CSEL

Rapports des TP

Fribourg, April 7, 2022

Auteurs
Macherel Rémy
remy.macherel@master.hes-so.ch
Raemy Mathis
mathis.raemy@master.hes-so.ch

Version: 1.0

Contents

Chapter 1

TP 1

1.1 Résumé du travail pratique 1

Dans ce TP, nous allons mettre en place un environnement de développement afin de faciliter les développements futures. Cet environnement utilise des container Docker (Docker-compose) pour assurer la compatibilité entre les différentes machines. Un container est composé de tous les outils nécessaire à la cross-compilation et l'autre pour la mise en place d'une partition réseau SAMBA afin de partager facilement les programmes cross-compilés avec le NanoPi.

1.2 Réponses aux questions

1.2.1 Comment faut-il procéder pour générer le U-Boot ?

On peut regénérer le u-boot uniquement avec la commande : make uboot-rebuild
See : u-boot makefile
–> TODO : Ajouter 5-6 lignes qui expliquent ce que fais le make de U-Boot

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

e ne comprends pas trop le sens de la question mais au sens général, il faut ajouter le binaire ainsi que des fichiers de configurations (qui définissent par exemple les dépendances). On peut par exemple importer depuis GitLab automatiquement un packet, ou encore ajouter des paquet pour le user space ou le kernel space et tout cela change les étapes a effectuer. Pour plus d'informations, la section 18 de la documentation.

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

La commande make linux-menuconfig permet de configurer le noyau et on peut ensuite ajouter/enlever des modules du noyaux et finalement faire make linux-rebuild afin de reconstruire le noyau.

1.2.4 Comment faut-il faire pour générer son propre rootfs ?

Comme c'est Buildroot qui s'occupe de générer le Rootfs, c'est lui qu'il faut configurer
cd /buildroot
make menuconfig
On peut aussi par exemple modifier le skelette utilisé pour modifier la structure des dossiers du rootfs. On peut aussi ajouter des fichiers/scripts/dossiers par défaut dans le rootfs en ajoutant un rootfs_overlay */buildroot/board/friendlyarm/nanopi-neo-plus2/rootfs_overlay/*. Ils seront automatiquement inséré dans le rootfs après chaques générations.

1.2.5 Comment faudrait-il procéder pour utiliser la carte eMMC en lieu et place de la carte SD ?

Il faut commencer par modifier le script de démarrage en modifiant les deux lignes suivantes. Il faut remplacer le **0** par le numéro de MMC que l'on veut :
fatload mmc **0** $kernel_addr_r Image
fatload mmc **0** $fdt_addr_r nanopi-neo-plus2.dtb
On peut avoir des informations sur les mmc disponible en interrompant le U-Boot et en tappant la commande :

```
=> mmc list mmc@1c0f000: 0 (SD) mmc@1c10000: 2 mmc@1c11000: 1
```

Il faut ensuite recompiler le script avec la commande make et reflasher la carte. Enfin il ne faut pas oublier également de charger l'Image et le device tree depuis la carte SD vers la mmc interne (a faire une seul fois) ainsi que de modifier le script de démarrage pour utiliser celui que l'on vient de compiler avec la commande :

```
setenv boot_scripts boot.cifs
```

Finalement, on peut sauvegarder les modifications avec la commande :

```
saveenv
```

1.2.6 Dans le support de cours, on trouve différentes configurations de l'environnement de développement. Qu'elle serait la configuration optimale pour le développement uniquement d'applications en espace utilisateur ?

Si on développe uniquement dans le userspace, on pourrait imaginer que l'on flash la eMMC comme décrit à la question précédente pour s'affranchir de carte SD et qu'on développe sur notre ordinateur. Pour le partage des programmes une partition NFS ou la partition CIFS sont très bien et permettent de rendre transparent le transfert des programmes.

1.3 Synthèse des connaissances acquises

1.3.1 Non acquis

Aucune

1.3.2 Acquis, mais à exercer

Les Makefile, debugging (GDB/HDB Server), Principe CIFS

1.3.3 Parfaitement acquis

Le reste comme par exemple ce qui est commande, connexion, Linux (déjà globalement vu dans un cours précédent)

1.4 Infos utiles à retenir

Pour monter automatiquement un disque réseau cifs :

```
1  echo "//<server_ip>/workspace /workspace cifs username=<username>,
2  password=<password>,port=1445,noserverino" >> /etc/fstab
```

Lorsqu'on créer une nouvelle partition, il faut l'aligner sur 2048 bits. On peut calculer le début de la prochaine partition par rapport à la précédente comme ceci :
<prev_end_sector> + (2048 - <prev_end_sector> % 2048)

1.5 Feedback global

Dans l'ensemble laboratoire très intéressant et bon point de départ pour un cours comme celui-ci. Assez dirigé pour nous indiquer la bonne voie mais pas trop pour qu'on puisse tester des choses. Nous aurions bien souhaité un peu plus de maitère sur les Makefiles.
Nous avons également pour le flash de la carte SD développé un script permettant ceci de manière automatique (à voir dans le dossier script le FlashSDCard.sh). Ce script permet donc le flash de la SD Card avec les fichiers nécessaires sans utiliser le logiciel *Balena*. Afin de l'utiliser, il suffirait de modifier le chemin d'accès à l'image (ligne 2 du script). Il a également été très intéressant de découvrir les differentes méthodes de debugging possibles lors du développement de systèmes embarqués. Parmis celles vues en cours, nous avons particulièrement testé la méthode utilisant GDB / GDB Server. La capacité de cette méthode à réaliser du débugging à distance était pour nous un point fort. Cette méthode permettait entre autres d'exécuter le code présent sur la cible pas à pas et de découvrir de potentielles erreurs.

1.5.1 Problèmes rencontrés

Pour un des membres du groupe, une fois le CIFS mis en place, le nanoPi était incapable de démarrer et semblait relancer sa séquence de boot indéfiniment. Nous avons par la suite déterminé que ceci était dû à une trop faible alimentation via un HUB USB qui faisait que lorsque le nanoPi tentait de se connecter via le réseau la tension n'était pas suffisante et celui-ci s'éteignait puis rebootait directement.

Chapter 2

TP 2

2.1 Résumé du laboratoire

Ce laboratoire concerne les modules noyaux. Il est très important et utile afin de mettre en pratique la manière vue durant les cours sur ces modules. Il permet de parcourir les différentes possibilités de ces modules comme par exemple:

- Modules out of tree
- Gestion de la mémoire
- Accès I/O
- Threads dans le noyau
- Mise en sommeil (relatif aux threads)
- Interruptions

Durant les différents exercices de ce travail pratique nous avons parcouru ces possibilités et implémenté chacune d'entre elles. Les fichiers sources se trouve dans le dossier cselworkspace/src/02_module/exercice..

2.2 Réponses aux questions

2.2.1 Trouvez la signification des 4 valeurs affichées lorsque l'on tape la commande cat /proc/sys/kernel/printk

Cela affiche les console_loglevel (niveaux de logs de la console) actuels, dans notre cas sur le nanopi les valeurs affichées sont 7 4 1 7, comme le montre la documentation l'ordre des chiffres correspond à :

1. current
2. default
3. minimum
4. boot-time default

Name	String	Alias function
KERN_EMERG	“0”	`pr_emerg()`
KERN_ALERT	“1”	`pr_alert()`
KERN_CRIT	“2”	`pr_crit()`
KERN_ERR	“3”	`pr_err()`
KERN_WARNING	“4”	`pr_warn()`
KERN_NOTICE	“5”	`pr_notice()`
KERN_INFO	“6”	`pr_info()`
KERN_DEBUG	“7”	`pr_debug()` and `pr_devel()` if DEBUG is defined
KERN_DEFAULT	“”	
KERN_CONT	“c”	`pr_cont()`

Figure 2.1: Console log levels correspondance

2.3 Synthèse des connaissances acquises

2.3.1 Non acquis

2.3.2 Acquis, mais à exercer

2.3.3 Parfaitement acquis

2.4 Feedback exercices

2.4.1 Exercice 01

L’exercice 01 ne nous a globalement pas posé de problèmes, nous avons suivi les slides du cours afin de créer le skeleton.c ainsi que son makefile. Nous l’avons ensuite compilé puis chargé dans le nanoPi. Nous avons ensuite effectué la commande *dmesg* afin de vérifier que le noyau a bien été chargé en affichant les *printk* du code.

2.4.2 Exercice 02

Dans cet exercice, il suffisait d’adapter le module pour qu’il puisse recevoir des paramètres, pour ce faire, nous avons ajouté comme les lignes de la figure suivante à la suite des include.

```
static char* text = "dummy text";
module_param(text, charp, 0664);
static int  elements = 1;
module_param(elements, int, 0);
```

Figure 2.2: Add modules parameters

Nous avons suivi la démarche décrite dans le cours. Les paramètres se définissent donc de la manière suivante dans la fonction *moduleparam*:
nom de la variable, type de la variable, permissions du fichier correspondant dans le sysfs. Il serait aussi possible d’ajouter une description au paramètre en utilisant la macro

MODULE_PARAM_DESC(nomVariable,'description'). Les informations concernant ceci se trouvent dans le cours ainsi que sur ce site.

2.4.3 Exercice 3

Voir 2.2.1.

2.4.4 Exercice 4

L'exercice 4 consistait à définir en paramètre du module un nombre d'élément à créer ainsi qu'un texte à créer dans ces éléments. Nous avons créé une structure *elem* contenant le text ainsi qu'un identifiant et un pointeur vers l'élément suivant. Dans la fonction d'initialisation du module on se charge de créer un nombre d'éléments égal au paramètre transmis. Ensuite dans la méthode de sortie du module nous utilisons *kfree* afin de libérer la mémoire allouée pour chaque élement.

2.4.5 Exercice 5

Dans cet exercice il s'agissait de développer un module noyau permettant d'afficher diverses informations sur le processeur. Il 'sagissait d'utiliser les fonctions *request_mem_region* afin de demander une région mémoire. Nous nous sommes tout d'abord interrogés sur le nombre de bytes à réserver puis après discussion avec le professeur nous avons décidà de réserver l'entièreté de la page (soit 0x1000). Nous avons également remarqué que la plupart du temps cette fonction se retrouvait avec un code d'erreur (0) à sa sortie. Cela est certainement dû au fait que des informations telles que la température ou la MAC adress sont certainement souvent requises par d'autres processus et donc déjà réservées. Nous avons ensuite utilisé *ioremap* afin de mapper dans la mémoire virtuelle les adresses physiques requises pour que l'on puisse ensuite lire les valeurs nécessaires.

2.4.6 Exercice 6

Rien de spécial à signaler dans cet exercice. Il s'agissait d'instantier un thread qui affiche un message toutes les 5s.

2.4.7 Exercice 7

Nous avons initialement dans l'exercice 7 peiné à utiliser les fonctions wait_event_interruptible et wake_up_interruptible. Nous avions tout d'abord pensé qu'il n'était pas nécessaire d'utiliser un verrou atomique car nous pensions que la fonction wake_up_interruptible générait le signal permettant de réveiller le wait, cependant il s'est avéré que ce n'était pas le cas et nous avons donc dû (comme dans la solution fournie) implémenter un verrou permettant de bloquer et de signaler au wait que celui-ci doit se réveiller et afficher son message. Nous avons donc conclus que la queue était présente pour que le thread se mette en attente si la condition est fausse. Dans notre cas, comme au début nous avions mis une condition à true en permanence cela tournait dans une boucle infinie car le fait que la condition soit vraie lui permettait de toujours passer par dessus la queue. Le wake_up sert donc juste à indiquer au wait qu'il doit re-vérifier sa condition. Ces informations furent découvertes en faisant des tests ainsi qu'en discutant lors de la séance de laboratoire.

2.4.8 Exercice 8

Chapter 3

TP 3

3.1 Résumé du laboratoire

3.2 Réponses aux questions

3.3 Synthèse des connaissances acquises

3.3.1 Non acquis

3.3.2 Acquis, mais à exercer

3.3.3 Parfaitement acquis

3.4 Feedback

3.4.1 Exercice 01

Les pilotes orientés mémoire permettent d'accéder directement aux registres en programmant la MMU sans avoir besoin de créer un module noyau. Pour cet exercice il s'agissait de créer un pilote orienté mémoire afin de permettre de lire le Chip-ID du processeur. Nous n'avons pas rencontré de problème particulier dans la réalisation de cet exercice.

3.4.2 Exercice 02

Afin de stocker dans une variable globale du pilote des données reçues par les commandes echo (write) et cat (read) nous avons implémenté un pilote orienté caractère. Pour ce faire il fallait (comme démontré dans les slides du cours) implémenter les différentes *file operations* que sont read,write,open,release afin que celles-ci permettent les opérations demandées dans la consigne. Une fois ceci fait, la démarche veut que l'on place dans une structure de type *file_operations* ces différentes opérations comme le montre la figure suivante :

```
static struct file_operations skeleton_fops = {
    .owner   = THIS_MODULE,
    .open    = skeleton_open,
    .read    = skeleton_read,
    .write   = skeleton_write,
    .release = skeleton_release,
};
```

Figure 3.1: File operations structure

Une fois ces operations implémentées, il faut, dans l'initialisation du module, utiliser *alloc_chrdev_region* afin d'allouer un espace pour notre device. Si cette allocation se passe correctement, on peut utiliser *cdev_init* afin d'initialiser le device avec ses file operations. Ces opérations permettent de définir les opérations supportées par notre pilote.
A la destruction (exit) du device, on peut supprimer celui-ci à l'aide de *cdev_del* et désallouer la mémoire avec *unregister_chrdev_region*.
Lorsque le code est écrit il suffisait de le compiler a l'aide la commande *make*, puis sur la cible utiliser

```
insmod mymodle.ko #(si pas d'arguments) ou
insmod mymodule.ko <argName1>=<value1> <argName2>=<value2> #(si
↪ arguments)
```

afin de charger le device. Dans notre cas, nous n'avions pas print le major number dans les fonctions d'initialisation du device et donc afin de retrouver celui-ci nous avons utilisé la commande

```
cat /proc/devices
```

pour le retrouver. Une fois le major number connu, nous avons effectué la commande

```
mknod /dev/mymodule c 511 0
```

afin de créer un node pour le device. Une fois ceci fait il suffisait de tester l'écriture et la lecture avec par exemple

```
echo "salut" > /dev/mymodule #(pour écrire)
cat /dev/mymodule #(pour lire)
```

Une fois que le node est créé, on peut lister le major/minor avec la commande

```
# ls -al /dev/mymodule
crw-r--r-- 1 root root 511, 0 Jan  1 01:35 /dev/mymodule
```

Figure 3.2: Major and Minor number

3.4.3 Exercice 03

L'exercice 03 est une adaptation de l'exercice 2 afin de pouvoir définir par paramètre (par défaut 3) le nombre de devices pouvant être créés. Afin de tester ce fonctionnement nous avons tout d'abord adapté le code (voir code source), puis nous avons essayé de créer 4 nodes (commande *mknode* avec 4 minor number différents). La création s'est effectuée mais lorsque l'on souhaite lire ou écrire dans le 3ème node cela ne marche pas. Cela est logique car le nombre de noeuds est limité à 2, cela affiche que le device n'existe pas. Exemple :

```
# insmod ./mymodule.ko instances=2
# mknod /dev/mymodule c 511 0
# mknod /dev/mymodule1 c 511 1
# mknod /dev/mymodule2 c 511 2
# echo "test0" > /dev/mymodule
# echo "test1" > /dev/mymodule1
# echo "test2" > /dev/mymodule2
-sh: can't create /dev/mymodule2: No such device or address
# cat /dev/mymodule
test0
# cat /dev/mymodule1
test1
# cat /dev/mymodule2
cat: /dev/mymodule2: No such device or address
```

Figure 3.3: Exercice 03 tests

3.4.4 Exercice 04

Nous n'avons pas spécialement eu de difficultés à réaliser ces exercices mais ceci nous a permis de tester l'écriture dans un character device et de voir que l'on peut écrire jusqu'à ce que le buffer soit plein (le retour est -EIO). Une fois que celui-ci est plein, le programme casse la boucle et on va lire le contenu du device.

3.4.5 Exercice 05

Implémentation d'une classe permettant de valider le fonctionnement du sysfs. Nous avons décidé de réaliser l'implémentation uniquement avec une classe et non des misc ou platform device. Nous avons rencontrés passablement de problèmes avec l'utilisations de macros telles que *DEVICE_ATTR* et de fonctions comme *device_create_file* car certains de leurs arguments ne nous étaient pas clair et il nous a été assez compliqué de trouver de la documentation dessus.

3.4.6 Exercice 07