

CONSTRUCTION DE SYSTÈMES
EMBARQUÉS SOUS LINUX
Rapport de laboratoire
Master HES-SO

Émilie GSPONER, Grégory EMERY

4 janvier 2016
version 1.0

Table des matières

1	Introduction	4
2	Environnement Linux Embarqué	4
2.1	Objectifs	4
2.2	Informations pratiques	4
2.3	Gravure de la carte SD	5
2.4	Mise en place de l'infrastructure	5
2.4.1	Configuration de la machine virtuelle	5
2.4.2	Accès ssh sans mot de passe	7
2.4.3	Création de l'espace de travail	8
2.5	Debugging de l'application	9
2.6	Travail à réaliser	12
2.7	Configuration du noyau pour monter un usrfs	12
2.8	Test des différents exemples proposés	13
2.8.1	Fibonacci	13
2.8.2	Tracing	13
2.8.3	Core dumps	13
2.8.4	Backtrace	14
2.8.5	System calls	15
2.8.6	Memory leaks	15
2.9	Mise en production de l'ODROID-XU3	16
2.10	Réponse aux questions	17
3	Programmation noyau : Module noyau	18
3.1	Module noyau	18
3.1.1	Exercice 1	18
3.1.2	Exercice 2	20
3.1.3	Exercice 3	21
3.2	Gestion de la mémoire, bibliothèques et fonctions utiles	22
3.2.1	Exercice 4	22
3.2.2	Exercice 5	22
3.3	Accès aux entrées/sorties	23
3.3.1	Exercice 6	23
3.4	Threads du noyau	24
3.4.1	Exercice 7	24
3.5	Mise en sommeil	24
3.5.1	Exercice 8	24
3.6	Gestion des interruptions	25
3.6.1	Exercice 9	25
4	Programmation noyau : Pilotes de périphériques	27
4.1	Pilotes orientés mémoire	27
4.1.1	Exercice 1	27
4.1.2	Exercice 2	27
4.2	Pilotes orientés caractères	28

4.2.1	Exercice 3	28
4.2.2	Exercice 4	29
4.2.3	Exercice 5	30
4.3	Opérations bloquantes	31
4.3.1	Exercice 6	31
4.4	sysfs	32
4.4.1	Exercice 7	32
4.5	ioctl (optionnel)	32
4.5.1	Exercice 8	32
4.6	procfs (optionnel)	32
4.6.1	Exercice 9	32
4.7	Gestionnaires de périphériques	32
4.7.1	Exercice 10	32
5	Programmation système : Système de fichiers	34
5.1	Contexte	34
5.2	Travail à réaliser	34
5.3	Travail réalisé	34
5.3.1	Description	34
5.3.2	Configuration des GPIO	35
5.3.3	Exécution du code	35
5.3.4	Syslog	36
5.3.5	Mesure de performance	36
5.3.6	Amélioration possibles	37
6	Programmation système : Multiprocessing	38
6.1	Processus, signaux et communication	38
6.1.1	Exercice 1	38
6.2	CGroups	39
6.2.1	Exercice 2	39
6.2.2	Exercice 3	42
7	Optimisation : Performances	47
7.1	Installation de perf	47
7.2	Prise en main de perf	47
7.2.1	Exercice 1	47
7.2.2	Exercice 2	48
7.2.3	Exercice 3	48
7.2.4	Exercice 4	49
7.2.5	Exercice 5	49
7.2.6	Exercice 6	50
7.3	Analyse et optimisation d'un programme	50
7.3.1	Exercice 1	50
7.3.2	Exercice 2	51
7.3.3	Exercice 3	51
7.3.4	Exercice 4	51
7.4	Parsing de logs apache	53

7.4.1	Exercice 1	53
7.4.2	Exercice 2	53
7.4.3	Exercice 3	53
7.4.4	Exercice 4	54
7.4.5	Exercice 5	55
8	Mini Projet - Programmation noyau et système	56
8.1	Travail à réaliser	56
8.2	Installation des différentes parties	56
8.2.1	Daemon	56
8.2.2	Module noyau	57
8.2.3	Application utilisateur	57
8.3	Daemon en espace utilisateur	57
8.4	Module de contrôle du ventilateur	58
8.5	Application utilisateur	59
8.6	Mesure de performances	59
8.7	Points à améliorer	59

1 Introduction

Ce rapport présente les résultats obtenus tout au long des travaux pratiques fournis durant le cours de CSEL1, construction de systèmes embarqués sous Linux. Le document est structuré en sections, représentant les séries d'exercices données, en sous-sections présentant les thèmes proposés pour les travaux et en sous-sous-sections pour les réponses à chacune des questions posées dans le document.

Ce cours est effectué avec la cible Odroid XU3¹ et U-Boot² dans le cadre du cours de Master HES-SO en systèmes embarqués, orientation TIN et TIC.

2 Environnement Linux Embarqué

2.1 Objectifs

Ce travail pratique vise les objectifs suivants :

1. Mise en œuvre d'un système embarqué sous Linux
2. Mise en oeuvre de l'environnement de développement de systèmes embarqués sous Linux
3. Debugging d'applications sous Linux embarqué
4. Mise en production d'un système embarqué sous Linux

2.2 Informations pratiques

Nous avons choisi l'option de travailler directement avec la machine virtuelle fournie.

1. Lien : http://www.hardkernel.com/main/products/prdt_info.php?g_code=G140448267127
 2. Lien : <http://www.denx.de/wiki/U-Boot>

2.3 Gravure de la carte SD

Avant de pouvoir graver la carte, il faut trouver le nom du périphérique auquel elle est attachée. Il peut être obtenu avec la commande suivante :

```
1 $ mount
2 ...
3 /dev/sd2 on /media/lmi/usrfs type ext4 (rw,nosuid,nodev,uhelper=udisks2)
4 /dev/sd1 on /media/lmi/5a13f590-5792-413e-ba62-403debd56a5 type ext4 (rw,
   nosuid,nodev,uhelper=udisks2)
5 ...
```

La commande va lister tous les périphériques connectés, dans notre cas, la carte SD se nomme lmi et est liée à /dev/sdb2 et /dev/sdb1.

Un script a ensuite été écrit, regroupant les différentes commandes à exécuter pour la gravure de la carte.

Emplacement du script : */EnvLinuxEmbarque/flasheMMC.sh*

Le plus simple est de placer le script directement dans le workspace CSEL. Il s'exécute à l'aide de la commande suivante :

```
1 ./flasheMMC.sh
```

Il va aller détacher les volumes attachés à la carte SD, créer la table de partition et graver les différents firmwares et images.

Une fois la carte gravée et insérée dans la cible, le ventilateur se met à tourner si tout s'est bien passé. Si rien ne se passe, il faut également contrôler que le switch de l'Odroid est dans la position pour booter sur la carte SD.

2.4 Mise en place de l'infrastructure

La cible ODROID-XU3 doit avoir l'adresse IP 192.168.0.11 et la machine hôte l'IP 192.168.0.4.

2.4.1 Configuration de la machine virtuelle

Pour associer la carte réseau de l'ordinateur à la machine virtuelle, il faut suivre les étapes ci-dessous :

1. Éteindre la vm, aller dans *edit->Virtual Network Editor...->change settings*
2. Dans la fenêtre des réseaux, changer la configuration en bridged to : Il faut choisir la carte réseau de l'ordinateur. Ici, Contrôleur PCIe (propre à l'ordinateur).

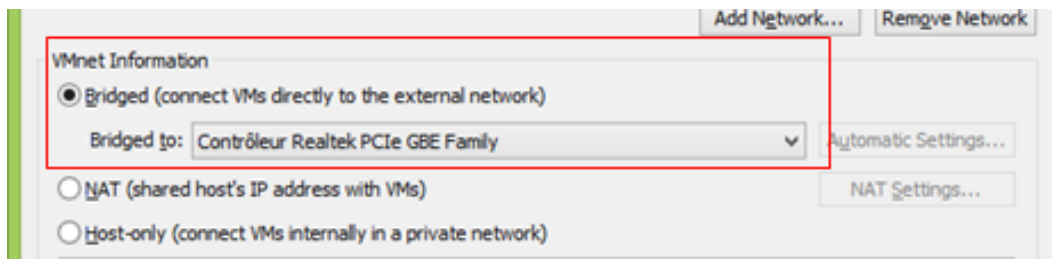


FIGURE 1 – Configuration de la carte réseau

Puis dans les settings de la VM, il faut aller changer le réseau pour utiliser celui que l'on vient de configurer.

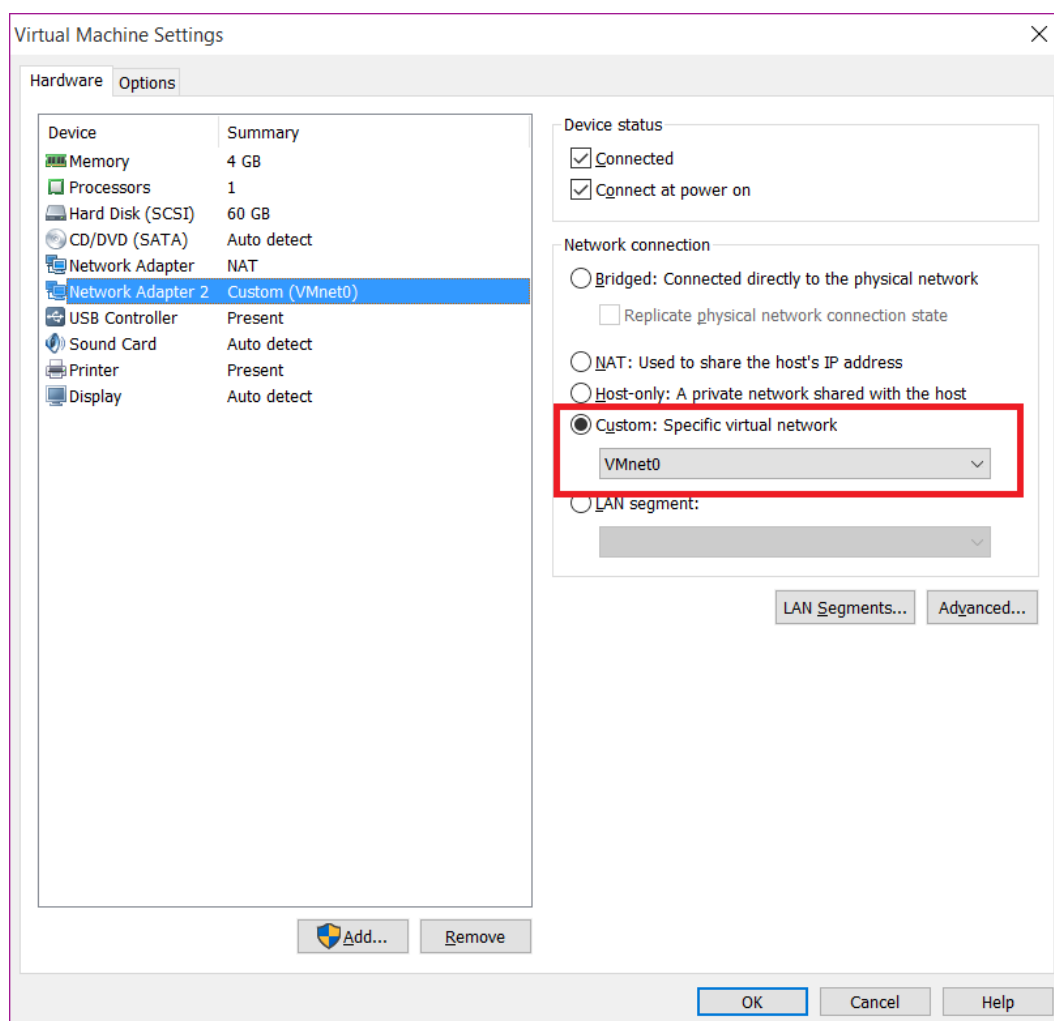


FIGURE 2 – Association de la carte réseau à la machine virtuelle

La dernière étape est d'activer le réseau dans la machine virtuelle (icône réseau -> enabling networking).

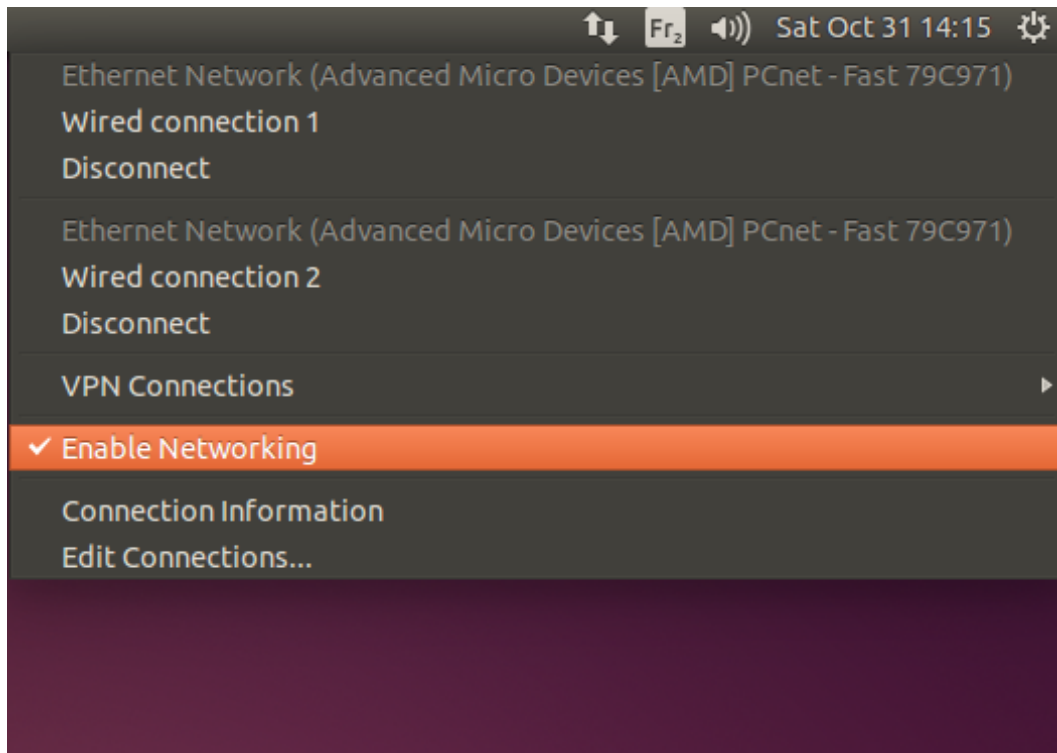


FIGURE 3 – Activation du réseau

2.4.2 Accès ssh sans mot de passe

Il faut aller modifier le fichier `/etc/ssh/sshd_config` sur la cible pour autoriser l'accès sans mot de passe.

Pour accéder au fichier, il faut entrer sur la cible par la connexion série :

```

1 $ sudo minicom
2
3 Welcome to minicom 2.7
4
5 OPTIONS: I18n
6 Compiled on Jan  1 2014, 17:13:22.
7 Port /dev/ttyUSB0, 13:10:43
8
9 Press CTRL-A Z for help on special keys
10
11
12 Welcome to Hardkernel ODROID_XU3 board
13 odroidxu3 login: root
14 #

```

On peut ensuite rechercher le fichier et l'éditer avec vi ou vim pour changer l'attribut suivant à "yes" : "PermitEmptyPassword yes". Il faut ensuite redémarrer la cible :

```

1 # reboot

```

Normalement, on devrait avoir accès à la cible en ssh en entrant la commande suivante dans la machine hôte :

```
1 lmi@cse11:~$ ssh root@192.168.0.11
2 #
```

Pour valider la connexion Ethernet/IP, on peut également arrêter la cible dans son U-boot en tapant la touche "carriage return" et faire un ping de la machine hôte :

```
1 lmi@cse11:~$ sudo minicom
2 ...
3 # reboot
4 ...
5 Press 'Enter' or 'Space' to stop autoboot: 0
6
7 ODROIDXU3> usb start
8 (Re)start USB...
9 ...
10
11 ODROIDXU3> ping 192.168.0.4
12 Waiting for Ethernet connection... done.
13 Using sms0 device
14 host 192.168.0.4 is alive
15
16 ODROIDXU3> run nfsboot
```

Si la machine hôte répond au ping, tout a bien été configuré. La dernière commande "run nfsboot" permet de démarrer la cible en mode développement.

2.4.3 Création de l'espace de travail

Le but est de partager le répertoire de projet de la machine hôte avec la cible. Pour cela, il faut accéder à la cible via le port série ou par ssh et taper les commandes indiquées dans la donnée. Celles ci-dessous montrent comment attacher automatiquement l'espace de travail.

```
1 # mkdir /usr/workspace
2 # vi /etc/fstab
3 # /etc/fstab: static file system information.
4 #
5 # <file system> <mount pt>      <type>    <options>          <dump> <pass>
6 /dev/root      /                ext2       rw,noauto          0       1
7 proc           /proc            proc       defaults            0       0
8 devpts         /dev/pts         devpts     defaults,gid=5,mode=620 0       0
9 tmpfs          /dev/shm         tmpfs      mode=0777           0       0
10 tmpfs          /tmp             tmpfs      mode=1777           0       0
11 sysfs          /sys             sysfs      defaults            0       0
12 192.168.0.4:/home/lmi/workspace /usr/workspace  nfs         hard,intr,nolock
13 # mount -a
14 # reboot
```


Pour contrôler que le répertoire est bien partagé avec la cible, on peut y entrer la commande mount et normalement on voit le répertoire partagé. Il sera maintenant attaché automatiquement à chaque démarrage de la cible en mode de développement.

```

1 # mount
2 ...
3 192.168.0.4:/home/lmi/workspace on /usr/workspace type nfs (rw,relatime,vers=3,
   rsize=131072,wsz=131072,namlen=255,hard,nolock,proto=tcp,timeo=600,
   retrans=2,sec=sys,mountaddr=192.168.0.4,mountvers=3,mountproto=tcp,
   local_lock=all,addr=192.168.0.4)

```

2.5 Debugging de l'application

Cette section présente les différentes étapes à réaliser pour configurer l'environnement Eclipse pour qu'il utilise une connexion ssh entre l'hôte et la cible et pouvoir debugger à distance une application. Pour cela, il faut commencer par charger un projet dans Eclipse :

1. File -> Import... (si le projet existe déjà)
2. C/C++ -> Existing Code as Makefile Project (configure pour utiliser le Makefile du projet et non celui généré par Eclipse)

Une fois le projet importé dans l'espace de travail, il faut configurer le debugger :

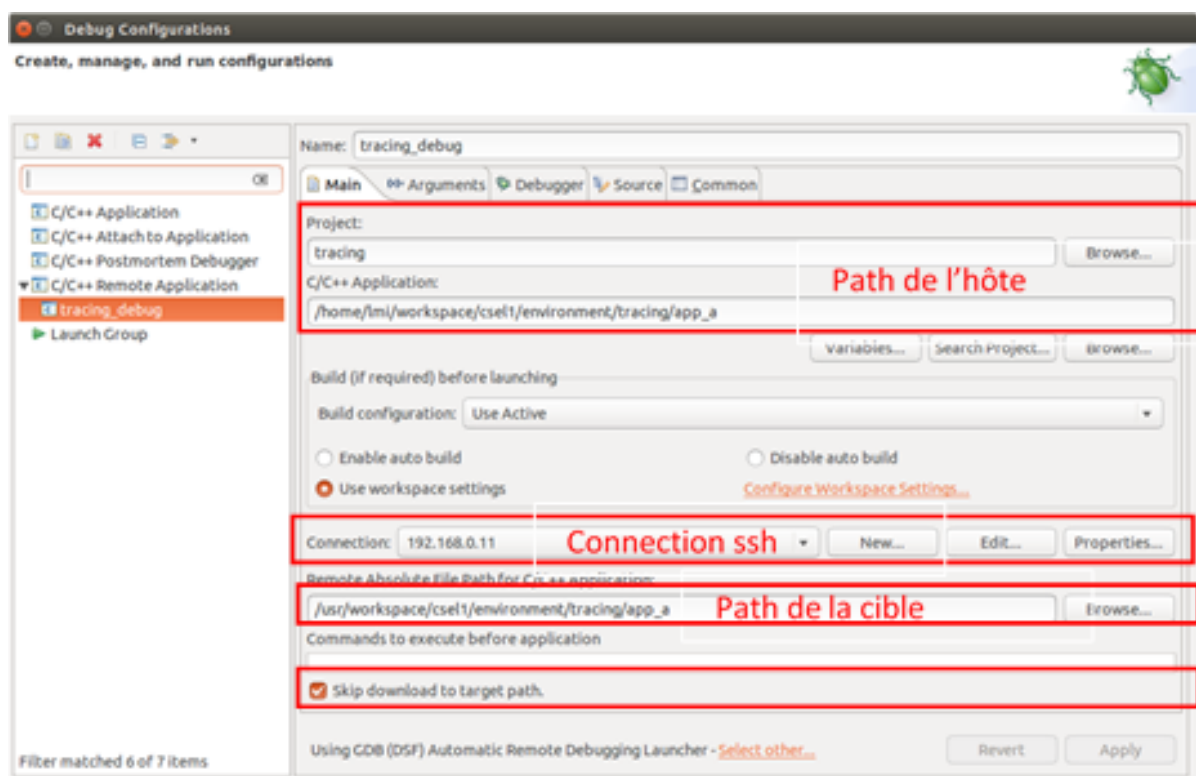


FIGURE 4 – Configuration du debugger

Pour pouvoir entrer le path de la cible, il faut impérativement que la connexion ssh soit configurée comme ci-dessous :

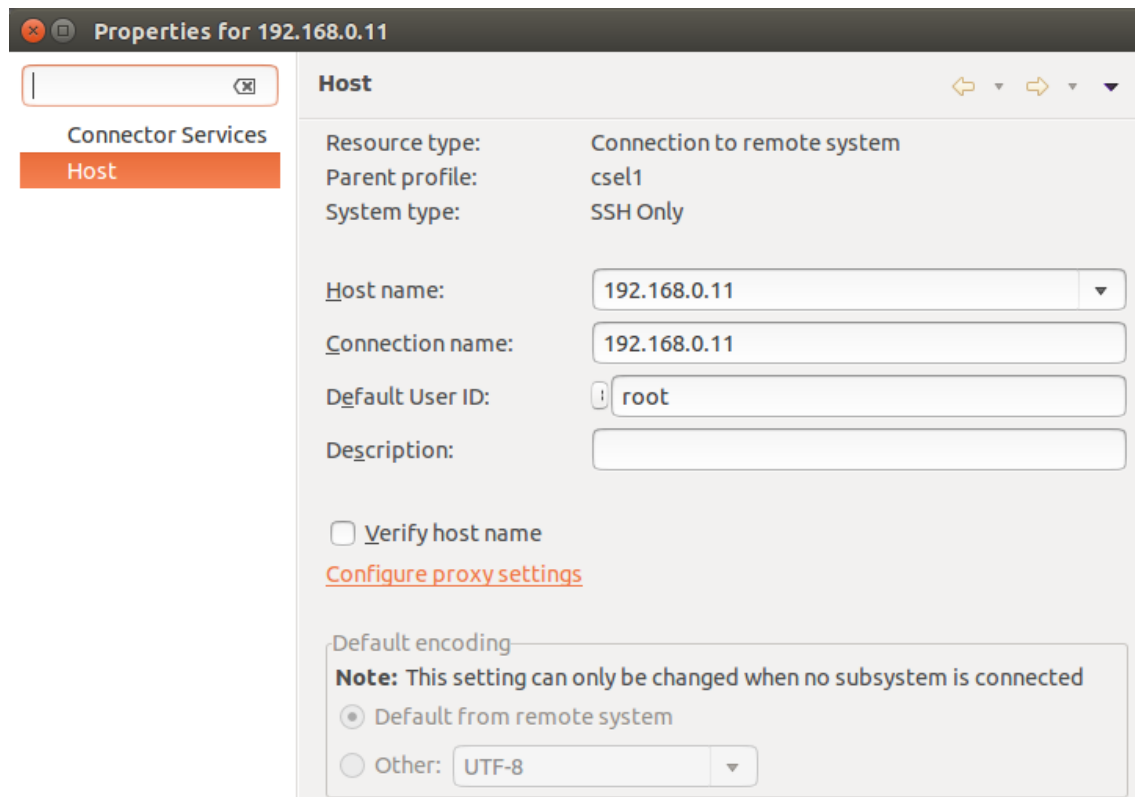


FIGURE 5 – Configuration de l'accès ssh

Il faut encore aller configurer le debugger gdb :

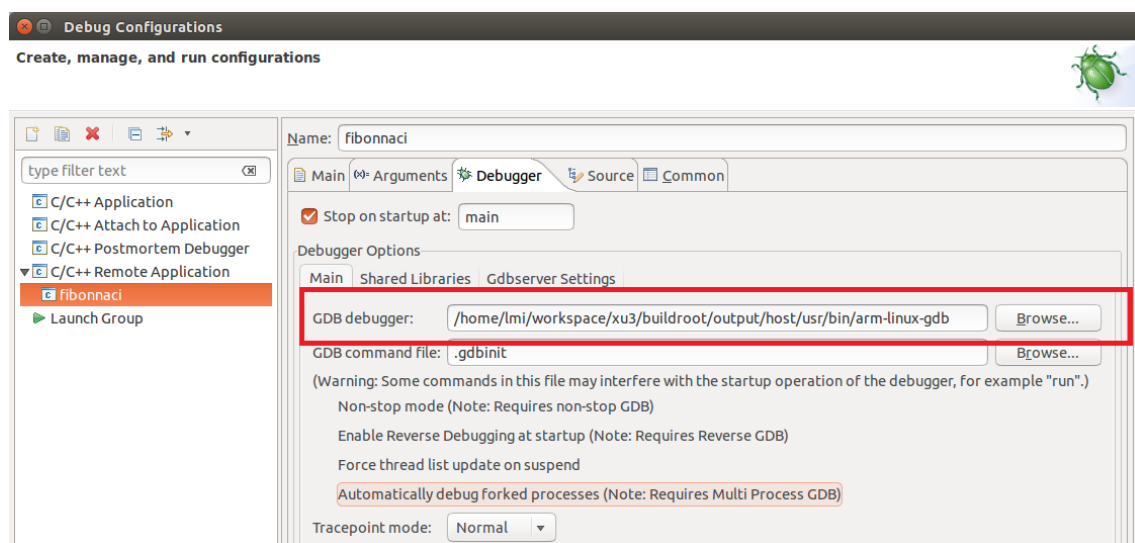


FIGURE 6 – Configuration du server gdb

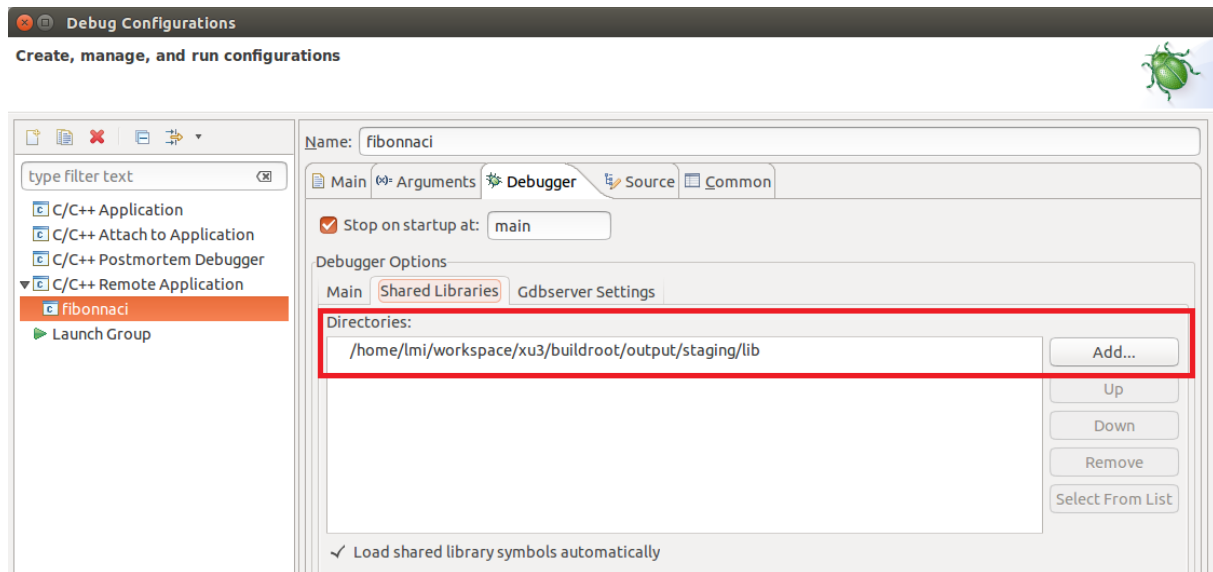


FIGURE 7 – Configuration des shared library

Avec cette configuration, on peut ensuite debugger pas à pas les exercices d'exemples. Voici un exemple avec fibonacci :

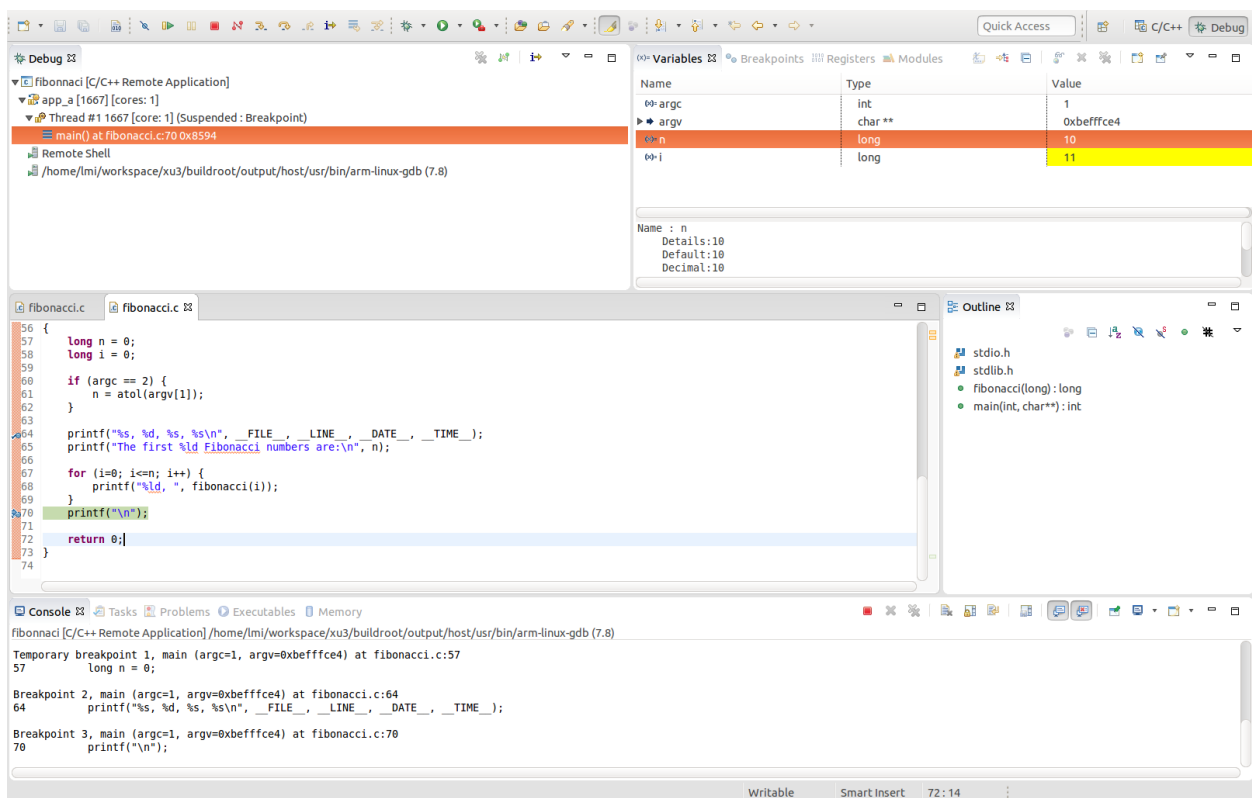


FIGURE 8 – Debug de l'exemple Fibonacci

2.6 Travail à réaliser

1. Installer l'environnement de développement sur la machine hôte, selon les instructions ci-dessus, et configurer la cible en mode de développement avec NFS sous Eclipse. Installer/configurer SSH pour accès à distance.
2. Créer un script permettant de générer la carte SD.
3. Configurer le noyau Linux afin de mounter un usrfs sous ext4 depuis la carte SD. Ce dernier sera monté dans le répertoire `"/usr/local/"`.
4. Tester les différentes méthodes et techniques de débogage proposées par l'environnement Linux. Pour cela générer les exemples fournis dans le répertoire `"/workspace/c-sell/environnement/"` en suivant les indications des slides.
5. Sur la base de l'exercice 4, mettre l'ODROID-XU3 en mode production. Un petit programme situé dans le `"usrfs"` sera démarré au lancement de la cible.

À ce point du rapport, les points 1 et 2 ont déjà été réalisés.

2.7 Configuration du noyau pour monter un usrfs

Le `usrfs` ou `user file system` est une des partitions que nous avons créée sur la carte SD. Le `usrfs` peut contenir des programmes spécifiques à une application de la plateforme. Il est possible de monter cette partition dans un répertoire de notre choix, ici `"/usr/local"`. Pour monter le `usrfs`, il suffit de taper les commandes suivantes sur la cible :

```
1 # mkdir /usr/local
2 # mount -t ext4 /dev/mmcbk0p2 /usr/local
```

`"mmcbk0p2"` indique le bloc mémoire 0 partition 2. Si l'on regarde la configuration de la carte SD faite plus haut, c'est à cet endroit que l'on a placé le `usrfs`.

Si l'on veut que ce montage se fasse automatiquement au démarrage de la cible, il faut d'ajouter au fichier `/etc/fstab` la ligne suivante :

```
1 /dev/mmcbk0p2 /usr/local ext4 defaults 0 0
```

Il faut ensuite activer la configuration, redémarrer et vérifier que le montage s'est fait correctement :

```
1 # mount -a
2 # reboot
3 ...
4 # mount
5 ...
6 /dev/mmcbk0p2 on /usr/local type ext4 (rw,relatime,data=ordered)
```

2.8 Test des différents exemples proposés

Pour la suite du rapport, les symboles suivant sont définis :

1. `$` : commande sur la machine hôte
2. `#` : commande sur la cible
3. `>` : commande sur la cible arrêtée avant démarrage

2.8.1 Fibonacci

```
1 lmi@cse11:~/workspace/cse11/environment/samples/fibonacci$ make clean all
2
3 lmi@cse11:~$ ssh root@192.168.0.11
4 # cd ../usr/workspace/cse11/environment/samples/fibonacci/
5 # ./app_a 10
6 fibonacci.c, 64, Oct 31 2015, 15:27:54
7 The first 10 Fibonacci numbers are:
8 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,
```

2.8.2 Tracing

Avec la trace active

```
1 lmi@cse11:~/workspace/cse11/environment/samples/tracing$ make DEBUG=1 clean all
2
3 # cd ../tracing/
4 # ./app_a 12
5 fibonacci.c, 70, Oct 31 2015, 15:35:42
6 The first 12 Fibonacci numbers are:
7 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
```

Avec la trace inactive

```
1 lmi@cse11:~/workspace/cse11/environment/samples/tracing$ make clean all
2
3 # ./app_a 12
4 The first 12 Fibonacci numbers are:
5 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
```

2.8.3 Core dumps

```
1 lmi@cse11:~/workspace/cse11/environment/samples/core_dumps$ make clean all
2
3 # cd ../core_dumps/
4 # ulimit -c unlimited
5 # ./app_a
6 Segmentation fault (core dumped)
7 # gdb app_a core
```

```

8 ...
9 Core was generated by './app_a'.
10 Program terminated with signal SIGSEGV, Segmentation fault.
11 #0 0x000083e0 in access_data () at core_dumps.c:31
12 31 *p=10;
13 (gdb) bt
14 #0 0x000083e0 in access_data () at core_dumps.c:31
15 #1 0x00008424 in call (n=0) at core_dumps.c:37
16 #2 0x00008420 in call (n=1) at core_dumps.c:36
17 #3 0x00008420 in call (n=2) at core_dumps.c:36
18 #4 0x00008420 in call (n=3) at core_dumps.c:36
19 #5 0x00008420 in call (n=4) at core_dumps.c:36
20 #6 0x00008420 in call (n=5) at core_dumps.c:36
21 #7 0x00008420 in call (n=6) at core_dumps.c:36
22 #8 0x00008420 in call (n=7) at core_dumps.c:36
23 #9 0x00008420 in call (n=8) at core_dumps.c:36
24 #10 0x00008420 in call (n=9) at core_dumps.c:36
25 #11 0x00008420 in call (n=10) at core_dumps.c:36
26 #12 0x00008448 in main () at core_dumps.c:43

```

2.8.4 Backtrace

Cet exemple s'effectue uniquement sur la machine hôte

```

1 lmi@cse11:~/workspace/cse11/environment/samples/core_dumps$ cd ../backtrace/
2 lmi@cse11:~/workspace/cse11/environment/samples/backtrace$ make clean all
3 ...
4 lmi@cse11:~/workspace/cse11/environment/samples/backtrace$ ./app_h
5 backtrace() returned 17 addresses
6 ./app_h[0x80484fc]
7 [0xb7727404]
8 ./app_h[0x804854a]
9 ./app_h[0x8048571]
10 ./app_h[0x804856c]
11 ./app_h[0x804856c]
12 ./app_h[0x804856c]
13 ./app_h[0x804856c]
14 ./app_h[0x804856c]
15 ./app_h[0x804856c]
16 ./app_h[0x804856c]
17 ./app_h[0x804856c]
18 ./app_h[0x804856c]
19 ./app_h[0x804856c]
20 ./app_h[0x804859c]
21 /lib/i386-linux-gnu/libc.so.6(__libc_start_main+0xf3)[0xb757ca83]
22 ./app_h[0x8048401]
23 Segmentation fault (core dumped)
24 lmi@cse11:~/workspace/cse11/environment/samples/backtrace$ addr2line -e app_h 0
    x804859c
25 /home/lmi/workspace/cse11/environment/samples/backtrace/main.c:61

```

2.8.5 System calls

```

1 lmi@cse11:~/workspace/cse11/environment/samples/system_calls$ make clean all
2
3 # cd ../system_calls/
4 # ./app_a
5 current temperature: 46.00 degree Celcius
6
7 # strace ./app_a
8 execve("./app_a", [ "./app_a" ], [ /* 21 vars */ ]) = 0
9 brk(0)                                = 0x11000
10 ...
11 +++ exited with 0 +++
12
13 # strace -e trace=mmap2 ./app_a
14 ...
15 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
   x6ff9000
16 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0
   x6ffa000
17 current temperature: 47.00 degree Celcius
18 +++ exited with 0 +++

```

2.8.6 Memory leaks

```

1 lmi@cse11:~/workspace/cse11/environment/samples/memory_leaks$ make clean all
2
3 # cd ../memory_leaks/
4 # ./app_a
5 # valgrind --leak-check=full ./app_a
6 ==1714== Memcheck, a memory error detector
7 ==1714== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
8 ==1714== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
9 ==1714== Command: ./app_a
10 ==1714==
11 ==1714==
12 ==1714== HEAP SUMMARY:
13 ==1714==    in use at exit: 31,880 bytes in 3,985 blocks
14 ==1714==    total heap usage: 4,000 allocs, 15 frees, 32,000 bytes allocated
15 ==1714==
16 ==1714== 31,880 (8 direct, 31,872 indirect) bytes in 1 blocks are definitely
   lost in loss record 2 of 2
17 ==1714==    at 0x483535C: malloc (in /usr/lib/valgrind/vgpreload_memcheck-arm-
   linux.so)
18 ==1714==
19 ==1714== LEAK SUMMARY:
20 ==1714==    definitely lost: 8 bytes in 1 blocks
21 ==1714==    indirectly lost: 31,872 bytes in 3,984 blocks
22 ==1714==    possibly lost: 0 bytes in 0 blocks
23 ==1714==    still reachable: 0 bytes in 0 blocks
24 ==1714==    suppressed: 0 bytes in 0 blocks
25 ==1714==
26 ==1714== For counts of detected and suppressed errors, rerun with: -v

```

```
27 ==1714== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

2.9 Mise en production de l'ODROID-XU3

Le fichier S60appl a été légèrement modifié pour que le path vers app_a corresponde :

```
1 #!/bin/sh
2 #
3 # Daemon application
4 #
5 case "$1" in
6     start)
7         /usr/workspace/csel1/environment/samples/daemon/app_a
8         ;;
9     stop)
10        killall app_a
11        ;;
12    restart|reload)
13        killall app_a
14        /usr/workspace/csel1/environment/samples/daemon/app_a
15        ;;
16    *)
17        echo "$Usage: $0 {start|stop|restart}"
18        exit 1
19 esac
20
21 echo "Daemon application launched"
22
23 exit $?
```

Il suffit ensuite de copier ce fichier dans */etc/init.d* et de faire un reboot. Au démarrage de la cible, le script */etc/init.d/rcs* va effectuer tous les fichiers S?? présent dans le répertoire */etc/init.d*, donc notre application également.

```
1 # cp /usr/workspace/csel1/environment/samples/daemon/S60appl /etc/init.d
2 #reboot
3 ...
4 Starting logging: OK
5 Starting mdev...
6 Initializing random number generator... done.
7 Starting network...
8 ip: RTNETLINK answers: File exists
9 Starting sshd: OK
10 Daemon application launched
11 [ 9.933833] [c4] pwm-samsung: tin parent at 66600000
```


2.10 Réponse aux questions

1. Comment faut-il procéder pour générer l'U-Boot ?

2. Comment peut-on ajouter et générer un package supplémentaire dans le Buildroot ?

3. Comment doit-on procéder pour modifier la configuration du noyau Linux ?

4. Comment faut-il faire pour générer son propre RootFS ?

5. Comment faut-il procéder pour utiliser la carte eMMC en lieu et place de la carte SD ?

3 Programmation noyau : Module noyau

3.1 Module noyau

3.1.1 Exercice 1

Donnée : Générer un module noyau "out of tree" pour la cible ODROID-XU3

Point a : Créer le squelette d'un module noyau et générer le en dehors des sources du noyau à l'aide d'un Makefile. Le module devra afficher un message lors de son enregistrement et lors de sa désinstallation.

Emplacement du code : */ModulesNoyau/exercice1-Module/pointA*

Le code a été écrit conformément aux exemples présentés dans le support de cours.

Exécution du code :

Le code doit être compilé sur la machine hôte à l'aide de la commande suivante :

```
1 $ make clean all
```

Point b : Tester sur la machine hôte la commande "modinfo" sur votre squelette de module et comparer les informations retournées avec celles du code source.

```
1 $ modinfo mymodule.ko
2 filename:          /home/lmi/workspace/csel1/environment/module_noyau/exercice1/
   pointa/mymodule.ko
3 license:           GPL
4 description:       Module Skeleton
5 author:            Emilie Gsponer
6 depends:
7 vermagic:          3.10.63 SMP preempt mod_unload ARMv7 p2v8
```

Ces informations correspondent à celles entrée dans le skeleton du module :

```
1 MODULE_AUTHOR("Emilie Gsponer");
2 MODULE_DESCRIPTION("Module Skeleton");
3 MODULE_LICENSE("GPL");
```

Point c : Installer le module (insmod), contrôler le log du noyau (dmesg).

Exécution du code : Le module doit être installé sur la cible à l'aide la commande *insmod*. On peut observer que le module a bien été installé à l'aide de la commande *dmesg*.

```
1 # insmod mymodule.ko
2 # dmesg | tail -n 4
3 [ 7.704536] [c5] Freeing unused kernel memory: 436K (c089c000 - c0909000)
4 [ 7.877462] [c4] EXT4-fs (mmcblk0p1): re-mounted. Opts: errors=remount-ro,
   data=ordered
5 [ 8.967876] [c7] pwm-samsung: tin parent at 66600000
6 [ 53.629226] [c4] Linux module skeleton loaded
```

Point d : Comparer les résultats obtenus par la commande "lsmod" avec ceux obtenus avec la commande "cat /proc/modules".

Exécution des commandes : Les deux commandes présentent le même résultat, seule la mise en forme est différente.

```

1 # cat /proc/modules
2 mymodule 687 0 — Live 0xbf004000 (O)
3
4 # lsmod mymodule.ko
5 Module                Size  Used by    Tainted: G
6 mymodule              687    0

```

Point e : Désinstaller le module (rmmod).

Exécution de la commande : On peut voir à l'aide des commandes *cat /proc/modules*, *lsmod* et *dmesg* que le module a bien été désinstallé.

```

1 # rmmod mymodule
2 # cat /proc/modules
3 # lsmod mymodule.ko
4 Module                Size  Used by    Tainted: G
5 # dmesg | tail -n 4
6 [  7.877462] [c4] EXT4-fs (mmcblk0p1): re-mounted. Opts: errors=remount-ro,
   data=ordered
7 [  8.967876] [c7] pwm-samsung: tin parent at 66600000
8 [  53.629226] [c4] Linux module skeleton loaded
9 [  73.538490] [c5] Linux module skeleton unloaded

```

Point f : Adapter le Makefile du module pour autoriser l'installation du module avec les autres modules du noyau permettant l'utilisation de la commande "modprobe". Le module devra être installé dans le root filesystem utilisé en nfs par la cible.

Emplacement du code : */ModulesNoyau/exercice1-Module/pointF*

Pour que le module puisse être ajouté dans le noyau, il suffit d'ajouter les lignes suivantes au Makefile :

```

1 MODPATH := /tftpboot/odroidxu3
2
3 install:
4 $(MAKE) -C $(KDIR) M=$(PWD) INSTALL_MOD_PATH=$(MODPATH) modules_install

```

Exécution du code : Le code doit être compilé et installé depuis la machine hôte à l'aide des commandes suivantes :

```

1 $ make clean all
2 $ sudo make install

```

Il faut ensuite démarrer la cible en mode nfs pour y installer le module à l'aide de la commande *modprobe*

```
1 lmi@cse11:~/.ssh$ sudo minicom
2 ...
3 ODROIDXU3> run nfsboot
4 (Re)start USB...
5 ...
6 Welcome to Hardkernel ODROID_XU3 board
7 odroidxu3 login: root
8 # modprobe mymodule
9 [ 98.024580] [c2] Linux module skeleton loaded
10 # modprobe -r mymodule
11 [ 99.673514] [c0] Linux module skeleton unloaded
```

3.1.2 Exercice 2

Donnée : Adapter le module de l'exercice précédent afin qu'il puisse recevoir deux ou trois paramètres de votre choix. Ces paramètres seront affichés dans la console. Adapter également le rootfs afin de pouvoir utiliser la commande "modprobe".

Emplacement du code : */ModulesNoyau/exercice2-Parameter*

Exécution du code : Ce module prend deux paramètres d'entrée par défaut, text = "hello" et elements = 2. La valeur de ces éléments peut être modifiée lors de l'installation du module dans le noyau, comme le montre la démonstration suivante.

```
1 # modprobe mymodule
2 [ 922.323941] [c1] Linux module skeleton loaded
3 [ 922.326835] [c1] text:hello
4 [ 922.326835] elements:2
5 # modprobe -r mymodule
6 [ 925.223513] [c0] Linux module skeleton unloaded
7 # modprobe mymodule text="hello world" elements=10
8 [ 1196.597223] [c2] Linux module skeleton loaded
9 [ 1196.600182] [c2] text:hello world
10 [ 1196.600182] elements:10
11 # modprobe -r mymodule
12 [ 1199.518513] [c0] Linux module skeleton unloaded
```

Remarque : Pour tous les exercices suivants, on reprendra le même Makefile permettant d'installer le module directement dans le noyau à l'aide de la commande *modprobe*. Il faudra pour cela que la cible soit démarrée en mode nfs. Le module de chaque exercice devra être compilé et installé à l'aide des commandes suivantes :

```
1 $ make clean all
2 $ sudo make install
```

3.1.3 Exercice 3

Donnée : Trouver la signification des 4 valeurs affichées lorsque l'on tape la commande "cat /proc/sys/kernel/printk".

Exécution de la commande :

```
1 # cat /proc/sys/kernel/printk
2 10      4      1      7
```

Ces chiffres représentent le log level du kernel utilisé pour déterminer l'importance d'un message.

1. 10 : current loglevel
2. 4 : default loglevel
3. 1 : minimum loglevel
4. 7 : boot-time-default loglevel

Cela permet de définir la priorité des messages. Si un message a une priorité plus grande (chiffre plus petit) que le loglevel courant du kernel, il sera affiché dans la console. Dans le cas contraire, le message n'est pas affiché. L'exemple suivant le démontre en passant le loglevel à 1, le module n'affiche plus de message dans la console.

```
1 # echo 10 > /proc/sys/kernel/printk
2 # cat /proc/sys/kernel/printk
3 10      4      1      7
4 # modprobe mymodule
5 [ 2139.129459] [c2] Linux module skeleton loaded
6 [ 2139.132348] [c2] text:hello
7 [ 2139.132348] elements:2
8 # modprobe -r mymodule
9 [ 2143.683511] [c0] Linux module skeleton unloaded
10
11 # echo 1 > /proc/sys/kernel/printk
12 # cat /proc/sys/kernel/printk
13 1      4      1      7
14 # modprobe mymodule
15 # modprobe -r mymodule
16
17 # echo 10 > /proc/sys/kernel/printk
18 # modprobe mymodule
19 [ 2173.021095] [c2] Linux module skeleton loaded
20 [ 2173.024050] [c2] text:hello
21 [ 2173.024050] elements:2
```

Source : http://elinux.org/Debugging_by_printing

3.2 Gestion de la mémoire, bibliothèques et fonctions utiles

3.2.1 Exercice 4

Donnée : Créer dynamiquement des éléments dans le noyau. Adapter un module noyau afin que l'on puisse lors de son installation spécifier un nombre d'éléments à créer ainsi qu'un texte initial à stocker dans les éléments précédemment alloués. Chaque élément contiendra également un numéro unique, Les éléments seront créés lors de l'installation du module et chaînés dans une liste. Ces éléments seront détruits lors de la désinstallation du module. Des messages d'information seront émis afin de permettre le debugging du module.

Emplacement du code : */ModulesNoyau/exercice4-List*

Exécution du code :

```

1 # modprobe mymodule element_num=5 text="Hello world"
2 [ 2250.418060] [c2] New element added to the list
3 [ 2250.421107] [c2] New element added to the list
4 [ 2250.425536] [c2] New element added to the list
5 [ 2250.429937] [c2] New element added to the list
6 [ 2250.434368] [c2] New element added to the list
7 [ 2250.438773] [c2] ID of element : 0
8 [ 2250.438773] String is : Hello world
9 [ 2250.445642] [c2] ID of element : 1
10 [ 2250.445642] String is : Hello world
11 [ 2250.452489] [c2] ID of element : 2
12 [ 2250.452489] String is : Hello world
13 [ 2250.459415] [c7] ID of element : 3
14 [ 2250.459415] String is : Hello world
15 [ 2250.466444] [c7] ID of element : 4
16 [ 2250.466444] String is : Hello world
17 # modprobe -r mymodule
18 [ 2267.933575] [c1] Element popped
19 [ 2267.935161] [c1] Element popped
20 [ 2267.938196] [c1] Element popped
21 [ 2267.941283] [c1] Element popped
22 [ 2267.944313] [c1] Element popped
23 [ 2267.947308] [c1] Module removed

```

3.2.2 Exercice 5

Donnée : Indiquer les différents allocateurs SLAB disponibles dans le noyau Linux pour la cible ORDOID-XU3.

Réponse :

1. SLAB : "as cache frendly as possible, benchmark frendly"
2. SLOB : "as compact as possible"
3. SLUB : "Simple and instruction cost counts. Superior Debugging. Defragmentation. Execution time friendly"

Source : https://www.google.ch/url?sa=t&rct=j&q=&esrc=s&source=web&cd=3&cad=rja&uact=8&ved=0CDEQFjACahUKEwiqj6GfhKbIAhWLXBoKHXDUAw&url=http%3A%2F%2Fwww.cs.berkeley.edu%2F~kubitron%2Fcourses%2Fcs194-24-S14%2Fhand-outs%2Fbonwick_slab.pdf&usg=AFQjCNENx6NuNkg&sig2=ZdJ_jUWHIf01qFIiIkEyHA

3.3 Accès aux entrées/sorties

3.3.1 Exercice 6

Donnée : À l'aide d'un module noyau, réserver la zone mémoire correspondante au registre du uP décrivant son identification. Adress de départ 0x1000'0000, taille de la zone 0x100. Valider cette réservation à l'aide de la commande "cat /proc/iomem".

Adapter ce module afin d'afficher cet identifiant dans la console de débogage "dmesg". Ce dernier est composé des champs suivants :

1. Bit 31..12 : product id
2. Bit 11..8 : package id
3. Bit 7..4 : major revision
4. Bit 3..0 : minor revision

Emplacement du code : */ModulesNoyau/exercice6-IO*

Complément : L'identifiant demandé par la donnée est en fait l'identifiant de la zone mémoire réservée. Il peut être obtenu grâce à la méthode *ioread*. Mais pour pouvoir lire cette zone, il faut la mapper dans la mémoire virtuelle du noyau avec la méthode *ioremap*, car le noyau n'a pas directement accès aux entrées/sorties.

La zone mémoire réservée par le code a été nommée "uP register".

Exécution du code :

```

1 # modprobe mymodule
2 [ 30.336553] [c5] Linux module skeleton loaded
3 [ 30.339495] [c5] Memory allocated
4 [ 30.342733] [c5] uP register: Bit 31..12 : product id=0x65422
5 [ 30.348494] [c5] uP register: Bit 11..8 : package id=0x0
6 [ 30.353877] [c5] uP register: Bit 7..4 : major revision=0x0
7 [ 30.359597] [c5] uP register: Bit 3..0 : minor revision=0x1
8 # cat /proc/iomem
9 03000000-03048fff : /lpas@03810000
10 03810000-038100ff : /lpas@03810000
11 03830000-038300ff : samsung-i2s
12 03860000-03860fff : /pinctrl@03860000
13 03880000-03880fff : /amba/adma@03880000
14 03880000-03880fff : /amba/adma@03880000
15 10000000-100000ff : uP register
16 ...
17 # dmesg | tail -n 10
18 [ 9.041140] [c4] VFS: Mounted root (nfs filesystem) on device 0:13.
19 [ 9.048765] [c4] devtmpfs: mounted
20 [ 9.050869] [c4] Freeing unused kernel memory: 436K (c089c000 - c0909000)

```

```

21 [ 12.571761] [c4] pwm-samsung: tin parent at 66600000
22 [ 30.336553] [c5] Linux module skeleton loaded
23 [ 30.339495] [c5] Memory allocated
24 [ 30.342733] [c5] uP register: Bit 31..12 : product id=0x65422
25 [ 30.348494] [c5] uP register: Bit 11..8 : package id=0x0
26 [ 30.353877] [c5] uP register: Bit 7..4 : major revision=0x0
27 [ 30.359597] [c5] uP register: Bit 3..0 : minor revision=0x1
28 # modprobe -r mymodule
29 [ 88.338556] [c0] Linux module skeleton unloaded
30 [ 88.341621] [c0] Memory released

```

3.4 Threads du noyau

3.4.1 Exercice 7

Donnée : Développer un petit module permettant d’instancier un thread dans le noyau. Ce thread affichera un message toutes les 5 secondes. Il pourra être mis en sommeil durant ces 5 secondes à l’aide de la fonction « `ssleep(5)` » provenant de l’interface `<linux/delay.h>`.

Emplacement du code : */ModulesNoyau/exercice7-Thread*

Exécution du code :

```

1 # modprobe mymodule
2 [ 1434.469821] [c2] Thread created
3 # [ 1439.473573] [c1] Thread awake
4 [ 1444.478588] [c2] Thread awake
5 [ 1449.483589] [c3] Thread awake
6 # modprobe -r mymodule
7 [ 1454.488597] [c0] Thread awake
8 [ 1454.490139] [c1] Thread stopped

```

3.5 Mise en sommeil

3.5.1 Exercice 8

Donnée : Développer un petit module permettant d’instancier deux threads dans le noyau. Le premier thread attendra une notification de réveil du deuxième thread et se remettra en sommeil. Le 2ème thread enverra cette notification toutes les 5 secondes et se rendormira. On utilisera les waitqueues pour les mises en sommeil. Afin de permettre le debugging du module, chaque thread affichera un petit message à chaque réveil.

Emplacement du code : */ModulesNoyau/exercice8-MiseEnSommeil*

Exécution du code :

```

1 # modprobe mymodule
2 [ 1523.052680] [c2] Init wait queue

```



```

3 [ 1523.054851] [c2] Threads created
4 # [ 1528.058588] [c3] Thread2 (notif each 5s) awake
5 [ 1528.061596] [c1] Thread1 (wait notif) awake
6 [ 1533.063590] [c0] Thread2 (notif each 5s) awake
7 [ 1533.066587] [c1] Thread1 (wait notif) awake
8 [ 1538.068588] [c0] Thread2 (notif each 5s) awake
9 [ 1538.071587] [c1] Thread1 (wait notif) awake
10 # modprobe -r mymodule
11 [ 1543.073563] [c0] Thread2 (notif each 5s) awake
12 [ 1543.076562] [c2] Thread1 (wait notif) awake
13 [ 1548.078569] [c0] Thread2 (notif each 5s) awake
14 [ 1548.081573] [c1] Threads stopped

```

3.6 Gestion des interruptions

3.6.1 Exercice 9

Donnée : Développement d'un petit module permettant de capturer les pressions exercées sur les swiches de la carte d'extension par interruption. Afin de permettre le debugging du module, chaque capture affichera un petit message.

Informations fournies :

- Configurer la direction des GPIO en entrée :

```
1 gpio_request (EXYNOS5_GPX<gpio_nr>(<pin_nr>));
```

```
1 gpio_direction_input (EXYNOS5_GPX<gpio_nr>(<pin_nr>));
```

- Obtenir le vecteur d'interruption avec le service suivant :

```
1 gpio_to_irq (<io_nr>);
```

- Informations sur les switches de la carte d'extension
 - sw1 - gpio_nr=2, pin_nr=5, io_nr=29
 - sw2 - gpio_nr=2, pin_nr=6, io_nr=30
 - sw3 - gpio_nr=1, pin_nr=6, io_nr=22
 - sw4 - gpio_nr=1, pin_nr=2, io_nr=18

Complément : Les switches de 1 à 4 sont interceptés.

Emplacement du code : */ModulesNoyau/exercice9-Interrupt*

Exécution du code : Et voici la preuve que tout fonctionne correctement, avec un message s'affichant pour chaque bouton pressé :

```
# modprobe mymodule.
Interrupt handler module loaded in kernel
[ 538.666271] Configuring pins[ 538.669335] _gpio_request: gpio-176 (SW1) status -6
[ 538.674249] [c6] _gpio_request: gpio-177 (SW2) status -16
[ 538.679529] [c6] _gpio_request: gpio-168 (SW3) status -16
[ 538.684901] [c6] _gpio_request: gpio-164 (SW4) status -16
[ 538.690281] [c6] Configuring switches interrupts
[ 538.694747] Pins and interrupts have been configured.# [ 546.004819] Some switch
[ 546.712050] [c0] Some switch has been pressed
[ 547.684521] [c0] Some switch has been pressed
[ 548.492088] [c0] Some switch has been pressed
[ 549.336592] [c0] Some switch has been pressed

# modprobe -r mymodule.
[ 562.272793] [c0] Freeing interrupts
[ 562.274685] Interrupts freed# █
```

FIGURE 9 – Affichage du chargement du module, des pressions sur les boutons et de la suppression du module

4 Programmation noyau : Pilotes de périphériques

4.1 Pilotes orientés mémoire

4.1.1 Exercice 1

Donnée : Réaliser un pilote orienté mémoire permettant de mapper en espace utilisateur les registres de la FPGA en utilisant le fichier virtuel `/dev/mem`. Ce pilote permettra de lire l'identification du uP (chip id) décrit dans l'exercice no 6 du cours sur la programmation de modules noyau.

Complément : Ce pilote n'est pas un module noyau, il faut prendre le Makefile d'un des codes d'exemples, par exemple Fibonacci. La cible doit tout de même être démarrée en mode nfs.

Emplacement du code : `/PilotesPeripheriques/exercice1-mmap`

Exécution du code :

```
1 # ./app_a
2 uP register: Bit 31..12 : product id=0x65422
3 uP register: Bit 11..8  : package id=0x0
4 uP register: Bit 7..4   : major revision=0x0
5 uP register: Bit 3..0   : minor revision=0x1
```

Remarques : On obtient bien le même id qu'avec le module noyau de l'exercice 6 de la série précédente.

Pour utiliser correctement la commande `mmap`, il ne faut pas mettre l'offset à 0, mais à `0x1000000` (adresse du chipid de l'exercice6), sinon le code ne fonctionne pas, il essaie de lire une zone mémoire interdite ou inexistante.

4.1.2 Exercice 2

Donnée : Sur la base de l'exercice 1, développer un pilote orienté caractère permettant de mapper en espace utilisateur ces registres (implémentation de l'opération de fichier « `mmap` »). Le driver orienté mémoire sera ensuite adapté à cette nouvelle interface. Remarque : à effectuer après les exercices des pilotes orientés caractère

Emplacement du code :

`/PilotesPeripheriques/exercice2-mmapModule/user`
`/PilotesPeripheriques/exercice2-mmapModule/noyau`

Remarque : Cet exercice a été compliqué à réaliser. Le code est un mélange de l'exercice 1 et 5. Il faut garder en tête que pour mapper les registres en espace utilisateur, il faut utiliser les fonction standard des `file_operations` (`open`,`close`,`mmap`) et en plus, ajouter les `vm_operations_struct` (`open`,`close`) pour agir sur la zone mémoire. Dans le pilote orienté ca-

ractère, on utilise la fonction *remap_pfn_range* pour mapper la zone mémoire.

Exécution du code :

```

1 # modprobe mymodule
2 [ 854.599130] [c2] mod: successfully loaded with major 249
3 # mknod /dev/mod c 249 0
4 # ./app_a /dev/mod
5 [ 1166.619318] [c0] mod: open
6 [ 1166.620800] [c0] mod: mmap
7 [ 1166.623246] [c0] VMA open, virt b6f71000, phys 10000000
8 [ 1166.628986] [c0] VMA close.
9 [ 1166.631259] [c0] mod: release
10 file /dev/mod open
11 Physical memory: Bit 31..12 : product id=0x65422
12 Physical memory: Bit 11..8 : package id=0x0
13 Physical memory: Bit 7..4 : major revision=0x0
14 Physical memory: Bit 3..0 : minor revision=0x1
15
16 Virtual memory: Bit 31..12 : product id=0x43108
17 Virtual memory: Bit 11..8 : package id=0x3
18 Virtual memory: Bit 7..4 : major revision=0x2
19 Virtual memory: Bit 3..0 : minor revision=0xa
20 file /dev/mod close
21 # modprobe -r mymodule
22 [ 1259.338388] [c0] mod: successfully unloaded

```

4.2 Pilotes orientés caractères

4.2.1 Exercice 3

Donnée : Implémenter un pilote de périphérique orienté caractère. Ce pilote sera capable de stocker dans une variable globale au module les données reçues par l'opération *write* et de les restituer par l'opération *read*. Pour tester le module, on utilisera les commandes « *echo* » et « *cat* ».

Emplacement du code : */PilotesPeripheriques/exercice3-ReadWrite*

Exécution du code :

```

1 $ make clean all
2 $ sudo make install

```

```

1 # modprobe mymodule
2 [ 4107.528630] [c2] mod: successfully loaded with major 249
3 # mknod /dev/mod c 249 0
4 # echo -n test module > /dev/mod

```

```

5 [ 4134.907095] [c1] mod: open()
6 [ 4134.908540] [c1] mod: write test module
7 [ 4134.912539] [c1] mod: release()
8 # cat /dev/mod
9 [ 4145.581863] [c0] mod: open()
10 [ 4145.583312] [c0] mod: read test module
11 [ 4145.587140] [c0] mod: read test module
12 [ 4145.590820] [c0] mod: release()
13 # modprobe -r mymodule
14 [ 4157.528779] [c1] mod: successfully unloaded

```

4.2.2 Exercice 4

Donnée : Etendre la fonctionnalité du pilote de l'exercice #3 afin que l'on puisse à l'aide d'un paramètre module spécifier le nombre d'instance. Pour chaque instance on créera une variable unique permettant de stocker les données échangées avec l'application en espace utilisateur.

Emplacement du code : */PilotesPeripheriques/exercice4-multi_instance*

Exécution du code :

```

1 # mknod /dev/mydev_0 c 248 0
2 # mknod /dev/mydev_1 c 248 1
3 #
4 # echo -n "bonjour, monsieur" > /dev/mydev_0
5 [12232.424660] [c1] skeleton : open operation, major version : 248, minor
   version : 0
6 [12232.430827] [c1] skeleton : opened for reading and writing...
7 [12232.436572] [c1] minor : 0
8 [12232.436572] count = 17
9 [12232.441568] [c1] skeleton : efault value=-14
10 [12232.445802] [c1] skeleton : write operation... written=17
11 [12232.451198] [c1] skeleton : release operation
12 #
13 # echo -n "bien le bonjour, mademoiselle" > /dev/mydev_1
14 [12254.024382] [c1] skeleton : open operation, major version : 248, minor
   version : 1
15 [12254.030622] [c1] skeleton : opened for reading and writing...
16 [12254.036288] [c1] minor : 1
17 [12254.036288] count = 29
18 [12254.041274] [c1] skeleton : efault value=-14
19 [12254.045522] [c1] skeleton : write operation... written=29
20 [12254.050909] [c1] skeleton : release operation
21 #
22 # cat /dev/mydev_0
23 [12278.473668] [c0] skeleton : open operation, major version : 248, minor
   version : 0
24 [12278.479815] [c0] skeleton : opened for reading and writing...
25 [12278.485576] [c0] skeleton : read operation... read=17
26 [12278.490663] [c6] skeleton : read operation... read=0

```

```

27 [12278.495527] [c6] skeleton : release operation
28 bonjour , monsieur
29 #
30 # cat /dev/mydev_1
31 [12291.694063] [c0] skeleton : open operation , major version : 248, minor
    version : 1
32 [12291.700284] [c7] skeleton : opened for reading and writing...
33 [12291.706190] [c7] skeleton : read operation... read=29
34 [12291.710985] [c7] skeleton : read operation... read=0
35 [12291.715919] [c7] skeleton : release operation
36 bien le bonjour , mademoiselle

```

4.2.3 Exercice 5

Donnée : Développer une petite application en espace utilisateur permettant d'accéder à ces pilotes orientés caractère. L'application devra écrire un texte dans le pilote et le relire.

Complément : Le module noyau a été repris de l'exercice 3.

Emplacement du code :

/PilotesPeripheriques/exercice5-UserAccess/user
/PilotesPeripheriques/exercice5-UserAccess/noyau

Exécution du code :

```

1 lmi@cse11:~/workspace/cse11/environment/peripheral/exercice5/user$ make clean
  all
2 lmi@cse11:~/workspace/cse11/environment/peripheral/exercice5/user$ cd ../noyau/
3 lmi@cse11:~/workspace/cse11/environment/peripheral/exercice5/noyau$ make clean
  all
4 lmi@cse11:~/workspace/cse11/environment/peripheral/exercice5/noyau$ sudo make
  install

```

```

1 # modprobe mymodule
2 [ 231.111019] [c2] mod: successfully loaded with major 249
3 # mknod /dev/mod c 249 0
4 # ./app_a /dev/mod HELLO
5 [ 234.969067] [c0] mod: open()
6 [ 234.970719] [c0] mod: write HELLO
7 [ 234.973927] [c0] mod: read HELLO
8 [ 234.977039] [c0] mod: release()
9 file /dev/mod open
10 write HELLO
11 read HELLO
12 file /dev/mod close
13 # modprobe -r mymodule
14 [ 261.512478] [c0] mod: successfully unloaded

```

4.3 Opérations bloquantes

4.3.1 Exercice 6

Donnée : Développer un pilote et une application utilisant les entrées/sorties bloquantes pour signaler une interruption matérielle provenant de l'un des switches de la carte d'extension de ODROID-XU3. L'application utilisera le service select pour compter le nombre d'interruptions.

Remarque : les switches non pas d'anti-rebond, par conséquent il est fort probable que vous comptiez un peu trop d'impulsions ; effet à ignorer.

Remarque : L'application utilisateur attend de recevoir 5 interruptions des boutons avant s'arrêter.

Emplacement du code :

/PilotesPeripheriques/exercice6-SelectButton/user

/PilotesPeripheriques/exercice6-SelectButton/noyau

Exécution du code :

```

1 # modprobe mymodule
2 [ 90.125005] [c2] mod: successfully loaded with major 249
3 [ 90.129074] [c2] Interrupt handler module loaded in kernel
4 [ 90.134167] Configuring pins[ 90.137059] _gpio_request: gpio-176 (SW1)
   sta6
5 [ 90.142048] [c2] _gpio_request: gpio-177 (SW2) status -16
6 [ 90.147391] [c2] _gpio_request: gpio-168 (SW3) status -16
7 [ 90.152778] [c2] _gpio_request: gpio-164 (SW4) status -16
8 [ 90.158239] [c4] Configuring switches interrupts
9 [ 90.162625] Pins and interrupts have been configured.[ 90.167598] Init
   waie
10 # ./app_a /dev/mod
11 [ 97.905744] [c0] mod: open
12 [ 97.907156] [c0] Threads started
13 file /dev/mod open
14 [ 102.317965] [c0] Some switch has been pressed
15 [ 102.320893] [c2] Thread awake
16 Interrupt 1
17 [ 103.728868] [c0] Some switch has been pressed
18 [ 103.731800] [c0] Thread awake
19 Interrupt 2
20 [ 104.686291] [c0] Some switch has been pressed
21 Interrupt 3
22 [ 106.084775] [c0] Some switch has been pressed
23 [ 106.087702] [c1] Thread awake
24 Interrupt 4
25 [ 107.048387] [c0] Some switch has been pressed
26 [ 107.051374] [c1] mod: release
27 Interrupt 5
28 [ 107.707804] [c0] Some switch has been pressed
29 [ 107.710718] [c0] Thread awake
30 [ 107.713669] [c1] Threads stopped

```

```
31 | file /dev/mod close
32 | # modprobe -r mymodule
33 | [ 119.950452] [c0] Freeing interrupts
34 | [ 119.952344] Interrupts freed [ 119.955165] mod: successfully unloaded
```

4.4 sysfs

4.4.1 Exercice 7

Donnée : Développer un pilote de périphérique orienté caractère permettant de valider la fonctionnalité du sysfs. Le pilote offrira des attributs de périphérique afin pouvoir lire et écrire un bloc de données composé de quelques membres et de pouvoir modifier le contenu de la valeur entière. Seules les commandes « echo » et « cat » doivent être nécessaire pour manipuler ces attributs.

Emplacement du code : */PilotesPeripheriques/exercice7-sysfs*

Exécution du code :

```
1 # echo -n "salut" > /sys/devices/platform/skeleton/attr
2 #
3 #
4 # cat /sys/devices/platform/skeleton/attr
5 salut#
```

4.5 ioctl (optionnel)

4.5.1 Exercice 8

Cet exercice n'a pas été réalisé.

4.6 procfs (optionnel)

4.6.1 Exercice 9

Cet exercice n'a pas été réalisé.

4.7 Gestionnaires de périphériques

4.7.1 Exercice 10

Donnée : Implémenter à l'intérieur d'un pilote de périphérique orienté caractère, les mécanismes nécessaires à la création du fichier d'accès au pilote (remplacement de la commande « mknod ») par l'utilitaire « mdev » de la BusyBox.

Emplacement du code : */PilotesPeripheriques/exercice10-periph_handler*

Exécution du code :

5 Programmation système : Système de fichiers

5.1 Contexte

La carte ODROID-XU3 Lite est équipé d'un petit ventilateur afin d'évacuer la chaleur produite par l'activité du microprocesseur. La vitesse du ventilateur est contrôlée par un PWM (Pulse-Width Modulation), dont le rapport de cycle (duty) dépendra de la température du microprocesseur. Le ventilateur sera régulé sur la base de 20kHz. Sur la carte ODROID-XU3 Lite ce PWM peut être réalisé avec la porte d'entrée/sortie "gpb2_0".

5.2 Travail à réaliser

Sur le site moodle vous trouverez une petite application qui contrôle la vitesse du ventilateur. Ce code n'a pas été très bien programmé et utilise le 100% d'un cœur du processeur (à mesurer avec top). Concevez une application permettant de gérer la vitesse de rotation du ventilateur de l'ODROID-XU3 à l'aide des trois boutons poussoir. Quelques indications pour la réalisation de l'application :

1. Au démarrage le duty cycle du ventilateur sera réglé à 50%
2. Utilisation des boutons poussoir
 - (a) « sw1 » pour augmenter la vitesse du ventilateur
 - (b) « sw2 » pour remettre la vitesse du ventilateur à sa valeur initiale
 - (c) « sw3 » pour diminuer la vitesse du ventilateur
 - (d) une pression continue exercée sur un bouton indiquera une auto incrémentation ou décrémentation du duty cycle.
3. Tous les changements de vitesse du ventilateur seront logger avec syslog de Linux.
4. Le multiplexage des entrées/sorties devra être utilisé.

5.3 Travail réalisé

5.3.1 Description

Tous les points de la donnée ont été implémentés. La fréquence du ventilateur a été augmentée à 50 kHz, car à 20kHz le ventilateur émet un son strident très agaçant.

Pour le point de l'auto incrémentation, la duty cycle est augmenté/diminué de 2% chaque 40 us quand un bouton est maintenu appuyé. Si par hasard, on maintient appuyé deux boutons en même temps, le programme attend qu'un des deux soit relâché pour changer le duty cycle. En plus des points demandés, les LEDs de la carte d'extensions sont utilisées.

1. LED1 : S'allume et s'éteint en fonction de la pression sur le switch 1.
2. LED2 : Clignote à la fréquence de la PWM. La vitesse est trop rapide pour la voir clignoter, on peut par contre observer un changement de luminosité entre un duty cycle proche de 0% et un proche de 100%.
3. LED3 : S'allume et s'éteint en fonction de la pression sur le switch 3.

Emplacement du code : */FanControl*

5.3.2 Configuration des GPIO

Une des recherches à faire pour ce projet était de trouver à quelle GPIO correspondait les boutons, les LEDs et le ventilateur.

Pour cela, la schématique de la carte d'extension a été utilisée.

Source : http://dn.odroid.com/ODROID-XU/Expansion_Board/ExpansionBoard.pdf

Si on prend comme exemple la LED D1, on trouve dans le schéma qu'elle correspond à la pin XE.INT23. Pour obtenir la GPIO correspondante, cette page a été très utile.

Source : <http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu>

On apprend grâce à ce site que la pin XE.INT23 correspond au GPX2.7.

Pour trouver le numéro de GPIO correspondant, il suffit d'entrer les commandes suivantes sur la cible :

```

1 # mount -t debugfs none /sys/kernel/debug
2 # cat /sys/kernel/debug/gpio
3 ...
4 GPIOs 16-23, platform/13400000.pinctrl, gpx1:
5 gpio-17 (dwc3_id_gpio) in hi
6 gpio-19 (sysfs) out lo
7 gpio-22 (sysfs) in hi
8 gpio-23 (scl) out hi
9
10 GPIOs 24-31, platform/13400000.pinctrl, gpx2:
11 gpio-27 (red:activity) out lo
12 gpio-28 (sysfs) out hi
13 gpio-29 (sysfs) in hi
14 gpio-30 (sysfs) in hi
15 gpio-31 (sysfs) out lo
16 ...

```

Ces commandes nous renseignent sur le numéro de GPIO. Si l'on reprend l'exemple de la LED D1, on va lire que le GPX2 correspond au numéro 24. Comme la LED est sur le GPX2.7, il faut encore rajouter l'offset. On obtient donc $24+7 = 31$ comme numéro de GPIO pour la LED D1.

Pour configurer les entrées/sorties, le code pour atteindre la pwm fourni dans l'exemple a été repris. Pour les boutons ils suffit d'ajouter le flanc de détection pour l'interruption.

5.3.3 Exécution du code

Le code s'exécute dans l'espace utilisateur et non plus dans le noyau comme les exercices précédents. Le code doit être compilé sur l'hôte et lancé sur la cible (en mode nfs). L'application n'affiche rien directement dans la console, tout est dans le syslog.

```

1 lmi@cse11:~/workspace/cse11/environment/fanCtrl$ make clean all

```

```

1 # cd /usr/workspace/csel1/environment/fanCtrl/
2 # ./app_a

```

5.3.4 Syslog

Une fois l'application lancée, on peut aller voir le *syslog*. Le duty cycle du ventilateur y est affiché chaque fois qu'il change. Comme l'application est lancée depuis la connexion série, il faut accéder à la cible par connexion ssh. Chaque pression sur un des boutons fera afficher une nouvelle ligne.

```

1 lmi@cse11:~$ ssh root@192.168.0.11
2 # tail -f /var/log/messages
3 Jan  1 00:06:37 odroidxu3 local1.notice fan control[1625]: duty :64
4 Jan  1 00:06:37 odroidxu3 local1.notice fan control[1625]: duty :66
5 Jan  1 00:06:37 odroidxu3 local1.notice fan control[1625]: duty :68
6 Jan  1 00:06:37 odroidxu3 local1.notice fan control[1625]: duty :70
7 Jan  1 00:06:37 odroidxu3 local1.notice fan control[1625]: duty :72
8 ...

```

5.3.5 Mesure de performance

La commande *top* permet de mesurer les performances de l'application. Le code fourni en exemple utilise 100% d'un cœur du processeur.

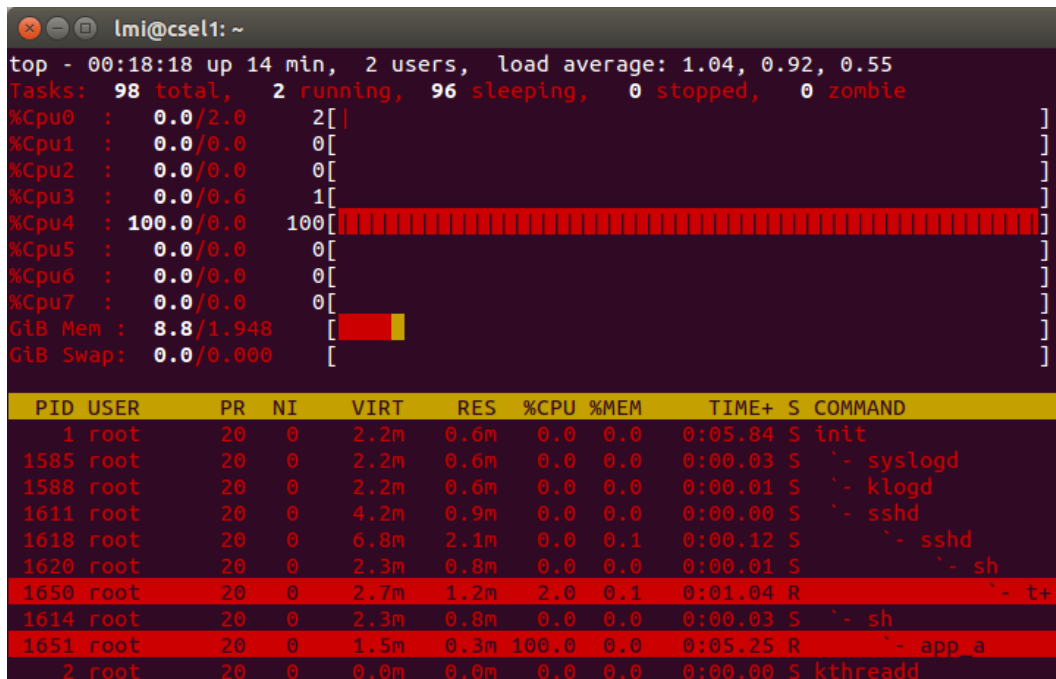


FIGURE 10 – Occupation du cœur avant modifications

L'image ci-dessous montre que notre application est meilleure, elle consomme moins de 3% d'un cœur.

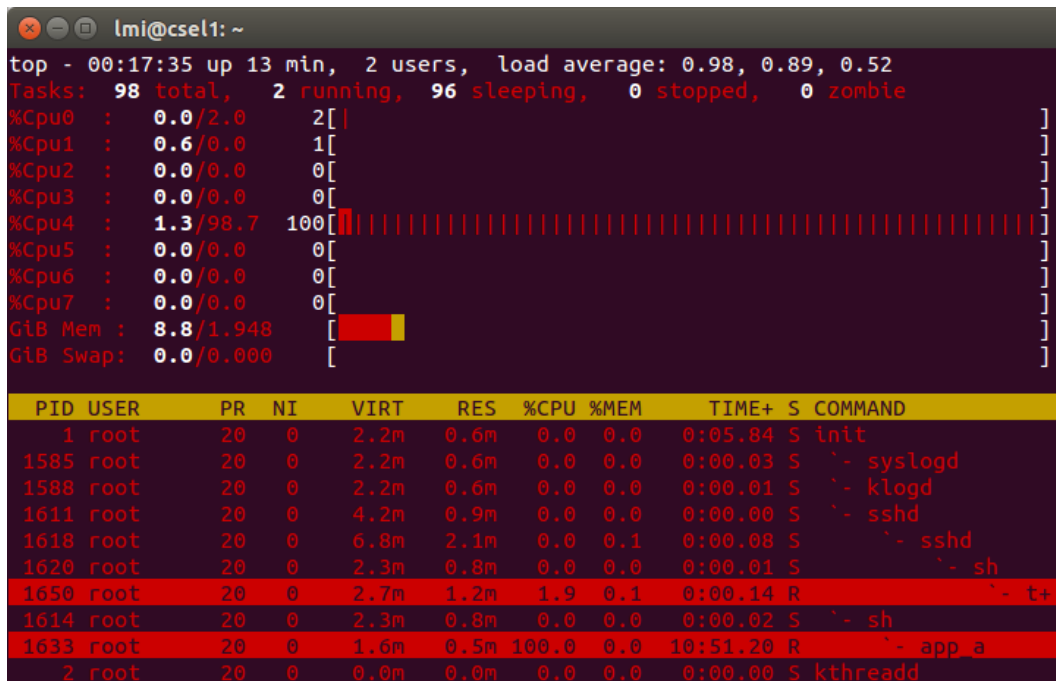


FIGURE 11 – Occupation du cœur après modifications

5.3.6 Amélioration possibles

Les switch n'ont pas d'anti-rebond. Parfois, l'état du bouton du programme ne correspond pas à l'état physique du bouton. La conséquence est que le duty cycle s'incrémente/décrompte alors qu'aucun bouton n'est appuyé. Il faudrait implémenter un anti-rebond software. Ce point n'a pas été réalisé, car ce n'était pas l'objectif du labo. Si le problème se présente, il suffit de relancer le programme.

6 Programmation système : Multiprocessing

Warning : Pour cette exercice, il a fallu reconfigurer le noyau linux en activant certaines options. Pour cela, se référer à la slide 72 du support de cours. Cette opération prend un temps considérable.

6.1 Processus, signaux et communication

6.1.1 Exercice 1

Donnée : Concevez et développez une petite application mettant en œuvre un des services de communication proposé par Linux (p.ex. « socketpair ») entre un processus parent et un processus enfant. Le processus enfant devra émettre quelques messages sous forme de texte vers le processus parent, lequel les affichera sur la console. Le message « exit » permettra de terminer l'application. Cette application devra impérativement capturer tous les signaux et les ignorer. Seul un message d'information sera affiché sur la console.

Emplacement du code : */Multiprocessing/Exercice1-Comm*

Exécution du code :

Le processus enfant demande à l'utilisateur de taper quelque chose, qui est ensuite retransmis au parent, qui l'affiche.

```
1 lmi@cse11:~$ cd workspace/cse11/environment/multiproc/exercice1
2 lmi@cse11:~/workspace/cse11/environment/multiproc/exercice1$ make clean all
```

```
1 # cd /usr/workspace/cse11/environment/multiproc/exercice1/
2 # ./app_a
3 In the main process.
4
5 In the parent process
6
7 In the children process.
8
9 Enter a msg to send to the parent :
10 Hello
11 Hello will be written to the parent.
12 Enter a msg to send to the parent :
13 Received 50 from the children. String is : Hello
14 Bla
15 Bla will be written to the parent.
16 Enter a msg to send to the parent :
17 Received 50 from the children. String is : Bla
18 Received signal is : Interrupt
19 Received signal is : Interrupt
20 Bla will be written to the parent.
21 Enter a msg to send to the parent :
22 Received 50 from the children. String is : Bla
23 Received signal is : Stopped
24 Received signal is : Stopped
```

```

25| Bla will be written to the parent.
26| Enter a msg to send to the parent :
27| Received 50 from the children. String is : Bla
28| exit
29| exit will be written to the parent.
30| Received 50 from the children. String is : exit
31| #

```

Ce second output montre que le SIGINT est géré pour afficher un message en écrivant *Received signal is* : suivi du type de signal. Dans le cas suivant, c'est un ctrl-c qui a été envoyé.

```

1| Enter a msg to send to the parent : Received 50 from the children. String is :
2| ^CReceived signal is : Interrupt
3| Received signal is : Interrupt
4| will be written to the parent.
5| Enter a msg to send to the parent : Received 50 from the children. String is :

```

6.2 CGroups

6.2.1 Exercice 2

Donnée : Concevez une petite application permettant de valider la capacité des groupes de contrôle de limiter l'utilisation de la mémoire.

Emplacement du code : */Multiprocessing/Exercice2-cgroups*

Réponse aux questions :

a. Quel effet a la commande «echo \$\$ > ...» sur les cgroups ?

Par cette commande, on demande aux cgroups de surveiller la tâche représentée par le terminal de l'ordroid. Comme le programme que nous lançons depuis la ligne de commande devient le fils du terminal dans lequel il est lancé, on surveille aussi le fils car il est automatiquement affecté à la hiérarchie du parent.

b. Quel est le comportement du sous-système «memory» lorsque le quota de mémoire est épuisé ? Pourrait-on le modifier ? Si oui, comment ?

Le comportement par défaut est de tuer le processus dépassant les limites. Mais il est possible de modifier ce comportement en mettant un **1** dans «memory.oom_control». Si la modification a été apportée, le processus qui viendrait à dépasser les limites de mémoire se verrait mis en pause.

L'output en dessous montre qu'une fois le programme *app_a* lancé, après vingt allocations, il est tué à cause de son excès de consommation mémoire.

```

1| # ./app_a

```

```

2 Ten more megas allocated
3
4 Ten more megas allocated
5
6 Killed

```

c. Est-il possible de surveiller/vérifier l'état actuel de la mémoire ? Si oui, comment ?

Plusieurs commandes peuvent être utilisées pour visualiser l'état de la mémoire :

- *free -m*
- *cat /proc/meminfo*

La première produit ce genre d'output :

```

1 # free -m
2 total          used          free        shared    buff/cache       available
3 Mem:           1990          156          1808           0           24          1813
4 Swap:           0            0            0

```

Tandis que la seconde produit ça :

```

1 # cat /proc/meminfo
2 MemTotal:       2038540 kB
3 MemFree:        1852220 kB
4 Buffers:         0 kB
5 Cached:         5272 kB
6 SwapCached:      0 kB
7 Active:          3308 kB
8 Inactive:        2952 kB
9 Active(anon):    1012 kB
10 Inactive(anon):   88 kB
11 Active(file):    2296 kB
12 Inactive(file):  2864 kB
13 Unevictable:     0 kB
14 Mlocked:         0 kB
15 HighTotal:      1296384 kB
16 HighFree:       1147292 kB
17 LowTotal:       742156 kB
18 LowFree:        704928 kB
19 SwapTotal:      0 kB
20 SwapFree:        0 kB
21 Dirty:          0 kB
22 Writeback:       0 kB
23 AnonPages:      1000 kB
24 Mapped:          2084 kB
25 Shmem:          112 kB
26 Slab:           20200 kB
27 SReclaimable:   9264 kB
28 SUnreclaim:    10936 kB
29 KernelStack:    840 kB
30 PageTables:     124 kB
31 NFS_Unstable:   0 kB
32 Bounce:         0 kB
33 WritebackTmp:   0 kB

```



```

34 CommitLimit:      1019268 kB
35 Committed_AS:      5752 kB
36 VmallocTotal:      245760 kB
37 VmallocUsed:        16520 kB
38 VmallocChunk:       103580 kB

```

Si on veut monitorer la consommation de notre application, on peut aussi utiliser Valgrind et l'outil *massif* avec la commande suivante :

```

1 valgrind --tool=massif ./app_a

```

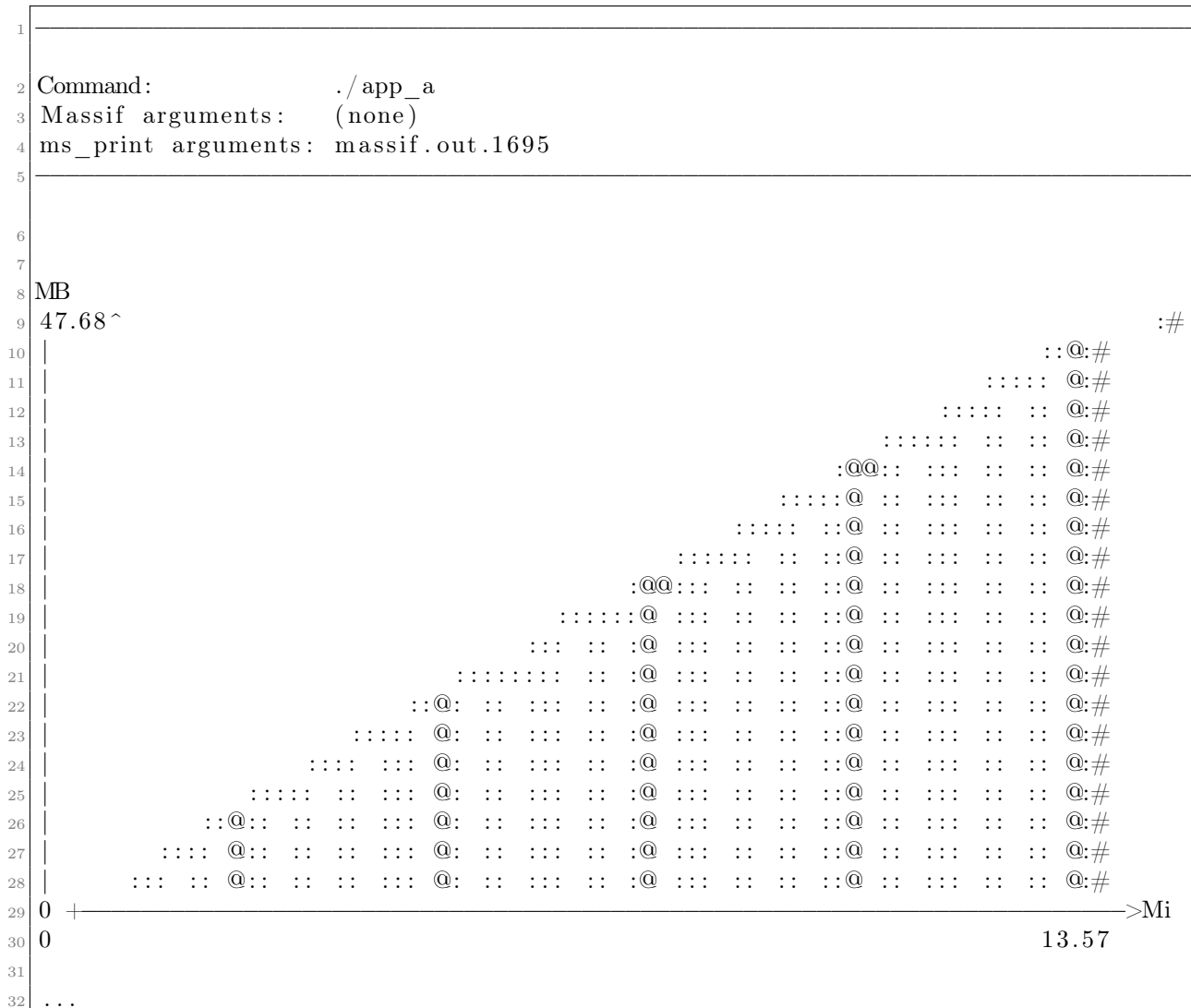
Le programme s'exécute normalement et l'outil de visualisation produit un fichier contenant tout le logging de la mémoire utilisée par l'application. On peut la voir avec la commande :

```

1 ms_print massif.out.<app_a pid>

```

Ce qui produit dans notre cas :



Pour nous rassurer, avec la commande et l'output suivant, on constate quand même que la mémoire a bien été libérée :

```
1 # valgrind ./app_a
2 ==1701== Memcheck, a memory error detector
3 ==1701== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
4 ==1701== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
5 ==1701== Command: ./app_a
6 ==1701==
7 Ten more megas allocated
8
9 Ten more megas allocated
10
11 Ten more megas allocated
12
13 Ten more megas allocated
14
15 Press enter to continue...
16
17
18 Ten more megas freed
19
20 Ten more megas freed
21
22 Ten more megas freed
23
24 Ten more megas freed
25
26 Ten more megas freed
27
28 ==1701==
29 ==1701== HEAP SUMMARY:
30 ==1701==      in use at exit: 0 bytes in 0 blocks
31 ==1701==    total heap usage: 50 allocs, 50 frees, 50,000,000 bytes allocated
32 ==1701==
33 ==1701== All heap blocks were freed — no leaks are possible
34 ==1701==
35 ==1701== For counts of detected and suppressed errors, rerun with: -v
36 ==1701== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Pour les derniers outputs, la limite d'allocation mémoire a été élevée à 100MB. Aucune solution n'a été trouvée autrement pour monitorer la mémoire du processus durant son exécution avec le contrôle de la mémoire. Valgrind est autant tué que l'application, sinon.

6.2.2 Exercice 3

Donnée : Afin valider la capacité des groupes de contrôle de limiter l'utilisation des CPU, concevez une petite application composée au minimum de 2 processus utilisant le 100% des ressources du processeur.

Emplacement du code : */Multiprocessing/Exercice3-CPU*

Monter les cgroup :

```

1 # mount -t tmpfs none /sys/fs/cgroup
2 # mkdir /sys/fs/cgroup/memory
3 # mount -t cgroup -o memory memory /sys/fs/cgroup/memory
4 # mkdir /sys/fs/cgroup/memory/mem
5 # mkdir /sys/fs/cgroup/cpuset
6 # mount -t cgroup -o cpu,cpuset cpuset /sys/fs/cgroup/cpuset
7 # mkdir /sys/fs/cgroup/cpuset/high
8 # mkdir /sys/fs/cgroup/cpuset/low
9 # echo 4 > /sys/fs/cgroup/cpuset/high/cpuset.cpus
10 # echo 0 > /sys/fs/cgroup/cpuset/high/cpuset.mems
11 # echo 3 > /sys/fs/cgroup/cpuset/low/cpuset.cpus
12 # echo 0 > /sys/fs/cgroup/cpuset/low/cpuset.mems

```

Réponse aux questions :

a. Les 4 dernières lignes sont obligatoires pour que les prochaines commandes fonctionnent correctement. Pouvez-vous en donner la raison ?

Les deux commandes écrivant dans le cpuset.cpus assignent un numéro de cœur pour le cpuset. Les autres écrivant dans le cpuset.mems limite l'espace mémoire et le crée. Cela crée l'architecture du CGroups. Pour l'avoir testé, si on n'affecte pas un cpuset.mems, l'application n'a pas de place en mémoire.

```

1 # echo $$ > /sys/fs/cgroup/cpuset/low/tasks
2 sh: write error: No space left on device

```

b. Ouvrez deux shells distincte et placez en une dans le cgroup high et l'autre dans le cgroup low.

Lancez ensuite votre application dans chacune des shells.

Quel devrait être le bon comportement ? Pouvez-vous le vérifier ?

Installation de l'application avec la connexion sériele :

Oui, on peut le vérifier, les commandes sont visibles un peu plus loin. Le comportement attendu est une utilisation à 100% des cœurs 3 et 4 (attribués lors de la création du CGroup) du processeur avec 50% pour le parent et 50% pour le parent. Pour l'instant, on n'a pas assigné de limitation dans l'utilisation du temps CPU.

```

1 lmi@cse11:~$ sudo minicom
2 ...
3 # echo $$ > /sys/fs/cgroup/cpuset/high/tasks
4 # ./app_a
5 In the main process.
6
7 In the parent process

```

```

8
9 In the children process.

```

Installation de l'application avec la connexion ssh :

```

1 lmi@cse11:~$ ssh root@192.168.0.11
2 # echo $$ > /sys/fs/cgroup/cpuset/low/tasks
3 # cd /usr/workspace/cse11/environment/multiproc/exercice3/
4 # echo $$ > /sys/fs/cgroup/cpuset/low/tasks
5 # ./app_a
6 In the main process.
7
8 In the parent process
9
10 In the children process.

```

L'utilisation du CPU peut être affichée avec la commande top en lançant l'application en arrière plan :

```

1 # ./app_a &
2 # In the main process.
3
4 In the parent process
5
6 In the children process.
7
8 # top

```

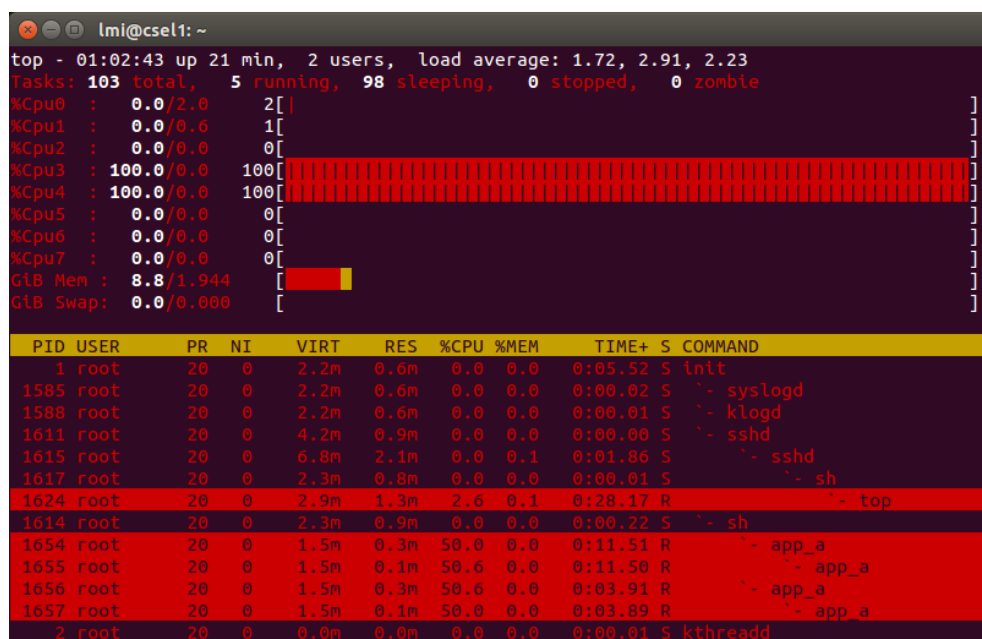


FIGURE 12 – Utilisation du processeur

c. Sachant que l'attribut `cpus.shares` permet de répartir le temps CPU entre différents cgroups, comment devrait-on procéder pour lancer deux tâches distinctes sur le cœur 6 de notre processeur et attribuer 75% du temps CPU à la première tâche et 25% à la deuxième ?

Voici les commandes utilisées pour la configuration demandée. Par défaut, l'attribut `cpu.shares` contient 1024, qui représente 50% du temps CPU. Pour plus de lisibilité, l'application a été compilée une fois sous le nom `app_a` et une fois sous le nom `app_b`.

Dans la connection ssh :

```
1 # top
```

Dans la connection série :

```
1 # echo 512 > /sys/fs/cgroup/cpuset/low/cpu.shares
2 # echo 1536 > /sys/fs/cgroup/cpuset/high/cpu.shares
3 # echo $$ > /sys/fs/cgroup/cpuset/low/tasks
4 # ./app_b &
5 # In the main process.
6
7 In the parent process
8
9 In the children process.
10
11 # echo $$ > /sys/fs/cgroup/cpuset/high/tasks
12 # ./app_a &
13 # In the main process.
14
15 In the parent process
16
17 In the children process.
18
19 #
```

Voici le résultat obtenu, l'`app_a` prend 75% du temps CPU (37.7% parent, 37.7% enfant) et l'`app_b` 25%.

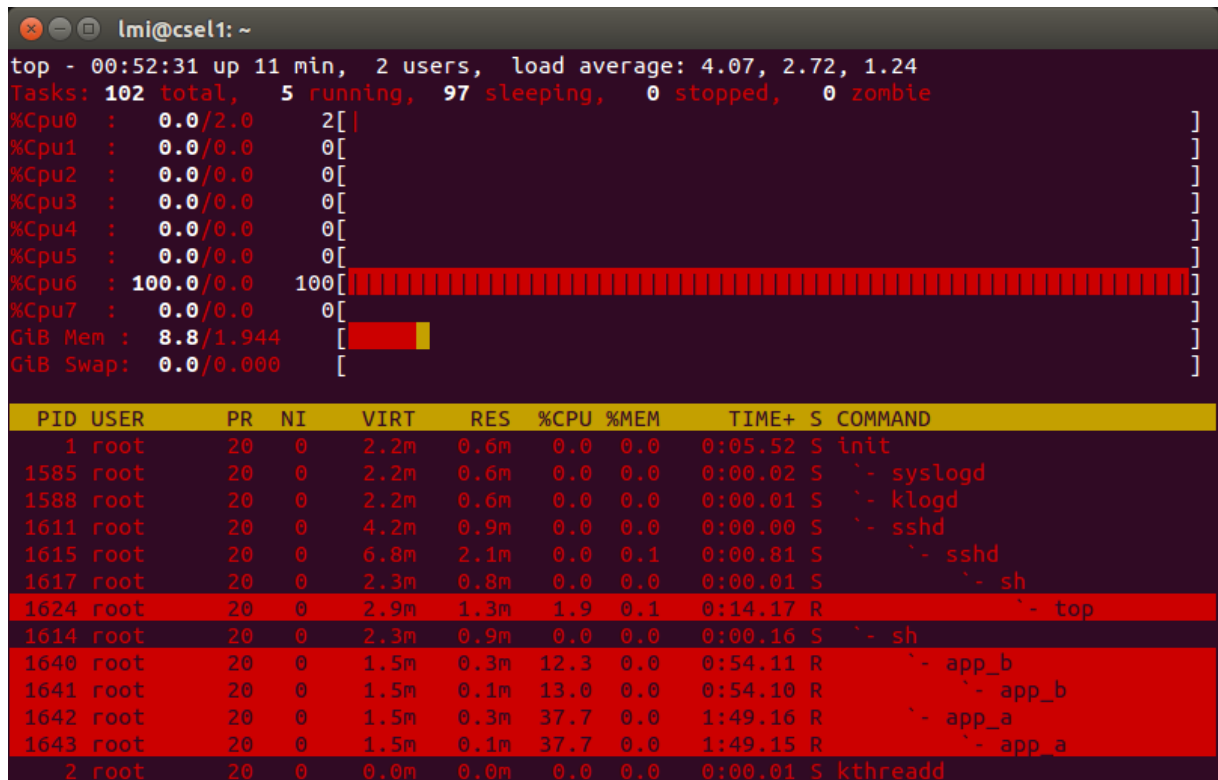


FIGURE 13 – Répartition du temps processeur

7 Optimisation : Performances

7.1 Installation de perf

En suivant pas à pas les instructions de la donnée, perf a pu être installé sans aucun problème sur la machine virtuelle.

7.2 Prise en main de perf

Cette première partie a été exécutée sur la cible ODROID4.

7.2.1 Exercice 1

Donnée : Décrivez brièvement ce que sont les événements suivants :

Hardware event

1. instructions : Compte le nombre d'instructions effectuées durant l'exécution du programme.
2. cache-misses : Compte le nombre de cache-misses, c'est à dire le nombre de fois où la mémoire cache n'avait pas l'information désirée et a dû aller la chercher dans la mémoire principale.
3. branch-misses : Les branch surviennent lorsqu'il y a par exemple des if-else. L'exécution est conditionnelle et donc pas toujours identique. Le système possède des algorithmes statistiques qui prévoient à l'avant comment le code sera exécuté (if ou else). Si la prédiction de l'algorithme est fausse, cela va incrémenté le compteur de branch-misses.

Hardware cache event

1. L1-dcache-load-misses : L'événement provient cette fois directement de la mémoire cache. Elle compte la même chose que le cache-misses

```
1 # perf stat -e L1-dcache-load-misses,cache-misses ./ex1
2 Performance counter stats for './ex1':
3
4 78862365 L1-dcache-load-misses
5 78862365 cache-misses
6
7 2.983792917 seconds time elapsed
8 #
```

Software event

1. cpu-migrations : Compte le nombre de fois où le programme a changé de cœur CPU durant son exécution.
2. context-switches : Compte le nombre de changement de contexte (nombre de fois que l'application a été interrompue par un autre programme)

7.2.2 Exercice 2

Donnée : Compilez (en utilisant le Makefile fourni) et exécutez le programme situé dans le dossier `exercices/01` en utilisant la commande “`perf stat`”

Emplacement du code : *performances/01/01-original*

Exécution du code :

Le temps d’exécution du code est relativement long. Une fois l’exécution terminée, les performances s’affichent dans la console. Les mesures relevées sont les compteurs de context-switches et d’instruction ainsi que le temps d’exécution de l’application.

```
1 lmi@cse11:~$ cd workspace/cse11/environment/exercices/01/  
2 lmi@cse11:~/workspace/cse11/environment/exercices/01$ make
```

```
1 # perf stat ./ex1  
2  
3 Performance counter stats for './ex1':  
4 ...  
5 56 context-switches          #      0.001 K/sec  
6 ...  
7 10453003007 instructions      #      0.11  insns per cycle  
8 ...  
9 45.705296089 seconds time elapsed
```

On constate que l’exécution du code prend énormément de temps.

7.2.3 Exercice 3

Donnée : Ouvrez maintenant le fichier “`main.c`” et analysez le code. Ce programme contient une erreur triviale qui empêche une utilisation optimale du cache. De quelle erreur s’agit-il ?

L’erreur vient du fait que l’on navigue dans le tableau en changeant de colonne. Cela donne énormément de travail à la cache qui doit aller chercher dans la RAM à chaque incrément de colonne. En navigant dans le tableau ligne par ligne, la cache à moins de travail, car elle peut charger une ligne entière, l’adresse ne change pas. Le temps d’exécution du code devrait être nettement plus rapide.

Emplacement du code : *performances/01/01-modified*

Modification du code :

```
1 for (k = 0; k < 50; k++)  
2 {  
3     for (i = 0; i < SIZE; i++)  
4     {
```



```
5   for (j = 0; j < SIZE; j++)
6   {
7       //array[j][i]++;
8       array[i][j]++;
9   }
10 }
11 }
```

7.2.4 Exercice 4

Donnée : Corrigez “l’erreur”, recompilez et mesurez à nouveau le temps d’exécution (soit avec perf stat, soit avec la commande time). Quelle amélioration constatez-vous ?

On constate que l’exécution de l’application est environ 15 fois plus rapide. La cache à moins besoin d’aller chercher des informations dans la RAM.

Exécution du code :

```
1 # perf stat ./ex1
2
3 Performance counter stats for './ex1':
4 ...
5 3 context-switches          #    0.001 K/sec
6 ...
7 6552198916 instructions      #    1.06  insns per cycle
8 ...
9 3.105906000 seconds time elapsed
```

7.2.5 Exercice 5

Donnée : Relevez les valeurs du compteur L1-dcache-load-misses pour les deux versions de l’application. Quelle facteur constatez-vous entre les deux valeurs ?

Exécution du code de base :

```
1 # perf stat -e L1-dcache-load-misses ./ex1
2
3 Performance counter stats for './ex1':
4
5 1252652488 L1-dcache-load-misses
6
7 45.638627964 seconds time elapsed
```

Exécution du code modifié :

```
1 # perf stat -e L1-dcache-load-misses ./ex1
2
3 Performance counter stats for './ex1':
4
5 78861576 L1-dcache-load-misses
6
7 3.097171626 seconds time elapsed
```

On obtient un facteur entre les L1-dcache-load-misses d'environ 15.9 entre les deux exécution. Cela prouve que notre modification a une réelle influence sur la cache.

7.2.6 Exercice 6

Donnée : Lors de la présentation de l'outil perf, on a vu que celui-ci permettait de profiler une application avec très peu d'impacts sur les performances. En utilisant la commande time, mesurez le temps d'exécution de notre application ex1 avec et sans la commande perf stat.

Emplacement du code : *performances/01/01-timeMeasure*

Exécution du code :

```
1 # ./ex1
2 elapsed time: 3052 [ms]
3 # perf stat ./ex1
4 elapsed time: 3072 [ms]
5
6 Performance counter stats for './ex1':
7 ...
8 #
```

La mesure de temps est affichée en millisecondes. On constate que la commande perf ne ralenti pas l'exécution du programme. La différence de temps entre les deux exécutions est de seulement 20ms.

7.3 Analyse et optimisation d'un programme

Sur la base du programme situé dans le dossier exercices/02/

7.3.1 Exercice 1

Donnée : Décrire en quelques mots ce que fait ce programme

Le programme est décomposé en deux parties :

- Première boucle générant un nombre aléatoire inférieur à 256 et supérieur ou égal à 0 qu'on place ensuite dans un tableau de 32768 positions.

- Boucle impriquée, où on somme 100000 fois chaque nombre supérieur ou égal à 128 présent dans le tableau cité précédemment.

7.3.2 Exercice 2

Donnée : Compilez le programme à l'aide du Makefile joint. Mesurez le temps d'exécution

Exécution du code :

```
1 #time ./ex2 > /dev/null
2 real    0m 27.58s
3 user    0m 27.57s
4 sys 0m 0.00s
```

On obtient donc 27.58s avec ce type de mesure.

7.3.3 Exercice 3

Donnée : Avant la fonction `int main()`, ajoutez la méthode suivante : `static int compare (const void* a, const void* b) return (short*)a - (short*)b;` Avant « `long long sum = 0;` », ajoutez le code suivant : `qsort(data, SIZE, sizeof(data[0]), compare);` Compilez et mesurez le temps d'exécution de la version modifiée

Avant que la somme soit calculée, un tri est effectué sur le tableau contenant les nombres aléatoires.

Voici la fonction de tri :

```
1 static int compare (const void * a, const void * b)
2 {
3     return *(short*)a - *(short*)b;
4 }
```

On obtient alors avec la commande suivante :

```
1 # time ./ex2 > /dev/null
2 real    0m 6.95s
3 user    0m 6.94s
4 sys 0m 0.00s
```

un temps environ 4 fois moindre de 6.94s.

7.3.4 Exercice 4

Donnée : Vous observez sans doute une nette amélioration sur le temps d'exécution. A l'aide de l'outil `perf` et de sa sous-commande `'stat'`, en utilisant différents compteurs déterminez pourquoi le programme modifié s'exécute plus rapidement.

Emplacement du code : *performances/02*

Les statistiques suivantes s'appliquent à l'application sans tri préalable. **Exécution du code :**

```

1 Performance counter stats for './test_not_sorted':
2
3 19545.395156      task-clock (msec)          #    1.000 CPUs utilized
4 11      context-switches          #    0.001 K/sec
5 1      cpu-migrations            #    0.000 K/sec
6 62      page-faults              #    0.003 K/sec
7 65'152'879'397    cycles                    #    3.333 GHz
8 32'321'278'233    stalled-cycles-frontend  #   49.61% frontend cycles idle
9 13'650'524'294    stalled-cycles-backend   #   20.95% backend  cycles idle
10 34'472'023'415    instructions             #    0.53  insns per cycle
11          #    0.94  stalled cycles per insn
12 6'558'873'421     branches                 #   335.571 M/sec
13 1'551'353'982     branch-misses            #    23.65% of all branches
14
15 19.550181930 seconds time elapsed

```

Emplacement du code : *performances/02/sorted*

Les statistiques suivantes s'appliquent à l'application avec tri préalable.

Exécution du code :

```

1 Performance counter stats for './test_sorted':
2
3 7422.583465      task-clock (msec)          #    1.000 CPUs utilized
4 6      context-switches          #    0.001 K/sec
5 0      cpu-migrations            #    0.000 K/sec
6 83      page-faults              #    0.011 K/sec
7 24'730'131'217    cycles                    #    3.332 GHz
8 14'861'808'053    stalled-cycles-frontend  #   60.10% frontend cycles idle
9 555'105'484      stalled-cycles-backend   #    2.24% backend  cycles idle
10 34'482'583'827    instructions             #    1.39  insns per cycle
11          #    0.43  stalled cycles per insn
12 6'563'029'266     branches                 #   884.197 M/sec
13 403'556          branch-misses            #    0.01% of all branches
14
15 7.424632902 seconds time elapsed

```

Le nombre qui ressort le plus en comparant les deux résultats est le pourcentage de branch-misses. Dans les processeurs modernes, un mécanisme de prédiction de branchements basé sur une étude statistique des derniers branchements effectués calcule la probabilité que le branchement suivant soit juste ou faux. Ainsi, si le mécanisme prédit correctement, on ne trouve pas le *pipe*, ce qui nous fait gagner de la vitesse.

Dans notre cas, les nombres sont triés dans l'ordre croissant, faisant que le mécanisme doive se *stabiliser* deux fois, une fois au début du parcours du tableau, où les nombres inférieurs à

128 sont rassemblés et une seconde fois quand la condition devient vraie tout le temps et que le mécanisme doit intégrer ça dans ses probabilités de branchement.

Quand le tableau n'est pas trié, il est impossible pour le mécanisme de calculer des probabilités qui soient solides. Aussi, il est plus facile de se tromper alors et de casser le *pipe* car on doit aller chercher une autre instruction que celle déjà *fetchée*.

7.4 Parsing de logs apache

7.4.1 Exercice 1

Donnée : Compilez l'application en utilisant le Makefile fourni et mesurez sur la machine virtuelle son temps d'exécution avec la commande "time".

Exécution du code :

```
1 time ./read-apache-logs access_log_NASA_Jul95_samples
2 Processing log file access_log_NASA_Jul95_samples
3 Found 14867 unique Hosts/IPs
4
5 real 0m19.116s
6 user 0m19.130s
7 sys 0m0.003s
```

Le code a été exécuté sur une machine Archlinux, kernel 4.2.5-1, Intel Core I5 Broadwell.

7.4.2 Exercice 2

Donnée : Nous allons maintenant profiler l'application avec perf record avec l'option -g.

```
1 $ perf record -g ./read-apache-logs access_log_NASA_Jul95_samples
```

L'exécution de cette commande doit produire un fichier de résultat perf, nommé perf.data. Si l'on exécute une nouvelle fois la commande, ce fichier sera copié vers perf.data.old et un nouveau perf.data correspondant à la dernière exécution sera créé. A quoi sert l'option -g ? En quoi son résultat être utile ?

L'option -g active l'enregistrement du call graph du programme. Ceci permet de voir les appels de fonctions,

7.4.3 Exercice 3

Donnée : Nous pouvons maintenant analyser les données collectées par perf avec la commande perf report. Cette commande lance un outil interactif (interface console de type "ncurses") : il est donc possible d'interagir avec les touches du clavier. Chaque ligne représente une fonction, celles récoltant le plus d'évènements sont montrées en haut. En vous basant sur les informations disponibles dans le wiki de perf (<https://perf.wiki.kernel>).

org/index.php/Tutorial), décrivez ce que représente chaque colonne.

1. Cette colonne décrit le pourcentage de temps par rapport au programme qui est passé dans cette fonction.
2. La seconde reporte le processus du quel les échantillons ont été pris. Dans le mode par-thread/par-processus, c'est toujours le nom de la commande monitorée.
3. Celle-ci montre le binaire duquel proviennent les échantillons. Si le programme est lié dynamiquement, des noms de bibliothèques peuvent apparaître.
4. La quatrième montre les privilèges alloués de la section prise en compte au moment où elle a été échantillonnée.
5. La dernière donne le nom du symbole, comme par exemple le nom de la fonction monitorée.

7.4.4 Exercice 4

Donnée : Sur la capture ci-dessus, on voit par exemple que la majorité des cycles de l'application sont passés dans la fonction `std::string::size()` qui est contenue dans la bibliothèque standard. Il nous manque cependant une information capitale : quelle fonction de notre application fait appel à cette fonction ? Grâce à l'option `-g` passée à `perf record`, nous pouvons afficher le call-graph complet. Il suffit de naviguer dans l'interface et développer la ligne `std::string::size()` avec la touche "Enter". Avec les instructions précédentes, déterminez quelle fonction de notre application fait (indirectement) appel à `std::string::size()`.

```

1 std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
  >::size
2   __gnu_cxx::__ops::_Iter_equals_val<std::__cxx11::basic_string<char, std::
    char_traits<char>, std::allocator<char> > const>::operator()<__gnu_cxx
    ::__normal_iterator<
3   std::__find_if<__gnu_cxx::__normal_iterator<std::__cxx11::basic_string<
    char, std::char_traits<char>, std::allocator<char> >*, std::vector<std
    ::__cxx11::basic_strin
4   std::__find_if<__gnu_cxx::__normal_iterator<std::__cxx11::basic_string<
    char, std::char_traits<char>, std::allocator<char> >*, std::vector<std
    ::__cxx11::basic_strin
5   std::find<__gnu_cxx::__normal_iterator<std::__cxx11::basic_string<char,
    std::char_traits<char>, std::allocator<char> >*, std::vector<std::
    __cxx11::basic_string<cha
6   HostCounter::isNewHost
7   HostCounter::notifyHost
8   ApacheAccessLogAnalyzer::processFile
9   main
10  __libc_start_main
11  0x128e258d4c544155

```

Astuce : l'aide apparaît avec la touche `?` et un filtrage peut être effectué grâce à `/`, ce qui fait apparaître une petite case de dialogue où on peut entrer une recherche.

7.4.5 Exercice 5

Donnée : Maintenant que vous savez quelle fonction utilise le plus de ressources CPU, trouvez une optimisation du code permettant de réduire drastiquement le temps d'exécution (vous devriez arriver à quelques dixièmes de secondes pour le fichier sample).

Emplacement du code : */performances/03/HostCounter.h*

```
1 #include <string>
2 #include <set>
3
4 class HostCounter
5 {
6     public:
7         HostCounter();
8         void notifyHost(std::string hostname);
9         int getNbOfHosts();
10    private:
11        bool isNewHost(std::string hostname);
12        std::set< std::string > mySetOfHots;
13};
```

Emplacement du code : */performances/03/HostCounter.cpp*

```
1 #include "HostCounter.h"
2 #include <set>
3
4 HostCounter::HostCounter() {
5 }
6
7 bool HostCounter::isNewHost(std::string hostname){
8     return mySetOfHots.find(hostname) == mySetOfHots.end();
9 }
10
11 void HostCounter::notifyHost(std::string hostname){
12     if(isNewHost(hostname))
13         mySetOfHots.insert(hostname);
14 }
15
16 int HostCounter::getNbOfHosts() {
17     return mySetOfHots.size();
18 }
```

Exécution du code :

```
1 time ./read-apache-logs access_log_NASA_Jul95_samples
2 Processing log file access_log_NASA_Jul95_samples
3 Found 14867 unique Hosts/IPs
4
5 real 0m0.229s
6 user 0m0.223s
7 sys 0m0.003s
```

8 Mini Projet - Programmation noyau et système

8.1 Travail à réaliser

Concevez une application permettant de gérer automatiquement et manuellement la vitesse de rotation du ventilateur de l'Odroid-XU3 Lite.

Cette application réalisera la fonctionnalité minimale suivante :

1. La supervision de la température du microprocesseur et la gestion automatique de la vitesse du ventialeur devront être réalisées par un module noyau. Ce module devra offrir via le sysfs une interface permettant de choisir :
 - le mode automatique ou manuel
 - le duty cycle à appliquer au PWM en mode manuel
2. La gestion du mode manuel devra être implémentée par un daemon en espace utilisateur. Ce daemon proposera deux interfaces distinctes :
 - (a) Interface physique via les boutons poussoir et LEDs de la carte d'extension
 - SW1 pour augmenter la vitesse de rotation du ventilateur, la pression du SW1 devra être signalisée sur la LED1
 - SW2 pour diminuer la vitesse de rotation du ventilateur, la pression du SW2 devra être signalisée sur la LED2
 - SW3 pour changer du mode automatique au mode manuel et vice versa. Le mode manuel sera signalé avec la LED3 allumée
 - (b) Interface IPC au choix du développeur permettant de dialoguer avec une application pour choisir le mode de fonctionnement et spécifier la valeur du duty cycle
3. Selon le choix de l'interface IPC, une petite application implémentant une interface utilisateur pour piloter le ventilateur via le daemon devra être réalisée.

8.2 Installation des différentes parties

Le miniprojet est constitué d'un module noyau, d'un daemon et d'une application utilisateur. Voici les différentes étapes pour les installer. Il faut bien évidemment que chaque partie aie été préalablement compilée.

8.2.1 Daemon

Emplacement du code : */miniprojet/daemon/*

```
1 // Install
2 # cp /usr/workspace/csel1/environment/miniProjet/daemon/S70appl /etc/init.d/
3 # reboot
4
5 // Uninstall
6 # rm /etc/init.d/S70appl
7 # reboot
```


8.2.2 Module noyau

Emplacement du code : */miniprojet/module_noyau/*

```
1 //Install
2 # modprobe mymodule
3
4 //Uninstall
5 # modprobe -r mymodule
```

8.2.3 Application utilisateur

Emplacement du code : */miniprojet/user/*

```
1 # cd /usr/workspace/csel1/environment/miniProjet/user/
2 # ./app_h
```

8.3 Daemon en espace utilisateur

Le daemon se lance automatiquement au démarrage de l'ODROID s'il a été correctement installé. Ses messages ont été redirigés dans le syslog. On peut les voir à l'aide des commandes suivantes :

```
1 //Pour voir les derniers messages
2 # cat /var/log/messages | tail -f
3
4 //ou pour afficher les messages en temps reel
5 # tail -f /var/log/messages
6
7 Jan  1 04:54:23 odroidxu3 daemon.info app_a[1620]: Daemon Mode manual
8 Jan  1 04:54:23 odroidxu3 user.info kernel: [ 1941.377057] [c5] Fan Control :
   changed mode to manual
9 Jan  1 04:54:23 odroidxu3 user.info kernel: [ 1941.380725] [c5] Duty set to :
   10486
10 Jan  1 04:54:24 odroidxu3 daemon.info app_a[1620]: daemon Duty 48
11 Jan  1 04:54:24 odroidxu3 user.info kernel: [ 1942.920102] [c5] Fan Control :
   duty of 48 trying to be set
12 Jan  1 04:54:24 odroidxu3 user.info kernel: [ 1942.924106] [c5] Fan Control :
   duty of 48 was set
13 Jan  1 04:54:24 odroidxu3 user.info kernel: [ 1942.928857] [c5] Duty set to :
   10066
14 Jan  1 04:54:25 odroidxu3 daemon.info app_a[1620]: daemon Duty 50
15 Jan  1 04:54:25 odroidxu3 user.info kernel: [ 1943.713271] [c5] Fan Control :
   duty of 50 trying to be set
16 Jan  1 04:54:25 odroidxu3 user.info kernel: [ 1943.717325] [c5] Fan Control :
   duty of 50 was set
17 Jan  1 04:54:25 odroidxu3 user.info kernel: [ 1943.722008] [c5] Duty set to :
   10486
```

```
18 | Jan  1 04:54:27 odroidxu3 daemon.info app_a[1620]: daemon Duty 48
```

À l'aide des boutons de la carte d'extension, on peut gérer le duty cycle du ventilateur et le mode manuel/auto. Le duty cycle du ventilateur n'est modifié que lorsque l'on est en mode manuel (option du module noyau). Les LEDs s'allument en fonction de la pression sur les boutons.

Une option a été ajoutée au cahier des charges. Quand on tient longtemps un bouton appuyé, le duty cycle s'auto incrémente.

L'IPC choisi est une fifo entre l'application utilisateur et le daemon.

Le daemon contient deux processus, un pour les boutons, l'autre pour l'IPC.

Warning : Contrairement à l'exemple des slides, le "passwd" pour donner les droits a du être changé à "root" au lieu de "daemon", car les gpio sont importées/exportées après l'initialisation dans le daemon body. Normalement, cette action devrait être faite avant.

8.4 Module de contrôle du ventilateur

Comme le module doit proposer une interface dans le sysfs permettant de régler le mode de fonctionnement et la vitesse en cas de mode manuel, deux fichiers d'attributs ont été créés :

- duty : permet, en mode manuel, de régler la vitesse de rotation du ventilateur. Un pourcentage doit lui être fourni.
- mode : permet de définir le mode de fonctionnement, automatique ou manuel.

Pour les utiliser, voici le type de commande qui peut être écrit :

1. Réglage du mode

```
1 echo manual > /sys/devices/platform/Fan_control_module/mode
2 echo auto > /sys/devices/platform/Fan_control_module/mode
```

2. Réglage du duty cycle

```
1 echo 52 > /sys/devices/platform/Fan_control_module/duty
```

3. Lecture du mode

```
1 cat /sys/devices/platform/Fan_control_module/mode
```

Retourne une chaîne de caractère ascii *auto* ou *manual*

4. Lecture du duty cycle

```
1 cat /sys/devices/platform/Fan_control_module/duty
```

Retourne un duty cycle en pourcents

8.5 Application utilisateur

L'application utilisateur permet de transmettre des commandes au daemon qui les retransmet au module noyau. Voici les trois commandes reconnues :

```

1 cd /usr/workspace/csel1/environment/miniProjet/user/
2 # ./app_h
3 User app
4 manual
5 auto
6 manual
7 duty10

```

Le module noyau n'accepte pas de duty cycle de 0.

Parfois la chaîne de caractère est mal transmise, le daemon reçoit par exemple "anual" au lieu de manual. Il suffit de retaper la commande. Il ne doit pas y avoir d'espace entre duty et le chiffre en pourcent désiré.

8.6 Mesure de performances

Les performances du miniprojet ont été mesurées avec la commande top :

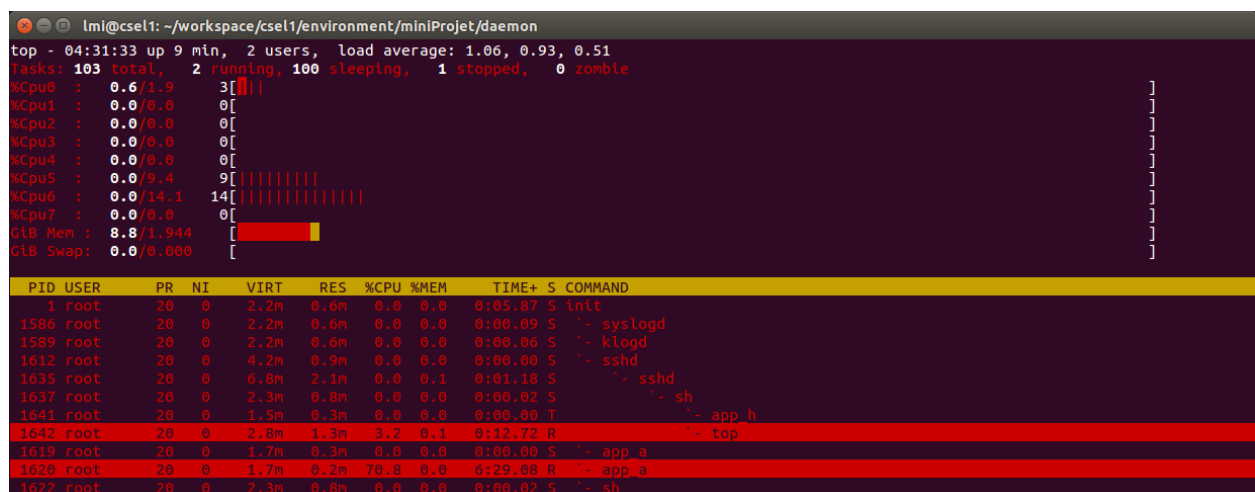


FIGURE 14 – Mesure de l'occupation du processeur

On peut constater que nos différentes parties occupent peu de place au niveau du processeur.

8.7 Points à améliorer

Comme pour le projet sur le fan control, les boutons n'ont pas d'anti-rebond, ils restent parfois bloqués dans le mauvais état.

Les valeurs de duty cycle et le mode géré par la carte d'extension et par l'application utilisateur ne sont pas synchronisés. Les deux tâches sont faites dans deux processus différents. Le duty cycle incrémenté par les boutons n'est pas identique au duty entré dans l'application

utilisateur. Cela ne pose pas de problème particulier au niveau de l'utilisation, mais pourrait être fait en faisant par exemple un pipe entre les deux processus pour synchroniser les données.