CSE 536: Advanced Operating Systems
Assignment 1: A Tale of a Boot ROM and a Bootloader
(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, you will **understand the boot process of an operating system using QEMU and write a custom bootloader for the xv6 OS kernel**.

GitHub Repo. Please clone as follows:

git clone -b lab1-bootloader https://github.com/ASTERISC-Teaching/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.

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 Inspecting QEMU Boot ROM

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.

Important note

Please use **./run gdb** to execute xv6 in debug mode. In a seperate terminal window, please execute **riscv64-unknown-elf-gdb** to start debugging with GDB.

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 (5 points)

After execution, Boot ROM jumps to the address found at Q1-(c) (let's call this address *bootloader-start*). QEMU automatically loads the bootloader binary (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 address this problem with a linker descriptor, in which a programmer specifies the addresses where program sections (and symbols) are placed.

Your task is to **write a linker descriptor in 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 also explain 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)

If the linker descriptor is correct, the system should jump to `_entry` in bootloader/entry.S. The reason for an assembly entry point is that C functions require a *stack* to start executing, which is not initially set-up.

Your task is to **set-up a stack so that C code can execute after `_entry`**. The stack is pre-allocated using the variable `bl_stack` (in the file bootloader/start.c). In `_entry`, follow the steps below:

- Load `bl_stack`'s address inside the 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`.
- Jump to the `start` function (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 CPU cores instead of 1?

Question-4

(4 points)

(a) By looking at the `start` function, 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 Load User-Selected OSs (30 points)

Bootloaders allow a user to boot up different OSs. Your task is to **set-up the functionality such that any of the provided OS kernel binaries can be booted**.

For this task, you are concerned with 3 kernel binaries (kernel[1-3]). The VM can be configured to boot with any of these kernels using: **./run kernel[1-3]**. QEMU will load the selected binary at address 0x84000000 (defined as RAMDISK in bootloader/defs.h).

2.4.1 Find the Kernel Load Address (5/30 points)

Even though the selected kernel is loaded at RAMDISK, it is compiled to run at a different address (we used a linker descriptor like in subsection 2.2 to randomize the kernel load address of each provided kernel). Thankfully, a binary already has the information specifying its range of addresses in its *ELF and program headers* (typically the first 4KB of the binary).

Suggested reading(s)

CSE 536 Lecture Slides #3 explains ELF and program headers

Your task 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

Outside of xv6, 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 xv6. (Use GDB inside the kernel)

2.4.2 Copy the Kernel Binary (15/30 points)

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

- Find the size of the binary using its ELF headers in `find_kernel_size` (located in bootloader/elf.c). Hint: Check the kernel binary size (e.g., `ls -al kernel`) 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: Remember to exclude atleast the kernel ELF header (first 4KB) when copying the binary.

Question-5

(4 points)

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

2.4.3 Execute the OS Kernel (5/30 points)

In this task, you will **jump to the entry function in the kernel.**

- 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 2.4.1).
- In `start`, specify the kernel entry function in the `mepc` register and execute the `mret`.
  start already specifies that the system should switch privilege into S-mode (recall Q4) when `mret` is executed. The location that the system will jump at this switch is the address in the `mepc` register.

Important note

When executing **./run kernel[1-3]** at this point, you should see statements printed on the screen similar to the ones provided in **outputs/kernel[1-3]**.

2.4.4 Pass System Information to Kernel (5/30 points)

Your task is to **provide system information to a booted kernel.** The bootloader must save this information at 0x80080000 (SYSINFOADDR). The kernel will read system information from SYSINFOADDR.

- Create 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.

Important note

Run **./run kernel3** and compare its output with **outputs/kernel3**. Ignore `expected_measurement` and `observed_measurement` fields of `struct sys_info` (they will all be 0 at this time).

2.5 Setup the RISC-V PMP Feature (20 points)

RISC-V's physical memory protection (PMP) allows an M-mode software to isolate memory from 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. Since the OS kernel executes at S-mode, we must configure PMP to enable memory access at S-mode, otherwise the OS will fail to execute.

PMP allows 16 different permission regions. For each region, there is a configuration and address register, e.g., pmpcfg0 and pmpaddr0. The configuration register specifies permissions and what configuration to use (explained below), while the address register holds the address range where the permissions apply.

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).

2.5.1 Use the TOR Configuration (10/20 points)

In the TOR configuration of PMP, a region is defined by a base address and a top-of-range (TOR) address. The TOR defines the upper limit of the range protected by that PMP region.

Your task is to **use TOR and isolate the upper 11 megabytes (MBs)**. In particular, your QEMU VM has 128 MB of physical memory, the OS should only be allowed to access 0 - 117 MB.

Follow the steps below and write the code under "#if defined(KERNELPMP1)" in `start`.

- *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.
- pmpaddr[0-15] registers do not use the full 56-bit physical address.

Important note

Execute **./run kernelpmp1** and compare with **outputs/kernelpmp1**.

2.5.2 Use the NAPOT Configuration (10/20 points)

NAPOT stands for "Naturally-Aligned Power Of Two". The NAPOT configuration allows you to specify a range of memory addresses by providing a base address and a size.

Your task is to **use NAPOT and isolate regions 118-120MB and 122-126MB**.

To complete this task, you will need to leverage 5 PMP address registers (*pmpaddr[0-4]*). Follow the steps below and write the code under "#if defined(KERNELPMP2)" in `start`.

- Use the TOR configuration on the first register to setup access till 118MB (like previous section).
- Write PMP address write functions for registers 1-4 (i.e., `w_pmpaddr[1-4]`) in riscv.h.
  Hint: Check how `w_pmpaddr0` is written.
- Using *pmpaddr[1-4]*, enable NAPOT-based protection for regions 118-120MB and 122-126MB.
  Hint: The lower bits of the pmpaddr encode the size of the NAPOT memory range, while the upper bits encode the significant bits of the start address.
- Figure out how to write to the corresponding PMP config registers, with the NAPOT bits set.

Important note

Execute **./run kernelpmp2** and compare with **outputs/kernelpmp2**.

2.6 Enable Secure Boot (20 points)

Secure boot allows the bootloader to verify that a *trusted* kernel binary is loaded (instead of a tampered one). Your task is to **enable a custom secure boot process.**

Our secure boot process works as follows. A user pre-records a SHA-256 hash ($hash_{exp}$) of the kernel binary in the bootloader. During boot, the bootloader reads the kernel and generates a new SHA-256 hash ($hash_{obs}$). If ($hash_{exp}$!= $hash_{obs}$), the bootloader retrieves and executes a *recovery kernel.*

Follow the steps below:

- Initialize $hash_{obs}$ using `sha256_init`.
- Update $hash_{obs}$ using `sha256_update`. Pass as argument the entire kernel binary.
  (Recall the kernel is loaded at RAMDISK)
- Finalize $hash_{obs}$ (using `sha256_final`) and compare it with $hash_{exp}$.
  $hash_{exp}$ is stored as `trusted_kernel_hash` in bootloader/measurements.h.
- If $hash_{exp}$!= $hash_{obs}$, load the recovery kernel.
  This kernel is loaded at 0x84500000 (stored as RECOVERYDISK in bootloader/layout.h).
  Follow steps in subsection 2.4 to load the kernel.

Important note

After the completion of this task, **./run kernel[1-3,pmp1,pmp2]tamper** should produce outputs specified in **outputs/kernel[1-3,pmp1,pmp2]tamper**.

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.

xv6-riscv/bootloader/bootloader.ld | 23 ++++++++++++++++++++-

xv6-riscv/bootloader/elf.c | 33 +++++++++++++++++++++++++++++++—

xv6-riscv/bootloader/entry.S | 10 +++++++++-

xv6-riscv/bootloader/riscv.h | 24 ++++++++++++++++++++++

xv6-riscv/bootloader/start.c | 81

++—————

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".
   Please do not include your QEMU or RISC-V toolchain!

2. Please create a PDF document and upload it to canvas under "Assignment 1: DOCUMENT".