautique.

TEMPS REEL DUR

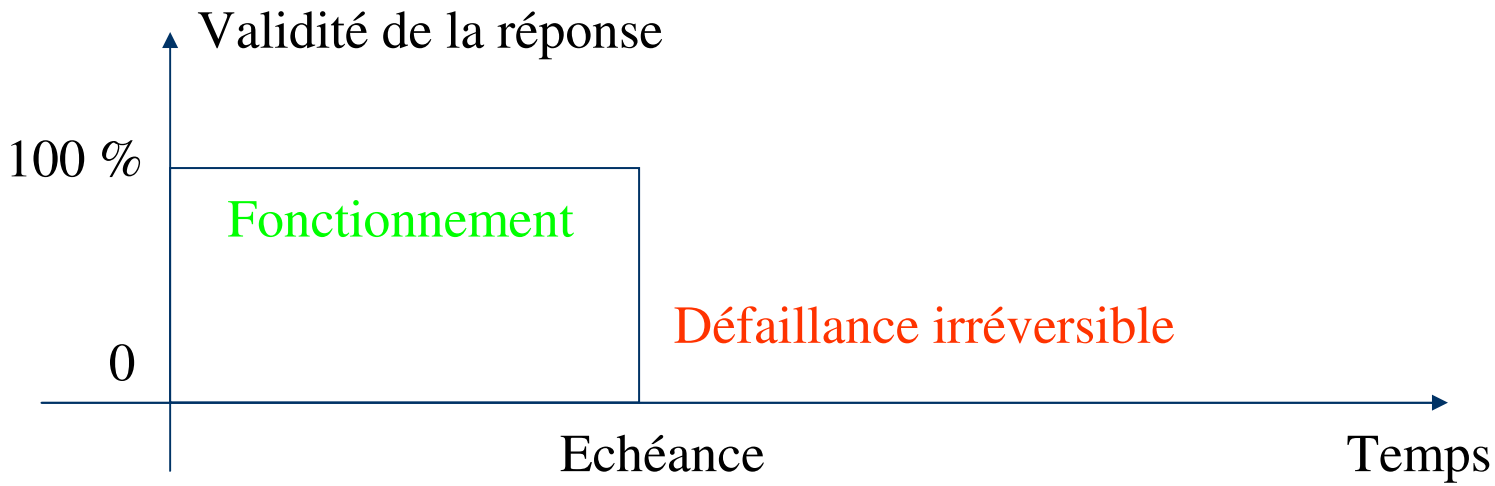
- **Ces systèmes garantissent un temps maximum d'exécution dans 100 % des cas.**

- On a ici une répartition stricte et totalitaire du temps CPU entre tâches.

- On peut dire qu'un système Temps Réel dur doit être prévisible (*prédictible*), les contraintes temporelles maximales garanties pouvant descendre à quelques µs.

- Le Temps Réel dur est celui de l'électronicien apparu dans les années 1960.

TEMPS REEL DUR



Temps réel dur

TEMPS REEL DUR

- **Le temps de latence maximum à respecter est fixé par le processus et son environnement à contrôler.**
- **Le temps de latence de son système doit rester inférieur à cette valeur.**
- **On ne cherche pas à avoir la valeur la plus petite pour son système qui remplit bien sûr la condition mais la plus juste car on risque sinon de mettre la pression sur le *design* du matériel et du logiciel par un surdimensionnement inutile…**

TEMPS REEL MOU

- On parle de **Temps Réel mou** (*soft Real Time*) quand les événements traités trop tardivement ou perdus sont sans conséquence catastrophique pour la bonne marche du système.

- **On ne garantit pas un temps de traitement maximum dans 100 % des cas tout en essayant de s'y rapprocher.**

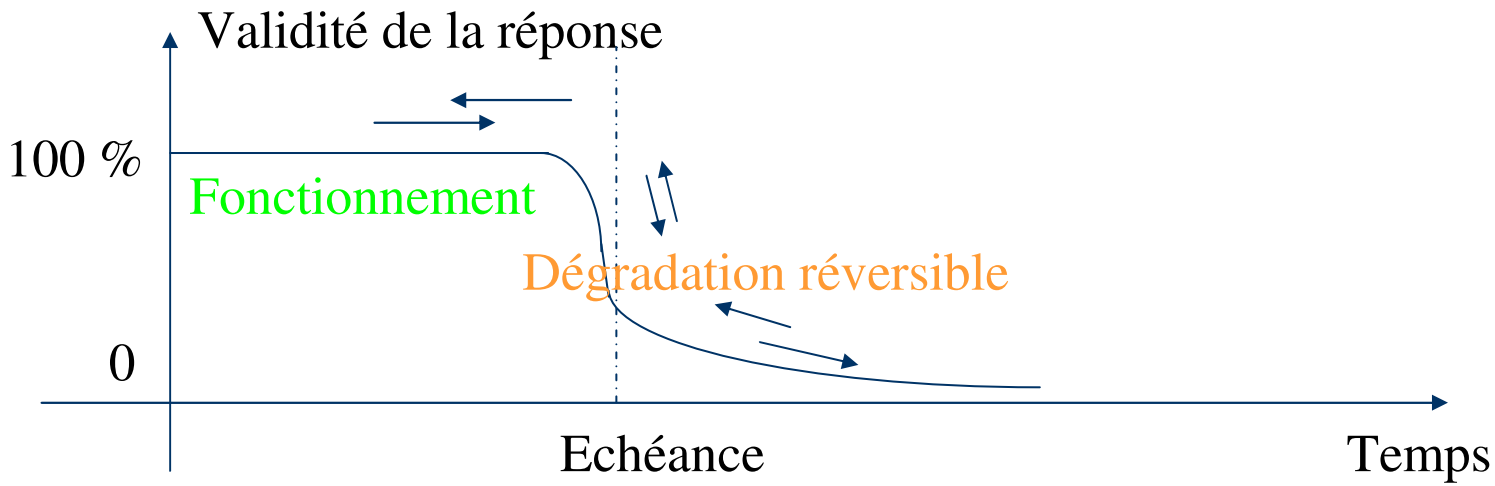
TEMPS REEL MOU

- On peut citer l'exemple des systèmes multimédia : si quelques images ne sont pas affichées correctement, cela ne met pas en péril le fonctionnement correct de l'ensemble du système. Au bout de quelques images perdues, le systèmes retrouve son fonctionnement normal. Cela ne nuit pas à l'usager…

- Le Temps Réel mou ne doit pas être utilisé s'il met en jeu la sécurité des personnes.

- Le Temps Réel mou est celui de l'informaticien apparu dans les années 2000.

TEMPS REEL MOU



Temps réel mou

TEMPS REEL ET QoS

- Le Temps Réel dur et le Temps Réel mou gèrent le temps de façon strict ou plus lâche.
- Cela définit une certaine Qualité de Service (QoS) sur la gestion du temps.
- Le Temps Réel peut être donc vu comme une QoS offerte à une application sur la gestion du temps !
- La QoS est imposée par le processus et son environnement extérieur à contrôler non pas par le système de contrôle lui-même !

QOS=*Quality of Service*

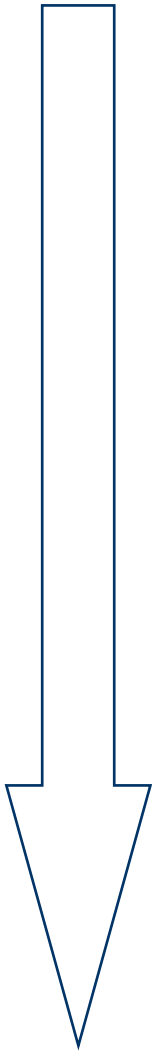
TEMPS REEL ET QoS

QoS lâche

Best effort : temps partagé

Temps réel mou

Temps réel dur



QoS stricte

CHAPITRE 2 :
LINUX ET LE TEMPS REEL

LINUX VANILLA ET LE TEMPS REEL

- Le noyau Linux de base ou standard sans application de modifications (*patchs*) sur les sources est appelé noyau *vanilla.*

- Les sources du noyau *vanilla* sont disponibles sur le site :

  https://www.kernel.org

- On considère alors le noyau *vanilla* avec sa configuration standard sans amélioration de la préemption :
  - Configuration *No Forced Preemption* (PREEMPT_NONE).

- A noter l'existence des configurations améliorant la préemption du noyau *vanilla* : *Voluntary Kernel Preemption* (PREEMPT_VOLONTARY) et *Preemptible Kernel* (PREEMPT).

LINUX VANILLA ET LE TEMPS REEL

- Le noyau *vanilla* n'est pas un système d'exploitation Temps Réel dur car :
  - Le noyau Linux possède de longues sections de code où tous les événements extérieurs sont masqués (non interruptible).
  - Le noyau Linux n'est pas préemptible durant toute l'exécution d'un appel système par un processus et ne le redevient qu'en retour d'appel système.

LINUX VANILLA ET LE TEMPS REEL

- Le noyau *vanilla* n'est pas un système d'exploitation Temps Réel dur car :
  - Le noyau Linux n'est pas préemptible durant le service d'une interruption ISR (*Interrupt Sub Routine*). La routine ISR acquitte **immédiatement** l'interruption (*Top Half*) puis programme un traitement **différé** des données dans une file (*workqueue*) par un *thread* du noyau (*Bottom Half*).
  - En cas d'occurrence d'une interruption durant l'exécution d'un appel système en mode noyau, le BH programmé par l'ISR (et éventuellement les autres BH des autres ISR) ne sera exécuté qu'à la fin de l'exécution complète de l'appel système d'où un temps de latence important et non borné !

LINUX VANILLA ET LE TEMPS REEL

- Le noyau *vanilla* n'est pas un système d'exploitation Temps Réel dur car :
  - L'ordonnanceur de Linux essaye d'attribuer de façon équitable le CPU à l'ensemble des processus. C'est une approche égalitaire et *best effort*.

LINUX VANILLA ET LE TEMPS REEL

- **Le noyau *vanilla* ne peut être pas être considéré comme Temps Réel dur. C'est un système d'exploitation généraliste GPOS.**

- Il existe néanmoins des solutions **Linux Temps Réel dur** pour une réactivité garantie dans 100 % des cas de quelques µs à quelques dizaines de µs.

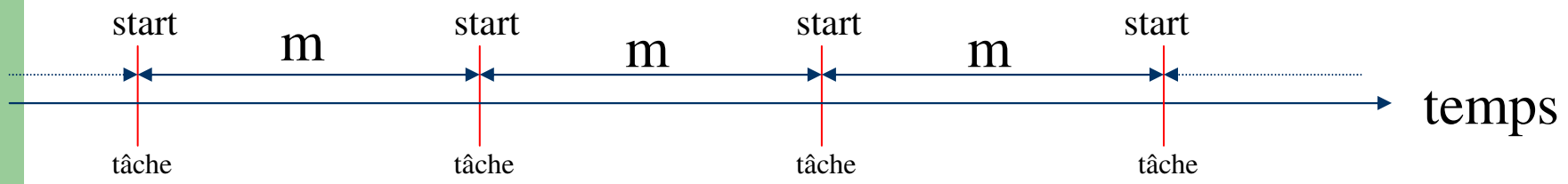
- Il existe des solutions **Linux Temps Réel mou** pour une réactivité non garantie dans 100 % des cas de quelques dizaines de µs à quelques centaines de µs.

GPOS=*General Purpose Operating System*

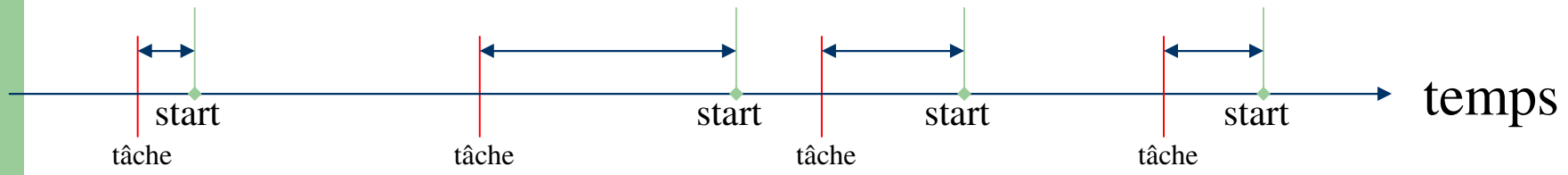
LINUX VANILLA ET LE TEMPS REEL

- Traitement des tâches périodiques :

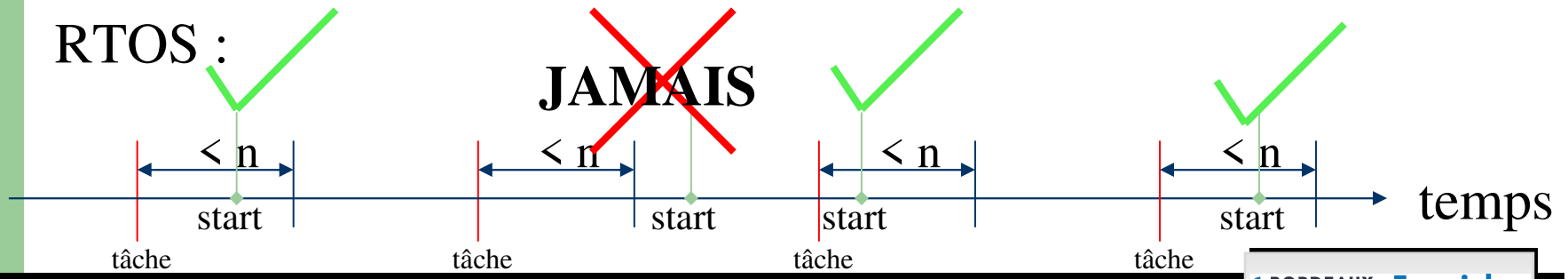
Idéalement :



Linux *vanilla* :



RTOS :



UNE PREMIERE EXPERIENCE

- Génération d'un signal périodique sur une broche du port parallèle d'un PC à l'aide d'un programme périodique.

- Le signal généré sur la broche 2 (bit D0) du port parallèle est théoriquement un signal périodique carré de demi-période de 50 ms.

- L'environnement de mesures est le suivant :
  - PC AMD Athlon : XP 1500+ avec 512 Mo de RAM.
  - Noyau 2.4.24 *vanilla* (PREEMPT_NONE).
  - Programme de mesure :
    - Fichier C *square.c*.

UNE PREMIERE EXPERIENCE

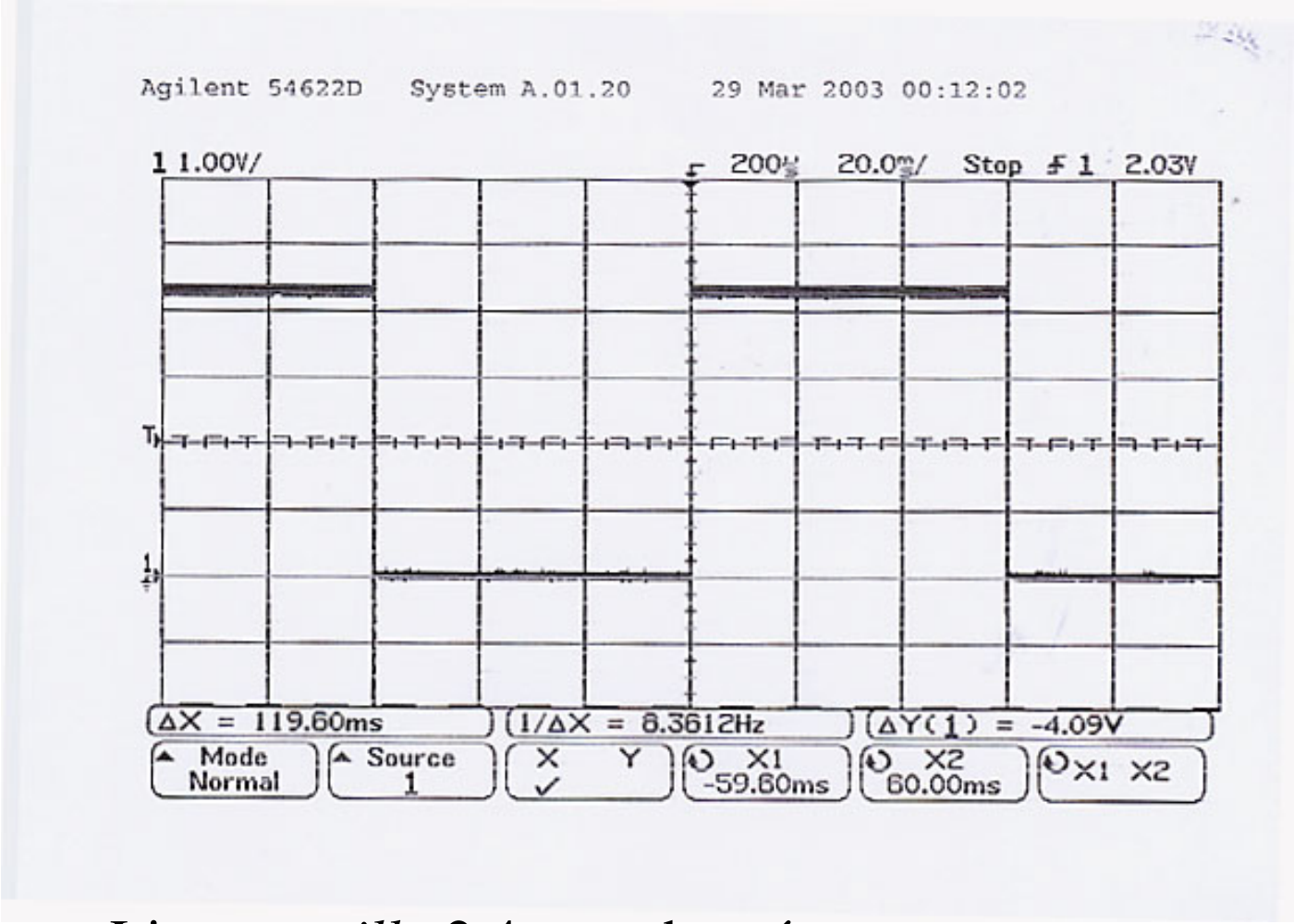
Fichier C *square.c* :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/io.h>

#define LPT 0x378
#define DEMI_PERIODE 50000
```

UNE PREMIERE EXPERIENCE

```
int main()
{
  setuid(0);

  if (ioperm(LPT, 1, 1) < 0) {
    perror("ioperm()");
    exit(-1);
  }
  while (1) {
    outb (0x00, LPT);
    usleep(DEMI_PERIODE);
    outb (0xff, LPT);
    usleep(DEMI_PERIODE);
  }

  exit(0);
}
```

UNE PREMIERE EXPERIENCE



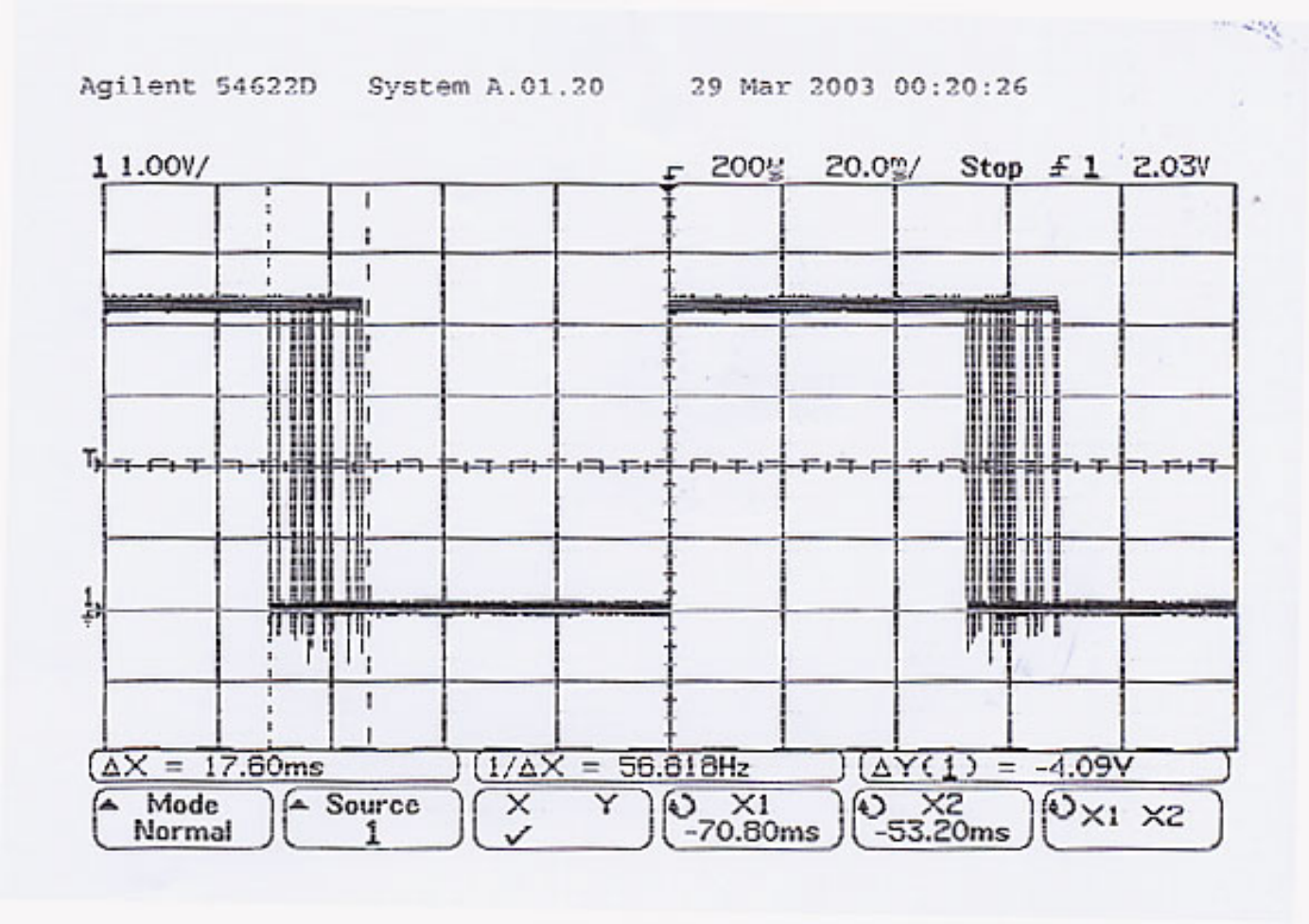
Linux *vanilla* 2.4 non chargé

UNE PREMIERE EXPERIENCE

- On remarque que l'on n'a pas une période de 100 ms mais de 119,6 ms dû au temps supplémentaire d'exécution des appels système.

- Dès que l'on stresse le système (écriture répétitive sur disque d'un fichier de 50 Mo), on observe le signal suivant :

UNE PREMIERE EXPERIENCE



Linux *vanilla* 2.4 chargé

UNE PREMIERE EXPERIENCE

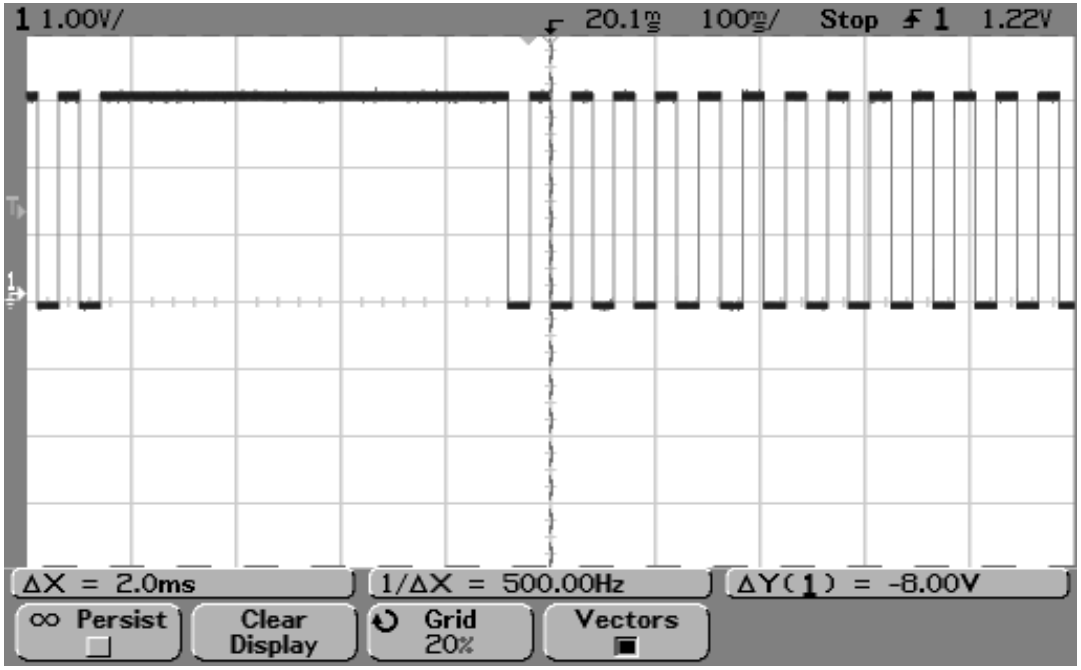
- On observe maintenant une gigue (*jitter*) sur le signal généré. La gigue maximale sur la durée de l'expérience est de 17,6 ms.

- La forme du signal varie maintenant au cours du temps, n'est pas de forme carrée mais rectangulaire. Linux *vanilla* 2.4 n'est donc plus capable de générer correctement ce signal.

- Il faut noter aussi que le front montant sur la figure précédente apparaît sans gigue car il a servi comme front de synchronisation de l'oscilloscope. La gigue observée est donc à voir comme la contribution de la gigue sur front montant et sur front descendant.

UNE PREMIERE EXPERIENCE

- Si l'on diminue la valeur de la demi-période, la gigue devient aussi importante que cette dernière et dans ce cas, Linux peut ne générer plus aucun signal :



Linux *vanilla* 2.4 chargé (période 40 ms)

UNE DEUXIEME EXPERIENCE

- Génération d'un signal périodique sur une broche PIN_11 (LED 6, voir TP) du port GPIO d'une carte Raspberry Pi à l'aide d'une tâche périodique de demi-période de 50 ms.

- L'environnement de mesures est le suivant :
  - Carte Raspberry Pi 3B avec 1 Go de RAM.
  - Noyau 5.6.14 *vanilla* (PREEMPT_NONE).
  - Programme de mesure :
    - Fichier C *square2.c*.
  - Mesures sur noyau chargé avec l'outil *stress*.

UNE DEUXIEME EXPERIENCE

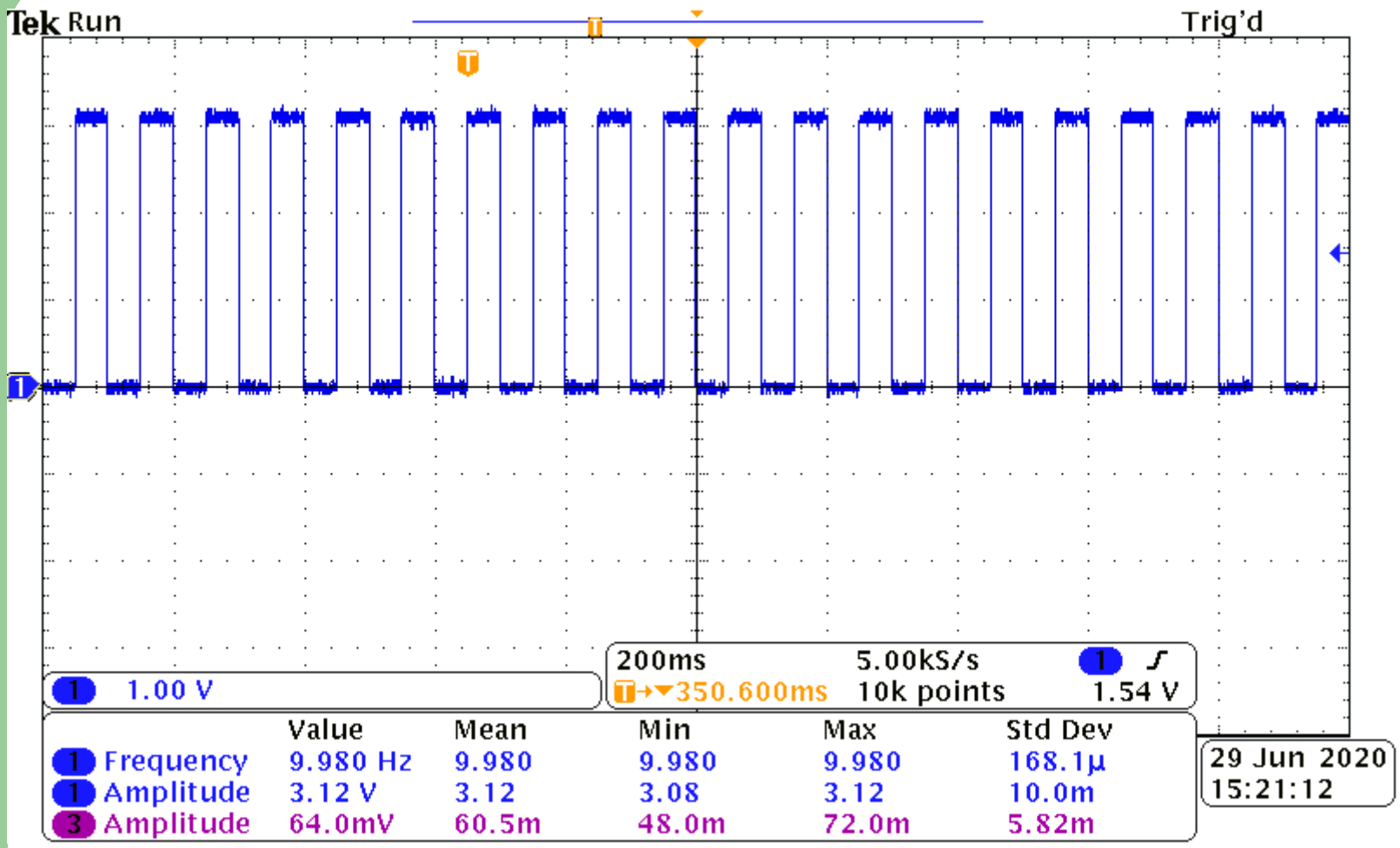
Fichier C *square2.c* :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

#include "bcm2835.h"
#include "bcm2835.c"
#include "bsp.h"
#include "bsp.c"

#define DEMI_PERIODE 50000 // en µs
```

UNE DEUXIEME EXPERIENCE

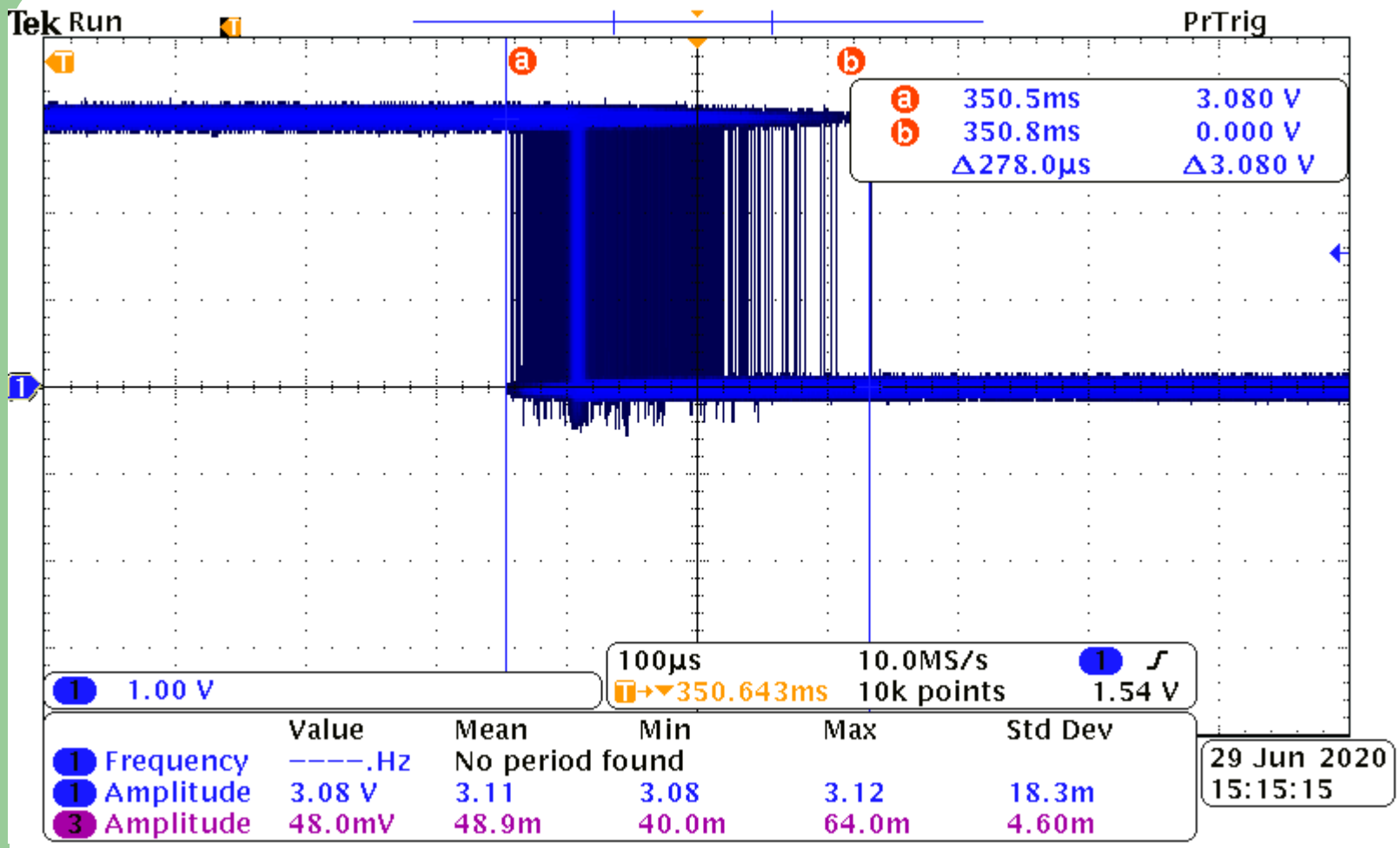
```
int main() {

  BSP_init();

  while(1) {
    BSP_clrLED(6);
    usleep(DEMI_PERIODE);
    BSP_setLED(6);
    usleep(DEMI_PERIODE);
  }

  BSP_release();
  exit(0);
}
```

UNE DEUXIEME EXPERIENCE



Linux *vanilla* 5.6 chargé

UNE DEUXIEME EXPERIENCE



Linux *vanilla* 5.6 chargé

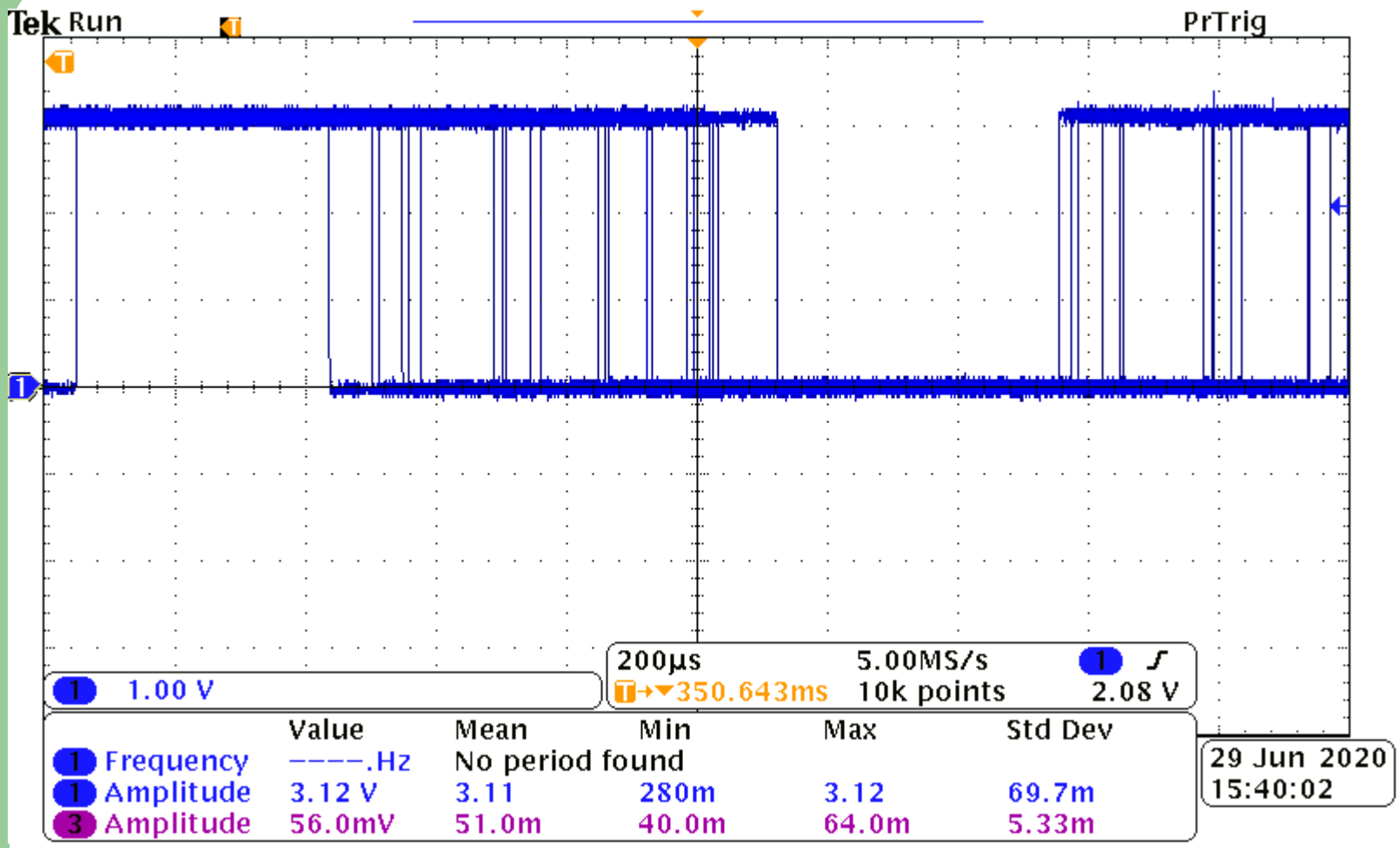
UNE DEUXIEME EXPERIENCE

- Le signal reste ici de forme carrée au cours du temps.

- On observe maintenant une gigue maximale sur la durée de l'expérience de 278 µs. Si l'on admet une erreur de 1% soit 1 ms pour une période de 100 ms, Linux *vanilla* 5.6 est donc capable de générer correctement ce signal.

- Si l'on diminue la valeur de la demi-période, en dessous de 5 ms, on observe des sauts de phase. Le signal généré est donc incorrect.

UNE DEUXIEME EXPERIENCE



Linux *vanilla* 5.6 chargé (période 2 ms)

UNE DEUXIEME EXPERIENCE

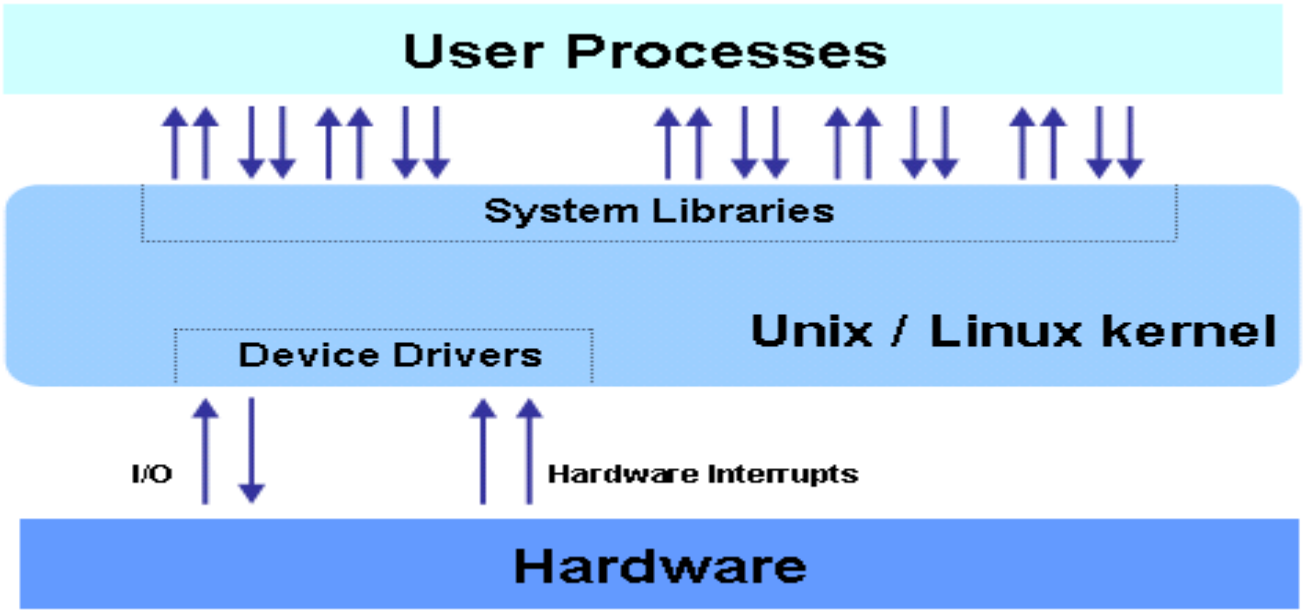
- Par rapport à la première expérience, on peut dire que :
  - On a ici une plateforme matérielle plus performante.
  - On a un noyau 5.6 *vanilla* nettement meilleur qu'un noyau 2.4 *vanilla* en mode (PREEMPT_NONE).

- On utilisera plutôt un noyau récent et on bannira tout noyau de version inférieure ou égale à 2.4.

- Même si avec cette deuxième expérience, cela semble faire l'affaire pour une période de 50 ms, on n'en est pas sûr dans 100 % des cas quelle que soit l'intensité de stress du noyau. Attention aux désillusions…

EXTENSIONS TEMPS REEL POUR LINUX

- Implémentation du noyau Linux *vanilla* :
  - Pas de support du Temps Réel.
  - Séparation entre le matériel et les processus Linux.
  - ...



EXTENSIONS TEMPS REEL POUR LINUX

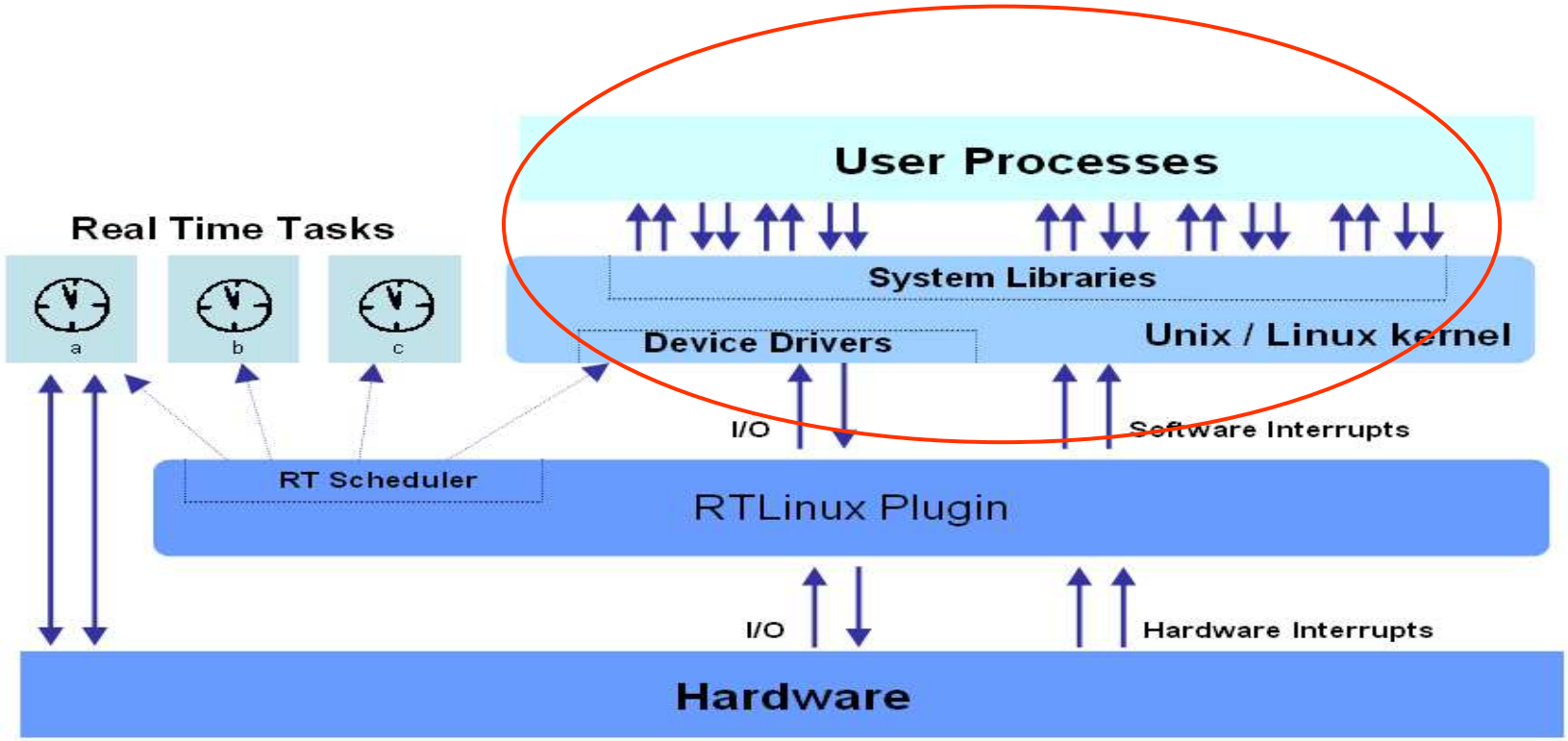
- Solution 1 pour une **extension Temps Réel mou** de Linux :
  - Modification du noyau Linux par application de *patchs* pour améliorer les contraintes Temps Réel : dévalider les interruptions le moins longtemps possible, appeler l'ordonnanceur le plus souvent possible (fonction *schedule()* du noyau), en retour d'interruption par exemple, réduction des temps de latence.

EXTENSIONS TEMPS REEL POUR LINUX

- Solution 2 pour une **extension Temps Réel dur** de Linux :
  - Ajout d'une couche d'abstraction entre le matériel et le noyau Linux.
  - Ajout d'un deuxième ordonnanceur (ou co-noyau) de tâches Temps Réel dur et considérer le noyau Linux et ses processus comme tâche de fond.
  - Pas de séparation entre le matériel et les tâches Temps Réel.
  - Plus difficile que la première solution.

EXTENSIONS TEMPS REEL POUR LINUX

- Solution 2 pour une **extension Temps Réel dur** de Linux :



LINUX COMME RTOS

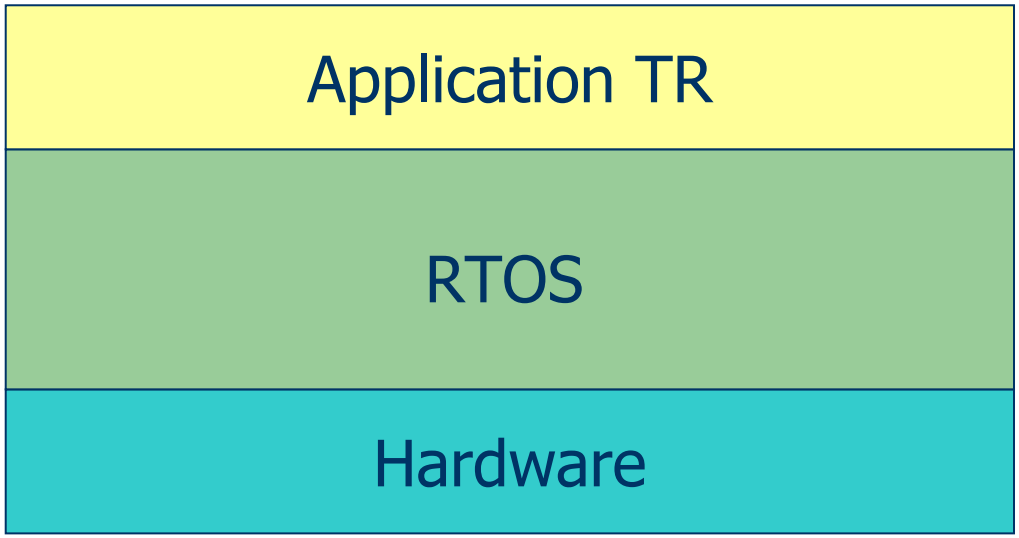
- Avantages : de nombreuses fonctionnalités, coopération entre tâches Temps Réel et processus Linux non Temps Réel.

- Inconvénients : n'est pas un véritable RTOS monolithique.

RTOS MONOLITHIQUE

- Avantages : simplicité, monolithique, fait pour le Temps Réel dur.

- Inconvénients : fonctionnalités plus ou moins limitées.



CHAPITRE 3 :
LES OFFRES LINUX TEMPS REEL

LES OFFRES LINUX TEMPS REEL

- Les offres de version de Linux embarqué et Temps Réel peuvent être rangées dans l'une des 2 catégories suivantes :
  - Les distributions Linux Temps Réel commerciales.
  - Les distributions Linux Temps Réel libres.

- Nous dresserons la liste des principales offres.

LINUX TEMPS REEL COMMERCIAL

- Montavista (Professional Edition) :
  https://www.mvista.com
- Wind River/Intel (RTLinux) :
  https://www.windriver.com
- LynuxWorks (BlueCat RT) :
  http://www.lynuxworks.com
- …

LINUX TEMPS REEL OPEN SOURCE

- Xenomai :
  http://xenomai.org

- RTAI (*Real Time Application Interface*) :
  http://www.aero.polimi.it/~rtai

- PREEMPT-RT :
  https://rt.wiki.kernel.org

RTOS COMMERCIAUX

- Il y a toujours les RTOS commerciaux non Linux :
  - VxWorks.
  - pSOS.
  - QNX.
  - µC/OS II.
  - …

RTOS OPEN SOURCE

- eCOS :
  http://sources.redhat.com/ecos

- FreeRTOS :
  http://www.freertos.org

CHAPITRE 4 :
COMPLEMENTS TECHNIQUES

INTRODUCTION

- Les informations présentées concernent un système d'exploitation de type *NIX.

- On se considèrera plus particulièrement le noyau Linux.

ORDONNANCEMENT

ORDONNANCEMENT

- 3 principales politiques d'ordonnancement sont disponibles sous Linux :
  - Temps partagé : SCHED_OTHER.
  - Temps Réel : SCHED_FIFO (*First In First Out*). Un processus ne peut être préempté que par un autre processus de plus forte priorité prêt à être exécuté.
  - Temps Réel : SCHED_RR (*Round Robin*). Pour un niveau de priorité donné, les processus prêts à être exécutés peuvent accéder chacun leur tour (en tourniquet) au processeur pendant un quantum de temps (*tick*).
- Il existe d'autres politiques d'ordonnancement comme SCHED_DEADLINE, SCHED_BATCH, SCHED_IDLE…

ORDONNANCEMENT

- Pour pouvoir utiliser les politiques d'ordonnancement SCHED_FIFO ou SCHED_RR, il faut être super utilisateur.

- Nous verrons plus loin comment fixer la politique d'ordonnancement quand nous aborderons le concept de *threads* POSIX.

POSIX= *Portable Operating System Interface eXchange*

ORDONNANCEMENT

- Les priorités de processus Temps Réel (mou ou dur) vont de 1 (moins prioritaire) à 99 (plus prioritaire).
- La politique d'ordonnancement est SCHED_FIFO ou SCHED_RR.
- La priorité Temps Réel 0 est l'ensemble des processus Linux standard à temps partagé. La politique d'ordonnancement est SCHED_OTHER.

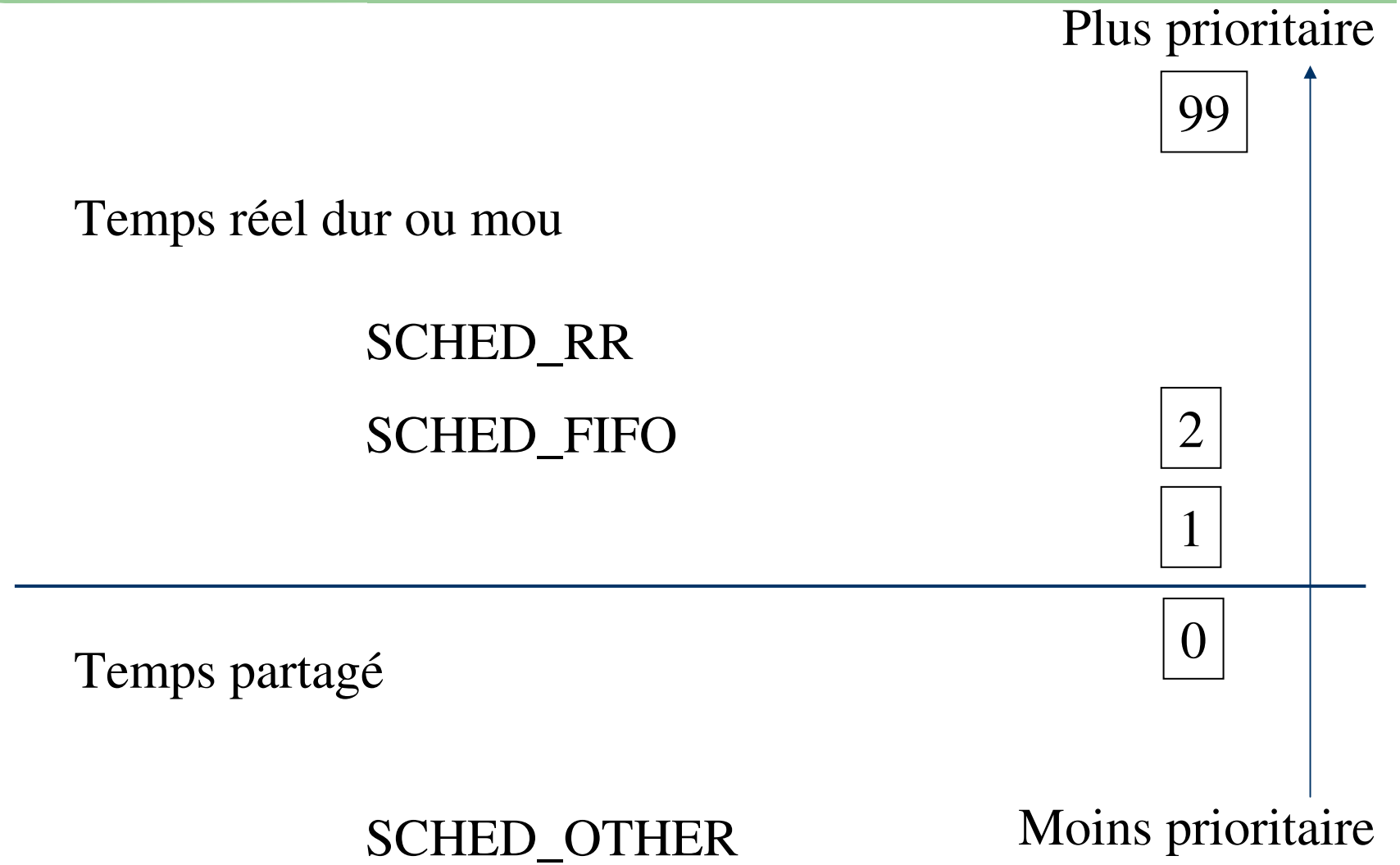
ORDONNANCEMENT

- Pour les processus Linux à temps partagé (priorité TR 0), on peut agir sur leur gentillesse (*nice*) entre -20 pour un processus à temps partagé agressif à +19 pour un processus à temps partagé gentil.

- Pour un processus Linux à temps partagé, on peut changer sa gentillesse avec la commande *nice* :
  - `# nice -n -20 commande  // -20`
  - `$ nice -n 19 commande  // +19`

- Pour une valeur inférieure à 0, il faut être super utilisateur.

ORDONNANCEMENT



ORDONNANCEMENT

- Il existe des outils pour changer dynamiquement la priorité et le type d'ordonnancement d'un exécutable à lancer ou bien d'un processus en cours d'exécution:
  - Commande `chrt`.
  - Outil `schedutils` :

  https://sourceforge.net/projects/schedutils

- Il faut être super utilisateur.

ORDONNANCEMENT

Exemple :

```
# chrt -f 99 ./mon_exe      // SCHED_FIFO priorité=99
# chrt -r 30 ./mon_exe      // SCHED_RR priorité=30
# chrt -o 0 ./mon_exe       // SCHED_OTHER priorité=0
# chrt -pf 99 1234   // Switch en SCHED_FIFO PID=1234
# chrt -pr 50 5678   // Switch en SCHED_RR PID=5678

# sched -f 99 ./mon_exe     // SCHED_FIFO priorité=99
# sched -r 30 ./mon_exe     // SCHED_RR priorité=30
# resched -f 99 1234        // Switch en SCHED_FIFO PID=1234
# resched -r 50 5678        // Switch en SCHED_RR PID=5678
```

ORDONNANCEMENT

- Le noyau Linux (depuis la version 2.6.25) possède un garde fou contre des processus Temps Réel (SCHED_FIFO ou SCHED_RR) qui consommeraient tout le temps CPU.

- Par défaut, 95 % du temps est réservé au plus aux processus Temps Réel et les 5 % restants sont pour les processus en temps partagé.

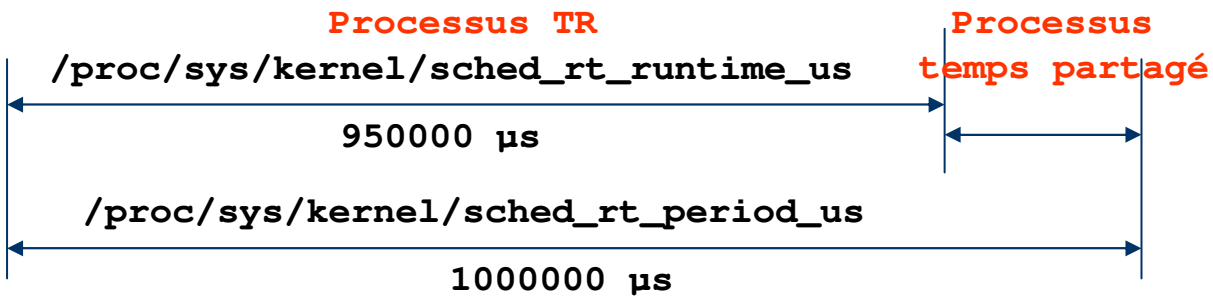
- On ne peut pas dépasser par défaut ces 95 % pour le Temps Réel : c'est le *throttling*.

- Cela évite d'avoir la sensation d'un système figé si les processus Temps Réel consommaient tout le temps CPU, ce qui est **pourtant le fonctionnement normal et attendu**.

ORDONNANCEMENT

- Le réglage se fait par 2 pseudo fichiers sous /proc :
  - `/proc/sys/kernel/sched_rt_runtime_us` : temps en µs sur une période réservé aux processus Temps Réel.
  - `/proc/sys/kernel/sched_rt_period_us` : valeur de la période en µs.

- La durée par défaut de la période est 1 seconde.



ORDONNANCEMENT

- Si l'on doit construire un système Temps Réel avec Linux, il faut dévalider le *throttling*.

- Cela se fait par le réglage dynamique à l'aide du pseudo fichier `sched_rt_runtime_us :`

```
# echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

GESTION DU TEMPS

GESTION DU TEMPS

- La gestion du temps est très importante dans un système Temps Réel.
- Il faut pouvoir faire des mesures précises (avec une erreur de mesure la plus faible possible).
- Il faut pouvoir avoir des *timers* pour mettre en place des *timeouts*.
- Il faut pouvoir avoir des *timers* pour mettre en place des tâches périodiques.

GESTION DU TEMPS

- Dans un système Linux, la valeur du *tick* est stockée dans une variable appelée HZ (quelques ms, par exemple 10 ms).

- Le nombre de *ticks* depuis le démarrage du système correspond aux *jiffies*. C'est pour beaucoup de RTOS la seule unité de temps disponible (voir µC/OS II avec `OStimeGet()`).

- En conséquence la plus petite période programmable était pendant longtemps la durée d'un *tick* jusqu'à l'apparition des timers haute résolution qui permettent d'avoir une période inférieure au *tick*.

GESTION DU TEMPS

- Le temps calendaire est mesuré depuis l'*epoch*.
- L'*epoch* est le temps zéro d'un système d'exploitation. Pour les systèmes *NIX, c'est le 1er janvier 1970 à 0h0m0s.
- Sous *NIX, le temps écoulé depuis l'*epoch* est stocké en secondes dans une variable signée de 64 bits.
- `time_t time(time_t &heure)`

  Renvoie dans la variable `heure` le temps écoulé en secondes depuis l'*epoch*.

GESTION DU TEMPS

- `int gettimeofday(struct timeval *tv, struct timezone *tz)`

  Renvoie dans une structure `timeval` le temps écoulé en secondes et µs depuis l'*epoch* en fonction de la zone.

Exemple :

```
#include <sys/time.h>
struct timeval tv;

gettimeofday(&tv, NULL);
printf("%ld.%06d\n" , tv.tv_sec, tv.tv_usec);
```

GESTION DU TEMPS

- On a aussi à disposition des *timers* POSIX haute résolution.
- Ces *timers* ont une résolution théorique de 1 ns dans leur implémentation **logicielle**.
- Ils sont reliés à un compteur **matériel** (*clock source*) qui s'incrémente à chaque période d'une horloge matérielle.
- Si la période de l'horloge matérielle est de 1 ns, alors la granularité du *timer* est 1 ns. Si la période de l'horloge est de 100 ns, alors la granularité du *timer* est 100 ns. On ne pourra pas descendre dans ce cas en dessous de 100 ns.

GESTION DU TEMPS

- Les *timers* POSIX utilisent une source d'horloge (rien à voir avec l'horloge matérielle) :
  - CLOCK_REALTIME : heure du système réglable qui fournit l'heure absolue. Elle est modifiable par les changements heure d'été/heure d'hiver et les zones.
  - CLOCK_MONOTONIC : heure non modifiable qui fournit le temps écoulé depuis le démarrage du système.

- Pour ne pas être perturbé, on utilise plutôt l'horloge CLOCK_MONOTONIC pour des mesures de temps et utiliser des *timers*.

GESTION DU TEMPS

- `int clock_getres(clockid_t horloge, struct timespec *resolution)`

  Renvoie dans `resolution` la résolution de l'heure en ns.

- `int clock_gettime(clockid_t horloge, struct timespec *heure)`

  Renvoie dans la structure `heure` l'heure.

- `int clock_settime(clockid_t horloge, struct timespec *heure)`

  Règle l'heure. Il faut être super utilisateur.

GESTION DU TEMPS

Exemple :

```
#include <time.h>
struct timespec ts;

clock_gettime(CLOCK_MONOTONIC, &ts);
printf("%ld.%09d\n" , ts.tv_sec, ts.tv_nsec);
```

GESTION DU TEMPS

- On peut alors mettre en œuvre les *timers* POSIX avec les appels système `timer_create()` et `timer_settime()`.

- `int timer_create(clockid_t horloge, struct sigevent *notification, timer_t *timer)`

  Initialise `timer`. Le premier argument est le type d'horloge, le deuxième argument précise comment réagir à l'expiration du *timer*, le troisième argument représente le *timer* à créer.

- `sigevent` est une structure composée de différents champs dont les plus intéressants sont :
  - `sigev_notify` : SIGEV_SIGNAL : génération d'un signal à l'expiration.
  - `sigev_signo` : numéro du signal. On choisira un signal Temps Réel comme SIGRTMIN par exemple.

GESTION DU TEMPS

- `int timer_settime(timer_t timer, int absolu, struct itimerspec *spec, struct itimerspec *precedent)`

  Arme le *timer*. Le premier argument est celui obtenu par `timer_create()`. Le deuxième argument précise la durée du *timer* en ns et le troisième argument ne sera pas utilisé ici.

  A l'expiration du *timer*, un signal pourra être généré ou un *thread* POSIX lancé.

Exemple :

```
#include <time.h>
timer_t timer,
struct sigevent event;
struct itimerspec spec;

mon_handler(){ . . . }
```

GESTION DU TEMPS

```
// Installation du handler
signal(SIGRTMIN, mon_handler);

// Type de notification
event.sigev_notify = SIGEV_SIGNAL;
event.sigev_signo  = SIGRTMIN;

// Periode du timer : 1s
spec.it_interval.tv_sec  = 1;
spec.it_interval.tv_nsec = 0;
spec.it_value = spec.it_interval;

timer_create(CLOCK_MONOTONIC, &event, &timer);
timer_settime(timer, 0, &spec, NULL);
```

CHAPITRE 5 :
MISE EN ŒUVRE DE LINUX TR MOU :
PRESENTATION DE PREEMPT-RT

PRESENTATION

INTRODUCTION

- Il existait au départ deux principaux *patchs* introduits avec le noyau Linux standard 2.4 permettant d'améliorer les temps de latence et la préemption en diminuant la taille des sections du noyau non préemptibles :
  - *Patch Preempt Kernel.*
  - *Patch Low Latency.*
- Ces *patchs* ont permis d'améliorer les temps de latence sur le noyau Linux standard version 2.4.

INTRODUCTION

- Ces *patchs* ont depuis été fondus dans le noyau *vanilla* 2.6 (et ce ne sont donc plus des *patchs*) et supérieur pour permettre une amélioration de la préemption du noyau *vanilla* :
  - Option *No Forced Preemption (Server)* : noyau *vanilla* non préemptible PREEMPT_NONE.
  - Option *Voluntary Kernel Preemption (Desktop)* : PREEMPT_VOLUNTARY.
  - Option *Preemptible Kernel (Low-Latency Desktop)* : PREEMPT (ou PREEMPT_DESKTOP).

- Ces améliorations n'ont rien à voir avec PREEMPT-RT.

PREEMPT-RT

- PREEMPT-RT est un *patch* à appliquer aux sources du noyau *vanilla* pour lui conférer la propriété Temps Réel mou.

- L'extension Temps Réel mou PREMPT-RT est donc à privilégier :
  - https://rt.wiki.kernel.org

- L'extension Temps Réel mou PREEMPT-RT donne de bien meilleurs résultats que les 2 options précédentes (PREEMPT_VOLUNTARY ou PREEMPT) du noyau *vanilla* avec un noyau totalement préemptible et des temps de latence plus faibles.

PREEMPT-RT

- Depuis septembre 2024 et le noyau 6.12, PREEMPT-RT a été intégré officiellement dans les sources du noyau *vanilla*.
- Cela veut que le Temps Réel mou est à la portée du tous !
- Aura-t-on un noyau Linux *vanilla* permettant par configuration lors de la compilation de ses sources de faire du temps partagé ou du Temps Réel mou ou Temps Réel dur ?
- La configuration/compilation de PREEMPT-RT revient à la configuration/compilation du noyau *vanilla*.

PREEMPT-RT

- Un des apports de PREEMPT-RT est la possibilité d'avoir des *timers* haute résolution (*High Resolution Timer*) dont la résolution descend en dessous du *tick* (HZ). Ils sont maintenant intégrés au noyau *vanilla* (voir gestion du temps juste avant).

- On met en œuvre ici un *timer* matériel (*clock event*) dont la plus petite période programmable peut atteindre 1 ns si le *timer* le supporte.

PREEMPT-RT

- Le noyau Linux PREEMPT-RT est totalement préemptible même en cas d'occurrence d'une interruption.

- Les ISR sont traitées comme des *threads* (*threaded interrupt*). Il faudra veiller aux priorités appliquées dans ce cas.

PREEMPT-RT

```
BusyBox v1.15.3 (2010-05-04 17:30:42 CEST) hush - the humble shell
root:/> ps
  PID USER       VSZ STAT COMMAND
    1 root      1832 S    /init
    2 root         0 SW<  [kthreadd]
    3 root         0 SW<  [sirq-high/0]
    4 root         0 SW<  [sirq-timer/0]
    5 root         0 SW<  [sirq-net-tx/0]
    6 root         0 SW<  [sirq-net-rx/0]
    7 root         0 SW<  [sirq-block/0]
    8 root         0 SW<  [sirq-tasklet/0]
    9 root         0 SW<  [sirq-sched/0]
   10 root         0 SW<  [sirq-hrtimer/0]
   11 root         0 SW<  [sirq-rcu/0]
   12 root         0 SW<  [posixcputmr/0]
   13 root         0 SW<  [watchdog/0]
   14 root         0 SW<  [desched/0]
   15 root         0 SW<  [rcu_sched_grace]
   16 root         0 SW<  [events/0]
   17 root         0 SW<  [khelper]
   77 root         0 SW<  [kswapd0]
   78 root         0 SW<  [aio/0]
   79 root         0 SW<  [nfsiod]
  595 root         0 SW<  [irq/4-JTAGUART]
  605 root         0 SW<  [irq/7-eth%d]
  609 root         0 SW<  [rpciod/0]
  627 root      1976 S    -/bin/sh
  628 root      1692 S    /sbin/inetd
  629 root      1968 R    ps
```

PREEMPT-RT
sur processeur NIOS II

MISE EN ŒUVRE SUR CIBLE X86
PERFORMANCES

PREEMPT-RT SUR X86

- Nous regarderons la mise en œuvre sur un PC (processeur x86) sous Linux (distribution Fedora ici) :
  - Noyau Linux *vanilla* version 6.12.7.

- Récupération des sources du noyau *vanilla* :

```
$ cd $HOME
$ mkdir preempt-rt
$ cd $HOME/preempt-rt
$ wget
  https://www.kernel.org/pub/linux/kernel/v6.x/linux-
  6.12.7.tar.gz
```

PREEMPT-RT SUR X86

- Décompression des sources du noyau *vanilla* :

```
$ cd $HOME/preempt-rt
$ tar -xvzf linux-6.12.7.tar.gz
```

- L'EXTRAVERSION du fichier *Makefile* est changée pour signifier que l'on a un noyau PREEMPT-RT :

```
$ cd $HOME/preempt-rt/linux-6.12.7
$ nano Makefile
```

PREEMPT-RT SUR X86

```
VERSION = 6
PATCHLEVEL = 12
SUBLEVEL = 7
EXTRAVERSION =
```

devient :

```
VERSION = 6
PATCHLEVEL = 12
SUBLEVEL = 7
EXTRAVERSION = -rt
```

PREEMPT-RT SUR X86

- Configuration du noyau. On laissera par défaut la configuration PREEMPT-RT :

```
$ cd $HOME/preempt-rt/linux-6.12.7
$ make x86_64_defconfig
```

Ou bien :

```
$ cp /boot/config-6.12.7-200.fc41.x86_64 .config
```

- On validera PREEMPT-RT :

```
$ make gconfig
```

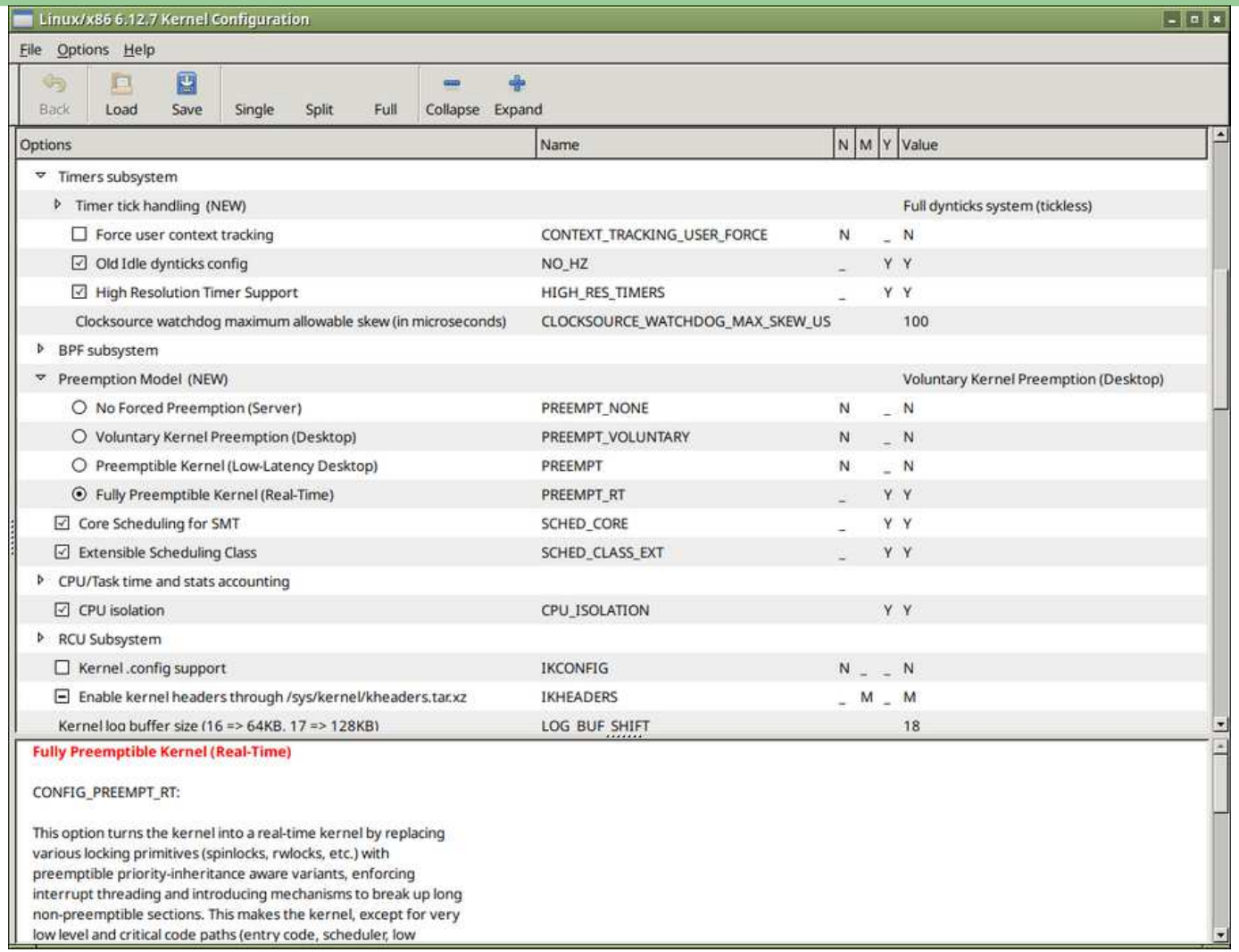
PREEMPT-RT SUR X86

- Une fenêtre de configuration apparaît. On cherchera les menus et sous-menus :
  - *General setup* > *Preemption Model* et validera l'option *Fully Preemptible Kernel (Real-Time)* (PREEMPT_RT).

- De même, on vérifiera que l'option HIGH_RES_TIMERS :
  - *General setup* > *Timers subsystem*

  est bien validée pour avoir le support des *timers* haute résolution pour pouvoir ensuite utiliser l'utilitaire *cyclictest*.

PREEMPT-RT SUR X86



PREEMPT-RT SUR X86

- Compilation du noyau PREEMPT-RT :

```
$ cd $HOME/preempt-rt/linux-6.12.7
$ make -j$(cat /proc/cpuinfo | grep processor | wc -l)
```

- Compilation des modules :

```
$ cd $HOME/preempt-rt/linux-6.12.7
$ make -j$(cat /proc/cpuinfo | grep processor | wc -l)
  modules
```

PREEMPT-RT SUR X86

- Installation du noyau PREEMPT-RT et de ses modules :

```
$ cd $HOME/preempt-rt/linux-6.12.7
$ sudo make modules_install
$ sudo make install
$ sudo cp System.map /boot/System.map-6.12.7-rt
$ sudo cp .config /boot/config-6.12.7-rt
```

- Génération du fichier *RAM Disk* pour le noyau PREEMPT-RT. Il n'y a rien à faire !

PREEMPT-RT SUR X86

- On a au final 4 nouveaux fichiers dans le répertoire */boot/* :

```
$ ls /boot/*rt*
/boot/config-6.12.7-rt
/boot/System.map-6.12.7-rt
/boot/initramfs-6.12.7-rt.img
/boot/vmlinuz-6.12.7-rt
```

- Mise à jour de *grub2*. Il n'y a rien à faire !

- Redémarrage du PC et au menu d'accueil choisir le noyau *Fedora Linux (6.12.7-rt) 41 (Forty One).*

PREEMPT-RT SUR X86

- On pourra alors vérifier que PREEMPT-RT est installé :

```
$ sudo dmesg | grep PREEMPT
[ 0.000000] Linux version 6.12.7-rt (patrice@fedora) (gcc
  (GCC) 14.2.1 20240912 (Red Hat 14.2.1-3), GNU ld
  version 2.43.1-5.fc41) #1 SMP PREEMPT_RT Sun Jan 5
  19:40:39 CET 2025
```

- On pourra enfin vérifier la version du noyau :

```
$ uname -r
6.12.7-rt
```

PREEMPT-RT SUR X86

- Les mesures de temps de latence se font d'abord avec un noyau Linux PREEMPT-RT non chargé puis chargé, c'est-à-dire stressé.

- Pour stresser le noyau Linux, il faut :
  - Consommer du temps processeur.
  - Générer des interruptions (*ping flooding*).
  - Faire des accès d'E/S (disque dur…).

PREEMPT-RT SUR X86

- Outils de la suite *rt-tests* pour l'outil *cyclictest* :

```
$ cd $HOME
$ git clone git://git.kernel.org/pub/scm/utils/rt-
  tests/rt-tests.git
$ cd rt-tests
$ make
$ sudo make install
```

- Outil *cyclictest* : mesure du temps de latence sur des *threads* périodiques :

```
$ sudo cyclictest -n -p 99 -i 1000
```

PREEMPT-RT SUR X86

- On pourra utiliser l'outil *stress* (http://weather.ou.edu/~apw/projects/stress) :

```
$ stress -c 20 -i 20 &
```

- Exemple de stress du noyau Linux. Script *load* lancé en étant super utilisateur :

```
#!/bin/bash
stress -c 20 -i 20 &
ping -f localhost &
```

PREEMPT-RT SUR X86

- L'environnement de mesures est le suivant :
  - PC AMD Intel i5-2500K à 3,3 GHz avec 24 Go de mémoire RAM.
  - Noyau PREEMPT-RT 6.12.7-rt.
  - Programme de mesure :
    - Outil PREEMPT-RT *cyclictest*.
  - Mesures sur noyau non chargé.
  - Mesures sur noyau chargé avec le script *load*.

PREEMPT-RT SUR X86

Outil *cyclictest* :

- Linux PREEMPT-RT non chargé :

```
$ sudo cyclictest -m -p 99 -i 1000
policy: fifo: loadavg: 1.31 0.71 0.27 1/223 2054

T: 0 ( 2029) P:99 I:1000 C:  30161 Min:      7 Act:   10 Avg:   13 Max:      14
```

- Linux PREEMPT-RT chargé :

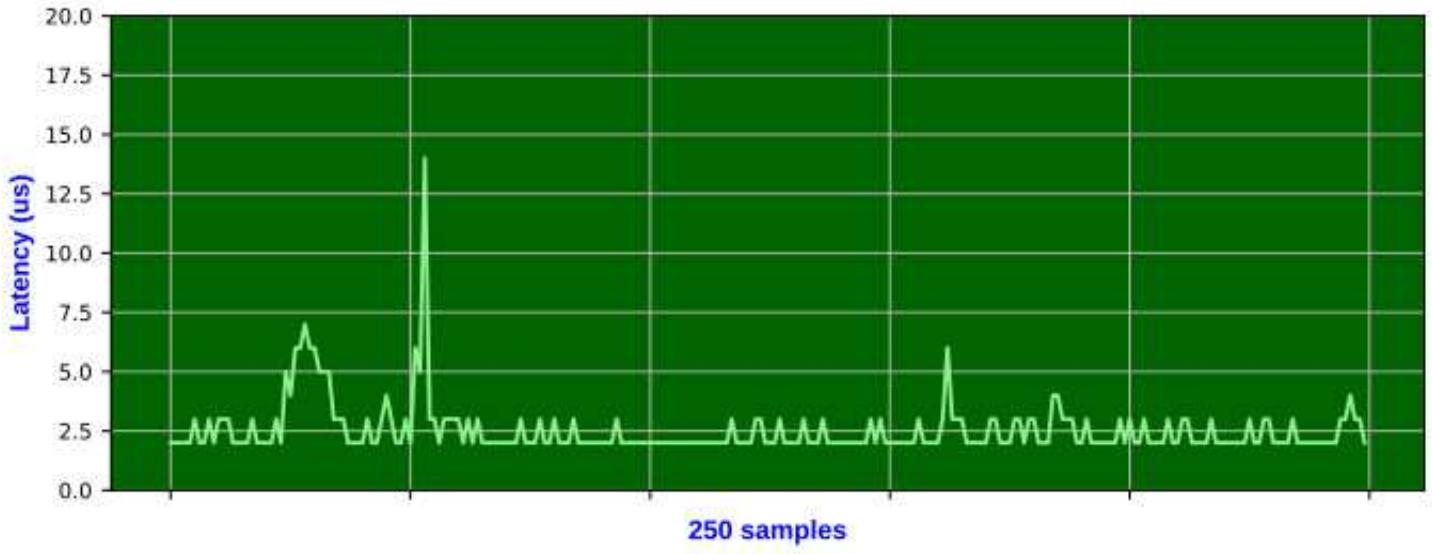
```
$ sudo cyclictest -m -p 99 -i 1000
policy: fifo: loadavg: 80.77 40.90 16.40 1/227 2203

T: 0 ( 2029) P:99 I:1000 C: 314106 Min:      6 Act:   10 Avg:   10 Max:      16
```

PREEMPT-RT SUR X86

Outil *cyclictest+oscilloscope* :

- Linux PREEMPT-RT non chargé :

```
$ sudo dnf install oscilloscope

$ sudo cyclictest –m –p 99 –i 1000 –v | oscilloscope –m 50
```



PREEMPT-RT SUR X86

- Linux PREEMPT-RT chargé :

  Vu la charge CPU, l'outil graphique *oscilloscope* (qui est alors un processus Linux non Temps Réel) ne marche plus (reste figé).

CHAPITRE 6 :
MISE EN ŒUVRE DE LINUX TR DUR :
PRESENTATION DE XENOMAI

PRESENTATION

INTRODUCTION

- Xenomai est un projet libre qui permet une migration simple d'applications basées sur des RTOS traditionnels généralement propriétaires vers l'environnement Linux.
- Xenomai facilite cette migration en proposant l'émulation des API propriétaires des RTOS.
- Xenomai factorise les points communs des API propriétaires des RTOS.

API=*Application Programming Interface*

INTRODUCTION

- 2001 : Xenomai 1
  - Support d'API propriétaires de RTOS.
  - Développement de la couche Adeos pour Linux et RTAI.
  - Fusion dans le projet RTAI pour former RTAI/fusion.
  - **Abandonné.**
- 2005 : Xenomai 2
  - Départ du projet RTAI.
  - Adeos évolue pour donner la couche *I-pipe*.
  - Portage sur 6 architectures de processeur.
  - **Obsolète.**

INTRODUCTION

- 2015 : Xenomai 3 (Xenomai *Cobalt*)
  - Réécriture complète pour donner Xenomai *Cobalt* et Xenomai *Mercury*.
  - Support de Linux en natif avec Xenomai *Mercury*.
  - **En production.**

- 2016 : Xenomai 4 (Xenomai *EVL*)
  - la couche *I-pipe* devient *Dovetail*. *Dovetail* est aussi disponible pour Xenomai 3.
  - Le noyau Temps Réel dur *Cobalt* de Xenomai 3 devient transitoirement le noyau *Steely* pour devenir définitivement le noyau *EVL* (moitié de la taille du noyau *Cobalt*).
  - Réécriture complète de Xenomai *EVL*.
  - Pas de support de l'API POSIX.
  - **En développement.**

INTRODUCTION

- Le projet Xenomai est maintenu par des développeurs de talent essentiellement français :
  - Philippe Gérum : créateur et mainteneur principal du projet.
  - Gilles Chanteperdrix (RIP): architecture ARM, x86, RTnet.
  - Jan Kiszla.
  - Alexis Berlemont, Jorge Mairez Ortiz, Wolfgang Grandegger.
  - …
- Xenomai est disponible à l'adresse :
  - http://xenomai.org

INTRODUCTION

- On considéra par la suite **Xenomai 3**.

- Xenomai fournit des garanties Temps Réel dur si le noyau Linux hôte ne peut le faire seul et offre des services POSIX Temps Réel.

- Cela signifie que Xenomai peut s'utiliser de 2 façons :
  - Préemption native. On utilise le noyau Linux hôte et l'on peut éventuellement réduire la latence et améliorer la préemption du noyau *vanilla* en utilisant optionnellement PREEMPT-RT si l'application le demande (Temps Réel mou). C'est Xenomai *Mercury* (configuration simple noyau).
  - Co-noyau Temps Réel dur pour obtenir les garanties temps réel requises si l'application demande des performances Temps Réel dur. C'est Xenomai *Cobalt* (configuration double noyau).

INTRODUCTION

- Xenomai a donc deux rôles :
  - Emuler fidèlement les API traditionnelles des RTOS propriétaires avec Xenomai *Mercury* et Xenomai *Cobalt*.
  - Fournir des garanties temps réel dur si le noyau Linux hôte ne peut pas le faire seul avec Xenomai *Cobalt* et son co-noyau Temps Réel dur.

INTRODUCTION

- Xenomai *Cobalt* et Xenomai *Mercury* sont un apport important de Xenomai 3 qui n'existait pas avec Xenomai 2. Ce n'est pas le seul apport.

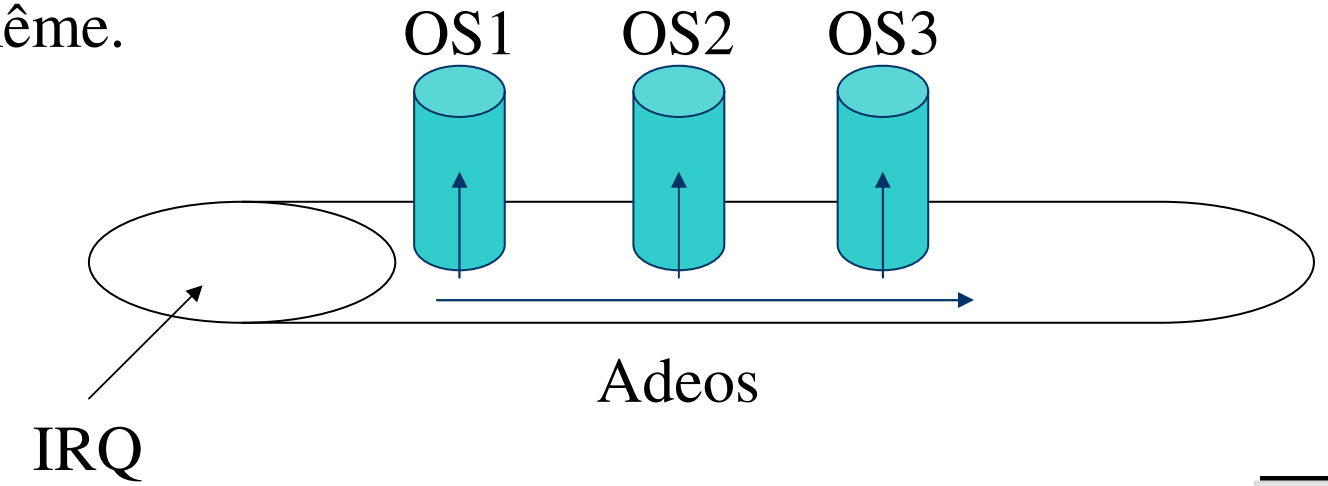
- Le co-noyau de Xenomai *Cobalt* s'appelle aussi *Cobalt* (co-noyau *Nucleus* dans Xenomai 2).

- Il s'appuie sur une couche logicielle appelée *pipeline* d'interruptions ou *I-pipe* issue du projet Adeos qui lui donne la priorité absolue sur la prise en charge des interruptions matérielles.

INTRODUCTION

- La couche Adeos (*Adaptative Domain Environment for Operating Systems*) a été proposée en 2001 par Karim Yaghmour pour capturer les interruptions sous la forme d'un *pipeline* logiciel afin de les redistribuer à différents systèmes d'exploitation (OS).

- Depuis 2005, Adeos est devenu l'*I-pipe*. Le principe reste le même.



INTRODUCTION

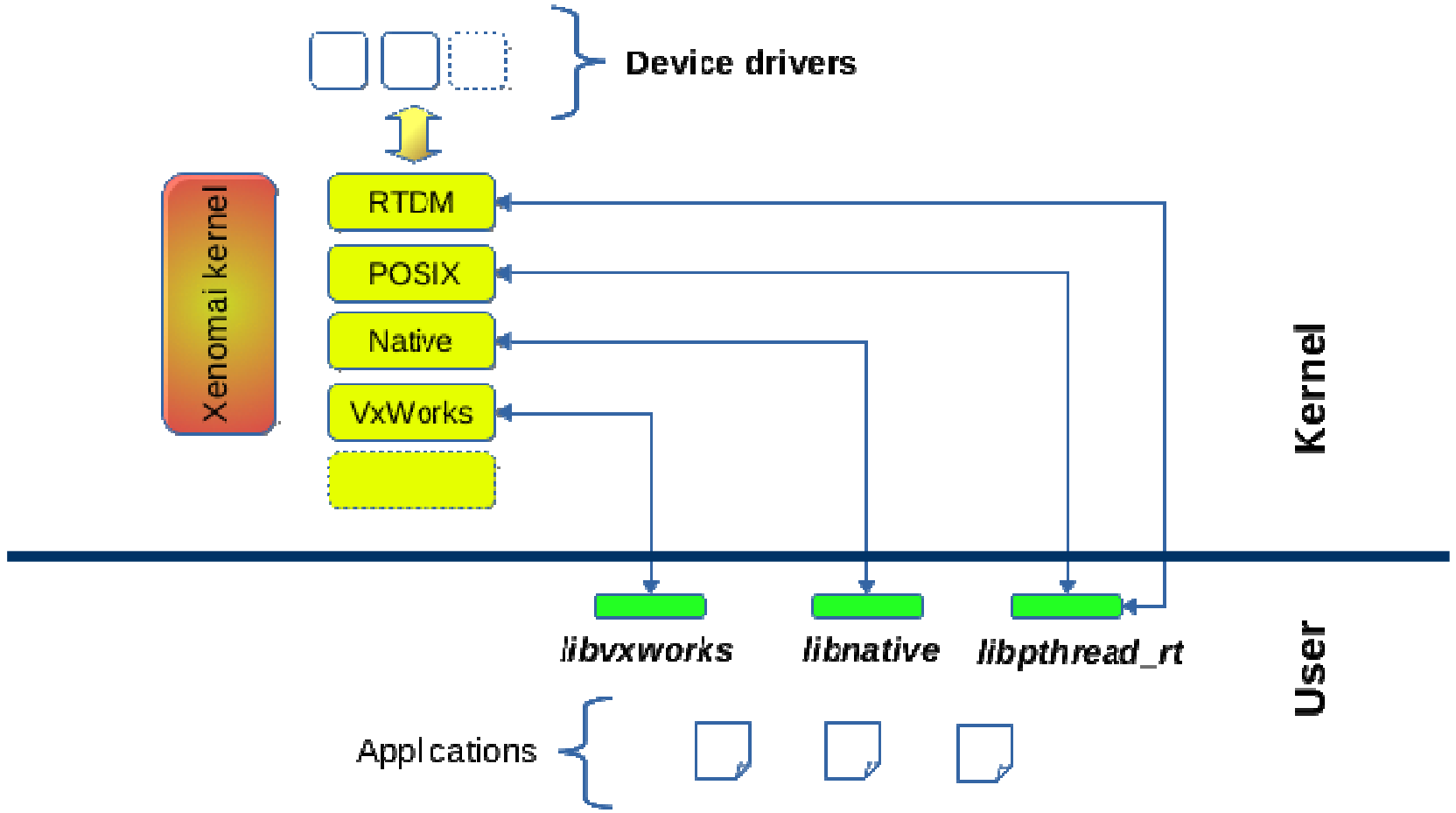
- Il s'agit d'une forme de virtualisation. On y définit des domaines (noyau *Cobalt* et noyau Linux) avec une priorité entre les domaines. Le domaine *Cobalt* à l'entrée de l'*I-pipe* est le plus prioritaire.

- L'*I-pipe* se greffe ainsi au noyau Linux et intercepte toutes les interruptions avant Linux pour les distribuer à qui de droit pour le traitement. C'est donc aussi un *patch* qu'il faut appliquer aux sources du noyau *vanilla*.

XENOMAI 2

- Xenomai 2 ne propose qu'une solution à base d'un co-noyau (co-noyau *Nucleus*).
- Xenomai 2 met toutes les émulations des API propriétaires de RTOS sur un même plan (y compris l'API POSIX).
- Toutes les émulations d'API sont dans l'espace noyau.
- On ne peut donc utiliser ces émulateurs qu'avec le co-noyau *Nucleus* uniquement dans une configuration double noyau.

XENOMAI 2



Architecture de Xenomai 2

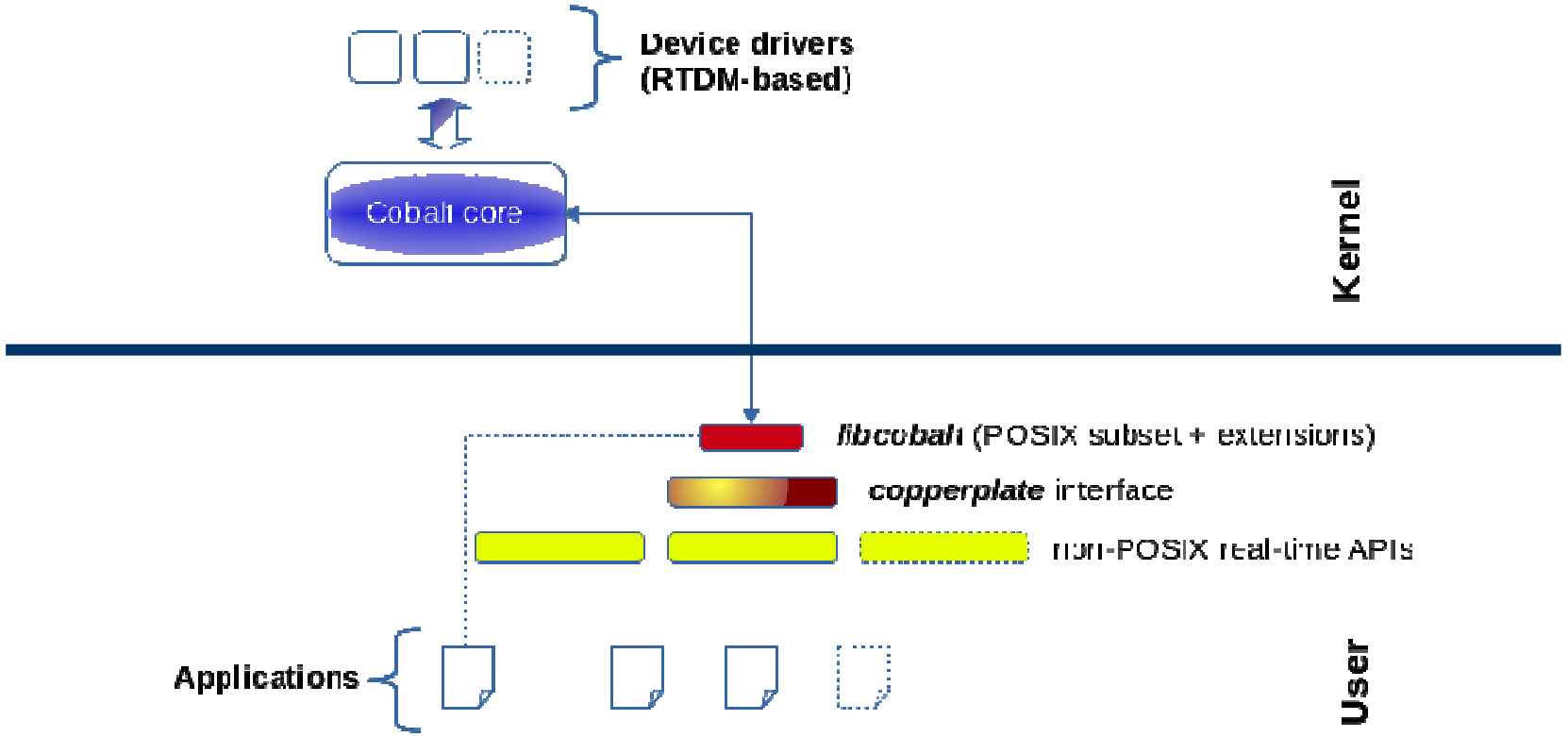
XENOMAI 3

- Xenomai 3 transfert les émulateurs des API propriétaires de RTOS dans l'espace utilisateur que ce soit pour Xenomai *Cobalt* ou Xenomai *Mercury*.

- On peut donc utiliser ces émulateurs dans une configuration simple noyau (*Mercury*) ou double noyau (*Cobalt*).

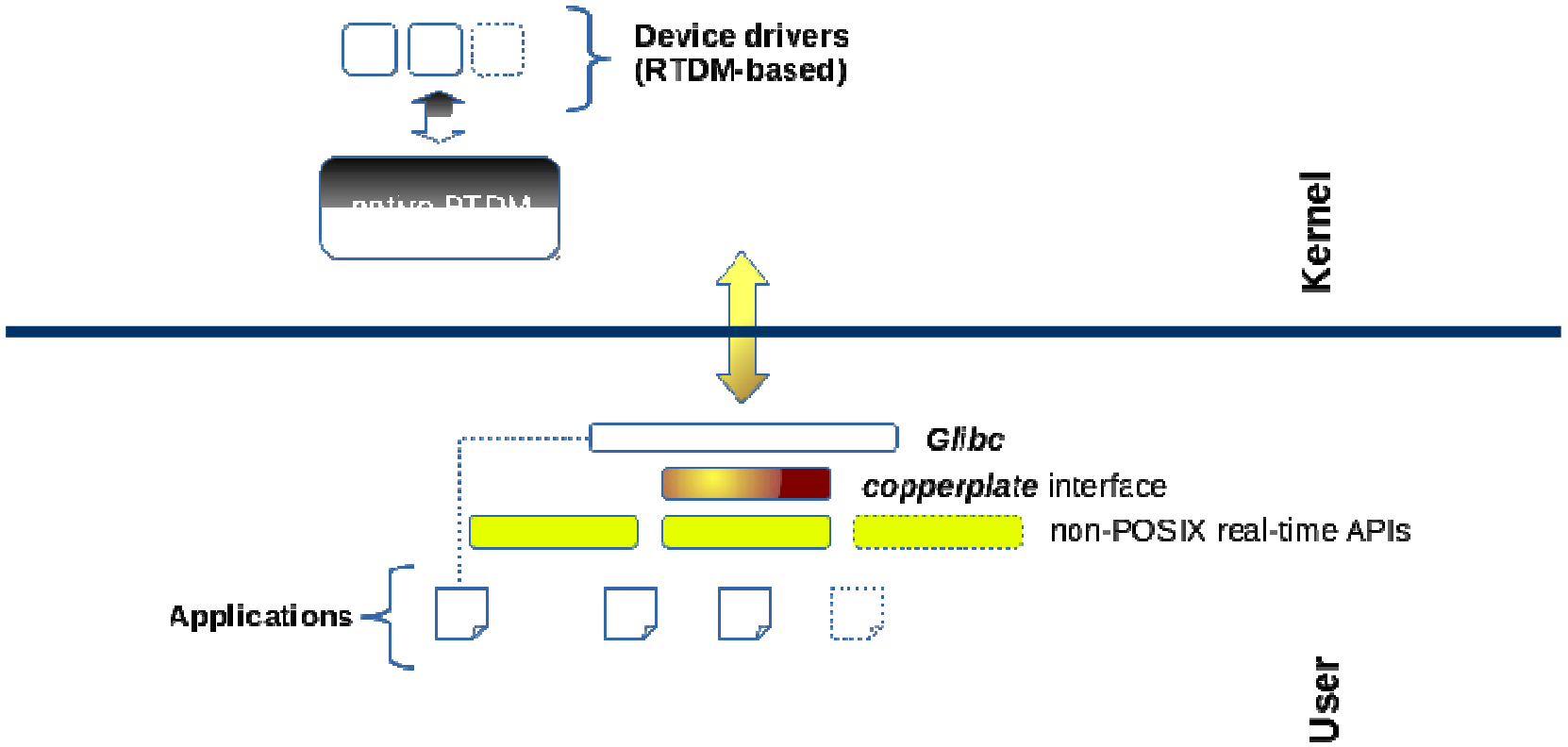
- Le co-noyau *Cobalt* intègre aussi une interface POSIX (en partie) à destination des applications de l'espace utilisateur et aussi une interface pour le développement de pilotes temps réel.

XENOMAI 3



Architecture de Xenomai 3 *Cobalt*

XENOMAI 3



Architecture de Xenomai 3 *Mercury*

XENOMAI 3

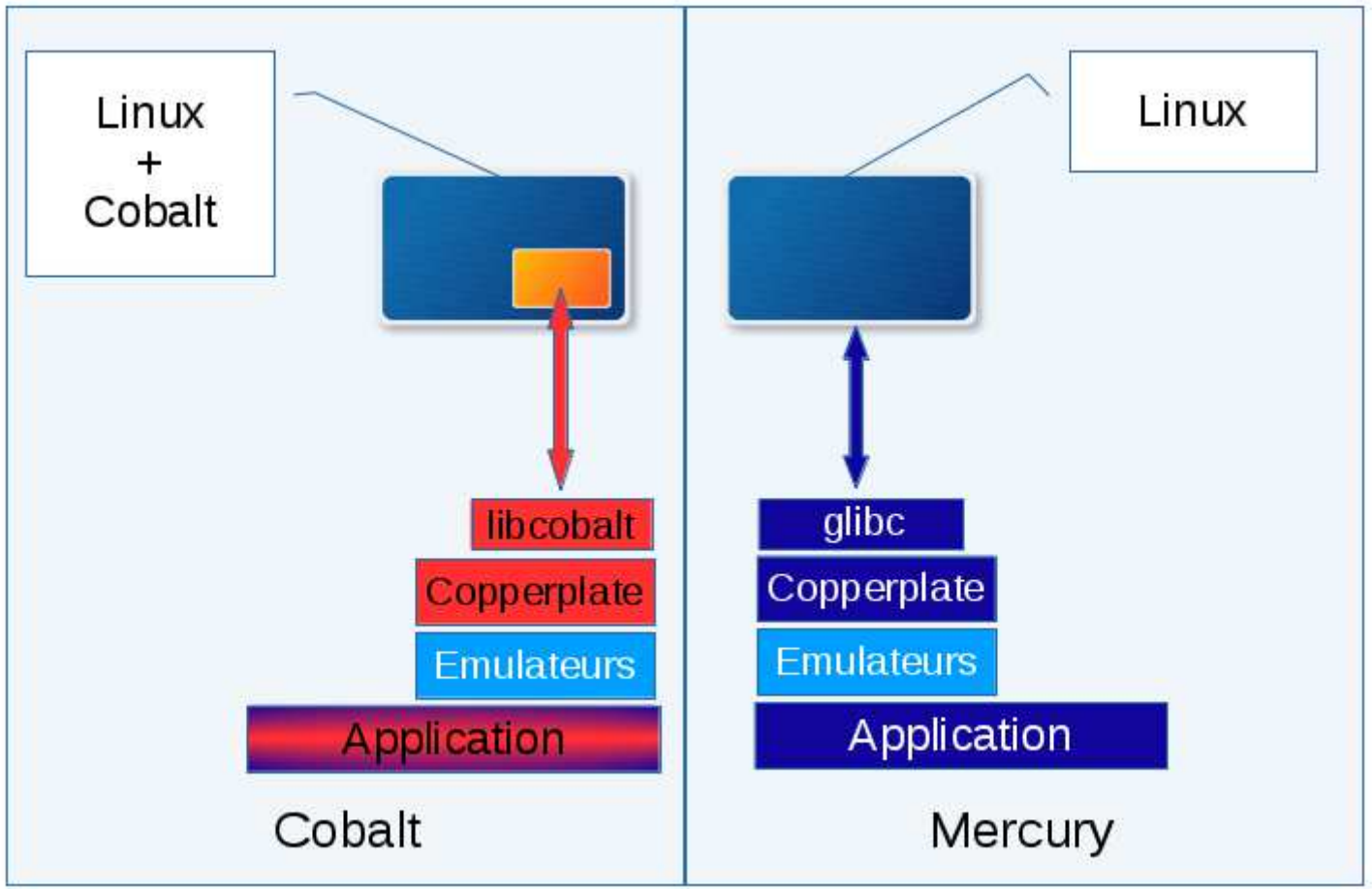
- L'interface pour le développement de pilotes Temps Réel s'appelle RTDM (*Real Time Driver Model*)
- L'API RTDM fournit un environnement pour le développement de pilotes de périphériques Temps Réel :
  - Périphérique caractère : UART, E/S…
  - Périphérique réseau : TCP/IP avec RTnet, bus CAN…
- Il y a donc dans l'espace noyau avec Xenomai 3 l'API POSIX (Xenomai *Cobalt*) et l'API RTDM…
- Toutes les autres API sont dans l'espace utilisateur implémentées au dessus d'une couche appelée *Copperplate*.

XENOMAI 3

- La couche *Copperplate* sert d'intermédiaire entre les émulateurs d'API propriétaires de RTOS et le noyau mais aussi entre les services Temps Réel fournis par le co-noyau *Cobalt*.

- Lorsqu'on utilise Xenomai *Cobalt*, *Copperplate* utilise les services POSIX du co-noyau *Cobalt* pour implémenter les briques d'émulation.

- Lorsqu'on utilise Xenomai *Mercury*, *Copperplate* utilise la bibliothèque *glibc* et donc les services POSIX natifs du noyau Linux hôte.



Rôle de *Copperplate* dans Xenomai

XENOMAI 3 COBALT

- Xenomai 3 *Cobalt* est l'environnement pour avoir un système Temps Réel dur sous Linux **à double noyau**.

- Dans ce cas, une application peut s'exécuter dans 2 modes :
  - Mode primaire : l'exécution se fait en Temps Réel par le co-noyau *Cobalt*.
  - Mode secondaire : l'exécution se fait par le noyau Linux.

- L'API de programmation conseillée étant POSIX, l'application est un *thread* POSIX.

XENOMAI 3 COBALT

- Le *thread* POSIX passe automatiquement d'un mode à l'autre en fonction des fonctions, des appels système et des exceptions levées.

- Si l'on développement un *thread* Temps Réel, passer en mode secondaire est un handicap car cela fait exploser les temps de latence surtout quand cela n'est pas voulu (mauvaise programmation).

- C'est un piège mortel !

- On verra comment essayer de l'éviter plus tard…

XENOMAI 3 COBALT

- L'API POSIX est intégrée dans la bibliothèque `libcobalt.{so,a}`.

- Toute application liée à cette bibliothèque a son *thread* principal (*main thread*) exécuté en mode primaire (*auto shadowing*).

- Comme Xenomai *Cobalt* et la bibliothèque *glibc* fournissent une API POSIX, pour sélectionner la bonne fonction POSIX de Xenomai *Cobalt*, on a un *wrapping* qui est fait automatiquement lors de la compilation avec les outils Xenomai.

XENOMAI 3 COBALT

- On a par exemple :
  - Dans le source du programme, `pthread_mutex_lock()` devient **`__wrap_`**`pthread_mutex_lock()`.

- Cela veut dire aussi que si l'on utilise une fonction POSIX qui n'est pas dans l'API POSIX de Xenomai *Cobalt*, on utilise alors la fonction de la bibliothèque *glibc* ce qui peut faire passer éventuellement le *thread* en mode secondaire.

XENOMAI 3 MERCURY

- Xenomai 3 *Mercury* est l'environnement avec un système hôte sous Linux **à simple noyau**. On peut éventuellement y ajouter le patch PREMPT-RT pour être Temps Réel mou avec les limites que l'on connaît sur les temps de latence et la QoS…

- C'est dans ce cas la migration d'applications écrites avec des API de RTOS propriétaires qui est la valeur ajoutée.

- Xenomai *Mercury* fournit les émulations suivantes :
  - **API *Alchemy*** (ex *skin* natif de Xenomai 2). L'API est aussi disponible avec Xenomai *Cobalt*.
  - VxWorks et pSOS.
  - RTDM natif en cours de développement…

PROCESSEURS SUPPORTES

- Xenomai supporte les cibles suivantes :
  - ARM 32 bits et 64 bits.
  - Blackfin.
  - PowerPC 32 bits et 64 bits.
  - x86 32 bits et 64 bits.

LICENCES LOGICIELLES

- Tout le code Xenomai s'exécutant dans le noyau est sous licence GPL v2.
- Les bibliothèques liées aux applications utilisateur sont sous licence LGPL v2.1.
- Il est préférable de développer son code source Xenomai dans l'espace utilisateur.

MISE EN ŒUVRE SUR CIBLE X86
PERFORMANCES

XENOMAI SUR X86

- Nous allons regarder la mise en œuvre de Xenomai sur un PC classique équipé de la distribution Fedora.

- La version de Xenomai utilisée ici est la version 3.

XENOMAI SUR X86

- Nous regarderons la mise en œuvre sur un PC (processeur x86) sous Linux (distribution Fedora ici) :
  - Noyau Linux+*Dovetail* version 6.12.0.
  - Xenomai 3.

- Récupération des sources du noyau Linux+*Dovetail* appelé par la suite noyau Xenomai. On supposera que par la suite, on travaille depuis son *Home Directory* :

```
$ cd $HOME
$ mkdir xenomai
$ cd $HOME/xenomai
$ wget https://source.denx.de/Xenomai/linux-dovetail/-
  /archive/v6.12.y-dovetail-rebase/linux-dovetail-
  v6.12.y-dovetail-rebase.tar.gz
```

XENOMAI SUR X86

- Décompression des sources du noyau Xenomai :

```
$ cd $HOME/xenomai
$ tar -xvzf linux-dovetail-v6.12.y-dovetail-rebase.tar.gz
```

- Récupération des sources des outils Xenomai :

```
$ cd $HOME/xenomai
$ tar -xvzf xenomai-master.tar.gz
```

- Décompression des sources des outils Xenomai :

```
$ cd $HOME/xenomai
$ tar -xvzf xenomai-master.tar.gz
```

XENOMAI SUR X86

- Création de liens symboliques :

```
$ cd $HOME/xenomai
$ ln -s xenomai-master xenomai
$ ln -s linux-dovetail-v6.12.y-dovetail-rebase linux-
  dovetail
```

- L'EXTRAVERSION du fichier *Makefile* est changée pour signifier que l'on a un noyau Xenomai :

```
$ cd $HOME/xenomai/linux-dovetail
$ nano Makefile
```

XENOMAI SUR X86

```
VERSION = 6
PATCHLEVEL = 12
SUBLEVEL = 0
EXTRAVERSION =
```

devient :

```
VERSION = 6
PATCHLEVEL = 12
SUBLEVEL = 0
EXTRAVERSION = -xenomai
```

XENOMAI SUR X86

- Application du *patch* Xenomai aux sources du noyau+*Dovetail* :

```
$ cd $HOME/xenomai/xenomai
$ ./scripts/prepare-kernel.sh --arch=x86_64 --
  linux=../linux-dovetail

$ cd $HOME/xenomai/linux-dovetail
$ make x86_64_defconfig
```

- On validera Xenomai :

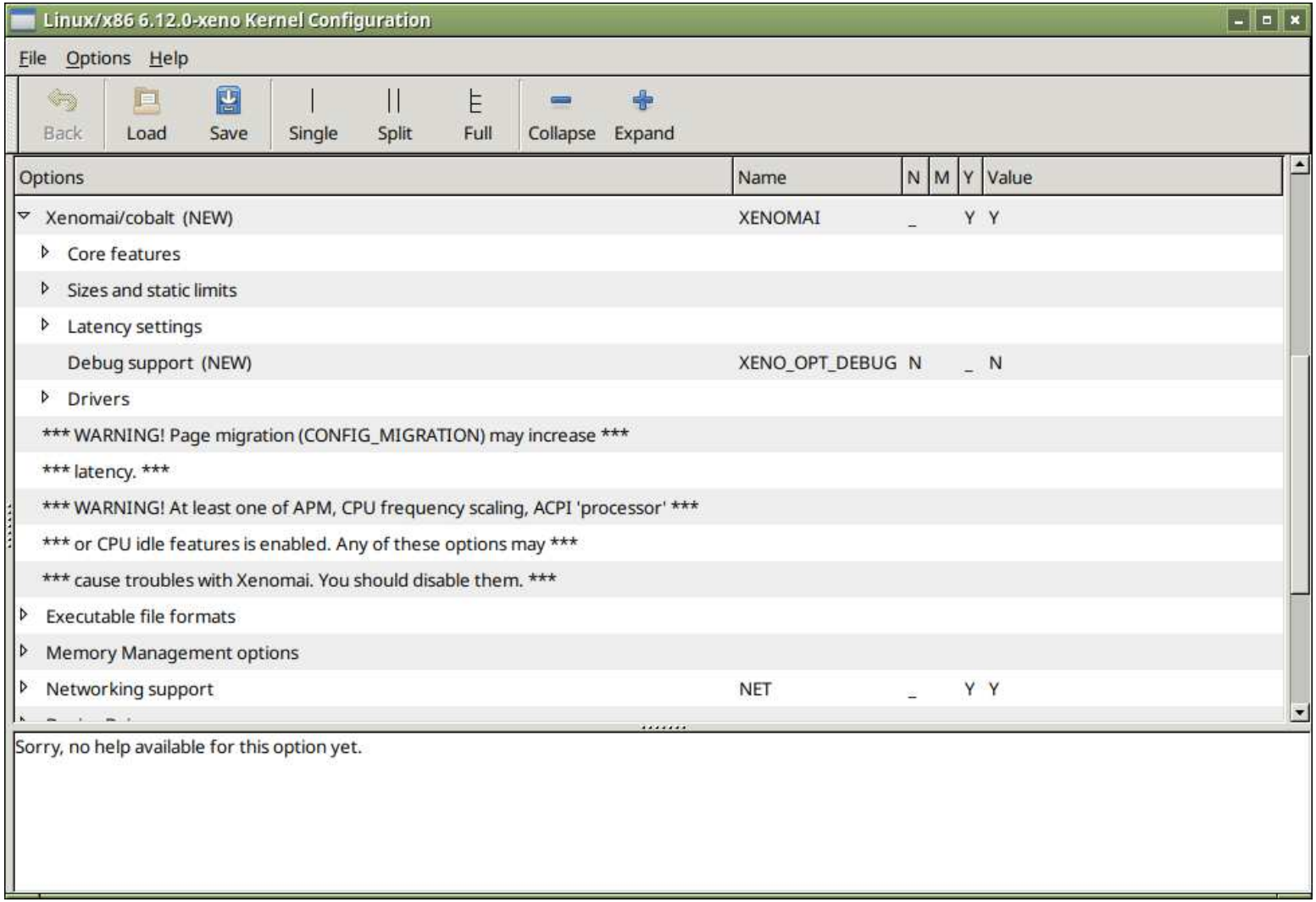
```
$ make gconfig
```

XENOMAI SUR X86

- Il faudra ensuite désactiver un certain nombre d'options :
  - APM.
  - CPU_FREQ.
  - ACPI_PROCESSOR.
  - INTEL_IDLE.

  qui posent problème avec Xenomai sur processeur x86. Voir la documentation https://v3.xenomai.org/tips/x86/common/.

XENOMAI SUR X86



XENOMAI SUR X86

- Compilation du noyau Xenomai :

```
$ cd $HOME/xenomai/linux-dovetail
$ make -j$(cat /proc/cpuinfo | grep processor | wc -l)
```

- Compilation des modules :

```
$ cd $HOME/xenomai/linux-dovetail
$ make -j$(cat /proc/cpuinfo | grep processor | wc -l)
  modules
```

XENOMAI SUR X86

- Installation du noyau Xenomai et de ses modules :

```
$ cd $HOME/xenomai/linux-dovetail
$ sudo make modules_install
$ sudo make install
$ sudo cp System.map /boot/System.map-6.12.0-xenomai
$ sudo cp .config /boot/config-6.12.0-xenomai
```

- Génération du fichier *RAM Disk* pour le noyau Xenomai. Il n'y a rien à faire !

XENOMAI SUR X86

- Compilation des outils Xenomai :

```
$ cd $HOME/xenomai/xenomai
$ ./scripts/bootstrap
$ ./configure --with-core=cobalt --enable-smp --enable-
  pshared
$ make
$ sudo make install
```

XENOMAI SUR X86

- On a au final 4 nouveaux fichiers dans le répertoire */boot/* :

```
$ ls /boot/*xenomai*
/boot/config-6.12.0-xenomai
/boot/initramfs-6.12.0-xenomai.img
/boot/System.map-6.12.0-xenomai
/boot/vmlinuz-6.12.0-xenomai
```

- Mise à jour de *grub2*. Il n'y a rien à faire !

- Redémarrage du PC et au menu d'accueil choisir le noyau *Fedora Linux (6.12.0-xenomai) 41 (Forty One).*

XENOMAI SUR X86

- On pourra alors vérifier que Xenomai est installé :

```
$ sudo dmesg|grep Xenomai
[ 1.128143] [Xenomai] scheduling class idle registered.
[ 1.128144] [Xenomai] scheduling class rt registered.
[ 1.128153] [Xenomai] SMI-enabled chipset found, but SMI
  workaround disabled
[ 1.128187] IRQ pipeline: high-priority Xenomai stage added.
[ 1.128825] [Xenomai] Cobalt v3.3 [LTRACE]
$ cat /proc/xenomai/version
3.3
```

- On pourra enfin vérifier la version du noyau :

```
$ uname -r
6.12.0-xenomai
```

XENOMAI SUR X86

- On pourra utiliser les outils de Xenomai :
  - Outil *cyclictest* : mesure du temps de latence sur des *threads* périodiques :

```
$ sudo /usr/xenomai/demo/cyclictest -n -p 99 -i 1000
```

  - Outil *latency* : mesure du temps de latence :

```
$ sudo /usr/xenomai/bin/latency -t[0,1,2] -p 1000
```

XENOMAI SUR X86

- L'environnement de mesures est le suivant :
  - PC AMD Intel i5-2500K à 3,3 GHz avec 24 Go de mémoire RAM.
  - Noyau Xenomai 6.12.0-xenomai.
  - Programme de mesure :
    - Outils Xenomai *cyclictest* et *latency*.
  - Mesures sur noyau non chargé.
  - Mesures sur noyau chargé avec le script *load*.

XENOMAI SUR X86

- Outil Xenomai *cyclictest*. Linux non chargé :

```
$ sudo/usr/xenomai/demo/cyclictest -n -p 99 -i 1000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.34 0.56 0.27 1/168 942

T: 0 (  942) P:99 I:1000 C:  54762 Min:      0 Act:    0 Avg:    0 Max:       3
```

- Outil Xenomai *cyclictest*. Linux chargé :

```
$ sudo /usr/xenomai/demo/cyclictest -n -p 99 -i 1000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 54.08 13.93 4.76 72/290 1101

T: 0 (  979) P:99 I:1000 C:  52422 Min:      0 Act:    4 Avg:    1 Max:      11
```

XENOMAI SUR X86

- Outil Xenomai *latency*. Linux non chargé :

```
$ sudo /usr/xenomai/bin/latency -t0 -p 1000
== 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|      0.109|      0.158|      1.919|       0|     0|      0.109|      1.919
. . .
RTD|      0.196|      0.216|      1.431|       0|     0|      0.109|      8.899

$ sudo /usr/xenomai/bin/latency -t1 -p 1000
== Sampling period: 1000 us
== Test mode: in-kernel periodic task
== All results in microseconds
warming up...
RTT|  00:00:01  (in-kernel periodic task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     -1.345|     -1.232|      0.200|       0|     0|     -1.345|      0.200
. . .
RTD|     -1.265|     -1.239|     -0.474|       0|     0|     -1.345|      0.509
```

XENOMAI SUR X86

- Outil Xenomai *latency*. Linux chargé :

```
$ sudo /usr/xenomai/bin/latency -t0 -p 1000
== 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|      0.190|      2.248|      7.456|       0|     0|      0.190|      7.456

RTD|      0.123|      1.874|      4.725|       0|     0|      0.119|      8.449

$ sudo /usr/xenomai/bin/latency -t2 -p 1000
== Sampling period: 1000 us
== Test mode: in-kernel periodic task
== All results in microseconds
warming up...
RTT|  00:00:01  (in-kernel periodic task, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD|     -1.204|      0.355|      4.860|       0|     0|     -1.204|      4.860
RTD|     -1.015|     -0.204|      1.258|       0|     0|     -1.272|      5.639
```

MISE EN ŒUVRE SUR CIBLE ARM PERFORMANCES

XENOMAI SUR ARM

- Nous regarderons la mise en œuvre sur une carte Raspberry Pi 3B sous Linux.
  - Noyau Linux version 4.19.82.
  - Xenomai 3.1 (Xenomai 3).

- Le *bootloader* u-boot est installé sur la carte SD de la carte RPi (non décrit ici).

- Le noyau Linux Xenomai et le système de fichiers *root* sont construits à la main comme présentés en TP. La compilation est ici une compilation croisée.

XENOMAI SUR ARM

- Carte Raspberry Pi :
  - http://www.raspberrypi.org
  - Carte comprenant :
    - SoC Broadcom avec processeur Cortex et processeur graphique propriétaire.
    - Jusqu'à 8 Go de RAM, carte SD ou microSD.
    - Modèles Zero, A, A+, B, B+, 2B, **3B**, 3B+,4B, 5B.
    - 55 € - 120 €.

XENOMAI SUR ARM

- Récupération des sources du noyau *vanilla* depuis un site miroir. On supposera que par la suite, on travaille depuis son *Home Directory* :

```
$ cd
$ wget
  https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/l
  inux-4.19.82.tar.gz
```

XENOMAI SUR ARM

- Décompression des sources du noyau *vanilla* :

```
$ tar -xvzf linux-4.19.82.tar.gz
```

- Récupération des sources de Xenomai :

```
$ wget
  https://xenomai.org/downloads/xenomai/stable/xenomai-
  3.1.tar.bz2
```

- Récupération du *patch I-pipe* :

```
$ wget
  https://xenomai.org/downloads/ipipe/v4.x/arm/ipipe-
  core-4.19.82-arm-6.patch
```

XENOMAI SUR ARM

- Décompression des sources de Xenomai :

```
$ tar -xvjf xenomai-3.1.tar.bz2
```

- Application du *patch I-pipe* aux sources du noyau *vanilla* :

```
$ cd

$ cd xenomai-3.1
$ ./scripts/prepare-kernel.sh --arch=arm \
--ipipe=../ipipe-core-4.19.82-arm-6.patch \
--linux=../linux-4.19.82
. . .
```

XENOMAI SUR ARM

- Configuration du noyau :

```
$ cd
$ cd linux-4.19.82
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
  bcm2709_defconfig
```

- Il faut noter que la configuration du noyau est modifiée pour valider notamment l'accès à l'interface USB Ethernet de la carte RPi (voir TP).

XENOMAI SUR ARM

- Compilation du noyau Xenomai :

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi-
  zImage modules dtbs -j8
```

- On pourra utiliser les outils de Xenomai :

```
$ ./configure CFLAGS="-march=armv7-a -msoft-float"
  LDFLAGS="-march=armv7-a" --enable-smp -host=arm-none-
  linux-gnueabi CC=arm-none-linux-gnueabi-gcc AR=arm-
  none-linux-gnueabi-ar LD=arm-none-linux-gnueabi-ld
$ make
```

- Les outils de Xenomai sont installés dans le système de fichiers *root* (voir TP) :

```
$ make -k install DESTDIR=../ramdisk/root_fs
```

XENOMAI SUR ARM

- Le fichier *zImage* du noyau Linux Xenomai sera ensuite téléchargé en RAM par *tftp* avec *u-boot*.
- Il en est de même du système de fichiers *root*.
- On utilisera ensuite une commande *u-boot* pour lancer le noyau Linux Xenomai (voir TP).

XENOMAI SUR ARM

- On pourra alors vérifier que Xenomai est installé :

```
RPi3# dmesg |grep I-pipe
I-pipe domain: (null)
I-pipe, 1.000 MHz clocksource, wrap in 4294967 ms
I-pipe, 19.200 MHz clocksource, wrap in 960767920505705
  ms
I-pipe: disabling SMP code
I-pipe: head domain Xenomai registered.

RPi3# dmesg |grep Xenomai
[Xenomai] scheduling class idle registered.
[Xenomai] scheduling class rt registered.
I-pipe: head domain Xenomai registered.
[Xenomai] Cobalt v3.1
```

XENOMAI SUR ARM

- Outil *cyclictest* : mesure du temps de latence sur des *threads* périodiques :

```
# /usr/xenomai/demo/cyclictest -n -p 99 -i 1000
```

- Outil *latency* : mesure du temps de latence :

```
# /usr/xenomai/bin/latency -t[0,1,2] -p 1000
```

XENOMAI SUR ARM

- L'environnement de mesures est le suivant :
  - Carte Raspberry Pi 3B avec 1 Go de RAM.
  - Noyau Xenomai 4.19.82.
  - Programme de mesure :
    - Outils standards *cyclictest*.
    - Outils Xenomai *cyclictest* et *latency*.
  - Mesures sur noyau non chargé.
  - Mesures sur noyau chargé.

XENOMAI SUR ARM

- Pour stresser le noyau, on utilisera l'utilitaire *stress* :

```
RPi3# stress -c 50 -i 50 &
```

- On dévalidera le *throttling* :

```
RPi3:# echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

- On dévalidera l'anticipation sur la latence minimale de Xenomai :

```
RPi3:# echo 0 > /proc/xenomai/latency
```

XENOMAI SUR ARM

- Outil standard *cyclictest*. Linux non chargé :

```
RPi3# cyclictest -n -p 99 -i 1000
# /dev/cpu_dma_latency set to 0us
WARN: Running on unknown kernel version...YMMV
policy: fifo: loadavg: 0.00 0.00 0.00 1/39 88


T: 0 ( 87) P:99 I:1000 C: 302273 Min: 27 Act: 34 Avg: 45 Max: 132
```

- Outil standard *cyclictest*. Linux chargé :

```
RPi3# cyclictest -n -p 99 -i 1000
# /dev/cpu_dma_latency set to 0us
WARN: Running on unknown kernel version...YMMV
policy: fifo: loadavg: 100.22 65.65 28.99 55/140 193
T: 0 ( 192) P:99 I:1000 C: 302412 Min: 32 Act: 37 Avg: 39 Max: 150
```

XENOMAI SUR ARM

- Outil Xenomai *cyclictest*. Linux non chargé :

```
RPi3# /usr/xenomai/demo/cyclictest -n -p 99 -i 1000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.08 0.03 0.00 1/40 85


T: 0 ( 84) P:99 I:1000 C: 302091 Min: 5 Act: 10 Avg: 8 Max: 16
```

- Outil Xenomai *cyclictest* . Linux chargé :

```
RPi3# /usr/xenomai/demo/cyclictest -n -p 99 -i 1000
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 100.57 89.97 51.77 101/141 197

T: 0 ( 196) P:99 I:1000 C: 307345 Min: 5 Act: 9 Avg: 9 Max: 17
```

- Outil Xenomai *latency*. Linux non chargé :

```
# /usr/xenomai/bin/latency -t0 -p 1000
== 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| 10.312| 10.648| 12.239| 0| 0| 10.312| 12.239
. . .
RTD| 10.342| 10.633| 14.561| 0| 0| 10.061| 17.292

# /usr/xenomai/bin/latency -t2 -p 1000
== Sampling period: 1000 us
== Test mode: in-kernel timer handler
== All results in microseconds
warming up...
RTT| 00:00:01 (in-kernel timer handler, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD| 2.291| 2.415| 2.812| 0| 0| 2.291| 2.812
. . .
RTD| 2.102| 2.431| 2.779| 0| 0| 0.631| 4.659
```

- Outil Xenomai *latency*. Linux chargé :

```
# /usr/xenomai/bin/latency -t0 -p 1000
== 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| 10.521| 11.370| 13.646| 0| 0| 10.521| 13.646
. . .
RTD| 10.398| 11.317| 13.315| 0| 0| 7.562| 21.069

# /usr/xenomai/bin/latency -t2 -p 1000
== Sampling period: 1000 us
== Test mode: in-kernel timer handler
== All results in microseconds
warming up...
RTT| 00:00:01 (in-kernel timer handler, 1000 us period, priority 99)
RTH|----lat min|----lat avg|----lat max|-overrun|---msw|---lat best|--lat worst
RTD| 2.083| 2.445| 3.645| 0| 0| 2.083| 3.645
. . .
RTD| 2.063| 2.376| 3.730| 0| 0| 0.606| 5.296
```

XENOMAI SUR ARM

- On utilise maintenant des outils graphiques basés sur l'outil *gnuplot*.
- Nous allons uniquement exploiter *cyclictest*.
- Nous aurons ainsi 2 types de graphiques :
  - L'histogramme : ce graphique donne le nombre de fois que l'on obtient un temps de latence donné sur la durée de la mesure.
  - La latence : ce graphique donne l'évolution du temps de latence au cours du temps.

XENOMAI SUR ARM

- La mesure durera 5 minutes. Est-ce suffisant ?

- On utilisera l'outil *cyclictest* standard :

```
RPi3:# cyclictest -l 300000 -n -m -p 99 -i 1000 -v >
ons.log
```

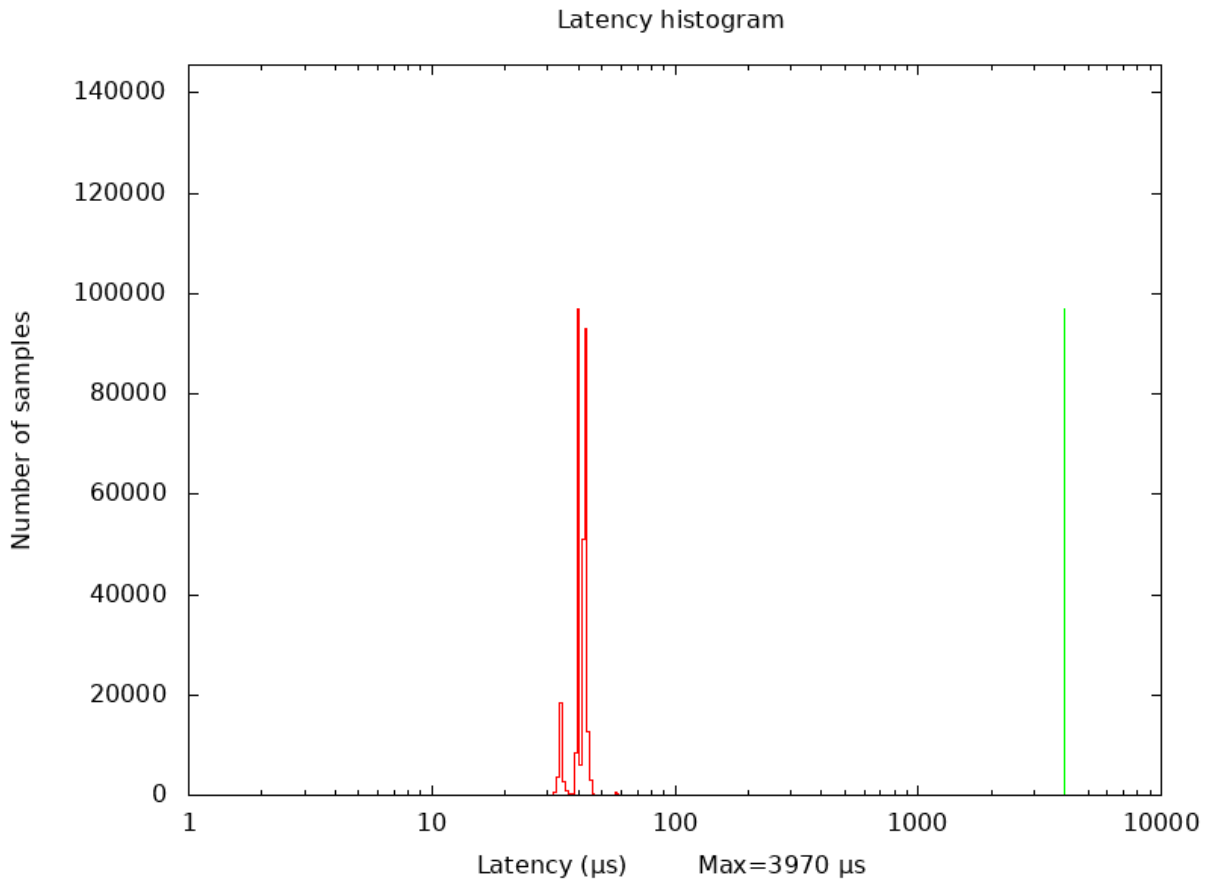
- On utilisera l'outil *cyclictest* Xenomai :

```
RPi3:# /usr/xenomai/demo/cyclictest -l 300000 -n -m -p 99
-i 1000 -v > onx.log
```

- Les fichiers `.log` générés sont ensuite traités par 2 *scripts* maison *gohist* et *golat* :

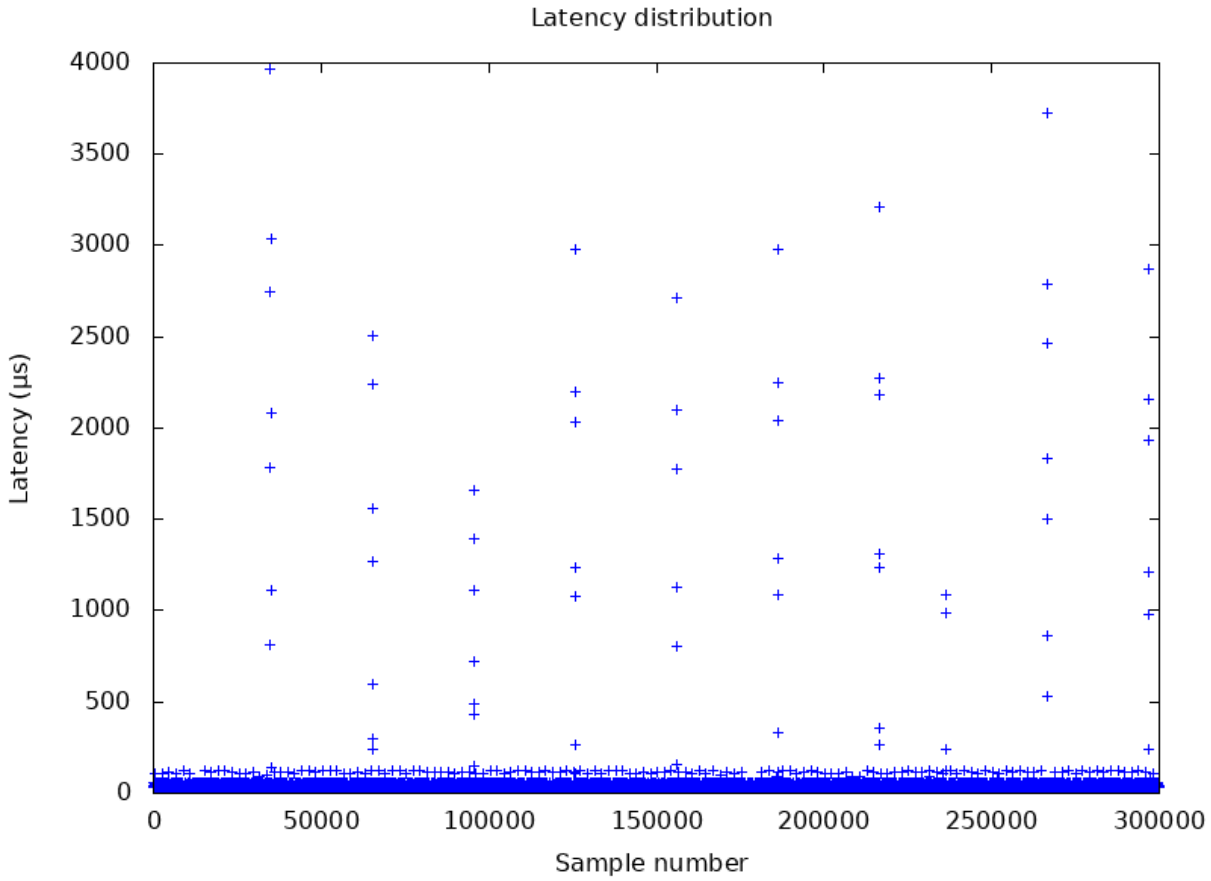
```
$ gohist ons.log
$ golat ons.log
```

XENOMAI SUR ARM



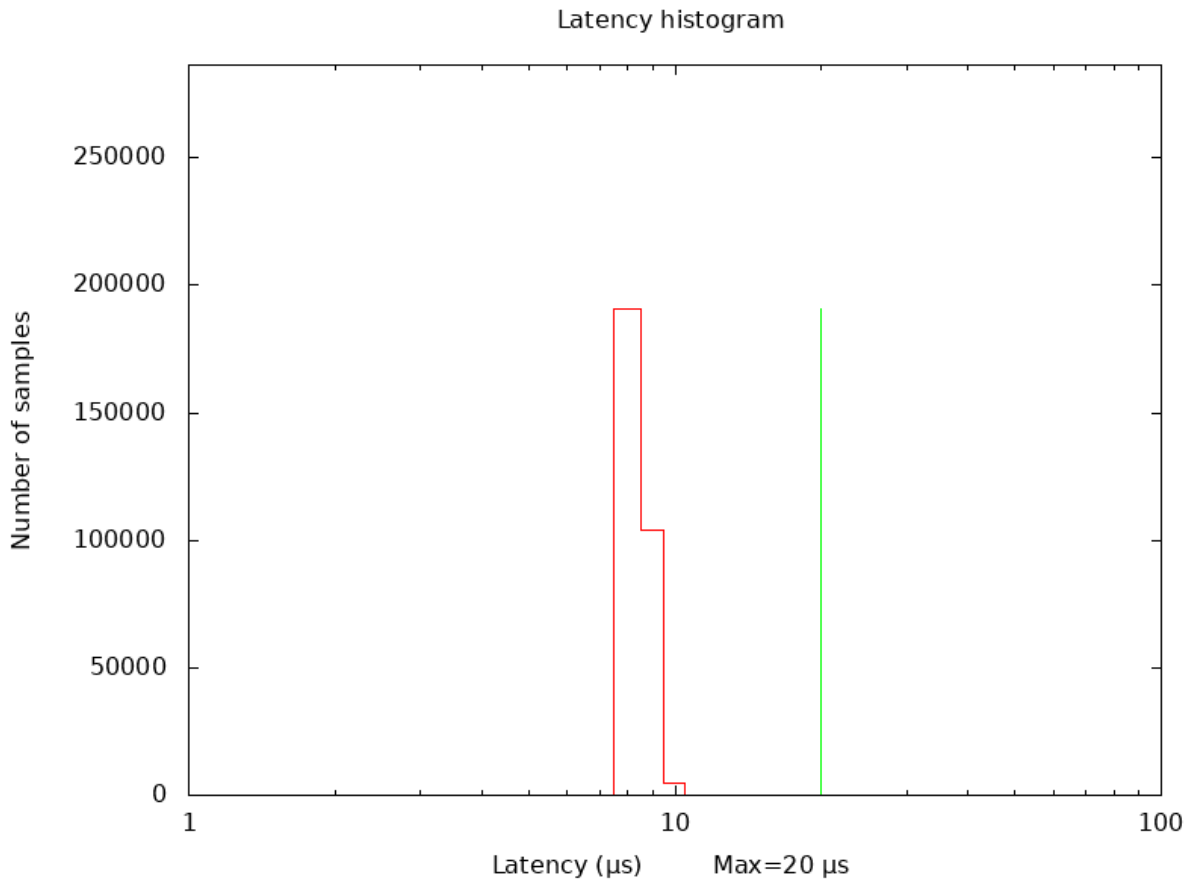
L cyclictest standard sur Linux Xenomai 4.19 non chargé

XENOMAI SUR ARM



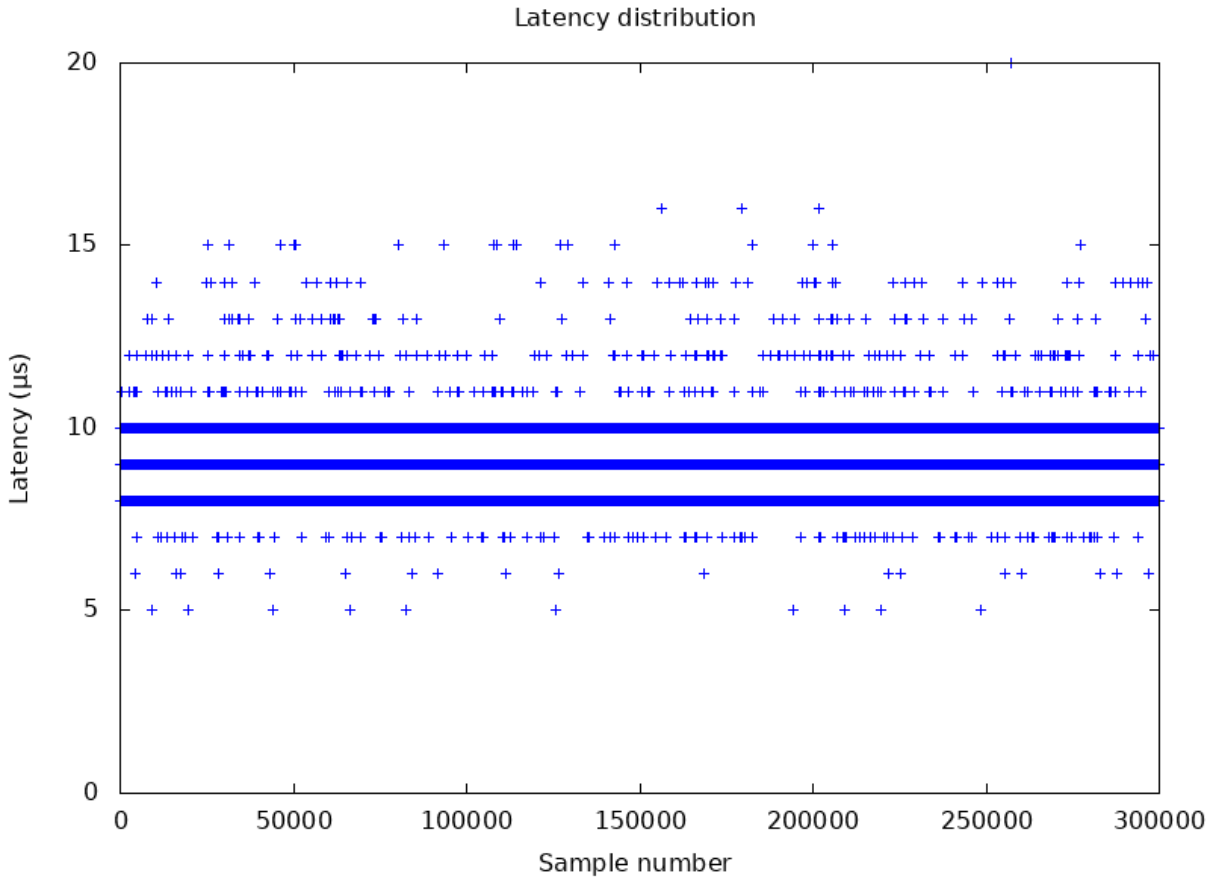
H cyclictest standard sur Linux Xenomai 4.19 non chargé

XENOMAI SUR ARM



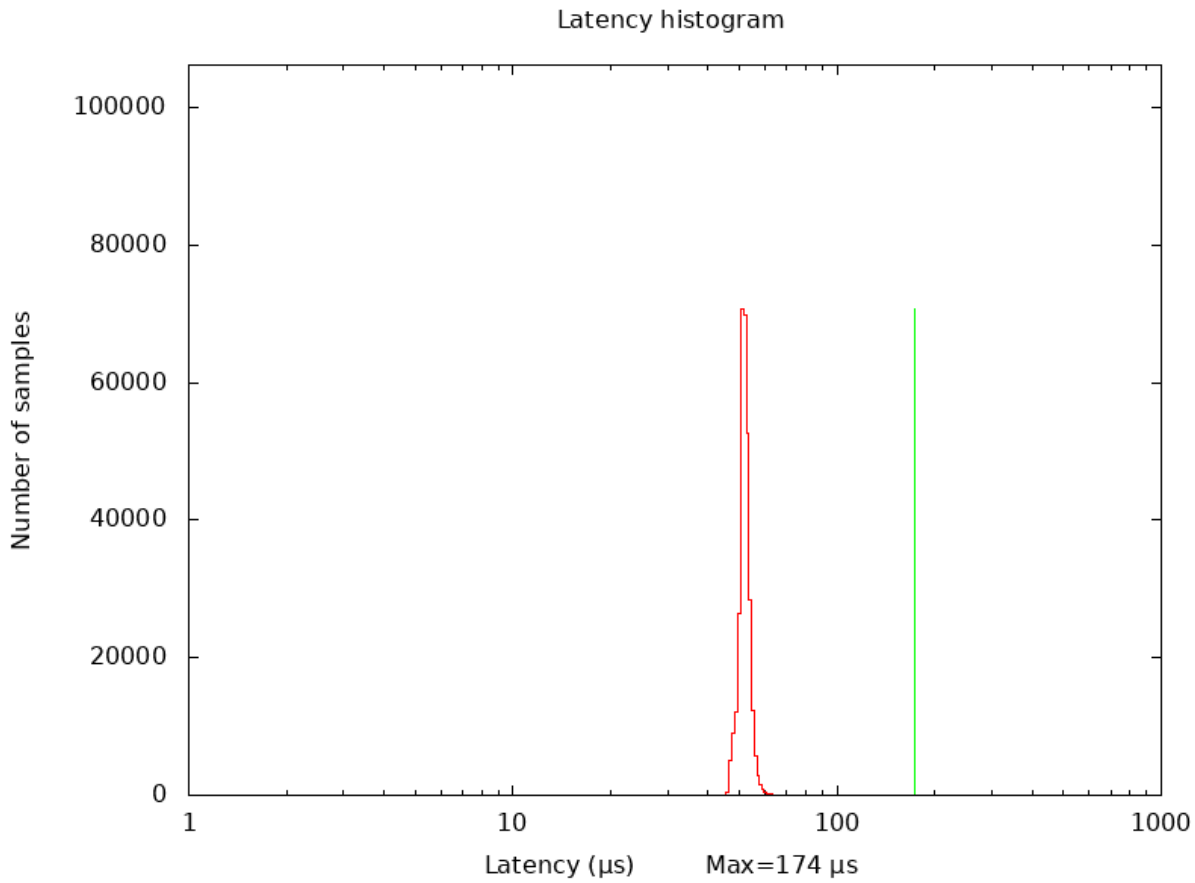
L cyclictest Xenomai sur Linux Xenomai 4.19 non chargé

XENOMAI SUR ARM



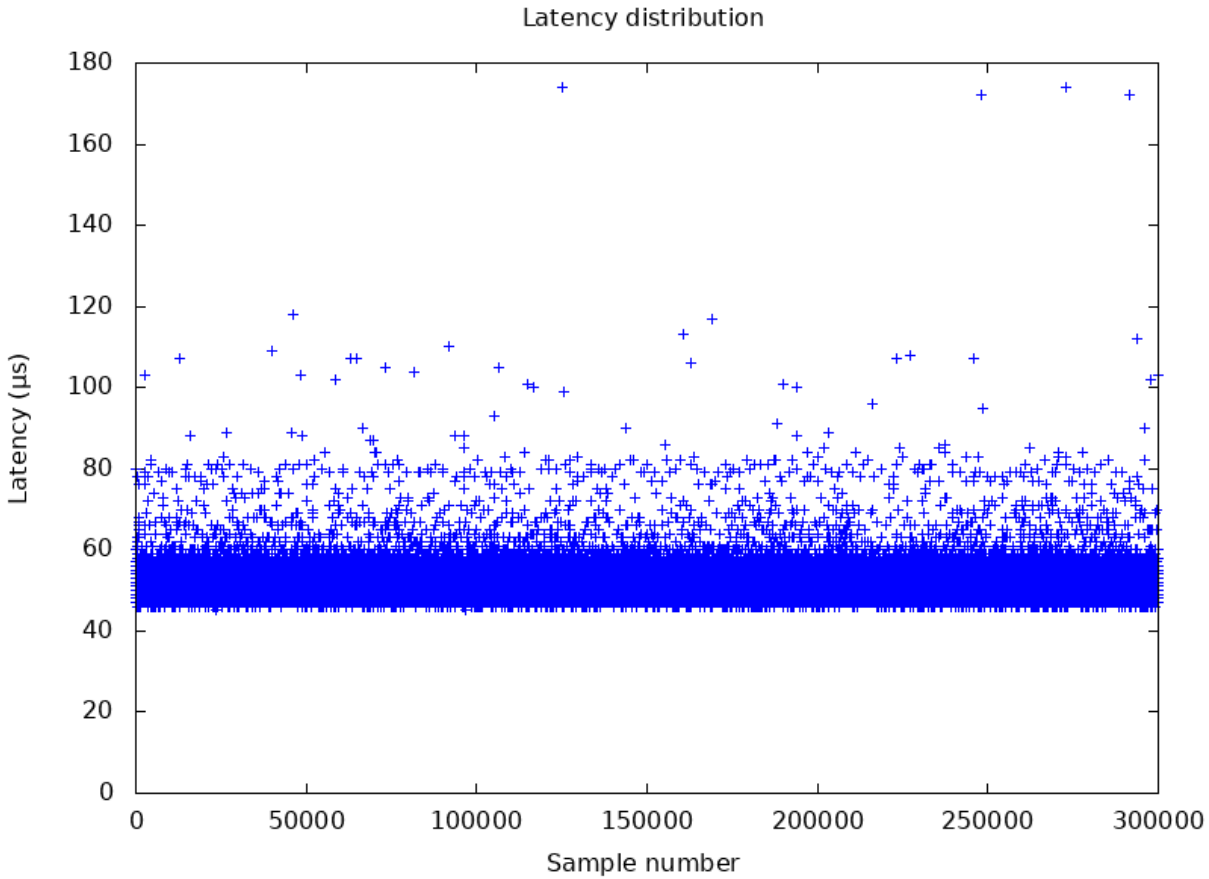
H cyclictest Xenomai sur Linux Xenomai 4.19 non chargé

XENOMAI SUR ARM



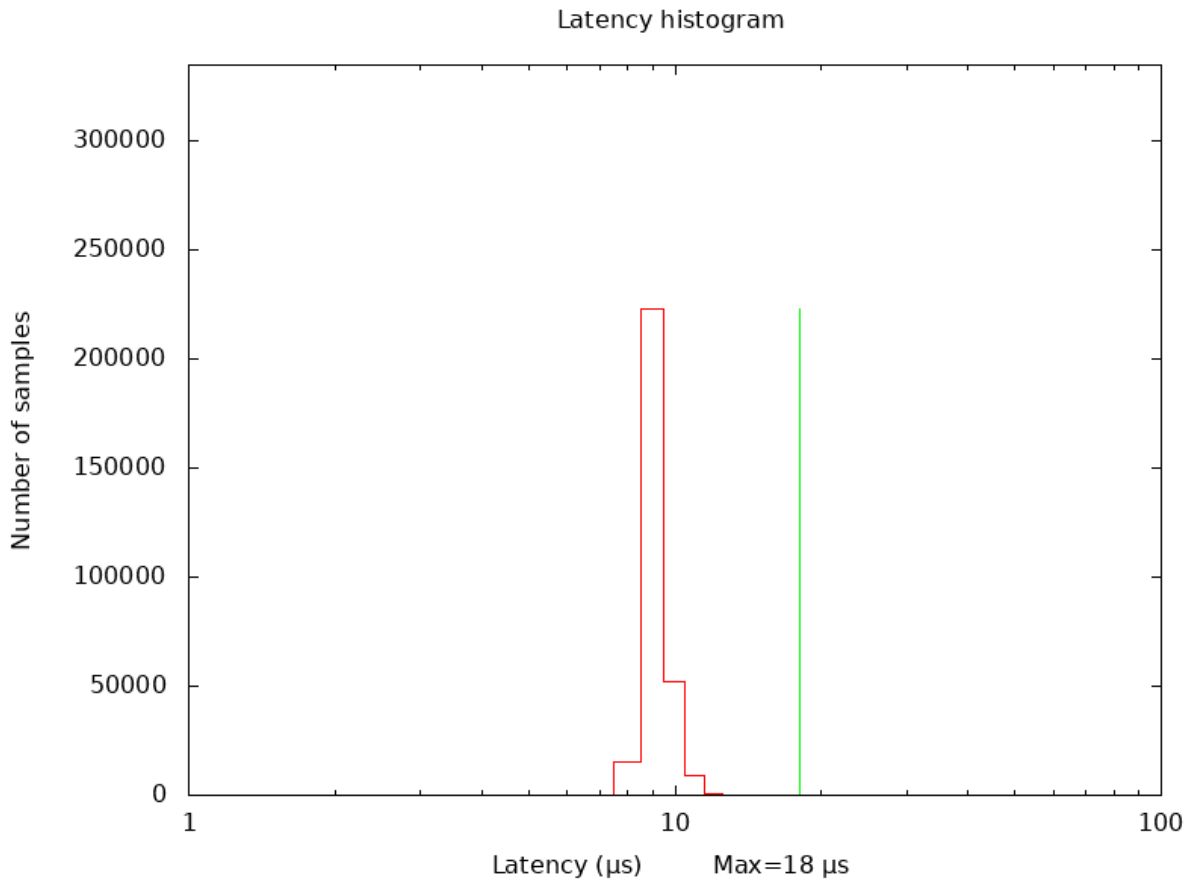
L cyclictest standard sur Linux Xenomai 4.19 chargé

XENOMAI SUR ARM



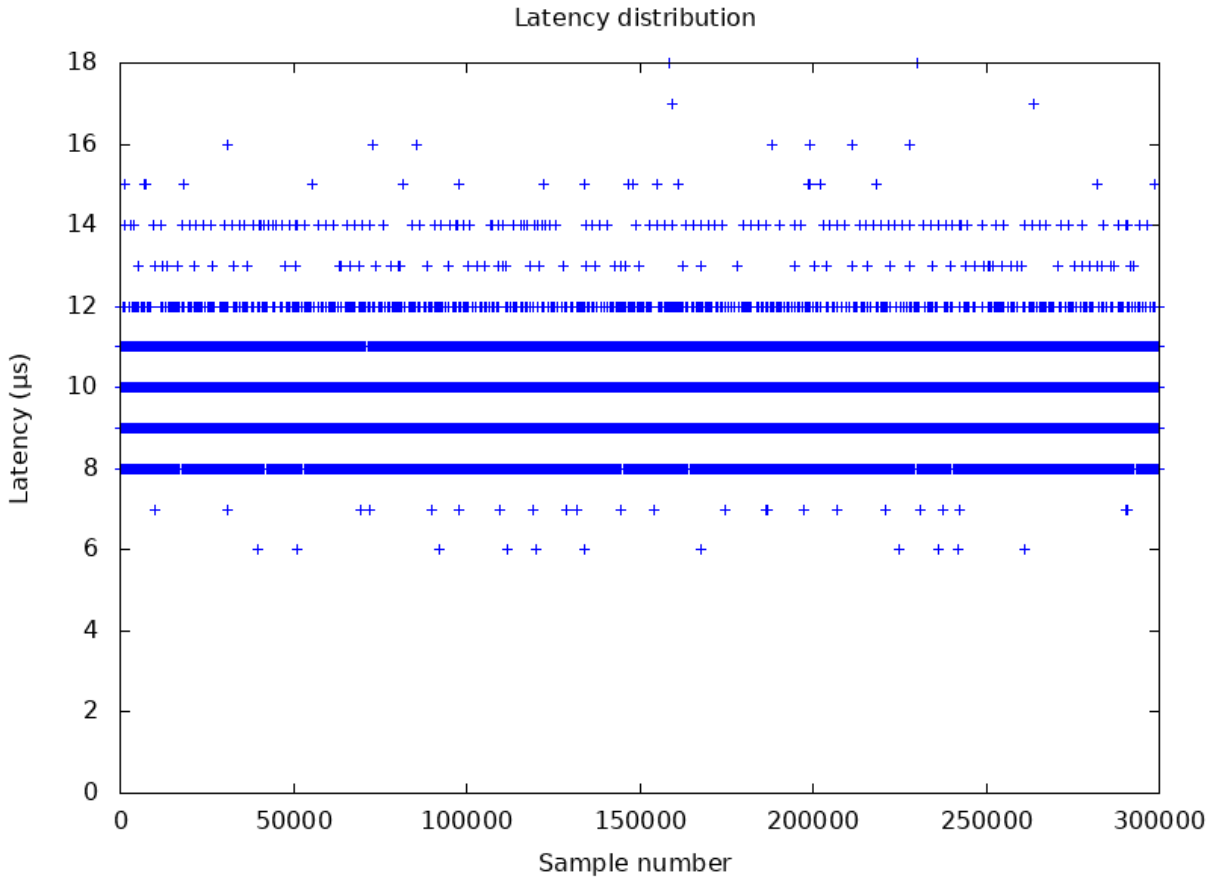
H cyclictest standard sur Linux Xenomai 4.19 chargé

XENOMAI SUR ARM



L cyclictest Xenomai sur Linux Xenomai 4.19 chargé

XENOMAI SUR ARM



H cyclictest Xenomai sur Linux Xenomai 4.19 chargé

XENOMAI SUR ARM

- On reprend le principe de la deuxième expérience.

- Génération d'un signal périodique sur une broche PIN_11 (LED 6, voir TP) du port GPIO d'une carte Raspberry Pi à l'aide d'une tâche périodique de demi-période de 50 ms.

- L'environnement de mesures est le suivant :
  - Carte Raspberry Pi 3B avec 1 Go de RAM.
  - Noyau Xenomai 4.19.82.
  - Programme de mesure :
    - Fichier C *posix.c* (l'API POSIX sera détaillée plus tard).
  - Mesures sur noyau chargé avec l'outil *stress*.

Fichier C *posix.c* :

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <pthread.h>
#include <sched.h>

#include <alchemy/task.h>
#include "bcm2835.h"
#include "bcm2835.c"
#include "bsp.h"
#include "bsp.c"
```

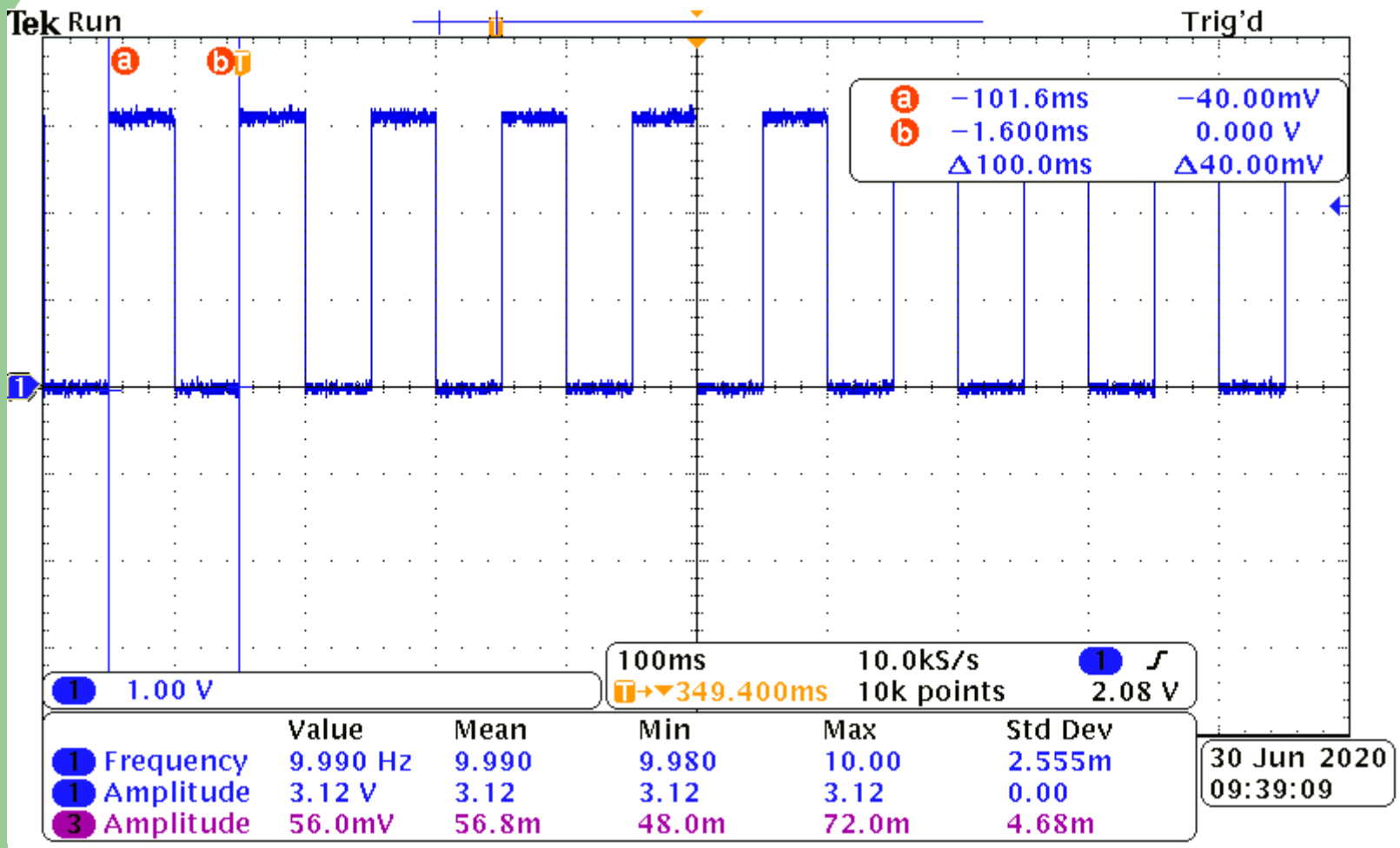
XENOMAI SUR ARM

```
pthread_attr_t attr_thread1;
pthread_t thread1;
#define PRIO1 99
#define DEMI_PERIODE 50000000

struct sched_param param;

void *task1() {
  struct timespec ts;

  ts.tv_sec = 0; // En s
  ts.tv_nsec = DEMI_PERIODE; // En ns

  while(1) {
    BSP_clrLED(6);
    clock_nanosleep(CLOCK_MONOTONIC, 0, &ts, NULL);
    BSP_setLED(6);
    clock_nanosleep(CLOCK_MONOTONIC, 0, &ts, NULL);
  }
}
```

XENOMAI SUR ARM

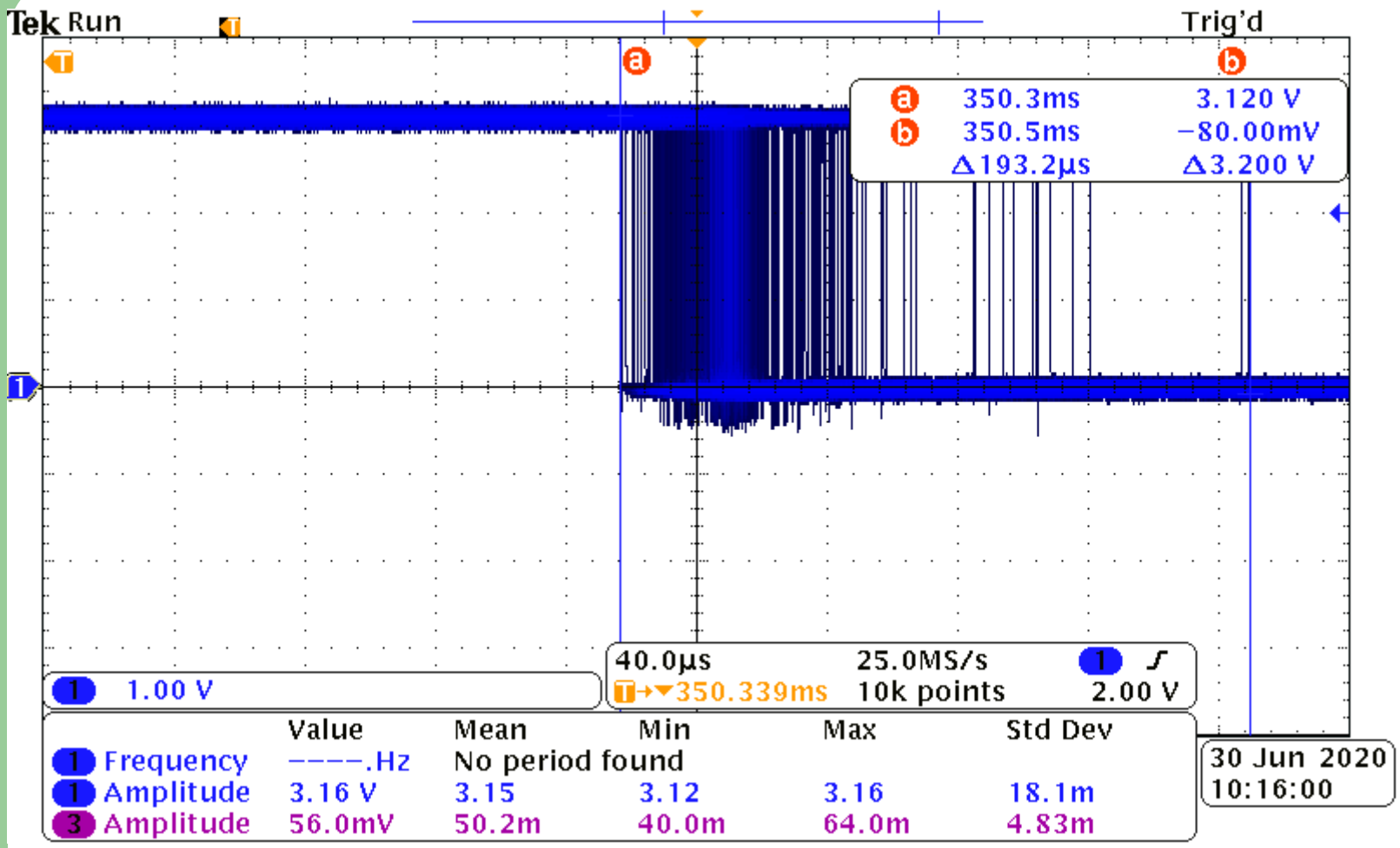
```
int main() {

  mlockall(MCL_CURRENT|MCL_FUTURE);

  BSP_init();

  pthread_attr_init (&attr_thread1);
  pthread_attr_setinheritsched(&attr_thread1, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy(&attr_thread1, SCHED_FIFO);

  param.sched_priority = PRIO1;
  pthread_attr_setschedparam(&attr_thread1, &param);

  pthread_create(&thread1, &attr_thread1, task1, NULL);

  pthread_join(thread1, NULL);

  BSP_release();
  exit(0);
}
```

XENOMAI SUR ARM



Linux Xenomai 4.19 chargé

XENOMAI SUR ARM



Linux Xenomai 4.19 chargé

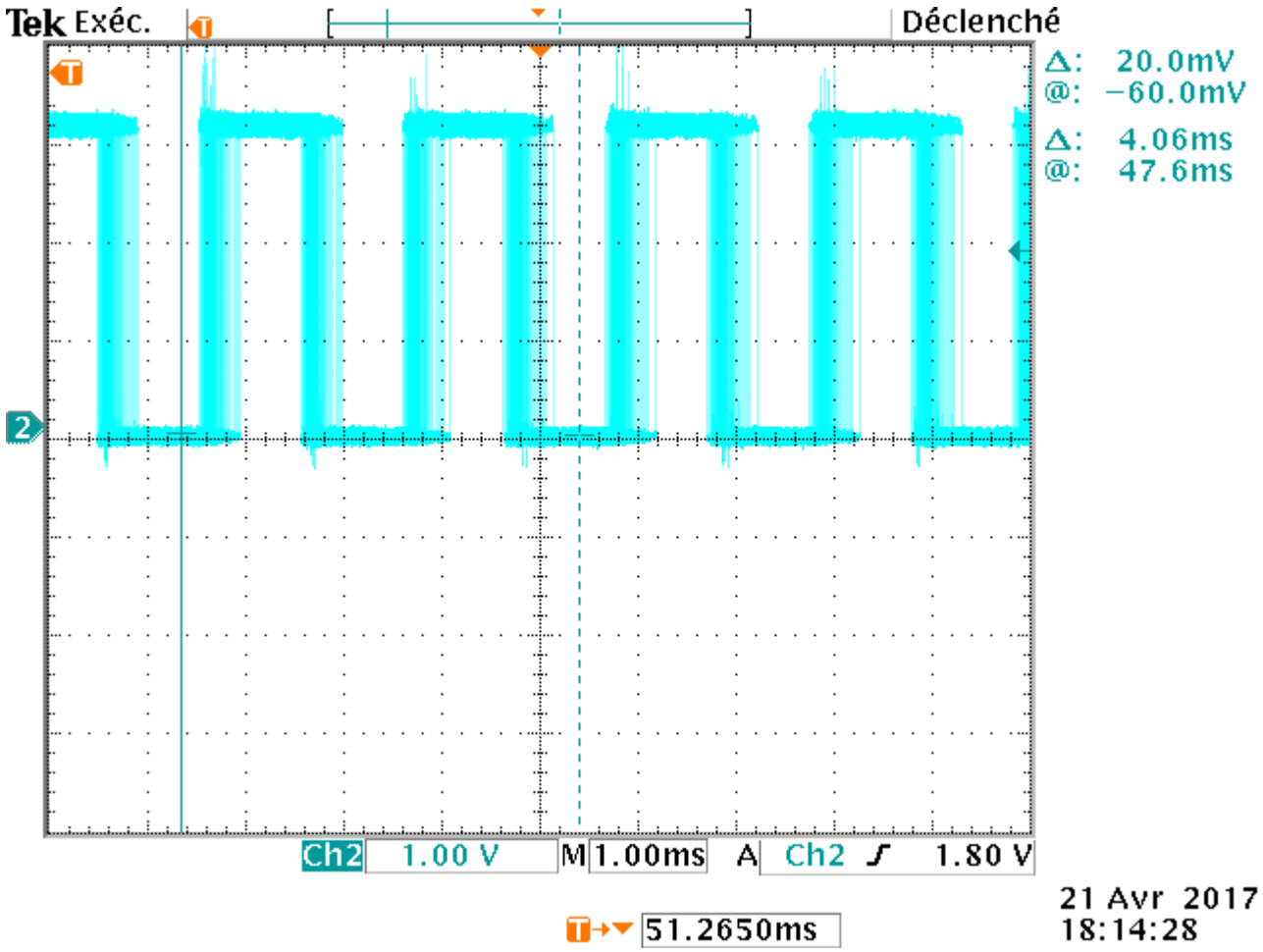
XENOMAI SUR ARM

- Le signal reste ici de forme carrée au cours du temps.

- On observe maintenant une gigue maximale sur la durée de l'expérience de 193 µs. Si l'on admet une erreur de 1% soit 1 ms pour une période de 100 ms, Linux Xenomai 4.19 génère correctement ce signal.

- Si l'on diminue la valeur de la demi-période, en dessous de 1 ms, on observe un glissement du signal…

XENOMAI SUR ARM



Linux Xenomai 4.19 chargé (période 2ms)

BILAN

- Les résultats donnés sont à interpréter **avec soin** et à considérer de **façon relative et non absolue**.
- Xenomai (Temps Réel dur) donne de bien meilleurs résultats que PREEMPT-RT (Temps Réel mou).
- Si l'on choisit Xenomai, les résultats sont meilleurs en mode noyau qu'en mode utilisateur.

CHAPITRE 7 :
LA PROGRAMMATION TEMPS REEL

PROGRAMMATION TEMPS REEL

- La programmation Temps Réel ressemble fort à la programmation informatique standard mais :
  - Le programmeur reste attentif à la quantité de ressources informatiques consommées (mémoire, temps CPU) car l'informatique Temps Réel opère généralement dans un environnement embarqué contraint voire très contraint.
  - La gestion du Temps est primordiale. C'est même une obsession en plus des communications interprocessus IPC (*Inter Processus Communication*).

- En conséquence, des règles de bon sens s'appliquent que l'on verra en Travaux Pratiques (TP).

PROGRAMMATION TEMPS REEL

- On écrira des programmes les plus simplement possibles sans expressions alambiquées.
- La simplicité et la lisibilité du code source est un gage de robustesse. De l'*Extrem Programming* avant l'heure…
- Cela veut dire que si l'on est brouillon, son code source a de fortes chances de l'être aussi… Soyons lucide !

PROGRAMMATION TEMPS REEL

- Plus pratiquement, on s'appuiera sur les facilités du multitâche du noyau Temps Réel.

- Donc il faut écrire des tâches Temps Réel qui communiquent entre elles avec des mécanismes IPC standards :
  - Sémaphore ou *mutex* (*MUtual EXclusion*).
  - File de messages.
  - Mémoire partagée.

PROGRAMMATION TEMPS REEL

- Si son application Temps Réel composée de tâches Temps Réel est bien conçue, elle doit passer son temps à dormir ou être bloquée sur l'attente d'un événement extérieur comme une interruption.

- On ne doit **jamais** avoir une boucle d'attente du style :

```
For(i=0; i<100000; i++)
    ;
```

- On utilisera plutôt une primitive d'endormissement de la tâche ou bien on armera un *timer* de durée égale au temps d'attente.

PROGRAMMATION TEMPS REEL

- On aura donc un système avec des tâches en attente et des routines d'interruption.

- On évitera si l'on peut la scrutation active (*polling*) au profit d'une gestion par interruption.

- En cas d'occurrence d'un événement extérieur sous forme d'un interruption, l'ISR fera le minimum vital :
- *1. Top Half* (au sens Linux) :
  - Acquittement de la source d'interruption en remettant à 0 ou à 1 un bit dans le registre d'état du périphérique qui a généré l'interruption.
  - Réveil d'une tâche bloquée en attente par libération d'un sémaphore ou par l'envoi d'un message dans une file de messages.
- *2. Bottom Half* :
  - Traitement des données reçues hors contexte d'interruption (pas dans l'ISR !) par la tâche précédemment réveillée.

- On fera une initialisation des variables par instruction au lieu de façon statique :

On écrira plutôt :

```
char c;
. . .
c = 0;
```

Que :

```
char c = 0;
```

PROGRAMMATION TEMPS REEL

- Dans le cas de l'usage d'un OS Temps Réel à double noyau comme Xenomai *Cobalt*, il faudra scinder en deux son application Temps Réel :
  - Une partie Temps Réel dur.
  - Une partie non Temps Réel.

- On mettra au milieu des mécanismes IPC : file de messages, signaux Temps Réel fournis par Xenomai.

PROGRAMMATION TEMPS REEL

- On se méfiera des appels système Linux dans la partie Temps Réel. Ils risquent de n'être pas Temps Réel.

- On se méfiera des *drivers* Linux. On utilisera plutôt les *drivers* Xenomai RTDM s'ils existent.

BILAN

- Je viens d'édicter quelques règles de bon sens issues de mon expérience.

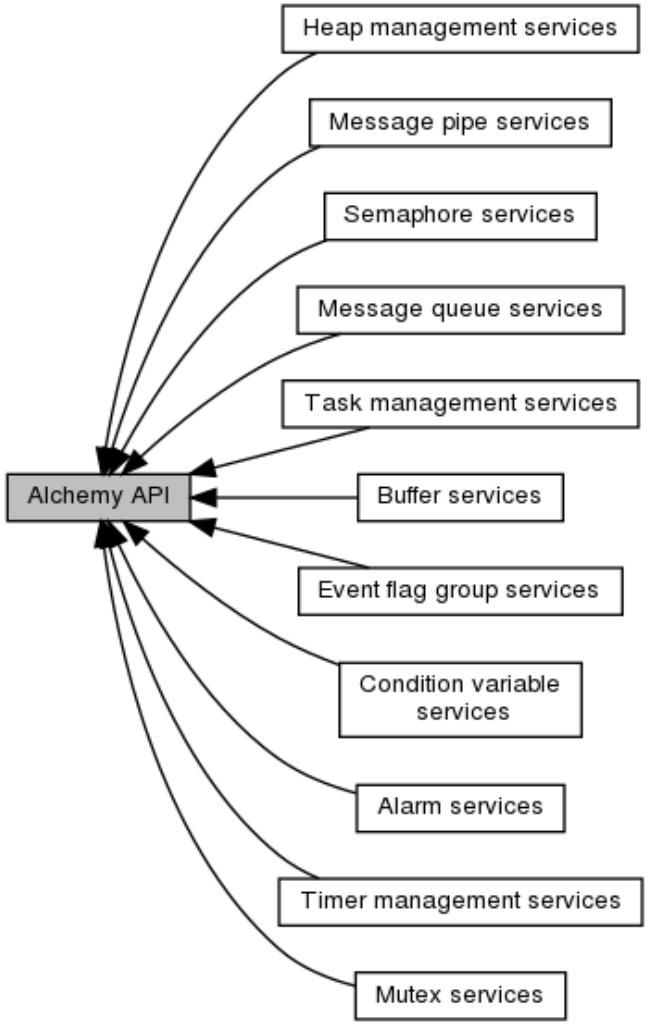
- Vous en découvrirez d'autres en TP et quand vous serez ingénieur…

CHAPITRE 8 :
PROGRAMMATION SOUS XENOMAI :
API ALCHEMY

API XENOMAI ALCHEMY

- L'API *Alchemy* pour Xenomai 3 est le nouveau nom de l'API native de Xenomai 2. Elle est disponible avec Xenomai *Mercury* et Xenomai *Cobalt*.
- L'API *Alchemy* est donc une réimplémentation *from scratch* de l'API native de Xenomai 2.
- L'API native était l'API conseillée avec Xenomai 2.
- L'API POSIX est l'API conseillée avec Xenomai 3.

API XENOMAI ALCHEMY



API XENOMAI ALCHEMY

- On retrouve dans l'API *Alchemy* les fonctionnalités essentielles d'un noyau Temps Réel dur :
  - Gestion des tâches.
  - Gestion du temps.
  - IPC : sémaphore et *mutex* (*MUTual EXclusion*).
  - IPC : variable condition.
  - IPC : *message queue*.
  - IPC : *event flag*.
  - …
- Une primitive de l'API *Alchemy* est préfixée par :
  - `rt_xxx`.

API XENOMAI ALCHEMY

- Nous allons décrire les principales primitives de l'API *Alchemy*.
- Nous ne décrirons pas complètement l'API qui est disponible en ligne :
  - https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__alchemy.html

GESTION DES TACHES

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

  Create a task with Alchemy personality.

  This service creates a task with access to the full set of *Alchemy* services. If *prio* is non-zero, the new task belongs to Xenomai's real-time FIFO scheduling class, aka SCHED_FIFO. If *prio* is zero, the task belongs to the regular SCHED_OTHER class.

- `int rt_task_start (RT_TASK *task, void(*)(void *arg) entry, void *arg)`

  Start a real-time task.

  This call starts execution of a task previously created by `rt_task_create()`. This service causes the started task to leave the initial dormant state.

GESTION DES TACHES

- `int rt_task_spawn (RT_TASK *task, const char *name, int stksize, int prio, int mode, void(*)(void *arg) entry, void *arg)`

  Create and start a real-time task.

  This service spawns a task by combining calls to `rt_task_create()` and `rt_task_start()` for the new task.

- `int rt_task_delete (RT_TASK *task)`

  Delete a real-time task.

  This call terminates a task previously created by `rt_task_create()`.

- `int rt_task_sleep (RTIME delay)`

  Delay the current real-time task (with relative delay).

GESTION DES TACHES

- `int rt_task_set_periodic (RT_TASK *task, RTIME idate, RTIME period)`

  Make a real-time task periodic.

  Make a task periodic by programing its first release point and its period in the processor time line. *task* should then call `rt_task_wait_period()` to sleep until the next periodic release point in the processor timeline is reached.

- `int rt_task_wait_period (unsigned long *overruns_r)`

  Wait for the next periodic release point.

  Delay the current task until the next periodic release point is reached. The periodic *timer* should have been previously started for *task* by a call to `rt_task_set_periodic()`.

LES SEMAPHORES

- Un sémaphore est une variable protégée (ou un type de donnée abstrait) et constitue la méthode utilisée couramment pour restreindre l'accès à des ressources partagées (par exemple un espace de stockage) dans un environnement de programmation concurrente.

- Le sémaphore a été inventé par Edsger Dijkstra.

LES SEMAPHORES

- On manipule un sémaphore de façon **atomique** : la fonction de manipulation d'un sémaphore ne pourra être interrompue ou préemptée par quoi que ce soit.

- C'est un temps mort de non réponse du SE (section critique pour le SE).

- 3 opérations atomiques de manipulation d'un sémaphore existe :
  - Initialisation d'un sémaphore.
  - Prise du sémaphore.
  - Libération du sémaphore

LES SEMAPHORES

```
Init(sem, n)
   disable_interrupt          (ordonnanceur dévalidé !)
   valeur_sem = n
   enable_interrupt

P(sem)
   disable_interrupt          (ordonnanceur dévalidé !)
   valeur_sémaphore = valeur_sémaphore – 1
   si (valeur_sémaphore] < 0) alors
              étatProcessus = Bloqué
              mettre processus en file d'attente sémaphore
   finSi
   invoquer l'ordonnanceur
   enable_interrupt
```

LES SEMAPHORES

```
V(sem)
   disable_interrupt          (ordonnanceur dévalidé !)
   valeur_sémaphore = valeur_sémaphore + 1
   si (valeur_sémaphore == 0) alors
              extraire processus de file d'attente sémaphore
              étatProcessus = Prêt
   finSi
   invoquer l'ordonnanceur
   enable_interrupt
```

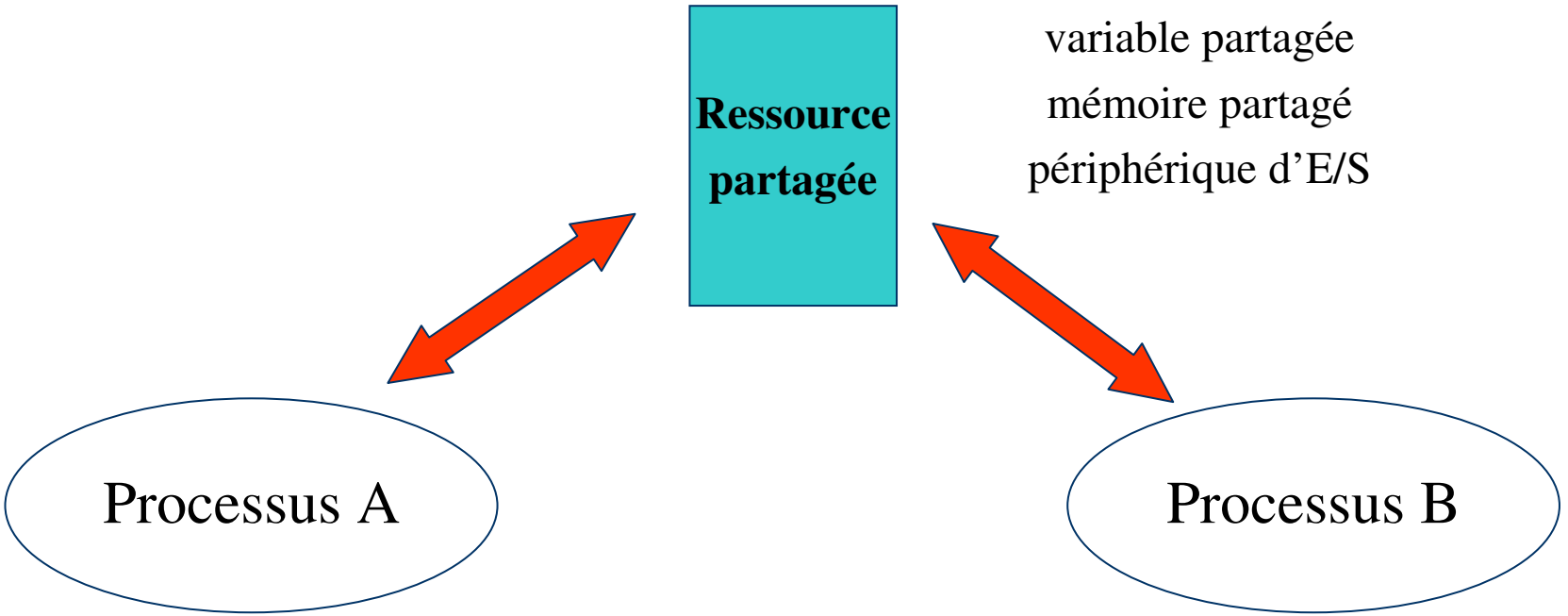
LES SEMAPHORES

- On parle de sémaphore à compteur quand le sémaphore est initialisé à une valeur > 1.

- On parle de sémaphore binaire quand le sémaphore est initialisé à 1. On parle aussi de *mutex* (POSIX).

- Le sémaphore permet de gérer 2 cas de figure complexes :
  - Accès exclusif à une ressource partagée et basé sur le principe de l'exclusion mutuelle : variable partagée, mémoire partagé ou périphérique d'E/S.
  - Synchronisation de processus.

LES SEMAPHORES



Accès exclusif à une ressource partagée

LES SEMAPHORES

```
Init(sem, 1)
```

```
. . .
P(sem)
Accès ressource partagée
V(sem)
. . .
```

Processus A

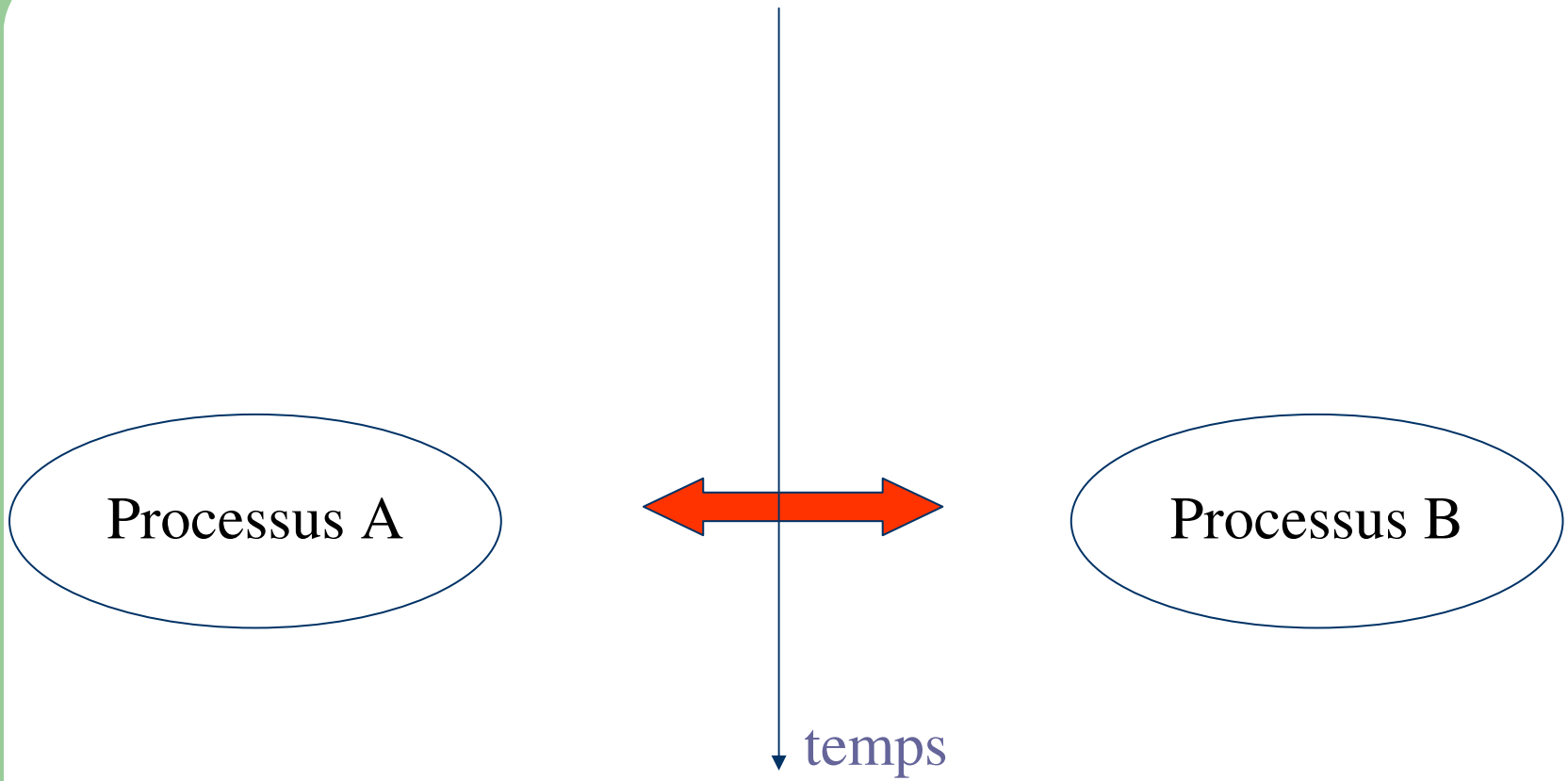
```
. . .
P(sem)
Accès ressource partagée
V(sem)
. . .
```

Processus B

Accès exclusif à une ressource partagée

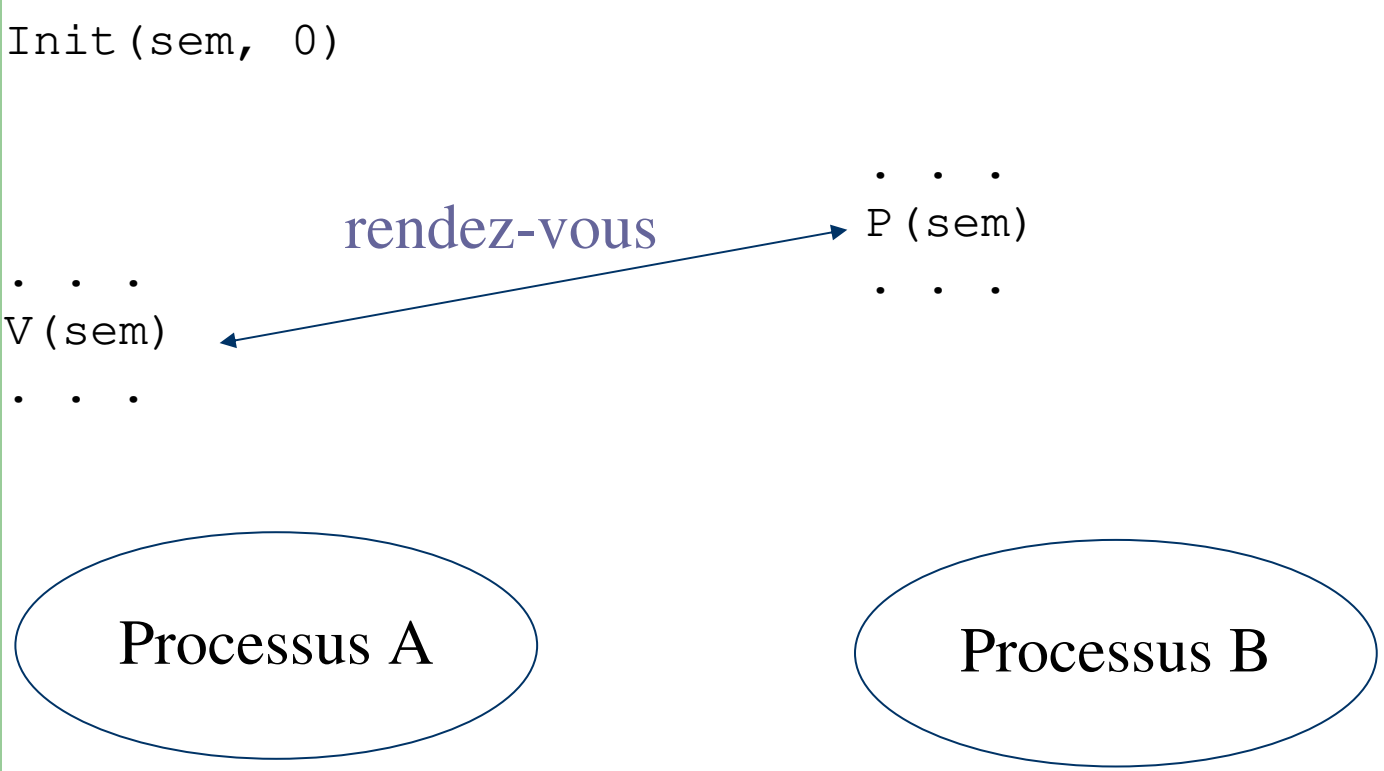
- Le sémaphore binaire est initialisé par un seul processus à **1**.

LES SEMAPHORES



Synchronisation de processus : le rendez-vous

LES SEMAPHORES



Synchronisation de processus : le rendez-vous

- Le sémaphore binaire est initialisé par un seul processus à **0**.

GESTION DES SEMAPHORES

- `int rt_sem_create (RT_SEM *sem, const char *name, unsigned long icount, int mode)`
  Create a counting semaphore.

- `int rt_sem_p (RT_SEM *sem, RTIME timeout )`
  Pend on a semaphore (with relative scalar timeout).

- `int rt_sem_v (RT_SEM *sem)`
  Signal a semaphore.
  If the semaphore is pended, the task heading the wait queue is immediately unblocked. Otherwise, the semaphore count is incremented by one.

- `int rt_sem_delete (RT_SEM *sem)`
  Delete a semaphore.

EXEMPLE ALCHEMY HELLO WORLD

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <pthread.h>
#include <sched.h>

#include <alchemy/task.h>

// Periode de 1 s en ns
#define TIMESLEEP 1*1000*1000*1000

RT_TASK rt_task1;
#define PRIO1 99
```

EXEMPLE ALCHEMY HELLO WORLD

```
void task1() {

  rt_printf("Starting Xenomai task1...\n");

  while(1) {
    rt_printf("Hello World from task1!\n");
    rt_task_sleep(TIMESLEEP);
  }
}

int main() {

  mlockall(MCL_CURRENT|MCL_FUTURE);

  rt_task_create(&rt_task1, "task1", 0, PRIO1, 0);
  rt_task_start(&rt_task1, &task1, 0);

  sleep(10);

  rt_task_delete(&rt_task1);

  exit(0);
}
```

BILAN

- On se méfiera des appels système Linux dans la partie Temps Réel. Ils risquent de n'être pas Temps Réel.
- Par exemple, l'appel système `printf()` n'est pas Temps Réel.
- On utilisera plutôt la fonction `rt_printf()` qui l'est.

BILAN

- Une tâche *Alchemy* est du type `RT_TASK *`. Un sémaphore *Alchemy* est du type `RT_SEM *`.

- Nous n'avons pas décrit toutes les primitives de l'API *Alchemy*. Cette API est semblable à celle de µC/OS II par exemple.

- Comme toutes les API propriétaires (comme VxWorks, pSOS) se pose la question de la portabilité du code source développé.

- Il vaudra alors mieux privilégier l'API POSIX.

CHAPITRE 9:
PRESENTATION DE LA NORME POSIX

NORME POSIX

- La norme POSIX est d'abord une norme de l'IEEE.
- Le but de la norme POSIX est d'obtenir la portabilité du code source des applications.
- Un programme qui est destiné à un système d'exploitation qui respecte POSIX doit pouvoir être porté sur un autre système POSIX.

POSIX= *Portable Operating System Interface eXchange*

NORME POSIX

- En théorie, le portage d'une application POSIX d'un système vers un autre doit se résumer à une recompilation des sources du programme.

- POSIX a initialement été mis en place pour les systèmes de type *NIX mais d'autres systèmes d'exploitation comme Windows sont aujourd'hui conformes à POSIX.

NORME POSIX

- La norme POSIX est divisée en plusieurs normes :
  - IEEE 1003.1-1990 : API système. Définition d'interfaces de programmation standards pour les systèmes de type *NIX, connu également sous l'appellation ISO 9945-1. Ce standard contient la définition de ces fonctions en langage C.
  - IEEE 1003.2-1992 : Interface applicative pour le s*hell* et applications annexes. Définition des fonctionnalités du *shell* et commandes annexes pour les systèmes de type *NIX.

NORME POSIX

- La norme POSIX est divisée en plusieurs normes :
  - **IEEE 1003.1b-1993** : API pour le Temps Réel. Ajout du support de programmation Temps Réel au standard précédent. On parle également de POSIX.4.
  - **IEEE 1003.1c-1995** : API pour le *multithreading*. Définition des *threads* POSIX appelés aussi *pthreads*.

- Ces 2 dernières normes sont les plus intéressantes pour ce cours et nous allons les détailler.

NORME POSIX 1003.1b

- La norme POSIX 1003.1b définit la notion de Temps Réel pour un système d'exploitation à des fins de normalisation :
  - *“Realtime in operating systems: the ability of the operating system to provide a required level of service in a bounded response time”.*

- Cela implique pour le système d’exploitation :
  - D’avoir un ordonnancement préemptif des tâches (avec priorité).
  - De coller en mémoire les pages virtuelles.
  - De supporter des signaux Temps Réel.
  - D’avoir des mécanismes de communication inter processus performants.
  - D’avoir des *timer*s Temps Réel.

NORME POSIX 1003.1b

- Linux ne supporte que partiellement la norme POSIX 1003.1b :
  - `mlock()`.
  - `setsched()`.
  - ...
- Il convient donc de modifier Linux pour le rendre Temps Réel et éventuellement conforme aussi à la norme POSIX 1003.1b.
- On pourra par exemple rajouter à Linux un deuxième noyau Temps Réel (co-noyau) qui sera conforme à la norme POSIX 1003.1b.

NORME POSIX 1003.1c

- La norme POSIX 1003.1c concerne le *multithreading*, c'est-à-dire la possibilité d'écrire des programmes concurrents.
- La programmation concurrente est le fait que les programmes s'exécutent les uns indépendamment des autres dans n'importe quel ordre ou en même temps.
- La programmation parallèle permet elle l'exécution simultanée de programmes concurrents sur plusieurs processeurs.

NORME POSIX 1003.1c

- La programmation parallèle est concurrente, l'inverse peut être faux.

- La norme POSIX 1003.1c avec le *multithreading* permet ainsi la programmation concurrente, le parallélisme est possible mais reste à la charge du programmeur…

THREAD POSIX

- Un *thread* POSIX ou *pthread* ou fil est un processus léger à contrario d'un processus *NIX qui est un processus dit lourd.

- La création d'un processus *NIX (processus lourd) demande la réservation des ressources telles qu'un espace mémoire, la table des descripteurs de fichiers ouverts et une pile interne.

THREAD POSIX

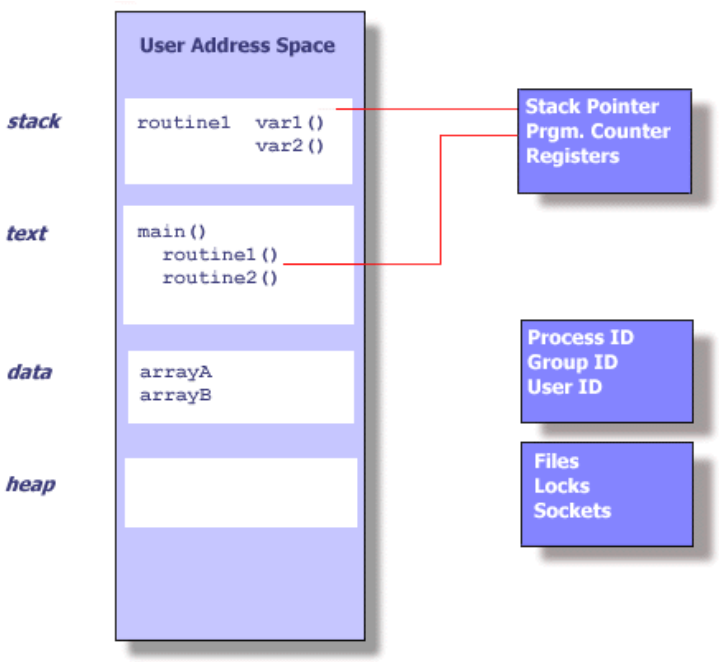
- Un *thread* partage le même espace mémoire et la même table des descripteurs que son processus créateur (routine `main()`) mais dispose de son propre environnement d'exécution.

- Ainsi, chaque *thread* possède :
  - Son propre identificateur (*Thread Id*).
  - Sa priorité.
  - Son jeu de registres et son pointeur de pile.
  - Sa pile pour les variables locales et les adresses de retour d'appel.

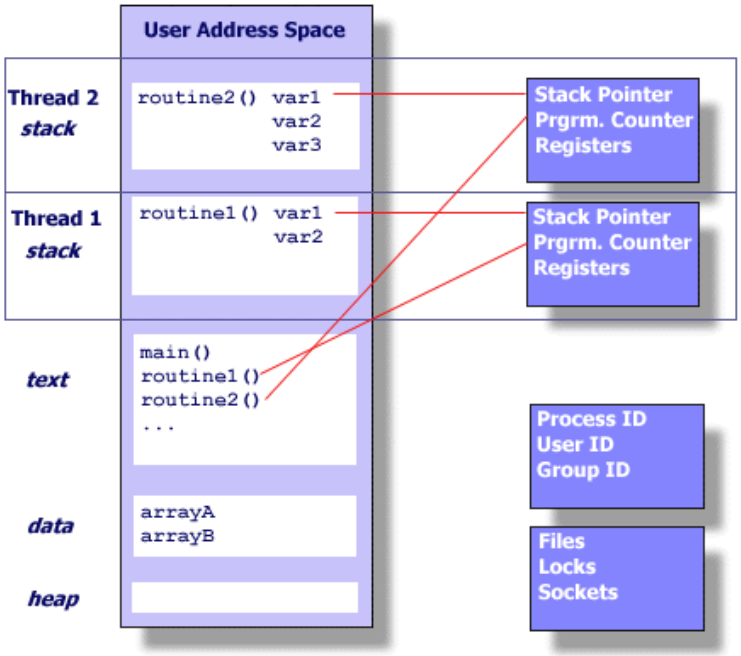
THREAD POSIX

- Sous *NIX, un pour plusieurs *threads* sont embarqués dans un processus *NIX.
- La routine `main()` du processus permet de créer les *threads* et peut être vue comme le *thread* principal (*main thread*).
- Un processus peut alors être vu comme un container à *threads*.
- Un processus sans création explicite de *threads* peut être aussi vu comme un processus contenant un seul *thread* implicite, le *main thread*.

THREAD VS PROCESSUS



Processus



Threads dans un processus

THREAD VS PROCESSUS

- La création d'un *thread* prend moins de temps que la création d'un processus avec `fork()` ou `exec()`.
- Les communications inter *threads* sont plus performantes que les communications inter processus.
- Les processus ayant une filiation partagent les descripteurs de fichiers ouverts avant le `fork()` et les variables en lecture seulement.
- Les processus avec ou sans filiation ne partagent pas le même espace d'adressage d'où l'usage de la mémoire partagée.

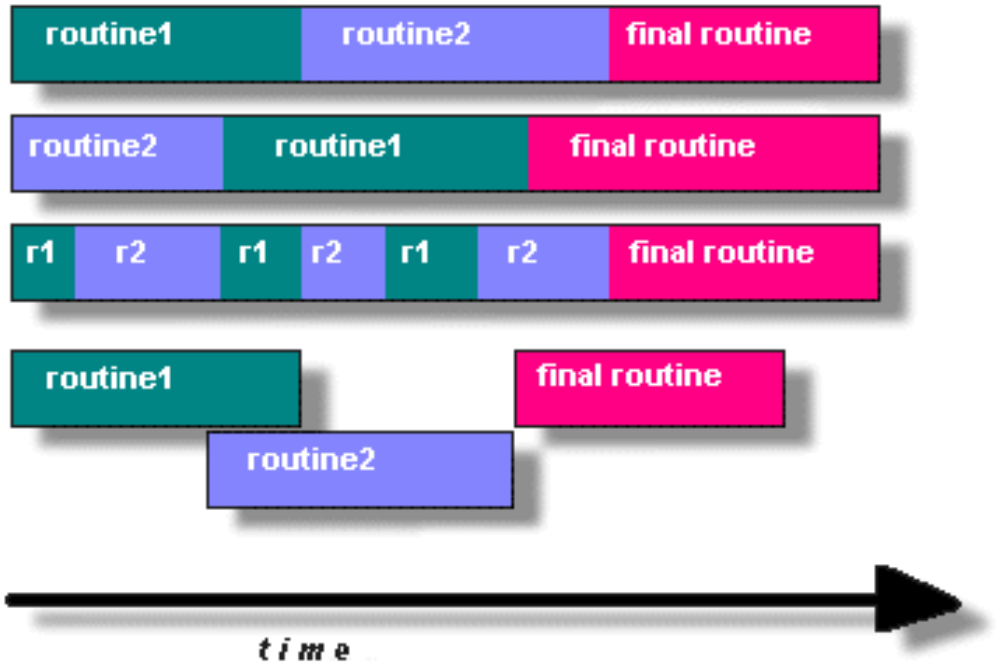
THREAD VS PROCESSUS

- Les *threads* partagent les descripteurs de fichiers ouverts et les variables en lecture et en écriture. Il n'y a donc pas besoin de mémoire partagée pour y stocker des variables partagées entre *threads*.

- Les *threads* partagent aussi le même espace d'adressage.

PROGRAMMATION CONCURRENTE

- Si l'on considère 2 routines *routine1* et *routine2*, si *routine1* et *routine2* peuvent être interchangées, entrelacées ou recouvertes l'une par l'autre, alors on peut utiliser la programmation concurrente par *multithreading*.



Interversion

Entrelaçage

Recouvrement

PROGRAMMATION CONCURRENTE

- Nous allons prendre l'exemple d'un programme composée de 3 routines et écrits de différentes façons :
  - *Routine1* = `do_one_thing()`
  - *Routine2* = `do_another_thing()`
  - *Final routine* = `do_wrap_up()` du `main()`

- Programmation séquentielle : 1 seul processus

```
#include <stdio.h>

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;

int main(void)
{
  do_one_thing(&r1);
  do_another_thing(&r2);
  do_wrap_up(r1, r2);

  exit(0);
}
```

```
void do_one_thing(int *pnum_times)
{. . .}

void do_another_thing(int *pnum_times)
{. . .}

void do_wrap_up(int one_times, int another_times)
{. . .}
```

- Programmation concurrente : plusieurs processus

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <errno.h>

void do_one_thing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int   shared_mem_id;
int   *shared_mem_ptr;
int   *r1p;
int   *r2p;
```

```
main(void)
{
  pid_t  child1_pid, child2_pid;
  int  status;

  /* initialize shared memory segment */
  if ((shared_mem_id = shmget(IPC_PRIVATE, 2*sizeof(int), 0660)) == -1){
    perror("shmget");
    exit(1);
  }

  if ((shared_mem_ptr = (int *)shmat(shared_mem_id, (void *)0, 0)) == (void *)-1){
    perror("shmat failed");
    exit(1);
  }

  r1p = shared_mem_ptr;
  r2p = (shared_mem_ptr + 1);

  *r1p = 0;
  *r2p = 0;
```

```
if ((child1_pid = fork()) == 0) {
  /* first child */
  do_one_thing(r1p);
  return 0;
} else if (child1_pid == -1) {
  perror("fork"), exit(1);
}

/* parent */
if ((child2_pid = fork()) == 0) {
  /* second child */
  do_another_thing(r2p);
  return 0;
} else if (child2_pid == -1) {
  perror("fork"), exit(1);
}
```

```
  /* parent */
  if ((waitpid(child1_pid, &status, 0) == -1))
    perror("waitpid"), exit(1);
  if ((waitpid(child2_pid, &status, 0) == -1))
    perror("waitpid"), exit(1);

  do_wrap_up(*r1p, *r2p);

  exit(0);
}

void do_one_thing(int *pnum_times)
{. . .}

void do_another_thing(int *pnum_times)
{. . .}

void do_wrap_up(int one_times, int another_times)
{. . .}
```

- Programmation concurrente : plusieurs *threads*

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>

void *do_one_thing(void *);
void *do_another_thing(void *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;
```

```
main(void)
{
  pthread_t thread1, thread2;

  if (pthread_create(&thread1, NULL, do_one_thing, (void *) &r1) != 0)
    perror("pthread_create"), exit(1);

  if (pthread_create(&thread2, NULL, do_another_thing, (void *) &r2) != 0)
    perror("pthread_create"), exit(1);

  if (pthread_join(thread1, NULL) != 0)
    perror("pthread_join"),exit(1);

  if (pthread_join(thread2, NULL) != 0)
    perror("pthread_join"),exit(1);

  do_wrap_up(r1, r2);

  exit(0);
}
```

```
void *do_one_thing(void *pnum_times)
{. . . }

void *do_another_thing(void *pnum_times)
{. . . }

void do_wrap_up(int one_times, int another_times)
{. . . }
```

- Il existe enfin 3 types de modèles de programmation par *threads* :
  - Modèle *manager/worker* : un seul *thread*, le *manager* (généralement le *main thread*) distribue le travail aux autres *threads*, les *workers*. Typiquement, le *manager* gère toutes les données d'entrée et les distribue aux autres *threads*.
  - Modèle *pipeline* : le travail est scindé en une série d'opérations dont chacune est traitée en série mais non en concurrence par un *thread* différent.
  - Modèle *Peer* : il est similaire au modèle *manager/worker* mais après que le *main thread* crée les autres *threads*, celui-ci participe comme les autres *threads* au travail commun.

OBJETS POSIX 1003.1c

- Il y a environ une centaine de fonctions dans l'API *pthread* qui sont préfixées avec le préfixe `pthread_`.

- On peut les ranger dans 4 groupes :
  - Gestion des *threads* : création, jonction, détachement, destruction…
  - Gestion des *mutex* : gestion de l'accès exclusif similaire à un sémaphore binaire.
  - Gestion des variables condition : mécanisme de communication entre *threads* utilisant un *mutex* pour signaler un événement sur une variable partagée.
  - Gestion de la synchronisation entre *threads* : verrous (*read/write lock*) et barrière (*barrier*).

OBJETS POSIX 1003.1c

- Pour les objets des 3 premiers groupes (*thread*, *mutex* et variable condition), il existe en plus un objet d'attributs qui permet de configurer les propriétés de l'objet.

- Pour chaque attribut d'un objet, il existe une valeur par défaut que l'on peut redéfinir.

- Cela vaut dire aussi que lorsque que l'on crée un objet, un *thread* par exemple, il faut aussi créer son objet attribut que l'on modifie ou pas.

OBJETS POSIX 1003.1c

- Un *mutex* est un sémaphore binaire.

- Il possède 3 fonctions de manipulation :
  - Création : elle est statique ou dynamique. On choisira l'initialisation dynamique avec la fonction `pthread_mutex_init()`. A sa création, le *mutex* est déverrouillé.
  - Verrouillage : fonction `pthread_mutex_lock()`. Cela correspond à une prise de sémaphore binaire.
  - Déverrouillage : fonction `pthread_mutex_unlock()`. Cela correspond à une libération de sémaphore binaire.

OBJETS POSIX 1003.1c

- On notera que les sémaphores à compteur existe mais ne font pas partie de la norme POSIX 1003.1c sur les *threads*.
- Ils ont été définis dans la norme POSIX 1003.b sur le Temps Réel.
- Le préfixe utilisé pour les fonctions n'est pas `pthread_` mais `sem_`.
- Les sémaphores sont utilisables avec l'API *pthread*.

OBJETS POSIX 1003.1c

- Alors qu'un *mutex* permet à des *threads* de se synchroniser pour l'accès à des données, une variable condition permet de se synchroniser sur la valeur d'une donnée.

- En effet, des *threads* peuvent attendre qu'une donnée atteigne une certaine valeur ou bien qu'un certain événement apparaisse.

- Une variable condition permet donc une notification entre *threads*.

OBJETS POSIX 1003.1c

- Sans variable condition, un *thread* devrait faire une scrutation active (*polling*) sur la donnée qui consomme du temps CPU tant qu'elle n'atteint pas la valeur souhaitée, ce qui est gênant dans une section critique.

- Cette structure de synchronisation peut être comparée à une cloche (d'après C. Blaess). Un *thread* peut s'endormir en attente sur la cloche et un autre *thread* peut venir à tout moment le réveiller en donnant un coup de marteau sur la cloche.

- Si aucun *thread* n'est en attente au moment du coup de marteau, la notification est alors perdue.

OBJETS POSIX 1003.1c

- Une condition variable s'utilise toujours avec un *mutex*. Elle indique donc l'occurrence d'un événement et ne possède pas de valeur.

Exemple :

```
pthread_cond_t cond;
pthread_mutex_t lock;

. . .
```

OBJETS POSIX 1003.1c

```
void *thread1(void *arg) {
. . .
pthread_mutex_lock(&lock);
while (1) {
  pthread_cond_wait(&cond, &lock);
  // Notification recue, traitement des donnees
. . .
}
pthread_mutex_unlock(&lock);
. . .
}

void *thread2(void *arg) {
. . .
while (1) {
  // Attente de données externes
  . . .
  // Notification du thread thread1
  pthread_mutex_lock(&lock);
  pthread_cond_signal(&cond);
  pthread_mutex_unlock(&lock);
}
. . .
}
```

OBJETS POSIX 1003.1c

- La fonction `pthread_cond_wait()` contient plusieurs étapes successives :
  - 1. Déverrouillage du *mutex* (comme `pthread_mutex_unlock()`).
  - 2. Mise en sommeil en attente de la notification.
  - 3. Quand la notification est signalée, verrouillage du *mutex* (comme `pthread_mutex_lock()`).
  - 4. Retour de fonction.

- Les deux premières étapes sont atomiques. Il n'est pas possible qu'un *thread* invoque `pthread_cond_signal()` sans qu'un autre *thread* ne soit véritablement en attente.

OBJETS POSIX 1003.1c

Préfixe	**Groupe fonctionnel**
`pthread_`	*Threads*
`pthread_attr_`	Attributs d'un *thread*
`pthread_mutex_`	*Mutex*
`pthread_mutexattr_`	Attributs d'un *mutex*
`pthread_cond_`	Variables condition
`pthread_condattr_`	Attributs d'une variable condition
`pthread_key_`	*Thread data keys*
`pthread_rwlock_`	*Read/write locks*
`pthread_barrier_`	*Synchronization barriers*

OBJETS POSIX 1003.1c

- Il faudra inclure le fichier *header* suivant dans son source C pour avoir accès à l'API *pthread* :

```
#include <pthread.h>
```

- Il faudra inclure la bibliothèque *pthread* lors de la compilation du source C :

```
$ gcc -o mon_exe mon_source.c -lpthread
```

BILAN

- L'API POSIX est une API intéressante pour assurer la pérennité et la portabilité du code développé au niveau source.
- L'API POSIX devient de facto avec Xenomai 3 l'API par défaut comme elle l'était déjà dans le monde *NIX. C'est une petite révolution.
- Enfin, la maîtrise de l'API POSIX n'est pas plus difficile que l'apprentissage de l'API d'un RTOS propriétaire bien au contraire, elle est par nature réutilisable.

CHAPITRE 10 :
PROGRAMMATION SOUS XENOMAI :
API POSIX COBALT

API XENOMAI COBALT

- Depuis la version 3, l'API POSIX ou API *Cobalt* est l'API de base pour la programmation Temps Réel sous Xenomai.
- L'API *Cobalt* de Xenomai est conforme à l'API standard POSIX mais n'implémente pas toutes les fonctionnalités.
- Les fonctions POSIX renvoient toutes un code de retour qu'il faut tester.
- Elles renvoient 0 si OK et une valeur différente conforme à celles du fichier `errno.h` si erreur.

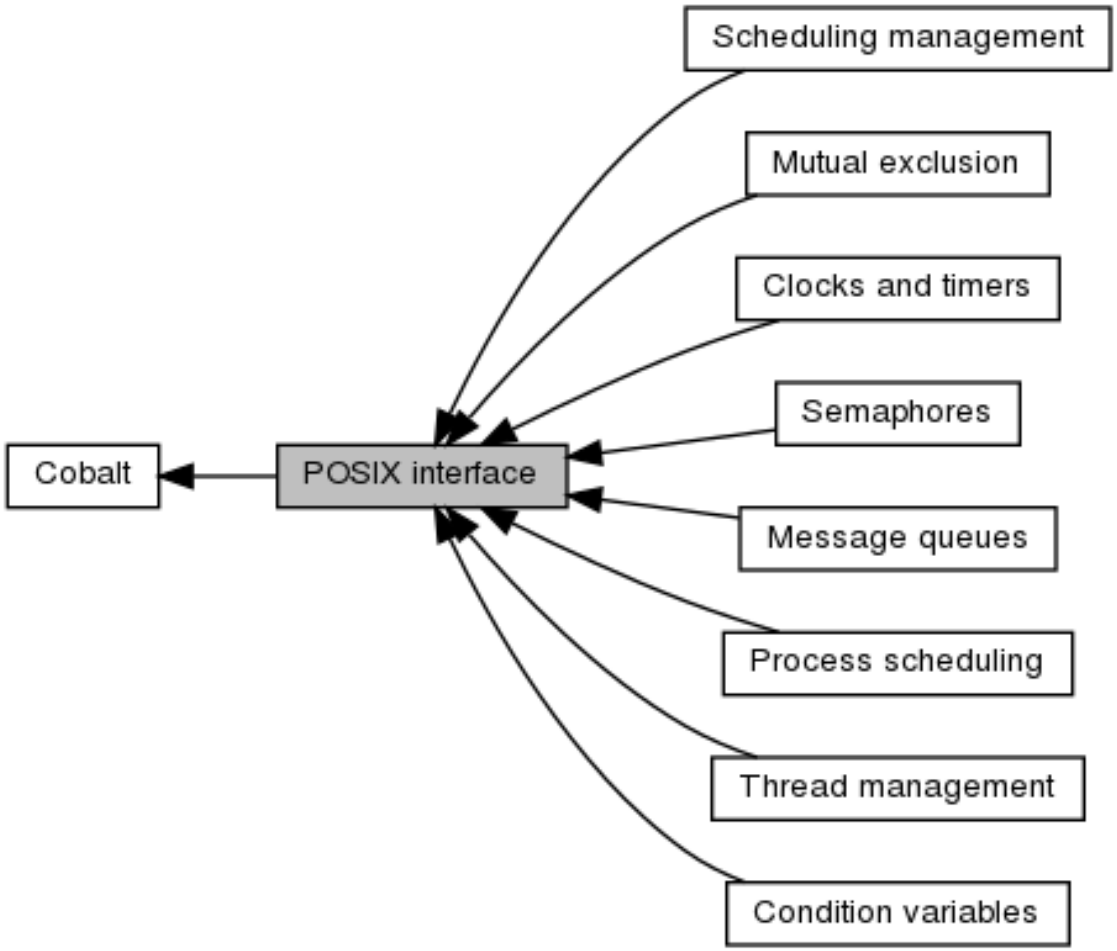
API XENOMAI COBALT

- Les fonctions POSIX n'utilisent pas la variable globale `errno` qui est renseignée après chaque exécution d'un appel système *NIX.
- Nous allons décrire les principales primitives de l'API *Cobalt*.
- Nous ne décrirons pas complètement l'API qui est disponible en ligne :
  - https://xenomai.org/documentation/xenomai-3/html/xeno3prm/group__cobalt__api.html

API XENOMAI COBALT

- Nous allons ainsi regarder les fonctions principales des 3 premiers groupes de l'API *Cobalt* :
  - Gestion des *threads*.
  - IPC : *mutex*.
  - IPC : variables condition.
- Mais aussi la :
  - Gestion de l'ordonnancement.
  - Gestion du temps.
  - IPC : sémaphores

API XENOMAI COBALT



GESTION DES THREADS

- Un *thread* s'utilise avec un objet précisant ses attributs.

- On a donc :
  - L'objet *thread* du type `pthread_t`. Cela correspond à son identificateur ou *Thread ID* (TID) comme le PID pour un processus.
  - L'objet attribut du *thread* du type `pthread_attr_t`.

GESTION DES THREADS

- `int pthread_create(pthread_t *ptid_r, const pthread_attr_t *attr, void *(*)(void *)start, void *arg)`

  Crée un nouveau *thread* géré par le noyau *Cobalt*.

  Les attributs du nouveau *thread* créé dépendent de l'argument `attr`. Si `attr` vaut `NULL`, les valeurs par défaut des attributs sont utilisées.

Exemple :

```
pthread_t th1;

pthread_create(&th1, NULL, th1_routine, NULL);
```

GESTION DES THREADS

- `void pthread_exit(void *retval)`

  Termine le *thread* appelant `thread` et retourne une valeur `retval` (si le *thread* est joignable) disponible pour un autre *thread* qui appelle alors `pthread_join()`.

Exemple :

```
pthread_t th1;
int ret;

pthread_exit((void *)&ret);
pthread_exit(NULL);
```

GESTION DES THREADS

- `int pthread_cancel(pthread_t` *`thread`*`)`

  Envoie une requête d'annulation au *thread* `thread`. La façon dont le *thread* réagira à cette requête dépend de 2 attributs du *thread* : son état d'annulation et le type d'annulation (asynchrone…).

Exemple :

```
pthread_t th1;

pthread_cancel(&th1);
```

GESTION DES THREADS

- Pour utiliser `pthread_cancel()` pour détruire un *thread* de façon asynchrone (majorité des cas), il faut que celui-ci soit correctement paramétré par ses attributs.

Exemple :

```
void *th1_routine (void * arg) {
. . .
pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
pthread_setcanceltype(PTHREAD_CANCEL_ASYNCHRONOUS, NULL);
. . .
}

pthread_create(&th1, NULL, th1_routine, NULL);
. . .
pthread_cancel(&th1);
```

GESTION DES THREADS

- `int pthread_join (pthread_t` *`thread`*`, void **`*`retval`*`)`
  Attend la fin du *thread* `thread`.

  Si le *thread* s'exécute et est joignable, l'appel est bloquant jusqu'à ce que le *thread* se termine. Quand le *thread* est terminé, l'appel est débloqué et on récupère à l'adresse `retval` le code de retour du *thread* `thread`.

Exemple :

```
pthread_t th1;

pthread_join(&th1, NULL);
```

GESTION DES THREADS

- Quand un *thread* est créé, l'un de ses attributs définit s'il est joignable (*joignable*) ou détaché (*detached*).
- Seul un *thread* créé comme joignable l'est. S'il est créé détaché, il n'est plus joignable.
- La norme POSIX spécifie qu'un *thread* est joignable par défaut.

GESTION DES THREADS

- Un *thread* joignable permet d'attendre sa mort par un autre *thread* avec la fonction `pthread_join()`.

- On peut maintenant dire que l'on peut **synchroniser** des *threads* de **3 façons** avec :
  - L'usage de `pthread_join()`.
  - Les *mutex*.
  - Les variables condition.

GESTION DES THREADS

Pour créer un *thread* joignable (par défaut) :

```
pthread_t th1;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

pthread_create(&th1, &attr, th1_routine, NULL);
```

GESTION DES THREADS

Pour créer un *thread* détaché :

```
pthread_t th1;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

pthread_create(&th1, &attr, th1_routine, NULL);
```

GESTION DES THREADS

- `int pthread_detach(pthread_t` *`thread`*`);`

  Marque le *thread* `thread` comme détaché.

  Cette fonction peut être utilisée pour détacher explicitement un *thread* même s'il a été créé comme joignable.

Exemple :

```
pthread_t th1;

pthread_detach(&th1);
```

GESTION DE L'ORDONNANCEMENT

- Nous avons besoin de gérer la politique d'ordonnancement d'un *thread* mais aussi de fixer sa priorité.

- Pour que le *thread* créé soit traité comme un *thread* Temps Réel par Xenomai, il doit utiliser la politique d'ordonnancement SCHED_FIFO (ou SCHED_RR).

- Pour cela, on utilisera conjointement les fonctions POSIX suivantes sur les attributs du *thread* avant l'appel de `pthread_create()` :
  - `pthread_attr_setinheritsched()`.
  - `pthread_attr_setschedpolicy()`.
  - `pthread_attr_setschedparam()`.

GESTION DE L'ORDONNANCEMENT

- Nous rappelons que Xenomai dans sa configuration *Cobalt* exécute 2 noyaux :
  - Le co-noyau Temps Réel Xenomai avec sa politique d'ordonnancement Temps Réel dur.
  - Le noyau Linux, ses *thread*s et ses processus exécutés quand rien de Temps Réel est à exécuter.

- Nous rappelons qu'un *thread* peut s'exécuter dans 2 modes :
  - Mode primaire : il est ordonnancé par Xenomai *Cobalt*.
  - Mode secondaire : il est ordonnancé par le noyau Linux comme un *thread* Linux.

GESTION DE L'ORDONNANCEMENT

- Un tel *thread* change de mode dynamiquement, ce qui veut dire que :
  - S'il appelle un service de Xenomai et qu'il est en mode secondaire, il passe en mode primaire.
  - S'il appelle n'importe quel service qui n'appartient pas au domaine Xenomai ou qu'il appelle un service de Linux (ou suite à une exception) et qu'il est en mode primaire, il passe en mode secondaire.

GESTION DE L'ORDONNANCEMENT

- Il faudra donc faire attention aux générations d'exceptions (FPU, mémoire…) et aux appels système Linux (`printf()`, `malloc()`, `open()`, `read()`, `write()`, `ioctl()`, `socket()`, `connect()`, `sendto()`, `recvfrom()`…).

- Cela peut aboutir à des temps de latence non maîtrisés ou à des problèmes d'inversion de priorité.

- Il est possible de le vérifier en cours d'exécution par la fonction :
  - `pthread_set_mode_np(0, PTHREAD_WARNSW, NULL);` (np pour *Non Posix*).

Thread1	Thread 2	Action
`pthread_mutex_lock(&mutex);`	`...`	*Thread1* passe en mode primaire pour prendre le *mutex*
`...`	`pthread_mutex_lock(&mutex);`	*Thread2* suspendu en attente sur le *mutex*
`write(fd, buffer, sizeof(buffer));`	`...`	*Thread1* passe en mode secondaire et peut être préempté par le noyau Linux. *Thread2* se trouve piégé !

Exemple de piège

GESTION DE L'ORDONNANCEMENT

- Pour affecter une priorité à un *thread*, on utilisera la structure suivante `sched_param` :

```
struct sched_param param;
```

- La fonction suivante permet de fixer la priorité du *thread* via son objet attributs `attr` :

```
int pthread_attr_setschedparam(pthread_attr_t *attr,
const struct sched_param *param)
```

GESTION DE L'ORDONNANCEMENT

- La fonction suivante permet de fixer la politique d'ordonnancement du *thread* via son objet attributs `attr` :

```
int pthread_attr_setschedpolicy(pthread_attr_t
*attr, int policy)
```

`policy` peut valoir SCHED_OTHER, SCHED_FIFO ou SCHED_RR.

Pour utiliser l'API *Cobalt*, `policy` doit valoir obligatoirement SCHED_FIFO (ou SCHED_RR).

GESTION DE L'ORDONNANCEMENT

- La fonction suivante permet de préciser l'héritage sur le type d'ordonnancement du *thread* :

```
int pthread_attr_setinheritsched(pthread_attr_t
*attr, int inheritsched)
```

Si `inheritsched=PTHREAD_EXPLICIT_SCHED` : le *thread* à créer utilise les paramètres contenus dans son objet attributs `attr`.

Si `inheritsched=PTHREAD_INHERIT_SCHED` : le *thread* à créer héritera des propriétés d'ordonnancement de son processus créateur.

GESTION DE L'ORDONNANCEMENT

Exemple : création d'un *thread* Xenomai de priorité 99

```
pthread_t thread1;
pthread_attr_t attr_thread1;
struct sched_param param;

pthread_attr_init (&attr_thread1);
pthread_attr_setinheritsched(&attr_thread1,
   PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr_thread1, SCHED_FIFO);

param.sched_priority = 99;
pthread_attr_setschedparam(&attr_thread1, &param);
. . .
pthread_create(&thread1, &attr_thread1, task1, NULL)
```

GESTION DES MUTEX

- Un *mutex* s'utilise comme un sémaphore binaire avec un objet définissant ses attributs.

- On a donc :
  - L'objet *mutex* du type `pthread_mutex_t`.
  - L'objet attribut du *mutex* du type `pthread_mutexattr_t`.

GESTION DES MUTEX

- Il y a 2 façons d'initialiser un *mutex* :
  - Initialisation statique :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

  - Initialisation dynamique par la fonction `pthread_mutex_init()`.

- L'API *Cobalt* ne supporte que l'initialisation dynamique (il vaut toujours mieux avoir une initialisation par instruction).

GESTION DES MUTEX

- `int pthread_mutex_init(pthread_mutex_t *`*`mutex`*`, const pthread_mutexattr_t *`*`attr`*`)`
  Initialisation d'un *mutex*. Le *mutex* est déverrouillé par défaut.

- `int pthread_mutex_lock(pthread_mutex_t *`*`mutex`*`)`
  Verrouillage du *mutex*.

- `int pthread_mutex_unlock (pthread_mutex_t *`*`mutex`*`)`
  Déverrouillage du *mutex*.

Exemple :

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

GESTION DES VARIABLES CONDITION

- Une variable condition s'utilise avec un objet définissant ses attributs.

- On a donc :
  - L'objet variable condition du type `pthread_cond_t`.
  - L'objet attribut de la variable condition du type `pthread_condattr_t`.

GESTION DES VARIABLES CONDITION

- Il y a 2 façons d'initialiser une variable condition :
  - Initialisation statique :

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER ;
```

  - Initialisation dynamique par la fonction `pthread_cond_init()`.

- L'API *Cobalt* ne supporte que l'initialisation dynamique (il vaut toujours mieux avoir une initialisation par instruction).

GESTION DES VARIABLES CONDITION

- `int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr)`
  Initialise une variable condition.

- `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`
  Attend sur une variable condition.

  La fonction déverrouille de façon atomique le *mutex mx*, et bloque le *thread* appelant jusqu'à ce que la variable condition *cnd* soit signalée par `pthread_cond_signal()` ou `pthread_cond_broadcat()`. Quand la variable condition est signalée, la fonction verrouille le *mutex* avant de se terminer.

GESTION DES VARIABLES CONDITION

- `int pthread_cond_signal(pthread_cond_t *cond)`
  Signale une variable condition.

  Cette fonction débloque un *thread* bloqué sur la variable condition *cnd*. Si plus d'un *thread* est bloqué sur la variable condition, c'est le *thread* de plus forte priorité qui est débloqué.

- `int pthread_cond_broadcast(pthread_cond_t *cond)`
  Signale une variable condition à tous les *threads*.

  Cette fonction débloque tous les *threads* bloqués sur la variable condition *cnd*.

- `int pthread_cond_destroy(pthread_cond_t *cond)`
  Détruit une variable condition.

GESTION DES SEMAPHORES

- Les sémaphores ne font pas partie de l'API *pthread* POSIX 1003.1c mais de l'API 1003.1b sur l'extension Temps Réel POSIX.

- On préfèrera le *mutex* au sémaphore binaire.

GESTION DES SEMAPHORES

- `int sem_init(sem_t *sem, int pshared, unsigned int value)`

  Initialize an unnamed semaphore.

  This service initializes the semaphore *sem*, with the value *value*.

- `int sem_destroy(sem_t *sem)`

  Destroy an unnamed semaphore

  This service destroys the semaphore *sem*. *Threads* currently blocked on *sem* are unblocked and the service they called return -1 with *errno* set to EINVAL.

GESTION DES SEMAPHORES

- `int sem_post(sem_t *sem)`

  Post a semaphore

  This service posts the semaphore *sem*.

  If no *thread* is currently blocked on this semaphore, its count is incremented. If a *thread* is blocked on the semaphore, the *thread* heading the wait queue is unblocked.

- `int sem_wait(sem_t *sem)`

  Decrement a semaphore

  This service decrements the semaphore *sem* if it is currently if its value is greater than 0. If the semaphore's value is currently zero, the calling *thread* is suspended until the semaphore is posted, or a signal is delivered to the calling *thread*.

GESTION DES SEMAPHORES

Exemple :

```
#include <semaphore.h>
sem_t sem;

sem_init(&sem, 0, 0); // Initialisation à 0
sem_wait(&sem);
sem_post(&sem);
```

GESTION DU TEMPS

- L'API *Cobalt* supporte les 2 horloges suivantes :
  - CLOCK_MONOTONIC.
  - CLOCK_REALTIME.

- Avec Xenomai 3, il n'est plus possible d'utiliser `pthread_make_periodic_np()` and `pthread_wait_np()` de Xenomai 2 pour réaliser un *thread* périodique.

GESTION DU TEMPS

- Comme l'API *Cobalt* supporte les signaux Temps Réel, on utilisera à la place un *timer* périodique via la combinaison `timer_create()` + `timer_settime()` et attente de l'expiration du *timer* qui provoquera l'émission d'un signal vers le *thread* qui est en attente du signal avec `sigwaitinfo()`.

- On utilisera pur cela les signaux Temps Réel comme SIGRTMIN à SIGRTMIN+15 et SIGRTMAX-14 à SIGRTMAX.

GESTION DU TEMPS

- `int clock_nanosleep(clockid_t` *`clock_id`*`, int` *`flags`*`, const struct timespec *`*`rqtp`*`, struct timespec *`*`rmtp`*`)`
  Sleep some amount of time.

- `int nanosleep(const struct timespec *`*`rqtp`*`, struct timespec *`*`rmtp`*`)`
  Sleep some amount of time.

Exemple :

```
struct timespec ts;

ts.tv_sec = 0; // En s
ts.tv_nsec = 100000000L; // 100 ms en ns
clock_nanosleep(CLOCK_MONOTONIC, 0, &ts, NULL);
nanosleep(&ts, NULL);
```

EXEMPLE COBALT HELLO WORLD

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <pthread.h>
#include <sched.h>

#include <alchemy/task.h>

#define PRIO1 99
pthread_attr_t attr_thread1;
pthread_t thread1;

struct sched_param param;
```

EXEMPLE COBALT HELLO WORLD

```
void *task1() {
  struct timespec ts;

  ts.tv_sec = 1; // En s
  ts.tv_nsec = 0; // En ns

  rt_printf("Starting Xenomai thread1...\n");

  while(1) {
    rt_printf("Hello World from thread1!\n");

    clock_nanosleep(CLOCK_MONOTONIC, 0, &ts, NULL);
  }
  pthread_exit(NULL);
}
```

EXEMPLE COBALT HELLO WORLD

```
int main() {

  mlockall(MCL_CURRENT|MCL_FUTURE);

  pthread_attr_init (&attr_thread1);
  pthread_attr_setinheritsched(&attr_thread1, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy(&attr_thread1, SCHED_FIFO);

  param.sched_priority = PRIO1;
  pthread_attr_setschedparam(&attr_thread1, &param);

  // Creation et lancement du thread Xenomai
  pthread_create(&thread1, &attr_thread1, task1, NULL);

  sleep(10);

  // Destruction du thread Xenomai
  pthread_cancel(thread1);

  exit(0);
}
```

EXEMPLE COBALT VARIABLE CONDITION

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/time.h>

pthread_t th1;
pthread_cond_t cond;
pthread_mutex_t lock;

unsigned int poids;

void *thread1(void *arg)
{

  pthread_mutex_lock(&lock);
  while (1) {
    pthread_cond_wait(&cond, &lock);
    printf("Poids=%d\n", poids);
  }
  pthread_exit(0);;
}
```

EXEMPLE COBALT VARIABLE CONDITION

```
int main (void)
{
  pthread_mutex_init(&lock, NULL);
  pthread_cond_init(&cond, NULL);

  pthread_create(&th1, NULL, thread1, NULL);

  srand(time(NULL));

  while (1) {
    poids = rand()%100;

    pthread_mutex_lock(&lock);
    if(poids > 60)
      pthread_cond_signal(&cond);
    pthread_mutex_unlock(&lock);

    sleep(1);
  }
  pthread_join(th1, NULL);
  exit(0);
}
```

A l'exécution :

```
% ./cond
Poids=96
Poids=77
Poids=67
Poids=88
Poids=75
```

EXEMPLE COBALT TIMER

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/mman.h>
#include <sys/time.h>
#include <pthread.h>
#include <sched.h>

#define PRIO1 99
pthread_attr_t attr_thread1;
pthread_t thread1;

struct sched_param param;

void mon_handler(int unused)
{
  printf("."); fflush(stdout); // Non TR
}
```

EXEMPLE COBALT TIMER

```
void *task1() {
  struct timespec ts;
  timer_t timer;
  struct sigevent event;
  struct itimerspec spec;
  sigset_t set;
  siginfo_t info;

  signal(SIGRTMIN, mon_handler);
  event.sigev_notify = SIGEV_SIGNAL;
  event.sigev_signo  = SIGRTMIN;
  spec.it_interval.tv_sec  = 0;
  spec.it_interval.tv_nsec = 500000000LL; // 500 ms
  spec.it_value = spec.it_interval;

  timer_create(CLOCK_MONOTONIC, &event, &timer);
  printf("Starting Xenomai thread1...\n");
  timer_settime(timer, 0, &spec, NULL);

  while(1)
    sigwaitinfo(&set, &info);

  pthread_exit(NULL);
}
```

EXEMPLE COBALT TIMER

```
int main() {

  mlockall(MCL_CURRENT|MCL_FUTURE);

  pthread_attr_init (&attr_thread1);
  pthread_attr_setinheritsched(&attr_thread1, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy(&attr_thread1, SCHED_FIFO);

  param.sched_priority = PRIO1;
  pthread_attr_setschedparam(&attr_thread1, &param);

  pthread_create(&thread1, &attr_thread1, task1, NULL);

  pthread_join(thread1, NULL);
  exit(0);
}
```

A l'exécution :

```
# ./timer
Starting Xenomai thread1...
.........................
```

CONTENU DE L'API COBALT POSIX

- Liste exhaustive des fonctions de l'API *Cobalt* :

```
pthread_attr_init pthread_create pthread_getschedparam
pthread_setschedparam pthread_setschedprio pthread_mutex_init
pthread_mutex_destroy pthread_mutex_lock pthread_mutex_timedlock
pthread_mutex_trylock pthread_mutex_unlock
pthread_mutex_setprioceiling pthread_mutex_getprioceiling
pthread_cond_init  pthread_cond_destroy pthread_cond_wait
pthread_cond_timedwait pthread_cond_signal
pthread_cond_broadcast pthread_kill pthread_join pthread_yield
pthread_setname_np clock_getres clock_gettime clock_settime
clock_adjtime clock_nanosleep nanosleep timer_create
timer_delete timer_settime timer_gettime timer_getoverrun
sched_yield sched_get_priority_min sched_get_priority_max
sched_setscheduler sched_getscheduler sem_init sem_destroy
sem_post sem_wait sem_timedwait sem_trywait sem_getvalue
sem_open sem_close sem_unlink mq_open mq_close mq_unlink
mq_getattr mq_setattr mq_send mq_timedsend mq_receive
mq_timedreceive mq_notify timerfd_create timerfd_settime
timerfd_gettime sigpending sigwait sigwaitinfo sigtimedwait
sigqueue
```

BILAN

- Nous avons pu voir les principales fonctionnalités de L'API POSIX de Xenomai *Cobalt*.

- L'API POSIX devient de facto avec Xenomai 3 l'API par défaut avec Xenomai *Cobalt* comme elle l'était déjà dans le monde *NIX.

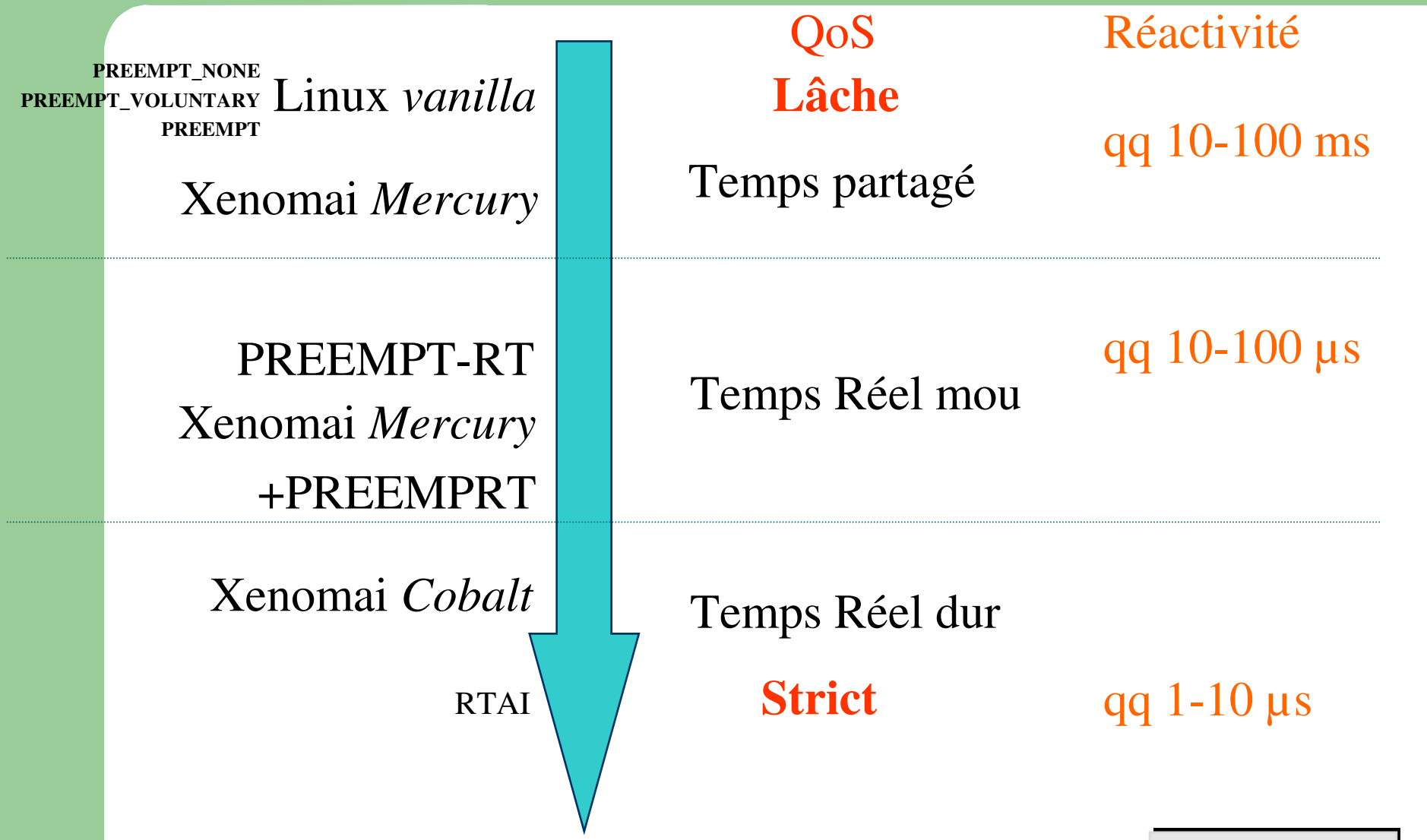
- Il faudra surtout attention à la commutation non voulue des *threads* POSIX du mode primaire au mode secondaire induisant des temps de latence incontrôlés. RIGUEUR RIME AVEC RIGUEUR…

CONCLUSION

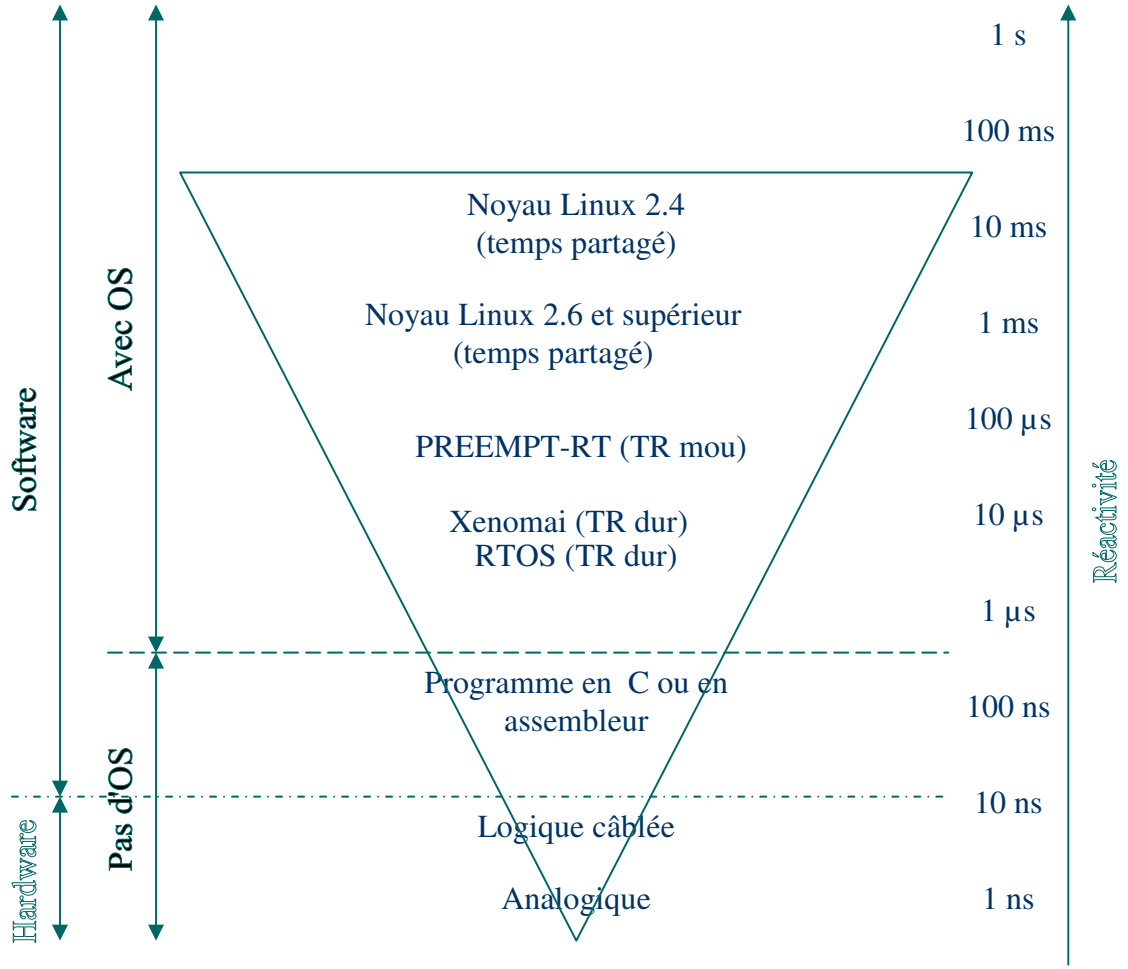
LE CHOIX D'UN LINUX TEMPS REEL LIBRE ?

- Le choix d'un Linux Temps Réel est à faire en fonction des contraintes Temps Réel que doit respecter le système :
  - Pas de contraintes. Temps partagé. Réactivité de qq 10ms à qq 100 ms :
    - Noyau *vanilla,* Xenomai Mercury.
  - Temps Réel mou. Réactivité de qq 10 µs à qq 100 µs :
    - PREEMPT-RT, Xenomai Mercury avec *patch* PREEMPT-RT.
  - Temps Réel dur. Réactivité de qq µs à qq 10 µs :
    - Xenomai Cobalt.

LE CHOIX D'UN LINUX TEMPS REEL LIBRE ?



LE CHOIX D'UN LINUX TEMPS REEL LIBRE ?



LE CHOIX D'UN LINUX TEMPS REEL LIBRE ?

- Le noyau Linux *vanilla* n'est pas fondamentalement un système d'exploitation Temps Réel dur car trop généraliste.

- Il est possible d'avoir un système Linux Temps Réel mou en utilisant l'extension PREEMPT-RT. On pourra aussi mettre en œuvre Xenomai *Mercury* dans une approche simple noyau pour l'émulation d'API de RTOS propriétaires.

- Il est possible d'avoir un système Linux Temps Réel dur en utilisant l'extension Temps Réel Xenomai *Cobalt* dans une approche double noyau pour l'émulation d'API de RTOS propriétaires.

LE CHOIX D'UN LINUX TEMPS REEL

- Le choix final se fera en fonction des contraintes temporelles imposées par le processus à contrôler depuis Linux en prenant aussi en compte la complexité de mise en oeuvre dans chaque cas.

- Choisir là aussi un Linux Temps Réel commercial peut être rassurant. Cela a aussi un coût.

CONCLUSION

- Dans ce cours, nous avons aussi abordé différents points :
  - Une introduction aux systèmes Temps Réel et son usage sous Linux.
  - Les offres Linux Temps Réel commerciales et libres.
  - La présentation de PREEMPT-RT et sa mise en œuvre sur cible x86.
  - La présentation de Xenomai et sa mise en œuvre sur cibles x86 et ARM (Raspberry Pi).
  - Des compléments techniques indispensables sur Linux : ordonnancement, gestion du temps, programmation concurrente, norme POSIX et *threads*.
  - La mise en œuvre pratique des API Xenomai *Alchemy* (API native) et *Cobalt*.

BIBLIOGRAPHIE

REFERENCES BIBLIOGRAPHIQUES

- Linux embarqué. P. Ficheux. Editions Eyrolles.

  LA REFERENCE !

- Solutions Temps Réel sous Linux. C. Blaess. Editions Eyrolles.

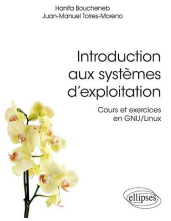
  LA REFERENCE !

- Développement système sous Linux. C. Blaess. Editions Eyrolles.

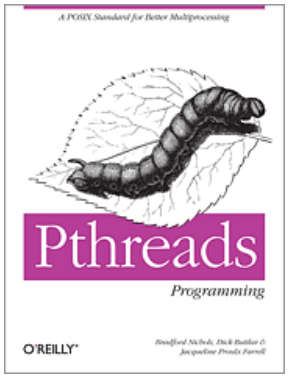
  LA REFERENCE !

REFERENCES BIBLIOGRAPHIQUES

- Introduction aux systèmes d'exploitation. H. Boucheneb. Editions Ellipses.



- PThreads Programming. A POSIX Standard for Better Multiprocessing. D. Buttlar and al. Editions O Reilly.



REFERENCES BIBLIOGRAPHIQUES

- Projet Xenomai :

  https://gitlab.denx.de/Xenomai/xenomai/-/wikis/Start_Here

- Xenomai 3 : hybride et caméléon. P. Gérum. Open Silicium No 16. octobre 2015 :

  https://connect.ed-diamond.com/Open-Silicium/OS-016/Xenomai-3-hybride-et-cameleon

- Porting a Linux application to Xenomai dual kernel J. Kiszka :

  https://gitlab.denx.de/Xenomai/xenomai/-/wikis/Porting_To_Xenomai_POSIX

REFERENCES BIBLIOGRAPHIQUES

- POSIX Threads Programming. Blaise Barney :
  https://computing.llnl.gov/tutorials/pthreads

- Programmation multitâche temps réel en langage C avec la norme POSIX. G. Kemayo :
  https://gkemayo.developpez.com/tempsreel/programmation-multitache-posix

- Notifications rapides d'une variable condition. C. Blaess :
  https://www.blaess.fr/christophe/2013/08/19/notifications-rapides-dune-variable-condition

REFERENCES BIBLIOGRAPHIQUES

- Recompilation du noyau Fedora. Patrice Kadionik :

  https://doc.fedora-fr.org/wiki/Recompilation_du_noyau_Fedora

- Mise en œuvre de PREEMPT-RT pour créer un noyau Temps Réel mou. Patrice Kadionik :

  https://doc.fedora-fr.org/wiki/Mise_en_%C5%93uvre_de_PREEMPT-RT_pour_cr%C3%A9er_un_noyau_Temps_R%C3%A9el_mou