Laboratoire 2
Compilation d'un système Linux minimal et introduction à U-Boot

ELE4205 - Département de génie électrique
Polytechnique Montréal

25 juillet 2018

Table des matières

1 Introduction

Le but de ce laboratoire est de vous familiariser avec les démarches à suivre pour construire un système minimal Linux. Nous verrons également comment déverminer un démarrage d'un système d'exploitation. Vous allez donc construire un système minimal pour Odroid-C2 avec un programme de démarrage. Vous avec déjà utilisé le projet Yocto pour construire le même système, mais complet, de façon « automatique ». Vous serez à même de constater que l'utilisation de Yocto facilite grandement le processus. De plus, pour une nouvelle « architecture » Yocto permet d'utiliser un canevas de départ que l'on pourra adapter puis raffiner afin de rendre la plateforme complètement « compatible » avec Yocto. De nombreux fournisseurs de « plateformes » fournissent des BSP (*Board Specification Package*) qui sont des couches Yocto adaptées au matériel permettant ainsi le développement rapide d'une application sur ces systèmes.

Les manipulations qui suivent montrent comment compiler une image de noyau (*kernel*) Linux ainsi qu'un système de fichier minimal.

2 Compilateur croisé

Pour construire un système Linux, il faut utiliser un compilateur compatible avec le Odroid-C2. Il faudrait donc générer un chaîne de compilation croisée complète pour l'architecture cible : une tâche assez longue et complexe que nous ne ferons pas dans le cadre de ce laboratoire. Lors du laboratoire 1 nous avons généré une image pour le Odroid-C2 et une des étapes de cette génération était la production des outils de compilation croisée. Avec l'aide de notre configuration pour le Odroid-C2, nous allons maintenant générer une chaîne de compilation (*toolchain* ou SDK) dans le même répertoire de *build* que lors du laboratoire 1 et l'installer localement dans `/export/tmp/4205_nn/`. Démarrer un terminal.

```
% cd /export/tmp/4205_nn/poky/
% bash
$ source oe-init-build-env build-oc2
$ umask a+rx u+rwx
$ bitbake -c populate_sdk core-image-base
$ sh ./tmp/deploy/sdk/poky-glibc-x86_64-core-image-base-aarch64-toolchain-2.1.3.sh
Poky (Yocto Project Reference Distro) SDK installer version 2.1.3
==========================================================
Enter target directory for SDK (default: /opt/poky/2.1.3): /export/tmp/4205_nn/opt/poky
You are about to install the SDK to "/export/tmp/4205_nn/opt/poky". Proceed[Y/n]?
Extracting SDK.....................................................................done
Setting it up...done
SDK has been successfully set up and is ready to be used.
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
```

```
$ . /export/tmp/4205_nn/opt/poky/environment-setup-aarch64-poky-linux
```

Maintenant, pour avoir un envirionnement de compilation croisée, il faut démarrer un terminal, passer en `bash` et faire un *source* du SDK

```
% cd /export/tmp/4205_nn/
% bash
$ source /export/tmp/4205_nn/opt/poky/environment-setup-aarch64-poky-linux
```

3 Utilisation des outils de U-Boot

Le logiciel U-Boot est le second chargeur de démarrage[1] (*boot*) qui permet de spécifier les points d'entrée du système i.e. l'adresse de l'image du noyau, l'adresse des données en écriture, etc. Nous verrons plus tard comment utiliser ce logiciel à l'aide d'un câble sériel USB.

Normalement, nous devrions compiler les outils de U-Boot pour la gestion du chargement du noyau selon les instructions du site d'Odroid.

Nous allons plutôt utiliser les binaires de la version qui est compatible avec le noyau que nous allons compiler (**Attention** aux points « . » en fin de ligne des commandes `cp`).

```
% cd /export/tmp/4205_nn/
% mkdir linux_scratch
% cd linux_scratch
% mkdir oc2sdcard
% cd oc2sdcard/
% cp ../../poky/build-oc2/tmp/deploy/images/odroid-c2/*b*bin* .
% cp ../../poky/build-oc2/tmp/deploy/images/odroid-c2/boot.ini  .
% ls
bl1.bin.hardkernel  boot.ini  u-boot.bin
```

4 Compilation du *kernel* Linux

Le *kernel* de Linux est le coeur du système d'exploitation : un « logiciel » spécialisé qui fourni, entre autres, une couche d'abstraction du matériel vers le logiciel. Il gère également l'ordonnancement des processus ainsi que de la gestion de la mémoire. Il est donc primordial d'utiliser la bonne version de *kernel* afin de bien cibler le matériel. Voici quelques exemples des tâches du noyau :

1. Le premier chargeur est dans la mémoire ROM du Odroid-C2 et appelle dans ce cas ci U-Boot situé dans la eMMC ou sur la carte microSD.

1. La gestion des processus : lorsque vous regardez la liste des processus avec la commande `top`, on peut penser que les applications sont toutes exécutées en parallèle. En réalité, il s'agit d'un pseudo-parallélisme, c'est-à-dire que le noyau réaffecte fréquemment la priorité d'exécution à toutes les applications. L'ordonnanceur, le service qui gère l'exécution des programmes, doit donc garder en mémoire dans une table l'état du processus, le compteur ordinal, le pointeur de pile, etc.
2. La gestion de la mémoire : une application consiste en trois blocs de mémoire : le code, la pile et les données. Vous avez sans doute programmé une application qui utilisait un index de tableau supérieur à sa taille. L'erreur `SEGFAULT` indique que vous avez tenté d'accéder à une plage mémoire qui n'était pas celle de l'application. En effet, la MMU, i.e. *Memory Management Unit*, se charge de vérifier ce genre d'erreur. Sans cette fonctionnalité, une telle erreur peut causer un blocage global du système !
3. La gestion de périphérique d'E/S (Entrée/Sortie) : Votre disque dur est constitué d'une tête de lecture qui écrit rapidement à des endroits spécifiques. Les disques durs qui adoptent le protocole SATA est monnaie courante aujourd'hui. Du côté logiciel, une écriture/lecture de fichier en C/C++ peut se faire très facilement avec la librairie standard (`glibc` par exemple). Ceci n'est qu'une abstraction offerte par le système d'exploitation car en réalité plusieurs paramètres matériels doivent être tenus en compte.

   Puisque le développeur de logiciels n'est pas intéressé à réinventer la roue, le système d'exploitation offre de nombreux drivers pour garantir la compatibilité avec le matériel. Cette couche d'abstraction rend la programmation très portable d'une architecture vers une autre.

Le *kernel* se charge de beaucoup d'autres tâches et sa complexité va se traduire par un temps de compilation assez long.

Il faut maintenant télécharger les sources du *kernel* Linux. Nous allons utiliser un `git clone` pour cette étape. Nous allons télécharger une révision d'une branche spécifique. Mais avant, il faut télécharger le fichier requis pour la création de la carte microSD et le placer dans le répertoire `/export/tmp/4205_nn/linux_scratch`.

```
% bash
$ cd ..
$ pwd
/export/tmp/4205_nn/linux_scratch
$ cp ../poky/meta-odroid/recipes-kernel/linux/linux-hardkernel-3.14/add_uboot.patch .
$ cp ../poky/meta-odroid/recipes-kernel/linux/linux-hardkernel-3.14/odroid-c2/defconfig .
$ cp ../poky/meta-odroid/recipes-kernel/linux/linux-hardkernel-3.14/0001-compiler-gcc-integrate-the-
```

```
$ ls
0001-compiler-gcc-integrate-the-various-compiler-gcc-345-.patch
add_uboot.patch
defconfig
makesdcardoc2
$ git clone -b odroidc2-3.14.y https://github.com/hardkernel/linux.git
$ cd linux
$ git checkout 6ad167426fbad87ff62af517fc01ad9655a89e18
$ patch -p1 -i ../0001-compiler-gcc-integrate-the-various-compiler-gcc-345-.patch
$ patch -p1 -i ../add_uboot.patch
$ source /export/tmp/4205_nn/opt/poky/environment-setup-aarch64-poky-linux
```

Normalement pour compiler un *kernel*, il faut passer par une étape de configuration assez longue. Un fichier de configuration a été copié du `meta-odroid`.

Mettez ce fichier dans le dossier de configuration des plateformes ARM64 comme suit :

```
$ cp ../defconfig arch/arm64/configs/
```

Vous êtes maintenant presque prêt à compiler le *kernel*. Il y une erreur de `LDFLAGS` dans le `environment-setup-aarch64-poky-linux` qu'il faut corriger.

```
$ export LDFLAGS=" "
```

De plus, il y une erreur de dependance du `Makefile` sur `libgcc.a` que l'on corrige comme suis :

```
$ gedit  arch/arm64/Makefile &
```

et commentant la ligne 51 comme suit

```
#libs-y += $(LIBGCC)
```

Notre environnement est maintenant prêt pour la compilation. La première commande qui suit indique que la architecture ciblée est le `arm64` utilisé par le Odroid-C2 et la deuxième que la chaîne de compilation à utiliser a le prefixe `aarch64-poky-linux-` [2] ce qui a été réalisé par `environment-setup-aarch64-poky-linux`. :

2. Le prefixe s'ajoute au nom des outils natifs de développement, par exemple `gcc` devient `aarch64-poky-linux-gcc`. Le système de prefixe est aussi utilisé pour la configuration des IDE (Integrated Development Environment) pour la compilation croisée par exemple avec Eclipse que nous utiliserons dans un laboratoire subséquent.

```
$ echo $ARCH
arm64
$ echo $CROSS_COMPILE
aarch64-poky-linux-

$ make mrproper
$ make defconfig
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
$ make -j8 Image dtbs
...
...
  GEN     .version
  CHK     include/generated/compile.h
  UPD     include/generated/compile.h
  CC      init/version.o
  LD      init/built-in.o
  KSYM    .tmp_kallsyms1.o
  KSYM    .tmp_kallsyms2.o
  LD      vmlinux
  SORTEX  vmlinux
  SYSMAP  System.map
  OBJCOPY arch/arm64/boot/Image
$
```

La compilation va prendre environ 5 minutes (plusieurs *warnings*). L'image du noyau a 13Mo, il est possible d'utiliser la procedure de configuration pour diminuer la dimension de l'image.

Une fois que la compilation sera terminée, vous trouverez ces fichiers dans ce répertoire :

```
$ ls arch/arm64/boot/
```

```
dts  Image  install.sh  Makefile
$ ls arch/arm64/boot/dts/m*
arch/arm64/boot/dts/meson64_odroidc2.dtb
arch/arm64/boot/dts/meson64_odroidc2.dts
```

Le fichier `Image` contient à la fois l'image du noyau mais également les paramètres de configuration de l'environnement U-Boot tandis que le fichier `.dtb` contient le *device tree*, c'est-à-dire la structure de données qui décrit le matériel (i.e. les processeurs, les mémoires, etc.). Nous allons copier les fichiers requis pour notre image de carte SD :

```
$ cp  arch/arm64/boot/Image ../oc2sdcard/
$ cp  arch/arm64/boot/dts/meson64_odroidc2.dtb ../oc2sdcard/
```

5 Les *boot loaders*

1. `bl1.bin.hardkernel` : Le premier *bootloader* externe. Le matériel va lire ce fichier et va charger le second *bootloader* spécifié par ce fichier.
2. `u-boot.bin` : L'image du second *bootloader* externe ; c'est le programme qui va spécifier le point d'entrée du *kernel*.

Nous verrons plus loin comment utiliser l'environnement U-Boot.

6 Téléchargement de BusyBox et création du système de fichiers

Vous utilisez ce programme depuis que vous avez commencé ce laboratoire. Les systèmes POSIX disposent de commandes standards en ligne de commande tels que `cd`, `ls`, `pwd`, etc. qui sont fournis par une application : BusyBox. En effet, la ligne de commande n'est rien d'autre qu'une application dans laquelle les paramètres d'entrée sont ceux inscrits par l'utilisateur. Faites ces commandes pour constater que l'ordinateur de laboratoire utilise bel et bien `busybox` :

```
$ which busybox
/sbin/busybox
$ /sbin/busybox ls
COPYING          REPORTING-BUGS  include          scripts
CREDITS          System.map      init             security
Documentation    android         ipc              sound
Kbuild           arch            kernel           tools
```

```
Kconfig          block           lib             usr
MAINTAINERS      crypto          linaro          virt
Makefile         drivers         mm              vmlinux
Module.symvers   firmware        net             vmlinux.o
README           fs              samples
$ which busybox ls
/sbin/busybox
/bin/ls
```

Remarquez ceci : `busybox ls` se comporte comme un appel à `ls`. Ce qui permet d'utiliser un seul programme pour plusieurs fonctions (faites un `busybox --help` pour connait ce que `busybox` peut faire comme opération dans votre distribution). On pourra donc exploiter cette fonctionnalité pour la séquence de *boot* que nous allons utiliser plus loin.

Nous devons maintenant créer un système de fichiers de base. Créer un répertoire `rootfs` et créez ces sous-répertoires :

```
$ cd ../oc2sdcard/
$ pwd
/export/tmp/4205_nn/linux_scratch/oc2sdcard
$ umask a+rx u+rwx
$ mkdir rootfs
$ cd rootfs/
$ mkdir bin dev proc sys
```

Obtenez un busybox précompilé compatible avec notre architecture (l'Odroid-C2 est un armv8 dit Aarch64) :

```
wget https://busybox.net/downloads/binaries/1.21.1/busybox-armv7l -O bin/busybox
```

Changez ses permissions pour qu'il puisse être exécuté comme un exécutable :

```
$ chmod +x bin/busybox
```

Créez un lien symbolique `init` qui va être utilisé comme point d'entrée de la séquence de *boot* et le lien pour le *shell* `sh` :

```
$ ln -s busybox bin/init
$ ln -s busybox bin/sh
$ ls bin
busybox  init  sh
```

7 Préparation de la carte microSD

Nous allons créer une image de carte microSD avec le script `makesdcardoc2` téléchargé précédemment qu'il faudra éditer pour tenir compte de votre répertoire de travail en ajustant la ligne suivante :

```
POKYBUILD_LOCATION=/export/tmp/4205_nn/poky/build-oc2/
```

Il faut ensuite éditer le fichier `boot.ini`

```
$ cd ..
$ pwd
/export/tmp/4205_nn/linux_scratch/oc2sdcard
$ gedit boot.ini &
```

pour avoir aux lignes 68 et 73 (ligne 73 en une seule ligne)

```
setenv condev "console=ttyS0,115200n8"

setenv bootargs "root=/dev/mmcblk0p2 rootwait rw ${condev} no_console_suspend
    hdmimode=${m} m_bpp=${m_bpp} vout=${vout} init=/bin/init"
```

Si tout a été fait correctement, vous pouvez créer l'image disque avec

```
$ chmod 755 ../makesdcardoc2
$ ../makesdcardoc2
```

et écrire cette image sur la carte avec (remplacer `/dev/sdc` par le bon *device*)

```
$ dd if=oc2_lab2.sdcard of=/dev/sdc
$ sync && sync
```

Votre carte devrait maintenant être prête pour la séquence de démarrage.

8 Utilisation de `minicom` et séquence de démarrage à l'aide de l'environnement U-Boot

Pour communiquer avec le Odroid-C2 lors de la séquence de *boot*, vous ne pourrez pas utiliser la communication SSH par USB. En effet, la plateforme n'a pas encore chargé le pilote du périphérique USB et par conséquent ce dernier n'est pas disponible.

Il vous faut un câble sériel à USB ; des instructions de connexion vous ont été fournies au laboratoire 1.

Pour utiliser la communication sérielle, nous allons utiliser le logiciel `minicom`. La connexion a déjà été préalablement configurée dans les ordinateurs de laboratoire (`/dev/ttyUSB0`, `baud=115200, bits=8, parity=n, stop bits=1, handshake=none`) [3].

Connectez le câble sériel à l'ordinateur et exécutez `minicom` :

```
minicom
```

Connectez maintenant l'alimentation USB du Odroid-C2 et appuyez sur `Entrée` deux fois lorsque vous verrez que U-Boot a démarré :

```
U-Boot 2015.01-00088-g375645c (Apr 07 2016 - 10:22:59)

DRAM:  2 GiB
Relocation Offset is: 76f41000
------------------------------------------------
* Welcome to Hardkernel's ODROID-C2
------------------------------------------------
CPU : AMLogic S905
S/N : HKC213254DFD0694
MAC : 00:1e:06:33:53:14
BID : HKC2211605
------------------------------------------------
register usb cfg[1][0] = 0000000077f988b8
register usb cfg[0][1] = 0000000077f988d8
vpu detect type: 5
vpu clk_level = 7
set vpu clk: 666667000Hz, readback: 666660000Hz(0x300)
MMC:   aml_priv->desc_buf = 0x0000000073f39d30
aml_priv->desc_buf = 0x0000000073f3bec0
SDIO Port B: 0, SDIO Port C: 1
ret = 1 .[mmc_init] mmc init success
In:    serial
Out:   serial
Err:   serial
---------------------------------
MMC Size : 8 GB
---------------------------------
reading boot-logo.bmp.gz
** Unable to read file boot-logo.bmp.gz **
```

3. La commande `minicom -s` permet de configurer les paramètres de communication

```
reading boot-logo.bmp
** Unable to read file boot-logo.bmp **
movi: the partiton 'logo' is reading...

MMC read: dev # 0, block # 58976, count 4096 ... 4096 blocks read: OK
hpd_state=1
[CANVAS]addr=0x3f800000 width=3840, height=1440

[720p60hz] is invalid for cvbs.
set hdmitx VIC = 4
hdmitx phy setting done
set hdmitx VIC = 4
hdmitx phy setting done
Error: Bad gzipped data
There is no valid bmp file at the given address
Saving Environment to MMC...
Writing to MMC(0)... done
Net:   Meson_Ethernet
Hit [Enter] key twice to stop autoboot:  0
odroidc2#
```

Vous êtes maintenant dans l'environnement U-Boot ! Nous allons d'abord explorer cet environnement afin de vérifier que vous avez bien installé les fichiers sur votre carte microSD. Entrez la commande `help` :

```
odroidc2#help
...
loop    - infinite loop on address range
ls      - list files in a directory (default /)
md      - memory display
...
```

On remarque que la commande `ls` affiche les fichiers dans un répertoire comme sur un système Linux.

```
odroidc2#help fatls
fatls - list files in a directory (default /)

Usage:
fatls <interface> [<dev[:part]>] [directory]
    - list files from 'dev' on 'interface' in a 'directory'
```

Deux mémoires de type MMC peuvent être installées sur votre système. La première est la carte microSD (`mmc 0`) et la deuxième est la eMMC du Odroid-C2 (`mmc 1`), qui permet en autre de précharger un système d'exploitation (pas présente dans notre cas).

Affichez les fichiers de votre carte microSD dans les deux partitions (elles sont numérotées à partir de 1) :

```
odroidc2#fatls mmc 0:1
 12221088   image
    27643   meson64_odroidc2.dtb
     2693   boot.ini

3 file(s), 0 dir(s)

odroidc2#ext4ls mmc 0:2
<DIR>       1024 .
<DIR>       1024 ..
<DIR>      12288 lost+found
<DIR>       1024 bin
<DIR>       1024 sys
<DIR>       1024 dev
<DIR>       1024 proc
```

Pour un *boot* du système, il faut spécifier ses options. Agrandissez la fenêtre de la console en plein écran car `minicom` ne gère pas le retour à la ligne et entrer les commandes suivantes :

```
odroidc2#setenv m "1080p60hz"
odroidc2#setenv m_bpp "32"
odroidc2#setenv vout "dvi"
odroidc2#setenv condev "console=ttyS0,115200n8"
odroidc2#setenv bootargs "root=/dev/mmcblk0p2 rootwait rw ${condev} no_console_suspend hdmimode=${m} m_bpp=${m_bpp} vout=${vout} init=/bin/init"
odroidc2#setenv loadaddr "0x11000000"
odroidc2#setenv dtb_loadaddr "0x1000000"
odroidc2#fatload mmc 0:1 ${loadaddr} Image
reading Image
12221088 bytes read in 816 ms (14.3 MiB/s)
odroidc2#fatload mmc 0:1 ${dtb_loadaddr} meson64_odroidc2.dtb
reading meson64_odroidc2.dtb
27643 bytes read in 7 ms (3.8 MiB/s)
```

Voici la signification de certains des paramètres :

— `setenv` : Indique qu'on assigne une nouvelle à la variable indiquée en deuxième argument.

— `bootargs` : Le paramètre dans l'environnement U-Boot qui indique les options de boot.

— `console=ttyS0,115200n8` : Indique que le device à utiliser pour la communication sérielle i.e. la console en cours est `ttyS0`. Si vous ne mettez pas cet argument, la console n'affichera pas l'exécution du *kernel*. L'option `115200n8` indique le `baud rate`. Un baud fait référence à un bit. Le taux de transmission est donc de 115200 bits/s soit de 1.4 ko/s.

— `root=/dev/mmcblk0p2` : indique la partition sur laquelle le système de fichiers se situe.

— `rootwait` : indique qu'on attend l'utilisateur pour démarrer la ligne de commande.

— `rw` : indique que le système de fichier sera un lecture/écriture.

— `init=/bin/init` : spécifie le programme qu'on doit exécuter une fois que le *kernel* a été chargé. Souvenez vous, `/bin/init` est un lien symbolique vers `busybox` qui sera donc le programme qui s'exécute à l'initialisation du système.

— `loadaddr` : l'emplacement de chargement du *kernel*.

— `dtb_loadaddr` : l'emplacement de chargement du *device tree*.

Vous êtes maintenant prêt à démarrer l'image du *kernel*. Analysons la commande `booti` :

```
odroidc2#help booti
booti - boot arm64 Linux Image image from memory

Usage:
booti [addr [initrd[:size]] [fdt]]
    - boot Linux Image stored in memory
        The argument 'initrd' is optional and specifies the address
        of the initrd in memory. The optional argument ':size' allows
        specifying the size of RAW initrd.
        Since booting a Linux kernel requires a flat device-tree
        a third argument is required which is the address of the
        device-tree blob. To boot that kernel without an initrd image,
        use a '-' for the second argument.
```

Dans notre cas, nous n'avons pas de `initrd` et nous avons un *device-tree*. Nous devons donc lancer cette commande de cette manière :

```
odroidc2#booti ${loadaddr} - ${dtb_loadaddr}
Starting kernel ...
```

```
uboot time: 4265901 us
[    0.000000] Initializing cgroup subsys cpuse
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Initializing cgroup subsys cpuacct
[    0.000000] Linux version 3.14.29-56-yocto-standard+ (gourdeau@gourdeau-Aspire-E5-
[    0.000000] CPU: AArch64 Processor [410fd034] revision 4

...

Freeing unused kernel memory: 860K (ffffffc001a7b000 - ffffffc001b52000)
can't run '/etc/init.d/rcS': No such file or directory

Please press Enter to activate this console.
```

Si vous obtenez le résultat ci-dessus, vous avez bien compilé votre *kernel* Linux ! Appuyez sur Entrée et essayez la commande `ls` :

```
/ # ls
-/bin/sh: ls: not found
```

En effet, nous n'avons pas créé de lien symbolique qui pointe vers `busybox`. Créez un lien symbolique `ls` qui pointe vers `busybox` :

```
/ # busybox printenv
USER=root
HOME=/
TERM=vt102
PATH=/sbin:/usr/sbin:/bin:/usr/bin
SHELL=/bin/sh
PWD=/
/ # busybox ln -s /bin/busybox /bin/ls
/ # ls
bin   dev   proc  sys
/ # busybox which ls
/bin/ls
```

Puisque le répertoire `/bin` se trouve dans la variable de chemins du système, le système d'exploitation va savoir où chercher le lien symbolique. Il est donc suggéré de créer des liens symboliques pour chacune des commandes de `busybox` dans `bin`

(ou tout autres répertoire dans le `PATH`) afin de permettre à tous les utilisateurs de les utiliser sans spécifier `busybox`. Mais quelles commandes sont disponibles ? Il suffit de demander !

```
/ # busybox --help | busybox more
BusyBox v1.21.1 (2013-07-08 10:26:30 CDT) multi-call binary.
BusyBox is copyrighted by many authors between 1998-2012.
Licensed under GPLv2. See source distribution for detailed
copyright notices.

Usage: busybox [function [arguments]...]
   or: busybox --list[-full]
   or: busybox --install [-s] [DIR]
   or: function [arguments]...

        BusyBox is a multi-call binary that combines many common Unix
        utilities into a single executable.  Most people will create a
        link to busybox for each function they wish to use and BusyBox
        will act like whatever it was invoked as.

Currently defined functions:
        [, [[, acpid, add-shell, addgroup, adduser, adjtimex, arp, arping, ash,
        awk, base64, basename, beep, blkid, blockdev, bootchartd, brctl,
        bunzip2, bzcat, bzip2, cal, cat, catv, chat, chattr, chgrp, chmod,
        chown, chpasswd, chpst, chroot, chrt, chvt, cksum, clear, cmp, comm,
        conspy, cp, cpio, crond, crontab, cryptpw, cttyhack, cut, date, dc, dd,
        deallocvt, delgroup, deluser, depmod, devmem, df, dhcprelay, diff,
        dirname, dmesg, dnsd, dnsdomainname, dos2unix, du, dumpkmap,
--More--
...
```

Vous venez d'entrer manuellement les instructions du `boot.ini` déjà présent sur la carte microSD.

Faites la commande `busybox reboot` pour voir le *boot* automatique.

La commande `busybox poweroff` est aussi disponible.

9 Compilation et installation d'un `helloworld`

Nous allons maintenant compiler un `helloworld` pour notre cible. Ouvrez un nouveau terminal et créez un fichier `main.c` et un fichier `CMakeLists.txt` :

```
% cd /export/tmp/4205_nn/linux_scratch/
% mkdir helloworld
% cd helloworld/
% touch main.c CMakeLists.txt
```

Mettez ce contenu dans les deux fichiers :

main.c

```
#include <stdio.h>
#include <unistd.h>

int main()
{
        while(1)
        {
                printf("Hello World !\n");
                sleep(1);
        }
        return 0;
}
```

CMakeLists.txt

```
add_executable(helloworld main.c)
```

Compilez le programme sur la machine du laboratoire pour tester sa fonctionnalité (CTRL-C pour interrompre le programme) :

```
% mkdir build_dyn
% cd build_dyn
% cmake ..
-- The C compiler identification is GNU 4.4.7
-- The CXX compiler identification is GNU 4.4.7
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
```

```
-- Build files have been written to: /export/tmp/scratch_test/helloworld
% make
Scanning dependencies of target helloworld
[100%] Building C object CMakeFiles/helloworld.dir/main.c.o
Linking C executable helloworld
[100%] Built target helloworld
% ./helloworld
Hello World !
Hello World !
Hello World !
^C
%
```

Tel que compilé en ce moment, le programme est compilé dynamiquement, c'est-à-dire que les librairies utilisées ne sont pas inclues dans l'exécutable. Vous pouvez vérifier ce constate avec la commande `ldd` :

```
% ldd helloworld
    linux-vdso.so.1 =>  (0x00007ffd623c4000)
    libc.so.6 => /lib64/libc.so.6 (0x0000003497000000)
    /lib64/ld-linux-x86-64.so.2 (0x0000003496c00000)
```

Ces librairies sont généralement inclues dans la librairie standard GLibC. Nous pourrions donc installer l'exécutable `helloworld` ainsi que les trois librairies ci-dessus dans le chemin standard des librairies mais il existe un autre moyen : la compilation statique. Un exécutable statique inclut toutes ses dépendances vers les librairies externes au prix d'une taille d'exécutable plus grosse.

Revenez dans le dossier qui contient votre `main.c` :

```
% cd ..
% mkdir build_sta
% cd build_sta
% cmake -DCMAKE_EXE_LINKER_FLAGS="-static" ..
% make
% ./helloworld
Hello World !
Hello World !
^C
% ldd helloworld
not a dynamic executable
```

L'exécutable n'a plus de dépendances externes ! Nous n'aurions qu'à installer le helloworld compilé pour l'architecture ARM sur notre système de fichiers. **Attention !!** une édition des liens statique avec GLibC (*GNU C Library*) qui est sous license LGPL rend votre programme *open source*. Donc, si vous faites un produit commercial, vous devrez rendre les sources de votre application disponibles. De plus, si plus d'une application utilise GLibC, le gain d'espace est perdu par la copie multiple dans les différents exécutables.

Compilez le projet cette fois-ci avec le SDK utilisé pour compiler le *kernel* :

```
% cd ..
% mkdir build_arm64
% cd build_arm64
% bash
$ source /export/tmp/4205_nn/opt/poky/environment-setup-aarch64-poky-linux
$ cmake ..
-- The C compiler identification is GNU 5.3.0
-- The CXX compiler identification is GNU 5.3.0
-- Check for working C compiler: /export/tmp/4205_nn/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aarc
-- Check for working C compiler: /export/tmp/4205_nn/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aarc
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /export/tmp/4205_nn/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aa
-- Check for working CXX compiler: /export/tmp/4205_nn/opt/poky/sysroots/x86_64-pokysdk-linux/usr/bin/aarch64-poky-linux/aa
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /export/tmp/4205_nn/linux_scratch/helloworld/build_arm64
$ make
Scanning dependencies of target helloworld
[100%] Building C object CMakeFiles/helloworld.dir/main.c.o
Linking C executable helloworld
[100%] Built target helloworld
$ ./helloworld
bash: ./helloworld: cannot execute binary file
```

Il ne reste plus qu'à copier le `helloworld` sur votre carte microSD dans la partition `root`, dans le dossier `/bin/`, mais il ne faut pas oublier nos libraries dynamiques.

```
$ readelf -d helloworld
Dynamic section at offset 0xa30 contains 24 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x400770
 0x000000000000000d (FINI)               0x4009e8
...
```

Bon, `helloworld` a besoin de la librairie `libc.so.6`.

```
$ pushd ../../../opt/poky/sysroots/aarch64-poky-linux/lib/
/export/tmp/4205_nn/opt/poky/sysroots/aarch64-poky-linux/lib /export/tmp/4205_nn/linux_scratch/hello
$ ls -l libc.*
lrwxrwxrwx 1 4205_nn cours 12 Jul 23 13:52 libc.so.6 -> libc-2.23.so
```

qui est un lien symbolique vers `libc-2.23.so`. Mais est-ce `libc-2.23.so` a aussi des dépendances ?

```
$ readelf -d libc.so.6

Dynamic section at offset 0x133b98 contains 23 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [ld-linux-aarch64.so.1]
 0x000000000000000e (SONAME)             Library soname: [libc.so.6]
 0x000000000000000c (INIT)               0x1f608
...
```

oui : `ld-linux-aarch64.so.1` :

```
$ ls -l ld-linux-aarch64.so.1
lrwxrwxrwx 1 4205_nn cours 10 Jul 23 13:52 ld-linux-aarch64.so.1 -> ld-2.23.so
$ readelf -d ld-linux-aarch64.so.1

Dynamic section at offset 0x1ce40 contains 19 entries:
  Tag        Type                         Name/Value
 0x000000000000000e (SONAME)             Library soname: [ld-linux-aarch64.so.1]
 0x0000000000000004 (HASH)               0x190
 0x000000006ffffef5 (GNU_HASH)           0x258

...
```

Il faudra donc copier les librairies en plus de l'exécutable.

```
$  popd
/export/tmp/4205_nn/linux_scratch/helloworld/build_arm64
$ cp helloworld ../../oc2sdcard/rootfs/bin
$ cd  ../../oc2sdcard/rootfs/
$ mkdir lib
$ pushd ../../../opt/poky/sysroots/aarch64-poky-linux/lib/
/export/tmp/4205_nn/opt/poky/sysroots/aarch64-poky-linux/lib /export/tmp/4205_nn/linux_scratch/oc2sd
$ cp libc-2.23.so ld-2.23.so /export/tmp/4205_nn/linux_scratch/oc2sdcard/rootfs/lib/
$ popd
/export/tmp/4205_nn/linux_scratch/oc2sdcard/rootfs
$ cd lib
$ ln -s libc-2.23.so libc.so.6
$ ln -s ld-2.23.so ld-linux-aarch64.so.1
```

```
$ ls -l
total 1380
-rwxr-x--- 1 4205_nn cours  124544 Aug  3 16:48 ld-2.23.so
lrwxrwxrwx 1 4205_nn cours      10 Aug  3 16:51 ld-linux-aarch64.so.1 -> ld-2.23.so
-rwxr-x--- 1 4205_nn cours 1275384 Aug  3 16:48 libc-2.23.so
lrwxrwxrwx 1 4205_nn cours      12 Aug  3 16:50 libc.so.6 -> libc-2.23.so
```

Mettez à jour votre carte microSD.

```
$ cd ../../
$ ../makesdcardoc2
$ dd if=oc2_lab2.sdcard of=/dev/sdc && sync && sync
```

Redémarrer avec votre carte mise à jour et une fois le *boot* complété vérifiez que `helloworld` est bien sur votre carte microSD :

```
/ # ls bin
busybox     helloworld  init        ls          sh
/ # ls lib
ld-2.23.so              libc-2.23.so
ld-linux-aarch64.so.1   libc.so.6
/ # helloworld
Hello World !
Hello World !
Hello World !
^C
/ #
```

On aurait pu spécifier `helloworld` comme le programme de démarrage en spécifiant dans `boot.ini` un `bootargs` :

```
setenv bootargs "root=/dev/mmcblk0p2 rootwait rw ${condev} no_console_suspend
   hdmimode=${m} m_bpp=${m_bpp} vout=${vout} init=/bin/helloworld"
```

Pour obtenir, au lieu de l'invite de `sh`,

```
Hello World !
Hello World !
Hello World !
```

Des `Hello World!` devraient apparaitrent indéfiniment (le `init` est rechargé automatiquement si interrompu). On peut donc spécifier n'importe quel programme de démarrage par défaut !

Vous pouvez tester cette façon de faire sans réécrire votre carte microSD en changeant le fichier `boot.ini` directment sur la carte branchée sur l'adaptateur à votre poste de travail (n'oubliez pas le *eject*).

10 Évaluation

Il n'y pas de rapport à écrire pour ce laboratoire. La présence est obligatoire et ce laboratoire compte pour 3% de la note finale. Vous devez compléter toutes les étapes pour réussir ce laboratoire.