南開大學
Nankai University

密码与网络空间安全学院
操作系统第一次实验实验报告

姓名：邹佳衡

学号：2312322

专业：信息安全

2025 年 10 月 9 日

1 实验介绍

1.1 任务描述

实验 1 主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识, 以及通过 opensbi 固件来通过服务。

1.2 实验环境

在 VMware Ubuntu 虚拟机上，下载 riscv64-unknown-elf-gcc 预编译工具链，qemu-4.1.1 模拟器。

2 环境配置

2.1 安装编译器

首先，我选择在 VMware 上配置 Ubuntu18.04.6 版本的虚拟机，在虚拟机上进行实验。

完成虚拟机的配置后，接下来进行 riscv-gcc 编译器的安装，为了方便，这里直接选择使用预编译工具链，进入 https://github.com/sifive/freedom-tools/releases 后选择符合我们版本的工具链，如下图所示。

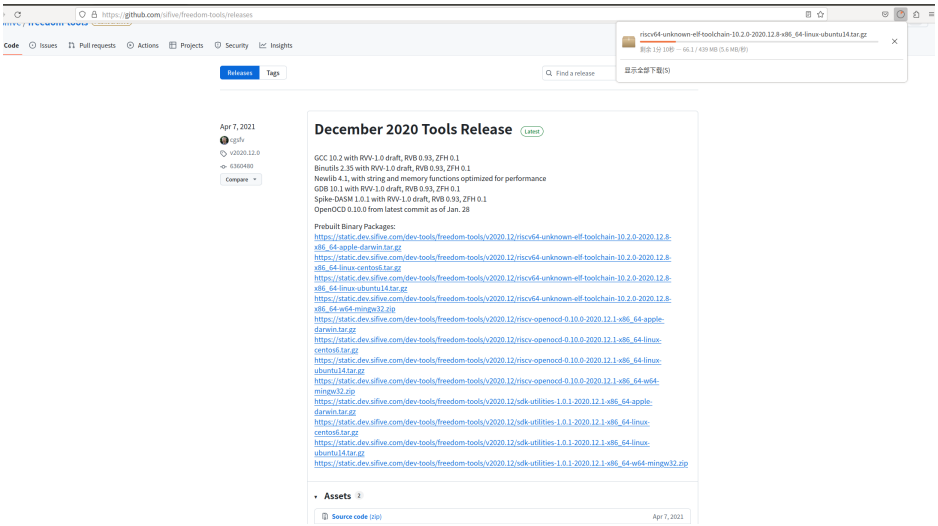

图 1: 下载实验所需的预编译工具链

下载完后，将它放入提前准备好的文件夹中，记录下文件夹路径，这里要注意文件夹路径里不能包含中文，将其解压后，输入命令 gedit /.bashrc 打开.bashrc 文件，在最下方添加两行命令:

export RISCV=PATH_TO_INSTALL

export PATH=$RISCV/bin:$PATH

其中，PATH_TO_INSTALL 是解压后文件夹的地址，如下图所示:

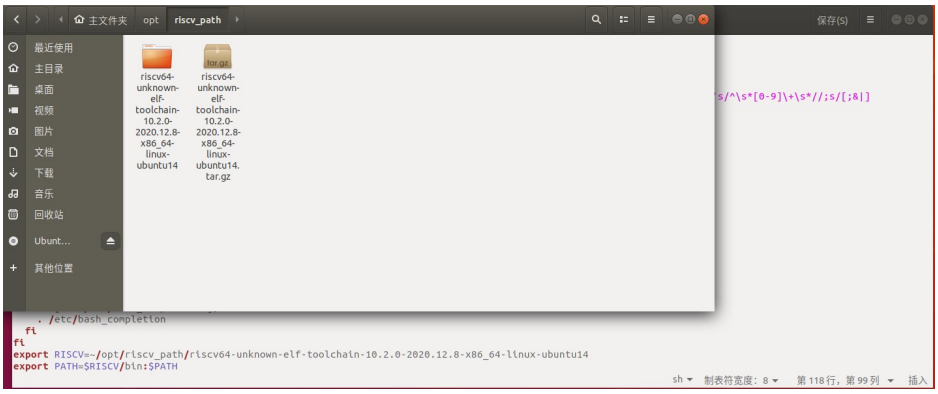

图 2: 配置路径

保存后，在终端运行命令 source ~/.bashrc 即可使配置生效。

2.2 安装模拟器

随后，我们进行模拟器 qemu 的安装，所谓模拟器，就是在 x86 架构的计算机上，用软件模拟出一台 riscv64 架构的计算机，从而能够运行 riscv64 的目标代码。依次输入以下命令：

wget https://download.qemu.org/qemu-4.1.1.tar.xz

tar xvJf qemu-4.1.1.tar.xz

cd qemu-4.1.1

./configure –target-list=riscv32-softmmu,riscv64-softmmu

make -j

sudo make install

完成后，我们检查一下 riscv-gcc 和 qemu 是否真的安装好了，并且保证 qemu 版本在 4.1.0 以上。

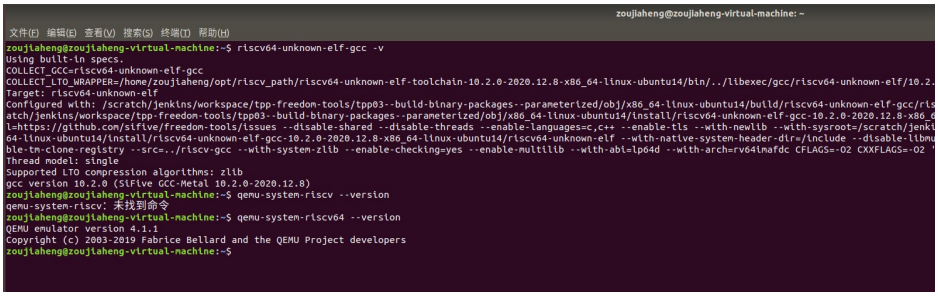

图 3: 实验环境

2.3 运行 openSBI

新版 Qemu 中内置了 OpenSBI 固件（firmware），它主要负责在操作系统运行前的硬件初始化和加载操作系统的功能。我们使用以下命令尝试运行一下：

qemu-system-riscv64 \

–machine virt \

–nographic \

–bios default

图 4: openSBI 试运行

至此，所有实验环境都配置完成。

3 练习一

首先，我们打开文件，看到 entry.S 代码如下所示：

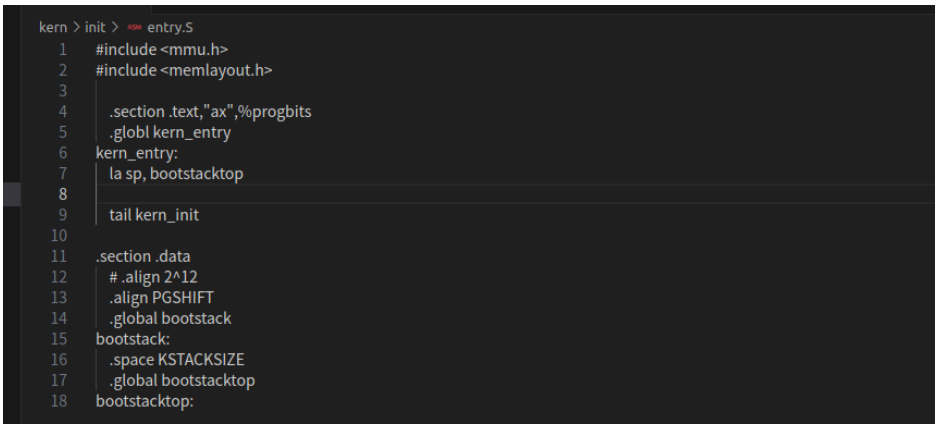

图 5: entry.S 代码

可见需要我们分析的指令在第七行，la sp,bootstacktop 其中 la 是加载指令，sp 是寄存器，bootstacktop 是栈顶地址，可见这条指令是要把栈顶地址加载到 sp 寄存器中，很可能是将栈顶地址记录下来开辟一块新的栈帧，从而为接下来 kern_init 函数的运行做好准备。

tail kern_init 是跳转到对应的函数处执行代码，而这个函数就在 init.c 文件里，具体的代码和解析如下图所示：

```c
kern > init > C init.c > ...
#include <stdio.h>
#include <string.h>
#include <sbi.h>
int kern_init(void) __attribute__((noreturn)); //设置一个没有返回的函数，是操作系统内核的模拟程序

int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);  //将.bss段的数据清零

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);  //输出字符串
    while (1) //无限循环，在实际完整内核中，这里会被替换为调度器或空闲任务
        ;
}
```

图 6: init.c 代码

这是一个模拟内核操作系统的函数，它包括三部分，首先通过 memset 将模拟的.bss 段数据清零，再输出字符串 (THU.CST)os is loading... 表明我们成功加载了这个函数，然后陷入空循环中。在完整的内核中，空循环会替换成调度器的循环，kern_init 还会承担内核初始化的任务，这个程序中没有明显体现。在执行 kern_init 前，程序会先执行 kern_entry 函数，它的代表指令就是练习一里那两条。

4 练习二

为了熟悉使用 QEMU 和 GDB 的调试方法，我们要使用 GDB 跟踪 QEMU 模拟的 RISC-V 从加电开始，直到执行内核第一条指令（跳转到 0x80200000）的整个过程。

首先，在 Lab1 文件夹内从终端输入指令 make debug ，makefile 文件会告诉 make 指令如何编译和链接程序，在这一步后会多出几个文件夹。

然后，打开一个新的终端，输入指令 make gdb 开始分析汇编指令，输入 x/10i $pc 可以显示该地址后的十行指令，如下图所示：

```
For help, type "help".
Type "apropos word" to search for commands related to "word".
Reading symbols from bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using localhost:1234
0x0000000000001000 in ?? ()
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a1,t0,32
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      t0,24(t0)
   0x1010:      jr      t0
   0x1014:      unimp
   0x1016:      unimp
   0x1018:      unimp
   0x101a:      0x8000
   0x101c:      unimp
(gdb) si
0x0000000000001004 in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x0000000000001008 in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x000000000000100c in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si
0x0000000000001010 in ?? ()
(gdb) info r t0
t0             0x80000000       2147483648
(gdb)
```

图 7: 开头十行汇编指令

这就是最初执行的几条指令，它们在地址 0x1000 附近，首先是将 pc 也就是当前地址与立即数 0 组合后存入 t0，此时 t0=0x1000，然后是将 t0+32 的结果 0x0020 赋给 a1，然后获取进程的 id 值赋

给 a0，之后将 t0+24 的地址处也就是 0x1018 处加载一个双字（64 位）到 t0 处，最后跳转到 t0 储存的地址处继续执行。

接下来，输入 si 单步执行代码，直到 t0 存储的数据变化，可以看到接下来程序会到 0x80000000 处执行，我们输入 x/10i 0x80000000，可以看到 0x80000000 处后方十条指令的内容，如下图所示：

图 8: 0x80000000 处的汇编指令

随后输入命令 break *0x80200000 可以在此地插入一个断点，从返回的结果看，此处是 kern_entry 的第一条指令 la sp,bootstacktop，然后输入 continue 即可一直运行到断点处，此时 openSBI 被启动，如下图所示：

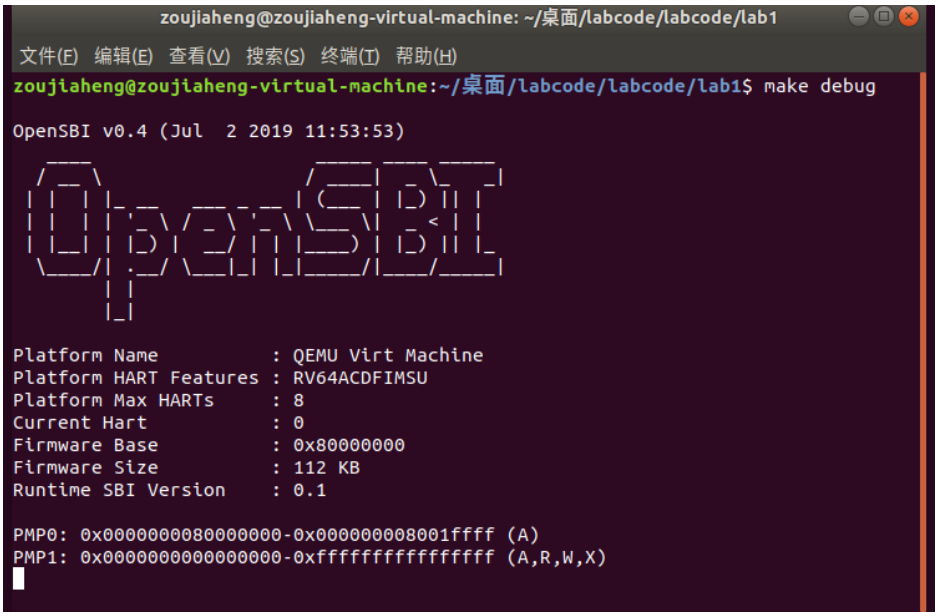

图 9: openSBI 被启动

然后，如果我们输入 x/10 0x80200000, 不难看出在 kern_entry 后面就是 kern_init。

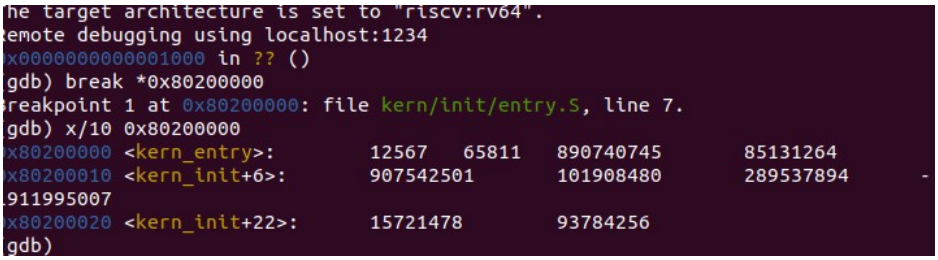

图 10: 0x80200000 处的代码

随后逐步执行指令，我们能看到练习一的两条指令被执行，还可以看到栈顶在什么位置。最后，一次执行所有指令，程序输出字符串并陷入循环。

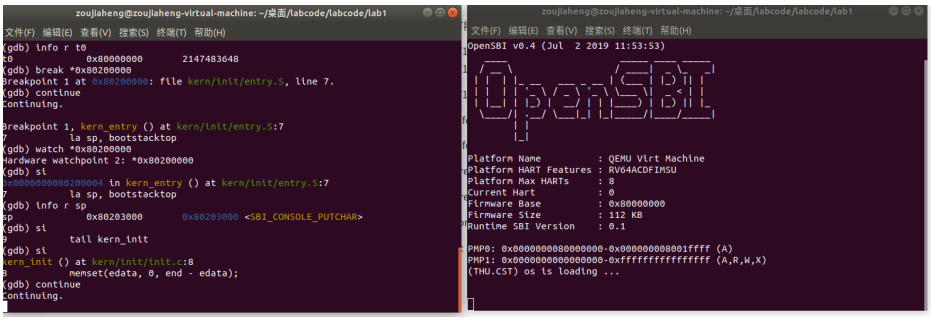

图 11: 程序运行结果