LO52 - Compte Rendu condensé des TPs

Mériadeg Guichard

5 janvier 2021

Sommaire

1 TP1 : Mise en place de l'environnement de développement

1.1 Mise en place de l'environnement de base

Tout d'abord, il fallait commencer par récupérer le github de l'UV, cela s'est fait avec la commande suivante :

```
git clone truc
```

Une fois positionné dans le répertoire ainsi crée, il fallait ensuite configurer mon nom d'utilisateur ainsi que mon adresse email. Pour cela, il fallait utiliser les commandes suivantes :

```
git config user.name "name"
git config user.email "address@email.com"
```

Pour finir, j'ai crée une branche à mon nom afin de différencier mon travail de celui des autres (et afin de ne pas polluer la branche master) et je me suis positionné dessus. Après celà, je pouvais enfin commencer à travailler.

```
git branch GUICHARD
git checkout GUICHARD
```

J'ai du dans un premier temps installer et configurer AndroidStudio. Mon ordinateur ne me permettant pas d'émuler une tablette ou un téléphone, j'ai dû utiliser mon propre matériel. J'ai donc du adapter la compilation à mon téléphone, pour cela j'ai utilisé la version 27 du SDK (Oreo 8.1), car au delà, mon téléphone était considéré comme pas assez récent. Je n'ai rencontré que peu de difficultés dans cette première partie du TP.

1.2 Création d'un projet HelloWorld

L'objectifdu TP, après avoir installé et mis en place tout le matériel nécessaire, était de réaliser un application simple : une première activité présentant un bouton qui, une fois cliqué, nous dirigeait vers une autre activité contenant une zone de texte affichant "hello world".

J'ai pour cela crée un nouveau projet vide, que j'ai placé dans le répertoire TP1 de mon repo git.Ce projet se décomposait en deux activités distinctes, faisant chacune appel à leur propre layout (fichiers xml décrivant l'interface et son organisation de l'activité). J'ai juste ajouté un bouton au layout de l'activité principale, une zone de texte dans celui de l'activité *helloWorld*. J'ai ensuite programmé le bouton pour nous diriger vers *helloWorld* une fois cliqué. Pour cela il a juste suffi d'instancier un *Intent* lors du clic sur le bouton, et de charger l'activité *helloWorld*.

Après test de l'application, j'ai obtenu le résultat suivant :

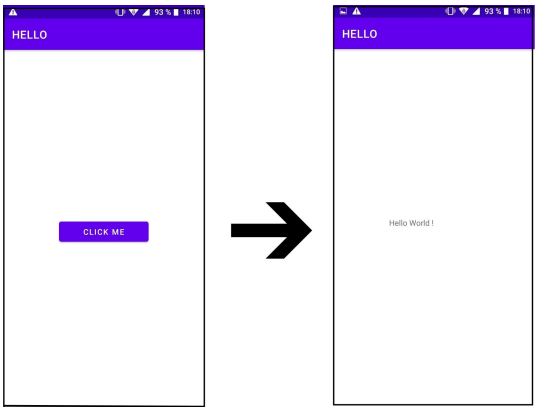

FIGURE 1 – Application HelloWorld

2 TP2 : Manipulation d'un Kernel Linux

2.1 Mise en place de l'environnement et configuration du kernel

Conformément au tutoriel qui nous était donné dans l'énoncé de TP, j'ai mis en place, dans un premier temps, l'environnement dans lequel j'allais travailler :

```
mkdir kernel && cd kernel
repo init -u https://android.googlesource.com/kernel/manifest -b hikey-linaro-android-
repo sync
```

Ma commande *repo init* diffère quelque peu de ce qui a pu être expliqué : en effet, j'avais constamment l'erreur suivante :

```
fatal: Cannot get https://gerrit.googlesource.com/git-repo/clone.bundle
```

Après quelques recherches, j'ai trouvé une solution : il est possible de télécharger directemet le fichier clone.bundle. Cela fait, en utilisant le paramètre *–repo-url /path/to/clone.bundle* il s'est avéré possible d'utiliser la commande et de récupérer les fichiers. Il n'y a eu aucun problème au niveau du téléchargement des sources, j'ai juste saturé l'intégralité de mon disque dur, ai du faire un peu de nettoyage et recommencer, ce qui n'a eu en soit aucune incidence sur mon matériel.

Pour ce qui est de la suite du TP, quelques liens, en particulier celui ci : https ://android.googlesource.com/platform/extern master-dev/android/docs/ANDROID-KERNEL.TXT m'auront aidé dans les différentes démarches à suivre.

J'ai ensuite informé le système que je comptais utiliser l'architecture arm64, j'ai donc pour cela fait un

```
export ARCH=arm64
```

J'ai ensuite chargé la configuration par défaut pour hikey-linaro à l'aide des commandes suivantes :

```
make clean
make mrproper
make hikeu_defconfig
mkdir configs
cp .config configs/hikey_defconfig
```

Je l'ai également copiée dans un répertoire configs afin de la sauvegarder pour le rendu du TP. J'ai ensuite commencé à me pencher sur la configuration par défaut pour une carte ranchu64 (comme indiqué dans le lien plus haut), j'ai donc identifié cette configuration et l'ai copiée puis chargé à l'aide des commandes suivantes afin de faire en sorte que cette configuration puisse être reprérée par le Makefile :

```
cp arch/mips/configs/generic/board.ranchu.config arch/arm64/configs
make oldconfig board.ranchu.config
cp .config configs/ranchuconfig
```

J'ai ensuite lancé un

```
diff configs/hikey_defconfig configs/ranchuconfig
make savedefconfig
cp defconfig configs/lastconfig
```

afin de repérer les différences entre les deux : il en résulte que l'une présente pas mal d'options étant en lien avec GoldFish. J'ai également compilé la fusion de ces deux configurations que j'ai enregistré sous le nom de *lastconfig* J'ai ensuite crée le script *myscript.sh* qui regroupe l'ensemble des commandes que j'ai pu utiliser. Je n'ai malheureusement pas réussi à finir le TP : le nombre d'options présent dans la configuration était absolument astronomique, et m'a complètement perdu. J'ai tenté quelques tests mais jamais rien de vraiment concluant.

3 TP3 : Création d'un device Android

3.1 Implémentation de la libusb

Dans une premier temps, il fallait déterminer les fichiers sources et les headers nécessaires à la compilation de la libusb sur Android, lesfichiers sources étaient : *core.c descriptor.c io.c os/darwini_sub.c os/linux_usbfs.c sync.c* tandis que les headers étaient *libusb.h libusbi.h os/darwin_usb.h os/linux_usbfs.h*

Il fallait ensuite compléter le fichier *Android.mk* présent dans le dossier *libusb*, le premier étant déjà complet. Je n'ai pas pu le faire valider par les enseignants du fait de la distanciation des TPs et du fait que la plupart des points étaient planifiés sur des horaires où je ne pouvais être présent, cependant, en m'aidant du précédent fichier *Android.mk* présent à la racine du répertoire, et de posts sur divers forums, je suis arrivé au fichier suivant :

```
LOCAL_PATH := $(call my-dir)
LOCAL_MODULE := libusb

LOCAL_C_INCLUDES += $(LOCAL_PATH) $(LOCAL_PATH)/os/
LOCAL_SRC_FILES := core.c descriptor.c io.c sync.c os/darwin_usb.c cos/linux_usbfs.c

include $(BUILD_SHARED_LIBRARY)
```

3.1.1 Erreurs présentes à la compilation

Lors de la compilation, une première erreur survient, sur la macro TIMESPEC_TO_TIMEVAL. N'étant pas définie, après quelques recherches il a fallu la définir manuellement dans le fichier io.c. comme suit :

```
do {
    (tv)-> tv_sec =(ts)->tv_sec;
    (tv)-> tv_usec =(ts)->tv_nsec/ 1000
} while ( 0 )
```

Une seconde erreur survient ensuite : il semblerait d'après mes recherches qu'il faille ajouter l'adresse de la libusb dans u fichier nommé *prelinkmap* mais je n'ai cependant pas réussi à corriger cette erreur.

3.2 Définition d'un nouveau produit Android

Mon produit devait suivre les contraintes suivantes :

1. Son nom devait être de la forme *lo52_nomDeMaBranche*
2. Il devait hériter du produit hikey_linaro

J'ai donc crée l'arborescence *device/utbm/lo52/guichard* dans laquelle j'ai ajouté le fichier *lo52_guichard.mk*. A la base, j'avais ajouté d'autres fichiers comme *vendorsetup.sh* mais n'ayant su les remplir, j'ai préféré les supprimer. :w Dans un premier temps, j'ai écrit mon fichier en suivant les contraintes émises :

```
$(call inherit-product, device/linaro/hikey/hikey.mk)

PRODUCT_NAME != lo52_guichard
PRODUCT_DEVICE := lo52_guichard
PRODUCT_BRAND := UTBM
PRODUCT_MODEL := LO52
PRODUCT_MANUFACTURER := utbm
```

J'ai enfin personnalisé les propritétés du fichier comme demandé dans le cours, c'est à dire :

1. en définissant la variable *PRODUCT_PROPERTY_OVERRIDES*
2. en affectant la valeur *lo52* a ro.hw
3. en affectant la valeur *8.8.8.8* à dns1
4. en affectant la valeur *4.4.4.4* à dns2

J'ai finalement ajouté les fichier de la libusb aux packages du produits, je n'ai cepandant pas réussi à surcharger le fichier *sym_keyboard*.

4 TP4 : Utilisation de la JNI

Pour ce TP, j'ai simplement crée un nouveau projet "Native C++" depuis Android Studio, après avoir téléchargé les sources pour utiliser le NDK. Contrairement au TP1, ce mini projet ne présente qu'une seule activité, qui va afficher l'ensemble des boutons et autres widgets nécessaires.

4.1 Codage de l'interface

Il fallait tout d'abord crée l'interface de l'application, j'ai pour cela choisi d'utiliser un fichier layout implémentant un ensemble de *linearLayout* regroupés au sein d'un *linearLayout* global. Cette interface présente 6 boutons, deux zones de texte et une zone éditable. Les 6 boutons, tels que précisés par le sujet sont UP, DOWN, RIGHT, LEFT, WRITE et READ. Concernant les deux zonesde textes, il y a la zone *input* dont la valeur par défaut est "Def". J'ai utilisé l'attribut *android :inputType="number"* afin d'éviter que l'utilisateur puisse rentrer n'importe quoi. Chacun de ses éléments possédait un identifiant unique afin de le récupérer depuis le code java. La zone *output* elle, est là pour afficher le texte renvoyé par l'utilisation des fonctions *read* et *write*. Enfin, la zone *translation* se situe entre les boutons LEFT et RIGHT et contient la traduction en Allemand des directions des boutons. La figure suivante donne un aperçu de cette interface :

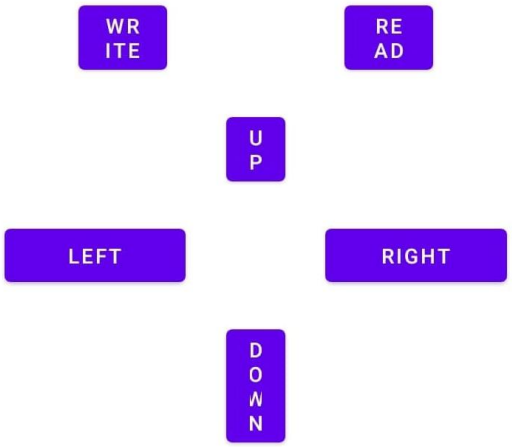

FIGURE 2 – Interface de l'application - La mise en forme peut ne pas convenir à tout le monde

4.2 Codage des fonctions natives

Pour implémenter nos fonctions natives, il faut dans un premier temps compléter le fichier *cpp/native-lib.cpp* selon le principe suivant :

1. Avant chaque fonction, il faut ajouter la ligne *extern "C" JNIEXPORT jstring JNICALL* pour préciser que l'on va coder une fonction native
2. Notre fonction doit ensuite s'écrire ainsi : *Java_com_mguichar_viertel_1tp_MainActivity_write* où il faut précisrr le nom du module, le nom de l'activité et finalement le nom de la méthode en elle même (ici, *write*)
3. Il faut déclarer ces méthodes au sein de notre activité, cela se fait par la ligne suivante : *private native String write(int i) ;*
4. Finalement, on précise l'origie des méthodes à l'aide de *System.loadLibrary("native-lib")*

4.3 Codage des différentes actions

Pourrécupérer les différents éléments de mon interface, j'ai utilisé la méthode classique qui consiste à employer *findViewById*. Cela fait, pour chaque bouton, il a fallu implémenter la méthode *onClick*. Ce qui se réduisait en l'appel des fonctions que j'ai implémenté plus tôt. Au final, après test de l'application, on pouvait avoir l'aperçu suivant :

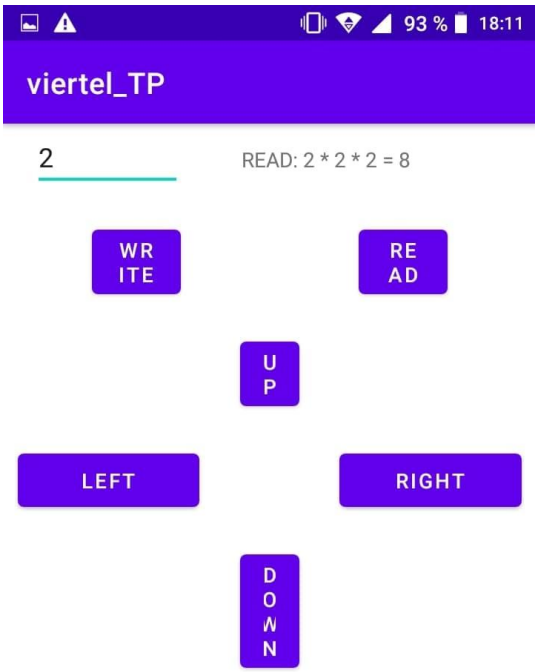

FIGURE 3 – Test de l'application