Report on Transplanting JOS to ARM

Tiancheng Jin, Xingyu Lin, Liangchuan Zou

January 22, 2016

1 Introduction

In this project, we transplanted the JOS from x86 into ARM architecture so that it can later be run on some embeded systems. The system is tested with basic functions provied by Lab1 through QEMU simulator. During this project, we mainly reference the tutorial [1] for setting up the environment. And we get valuable help from Ziyue Yang and Jialin Liu.

Our complete code can be found at [2] which should be running properly.

2 Environment for Developing

2.1 Toolchains

In order to cross compile for ARM architecture, we install the following packages:

apt-get install gcc-arm-none-eabi gdb-multiarch

2.2 Simulator

Although vanilla QEMU does not yet support the RaspberryPi hardware, there is a fork by Torlus [3] that emulates the RPi hardware, where we adopted the following configure: **–target-list=arm-softmmu,arm-linux-user,armeb-linux-user –enable-sdl**

3 Building and Simulation

3.1 Compiler Options

We maintain all our compiling options in file **GNUmakefile**.

- We need to specify the GCC and QEMU we need for ARM:

  GCCPREFIX = 'arm-none-eabi-', QEMU = qemu-system-arm

- Some language, warning, debugging options:

  -Wall -Wno-format -Wno-unused -Werror -gstabs

- Disable any references to the libraries or headers on the host system:

  -nostdlib -nostdinc -ffreestanding

- Since the ARM architecture does not support division and mod, we need one library for that:

  -lgcc

- Specify the target machine type

  -mcpu=arm1176jzf-s

3.2 Linking

The linking process in **kern/kernel.ld** is almost the same with that of x86. The only difference is that we delete the header where it sets the output format as i386-ELF.

3.3 Simulator option

Simulator option is **-kernel -cpu arm1176 -m 256 -M raspi -serial stdio -gdb tps::portnumber**.(We used a special version of qemu which is designed for RaspberryPi[3]).

Option **-kernel** means we use the kernel file as it is in the memory. And the **-cpu** and the **-M** options specified the target machine. The three following options are the debugging options, specified the port number and the mode.

4 Lab1

4.1 Entry of Kernel

According to the old version of the **kern/entry.S** on the i386 platform, there are 3 things we need to do before we enter the kernel.

- Prepare the stack.
- Prepare the basic memory manage unit (**kern/entrygpdir.c**)
- Set the cpu control words.

Due to the lack of the guidelines (The tutorial version is naïve). So we find a much bigger toy ARM OS by link from **osdev.org**[4]. The **pi-baremetal** gives us a more concrete and clear routine to build up our own **entry.S** and **entrypgdir.c**. Also, we have looked up the Linux kernel source code under **/arch/arm** to learn further about how to start the kernel.(Linux core is very complex and lengthy)

4.1.1 Stack Preparation

Actually this part is very confused because we skip the bootstrap and boot loader steps and start at the entry of the kernel. So the space below the address 0x100000 is free to use. According to the baremetal, ARM CPUs use different stacks while they are under different modes. For each mode, the OS should give them an independent memory region. The **cps** assembly instruction can change the CPU mode and sets them one by one.

4.1.2 MMU Preparation and Set the Control Words

First we should introduce the specific control registers named Translation Table Register 0 and Translation Table Register 1. Register0 saves the base physical address for user context (lower address space , which is process-specific and used for process), and Register1 saves for the OS and IO (higher address space , above 0x80000000). ARM is a kind of RISC architecture, so the usage of the assembly can be very interesting. We use

mrc p15, 0, <Rd>, c2, c0, 0

to read this register and use

mcr p15, 0, <Rd>, c2, c0, 0

to write.

Another register called Control Register is much more important. It controls MMU and PIC and so on. The only thing we need to know is that the 22-bit of Control Register is 'Unaligned Data Access'(convenient for programmer to write asm code) and the 0-bit is the MMU Enable Bit.(According to [5], **2.2.3, 3.2.7, 3.2.13, 3.2.14**. Notice that the first-level table's base address need to be aligned to 2^{14})

In the entry part, we need to build up a dummy translation table(first-level), so the second thing we need to know is the arrangement of the table. There are 5 kinds of the first-level descriptor. We only use the section style, in order to save time (do not need to write down the second-level page entries). The section descriptor only needs the section base address, directly mapped to a 1MiB region. The format of section base address descriptor is that the last 2-bit should be "10". We have made up a dummy first-level table in **kern/entrypgdir.c**, mapped the lower 16MiB and the first 16MiB above KERNBASE to the lower physical memory, and reserve the 3MiB for IO. An addition setting is the domain number. In that case we set the domain number 0. (According to [5], **6.11.1**)

The last thing is to set the Domain Access Control Register. This time we set all bits of the register 1 for the kernel.(according to [5], **2.2.3 3.2.16**, every two bits of "11" represents "Manager", which means that MMU doesn't check the access permission in TLB entry of this domain).

4.2 ARM I/O

In order to communicate with the device through terminal, we mapped the serial device into the standard input and output when setting up the QEMU simulator. RasperryPi specifies a memory region for manipulating serial I/O, which starts at address 0x20200000 [6]. In this region, the CPU keeps checking the UART0_FR for status and the data is passed through UART0_DR. Since the I/O will be used all the time, we have reserved a whole section starting at 0x20200000 when setting the memory mapping described in 4.1.2. After this, all I/O operations are carefully packaged into **cputchar()** and **getchar**

During the programming for this low-level I/O, we also referenced the code here [7]

4.3 Debugger

Debugging the kernel is very hard. Without gdb, the only thing we can do is to check our code line by line. Further more, the sources online for ARM booting is either complex or scketchy. So we set up our debugging tool - **arm-none-eabi-gdb** in order to figure out what was happening in the kernel's booting. Maybe the "gdbserver" need to update as well.

The gdb can be easily installed by using command *sudo apt-get install gdb-arm-none-eabi*. And after that, we should remove the file **.gdbinit.tmpl** because the original hook-stop is based on i386 platform. So the **.gdbinit** wouldn't be generated automatically. We only reserve the options for "target remote". Also, the making rules of **gdb** and **.gdbinit** in the GNUmakefile should be replaced by *arm-none-eabi-gdb* instead of the *gdb -x .gdbinit*.

Now, the operations for gdb debugging is very similar to the original version. First we use the command *make qemu-gdb* to simulate the kernel on the qemu for debugging (plus some debug options such like **-S -gdb tcp::portnumber**). Then, in another terminal, we use the command *make gdb* to start debugging. Actually the gdb will use the **.gdbinit** and the **tcp::portnumber** to start remote debugging, which is supported by gdbserver. It helps a lot while dealing with kernel bugs.

References

[1] Arm raspberrypi tutorial: http://wiki.osdev.org/arm_raspberrypi_tutorial_c.

[2] Our code: https://github.com/grayking/josarms.

[3] Torlus/qemu: https://github.com/torlus/qemu/tree/rpi.

[4] A complete boiler plate code: https://github.com/brianwiddas/pi-baremetal.

[5] Arm1176jzf-s technical reference manual.

[6] Bcm2835-arm-peripherals chapter 13.

[7] Low level i/o referenced code: https://github.com/ztane/zsos/blob/master/kernel/arch/arm-eabi/uart.cc#l17.