National Technical University of Athens
Electrical and Computer Engineering

Embedded Systems

5th Lab

Program cross-compiling for ARM architecture

Nanos Georgios

nanosgiwrgos1997@gmail.com

Preparing for the exercise In order to carry out the exercise correctly, we had to follow the following steps: 1. First, we created a second virtual machine (Virtual Machine) First of all, we created a virtual machine (Virtual Machine) in QEMU. To do this, we had to download the first virtual machine. The following files (corresponding to the files from lab exercise 3):

- `https://people.debian.org/~aurel32/qemu/armhf/debian_wheezy_armhf_standard.qcow2`
- `https://people.debian.org/~aurel32/qemu/armhf/initrd.img-3.2.0-4-vexpress`
- `https://people.debian.org/~aurel32/qemu/armhf/vmlinuz-3.2.0-4-vexpress`

2. Similarly with the 3rd laboratory exercise we can start the our virtual machine by running the following command:

```
sudo qemu-system-arm -M vexpress-a9 -kernel vmlinuz-3.2.0-4-vexpress -initrd
initrd.img3.2.0-4-vexpress -drive if=sd,file=debian_wheezy_armhf_standard.qcow2
-append "root=/dev/mmcblk0p2" -net nic -net user,hostfwd=tcp:127.0.0.1:22223-:22
```

3. We also needed to update the sources.list file in order to access the Debian repositories packages and to update the package lists (apt-get update), as we did for our previous virtual machine. Installing custom cross-compiler building toolchain crosstool-ng Steps: 1. First we downloaded the necessary files for building the toolchain from from the corresponding github repository. We execute the command:

```
:~$ git clone https://github.com/crosstool-ng/crosstool-ng.git
```

The command creates a folder in the directory where we ran it called crosstool-ng. 2. We enter the folder, and first run

```
:~/crosstool-ng$ ./bootstrap
```

3. Next, we need to create two folders in HOME directory as follows:

```
:~/crosstool-ng$ mkdir $HOME/crosstool && mkdir $HOME/src
```

In the first folder the crosstool-ng program will be installed and in the second, it will store the necessary packages downloaded to build the cross-compiler. 4. Execute the following command to configure the crosstool-ng installation:

```
:~/crosstool-ng$./configure --prefix=${HOME}/crosstool
```

During the execution of this command, many packets were missing. The were installed by searching for each package in a search engine, and then re-executed the above command. 5. We run the make command and makeinstall:

```
:~/crosstool-ng$ make && make install
```

6. Up to this point crosstool-ng has been installed. We go to the installation path $HOME-/crosstool/bin

```
:~/crosstool-ng$cd $HOME/crosstool/bin
```

In this folder we build our cross compiler. 7. We execute the command:

```
:~/crosstool/bin$ ./ct-ng list-samples
```

A list of many combinations of architectures, functionalities, and and C libraries provided by the tool to we can quickly and correctly configure the build cross-compiler we want to produce for a specific target machine.

```
arm-cortexa9_neon-linux-gnueabihf.
```

8. We execute the command

```
:~/crosstool/bin$ ./ct-ng arm-cortexa9_neon-linux-gnueabihf
```

to configure crosstool-ng for the specific architecture.

9. If we want to graphically change some of the features of the preconfigured target machine combination, such as which C library to use, run the command:

```
~/crosstool/bin$./ct-ng menuconfig
```

10. Finally after we have configured the crosscompiler we are ready to build it. Building the cross compiler is a relatively simple task. time-consuming process. We execute:

```
:~/crosstool/bin$./ct-ng build
```

11. After everything has gone well, the file was created $HOME/x-tools/arm-cortexa9_neon-linux-gnueabihf where inside the bin subfolder contains the executable files of your cross compiler Remark: Crosstool-ng could not download some files that were needed, so we downloaded them manually and placed them in the folder where folder indicated by the build.log file.

Installation of pre-compiled building toolchain linaro In addition to using the custom compiler toolchain, we also used a pre-compiled cross compiler provided by the `www.linaro.org/downloads`. To make use of the pre-compiled cross compiler we perform the following steps: 1. Download the binaries of the cross compiler from the following address:

```
~$mkdir~/linaro && cd ~/linaro
:~/linaro $wget
https://releases.linaro.org/archive/14.04/components/toolchain/binaries/
gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.bz2
```

2. Opening the parent directory of the tarball we downloaded (`https://releases.linaro.org/archive/14.04/components/toolchain/binaries/`), we notice at the bottom of the page that the binaries of the cross compiler include the following elements:

- Linaro GCC 4.8 2014.04
- A statically linked gdbserver
- Linaro Newlib 2.1 2014.02
- A system root
- Linaro Binutils 2.24.0 2014.04
- Manuals under share/doc/
- Linaro GDB 7.6.1 2013.10
- The system root contains the basic header files and libraries to link your programs against.

3. We extract the downloaded files.

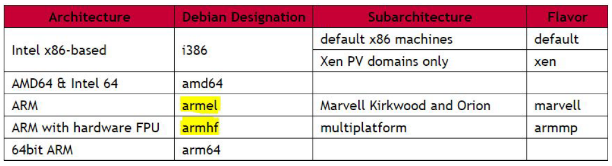
```
:~/linaro$ tar -xvf gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux.tar.bz24
```

4. The binaries of the crosscompiler should be in the package that downloaded in the folder:

```
$HOME/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin
```

Issue 1

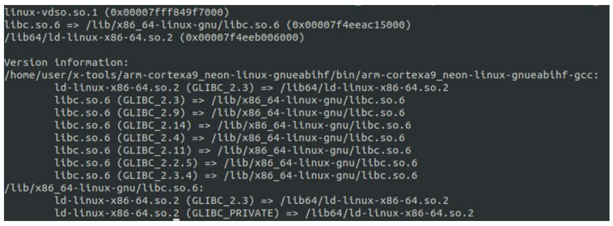
1. The armel (arm EABI little-endian) architecture supports the set of ARMv4 instruction set. This architecture handles number computation in compatibility mode that slows down the performance, but allows compatibility with code written for processors without floating-point units. Thus it can the armel architecture can be used to create systems high compatibility systems. The armhf (arm hard-float) architecture supports the ARMv7 platform, and in addition, it adds direct hardware support for mobile submodules. This means that the armhf architecture is faster than that of armel, but lacks compatibility with legacy architectures.



Architecture	Debian Designation	Subarchitecture	Flavor
Intel x86-based	i386	default x86 machines	default
		Xen PV domains only	xen
AMD64 & Intel 64	amd64		
ARM	armel	Marvell Kirkwood and Orion	marvell
ARM with hardware FPU	armhf	multiplatform	armmp
64bit ARM	arm64		

2. We used "arm-cortexa9_neon-linux-gnueabihf", because the guest machine has this architecture. Otherwise, if if we tried to run a cross-compiled executable of a different architecture, the executable would give us an error, as a a completely different architecture in terms of the Instruction Set Architecture of the system. 3. The C library we used is glibc, as it matches best suited to our needs. There were possibilities of using libraries smaller libraries, but these either do not support ARM or do not support the Debian distribution, so we can't use them. This can be seen by using the command:

```
$ ldd -v ~/x-tools/arm-cortexa9_neonlinux-gnueabihf/bin/arm-cortexa9_neon-
    linux-gnueabihf-gcc
```



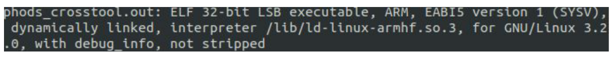
```
linux-vdso.so.1 (0x00007fff849f7000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f4eeac15000)
/lib64/ld-linux-x86-64.so.2 (0x00007f4eeb006000)

Version information:
/home/user/x-tools/arm-cortexa9_neon-linux-gnueabihf/bin/arm-cortexa9_neon-linux-gnueabihf-gcc:
        ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
        libc.so.6 (GLIBC_2.3) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.9) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.14) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.4) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.11) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.2.5) => /lib/x86_64-linux-gnu/libc.so.6
        libc.so.6 (GLIBC_2.3.4) => /lib/x86_64-linux-gnu/libc.so.6
/lib/x86_64-linux-gnu/libc.so.6:
        ld-linux-x86-64.so.2 (GLIBC_2.3) => /lib64/ld-linux-x86-64.so.2
        ld-linux-x86-64.so.2 (GLIBC_PRIVATE) => /lib64/ld-linux-x86-64.so.2
```

4. Using the cross compiler produced by crosstool-ng compile the phods.c code with flags -O0 -Wall -o phods_crosstool.out from the 2nd query of the 1st exercise (the simple phods code along with the gettimeofday() function).

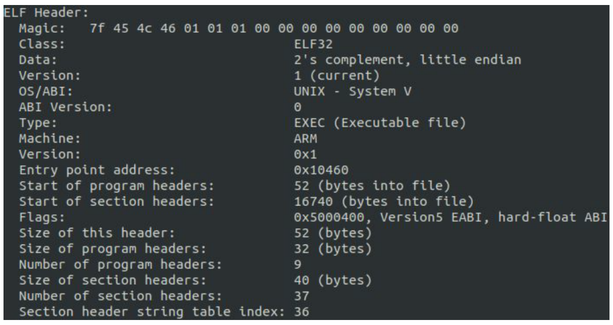
```
sudo ./arm-cortexa9_neon-linux-gnueabihf-gcc phods.c -O0 -Wall -o
phods_crosstool.out
```

We run the commands on the local machine:

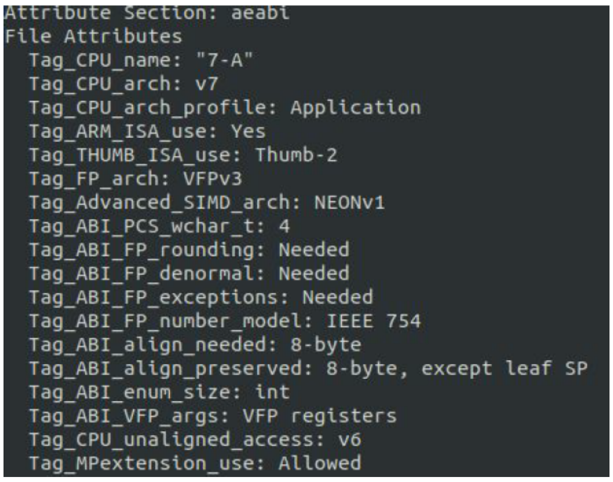
```
:~$ file phods_crosstool.out
:~$ readelf -h -A phods_crosstool.out
```



```
phods_crosstool.out: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
 dynamically linked, interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2
.0, with debug_info, not stripped
```



```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           ARM
  Version:                           0x1
  Entry point address:               0x10460
  Start of program headers:          52 (bytes into file)
  Start of section headers:          16740 (bytes into file)
  Flags:                             0x5000400, Version5 EABI, hard-float ABI
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         37
  Section header string table index: 36
```



```
Attribute Section: aeabi
File Attributes
  Tag_CPU_name: "7-A"
  Tag_CPU_arch: v7
  Tag_CPU_arch_profile: Application
  Tag_ARM_ISA_use: Yes
  Tag_THUMB_ISA_use: Thumb-2
  Tag_FP_arch: VFPv3
  Tag_Advanced_SIMD_arch: NEONv1
  Tag_ABI_PCS_wchar_t: 4
  Tag_ABI_FP_rounding: Needed
  Tag_ABI_FP_denormal: Needed
  Tag_ABI_FP_exceptions: Needed
  Tag_ABI_FP_number_model: IEEE 754
  Tag_ABI_align_needed: 8-byte
  Tag_ABI_align_preserved: 8-byte, except leaf SP
  Tag_ABI_enum_size: int
  Tag_ABI_VFP_args: VFP registers
  Tag_CPU_unaligned_access: v6
  Tag_MPextension_use: Allowed
```

These commands give us information, from the binary headers file, about its type, the ABI, the targeted architecture (Machine), etc. More specifically, the first thing we notice is that the architecture on which the file is running is ARM. Still, it works dynamic linking and the library file it calls is the /lib/ld-linux-armhf.so.3. We still have a lot of CPU information for the this file is built for, and also for the ABI, so it knows the functional details for handling operations, etc.

5. Using the cross compiler we downloaded from the linaro, we compile the same code as in question 3.

```
~/linaro/gcc-linaro-arm-linux-gnueabihf-4.8-2014.04_linux/bin$
./arm-linux-gnueabihf-gcc-4.8.3 phods.c -O0 -Wall -o phods_linaro.out
```

- arm-cortexa9: 18.2 kB (18,220 bytes)
- linaro: 8.4 kB (8,355 bytes)

So, we see that the phods_crosstool.out executable has the above more than twice the size of the phods_linaro.out executable. This because the custom cross-compiler uses 64-bit glibc while linaro 32-bit. This can be seen in the following command.

```
ldd -v arm-linux-gnueabihf-gcc-4.8.3
```

6. The executable produced by linaro (phods_linaro.out), of query 5, runs correctly on the target machine because a 64-bit system can run 32-bit executables. To make this happen the following was performed: Enable i386 package installation support:

```
sudo dpkg --add-architecture i386
sudo apt-get update
```

Install packages need to build source code on the system:

```
sudo apt-get install git build-essential fakeroot
```

gcc-multilib is the package which will enable running 32bit (x86) binaries on a 64bit (amd64/x86_64) system.

```
sudo apt-get install gcc-multilib
sudo apt-get install zlisudo apt-get install git build-essential fakeroot
```

7. We executed queries 4 and 5 with additional flag -static. The flag that flag asks the compiler to do static linking of the each compiler's respective C library. The sizes of the two files are shown below:

- arm-cortexa9: 2.9 MB (2,923,704 bytes)
- linaro: 507.9 kB (507,938 bytes)

We notice a difference in size, as the static linking has has all the 64-bit C libraries built in that reference the 64-bit C library phods program. That is, the libraries are part of the final executables which increases their size dramatically.

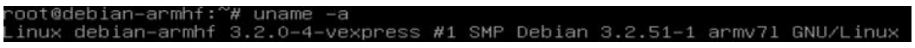
As can be seen from the above, the choice of cross-compiler may affect the size of the executable (as commented before), but the most important role is played by the choice between dynamic linking of the libraries and static. 8. We add a function of our own to mlab_foo() in glibc and create a cross-compiler with crosstool-ng that makes use of updated glibc. We create a file my_foo.c in which make use of the new function we created and make it cross compile with flags -Wall -O0 -o my_foo.out
A. If we run my_foo.out on the host machine, we we will notice that it cannot be executed because the executable has built for a different architecture (for the target machine).
B. If we run my_foo.out on the target machine, we will observe that it cannot be executed because the target machine does not have the new library to dynamically link it.
C. Add the -static flag and recompile the file my_foo.c. If we run my_foo.out on the target machine, we observe that it can now be executed because, now the executable has the binary of the new library embedded (since compiled with the -static flag).

Issue 2

1. We run in Qemu:

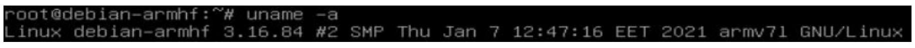
```
:~$ uname -a
```

Before installing a new kernel:



```
root@debian-armhf:~# uname -a
Linux debian-armhf 3.2.0-4-vexpress #1 SMP Debian 3.2.51-1 armv7l GNU/Linux
```

After installing a new kernel:



```
root@debian-armhf:~# uname -a
Linux debian-armhf 3.16.84 #2 SMP Thu Jan 7 12:47:16 EET 2021 armv7l GNU/Linux
```

What we see changing substantially is the version of the operating system, which is now 3.16.84, whereas before it was 3.2.0.4. 2. We have added a new system call to the linux kernel that uses the printk function to print to the kernel log the "Greeting from kernel and team no %d" along with the name of our team. The changes we made to the kernel source code are:

First we create a /new folder in the linux-source-3.16 directory with First, we start with the first file. A new.c and a Makefile that will convert it to a new.c file. object file.

```c
#include <linux/kernel.h>
asmlinkage long sys_new(void)
{
    printk(KERN_ALERT "Greeting from kernel and team 20!\n");
    return 0;
}
obj-y := new.o
```

In /include/linux/syscalls.h we add

```
asmlinkage long sys_new(void);
```

Then we add to /arch/arm/include/asm/unistd.h

```
#define __NR_sys_new (__NR_SYSCALL_BASE+378)
```

Then add to /arch/arm/kernel/calls.S

```
/* 385 */ CALL(sys_memfd_create)
CALL(sys_new)
```

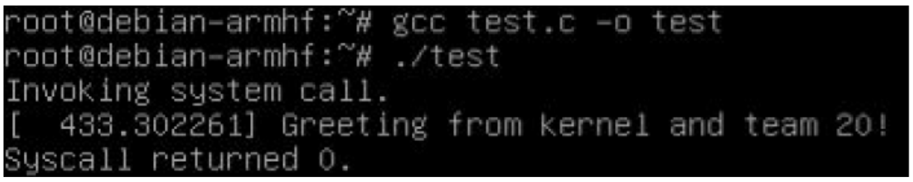
Finally we add our system call folder to the Makefile of kernel to the core-y rule, modifying the following one line in the /Makefile file.

```
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/
```

```
core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ new/
```

We compile and load the new kernel with the new call system. 3. Finally, we wrote a C program that makes use of the system call we added. Below, the test.c code for testing the correct operation of the new system call.

```
#include <unistd.h>
#include <sys/syscall.h>
#include <stdio.h>
#define SYS_new 386
int main(void) {
    printf("Invoking system call.\n");
    long ret = syscall(SYS_new);
    printf("Syscall returned %ld.\n", ret);
    return 0;
}
```



```
root@debian-armhf:~# gcc test.c -o test
root@debian-armhf:~# ./test
Invoking system call.
[  433.302261] Greeting from kernel and team 20!
Syscall returned 0.
```