404410030 資工三 鄭光宇

實驗名稱：

Lab.1: Cross Toolchain for ARM Linux (Binutils, GCC)

實驗目的：

學習如何自行編譯 Cross Toolchain，並在過程中學習排除錯誤的能力。 若能了解 Cross Toolchain 如何編譯，將來要新增功能至 Toolchain 時，會比較容易知道該怎麼做。

實驗步驟：

以下實驗步驟都在 Ubuntu 16.04 完成。為了方邊，我將整個流程寫成數個 bash script，主要內容如下：

1.使用 apt-get 安裝相依套件：

```
#!/bin/bash

sudo apt-get update # update repository list
sudo apt-get install -y subversion # install svn
sudo apt-get install -y git # install git
sudo apt-get install -y build-essential # gcc/g++, make, etc.
# sudo apt-get install -y ftp # install ftp client
sudo apt-get install -y gawk libgmp-dev libmpfr-dev libmpc-dev flex bison # requir
ements to build gcc
```

2.下載要安裝的各種 tarball：

```
#!/bin/bash

mkdir src # make source dir
cd src

# Download binutils and uncompress it
wget "ftp://ftp.gnu.org/gnu/binutils/binutils-2.29.1.tar.gz"
tar zxvf "binutils-2.29.1.tar.gz"

# Clone GCC from svn
svn co "svn://gcc.gnu.org/svn/gcc/tags/gcc_4_9_3_release" gcc

# Clone Linux kernel from raspberrypi
# The Linux kernel version on my RPi is 4.14.22+
# So we clone the corresponding brach
git clone -b "rpi-4.14.y" --single-branch --depth 1 https://github.com/raspberrypi
/linux

# Download GLIBC and uncompress it
wget "ftp://ftp.gnu.org/gnu/glibc/glibc-2.19.tar.gz"
tar zxvf "glibc-2.19.tar.gz"
```

3.編譯 binutils：

```
#!/bin/bash
# TARGET=arm-linux-gnueabihf
# PREFIX="$PWD/crossgcc1" # target path to metal binutils
cd src/binutils-* # enter whatever binutils dir
rm -rf build "$PREFIX" # remove old build
mkdir build # isolate build files from source codes (keep tarball clean)
cd build
../configure --prefix="$PREFIX"  --target="$TARGET"
### WARNING: PLEASE CHECK YOUR C_INCLUDE_PATH IS SET CORRECTLY! ###
make -j
make install
```

4.編譯 gcc：

```
#!/bin/bash

# TARGET=arm-linux-gnueabihf
# PREFIX="$PWD/crossgcc1"

cd src/gcc
rm -rf build # remove previous build
mkdir build
cd build
../configure \
    --prefix="$PREFIX" \
    --target="$TARGET" \
    --enable-languages=c,c++ --without-headers \
    --disable-libmudflap --disable-libatomic \
    --with-arch=armv6 --disable-shared \
    --enable-static --disable-decimal-float \
    --disable-libgomp --disable-libitm \
    --disable-libquadmath --disable-libsanitizer \
    --disable-libssp --disable-threads \
    --with-float=hard --with-fpu=vfp
### This may take a looooonnnnnnggggg timeeeee!!! ###
make -j
make install
```

5.安裝 kernel header：

```
# SYSROOT="$PWD/sysroot"
cd src/linux
make headers_install ARCH=arm \
    INSTALL_HDR_PATH="$SYSROOT/usr"
```

6.編譯 glibc：

```
# HOST=arm-linux-gnueabihf
# TARGET=$HOST
# SYSROOT="$PWD/sysroot"
cd src/glibc-*
rm -rf clean_build
mkdir clean_build
cd clean_build
### CHECK YOUR LD_LIBRARY_PATH IS CORRECT !!! ###
../configure --prefix=/usr \
    --build="x86_64-linux-gnu" \
    --host="$HOST" --target="$TARGET" \
    --with-headers="$SYSROOT/usr/include" \
    --includedir=/usr/include --enable-add-ons \
    --disable-multilib
### THIS SHOULD TAKE A LONG TIME ###
make -j
make install install_root="$SYSROOT"
```

7.編譯 cross-binutils：

```
#!/bin/bash
# TARGET=arm-linux-gnueabihf
# SYSROOT="$PWD/sysroot"
# PREFIX="$PWD/crossgcc2" # target path to metal binutils
cd src/binutils-* # enter whatever binutils dir
rm -rf build_cx "$PREFIX" # remove old build
mkdir build_cx # isolate build files from source codes (keep tarball clean)
cd build_cx
../configure --prefix="$PREFIX"  --target="$TARGET" \
    --with-sysroot="$SYSROOT"
### WARNING: PLEASE CHECK YOUR C_INCLUDE_PATH IS SET CORRECTLY! ###
make -j
make install
```

8.編譯 cross-gcc：

```
#!/bin/bash

# TARGET=arm-linux-gnueabihf
# SYSROOT="$PWD/sysroot"
# PREFIX="$PWD/crossgcc2"

cd src/gcc
rm -rf build_cx # remove previous build
mkdir build_cx
cd build_cx
../configure \
    --prefix="$PREFIX" \
    --target="$TARGET" \
    --enable-languages=c,c++ \
    --with-sysroot="$SYSROOT" \
    --with-arch=armv6  \
    --with-float=hard --with-fpu=vfp \
    --disable-libmudflap --enable-libgomp \
    --disable-libssp --enable-libquadmath \
    --enable-libquadmath-support \
    --disable-libsanitizer --enable-lto \
    --enable-threads=posix --enable-target-optspace \
    --with-linker-hash-style=gnu --disable-nls \
    --disable-multilib --enable-long-long
### This may take a looooonnnnnngggggg timeeeee!!! ###
make -j
make install

# echo "export PATH=$PREFIX/bin:\$PATH" >> "$HOME/.bashrc" # export path to toolch
ain to global scope
```

9.加上環境變數，將每個 Script 照順序跑完，完成 Toolchain 的編譯：

```
#!/bin/bash

TARGET=arm-linux-gnueabihf
CROSS_GCC1_PREFIX="$PWD/crossgcc1"
CROSS_GCC2_PREFIX="$PWD/crossgcc2"
SYSROOT="$PWD/sysroot"
STDLOG="$PWD/logs_std"
ERRLOG="$PWD/logs_err"

./1.install_requirements.sh || ( >&2 echo "Failed to run 1." && exit )
./2.download_tarballs.sh || ( >&2 echo "Failed to run 2." && exit)
TARGET=${TARGET} PREFIX=${CROSS_GCC1_PREFIX} ./3.build_binutils.sh || ( >&2 echo "
Failed to run 3." && exit)
TARGET=${TARGET} PREFIX=${CROSS_GCC1_PREFIX} \
    PATH="$CROSS_GCC1_PREFIX/bin:$PATH" ./4.build_gcc.sh || ( >&2 echo "Failed to
run 4." && exit)
SYSROOT=${SYSROOT} ./5.install_kernel_headers.sh || ( >&2 echo "Failed to run 5."
&& exit)
PATH="$CROSS_GCC1_PREFIX/bin:$PATH" SYSROOT=${SYSROOT} \
    HOST=${TARGET} TARGET=${TARGET} ./6.build_glibc.sh || ( >&2 echo "Failed to ru
n 6." && exit)
PATH="$CROSS_GCC1_PREFIX/bin:$PATH" SYSROOT=${SYSROOT} \
    TARGET=${TARGET} PREFIX=${CROSS_GCC2_PREFIX} ./7.build_cross_binutils.sh || (
>&2 echo "Failed to run 7." && exit)
PATH="$CROSS_GCC2_PREFIX/bin:$PATH" SYSROOT=${SYSROOT} \
    TARGET=${TARGET} PREFIX=${CROSS_GCC2_PREFIX} ./8.build_cross_gcc.sh || ( >&2 e
cho "Failed to run 8." && exit)

"${CROSS_GCC2_PREFIX}/bin/${TARGET}-gcc" "$PWD/zj_d799.c" -o "$PWD/zj_d799" || ( >
&2 echo "Failed to compile source code!" )
```

可以注意到，我編譯 gcc 時使用的參數 `--enable-languages=c,c++` ，所以編譯完後，我們可以使用 C/C++ 的程式碼來測試編譯器是否正常運作。並呼叫 glibc 的函式，看看 glibc 是否正常運作。

測試程式碼：

```
#include <iostream>
int main(void) {
    std::cout << "Hello, Pi!" << std::endl;
    return 0;
}
```

編譯得到如下組語：

```
    .arch armv6
    .eabi_attribute 27, 3
    .eabi_attribute 28, 1
```

```
    .fpu vfp
    .eabi_attribute 20, 1
    .eabi_attribute 21, 1
    .eabi_attribute 23, 3
    .eabi_attribute 24, 1
    .eabi_attribute 25, 1
    .eabi_attribute 26, 2
    .eabi_attribute 30, 6
    .eabi_attribute 34, 1
    .eabi_attribute 18, 4
    .file   "hello.cpp"
    .local  _ZStL8__ioinit
    .comm   _ZStL8__ioinit,1,4
    .section    .rodata
    .align  2
.LC0:
    .ascii  "Hello, Pi!\000"
    .text
    .align  2
    .global main
    .type   main, %function
main:
    .fnstart
.LFB1020:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    stmfd   sp!, {fp, lr}
    .save {fp, lr}
    .setfp fp, sp, #4
    add fp, sp, #4
    ldr r0, .L3
    ldr r1, .L3+4
    bl  _ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
    mov r3, r0
    mov r0, r3
    ldr r1, .L3+8
    bl  _ZNSolsEPFRSoS_E
    mov r3, #0
    mov r0, r3
    ldmfd   sp!, {fp, pc}
.L4:
    .align  2
.L3:
    .word   _ZSt4cout
    .word   .LC0
    .word   _ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
    .fnend
    .size   main, .-main
```

```
    .align  2
    .type   _Z41__static_initialization_and_destruction_0ii, %function
_Z41__static_initialization_and_destruction_0ii:
    .fnstart
.LFB1029:
    @ args = 0, pretend = 0, frame = 8
    @ frame_needed = 1, uses_anonymous_args = 0
    stmfd   sp!, {fp, lr}
    add fp, sp, #4
    sub sp, sp, #8
    str r0, [fp, #-8]
    str r1, [fp, #-12]
    ldr r3, [fp, #-8]
    cmp r3, #1
    bne .L5
    ldr r3, [fp, #-12]
    ldr r2, .L7
    cmp r3, r2
    bne .L5
    ldr r0, .L7+4
    bl  _ZNSt8ios_base4InitC1Ev
    ldr r0, .L7+4
    ldr r1, .L7+8
    ldr r2, .L7+12
    bl  __aeabi_atexit
.L5:
    sub sp, fp, #4
    @ sp needed
    ldmfd   sp!, {fp, pc}
.L8:
    .align  2
.L7:
    .word   65535
    .word   _ZStL8__ioinit
    .word   _ZNSt8ios_base4InitD1Ev
    .word   __dso_handle
    .cantunwind
    .fnend
    .size   _Z41__static_initialization_and_destruction_0ii, .-_Z41__static_initia
lization_and_destruction_0ii
    .align  2
    .type   _GLOBAL__sub_I_main, %function
_GLOBAL__sub_I_main:
    .fnstart
.LFB1030:
    @ args = 0, pretend = 0, frame = 0
    @ frame_needed = 1, uses_anonymous_args = 0
    stmfd   sp!, {fp, lr}
```

```
    add fp, sp, #4
    mov r0, #1
    ldr r1, .L10
    bl  _Z41__static_initialization_and_destruction_0ii
    ldmfd   sp!, {fp, pc}
.L11:
    .align  2
.L10:
    .word   65535
    .cantunwind
    .fnend
    .size   _GLOBAL__sub_I_main, .-_GLOBAL__sub_I_main
    .section    .init_array,"aw",%init_array
    .align  2
    .word   _GLOBAL__sub_I_main(target1)
    .hidden __dso_handle
    .ident  "GCC: (GNU) 4.9.3"
    .section    .note.GNU-stack,"",%progbits
```

再看看編譯好的執行檔是否為 arm 的執行檔：

```
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
 interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 2.6.16, with debug_info, not
stripped
```

看上去沒有什麼問題！實際拿 Raspberry Pi 在上面跑，也可以正常執行。

問題與討論：

之前在編譯這個 Toolchain 時，有遇到 PATH 設定不正確，無法正確找到執行檔的問題，所以後來乾脆寫死在 Script 前面，直接將環境變數傳入，就不會出狀況了。