Linux Embarqué

Travaux Pratiques

Prérequis

1. Désarchivage de linux_embedded.tar.bz2 dans le répertoire home
```
$ cd $HOME
$ tar jxvf /media/usb/linux_embedded.tar.bz2
```

Crosstool-NG

But : compiler une chaine de compilation croisée pour une cible autre que la machine hôte

1. Désarchivage du tarball :
```
$ cd $HOME/linux_embedded
$ tar jxvf tarballs/crosstool-ng-1.8.2.tar.bz2
```

2. Installation :
```
$ cd crosstool-ng-1.8.2
$ ./configure --prefix=$HOME/linux_embedded/crosstool-ng
$ make
$ make install
$ export PATH=$HOME/linux_embedded/crosstool-ng/bin:$PATH
$ cd ..
$ rm -rf crosstool-ng-1.8.2
```

3. Création d'un répertoire de travail :
```
$ mkdir crosstool-ng-build
$ cd crosstool-ng-build
```

4. Configuration :
```
$ cp ../configs/config_crosstool-ng .config
$ ct-ng menuconfig
```

5. Compilation :
```
$ ct-ng build
```

6. La chaine est installée dans $HOME/linux_embedded/x-tools :
```
$ export PATH=$HOME/linux_embedded/x-tools/arm-unknown-linux-
uclibcgnueabi/bin:$PATH
$ arm-unknown-linux-uclibcgnueabi-gcc -v
$ cd ..
```

7. Compilation du fichier hello.c :
```
$ arm-unknown-linux-uclibcgnueabi-gcc -static tarballs/hello.c -o hello
$ file hello
```

Conclusion : le fichier est bien compilé pour une machine ARM

Compilation d'un noyau avec la chaine croisée

But : configurer et compiler un noyau pour une cible ARM, démarrer le noyau dans l'émulateur

1. Désarchivage du tarball :
```
$ cd $HOME/linux_embedded
$ tar jxvf tarballs/linux-2.6.34.tar.bz2
```

2. Configuration du noyau :
```
$ cd linux-2.6.34
$ cp ../configs/config_linux26 .config
$ make ARCH=arm menuconfig
```

3. Compilation du noyau :
```
$ make -j2 ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
```
Note : remplacer -j2 par le nombre de CPU + 1
Note : on peut exporter les variables d'environnement ARCH et CROSS_COMPILE

4. Test du noyau avec QEmu :
```
$ qemu-system-arm -M versatilepb -m 16M -kernel arch/arm/boot/zImage -
append "clocksource=pit"
```

Conclusion : le noyau démarre mais produit un "kernel panic". Pourquoi ?

Création d'un initrd

But : Faire démarrer le noyau avec un fichier initrd

1. Création d'un répertoire qui contiendra les fichiers de l'initrd :
```
$ cd $HOME/linux_embedded
$ mkdir rootfs
```

2. Copie du fichier hello compilé plus haut, en le renommant init :
```
$ cp hello rootfs/init
```

3. Création d'une archive cpio compressée :
```
$ cd rootfs
$ find . | cpio -o -H newc | gzip > ../rootfs.cpio.gz
$ cd ..
```

4. Test du noyau avec QEmu, en précisant le chargement d'un fichier initrd :
```
$ qemu-system-arm -M versatilepb -m 16M -kernel linux-
2.6.34/arch/arm/boot/zImage -append "clocksource=pit" -initrd
rootfs.cpio.gz
```

Conclusion : le noyau démarre et exécute le fichier init

Création d'un initramfs avec une archive cpio

But : utiliser l'archive cpio créée précédemment comme initramfs

1. Configuration du noyau pour utiliser l'archive cpio :

```
$ cd $HOME/linux_embedded/linux-2.6.34
$ make ARCH=arm menuconfig
      General setup  --->
      [*] Initial RAM filesystem and RAM disk
      ($(HOME)/linux_embedded/rootfs.cpio.gz) Initramfs source file(s)
      Built-in initramfs compression mode (Gzip)
```

2. Compilation du noyau :

```
$ make -j2 ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
```

3. Test du noyau avec Qemu, sans préciser de charger un initrd :

```
$ qemu-system-arm -M versatilepb -m 16M -kernel linux-2.6.34/arch/arm/boot/zImage
-append "clocksource=pit"
```

Conclusion : Le noyau démarre mais affiche cette erreur :

```
Warning: unable to open an initial console.
```

Pourquoi ?
La réponse est dans le fichier `scripts/gen_initramfs_list.sh`, dans la fonction default_initramfs()

4. Ajout du device /dev/console

```
$ cd ../rootfs
$ sudo mkdir dev
$ sudo mknod dev/console c 5 1
$ find . | cpio -o -H newc | gzip > ../rootfs.cpio.gz
$ cd ..
```

S'il n'est pas possible d'utiliser su ou sudo, on pourra utiliser fakeroot pour créer les devices et l'archive cpio :

```
$ cd ../rootfs
$ fakeroot
# mknod dev/console c 5 1
# find . | cpio -o -H newc | gzip > ../rootfs.cpio.gz
# exit
```

5. Recompiler le noyau, retester avec QEmu.

Conclusion : le noyau démarre et exécute le fichier init

Création d'un initramfs avec un fichier de commandes

But : utiliser un fichier de commande pour créer l'initramfs

1. Edition d'un fichier de commande pour créer un initramfs :

```
$ cd $HOME/linux_embedded
$ nedit initramfs.txt
      dir /dev 0755 0 0
      nod /dev/console 0600 0 0 c 5 1
      dir /root 0700 0 0
      file /init ../rootfs/init 0755 0 0
```

2. Configuration du noyau :

```
$ cd $HOME/linux_embedded/linux-2.6.34
$ make ARCH=arm menuconfig
      General setup  --->
      [*] Initial RAM filesystem and RAM disk
      ($(HOME)/linux_embedded/initramfs.txt) Initramfs source
      Built-in initramfs compression mode (Gzip)
```

3. Compilation du noyau :

```
$ make -j2 ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
```

4. Test du noyau avec QEmu :

```
$ qemu-system-arm -M versatilepb -m 16M -kernel arch/arm/boot/zImage -append
"clocksource=pit"
```

Conclusion : Le noyau démarre et lance le processus init

Busybox

But : installer Busybox dans l'initramfs

1. Désarchivage du tarball :
```
$ cd $HOME/linux_embedded
$ tar jxvf tarballs/busybox-1.16.1.tar.bz2
$ cd busybox-1.16.1
```

2. Configuration de Busybox :
```
$ cp ../configs/config_busybox .config
$ make menuconfig
```

3. Compilation et installation :
```
$ make ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
CONFIG_PREFIX=$HOME/linux_embedded/rootfs install
```

4. Si la compilation de Busybox n'étant pas statique, copier les librairies manquantes :
```
$ cd $HOME/linux_embedded/rootfs
$ arm-unknown-linux-uclibcgnueabi-ldd --root . bin/busybox
libcrypt.so.0 not found
libc.so.0 not found
ld-uClibc.so.0 not found
$ mkdir lib
$ cp ~/linux_embedded/x-tools/arm-unknown-linux-uclibcgnueabi/arm-unknown-
linux-uclibcgnueabi/lib/libc.so.0 lib
...
```

5. Création du lien /init
```
$ ln -s bin/busybox init
```

6. Création du répertoire /dev
```
$ mkdir dev
$ sudo MAKEDEV -v -d dev generic console
ou
$ cd dev; sudo MAKEDEV -v generic console
```
On pourra aussi utiliser fakeroot mais il faudra créer une archive cpio sous le shell fakeroot.

7. Configuration du noyau :
```
$ cd $HOME/linux_embedded/linux-2.6.34
$ make ARCH=arm menuconfig
      General setup  --->
      [*] Initial RAM filesystem and RAM disk
      ($(HOME)/linux_embedded/rootfs/) Initramfs source file(s)
      Built-in initramfs compression mode (Gzip)
```

6. Compilation du noyau :
```
$ make -j2 ARCH=arm CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
```

7. Test du noyau avec QEmu :
```
$ qemu-system-arm -M versatilepb -m 16M -kernel arch/arm/boot/zImage -append
"clocksource=pit"
```

Conclusion : Le noyau démarre et lance le processus Busybox, nous donnant un shell.

Exercices optionnels :
- ne créer que les devices strictement nécessaires
- utiliser mdev pour créer dynamiquement les devices
- modifier le rootfs pour qu'il monte automatiquement /proc et /sys

U-Boot

But : utiliser U-Boot pour charger le noyau

1. Désarchivage du tarball :
```
$ cd $HOME/linux_embedded
$ tar jxvf tarballs/u-boot-2010.03.tar.bz2
$ cd u-boot-2010.03
```

2. Application du patch pour fonctionnement dans Qemu :
```
$ patch -p1 < ../tarballs/u-boot-2010.03.arm_versatile.patch
```

3. Compilation de U-Boot :
```
$ make versatile_config
$ make CROSS_COMPILE=arm-unknown-linux-uclibcgnueabi-
```

4. Test avec QEmu :
```
$ qemu-system-arm -M versatilepb -m 64 -kernel u-boot.bin -nographic
```

5. Création d'un fichier simulant une mémoire flash, comprenant le noyau :
```
$ touch dummy.txt
$ echo dummy.txt | cpio -o -H newc | gzip > dummy.cpio.g
$ tools/mkimage -A arm -C none -O linux -T ramdisk -d dummy.cpio.gz -a
0x00800000 -e 0x00800000 dummy.uimg
$ tools/mkimage -A arm -C none -O linux -T kernel -d ../linux-
2.6.34/arch/arm/boot/zImage -a 0x00010000 -e 0x00010000 kernel.uimg
$ dd if=/dev/zero of=flash.bin bs=1 count=6M
$ dd if=u-boot.bin of=flash.bin conv=notrunc bs=1
$ dd if=kernel.uimg of=flash.bin conv=notrunc bs=1 seek=2M
$ dd if=dummy.uimg of=flash.bin conv=notrunc bs=1 seek=4M
```

6. Test de l'image avec QEmu :
```
$ qemu-system-arm -M versatilepb -m 128M -kernel flash.bin -nographic
```
et
```
$ qemu-system-arm -M versatilepb -m 128M -kernel flash.bin
```

Conclusion : Nous pouvons démarrer U-Boot qui charge ensuite le noyau.

Buildroot

But : Construction d'une image flash avec buildroot

1. Désarchivage du tarball :
```
$ cd $HOME/linux_embedded
$ tar jxvf tarballs/buildroot-2010.05.tar.bz2
$ cd buildroot-2010.05
```

2. Configuration de Buildroot :
```
$ cp ../configs/config_buildroot .config
$ make menuconfig
```

3. Compilation :
```
$ make
…
$ ls -l output/images
```

4. Création de l'image flash :
```
$ output/build/u-boot-2010.03/tools/mkimage -A arm -C none -O linux -T
kernel -d output/images/zImage -a 0x00010000 -e 0x00010000 kernel.uimg
$ output/build/u-boot-2010.03/tools/mkimage -A arm -C none -O linux -T
ramdisk -d output/images/rootfs.cpio.gz -a 0x00800000 -e 0x00800000
rootfs.uimg
$ dd if=output/images/u-boot.bin of=flash.bin conv=notrunc bs=1
$ dd if=kernel.uimg of=flash.bin conv=notrunc bs=1 seek=2M
$ dd if=rootfs.uimg of=flash.bin conv=notrunc bs=1 seek=4M
```

5. Test de l'image avec QEmu :
```
$ qemu-system-arm -M versatilepb -m 128M -kernel flash.bin
```

Conclusion : Nous pouvons démarrer U-Boot, charger le noyau, qui monte le système rootfs.