CSE 536: Advanced Operating Systems

Assignment 1: A Tale of a Boot ROM and a Bootloader

Deadline: 11:59 PM on Feb 1, 2023

(100 points)

1 Brief Description

A boot ROM is CPU-specific code that executes when the machine starts. On our QEMU RISC-V system, the bootloader runs after the boot ROM. The bootloader is responsible for finding the operating system on the boot device and then passing control to it.

In this assignment, your goal is to **understand the boot process of an operating system using QEMU and write a custom bootloader for the xv6 OS kernel**. At a high level, the bootloader will perform four tasks. First, it will load on the machine before the OS starts and setup the execution of C code. Second, it will setup the hardware configuration to allow correct OS execution. Third, the bootloader will load the correct OS based on the user-provided configuration. Fourth, the bootloader will supply the OS with the system's configuration (e.g., starting address of bootloader binary).

GitHub Repo. Please clone as follows:

git clone -b lab1-bootloader https://github.com/ASTERISC-ASU/cse536-release.git

Initial provided code directory structure.

- bootloader: The directory that contains all the C code and headers for the bootloader. All changes in this assignment will be within this folder. Important files :
  - bootloader.ld
  - entry.S
  - start.c
  - riscv.h
  - elf.c
- kernel[1-4]: Different pre-compiled kernel images that you must load on the system.

Deliverables. You will submit your **updated code** (in an archived file) and a **PDF document** containing answers to all questions asked in this handout. The code is worth 80 points, the PDF is worth 20 points.

Suggested reading(s)

xv6: A Simple, Unix-like Teaching Operating System, Russ Cox, Franz Koshoek, and Robbert Morris.

Important note

First, if your assignment does not compile, you will get a zero. You are provided with a code that compiles; there is no reason to return a code that does not compile. **Second, console access in the bootloader code is disabled.** Hence, print statements do not work and you must leverage

GDB breakpoints for debugging. Please refer to assignment #0 for a GDB overview.

2 Breakdown of Tasks in this Assignment

2.1 Understanding the QEMU Boot ROM for RISC-V CPU (5 points)

When QEMU boots up a virtual machine (VM) in emulation mode (e.g., the VM you are using to run xv6), it automatically supplies a Boot ROM to the VM (Boot ROM is explained in CSE 536 lecture #3).

Your first task is to **inspect the execution of the supplied Boot ROM and document its process**. To complete this task, you must leverage GDB to interrupt the execution of the VM, single-step through the Boot ROM at an instruction-level, and answer the questions below.

Question-1

(6 points)

(a) At what address is the Boot ROM loaded by QEMU?

(b) At a high-level, what are the steps taken by the loaded Boot ROM?

(c) What address does our Boot ROM jump to at the end of its execution?

2.2 Write a Bootloader Linker Script and Boot into Assembly (10 points)

Our Boot ROM should next jump to the Bootloader at the address found at Q1-(c) (let's call this address *bootloader-start*). QEMU automatically loads the bootloader binary (which is the file bootloader/bootloader) at *bootloader-start*. The bootloader binary must only use addresses higher than *bootloader-start*, otherwise it will corrupt other parts of the system. Thankfully, compilers can easily address this problem with a linker descriptor (.ld) file. In this file, a programmer can specify the addresses where different program sections (and symbols) should be placed.

To complete this task, you must **write a linker descriptor in the file bootloader/bootloader.ld.** Please follow these steps:

- Specify the starting address of the bootloader binary as *bootloader-start*.
- Place different sections of the bootloader in the following order: .text, .data, .rodata, and .bss. Refer to the first suggested reading below to understand program sections.
- Specify the ending addresses of each section and the bootloader binary in the following variables:

  `ecode`: end of code (or text) section

  `edata`: end of data section.

  `erodata`: end of the read-only data section.

  `ebss`: end of the global or bss section.

  `end`: end of the binary program.

Suggested reading(s)

How is a binary executable organized? Julia Evans
CSE 536 Lecture Slides #3 (Slide 24) explains linker descriptors

Question-2

(2 points)

(a) What is the specified entry function of the bootloader in the linker descriptor?

(b) Once your linker descriptor is correctly specified, how can you check that your bootloader's entry function starts to execute after the Boot ROM code?

2.3 Setup a Stack for C code (5 points)

At this point, the assembly function (`_entry` in bootloader/entry.S) is executing after the Boot ROM code. The reason for an assembly entry point is that C functions require a *stack* to start executing, which is not initially set-up. In this task, your goal is to **set-up a stack so that C code can begin executing after your entry function**.

The stack is pre-allocated using the variable `bl_stack` (in the file bootloader/start.c). To complete this task, you must perform 2 steps in entry.S:

- You must load the `bl_stack`'s address inside the RISC-V stack pointer register (called `sp`), which is a per-CPU register. Recall that stack is used from high to low; hence, load the end of `bl_stack`.
- Once the stack is loaded, jump to the C function (`start` in bootloader/start.c).

Question-3

(4 points)

(a) What happens if you jump to C code without setting up the stack and why?

(b) How would you setup the stack if your system had 2 CPUs cores instead of 1?

Question-4

(4 points)

(a) By looking at start.c, can you explain how privilege is being switched from M-mode to S-mode in the `mstatus` register? Explain what fields in the register are being updated and why.

2.4 Setup the RISC-V PMP Feature (15 points)

RISC-V introduced a new CPU feature called physical memory protection (or PMP), which allows an M-mode software to enable memory protection for less privileged software (e.g., executing at S-mode or

U-mode). On initial boot, the PMP is configured to prevent all memory regions from being accessed by the S-mode software. Recall from CSE 536 lecture #2 that the OS kernel executes at S-mode. Hence, we must configure PMP to enable access of memory at S-mode, otherwise the OS kernel will fail to execute.

Suggested reading(s)

RISCV Instruction Set Volume II
Section 3.7 explains PMP, Figure 3.34 explains the format of PMP for RV64 (our system's ISA).

In this task, your goal is to **setup PMP to allow S-mode access to all physical memory of the machine, except for the higher 10 megabytes (MBs)**. In particular, your QEMU VM has 128 MB of physical memory (i.e., 0 - 127 MB), the OS should be allowed to access 0 - 117 MB.

PMP allows 16 different permission regions. For each region, there is a configuration and address register, e.g., region 0 has pmpcfg0 and pmpaddr0. The configuration register specifies memory permissions, while the address register holds the address range where the permissions apply.

To complete this task, you must perform the following steps:

- Read the RISC-V CPU manual to understand PMP. For instance, you need to know what bits of pmpcfg0 correspond to and how to specify the memory address in pmpaddr0.
- *pmpcfg0 register setup*. Set read, write, and execute permission bits. Also set the A field in the register to top-of-range (TOR). Check Table 3.10 in the RISC-V manual for setting the A field.
- *pmpaddr0 register setup*. Specify the highest physical address accessible to the OS in this register.

Tidbit

- In QEMU, physical addresses of the DRAM start from *bootloader-start*, not from 0x0. Hence, the highest accessible physical address would be *bootloader-start* + 128 MB, if we made all memory regions accessible to the OS.
- pmpaddr0 does not use the full 56-bit physical address.

Important note

Even if your PMP configuration is incorrect, it is possible that your kernel loads up (after completing subsection 2.5). Hence, once you are done with subsection 2.5, run `make qemu-kernel-pmp` to check if your PMP settings are correct or not.

2.5 Load User-Selected OSs (35 points)

Bootloaders allow a user to boot up different OSs on-demand. In this task, your goal is to **set-up the functionality such that any of the provided OS kernel binaries can be run on the system**.

Initially, you have been supplied with three different kernel binaries (kernel1, kernel2, and kernel3). The machine can be configured to run with either of these kernels using the command `make qemu-kernel[1-3]`. You must write code to identify which kernel is being asked to run and run it.

2.5.1 Read the Kernel Binary Header (10/35 points)

On `make qemu-kernel[1-3]`, QEMU will load the relevant kernel binary at address, 0x84000000 (also defined as RAMDISK in bootloader/defs.h). However, these kernels are not designed to run at this address (refer to subsection 2.2 to understand how a linker descriptor specifies the addresses used by the program). Thankfully, a binary already has the information specifying its range of addresses in its *ELF and program headers* (typically the first 4KB page of the binary).

Suggested reading(s)

CSE 536 Lecture Slides #3 (slide 25) explains ELF and program headers

In this task, your goal is to **read the headers and determine the starting address where the kernel should be loaded.** Let's call this address *kernload-start*.

For all our kernel binaries, *kernload-start* is the starting location of .text (or code) section in the binary. Please follow the steps outlined below to find this address. You must write the code in the function `read_kernel_load_addr` in the file bootloader/elf.c.

- In bootloader/elf.h, there are structs which correspond to the ELF and program section headers. Point an ELF struct (elfhdr) to RAMDISK (where the kernel is currently loaded) to initialize it with the kernel binary's ELF header.
- Grab the offset at which program header sections are specified (field `phoff` in the elfhdr) within the kernel binary and the size of the program headers (field `phsize` in the elfhdr).
- Navigate to the program header section's second address. This is the header for the .text section and its address is RAMDISK + phoff + phsize. Point a program header struct (proghdr) to this address to initialize it with the .text section's header.
- Find the starting address of the .text section by retrieving the `vaddr` field within proghdr.

Tidbit

You can use the `riscv64-unknown-elf-readelf` command to read the headers of the kernel binary. The `-a` flag will give you all headers, while the `-h` flag will only give you the ELF header. Use this to verify if you are retrieving the correct headers within your code. (Use GDB inside the kernel)

2.5.2 Copy the Kernel Binary (20/35 points)

In this task, your goal is to **copy the kernel binary to *kernload-start.***

Please follow these steps:

- Write code to find the size of the kernel binary using its ELF headers in `find_kernel_size` (located in bootloader/elf.c).
  Hint: Check the kernel binary size (e.g., for kernel1) on your filesystem and think about how you can obtain the same size using the information in the ELF headers.
- In bootloader/load.c, a function (`kernel_copy`) copies the kernel binary (currently at RAMDISK) to other memory regions. This function accepts a `buf` struct (defined in bootloader/buf.h) as argument.

In `start`, use this function to copy the kernel binary to *kernload-start*.
Hint: You should exclude the kernel headers when copying the binary.

Question-5

(4 points)

(a) How does `kernel_copy` work? Please document its steps.

2.5.3 Execute the OS Kernel (5/35 points)

In this task, your goal is to **jump to the entry function in the kernel.**

Please follow these steps:

- Write code to find the kernel entry function address in the `find_kernel_entry_addr` (bootloader/elf.c). Hint: This address is specified in the kernel ELF header (subsubsection 2.5.1).
- In `start`, specify the kernel entry function address in the `mepc` register and execute the `mret` instruction. The start function specifies that the system should switch privilege into S-mode (recall from Q4). This switch will only happen when the system executes the `mret` instruction. The location that the system will jump at this transition depends on the address in the `mepc` register.

If you are successful in completing all steps till now, you should see the following printed on screen when you execute `make qemu-kernel1`:

[*] Success! The xv6 kernel-1 is booting! Yayyy!

Run `make qemu-kernel2` and `make qemu-kernel3` to make sure the other kernels boot-up correctly too.

2.6 Make System Information Accessible to the Kernel (10 points)

In this task, your goal is to **provide several system information to the kernel4.**

Important note

Please run `make qemu-kernel4` for this step. It will print out the kernel information you provided.

The needed system information is specified by the `struct sys_info` in bootloader/start.c. We must create and populate the struct's entries at 0x80080000 (let's call it SYSINFOADDR). Once loaded, the OS kernel (kernel4) will read the system information from that address.

Please follow these steps:

- Define a `struct sys_info` that points to SYSINFOADDR.
- Specify the starting and ending address of the bootloader in the struct. This information was already saved when we created the bootloader's linker descriptor in subsection 2.2.
- Specify the starting and ending address of the DRAM in the struct.

- Specify the hardware architectural feature IDs for your system. You must write three new functions in bootloader/riscv.h (namely `r_vendor`, `r_architecture`, `r_implementation`) to read vendor ID, architectural ID, and implementation ID from RISC-V system registers.
  Hint: check the RISC-V manual to find out the needed registers.

3 Miscellaneous

3.1 GIT Diff of Changes Required

Provided below is a sample of how many changes would be required to complete this assignment. Note that this would vary based on code styles.

```
bootloader/bootloader.ld | 22 ++++++++++++++++++++–
bootloader/elf.c | 19 ++++++++++++++++—
bootloader/entry.S | 9 ++++++++-
bootloader/load.c | 4 +—
bootloader/riscv.h | 22 ++++++++++++++++——
bootloader/start.c | 39 +++++++++++++++++++++++++++++++——–
9 files changed, 92 insertions(+), 23 deletions(-)
```

3.2 Submitting your Assignment

1. Please zip the entire initial provided code directory (including bootloader, kernels, and Makefiles) and submit it to the Canvas under “Assignment 1: CODE”.
2. Please create a PDF document and upload it to canvas under “Assignment 1: DOCUMENT”.