S. Lavirotte
Polytech Nice Sophia
SI4
2020-2021

TD n° 04
Compilation Croisée

Le but de ce TD est de faire une première synthèse sur la création d'un système embarqué en incluant (presque) toutes les phases nécessaires à l'obtention d'un système fonctionnant sur une architecture différente de celle de votre station de travail et de créer un système embarqué avec un système d'exploitation qui soit fonctionnel.

1 Configuration de votre VM de travail

1.1 Aménagement mémoire de la VM de travail

Lors de la compilation des sources pour produire ce compilateur, il est nécessaire de disposer d'une quantité de mémoire d'au moins 2Go pour éviter les problèmes de swap ou d'erreur car votre système ne dispose pas d'assez de mémoire. Disposer de suffisamment de mémoire permettra aussi d'accélérer cette étape de compilation qui est assez longue (30 minutes environ sur une machine avec 6 cœurs `i7 4800MQ` utilisés pour la compilation, sans compter les temps de téléchargement). Il est donc nécessaire d'augmenter les capacités de votre machine virtuelle de travail pour avoir au moins 2Go.

1.2 Ajout d'un nouveau disque dur contenant les sources du cross-compilateur

Récupérer l'image disque suivante à ajouter à votre machine virtuelle de travail :

```
http://trolen.polytech.unice.fr/cours/isle/td04/sdd-cross_compiler.7z
```

Cette image intègre les sources du cross-compilateur avec lequel nous allons travailler ainsi que les paquetages des sources nécessaires (afin d'éviter des temps de téléchargement trop longs durant la séance de TD).

Il est nécessaire de monter ce nouveau système de fichier à votre système de fichier racine de votre machine de travail.

Il est impératif de monter ce disque dur dans le dossier `/work/td04`, si vous voulez suivre les instructions données dans ce sujet (d'autres chemins dépendent de celui-ci).

2 Construire une chaîne de cross-compilation

La construction d'une chaîne de compilation croisée est une étape un peu longue et très importante. En effet, tout programme sera généré à l'aide de ce compilateur. Donc plus il est optimisé et adapté à votre plateforme cible, plus votre système final sera efficace. Notre but aujourd'hui est de fabriquer cet environnement de compilation croisée afin d'obtenir un système embarqué avec un OS qui soit fonctionnel sur une autre cible que celle sur laquelle nous travaillons.

2.1 Quel système embarqué ?

Afin de construire une chaîne de compilation croisée, il faut déterminer le type de matériel pour lequel nous souhaitons mettre en place cet outil. Nous allons travailler avec `qemu` pour émuler un système ARM. Vous pourrez voir l'ensemble des plates-formes de type `arm` que `qemu` est capable d'émuler à l'aide de la commande :

```
qemu-system-arm -machine help
```

Nous allons travailler avec la plate-forme émulée `versatilepb` à savoir :

```
versatilepb    ARM Versatile/PB (ARM926EJ-S)
```

Nous configurerons donc l'ensemble des outils pour les caractéristiques de cette machine qui sont les suivantes :

- `ARM926E`, `ARM1136` or `Cortex-A8` CPU
- `PL190` Vectored Interrupt Controller
- Four `PL011` UARTs
- `SMC 91c111` Ethernet adapter
- `PL110` LCD controller

- `PL050 KMI` with PS/2 keyboard and mouse.
- PCI host bridge.
- PCI OHCI USB controller.
- `LSI53C895A` PCI SCSI Host Bus Adapter with hard disk and CD-ROM devices.
- `PL181` MultiMedia Card Interface with SD card.

2.2 Configuration du système

Nous allons utiliser `crosstool-ng` comme environnement permettant de produire l'ensemble des outils nécessaires pour faire de la compilation croisée. Plusieurs éléments sont nécessaires sur votre machine de travail pour produire le cross-compilateur. Vous veillerez donc à installer les paquetages suivants (la plupart d'entre eux sont déjà présents sur votre machine de travail) :

```
apt-get install automake build-essential bison bzip2 curl flex gperf gawk help2man
libexpat1-dev libncurses5-dev libtool libtool-bin ncurses-dev patch python-dev
texinfo wget
```

Attention : `crosstool-ng` impose que la compilation soit effectuée en tant qu'utilisateur standard (et non super-utilisateur). Il est donc nécessaire, à partir de maintenant, de disposer d'un terminal en tant que user (commande su).

```
chmod a+w /work/td04
su user
```

2.3 Préparation de votre environnement pour la compilation

Les sources de `crosstool-ng`, ont déjà été téléchargées et décompressées dans le disque virtuel que vous avez récupéré. Il vous reste toutefois à les configure et lancer la compilation et l'installation du programme utilitaire `ct-ng`. Cette étape produit les outils nécessaires à la configuration (dont la commande `ct-ng`) de la chaîne de compilation croisée, mais sans produire le cross-compilateur lui-même.

```
./configure -prefix=/work/td04/ct-ng
make install
```

2.4 Configuration de la chaîne de compilation croisée souhaitée

Rappel : `crosstool-ng` impose que la compilation soit effectuée en tant qu'utilisateur standard (et non super-utilisateur). Il est donc nécessaire par la suite de disposer d'un terminal en tant que user (commande su).

Une fois `ct-ng` en place, il est nécessaire de l'ajouter au chemin de recherche des exécutables disponibles sur votre système :

```
export PATH=$PATH:/work/td04/ct-ng/bin
```

La commande `ct-ng` accepte de nombreuses options :

```
ct-ng help
ct-ng list-samples
ct-ng show-arm-unknown-linux-uclibcgnueabi
```

La première commande permet d'avoir de l'aide générique sur les commandes possible de `ct-ng`, la deuxième permet d'avoir la liste des exemples prédéfinis de cross-compilateur et la troisième permet de connaître les informations précises sur une configuration donnée.

Nous allons créer un dossier dans `td04` pour contenir la configuration particulière du cross compilateur que l'on va produire.

```
mkdir /work/td04/arm-uclibc
cd /work/td04/arm-uclibc
```

TD n° 04
Compilation Croisée

```
ct-ng arm-unknown-linux-uclibcgnueabi
ct-ng menuconfig
```

Vous noterez que nous créons un **cross-compilateur pour ARM qui fonctionne avec la bibliothèque C µClibC** (Rubrique C library lors de la configuration).

Changer les options suivantes à cette configuration de base :

```
Path et misc options:
    Local tarballs directory /work/td04/src
    Prefix directory /work/td04/x-tools/${CT_TARGET}
Operating System
    Linux kernel 4.14.99
C Compiler
    gcc version 8.3.0
```

Pour information, une chaîne de cross-compilation n'est pas relocalisable (on ne peut pas déplacer les fichiers après la compilation).

2.5 Compilation de la chaîne de compilation croisée

Il ne vous reste plus qu'à lancer la compilation grâce à la simple commande :

```
ct-ng build
```

Cette commande intègre la gestion de la parallélisation des compilations donc il n'est pas besoin de lui passer un paramètre du type `-j` *Number of jobs*.

Suivant la vitesse de votre machine (et de votre connexion Internet si vous devez télécharger les paquetages nécessaires au fur et à mesure des besoins), cette opération pourra prendre entre `20` et `60` minutes. Au fur et à mesure de la compilation, vous pourrez regarder ce qui est créé dans le répertoire `/work/td04/x-tools`.

Pendant que votre chaîne de compilation croisée se compile, terminez le TD précédent si ce n'est pas le cas. Sinon, une pause-café s'impose.

3 Utilisation de la chaîne de compilation croisée

Nous disposons donc maintenant d'une chaîne de cross-compilation pour produire des exécutables pour une cible ARM, ces programmes utilisent la µClibC (et non la librairie C de GNU, la glibC). Nous allons utiliser celle-ci pour produire un système complet fonctionnant sur une plate-forme ARM.

3.1 Utilisation d'un noyau pour tester

Vous disposez dans le dossier `/work/td04` d'un fichier contenant un noyau linux compilé pour ARM. Vous pourrez ainsi lancer `qemu`, lui faire émuler une architecture ARM et avoir un début de démarrage de la machine.

```
qemu-system-arm –M versatilepb -cpu arm1176 –m 32 -dtb versatile-pb.dtb –kernel
vmlinuz-arm-4.14.79
```

Après les messages d'initialisation du noyau, vous obtiendrez bien entendu une erreur vous indiquant qu'il n'y a pas de système de fichier racine (nous ne lui en avons pas fourni !). Nous allons donc en réaliser un, comme nous avons pu le faire lors du TD précédent.

3.2 Compilation de BusyBox pour ARM

Le but de ce TD est de faire un système de fichier pour la cible avec la contrainte supplémentaire de produire un système de fichier avec des programmes pour architecture ARM à l'aide de la chaîne de cross-compilation que nous venons de mettre en place.

TD n° 04
Compilation Croisée

N'oubliez pas de positionner les variables d'environnement du Shell dans lequel vous lancerez la cross-compilation pour ARM (modification des variables `ARCH` et `CROSS_COMPILE` de votre Shell ou passage en paramètre à `make`).

```
export ARCH=…
export CROSS_COMPILE=…
export PATH=$PATH:…
```

Si vous rencontrez des erreurs à la compilation de `busybox`, à vous d'identifier d'où vient le problème et de proposer une solution.

Vous vérifierez grâce à la commande `file` que l'exécutable `busybox` que vous avez créé est bien généré pour la bonne architecture et avec les bonnes propriétés (format ELF, linké statiquement par exemple).

3.3 Création d'un système de fichiers pour la cible

Nous allons modifier le système de fichier `root.img` que vous avez obtenu lors du TD précédent. Nous allons donc en faire une copie sous le nom `rootarm.img`.

Une fois votre système de fichier rempli avec le résultat de la compilation de `BusyBox` pour votre nouvelle cible embarquée ARM, vous pourrez tester que votre système est fonctionnel grâce à `qemu` à l'aide d'une ligne de commande du type :

```
qemu-system-arm –M versatilepb -cpu arm1176 –m 32 -dtb versatile-pb.dtb –kernel
vmlinuz-arm-4.14.79 -drive format=raw,file=rootarm.img -append "root=/dev/sda
console=tty1"
```

Vous devez vérifier que votre cible ARM fonctionne bien avec ce système.

Vous pourrez utiliser le script `myqemu-arm-net.sh` pour activer le réseau et donc tester que le serveur.

```
qemu-system-arm –M versatilepb -cpu arm1176 –m 32 -dtb versatile-pb.dtb –kernel
vmlinuz-arm-4.14.79 -drive format=raw,file=rootarm.img -append "root=/dev/sda
console=tty1" –net nic,model=smc91c111 -net user,id=mynet0,net=192.168.10.0/24,
dhcpstart=192.168.10.10,hostfwd=tcp::5555-:80
```

Si vous contactez bien le serveur Web de votre cible embarquée, bravo ! Vous venez de construire votre premier système embarqué pour une cible ARM. Vous avez gagné le droit de bientôt travailler avec du vrai matériel.

3.4 Un programme Hello World pour la cible

Depuis votre machine de travail, créez un programme « Hello World ! »[1] que vous compilerez pour `x86_64` et pour ARM avec chargement dynamique des librairies et en tant qu'exécutable statique (vous obtiendrez donc `4` exécutables). Vous veillerez à l'ajouter à votre système ce programme « Hello World » pour vérifier qu'il fonctionne bien (ou pas) sur la cible (vous pourrez le vérifier pour vos cibles `x86_64` et ARM). Expliquez pourquoi certains exécutables fonctionnent et d'autres pas. Justifiez vos réponses.

Notez aussi les tailles de chacun des fichiers programmes obtenus pour les différentes configurations et réfléchissez à ce que vous avez réalisé pour répondre aux questions d'une prochaine évaluation…

[1] Pour ceux qui ne sauraient plus réaliser un « Hello World ! » en C, vous pourrez consulter l'adresse suivante : https://fr.wikipedia.org/wiki/Liste_de_programmes_Hello_world