



RAPPORT

IT332 - Systèmes d'exploitation temps réel.

Safae Ouajih
Ghita El Moussi
2020 - 2021

Encadrant : M. PATRICE KADIONIK

Sommaire

1	Introduction	2
2	Mise en oeuvre du noyau temps réel uC/OS II sur processeur BLACKFIN	2
2.1	TP 0 : Prise en main	2
2.2	TP 1 : Tests, mise en oeuvre du noyau μ C/OS II	2
2.3	TP2 : Multitâche, echo sur RS232 et plus un avec le noyau UC/OS II	3
2.4	TP3 : Multitâche, echo sur RS232, chenillard et plus un avec le noyau UC/OS II	3
2.5	TP4 : Sémaphores binaires, gestion de l'accès exclusif à une ressource partagée	4
2.6	TP5 : Sémaphores binaires, synchronisation de tâches, rendez-vous	4
2.7	TP6 : Sémaphore binaires, récapitulatif	4
2.8	TP7 : Sémaphore binaire, problème des philosophes	5
2.9	TP8 : Gestion du temps	6
3	Mise en oeuvre de l'extention temps réel XENOMAI COBALT sur une carte RPI	7
3.1	TP 0 :Prise en main	7
3.2	TP 1 :Génération du RAM disk pour le noyau Linux Xenomai	7
3.3	TP 2 :Compilation du noyau Linux Xenomai et boot	7
3.4	TP 3 :Mesure de temps de Latence avec le noyau Linux Xenomai	8
3.4.1	Outils standards	8
3.4.2	Outils graphiques	8
3.5	TP 4 :Application Hello World avec l'API Alchemy	10
3.6	EX 0 :Application Hello World avec l'API POSIX Cobalt	11
3.7	EX 1 :Multithreading. Chenillard, plus un et Hello World	11
3.8	EX 2 :Mutex. Gestion de l'accès exclusif à une ressource partagée	11
3.9	EX 3 :Mutex. Synchronisation de threads. Rendez-vous	12
4	Conclusion	12

1 Introduction

Le but de ce TP, dans une première partie, est la mise en oeuvre du noyau Temps Réel $\mu\text{C}/\text{OS II}$ sur une carte d'évaluation Blackfin BF537 ; une carte d'évaluation d'Analog Devices permettant de tester le processeur de traitement du signal Blackfin. Le processeur Blackfin supporte $\mu\text{C}/\text{OS II}$ que nous allons utiliser durant la première partie de ce TP.

Dans un premier temps, nous allons présenter le noyau temps réel $\mu\text{C}/\text{OS II}$. $\mu\text{C}/\text{OS II}$ permet d'exécuter plusieurs tâches sur un processeur. En effet, la tâche de plus forte priorité est exécutée en premier. Un noyau temps réel assure l'ordonnancement en fonction de la priorité des tâches. Le noyau $\mu\text{C}/\text{OS II}$ permet de diviser un projet en plusieurs tâches indépendantes, mais une seule tâche à la fois est exécutée par le processeur.

La deuxième partie de ce TP sert à présenter la mise en oeuvre de l'API POSIX avec l'extension Temps Réel dur Xenomai sur une carte Raspberry Pi. Nous allons dans un premier temps étudier le temps de latence et mesurer les performances temps réel. Nous étudierons des aspects de communication entre les tâches en se basant sur les méthodes utilisées durant la première partie.

2 Mise en oeuvre du noyau temps réel $\mu\text{C}/\text{OS II}$ sur processeur BLACKFIN

2.1 TP 0 : Prise en main

La première étape consiste à se connecter à la carte Blackfin. La commande Minicom déjà configurée assure cette connection.

Les commandes du bootloader u-boot sont comme suit :

- Télécharger par le réseau Ethernet le fichier app.elf :
bfin> tftp app.elf
- Lancer et exécuter le fichier app.elf
bfin> bootelf.
- Mettre à zéro une zone mémoire :
bfin> mw.b @adresse 0 4

2.2 TP 1 : Tests, mise en oeuvre du noyau $\mu\text{C}/\text{OS II}$

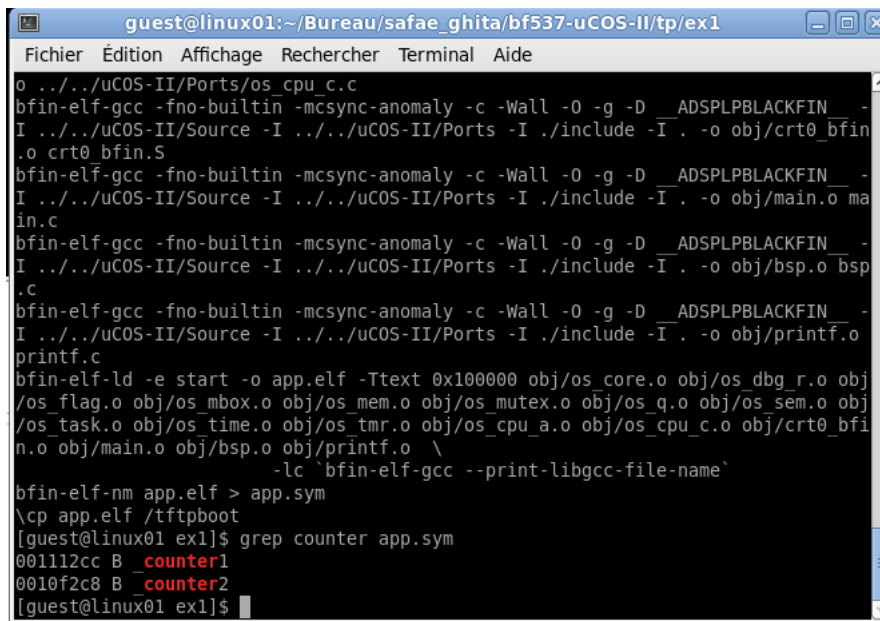
Nous allons initialement analyser le fichier main.c. Il existe trois fonctions dans ce fichier : main(), task1(), task2() et rootTask().

- La fonction main() permet d'initialiser les leds de la carte Blackfin, initialiser le noyau $\mu\text{C}/\text{OS II}$ et créer la tâche principale rootTask
- La tâche rootTask crée les deux tâches task1 et task2 puis fait clignoter la led 6 de la carte Blackfin et émet un point sur la liaison série tous les 100 ticks (1 s).
- La tâche task1 incrémente le compteur counter1 tous les 10 ticks (100 ms).
- La tâche task2 incrémente le compteur counter2 tous les 10 ticks (100 ms).

L'analyse du fichier Makefile permet d'extraire les informations suivantes :

- L'adresse du point d'entrée de l'application est : 0x10000.
- Le compilateur croisé utilisé est gcc.
- Le fichier binaire créé est app.elf de format elf.
- La table des symboles se trouve dans le fichier app.sym.

En analysant le fichier de la table des symboles, nous retrouvons l'adresse de la valeur courante des compteurs



```
guest@linux01:~/Bureau/safae_ghita/bf537-uCOS-II/tp/ex1
Fichier  Édition  Affichage  Rechercher  Terminal  Aide
o ../../uCOS-II/Ports/os_cpu.c
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D __ADSPLBLACKFIN__ -
I ../../uCOS-II/Source -I ../../uCOS-II/Ports -I ./include -I . -o obj/crt0_bfin
.o crt0_bfin.S
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D __ADSPLBLACKFIN__ -
I ../../uCOS-II/Source -I ../../uCOS-II/Ports -I ./include -I . -o obj/main.o ma
in.c
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D __ADSPLBLACKFIN__ -
I ../../uCOS-II/Source -I ../../uCOS-II/Ports -I ./include -I . -o obj/bsp.o bsp
.c
bfin-elf-gcc -fno-builtin -mcsync-anomaly -c -Wall -O -g -D __ADSPLBLACKFIN__ -
I ../../uCOS-II/Source -I ../../uCOS-II/Ports -I ./include -I . -o obj/printf.o
printf.c
bfin-elf-ld -e start -o app.elf -Ttext 0x100000 obj/os_core.o obj/os_dbg_r.o obj
/os_flag.o obj/os_mbox.o obj/os_mem.o obj/os_mutex.o obj/os_q.o obj/os_sem.o obj
/os_task.o obj/os_time.o obj/os_tmr.o obj/os_cpu_a.o obj/os_cpu_c.o obj/crt0_bfi
n.o obj/main.o obj/bsp.o obj/printf.o \
-lc `bfin-elf-gcc --print-libgcc-file-name`
bfin-elf-nm app.elf > app.sym
\cp app.elf /tftpboot
[guest@linux01 ex1]$ grep counter app.sym
001112cc B _counter1
0010f2c8 B _counter2
[guest@linux01 ex1]$
```

- L'adresse de la valeur courante du compteur counter1 est : 0011 12CC.
- L'adresse de la valeur courante du compteur counter2 est : 0010 F2C8.

On peut mettre à zéro les compteurs counter1 et counter2 on utilise la commande suivante :

bfin> mw.b @adresse 0 4

Les commandes suivantes sont utilisées afin de télécharger le fichier binaire et l'exécuter pour tester le fonctionnement de l'application.

host% cp app.elf /tftpboot

bfin> tftp app.elf

bfin> bootelf

On relève la valeur courante des compteurs : les compteurs ont la même valeur car chaque tâche incrémente le compteur à son tour.

2.3 TP2 : Multitâche, echo sur RS232 et plus un avec le noyau UC/OS II

Le but de ce TP est d'écrire un programme d'écho sur le port série de la carte Blackfin avec en concurrence l'incrémentation des deux compteurs.

- Les tâches 1 et 2 restent inchangées (incrémentation des compteurs)
- La tâche rootTask lance les tâches 1 et 2 et ensuite programme un écho sur le port série de la carte Blackfin.

Chaque compteur s'incrémente toutes les 100 ms de 1 et on affiche leurs valeurs.

2.4 TP3 : Multitâche, echo sur RS232, chenillard et plus un avec le noyau UC/OS II

Le but de cette partie est d'exécuter trois tâches différentes avec :

- La tâche 1 effectue un chenillard sur les leds 1 à 6.

- La tâche 2 reste inchangée (incrémentation du compteur)
- La tâche rootTask lance les tâches 1 et 2 et ensuite programme un écho sur le port série de la carte Blackfin.

2.5 TP4 : Sémaphores binaires, gestion de l'accès exclusif à une ressource partagée

Le but de ce TP est de gérer l'accès exclusif à une ressource partagée qui est l'état du led 1 dans notre cas. Nous allons utiliser les sémaphores pour gérer l'accès à cette ressource. En fait, les sémaphores servent à synchroniser les tâches. Un sémaphore est composé d'un compteur à deux opérateurs P() et V() pour prendre et vendre. Une valeur positive du compteur indique le nombre d'accès disponible alors qu'une valeur négative indique le nombre de processus en attente.

- La tâche rootTask crée un sémaphore binaire et lance les tâches 1 et 2 et libère le sémaphore.
- La tâche 1 bloquée sur le sémaphore, change l'état de la led et libère le sémaphore.
- La tâche 2 bloquée sur le sémaphore, change l'état de la led et libère le sémaphore.

Pour créer le sémaphore binaire, on utilise la primitive : *OSSemCreate()*. On l'initialise à 0 (pas encore libéré) en l'indiquant dans la primitive précédente : *OSSemCreate(0)*. Ensuite, on crée les tâches 1 et 2 avec la primitive : *OSTaskCreateExt()*. Pour finir, on utilise la primitive *OSSemPost()* pour libérer le sémaphore. Cela correspond à la fonction Vendre décrite précédemment, ainsi, le compteur du sémaphore est incrémenté, ce qui autorise les autres processus (donc les autres tâches) à accéder aux données.

La **tâche task1** et la **tâche task2** sont bloquées sur le sémaphore et changent l'état de la led 1 lors de la libération du sémaphore. On utilise *OSSemPend()* pour attendre la libération du sémaphore. Le compteur du sémaphore est décrémenté dès lors que le processus réussit à prendre le sémaphore. Lorsque le sémaphore est libéré, la tâche change l'état de la led et libère le sémaphore.

La led 1 clignote après l'exécution.

2.6 TP5 : Sémaphores binaires, synchronisation de tâches, rendez-vous

Ce cinquième exercice sert à synchroniser deux tâches à un instant donnée dans l'exécution de leur code. Il s'agit d'une des fonctions d'un sémaphore. Nous allons donc utiliser un sémaphore pour réaliser cette synchronisation. Nous allons utiliser la méthode du rendez-vous.

- La tâche rootTask crée un sémaphore binaire et lance les tâches 1 et 2 et libère le sémaphore.
- La tâche est bloquée sur le sémaphore. La primitive *OSSemPend()* permet d'attendre la libération du sémaphore binaire. On affiche "1 " à l'écran afin de savoir quand le sémaphore est attrapé par la tâche.
- La tâche 2 donne rendez-vous à la tâche 1 par libération du sémaphore, ce qui allumera la led 1 et permet à la tâche task1 d'afficher le message. La led est ensuite allumée par la tâche 2.

Après l'exécution "1" s'affiche et la led 1 s'allume.

2.7 TP6 : Sémaphore binaires, récapitulatif

Dans ce TP, nous avons pour objectif de dispatcher un travail avec la tâche rootTask en utilisant des sémaphores pour donner rendez-vous à trois tâches.

- La tâche rootTask crée trois sémaphores binaires et lance les tâches 1, 2 et 3 et libère les sémaphores par la suite. la libération des sémaphores se fait d'une manière respective pour

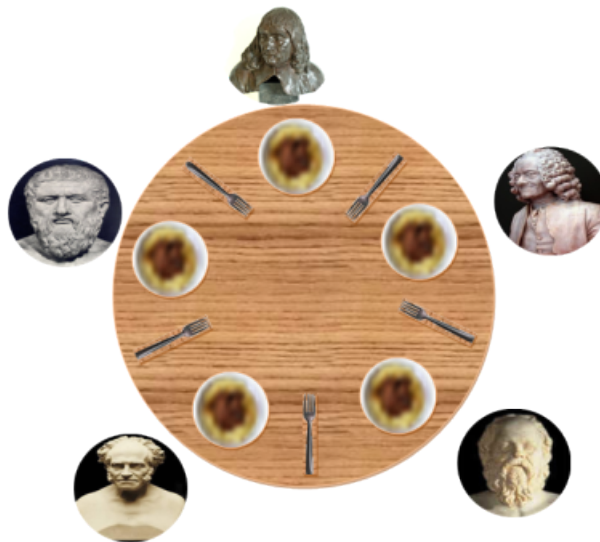
activer les tâches une par une. Le temps utilisé est 1s.

```
157
158     OSTimeDly(100);
159     OSSemPost(sem3);
160
161     OSTimeDly(100);
162     OSSemPost(sem2);
163
164     OSTimeDly(100);
165     OSSemPost(sem1);
166
```

- Les tâches 1,2 et 3 sont bloquées sur le sémaphore. Chaque tâche sert à allumer une led (led1,led2,led3). Chaque tâche attend la libération du sémaphore pour allumer la led correspondante.

Les trois leds s'allument les unes après les autres avec 1s de délais.

2.8 TP7 : Sémaphore binaire, problème des philosophes



Nous allons à travers ce TP aborder le problème classique des philosophes. Cinq philosophes se trouvent autour d'une table, chacun des philosophes a devant lui un plat de spaghetti à gauche de chaque plat de spaghetti se trouve une fourchette. Deux états sont possibles pour un philosophe : penser pendant un temps déterminé ou manger.

Pour manger, un philosophe doit attendre que les fourchettes soient libres ; il a besoin de deux fourchettes : celle qui se trouve à gauche de sa propre assiette, et celle qui se trouve à droite (c'est-à-dire les deux fourchettes qui entourent sa propre assiette) ; si un philosophe n'arrive pas à s'emparer d'une fourchette, il reste affamé en attendant de renouveler sa tentative.

Le problème consiste à trouver un ordonnancement des philosophes tel qu'ils puissent tous manger, chacun à leur tour. Cet ordre est imposé par la solution que l'on considère comme celle de Dijkstra avec les sémaphores.

- La tâche rootTask crée cinq sémaphores binaires et lance les tâches (1 à 5) et libère les séma-

phores par la suite. la libération des sémaphore se fait d'une manière respective pour activer les tâches une par une. Le temps utilisé est 1s.

```

258     OSTimeDly(100);
259     OSSemPost(sem5);
260
261     OSTimeDly(100);
262     OSSemPost(sem4);
263
264     OSTimeDly(100);
265     OSSemPost(sem3);
266
267     OSTimeDly(100);
268     OSSemPost(sem2);
269
270     OSTimeDly(100);
271     OSSemPost(sem1);

```

La première tâche, qui correspond au philosophe 1, essaie d'accéder d'abord à la fourchette 5 notée f5 (sémaphore 5) puis la fourchette 1 notée f1 à travers la fonction `OSSemPend()`. Si la tâche n'a pas réussi à avoir les deux fourchette, quand `OS_result` est égale à `OS_TIMEOUT` (avec `OS= 10 ticks`), le philosophe 1 lâche les deux fourchettes avec `OSSemPost(f5)`. Si la tâche réussit à récupérer les deux fourchettes, elle peut libère les fourchettes après. Le meme principe reste inchangé les autres tâches. Pour illustrer l'algorithme, nous proposons la figure suivante :

```

40 void task1(void *arg) {
41     INT8U OS_result;
42
43     OS_result = 0;
44
45     INT8U err= 0;
46
47
48     while(1) {
49
50
51         OSSemPend(sem5, 0, &err );
52         OSSemPend(sem1, 10, &err );
53         if( err == OS_TIMEOUT){
54             OSSemPost(sem5);
55             goto fin;
56         }
57         BSP_setLED (1);
58         OSTimeDly(10);
59         BSP_clrLED (1);
60         OSSemPost(sem1);
61         OSSemPost(sem5);
62         OSTimeDly(100);
63         fin;;
64         printf("1 ");
65
66     }
67 }

```

2.9 TP8 : Gestion du temps

L'objectif de ce TP est de faire exécuter quatre tâches distinctes par le noyau. Ces tâches affichent un message périodiquement avec des périodes différentes.

- La tâche `rootTask` lance les tâches (1 à 4).
- Les tâches `task1`, `task2`, `task3` doivent afficher un message respectivement toutes les 1, 2 et 10 secondes. Pour cela, on utilise dans chaque tâche `printf(i)` avec `i` le numéro de la tâche. La gestion du temps se fait avec la primitive `OSTimeDelay(t)`.
 Pour `task1`, nous utilisons `OSTimeDelay(10)`.
 Pour `task2`, nous utilisons `OSTimeDelay(50)`.
 Pour `task3`, nous utilisons `OSTimeDelay(100)`.

Après l'exécution, les leds s'allument avec les périodes correspondant.

3 Mise en oeuvre de l'extention temps réel XENOMAI CO-BALT sur une carte RPI

3.1 TP 0 :Prise en main

Pour se connecter à la carte Blackfin, nous utilisons la commande suivante pour indiquer la vitesse (115200 bit/sec) et le port (ttyUSB0) : **host%** *minicom -b 115200 -D /dev/ttyUSB0*

Durant toute cette étude, nous allons utiliser les commandes suivantes ;

Pour Charger l'image du kernel, nous utilisons :

U-Boot> *gok // tftp \$kernel_addr_r <filename of the kernel image>*

U-Boot> *gor // tftp \$ramdisk_addr_r <filename of the initial ramdisk image>*

U-Boot> *godtb // tftp \$fdt_addr_r <filename of the dtb>*

U-Boot> *ramboot//* Cette commande remplace les trois commandes précédentes.

3.2 TP 1 :Génération du RAM disk pour le noyau Linux Xenomai

Nous allons dans cette partie créer le RAM disk, c'est le système de fichiers root en mémoire. Il est volatile et disparaît au reboot de la carte. Dans un premier temps, nous allons créer le système de fichiers root_fs avec la commande suivante dans /RPI3B+/ramdisk :

host% *./goskel*

Nous compilons le busybox avec la commande suivante dans /RPI3B+/ramdisk/busybox :

host% *./go*

Nous générons les utilitaire de tests cyclicttest et stress.

Nous générons les utilitaires de tests de Xenomai cyclicttest, latency ainsi que le système de fichiers root_fs pour la cible avec : **host%** *./gorootfs*

host% *sudo ./goramdisk*

Et finalement, nous installons le RAM disk dans le répertoire de téléchargement d'u-boot /tftpboot/ avec : **host%** *./goinstall*

3.3 TP 2 :Compilation du noyau Linux Xenomai et boot

Cette partie sert à voir les étapes de compilation du noyau Linux Xenomai exécuté par le processeur ARM de la carte RPI.

Nous allons dans un premier temps appliquer le patch Xenomai sue le noyau Linux avec la commande :

host%> *./go-ipipe-4.19.82*

le patch I-pipe installe un pipeline redistribuant les interruptions entre le noyau linux (pour les interruptions non temps réel) et le noyau Cobalt (pour les interruptions temps réel).

Nous compilons le noyau Xenomai avec la commande suivante dans /linux-xenomai

host%> *./go*

Ensuite, nous installons le fichier du noyau Xenomai dans /tftpboot/ avec la commande :

host%> *./goinstall*

Nous utilisons la commande :

U-Boot> *run ramboot*

pour commencer le boot.

3.4 TP 3 : Mesure de temps de Latence avec le noyau Linux Xenomai

3.4.1 Outils standards

Nous allons mesurer le temps de latence sur le noyau Xenomai lorsque le noyau est non stressé et par la suite dans le cas d'un noyau stressé.

— Noyau Xenomai non stressé

En utilisant l'outil standard cyclicttest, On trouve un temps de latence maximum au bout de 5 minutes de 62 us.

Nous lançons l'outil xenomai cyclicttest qui est dans /usr/xenomai/ avec la commande :

RPi3 :# /usr/xenomai/demo/cyclicttest -n -p 99 -i 5000

Nous avons noté que le temps de latence maximum au bout de 5 minute est : 18us

Nous utilisons maintenant l'outil Xenomai latency dans 3 modes différents :

Mode0 : Periodic user mode task.

Mode1 : in-kernel periodic task.

Mode2 : in-kernel timer handler.

Le temps de latence maximum au bout de 5 minutes de test pour chaque mode est le suivant :

Mode0 : 20,082 us

Mode1 : 0,559 us

Mode2 : 5,152 us

— Noyau Xenomai stressé

Pour stresser le noyau, nous utilisons la commande suivante : **RPi3** :# stress -c 50 -i 50 &

En utilisant l'outil standard cyclicttest avec la commande : On trouve un temps de latence maximum au bout de 5 minutes de 58 us. En utilisant l'outil Xenomai cyclicttest /usr/xenomai/, nous avons 18 us en temps de latence. L'outil Xenomai latency lancé dans les trois modes indique le temps de latence au bout de 5 minutes :

Mode0 : 18,146 us

Mode1 : 2,55 us

Mode2 : 4,137 us

Le tableau suivant permet de faire une comparaison facile des résultats.

Temps de latence en us	Linux Xenomai non stressé	Linux Xenomai stressé
Cyclicttest standard	62	58
Cyclicttest Xenomai	18	18
Latency mode0 -t0	20.082	18.146
Latency mode0 -t1	0.559	2.55
Latency mode0 -t2	5.152	4.137

3.4.2 Outils graphiques

Cette partie a pour but l'obtention de graphiques. Uniquement l'outil cyclicttest est utilisé, nous répétons les mesures.

Nous lançons cyclicttest pour une mesure de 5 minute avec la commande :

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

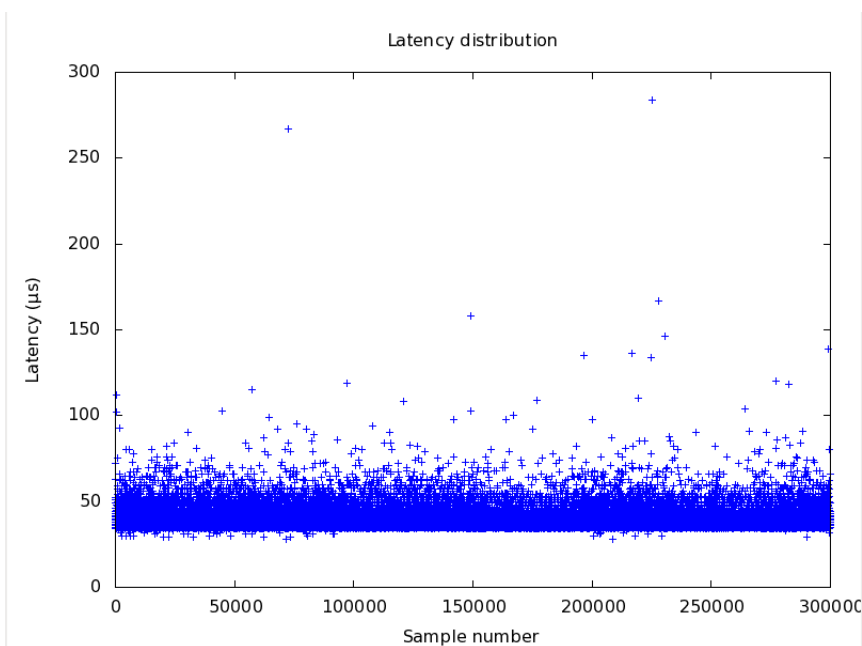
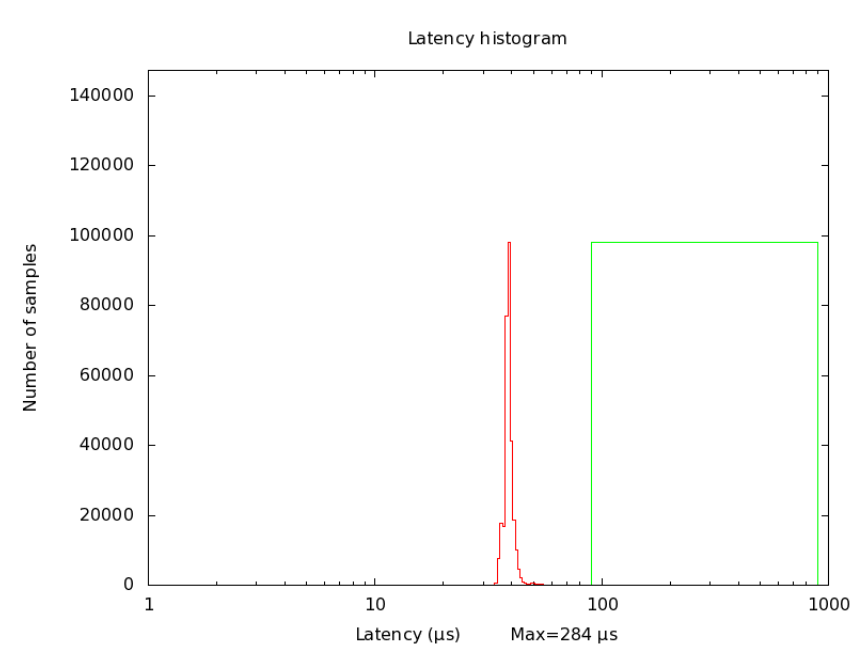
300000 correspond à 5 minute (1000 correspond à 1s).

Le fichier ons.log est ensuite transféré au PC. Nous générons les graphiques histogramme et latence avec les commandes :

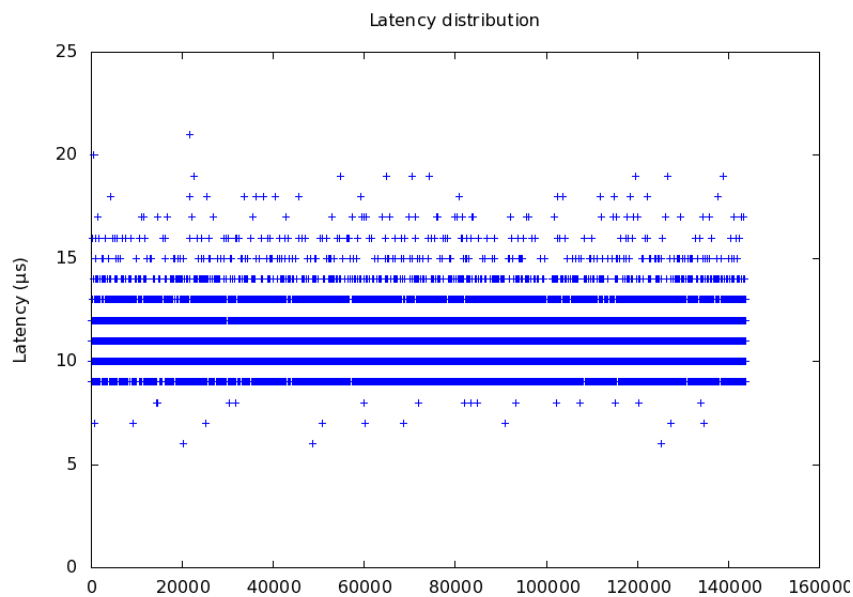
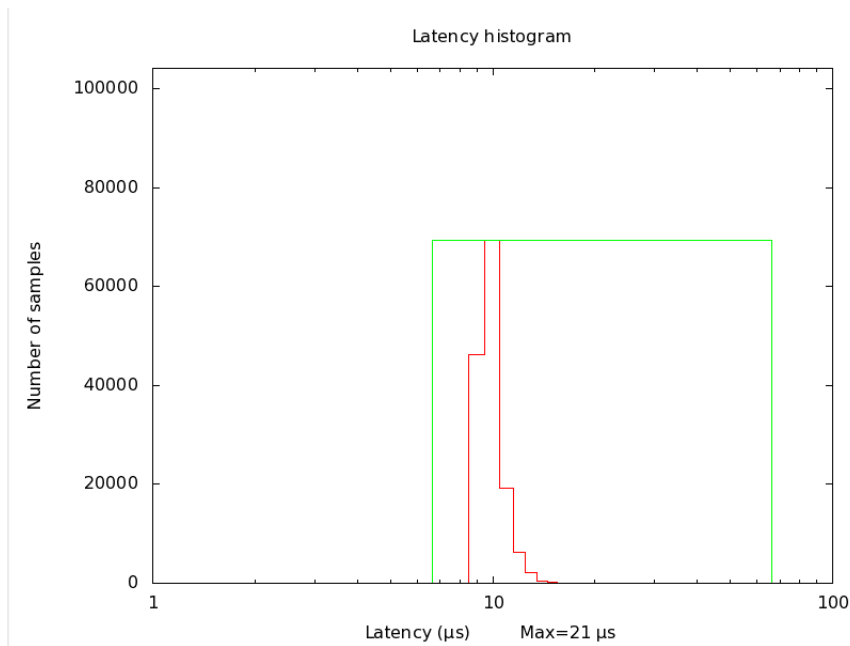
```
host%> gohist ons.log
```

```
host%> golat ons.log
```

Les graphiques 1/ Histogramme avec cyclicttest standard sur noyau non stressé. et 2/ Latence avec cyclicttest standard sur noyau non stressé. sont comme suit :



Nous allons par la suite générer les graphiques précédents avec le noyau stressé. nous lançons l'outil Xenomai cyclicttest après le stress et nous générons les graphiques dans le fichier onx.log. Les résultats sont comme suit :



3.5 TP 4 :Application Hello World avec l'API Alchemy

Cette partie vise à faire une compilation croisée de l'application "hello world" en utilisant l'API native de Xenomai : ALCHEMY.

l'appel système `mlockall()` sert à verrouiller toute la mémoire appartenant au processus, sa pile ... Nous utilisons `rt_printf()` au lieu de `printf()` puisqu'elle ne nécessite pas l'utilisation des bibliothèques C et car l'appel système `printf()` n'est pas Temps Réel.

En compilant l'application "Hello World" avec la commande :

```
host% ./go
```

Pour installer l'application dans le système de fichiers root, nous utilisons la commande :

```
host% ./goinstall
```

Nous générons le RAM disk avec la commande suivant dans le dossier /ramdisk :

host% `./goinstall`

Et nous relançons le noyau Linux avec la commande :

U-Boot» `run ramboot`

3.6 EX 0 :Application Hello World avec l'API POSIX Cobalt

Nous allons réaliser une compilation croisée de l'application "hello world" avec l'API Posix de Xenomai.

La fonction : `clock_nanosleep()` permet à la tâche de dormir pendant un intervalle déterminé avec une précision en nanoseconde.

Nous compilons l'application et recopions l'exécutable sous `/tftpboot/` avec les commandes :

host% `./go`

host% `./goinstall`

L'expression "Hello World from thread1 !" s'affiche périodiquement avec une période de 1s comme indiqué par la fonction `clock_nanosleep()`. La fonction `main()` gère la création, le lancement ainsi que la destruction du tâche avec les primitives suivantes :

- `pthread_create()` : permet de créer et lancer un thread.
- `pthread_cancel()` : permet de détruire un thread.

3.7 EX 1 :Multithreading. Chenillard, plus un et Hello World

Dans cet exercice, nous allons exécuter par le noyau Xenomai trois tâches distinctes. Les tâches sont les suivantes :

- Thread1 : affiche à l'écran Hello World chaque seconde.
- Thread2 : chenillard sur les leds 1 à 6.
- Thread3 : fait du « plus 1 » d'un compteur toutes les 2 secondes et affiche la valeur courante du compteur.

La fonction `main()` permet d'initialiser la carte, créer les tâches et attendre leurs fin.

Nous utilisons la primitive `clock_nanosleep()` pour chaque tâche pour avoir les périodes de fonctionnement désirées.

3.8 EX 2 :Mutex. Gestion de l'accès exclusif à une ressource partagée

Le but de ce TP est la gestion d'accès exclusif à une ressource partagée représentée par la led 1. Un Mutex est une primitive de synchronisation utilisée en programmation informatique pour éviter que des ressources partagées d'un système ne soient utilisées en même temps. Nous utilisons un mutex pour gérer l'accès à la led 1.

Les fonctions suivantes sont utilisées pour gérer le mutex :

Pour créer le mutex : `pthread_mutex_t mutex`

Pour prendre le mutex : `pthread_mutex_lock(&mutex)`

Pour libérer le mutex : `pthread_mutex_unlock(&mutex)`

Les tâches sont les suivantes :

- Thread1 : bloqué sur mutex. A sa libération, changement d'état de la led 1 puis libération de mutex.
- Thread2 : bloqué sur mutex. A sa libération, changement d'état de la led 1 puis libération de mutex.

La fonction `main()` fait la Création ,l'initialisation d'un mutex, la création et le lancement des threads `thread1` et `thread2` ainsi que la libération de mutex au bout de 3 secondes. la variable `mutex` de type

`pthread_mutex_t` est déclarée au début pour qu'elle soit partagée. Un thread seulement récupérera le mutex puis, le mutex étant acquis, change l'état de la led 1 et libère le mutex qui sera récupéré par l'autre thread.

3.9 EX 3 :Mutex. Synchronisation de threads. Rendez-vous

Nous cherchons, dans cette partie, à synchroniser deux threads à un instant donné dans l'exécution de leur code. Nous utilisant la méthode du rendez-vous expliquée précédemment. Les tâches sont :

- Thread1 : bloqué sur un mutex.
- Thread2 : donne un RDV au thread1 par libération de mutex au bout de 3 secondes et allume en conclusion la led 1 pour symboliser le rendez-vous.

Nous utilisons un mutex pour donner rendez vous à la tâche 1.

4 Conclusion

Nous avons réussi, à travers ce TP, à mettre en oeuvre un noyau Temps Réel μ C/OS II sur une carte Blackfin. Dans ce cadre, nous avons pu faire la gestion des tâches (multitasking) ainsi que le rootTask. Une grande partie du TP a comme but de faire communiquer les tâches entre eux, gérer les sémaphores binaires à travers des applications, effectuer une synchronisation des tâches, ou partager des ressources entre eux. On a fait des applications dans la gestion du temps. Nous avons pu manipuler plusieurs fonctions du manuel uC/OS-II. Dans la deuxième partie du TP, nous avons mis en oeuvre l'extention temps réel XENOMAI COBALT sur une carte RPI. Initialement, une étude du temps de latence du noyau est faite pour plusieurs cas (noyau stressé/,on stressé). Nous avons utilisé les outils standards et graphiques de Xenomai pour afficher les résultats. Nous avons manipulé les API ALCHEMY et POSIX Cobalt. Plusieurs applications de gestion des tâches sont utilisées et nous avons pu faire la gestion d'accès aux ressources partagées et la synchronisation des tâches avec le mutex.