Informatique embarquée : TP n°3

Temps estimé : ~ 3 heures.

Temps passé : ~ 15 heures.

Configuration :

Dans ***General setup,*** j'ai activé l'option Initial RAM filesystem and RAM disk, puis j'ai activé le support de *printk* dans le sous-menu ***Configure standard kernel features*** qui est un sous menu de ***General Setup***. Dans ***Executable file format / Emulation***, j'ai activé le support des fichier ELF et des shell scripts. Puis dans le chemin ***Device Driver***, ***Character Devices***, j'ai activé le TTY, ainsi que le support de *8250/16550* dans le sous-menu ***Serial Drivers*** puis de la console associée.

Problème soulevé :

La version 4.7.7 du noyau m'a posé des problèmes à l'exécution de Qemu. J'ai donc dû utiliser la dernière version du noyau. J'ai également dû enlever l'option *O*=…

Documentation :

La documentation est situé dans le répertoire suivant :

```
./linux-4.13.10/Documentation
```

Compilation :

J'ai compilé en utilisant la commande suivante :

```
make -j 4
```

La compilation a été effectuée en 3 minutes.

Fichier généré :

Le fichier généré est le suivant :

```
./arch/i386/boot/bzImage
```

C'est une version compressé du noyau Linux. Il existe également une autre version du noyau qui est la suivante :

```
./vmlinux
```

Si on avait généré le noyau pour notre machine de développement, il faudrait ensuite installer le noyau de la manière suivante :

```
make modules            // Compiler et installer les modules pour le noyau
make install_modules
cp ./arch/i386/boot/bzImage /boot/vmlinux-4.13.10

// System.map est un fichier contenant la table des symboles.
// Elle est utilisée par le noyau. Il faut donc la copier dans /boot
cp System.map /boot
```

En supposant qu'on utilise *LILO* comme chargeur d'amorçage, on doit éditer le fichier */etc/lilo.conf :*

```
image= /boot/vmlinux-4.13.10
  label = ''Linux 4.13.10''
```

Puis on modifie le zone d'amorçage du disque :

```
lilo -v
```

Enfin, on peut redémarrer la machine.

Source : *http://www.berkes.ca/guides/linux_kernel.html*

Qemu :

Après avoir compilé le noyau j'ai lancé *qemu-system-i386* de la manière suivante :

```
qemu-system-i386 -kernel arch/i386/boot/bzImage -nographic -append "console=ttyS0"
```

J'ai eu le message d'erreur suivant :

```
Kernel panic - not syncing: No working init found.  Try passing init= option to
kernel. See Linux Documentation/admin-guide/init.rst for guidance.
Kernel Offset: disabled
---[ end Kernel panic - not syncing: No working init found.  Try passing init=
option to kernel. See Linux Documentation/admin-guide/init.rst for guidance.
random: crng init done
```

Cela est normal car je n'ai pas défini le programme *init*.

Qemu-i386 et Qemu-system-i386 :

Il y a une différence fondamentale entre *qemu-i386* et *qemu-system-i386*. *qemu-i386* émule un programme comme s'il tournait sur une architecture donnée (en l'occurrence *i386*). Tandis que *qemu-system-i386* émule un système d'exploitation. De ce fait, si on utilise *qemu-system-i386* sur *hello*, il sera lancé comme un système d'exploitation.

Utiliser hello comme programme *init* :

Comme on veut utiliser le programme *hello* comme *init*, on doit générer le fichier *initramfs.cpio.gz* en utilisant la commande suivante :

```
find . -print0 | cpio --null -ov --format=newc | gzip -9 > ../initramfs.cpio.gz
```

On doit générer une archive car lors de l'exécution du noyau, Qemu va devoir chercher dans l'arborescence de fichier (en l'occurrence *root*) le programme *init* pour le lancer. Cette arborescence doit être dans une archive, afin de limiter sa taille dans la mémoire en moment où elle sera chargée.

En utilisant la commande *file* sur *init*/*hello*, on a l'affichage suivant :

```
root/sbin/init: ELF 32-bit LSB executable, Intel 80386, version 1 (GNU/Linux),
statically linked, for GNU/Linux 2.6.32,
BuildID[sha1]=49ca71d3d4625bc53c4d2d5b1a563bf7175d979f, not stripped
```

Ici, *init/hello* doit être généré en utilisant les liens statiques, car lorsqu'il sera exécuté en tant que processus *init*, si le lien est dynamique, alors il devra charger les bibliothèques dans l'arborescence de fichier, alors même que le système de fichiers n'est pas encore chargé (on est encore dans le « boot loader » au moment où on lance *init*).

Le programe init/hello fait 712 kio. J'ai calculé la taille en utilisant la commande *du*.

La taille des segments de code et de données, est de 661305 octets pour les segments de code, 4092 octets pour les segments de données.

A propos de la commande qemu-system-i386 :

qemu-system-i386 -kernel arch/i386/boot/bzImage -nographic -append "console=ttyS0" -initrd ../initramfs.cpio.gz

L'option *-append* permet d'exécuter la commande *"console=ttyS0"*. Si on enlève cette option, alors *"console=ttyS0"* sera perçu comme une option à fournir à Qemu.

Lorsque mon programme init se termine après avoir affiché « Hello World ! », il y a un *kernel panic*.

Un peu de dynamisme :

J'ai recompilé le programme *hello* sans l'édition statique de lien. Puis j'ai récupéré la liste des fichiers dynamiques auxquels était lié hello. Je les ai copiés dans l'arborescence *root/*, en l'occurrence :

```
/lib/ld-linux.so.2
/lib32/libc.so.6
```

Ces bibliothèques sont bien celle indiquées par la commande ldd :

```
linux-gate.so.1 (0xf76e1000)                    // Pas besoin de le mettre
libc.so.6 => /lib32/libc.so.6 (0xf7502000)
/lib/ld-linux.so.2 (0xf76e3000)
```

Le programme *hello a une taille de 8 kio.*

La taille des segments de code et de données, est de 1664 octets pour les segments de code, 308 octets pour les segments de données.

Compilation croisée :

J'ai généré le programme *hello_arm*, qui est l'exécutable du programme *hello* pour l'architecture ARM, comme on peut le voir avec le résultat de la commande file ./hello_arm :

```
hello_arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, for GNU/Linux 3.2.0, BuildID[sha1]=6f2048bc09f465c62f5dfce198d3a68e168b53c3, not stripped
```

Içi, il faut utiliser l'option *-static* car dans le cas contraire, à l'exécution, *hello_arm* va chercher la bibliothèque dynamique pour l'architecture ARM qui va lui permettre d'appeler les fonctions nécessaires à son exécution. Comme cette bibliothèque n'existe pas (ma machine étant en x86-64), une erreur au chargement du programme va se produire.

Note : Comme j'ai déjà Qemu sur ma machine, je n'ai pas pu voir la différence entre *./hello_arm* (dans l'hypothèse où Qemu n'est pas installé) et *./hello_arm* avec Qemu installé.

Toutefois, j'étais étonné par le fait que la commande :

```
./hello_arm
```

puisse fonctionner alors que l'exécutable ait été généré pour une exécution sur une architecture ARM. Cela est probablement dû au fait que *qemu-arm* soit lancé au moment où je lance *./hello_arm.*

BusyBox :

Un premier problème que j'ai rencontré est que Busybox ne compilait pas. Il ne trouvait pas *asm/errno.h.* Cette erreur est lié au fait que dans certaines distributions, le répertoire */usr/include/asm* soit renommé */usr/include/asm-generic*. J'ai donc d^crée un lien symbolique vers */usr/include/asm.*

J'ai pourtant eu un deuxième problème, qui était que le fichier d'entête *asm/byteorder.h* manquait. Après avoir fouillé dans les méandres de l'arborescence, j'ai constaté qu'il n'était effectivement pas présent dans le répertoire cité plus haut. Cependant il était présent dans le répertoire

/usr/arm-linux-gnueabi/include/asm/. J'ai donc dû refaire un lien symbolique vers ce répertoire au lieu de */usr/include/asm-generic*.

Après avoir compilé, installé Busybox, et mis à jour le répertoire */root/* j'ai relancé Qemu avec les même paramètres qu'avant. Puis j'ai rencontré un problème qui a été abordé sur le forum Moodle : je n'avais pas crée le répertoire */dev/* avec les fichiers *TTY*.

J'ai dû corriger le tout. J'aurai aussi pu créer le fichier */dev/null* mais cela n'était pas nécessaire.

Le fichier a une taille de 1,9 Mio. Il est sans doute possible de le configurer pour qu'il soit plus léger que Bash, mais je n'ai pas vérifier.