《操作系统》实验报告

实验1：最小可执行内核

蒋枘言 2313546

一、练习1：理解内核启动中的程序入口操作

1. 指令`la sp, bootstacktop`

这条指令将符号 `bootstacktop` 的地址加载到寄存器 `sp` 中。 `bootstacktop` 代表**栈顶的初始位置**，这个操作的目的是为内核**初始化栈指针**，使接下来的C函数（`kern_init`）能够正常使用栈。

2.指令`tail kern_init`

这条指令跳转到 `kern_init` 函数，并且不返回。这个操作的目的是汇编启动阶段结束，**开始真正的内核逻辑**，正式进入内核C代码执行阶段。 `kern_init` 函数定义在 C 代码中`kern/init/init.c`。

二、练习2：使用GDB验证启动流程

1. 调试过程

首先直接执行`make qemu`，得到如下结果：

```
jry@jry-VMware-Virtual-Platform: ~/OSLab/labcode/lab1
jry@jry-VMware-Virtual-Platform:~/OSLab/labcode/lab1$ make qemu
+ cc kern/init/entry.S
+ cc kern/init/init.c
+ cc kern/libs/stdio.c
+ cc kern/driver/console.c
+ cc libs/printfmt.c
+ cc libs/readline.c
+ cc libs/sbi.c
+ cc libs/string.c
+ ld bin/kernel
riscv64-unknown-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img

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)
(THU.CST) os is loading ...
```

发现成功显示了`(THU.CST) os is loading ...`，表明内核启动成功，下一步就是从OpenSBI跳转到0x80200000（内核入口）执行 `kern_entry`。接下来需要研究RISC-V从加电开始，到执行内核第一条指令（跳转到 0x80200000）的整个过程，我们使用GDB工具来完成。

加电复位阶段

在lab1文件夹里打开两个终端（tmux工具可以将终端分割成两栏，但是我目前还没用习惯，这次暂时不用），称为终端1和终端2。终端1中执行命令`make debug`。这个命令会启动QEMU，终端2中执行`make gdb`。现在程序停在了地址0x1000这个地方。也就是说，当“CPU刚上电”时，程序计数器PC的初始值就是0x1000。

```
GNU gdb (SiFive GDB-Metal 10.1.0-2020.12.7) 10.1
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-linux-gnu --target=riscv64-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://github.com/sifive/freedom-tools/issues>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

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)
```

接下来在终端2中继续调试。

引导程序阶段

执行`x/10i $pc`，查看从当前PC（0x1000）开始的10条指令，结果如下：

```
=> 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
```

这一段代码主要用于初始化硬件、建立S模式环境。

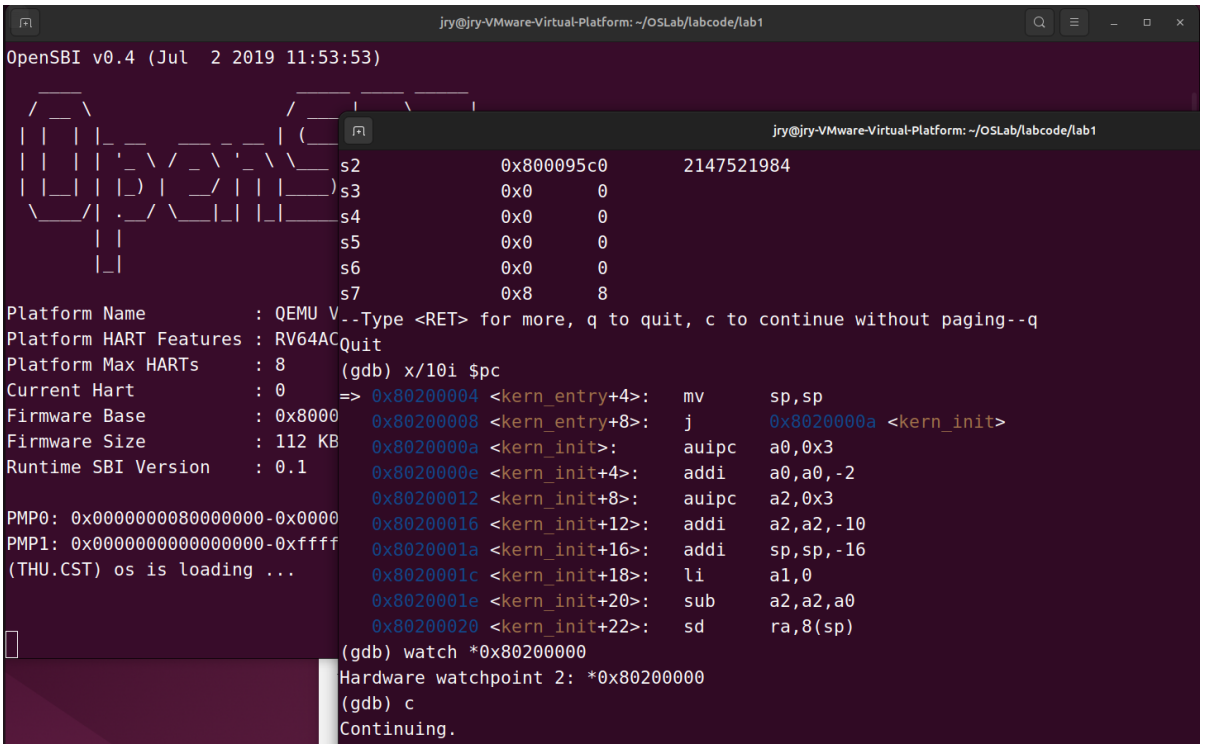
操作系统阶段

执行`b* kern_entry`，在函数kern_entry处打一个断点，随后执行`c`，内核会运行到设置好的断点处停止。执行`x/10i $pc`，查看从当前PC开始的10条指令，结果如下：

```
=> 0x80200000 <kern_entry>:     auipc   sp,0x3
   0x80200004 <kern_entry+4>:   mv      sp,sp
   0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init>
   0x8020000a <kern_init>:      auipc   a0,0x3
   0x8020000e <kern_init+4>:    addi    a0,a0,-2
   0x80200012 <kern_init+8>:    auipc   a2,0x3
   0x80200016 <kern_init+12>:   addi    a2,a2,-10
   0x8020001a <kern_init+16>:   addi    sp,sp,-16
   0x8020001c <kern_init+18>:   li      a1,0
   0x8020001e <kern_init+20>:   sub     a2,a2,a0
```

发现PC已经变成了0x80200000，接着马上就要开始执行内核。

执行`watch *0x80200000`和`c`，此时内存地址0x80200000处的内容发生变化，显示了`(THU.CST) os is loading ...`，内核已经启动成功。



2. 问题回答

RISC-V硬件加电后最初执行的几条指令**位于0x00001000处**，它们的功能如下：

```
0x1000: auipc   t0,0x0
```

这条指令把当前指令的地址（0x1000）加载到t0寄存器中。

> 指令auipc的全称叫**A**dd **U**pper **I**mmediate to **PC**，意思是把高位立即数加到当前PC上。指令格式是`auipc rd, imm20`，表示`rd ← PC + (imm20 << 12)`。

```
0x1004: addi    a1,t0,32
```

这条指令设置寄存器`a1 = 0x1020`（即`a1 ← t0 + 32 = 0x1000 + 0x20 = 0x1020`）。通常这是要传递给下一级（比如 OpenSBI 主函数）的参数，比如“设备树地址”或“引导信息表地址”。

```
0x1008: csrr    a0,mhartid
```

这条指令从mhartid控制寄存器中读取当前硬件线程（hart）的编号。RISC-V是多核设计，这一指令让引导代码知道"我是第几个核"。

```
0x100c: ld      t0,24(t0)
```

这条指令从内存地址0x1018处取出一个64位的值放入t0（即`t0 ← [0x1000 + 24] = [0x1018]`）。

```
0x1010: jr      t0
```

这条指令无条件跳转到t0指向的地址。

后面还有一些`unimp`，这些`unimp`是"未定义指令"（unimplemented），不是要执行的，只是占位。

总结：这些指令读取当前hartid、设定启动参数，并进行跳转，开始执行OpenSBI主程序，随后再引导操作系统内核。

三、实验中的重要知识点

1. 最小可执行内核的完整启动流程

第一步是**硬件初始化和固件启动。**QEMU 模拟器启动后，会模拟加电复位过程。此时 PC 被硬件强制设置为固定的复位地址0x1000，从这里开始执行一小段写死的固件代码（MROM，Machine ROM）。MROM 的功能非常有限，主要是完成最基本的环境准备，并将控制权交给OpenSBI。OpenSBI 被加载到物理内存的0x80000000处。

第二步是**OpenSBI 初始化与内核加载。**CPU 跳转到0x80000000处继续运行。OpenSBI 运行在 RISC-V 的最高特权级（M 模式），负责初始化处理器的运行环境。完成这些初始化工作后，OpenSBI 才会准备开始加载并启动操作系统内核。OpenSBI 将编译生成的内核镜像文件加载到物理内存的0x80200000地址处。

第三步是**内核启动执行。**OpenSBI 完成相关工作后，跳转到0x80200000地址，开始执行kern/init/entry.S。在0x80200000这个地址上存放的是kern/init/entry.S文件编译后的机器码，这是因为链接脚本将entry.S中的代码段放在内核镜像的最开始位置。entry.S设置内核栈指针，为C语言函数调用分配栈空间，准备C语言运行环境，然后按照RISC-V的调用约定跳转到`kern_init()`函数。最后，`kern_init()`调用`cprintf()`输出一行信息，表示内核启动成功。

2. RISC-V 的四种特权等级

RISC-V 架构中定义了 **4 个特权等级**，用于区分 CPU 执行指令时的权限高低。

等级	名称	缩写	权限级别	主要职责
0	User Mode	**U 模式**	最低	用户程序执行
1	Supervisor Mode	**S 模式**	中等	操作系统内核（内核态）
2	Hypervisor Mode	**H 模式**	高	虚拟机管理程序（可选）
3	Machine Mode	**M 模式**	最高	硬件/引导程序（固件）

CPU加电后首先进入**M模式**，执行引导程序（Bootloader）或OpenSBI，之后切换为S模式。操作系统内核运行在**S模式**，而用户程序运行在**U模式。**

四、其他重要知识点

本次实验暂无。