ENSEIRB-MATMECA

MISE EN ŒUVRE DE L'EXTENSION TEMPS REEL XENOMAI COBALT SUR CARTE RASPBERRY PI

Patrice KADIONIK
http://kadionik.vvv.enseirb-matmeca.fr/

TABLE DES MATIERES

1. But des travaux pratiques ... 3
2. Informations essentielles sur la carte carte rpi-xenomai ... 4
3. Fonctions utilitaires du Board Support Package pour la carte rpi-xenomai ... 7
4. TP 0 : prise en main ... 8
5. TP 1 : génération du RAM disk pour le noyau Linux Xenomai ... 10
6. TP 2 : compilation du noyau Linux Xenomai et boot ... 12
1 TP 3 : mesure de temps de latence avec le noyau Linux Xenomai ... 14
6.1. Outils standards ... 14
6.2. Outils graphiques ... 15
7. TP 4 : application Hello World avec l'API Alchemy ... 20
8. EX 0 : application Hello World avec l'API POSIX Cobalt ... 23
9. EX 1 : multithreading. Chenillard, plus un et Hello World ... 26
10. EX 2 : mutex. Gestion de l'accès exclusif à une ressource partagée ... 27
11. EX 3 : mutex. Synchronisation de threads. Rendez-vous ... 28
12. EX 4 : mutex. Récapitulatif ... 29
13. EX 5 : mutex. Problème des philosophes ... 30
14. EX 6 : exercice final ... 32
15. Conclusion ... 33
16. Références ... 34
17. Annexe 1 : schéma électronique de la carte d'E/S de la carte cible rpi-xenomai ... 35
18. Annexe 2 : configuration réseau hôtes et cibles ... 36

1. BUT DES TRAVAUX PRATIQUES

Ces Travaux Pratiques ont pour but de présenter la mise en œuvre de l'API (*Application Programming Interface*) POSIX (*Portable Operating System Interface eXchange*) avec l'extension Temps Réel dur Xenomai sur carte Raspberry Pi.

L'API POSIX permet de développer un code source portable avec le langage C sur différents systèmes d'exploitation moyennant une simple recompilation. Avec les systèmes d'exploitation Temps Réel, on est souvent obligé d'utiliser des API propriétaires ce qui nuit à la migration du code source vers un autre système d'exploitation. Une solution à cette problématique est d'utiliser l'API POSIX, ce que nous ferons ici.

Xenomai est une solution de migration d'applications développées avec des API propriétaires.

Xenomai propose 2 configurations possibles :

- Configuration simple noyau. On met en œuvre un système Linux hôte qui peut être éventuellement *patché* avec PREMPT-RT pour porter ou développer des applications. C'est Xenomai *Mercury*.
- Configuration double noyau. On met en œuvre un système Linux complété avec le co-noyau Temps Réel dur de Xenomai pour porter ou développer des applications. C'est Xenomai *Cobalt*.

Xenomai fournit une API générique dite API *Alchemy* qui permet de développer des applications avec Xenomai *Cobalt* et Xenomai *Mercury*.
Xenomai fournit aussi des API d'émulation pour porter des applications développées avec des API propriétaires comme VxWorks ou pSOS.
Xenomai permet enfin de développer des applications POSIX avec Xenomai *Cobalt* (API *Cobalt*) ou avec Xenomai *Mercury* (API POSIX native).

Nous allons dans un premier temps construire notre système Linux Xenomai *Cobalt* pour une carte Raspberry Pi et faire des mesures de performances Temps Réel. Nous développerons ensuite une tâche Temps Réel dur avec l'API *Alchemy* puis une avec l'API POSIX *Cobalt* pour comparer les 2 approches. Nous étudierons enfin quelques aspects de l'API *Cobalt* en mettant en œuvre des communications inter processus IPC en se basant sur les Travaux Pratiques déjà effectués sur µC/OS II.

Mots clés : Raspberry Pi, ARM, Linux embarqué, Xenomai *Cobalt*, langage C, API *Alchemy*, POSIX, API *Cobalt*, IPC

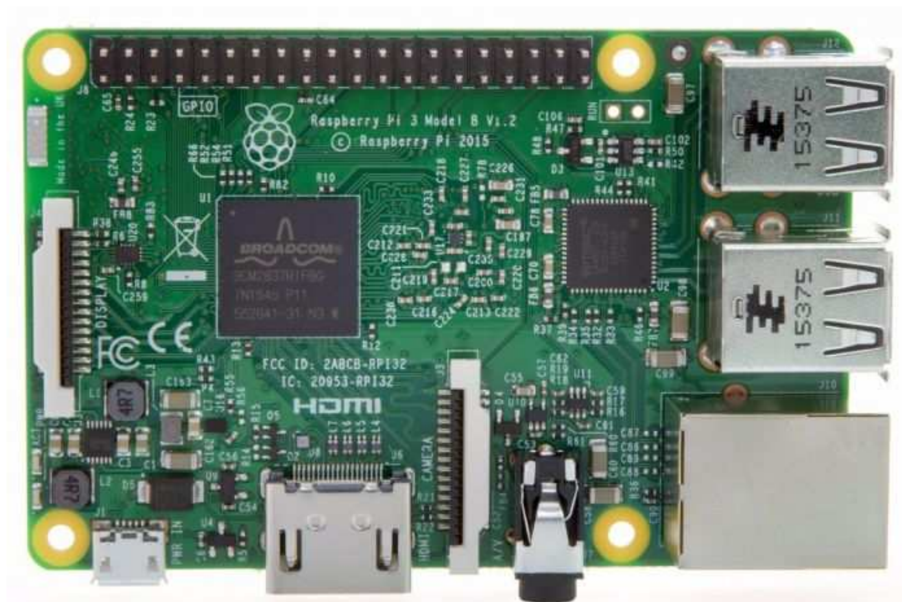
2. INFORMATIONS ESSENTIELLES SUR LA CARTE CARTE RPI-XENOMAI

La carte Raspberry Pi ou RPi est une carte bon marché largement utilisée pour le DIY (*Do It Yourself*) afin de développer de petits systèmes embarqués ou des objets connectés. Le modèle mis en œuvre dans ces TP est la carte Raspberry Pi 3B.

La carte RPi 3B possède ainsi les éléments suivants :

- Un SoC (*System on Chip*) Broadcom BCM2837 avec un processeur quadricœur ARM Cortex-A53 à 1,2 GHz.
- 1 Go de RAM.
- 4 ports USB.
- Sortie vidéo HDMI.
- Sortie audio HDMI et Jack 3,5 mm.
- Support microSD.
- Ethernet 10/100 Mb/s.
- 17 E/S GPIO, 1 UART, 1 bus I2C et 1 bus SPI.

L'image suivante présente la carte cible RPi 3B :



Carte cible RPi 3B

La carte cible rpi-xenomai est une carte « maison » construite autour d'une carte Raspberry Pi 3B. Elle est complétée d'une carte d'entrées/sorties (E/S) connectée à l'aide d'un connecteur 2x20 broches au connecteur d'E/S de la carte RPi.

La carte d'E/S de le carte cible rpi-xenomai possède :

- 6 leds : LED1 à LED6.
- 1 bouton poussoir : BP1.

Le schéma électronique de la carte d'E/S de la carte rpi-xenomai est donné en annexe 1.

La photo est de la carte rpi-xenomai est donnée ci-après.

Carte cible rpi-xenomai

On notera sur le tableau suivant la correspondance entre le numéro de la broche du connecteur 2x20 de la carte RPi et le nom du signal de la carte d'E/S.

Numéro de broche connecteur RPi	Nom du signal carte d'E/S	Fonction
1	V33	3,3 V
2	VCC	5 V
3	SDA	Bus I2C
5	SCL	Bus I2C
6	GND noir	RS232
7	PIN_7	BP1
8	TxD jaune	RS232
10	RxD orange	RS232
11	PIN_11	LED6
12	PIN_12	LED2
16	PIN_16	LED3
18	PIN_18	LED4
22	PIN_22	LED5
25	GND	Masse
26	PIN_26	LED1

Connecteur 2x20 broches et signaux de la carte rpi-xenomai

3. FONCTIONS UTILITAIRES DU BOARD SUPPORT PACKAGE POUR LA CARTE RPI-XENOMAI

Une bibliothèque de fonctions utilitaires a été écrite pour initialiser et utiliser les ressources matérielles de la carte rpi-xenomai. Il s'agit en fait du BSP (*Board Support Package*). Le BSP correspond aux deux fichiers `bsp.c` et `bsp.h`. On notera que le BSP est développé avec la bibliothèque BCM2835 (fichiers `bcm2835.c` et `bcm2835.h`).

Voici la description des fonctions utilitaires et leur prototype :

```
/*
** Fonction: BSP_init()
**              entree(s) : rien
**              sortie(e) : rien
** Description :
** Initialisation ressources et extinction des 6 leds
**/
void  BSP_init(void);


/*
** Fonction: BSP_setLED()
**              entree(s) : numéro de la led de 1 à 6
**              sortie(e) : rien
** Description :
** Allumage d'une led (1 à 6) de la carte rpi-xenomai
**/
void  BSP_setLED(unsigned char lite);


/*
** Fonction: BSP_clrLED()
**              entree(s) : numéro de la led de 1 à 6
**              sortie(e) : rien
** Description :
** Extinction d'une led (1 à 6) de la carte rpi-xenomai
**/
void  BSP_clrLED(usigned char lite);


/*
** Fonction: BSP_release()
**              entree(s) : rien
**              sortie(e) : rien
** Description :
** Libération ressources
**/
void  BSP_release(void);
```

4. TP 0 : PRISE EN MAIN

- Démarrer le PC sous Linux. Se connecter sous le nom **guest**, mot de passe : **guest** ☺.

- Se créer un répertoire de travail à son nom et s'y placer :
```
host% cd
host% mkdir mon_nom
host% cd mon_nom
```

- Etablir le schéma de l'environnement de développement :
  - Matériels.
  - Liaisons : série, réseau…
  - Logiciels et OS utilisés.
  - Adresses IP du PC de développement (hôte ou *host*) et de la carte cible (cible ou *target*).

- Se connecter à la carte d'évaluation (cible) en utilisant l'outil `minicom` :
```
host% minicom -b 115200 -D /dev/ttyUSB0
```
Pour sortir de `minicom`, il suffit de taper la combinaison de touches : CTRL A, Z pour accéder au menu et taper q pour quitter. On arrêtera le compte à rebours de 3 secondes d'*u-boot* en appuyant sur la touche espace du clavier…

- A quoi sert la macro `gok` ? Quel fichier est téléchargé et à quelle adresse ?
```
U-Boot> print gok
```

- A quoi sert la macro `gor` ? Quel fichier est téléchargé et à quelle adresse ?
```
U-Boot> print gor
```

- A quoi sert la macro `godtb` ? Quel fichier est téléchargé et à quelle adresse ?
```
U-Boot> print godtb
```

- A quoi sert la macro `go` ?
```
U-Boot> print go
```

- A quoi sert la macro `ramboot` ?
```
U-Boot> print ramboot
```

Par la suite, on adoptera les conventions suivantes :
Commande Linux PC hôte :

```
host% commande Linux
```

Commande Linux carte cible :

```
RPi3# commande Linux
```

Commande *u-boot* :

```
U-Boot> commande u-boot
```

Astuce !

Pour éviter de regénérer à chaque fois le système de fichiers *root*, on réalisera une compilation croisée de son application puis on recopiera sous `/tftpboot/` l'exécutable ainsi produit :

```
host% make
host% cp mon_appli /tftpboot
```

On pourra alors télécharger l'exécutable en utilisant la commande `tftp` et lancer l'application :

```
RPi3# tftp -g -r mon_appli @IP_host
RPi3# chmod u+x mon_appli
RPi3# ./mon_appli
```

5. TP 1 : GENERATION DU RAM *DISK* POUR LE NOYAU LINUX XENOMAI

Nous allons dans un premier temps créer un *RAM disk*. Un *RAM disk* est un système de fichiers *root* en mémoire RAM. Il est donc volatile et disparaît au *reboot* ou à l'arrêt de la carte cible.

- Se placer dans son répertoire de travail :
```
host% cd
host% cd mon_nom
```

- Se connecter à la carte cible en utilisant l'outil `minicom` en utilisant une fenêtre de terminal. Ce terminal nous permettra d'interagir avec le système Linux.
  Pour sortir de `minicom`, il suffit de taper la combinaison de touches : CTRL A, Z pour accéder au menu et taper q pour quitter. On arrêtera le compte à rebours de 5 secondes d'*u-boot* en appuyant sur la touche espace du clavier…

- Dans son répertoire à son nom, recopier le fichier `tp-RPi3B+.tgz` sous `~kadionik/` :
```
host% cp /home/kadionik/tp-RPi3B+.tgz .
```

- Décompresser et installer le fichier `tp-RPi3B+.tgz` :
```
host% tar -xvzf tp-RPi3B+.tgz
```

- Se placer ensuite dans le répertoire `RPi3B+/`. **L'ensemble du travail sera réalisé à partir de ce répertoire ! Les chemins seront donnés par la suite en relatif par rapport à ce répertoire...**
```
host% cd RPi3B+
```

- Créer le système de fichiers *root* squelette `root_fs` pour la carte cible RPi. Que fait le *shell script* `goskel` ?
```
host% cd ramdisk
host% ./goskel
```

- Compiler `busybox`. Que fait le *shell script* `go` ?
```
host% cd ramdisk
host% cd busybox
host% ./go
```

- Générer les utilitaires de tests `cyclictest`, `stress`...
```
host% cd tst
host% cd stress
host% ./goinstall
host% cd tst
host% cd schedutils
host% ./goinstall
host% cd tst
host% cd rt-tests
host% ./goinstall
```

- Générer les utilitaires de tests de Xenomai `cyclictest`, `latency`... Que fait le *shell script* `goconfig` ? Que fait le *shell script* `go` ? Que fait le *shell script* `goinstall` ?
```
host% cd xenomai
host% ./goconfig
host% ./go
host% ./goinstall
```

- Générer le système de fichiers *root* final `root_fs` pour la carte cible RPi. Il est demandé à un moment donné de rentrer son mot de passe. Que fait le *shell script* `gorootfs` ? Que fait le *shell script* `goramdisk` ?
```
host% cd ramdisk
host% ./gorootfs
host% sudo ./goramdisk
```

- Installer le nouveau *RAM disk* dans le répertoire de téléchargement d'*u-boot* `/tftpboot/` :
```
host% cd ramdisk
host% ./goinstall
```

6. TP 2 : COMPILATION DU NOYAU LINUX XENOMAI ET *BOOT*

Nous allons voir comment compiler le noyau Linux Xenomai ou tout simplement noyau Xenomai exécuté par le processeur ARM de la carte cible RPi.

- Appliquer le patch Xenomai sur le noyau Linux. Que fait le *shell script* `go-ipipe-4.19.114` ? A quoi correspond la valeur 4.19.114 ?
```
host% cd xenomai
host% ./go-ipipe-4.19.114
```

- Compiler le noyau Xenomai pour la carte cible RPi. Que fait le *shell script* `go` ?
```
host% cd linux-xenomai
host% ./go
```

- Installer le fichier du noyau Xenomai dans le répertoire de téléchargement d'*u-boot* `/tftpboot/`. Que fait le *shell script* `goinstall` ?
```
host% ./goinstall
```

- Depuis *u-boot* de la carte cible RPi, lancer la commande suivante. Quels sont les fichiers téléchargés depuis le PC hôte en RAM de la carte RPi et quel est leur rôle ?
```
U-Boot> run ramboot
```

- Observer les traces de boot du noyau Linux dans la fenêtre `minicom` :

```
Starting kernel ...

Booting Linux on physical CPU 0x0
Linux version 4.19.114-v7 (kadionik@ipcchip) (gcc version 4.8.3 20140320 (prerelease)
(Sourcery CodeBench Lite 2
014.05-29)) #5 SMP Mon Jun 29 17:26:28 CEST 2020
CPU: ARMv7 Processor [410fd034] revision 4 (ARMv7), cr=10c5383d
CPU: div instructions available: patching division code
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
OF: fdt: Machine model: Raspberry Pi 3 Model B
Memory policy: Data cache writealloc
random: get_random_bytes called from start_kernel+0x98/0x4a4 with crng_init=0
percpu: Embedded 19 pages/cpu s47424 r8192 d22208 u77824
Built 1 zonelists, mobility grouping on.  Total pages: 240555
Kernel command line: earlyprintk dwc_otg.lpm_enable=0 console=ttyAMA0,115200 console=tty1
root=/dev/ram ramdisk_
size=131072 rootfstype=ext4 rootwait rw
Dentry cache hash table entries: 131072 (order: 7, 524288 bytes)
Inode-cache hash table entries: 65536 (order: 6, 262144 bytes)
. . .
NR_IRQS: 16, nr_irqs: 16, preallocated irqs: 16
sched_clock: 32 bits at 1000kHz, resolution 1000ns, wraps every 2147483647500ns
clocksource: timer: mask: 0xffffffff max_cycles: 0xffffffff, max_idle_ns: 1911260446275 ns
I-pipe, 1.000 MHz clocksource, wrap in 4294967 ms
clocksource: ipipe_tsc: mask: 0xffffffffffffffff max_cycles: 0x1d854df40, max_idle_ns:
3526361616960 ns
bcm2835: system timer (irq = 27)
arch_timer: cp15 timer(s) running at 19.20MHz (phys).
I-pipe, 19.200 MHz clocksource, wrap in 960767920505705 ms
clocksource: ipipe_tsc: mask: 0xffffffffffffffff max_cycles: 0x46d987e47, max_idle_ns:
440795202767 ns
clocksource: arch_sys_counter: mask: 0xffffffffffffff max_cycles: 0x46d987e47, max_idle_ns:
440795202767 ns
sched_clock: 56 bits at 19MHz, resolution 52ns, wraps every 4398046511078ns
Switching to timer-based delay loop, resolution 52ns
Interrupt pipeline (release #8)
Console: colour dummy device 80x30
console [tty1] enabled
```

```
Calibrating delay loop (skipped), value calculated using timer frequency.. 38.40 BogoMIPS
(lpj=192000)
pid_max: default: 32768 minimum: 301
. . .
NET: Registered protocol family 1
RPC: Registered named UNIX socket transport module.
RPC: Registered udp transport module.
RPC: Registered tcp transport module.
RPC: Registered tcp NFSv4.1 backchannel transport module.
Trying to unpack rootfs image as initramfs...
rootfs image is not initramfs (no cpio magic); looks like an initrd
Freeing initrd memory: 3096K
[Xenomai] scheduling class idle registered.
[Xenomai] scheduling class rt registered.
I-pipe: head domain Xenomai registered.
[Xenomai] Cobalt v3.1
Initialise system trusted keyrings
workingset: timestamp_bits=14 max_order=18 bucket_order=4
FS-Cache: Netfs 'nfs' registered for caching
NFS: Registering the id_resolver key type
. . .
3f201000.serial: ttyAMA0 at MMIO 0x3f201000 (irq = 81, base_baud = 0) is a PL011 rev2
mmc0: host does not support reading read-only switch, assuming write-enable
mmc0: new high speed SDHC card at address 59b4
mmcblk0: mmc0:59b4 USD   7.47 GiB
console [ttyAMA0] enabled
. . .
smsc95xx v1.0.6
smsc95xx 1-1.1:1.0 eth0: register 'smsc95xx' at usb-3f980000.usb-1.1, smsc95xx USB 2.0
Ethernet, b8:27:eb:f0:98:aa
random: crng init done
EXT4-fs (ram0): mounted filesystem with ordered data mode. Opts: (null)
VFS: Mounted root (ext4 filesystem) on device 1:0.
devtmpfs: mounted
Freeing unused kernel memory: 1024K
Run /sbin/init as init process
EXT4-fs (ram0): re-mounted. Opts: (null)
uart-pl011 3f201000.serial: no DMA platform data

Please press Enter to activate this console.
RPi3:/#
```

1 TP 3 : MESURE DE TEMPS DE LATENCE AVEC LE NOYAU LINUX XENOMAI

6.1. Outils standards

Nous allons mesurer des temps de latence sur le noyau Xenomai dans le cas d'un noyau non stressé puis dans le cas d'un noyau stressé.

Pour stresser le noyau, on utilisera l'utilitaire `stress`.

- Dévalider le *throttling* :
  `RPi3:# echo -1 > /proc/sys/kernel/sched_rt_runtime_us`

- Dévalider l'anticipation sur la latence minimale de Xenomai :
  `RPi3:# echo 0 > /proc/xenomai/latency`

Noyau Xenomai non stressé. Outils standards :

- Lancer `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
  `RPi3:# cyclictest -n -p 99 -i 5000`

Noyau Xenomai stressé. Outils standards :

- Stresser le noyau avec `stress` :
  `RPi3:# stress -c 50 -i 50 &`

- Lancer `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
  `RPi3:# cyclictest -n -p 99 -i 5000`

Noyau Xenomai non stressé. Outils Xenomai :

On utilisera maintenant les outils Xenomai qui se trouvent dans le répertoire `/usr/xenomai/`.

- Lancer l'outil Xenomai `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
  `RPi3:# /usr/xenomai/demo/cyclictest -n -p 99 -i 5000`

- On utilise maintenant l'outil Xenomai `latency` dans 3 modes différents. A quoi correspondent ces 3 modes ? Noter pour les 3 modes le temps de latence maximum au bout de 5 minutes de tests :

```
RPi3:# /usr/xenomai/bin/latency -t0 -p 5000
RPi3:# /usr/xenomai/bin/latency -t1 -p 5000
RPi3:# /usr/xenomai/bin/latency -t2 -p 5000
```

Noyau Xenomai stressé. Outils Xenomai :

- Stresser le noyau avec `stress` :
  `RPi3:# stress -c 50 -i 50 &`

- Lancer l'outil Xenomai `cyclictest`. Noter le temps de latence maximum au bout de 5 minutes de tests :
  `RPi3:# /usr/xenomai/demo/cyclictest -n -p 99 -i 5000`

- Lancer l'outil Xenomai `latency` dans les 3 modes. Noter pour les 3 modes le temps de latence maximum au bout de 5 minutes de tests :
  ```
  RPi3:# /usr/xenomai/bin/latency -t0 -p 5000
  RPi3:# /usr/xenomai/bin/latency -t1 -p 5000
  RPi3:# /usr/xenomai/bin/latency -t2 -p 5000
  ```

On complètera le tableau suivant avec les mesures de temps de latence pour une comparaison facile :

Temps de latence en µs	**Linux Xenomai non stressé**	**Linux Xenomai stressé**
`cyclictest standard`		
`cyclictest Xenomai`		
`latency -t0`		
`latency -t1`		
`latency -t2`		

Une mesure pendant juste 5 minutes est-elle suffisante ? Les conditions de stress imposées au noyau sont-elles suffisantes ?

6.2. Outils graphiques

Nous allons répéter les mesures avec des outils supplémentaires pour obtenir des graphiques. 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. Si l'on a une mesure toutes les 1000 µs (1 ms), on en aura ainsi 1000 par seconde. On pourra alors visualiser l'évolution de la latence au cours du temps.

On ne produira ici que les graphiques dans le cas d'un noyau stressé.

Noyau Xenomai stressé. Outils standards :

- Stresser le noyau avec `stress` :
```
RPi3:# stress -c 50 -i 50 &
```

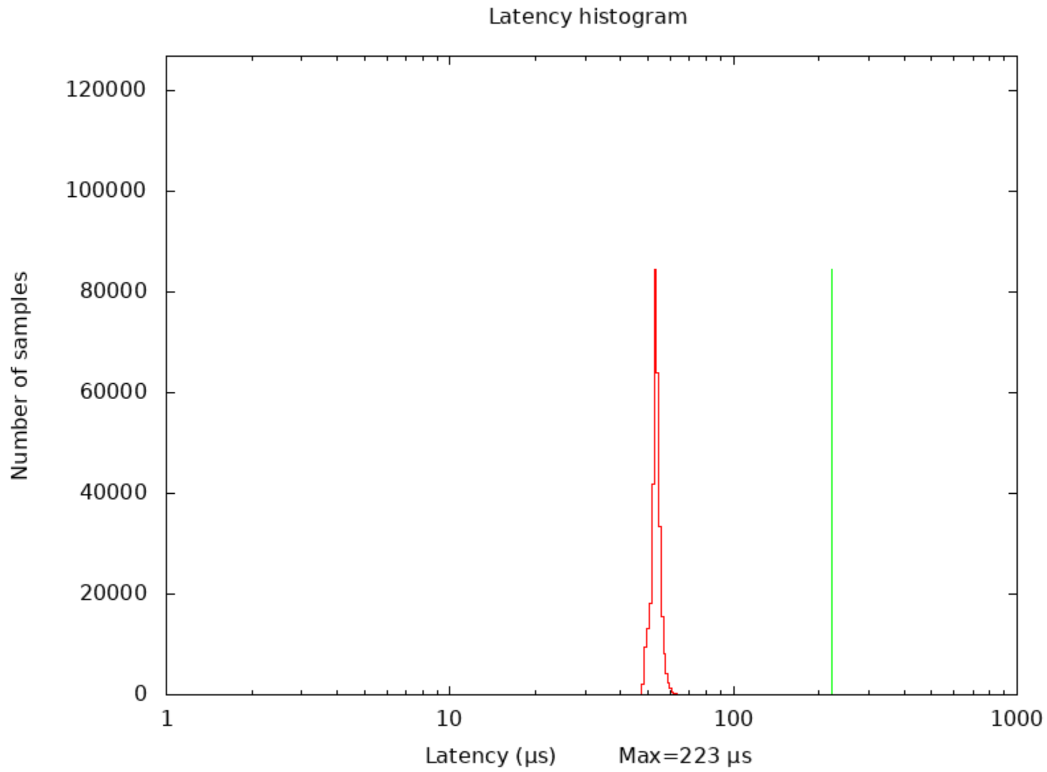
- Lancer `cyclictest` pour une mesure pour 5 minutes de tests. A quoi correspond la valeur 300000 ?
```
RPi3:# cyclictest -l 300000 -n -m -p 99 -i 1000 -v > ons.log
```

- Transférer le fichier `ons.log` (s pour standard) vers le PC hôte :
```
RPi3:# tftp -p -r ons.log @IP_host
```

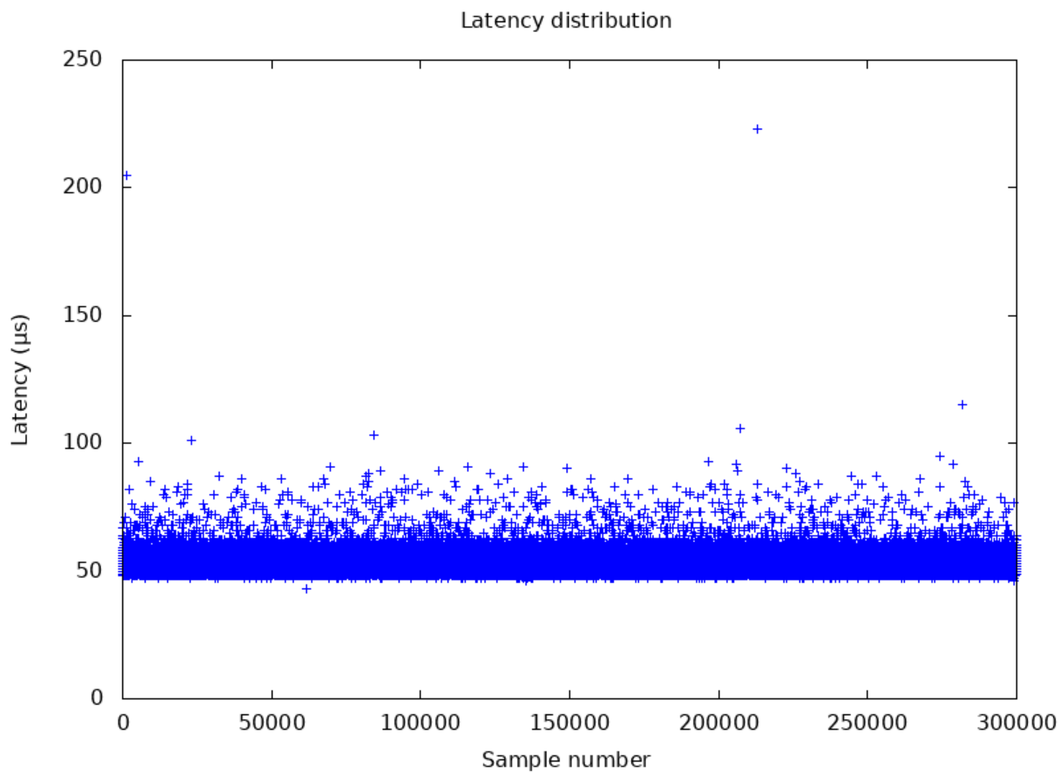
- Recopier le fichier `ons.log` dans son répertoire de travail :
```
host% cd tst
host% cp /tftpboot/ons.log .
```

- Créer les graphiques histogramme et latence avec les *shells scripts* `gohist` et `golat` (sous */bin/*) créés par l'auteur des TP :
```
host% cd tst
host% cp tftpboot/ons.log .
host% gohist ons.log
host% golat ons.log
```

On obtient par exemple les graphiques suivants :



Histogramme avec `cyclictest` standard sur noyau stressé



Latence avec `cyclictest` standard sur noyau stressé

Commenter les résultats obtenus.

Noyau Xenomai stressé. Outils Xenomai :

- Stresser le noyau avec `stress` :
  ```
  RPi3:# stress -c 50 -i 50 &
  ```

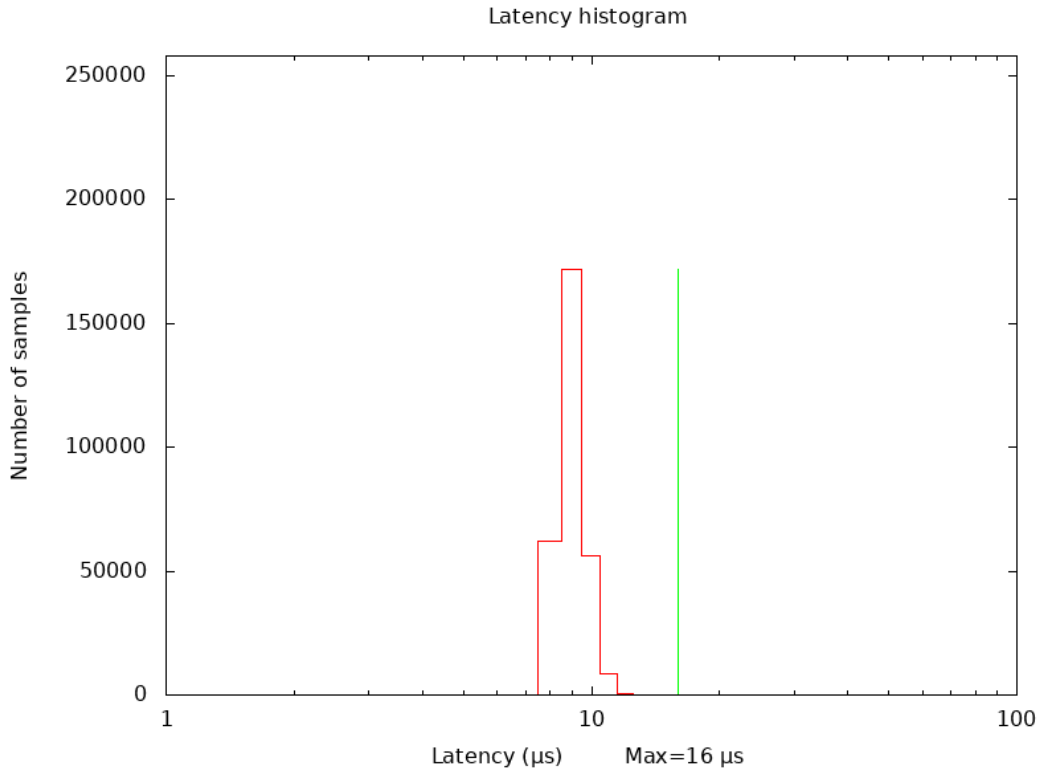
- Lancer l'outil Xenomai `cyclictest` pour une mesure pour 5 minutes de tests :
  ```
  RPi3:# /usr/xenomai/demo/cyclictest -l 300000 -n -m -p 99 -i
  1000 -v > onx.log
  ```

- Transférer le fichier `onx.log` (x pour Xenomai) vers le PC hôte :
  ```
  RPi3:# tftp -p -r onx.log @IP_host
  ```

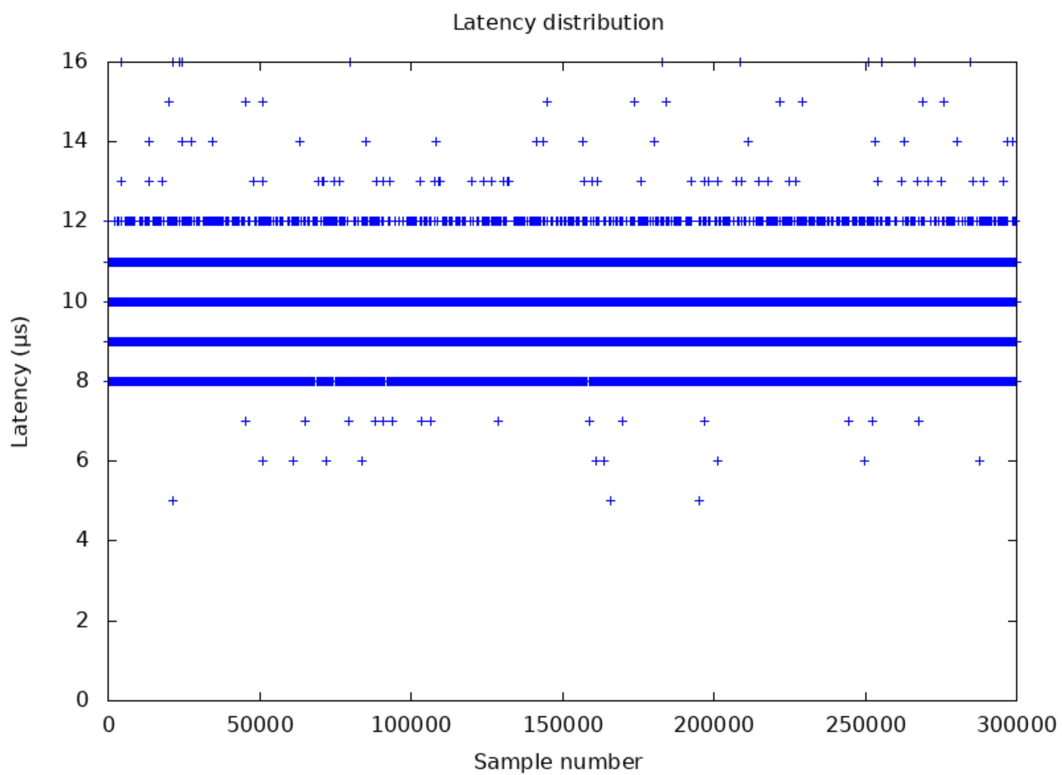
- Recopier le fichier `onx.log` dans son répertoire de travail :
  ```
  host% cd tst
  host% cp /tftpboot/onx.log .
  ```

- Créer les graphiques histogramme et latence :
  ```
  host% cd tst
  host% cp tftpboot/onx.log .
  host% gohist onx.log
  host% golat onx.log
  ```

On obtient par exemple les graphiques suivants :



Histogramme avec `cyclictest` Xenomai sur noyau stressé



Latence avec `cyclictest` Xenomai sur noyau stressé

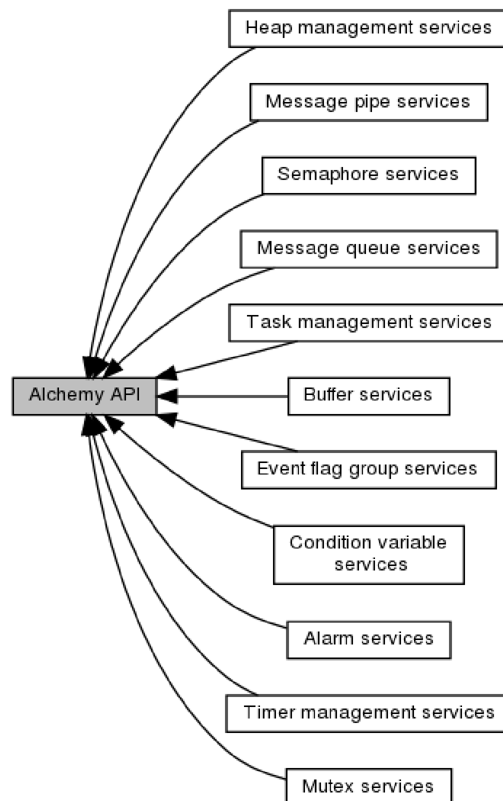
Commenter les résultats obtenus. Les comparer aux résultats précédents.

7. TP 4 : APPLICATION HELLO WORLD AVEC L'API ALCHEMY

Nous allons faire une compilation croisée de la célèbre application « *Hello World!* » avec l'API native de Xenomai.

Depuis la version 3 de Xenomai, l'API native est appelée *Alchemy* et elle est considérée maintenant comme un *skin* (comme VxWorks…), l'API de base (et donc native) étant désormais l'API POSIX ou API *Cobalt*.

La figure suivante présente l'API *Alchemy* :



API *Alchemy*

- Se placer dans le répertoire `tst/hello_native/` et étudier le fichier source `hello_native.c` :
  `host% cd tst/hello_native`

Le code source est le suivant :

```
#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

void task1() {

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

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

int main() {

  // Pas de memory swapping
  mlockall(MCL_CURRENT|MCL_FUTURE);

  // Creation et demarrage de la tache Xenomai
  rt_task_create(&rt_task1, "task1", 0, PRIO1, 0);
  rt_task_start(&rt_task1, &task1, 0);

  sleep(10);

  // Destruction de la tache Xenomai
  rt_task_delete(&rt_task1);

  exit(0);
}
```

On note que :

- `rt_task_create()` : permet de créer une tâche Xenomai.
- `rt_task_start()` : permet de lancer la tâche.
- `rt_task_delete()` : permet de détruire la tâche.
- `rt_task_sleep()` : permet d'endormir la tâche.
- `rt_task_set_periodic()` : permet de rendre périodique la tâche.
- `rt_task_wait_period()` : permet d'attendre la prochaine expiration de la période de la tâche.

- A quoi sert l'appel système `mlockall()` ? Pourquoi utilise-t-on l'appel `rt_printf()` et non l'appel classique `printf()` ? La tâche Xenomai créée est-elle périodique ? Quelle est sa priorité ? Peut-on créer une tâche de priorité supérieure ?

- Compiler l'application `hello_native` pour la carte cible RPi3. Que fait le *shell script* `go` ?
```
host% ./go
```

- Installer l'application `hello` dans le système de fichiers *root* qui servira de base au *RAM disk* :
```
host% ./goinstall
```

- Regénérer le *RAM disk* :
```
host% cd ramdisk
host% sudo ./goramdisk
host% ./goinstall
```

- Relancer le noyau Linux depuis *u-boot* :
```
U-Boot> run ramboot
```

- Tester l'application `hello_native`.

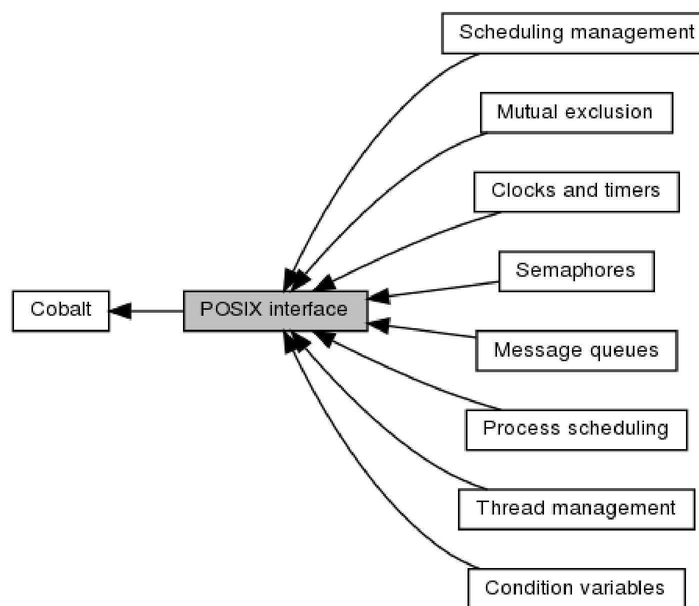
Par la suite, afin d'éviter de regénérer le *RAM Disk* et redémarrer le noyau Xenomai à chaque exercice, on pourra télécharger simplement l'exécutable depuis la carte cible RPi en utilisant la commande `tftp` et lancer l'application :
```
RPi3# tftp -g -r hello_native @IP_host
RPi3# chmod u+x hello_native
RPi3# ./hello_native
```

8. EX 0 : APPLICATION HELLO WORLD AVEC L'API POSIX COBALT

Nous allons faire une compilation croisée de la célèbre application « *Hello World!* » avec l'API POSIX de Xenomai.

Avec Xenomai 3, l'API POSIX (ou API *Cobalt*) est celle de base alors qu'avant c'était un *skin*.

La figure suivante présente l'API POSIX *Cobalt* :



API *Cobalt*

- Se placer dans le répertoire `tst/ex0/` et étudier le fichier source `ex0.c` :
  `host% cd tst/ex0`

Le code source est le suivant :

```
#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;

void *task1() {
  struct timespec ts;

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

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

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

    clock_nanosleep(CLOCK_REALTIME, 0, &ts, NULL);
  }
  pthread_exit(NULL);
}

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);
}
```

On note que :

- `pthread_attr_init()` : permet d'initialiser une structure `pthread_attr_t` d'un *thread* Xenomai.
- `pthread_attr_setschedpolicy()` : permet de choisir l'ordonnanceur Temps Réel `SCHED_FIFO`.
- `pthread_attr_setschedparam()` : permet de fixer la priorité du *thread* via la structure `sched_param`.
- `pthread_create()` : permet de créer et lancer un *thread*.
- `pthread_cancel()` : permet de détruire un *thread*.
- `pthread_exit()` : fin normal d'un *thread*.
- `pthread_join()` : attente de la fin d'un *thread*. Equivalent à `wait()` pour un processus.

- A quoi sert l'appel `clock_nanosleep()` et comment marche-t-il ?

- Compiler l'application `main` pour la carte cible RPi et recopier l'exécutable sous `/tftpboot/` :

```
host% ./go
host% ./goinstall
```

- Transférer par `tftp` l'application `main` dans la carte cible RPi et la tester.

9. EX 1 : MULTITHREADING. CHENILLARD, PLUS UN ET HELLO WORLD

Le but de ce TP est de faire exécuter par le noyau Xenomai 3 *threads* distincts.

On modifiera le fichier `main.c` avec les conditions suivantes :

- `main` : initialisation de la carte, création et lancement des *threads* `thread1`, `thread2` et `thread3`. Attente de la fin des *threads*.
- *Thread* `thread1` : affiche à l'écran de *Hello Wor*ld chaque seconde.
- *Thread* `thread2` : chenillard sur les leds 1 à 6.
- *Thread* `thread3` : fait du « plus 1 » d'un compteur toutes les 2 secondes et affiche la valeur courante du compteur.

- Modifier le programme `main.c` suivant les conditions précédentes. On utilisera les fonctions du BSP de la carte rpi-xenomai pour contrôler les leds :
```
host% cd ex1
```

- Compiler l'application `main` pour la carte cible RPi et recopier l'exécutable sous `/tftpboot/` :
```
host% ./go
host% ./goinstall
```

- Transférer par `tftp` l'application `main` dans la carte cible RPi et la tester.

Indications :

Pour l'attente dans un *thread*, on peut utiliser la fonction `clock_nanosleep()` ou `nanosleep()` :

Exemple :
```
struct timespec ts;


ts.tv_sec = 0; // En s
ts.tv_nsec = 100000000; // En ns

clock_nanosleep(CLOCK_REALTIME, 0, &ts, NULL);

nanosleep(&ts, NULL);
```

10. EX 2 : MUTEX. GESTION DE L'ACCES EXCLUSIF A UNE RESSOURCE PARTAGEE

Le but de ce TP est de gérer l'accès exclusif à une ressource partagée. Celle-ci sera représentée par la led 1.

On modifiera le fichier `main.c` avec les conditions suivantes :

- `main` : Création et initialisation d'un *mutex* `mutex`. Création et lancement des *threads* `thread1` et `thread2`. Libération de `mutex` au bout de 3 secondes. Attente de la fin des *threads*.
- *Thread* `thread1` : bloqué sur `mutex`. A sa libération, changement d'état de la led 1 puis libération de `mutex`.
- *Thread* `thread2` : bloqué sur `mutex`. A sa libération, changement d'état de la led 1 puis libération de `mutex`.

- Recopier le répertoire `ex1/` dans le répertoire `ex2/` et s'y placer :
```
host% cp -r ex1 ex2
host% cd ex2
```

- Modifier le programme `main.c` suivant les conditions précédentes.

- Compiler l'application `main` pour la carte cible RPi et recopier l'exécutable sous `/tftpboot/`.

- Transférer par `tftp` l'application `main` dans la carte cible RPi et la tester.

<u>Indications :</u>

Si tout va bien, la led 1 clignote.

Pour créer un *mutex* :
```
pthread_mutex_t mutex;

pthread_mutex_init(&mutex, NULL);
```

Pour utiliser le *mutex* :
```
// Prendre le mutex
pthread_mutex_lock(&mutex);

// Libérer le mutex
pthread_mutex_unlock(&mutex);
```

11. EX 3 : MUTEX. SYNCHRONISATION DE THREADS. RENDEZ-VOUS

Le but de ce TP est de synchroniser 2 *threads* à un instant donné dans l'exécution de leur code. Cela s'appelle un rendez-vous.

On modifiera le fichier `main.c` avec les conditions suivantes :

- `main` : Création et initialisation d'un *mutex* `mutex` Création et lancement des *threads* `thread1` et `thread2`. Libération de `mutex` au bout de 3 secondes. Attente de la fin des *threads*.
- *Thread* `thread1` : bloqué sur `mutex`.
- *Thread* `thread2` : donne un RDV au *thread* `thread1` par libération de `mutex` au bout de 3 secondes et allume en conclusion la led 1 pour symboliser le rendez-vous.

- Recopier le répertoire `ex2/` dans le répertoire `ex3/` et s'y placer :
```
host% cp -r ex2 ex3
host% cd ex3
```

- Modifier le programme `main.c` suivant les conditions précédentes.

- Compiler l'application `main` pour la carte cible RPi et recopier l'exécutable sous `/tftpboot/`.

- Transférer par `tftp` l'application `main` dans la carte cible RPi et la tester.

12. EX 4 : MUTEX. RECAPITULATIF

Le but de ce TP est de réaliser un *dispatching* d'un travail par une tâche.

On modifiera le fichier `main.c` avec les conditions suivantes :

- `main` : Création et initialisation de 3 *mutex* `mutex1`, `mutex2` et `mutex3`. Création et lancement des *threads* `thread1`, `thread2` et `thread3` bloqués sur leurs mutex `mutex1`, `mutex2` et `mutex3`. Libération des *mutex* un par un toutes les 2 secondes pour activer chaque *thread*. Attente de la fin des *threads.*
- *Thread* `thread1` : bloqué sur `mutex1`. A sa libération par `main`, allumage de la led 1 puis attente dans une boucle infinie.
- *Thread* `thread2` : bloqué sur `mutex2`. A sa libération par `main`, allumage de la led 2 puis attente dans une boucle infinie.
- *Thread* `thread3` : bloqué sur `mutex3`. A sa libération par `main`, allumage de la led 3 puis attente dans une boucle infinie.

- Recopier le répertoire `ex0/` dans le répertoire `ex6/` et s'y placer :
```
host% cp -r ex2 ex4
host% cd ex4
```

- Modifier le programme `main.c` suivant les conditions précédentes.

- Compiler l'application `main` pour la carte cible RPi et recopier l'exécutable sous `/tftpboot/`.

- Transférer par `tftp` l'application `main` dans la carte cible RPi et la tester.

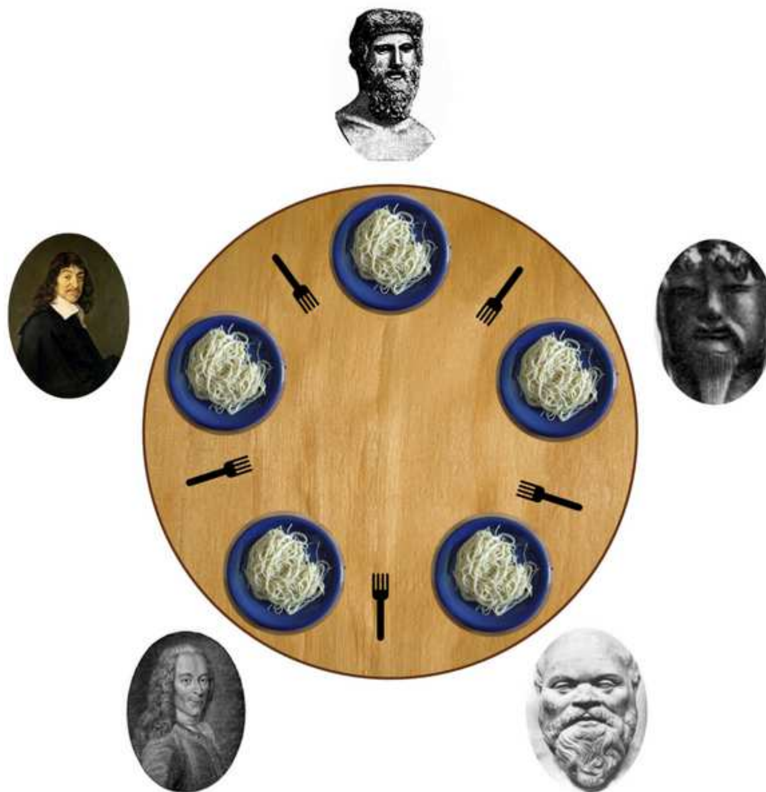
13. EX 5 : MUTEX. PROBLEME DES PHILOSOPHES

Le but de ce TP est traiter le problème des philosophes.

Le problème des philosophes et des spaghettis est un problème classique en théorie informatique et plus particulièrement pour les questions d'ordonnancement des processus. Ce problème a été énoncé par Edsger Dijkstra, l'inventeur du concept des sémaphores.

La situation est la suivante :

- 5 philosophes se trouvent autour d'une table.
- Chacun des philosophes a devant lui un plat de spaghettis.
- A gauche de chaque assiette se trouve une fourchette.



Un philosophe n'a que deux états possibles :

- Penser pendant un temps déterminé.
- Manger.

Des contraintes extérieures s'imposent à cette situation :
- Pour manger, un philosophe a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette et celle qui se trouve à gauche de celle de son voisin de droite (soit les deux fourchettes qui entourent sa propre assiette).
- Si un philosophe n'arrive pas à s'emparer d'une fourchette, il se met à penser pendant un temps déterminé, en attendant de renouveler sa tentative.

Le problème est le suivant : si tous les philosophes essayent en même temps de manger :
1. Ils vont tous vouloir saisir les mêmes fourchettes en même temps.
2. Tous vont échouer.
3. Tous vont se mettre à penser pendant un certain temps.
4. Tous vont renouveler leur tentative en même temps.
5. etc à l'infini.

L'une des principales solutions à ce problème est celle du sémaphore, proposée, comme ce problème, par Edsger Dijkstra.

On utilisera ici les *mutex* POSIX qui correspond à un sémaphore binaire.

- Recopier le répertoire `ex4/` dans le répertoire `ex5/` et s'y placer :
```
host% cp –r ex4 ex5
host% cd ex5
```

- Modifier le programme `main.c` suivant les conditions précédentes.

- Compiler l'application `main` pour la carte cible RPi et recopier l'exécutable sous `/tftpboot/`.

- Transférer par `tftp` l'application `main` dans la carte cible RPi et la tester.

<u>Indications :</u>

On pourra utiliser la fonction `pthread_mutex_trylock()` qui échoue si le *mutex* est déjà pris et qui renvoie alors la valeur EBUSY différente de 0.
On peut aussi utiliser la fonction `pthread_mutex_timedlock()` avec un *timeout* programmable.

Exemple :
```
pthread_mutex_t mutex1;
struct timespec ts;

pthread_mutex_trylock(&mutex1);

ts.tv_sec = 0; // En s
ts.tv_nsec = 10000000; // En ns soit 10 ms de timeout

pthread_mutex_timedlock(&mutex1, &ts);
```

14. EX 6 : EXERCICE FINAL

On réalisera un chronomètre au 1/10 ème de seconde affichant le temps courant sur la liaison série avec gestion du chronomètre :

- Start : touche `s`.
- Stop : touche `t`.
- Remise à zéro : touche `r`.
- Sortie du programme : touche `q`.

<u>Indications :</u>

Pour éviter l'écho du caractère et le mode canonique sous Linux, on pourra utiliser la fonction suivante pour remplacer l'appel `getchar()` :

```
#include <termios.h>

char mygetchar() {
  char c;
  struct termios old, new;

  tcgetattr(STDIN_FILENO, &old);

  new = old;
  new.c_lflag = ~(ICANON | ECHO);
  tcsetattr(STDIN_FILENO, TCSANOW, &new);

  c = getchar();

  tcsetattr(STDIN_FILENO, TCSANOW, &old);

  return(c);
}
```

15. CONCLUSION

On a pu voir la mise en œuvre de Xenomai sur une carte Raspberry Pi.

On a ensuite réalisé des mesures de temps de latence à l'aide de divers outils.

On a pu enfin étudier l'API POSIX *Cobalt* de Xenomai et développer différents programmes pour illustrer un certain nombre de concepts de la programmation Temps Réel.

Ces concepts sont finalement indépendants du système d'exploitation Temps Réel et utiliser en plus l'API POSIX permet d'assurer la portabilité au niveau source quel que soit le système d'exploitation Temps Réel…

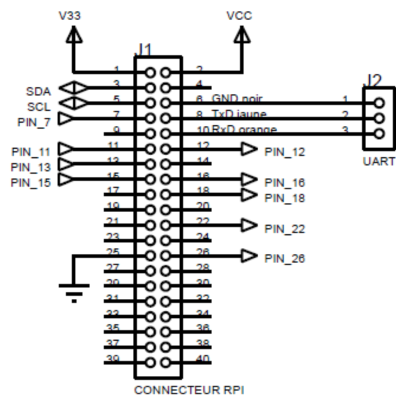
16. REFERENCES

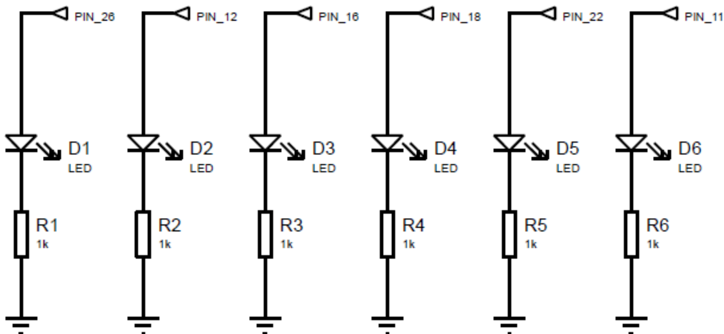
- Carte Raspberry Pi : https://www.raspberrypi.org/
- Projet Xenomai : https://gitlab.denx.de/Xenomai/xenomai/-/wikis/home
- *POSIX Threads programming* : https://computing.llnl.gov/tutorials/pthreads/

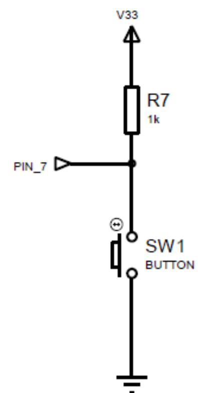
17. ANNEXE 1 : SCHEMA ELECTRONIQUE DE LA CARTE D'E/S DE LA CARTE CIBLE RPI-XENOMAI

CARTE E/S RPI XENOMAI

enseirb/pk/2019







18. ANNEXE 2 : CONFIGURATION RESEAU HOTES ET CIBLES

	POSTE PC01	
	NOM	**ADRESSE IP**
HOTE	rodirula	10.7.4.223
CIBLE	rpi001	10.7.2.118

	POSTE PC02	
	NOM	**ADRESSE IP**
HOTE	sarracenia	10.7.4.225
CIBLE	rpi002	10.7.2.119

	POSTE PC03	
	NOM	**ADRESSE IP**
HOTE	utricularia	10.7.4.227
CIBLE	rpi003	10.7.2.120

	POSTE PC04	
	NOM	**ADRESSE IP**
HOTE	ibicella	10.7.2.112
CIBLE	rpi004	10.7.2.121

	POSTE PC05	
	NOM	**ADRESSE IP**
HOTE	nepenthes	10.7.2.114
CIBLE	rpi005	10.7.2.122

	POSTE PC06	
	NOM	**ADRESSE IP**
HOTE	pinguicula	10.7.2.116
CIBLE	rpi006	10.7.2.123

Masque de sous réseau : **255.255.248.0**

Exemple : configuration réseau de la carte cible `rpi001` :

```
RPi3# ifconfig eth0 10.7.2.118 netmask 255.255.248.0
```