Lab1

一、实验目的

本实验旨在深入理解基于 RISC-V 架构的计算机系统启动流程与内核构建基础。通过本次实验掌握 Qemu 模拟器作为 RISC-V 硬件平台的工作原理，特别是其初始状态与内核加载机制。实现一个通过 OpenSBI 固件服务正确引导并运行在 Qemu 上的最小内核，从而建立对操作系统底层启动，硬件抽象层和固件交互机制的初步认知。为后续操作系统核心功能的开发奠定实践与理论基础。

二、实验要求

报告基于Markdown格式完成，以文本方式为主，针对每个基本练习（练习如下）完整填写要求内容。总结本实验重要知识点及其与OS原理对应知识点的含义、关系、差异理解；列出OS原理中重要但实验未覆盖的知识点。在目录labcodes/lab1下存放报告，每个小组建Gitee或GitHub仓库，通过git push上传代码和报告，按时提交。

练习1：理解内核启动中的程序入口操作

阅读 kern/init/entry.S内容代码，结合操作系统内核启动流程，说明指令 la sp, bootstacktop 完成了什么操作，目的是什么？ tail kern_init 完成了什么操作，目的是什么？

练习2: 使用GDB验证启动流程

为了熟悉使用 QEMU 和 GDB 的调试方法，请使用 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。通过调试，请思考并回答：RISC-V 硬件加电后最初执行的几条指令位于什么地址？它们主要完成了哪些功能？请在报告中简要记录你的调试过程、观察结果和问题的答案。

三、实验内容

（1）本实验中重要的知识点，以及与对应的OS原理中的知识

1、内核启动流程

含义：从硬件复位 (PC=0x1000 执行 MROM) 开始，跳转到 OpenSBI (0x80000000) 初始化硬件并加载内核 ELF 到 0x80200000，然后 entry.S 设置栈并调用 kern_init 输出信息进入死循环。

OS 原理对应：通用启动过程 (BIOS/UEFI → Bootloader → Kernel Entry)。

关系：二者关系密切，都是从固件到内核的特权级切换 (M-mode → S-mode 类似 x86 Real Mode → Protected Mode)，确保硬件就绪后移交控制。差异在于 RISC-V 的 SBI 接口更标准化 (ecall 陷阱)，而 x86 依赖 BIOS INT 中断。

2、链接脚本与内存布局

含义：指定 ELF 输出架构 (OUTPUT_ARCH(riscv))、入口 (ENTRY(kern_entry)) 和段布局 (.text at 0x80200000, .rodata, .data, .bss with ALIGN(0x1000))，确保代码地址相关性。

OS 原理对应：内核链接和内存布局 (Sections: Code, Data, BSS)。

关系：直接对应，链接脚本实现原理中的段隔离 (e.g., .text 只读执行)，关系为工具实现抽象 (ld 使用脚本生成位置无关代码)。差异：实验固定地址 (0x80200000) 为 QEMU 硬编码，原理支持动态加载 (e.g., Linux kASLR 随机化地址)。

3、交叉编译

含义：在 x86 Ubuntu 上用 RISC-V 工具链编译 ELF (make → objcopy → bin/ucore.img)，生成 QEMU 可加载镜像。

OS 原理对应：交叉编译在嵌入式 OS 开发中。

关系：原理中交叉编译是嵌入式 OS (e.g., RTOS) 标准实践，关系为工具链支持多架构 (e.g., GCC backend)。差异：实验用 prebuilt 链 (SiFive)，原理涉及自编译 (e.g., buildroot)。

4、Bootloader 与 SBI 调用

含义：OpenSBI (M-mode) 加载内核，提供服务 (SBI_CONSOLE_PUTCHAR via ecall)，封装为 sbi_call 内联汇编。

OS 原理对应：Bootloader (GRUB) 和系统调用 (syscall Trap)。

关系：高度相似，ecall 如 syscall (U→S trap)，OpenSBI 如微内核 (提供接口而不实现)。差异：RISC-V SBI 标准化 (v1.0)，x86 用 INT 80h/syscall 指令。

5、GDB 调试

含义：用 -s -S 暂停 QEMU，GDB 连接 1234 端口，break kern_entry/stepi/info reg 跟踪。

OS 原理对应：内核调试 (KGDB/KDB)。

关系：直接对应，远程调试如 KGDB over Ethernet。差异：实验用 QEMU 模拟 (virt 机)，原理支持真实硬件 (JTAG)。

（2）未提及的知识点

1、进程管理与调度：原理中 fork/exec 创建进程，调度器 分配 CPU 时间。实验是单线程死循环，缺失多任务上下文切换

2、虚拟内存与分页 ：原理中 MMU/TLB 实现地址翻译、需求分页。实验仅 memlayout.h 定义物理布局，无 mmu.h 实现，缺失页表走查。

（3）练习1

1. 指令 la sp, bootstacktop

将内核预先定义的栈顶地址（bootstacktop）加载到栈指针寄存器（sp）中。（la）是“Load Address” 的缩写，功能是把标签对应的内存地址计算后写入目标寄存器）

目的：为内核初始化阶段建立**合法的栈空间**。内核刚从引导程序（如 BIOS/UEFI）进入时，sp 寄存器值是无效的（无栈可用），而后续执行代码（尤其是函数调用）必须依赖栈保存临时数据、返回地址等。这条指令通过指定内核自己的栈（bootstack 是提前在内存中预留的栈区域，bootstacktop 是其栈顶），让 CPU 获得可正常使用的栈，为后续代码执行（如调用 kern_init）铺路。

2. 指令 tail kern_init

以 “尾调用” 方式跳转到 kern_init 函数执行，且不保存当前函数的返回地址到栈中。（区别于普通 jal 指令：jal 会把下一条指令地址存入 ra 寄存器，而 tail 仅跳转，不修改 ra）

目的：在进入内核 C 代码（kern_init 是内核初始化的 C 入口函数）时，**避免栈空间浪费并保证执行流连续性**。entry.S 是内核启动的汇编入口，其任务就是完成硬件初始化（如建立栈、关闭中断）后，把执行权交给 C 代码。由于 entry.S 后续无代码可执行，无需保存返回地址，用 tail 跳转可省去栈空间占用，同时直接将执行流无缝切换到 kern_init，让内核从汇编初始化阶段正式进入 C 语言的初始化逻辑（如初始化内存、设备、进程管理等）。

（4）练习2

tmux是一个终端复用器。在一个终端窗口中，可以使用命令分割出多个窗格，同时进行不同的操作。本次实验在终端输入tmux进入一个终端复用会话。然后使用命令Ctrl+B，然后按%分割出左右两个窗口。如图1所示，在左边窗口输入make debug，该命令启动qemu模拟器并监听端口1234。

在另一个窗口输入make gdb命令，该命令启动GDB调试器并连接正在运行的qemu模拟器。 ‘file bin/kernel’ 加载内核文件。 ‘set arch riscv:rv64’ 设置目标架构为RISCV 64位。 ‘target remote localhost:1234’ 连接到本地主机端口1234。

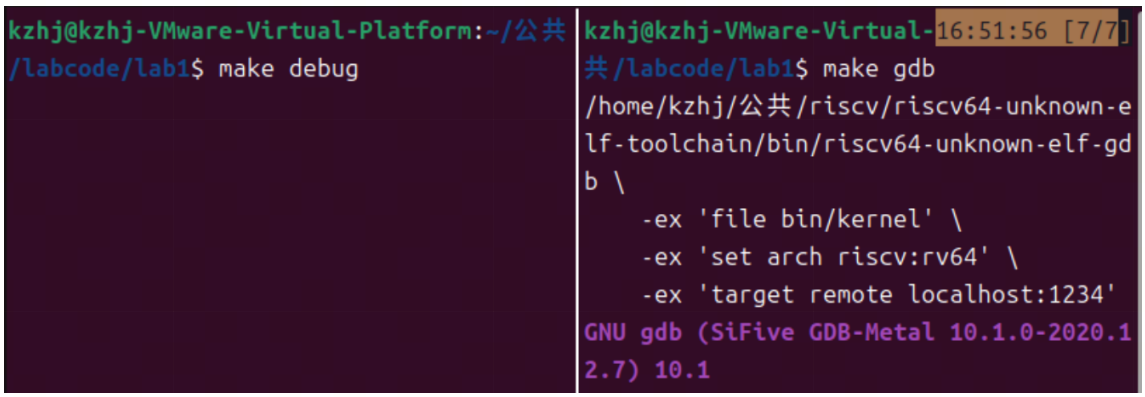

图1

gdb连接到qemu模拟器后，使用命令i r显示当前所有寄存器的值。如图2所示，观察到当前pc值为0x1000。RISCV加电后，此时PC被重置到0x1000地址开始执行。从该地址开始执行一小段固件代码。

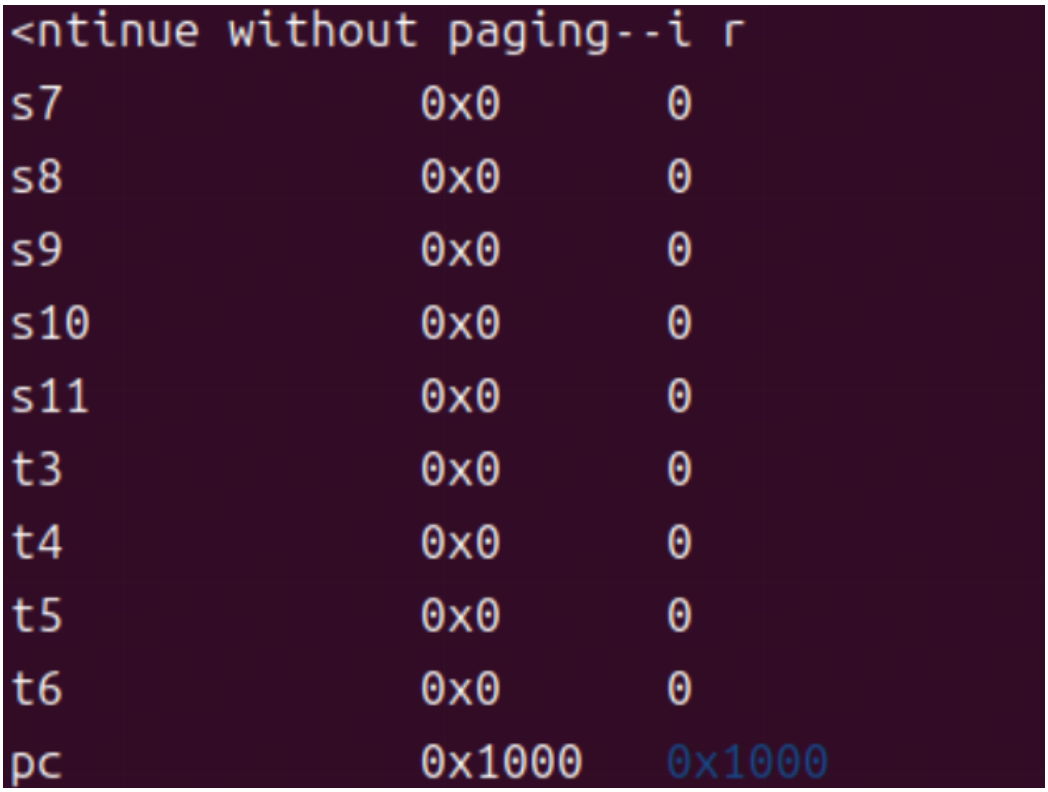

图2

使用命令x/10i $pc显示从当前pc地址开始的后面10条汇编指令。汇编代码如下图3所示，首先auipc指令将当前pc的高20位与立即数0左移 12 位相加，结果存入t0，此时t0=0x1000。addi指令实现将t0+32的结果存入a1，此时a1=0x1020。csrr指令读取mhartid寄存器（RISC-V 的硬件线程 ID 寄存器）的值到a0，传递当前核心的ID。a0，a1是下一阶段函数调用的参数。

ld指令实现从t0+24(即0x1018)处加载数据到t0中。0x1018处为unimp，表示未实现，存储的数据为0x0000。从该地址读取8字节，所以读取的数据为0x80000000(RISCV采用小端字节序)。最后jr跳转到t0存储的地址处执行，即跳转到OpenSBI加载的地址处。

```
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
```

图3

通过分析0x1000后的固件代码，该阶段代码功能为模拟加电复位过程，pc设置为固定复位地址0x1000，从0x1000处开始执行固件代码，然后将控制权转交给OpenSBI。使用命令b* 0x80000000在加载OpenSBI地址处设置断点，接着输入命令c执行到断点0x80000000，i r命令展示出pc寄存器的值如图4所示，为0x80000000。

```
t3             0x0        0
t4             0x0        0
t5             0x0        0
t6             0x0        0
pc             0x80000000       0x80000000
```

图4

输入命令x/15i $pc显示OpenSBI的部分功能代码如图5所示，该阶段代码功能为完成OpenSBI初始化，为内核准备运行环境，将内核加载到内存的0x80200000中。

```
(gdb) x/15i $pc
=> 0x80000000:  csrr    a6,mhartid
   0x80000004:  bgtz    a6,0x80000108
   0x80000008:  auipc   t0,0x0
   0x8000000c:  addi    t0,t0,1032
   0x80000010:  auipc   t1,0x0
   0x80000014:  addi    t1,t1,-16
   0x80000018:  sd      t1,0(t0)
   0x8000001c:  auipc   t0,0x0
   0x80000020:  addi    t0,t0,1020
   0x80000024:  ld      t0,0(t0)
   0x80000028:  auipc   t1,0x0
   0x8000002c:  addi    t1,t1,1016
   0x80000030:  ld      t1,0(t1)
   0x80000034:  auipc   t2,0x0
   0x80000038:  addi    t2,t2,988
```

图5

再次设置断点b*kern_entry，kern_entry与内核入口地址0x80200000对应。接着输入指令c执行到该断点。如图6所示，此时观察到pc的值变化为内核的入口地址0x80200000.

```
t4             0x0      0
t5             0x0      0
t6             0x82200000       2183135232
pc             0x80200000       0x8020<ntinue without pagi
ng--
0000 <kern_entry>
```

图6

执行到此处左边窗口输出结果如图7所示，该结果显示平台名称为qemu虚拟机，平台最大核心为8，当前核心数为0.固件基址为0x80000000，大小为112KB。该输出结果说明OpenSBI 固件已正常启动并完成硬件初始化。系统现在正等待加载和启动操作系统内核。

```
OpenSBI v0.4 (Jul  2 2019 11:53:53)
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name          : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs     : 8
Current Hart           : 0
Firmware Base          : 0x80000000
Firmware Size          : 112 KB
Runtime SBI Version    : 0.1

PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
```

图7

如图8所示，当CPU跳转到内核入口点时，输入命令si单步执行命令，跟踪执行过程。首先执行ld sp，bootstacktop。该命令初始化内核栈，为c函数调用做准备。接着执行tail kern_init命令，跳转到

init.c中执行函数kern_init。

```
(gdb) si
0x0000000080200004 in kern_entry ()
    at kern/init/entry.S:7
7           la sp, bootstacktop
(gdb) si
9           tail kern_init
```

图8

在kern_init函数中调用函数cprintf输出图9中的结果：(THU.CST) os is loading ... 。表明操作系统正在加载，并陷入while循环中。综合以上分析，该阶段代码功能为内核的启动过程，首先建立内核栈，为函数调用建立空间，准备C语言运行环境。然后跳转到C函数kern_init中输出信息表示内核加载启动成功。

```
PMP0: 0x0000000080000000-0x000000008001ffff (A)
PMP1: 0x0000000000000000-0xffffffffffffffff (A,R,W,X)
(THU.CST) os is loading ...
```