Rapport Lab 4

Kernel patching and cross-compilation for RPi

Clément **COLIN**, Xavier **GERONIMI**, Robin **LECLERC**

L'objectif de ce lab est de modifier le kernel du RaspberryPi afin de le rendre entièrement préemptif temps réels, c'est-à-dire, de manière à ce que l'ordonnanceur puisse interrompre à tout moment une tâche en cours d'exécution.

Download/Set the good version of Linux sources

Dans cette partie, nous téléchargerons les sources pour le nouveau kernel souhaité, puis nous nous assurerons de leur compatibilité avec le Raspberry Pi.

Question 1.

Nous installons *git* via la commande **sudo apt-get install git-core libncurses5-dev**.
sudo nous confère les droits d'administration pour réaliser l'opération.
apt-get avec l'option **install** est la commande qui permet d'installer le programme souhaité.

Question 2.

a)

Nous créons un dossier appelé *kernel_labs* dans notre home grâce à la commande **mkdir kernel_labs**.
mkdir est la commande qui permet de créer un dossier.

b) et c)

Nous récupérons les sources du kernel et les outils nécessaires à la compilation via les commandes **git clone https ://github.com/raspberrypi/tools.git** et **git clone -b rpi-4.4.y https ://github.com/raspberrypi/linux.git**.
Nous utilisons ici **git**, téléchargé à la question 1.

Question 3.

a)

Nous téléchargeons **get_hash.sh** depuis **campus.ece.fr**.

b)

Nous nous connectons au RaspberryPi grâce à son adresse IP *194.169.0.24* via la commande **ssh pi@194.169.0.24**.
Nous utilisons la commande **scp** de manière à pouvoir transférer des fichiers sur notre RasberryPi. Cette commande contient en argument le chemin vers le fichier source */Bureau/DossierPartage/files_lab4/get_hash.sh* , et celui de la destination *pi@193.169.0.36*.

c)

Connecté au RasberryPi, nous utilisons la commande **chmod** pour changer les droits du fichier transféré. Les droits *755* ont pour signification :

	Lecture	Ecriture	Exécution	Total
Propriétaire	400	200	100	700
Groupe	40	0	10	50
Autres	4	0	1	5
				755

d)

On lance le script *get_hash.sh* via **./get_hash.sh** qui indique le chemin relatif du script. Le résultat nous donne la version entièrement compatible avec le RaspberryPi.

e)

En se déplaçant dans *Bureau/kernel_labs/linux* via la commande **cd**, nous lançons la commande **git checkout** qui prend en argument le numéro de <versionhash> de manière à pointer sur la branche souhaitée.

f)

Pour finir, nous nettoyons l'arborescence du kernel grâce à l'option **mrproper** de manière à ne plus avoir les fichiers inutiles de la première version. C'est une question de sécurité.

Patch the kernel

Dans cette partie, nous appliquerons le patch apportant les modifications souhaitées.

Question 1.

Nous identifions la version des sources du patch via la commande **head -n 3 Makefile**.

Question 2.

Nous téléchargeons, sur *kernel.org*, la version la plus à jour du patch *PREMEPT-RT*.

Question 3.

Nous patchons le kernel via les commandes **gunzip patch-<version.patch-level.sub-level>-rt<last>.patch.gz** pour dézipper le fichier téléchargé et **cat patch-<version.patch-level.sub-level>-rt<last>.patch | patch -p1** pour appliquer le patch.
Dans le fichier patch, nous sommes obligés de spécifier les fichiers que l'on veut modifier, et dans ces fichiers, les lignes que l'on veut modifier. C'est pour cette raison qu'il faut lire le fichier *patch* grâce à la commande **cat**, et, mettre cette donnée en entrée à la commande **patch**.

Question 4.

Nous créons le dossier *rt-modules* grâce à la commande **mkdir ../rt-modules** et nous exportons la variable d'environnement *INSTALL_MOD_PATH* via la commande **export INSTALL_MOD_PATH= /Bureau/kernel_labs/rt-modules**.

Configure cross-compilation

Dans cette partie, nous allons configurer la compilation croisée. L'objectif étant d'activer le mode fully preemptible *(complètement préemptif), il faut configurer les sources avant de compiler, au risque d'appliquer la configuration par défaut (volontary : normal kernel). C'est pour cette raison qu'il nous faut générer un fichier .config.*

Question 1.

Nous exportons les variables nécessaires à la compilation croisée via la commande **export**.

Question 2.

L'objectif est d'avoir une bonne configuration qui va correspondre à l'architecture du RaspberryPi. En effet, le dossier arch contient toutes les ar-

chitectures qui existent (x86,...). On a besoin de savoir pour quel type d'architecture la compilation va être lancée (ici, c'est l'architecture ARM). La question qu'il faut se poser est : comment allons-nous lire l'architecture du RaspberryPi, c'est-à-dire, comment va-t-on savoir la configuration du kernel qui va correspondre avec le RaspberryPi ?
Nous avons deux possibilités pour répondre à cette question. La 2.1 ou 2.2. La méthode 2.2 est la plus sûre car elle nous garantit à 100% que l'on a la MÊME configuration qui tourne sur le RaspberryPi. Nous n'avons pas choisi la méthode 2.1 car elle peut être différente de la configuration qui tourne sur le kernel de la RaspberryPi.

a)

Nous allons donc récupérer la configuration du kernel qui tourne sur le RaspberryPi grâce à la commande **modprobe**. Nous avons donc tapé **sudo modprobe configs** afin d'exécuter la commande avec les droits de root. Nous constatons que le fichier *config.gz* a bien été crée via la commande **ls**.

b)

Ensuite, nous copions depuis le RaspberryPi cette configuration, qui se trouve dans le dossier */proc/config.gz*, sur notre machine dans le répertoire courant, sous le dossier linux, grâce à la commande **scp**. Nous tapons donc **scp -p 1030 pi@194.169.0.24 :/proc/config.gz**.

c)

Nous utilisons la commande **zcat linux/config.gz > .config** pour décompresser et enregistrer cette configuration dans un fichier qui se nomme *.config*.

Question 3.

Nous définissons les deux variables d'environnements, à savoir, l'architecture (pointer sur le bon dossier) et les outils de compilation afin de générer et compiler un code binaire qui fonctionne pour l'architecture ARM. Ces variables d'environnement vont nous permettre d'avoir le Kernel Features. Le cross compile contient le préfixe des outils de compilation pour l'architecture ARM. Nous exportons l'architecture pour signifier au Makefile de pointer sur le dossier ARM du dossier ARCH afin d'utiliser la bonne architecture et avoir une bonne compilation.

a)

Nous allons donc maintenant chercher le modèle préemptif afin de modifier notre kernel car, pour l'instant, c'est l'option *voluntary preemption* qui

est activée. Par conséquent, il faut supprimer cette option pour que l'option *preempt full* soit écrite à la place. Pour cela, nous avons deux choix. Soit nous le faisons manuellement, soit nous utilisons des outils adaptés. Nous choisissons d'utiliser les outils adaptés car il se peut qu'en activant l'option *FULL*, d'autres options et sous options s'activent. Le faire manuellement aurait été IMPOSSIBLE vu notre niveau car cela demande des connaissances très pointues de ces dernières. L'outils adapté qui va pouvoir nous faciliter cette tâche est le *menu config* accessible par la commande **make menuconfig**.

b)

En allant dans *Kernel Features* → *Preemption Model (Desktop)*, on constate bien que le voluntary preemption est activé. Nous devons donc activer le modèle temps réel. Nous sélectionnons le modèle *Fully Preemptible Kernel (RT)*.

c)

Après avoir activé notre modèle, nous nous assurons que le timer haute résolution (High Resolution Timer Support → timer software, horloge dynamique présente sur tous les kernel depuis la version 2.6, pour plus de précision) est activé.

Build the new kernel and modules

Dans cette partie, nous compilons le nouveau kernel et les modules associés.

Question 1.

*Grâce à la commande **cat /proc/cpuinfo**, nous allons chercher le nombre de coeurs du processeur (CPU). N'avons qu'un seul coeur. L'installation nous a, donc, pris du temps. La première étape est de construire le kernel.*

a)

La compilation du kernel nous donne une image compressée. Pour compiler le kernel, on utilise l'option zImage. **make zImage**.

b)

Pour compiler les modules et les mises à jour, on utilise l'option modules. **make modules**.

c)

Afin de gérer les chargements de modules, on installe le *device tree* via l'option dtbs. **make dtbs**.

d)

Enfin, les modules compilés précédemment doivent être installés dans $INSTALL_MOD_PATH, on utilise pour cela l'option modules_install. **make modules_install**.

Question 2.

La deuxième étape est de créer l'image du kernel. Pour cela, nous créons un fichier boot.
On commence par créer un dossier *boot/* grâce à la commande **mkdir**.
Nous copions l'image du kernel et du *Divice Tree* dans le fichier *boot*.
Pour terminer, nous téléchargeons et envoyons le nouveau kernel ainsi que les nouveaux modules dans le RaspberryPi via l'image compressée de la question 4.1 dans le temporary directory du RaspberryPi.

Question 3.

Le tout a bien été envoyé sur le RasberryPi. Dans cette sous partie, il va être question de décompresser le *kernel.tgz*. Pour cela, nous allons travailler exclusivement sur le RasberryPi. Nous utiliserons pour cela l'outil *tar* qui permettra la concaténation au sein d'une seule et même archive du *kernel.tgz*, ainsi que la commande **xzf** pour *eXtract*, *gZip* et *File*.
Pour copier le contenu dans le répertoire boot, on utilise la commande **cp -rd /tmp/boot/ /boot**. */tmp/boot/* représente la source et */boot* la destination dans laquelle il faut copier l'image. De même pour **cp -rd /tmp/lib/ /lib**.

Question 4.

On veut mettre hors service le Low Latency Mode (LLM). Pour y arriver nous allons lire le fichier *cmdline.txt* situe´ dans le boot pour y ajouter à la fin manuellement le *sdhci_bcm2708.enable_llm=0*.

Question 5.

On rebootons la carte SD via la commande **sudo reboot**.