lab1

李佳 2312668 朱晨瑞2312674 马溪怡 2311061

练习一

指令 `la sp, bootstacktop`

- **完成的操作**:
  `la` 是 *load address* 的伪指令，用来把符号的地址加载到寄存器中。
  这里将内核栈顶符号 `bootstacktop` 的地址加载到 `sp`（栈指针寄存器）。

  在 `.data` 段中，定义了:

  ```
  .align PGSHIFT
  .global bootstack
  bootstack:               #栈底
      .space KSTACKSIZE    #分配了大小为 KSTACKSIZE 的空间
  .global bootstacktop
  bootstacktop:
  ```

- **内存分配说明**:
  `.space KSTACKSIZE` 分配一段连续空间作为内核栈。
  `bootstacktop` 标记这段空间的高地址，即栈顶位置。
- **架构特性**:
  由于 RISC-V 架构中栈是**从高地址向低地址增长**，因此把 `sp` 设置为 `bootstacktop`，意味着栈准备就绪，可以从高地址往低地址分配空间。
- **目的**:
  在执行内核 C 语言代码之前，必须先建立栈环境。因为函数调用、局部变量保存、寄存器压栈等都依赖栈。

  这条指令就是在**初始化内核栈指针**，保证后续 C 代码运行时有合法的栈空间。
- **效果**:

  把符号 `bootstacktop` 解析成内核镜像里的一个具体地址。把这个地址加载到寄存器 `sp`，作为**栈指针** 的初始值。

指令 `tail kern_init`

- **完成的操作**:
  `tail` 在 RISC-V 汇编中相当于"跳转并不保存返回地址"。
  它的效果类似:

  ```
  j kern_init
  ```

  而不是:

```
jal ra, kern_init
```

即：直接跳转到 `kern_init`，并且不记录返回地址到 `ra` 寄存器。

- **目的**：
  将控制权从汇编入口 `kern_entry` 移交给 C 语言函数 `kern_init`，进入操作系统的初始化流程。
  不保存返回地址是因为 `kern_entry` 作为内核入口点，不会再被返回；逻辑上这是一次单向跳转。
  这样更高效，也更符合语义：从此以后，CPU 执行权由 `kern_init` 接管。

整体说明

执行顺序为：

- **初始化栈**
  `la sp, bootstacktop`
  将 `sp` 指向内核栈顶，确保 C 语言环境下函数调用、局部变量存取安全。
- **进入内核主流程**
  `tail kern_init`
  把执行流交给 C 语言入口 `kern_init`，开始内核初始化逻辑（如内存管理、进程管理等）。

内存示意图

```
高地址
   ▲
   |  bootstacktop  ← sp 初始化到这里
   |
   |  栈空间（KSTACKSIZE 大小）
   |
   ▼  bootstack
低地址
```

总结

- `la sp, bootstacktop` → 为内核准备好栈环境。
- `tail kern_init` → 跳转到内核主函数，进入 C 语言阶段。

练习二

实验过程

首先，用`make clean && make -j$(nproc)`把项目重新编译一遍，然后用QEMU加载并挂起在第一条指令，等待GDB的调试。

```
ubuntu@Ubuntu-lj:~/Desktop/lab1/lab1$ make clean && make -j$(nproc)
qemu-system-riscv64 -nographic -machine virt \
  -bios default -kernel bin/kernel -S -s
rm -f -r obj bin
+ 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
```

重新打开一个终端，利用如图所示的命令让GDB加载内核符号文件，连接本地1234端口上的QEMU，从而接管被挂起的CPU。

```
ubuntu@Ubuntu-lj:~/Desktop/lab1/lab1$ riscv64-unknown-elf-gdb -q bin/kernel \
  -ex "set arch riscv:rv64" \
  -ex "target extended-remote :1234"
Reading symbols from bin/kernel...
The target architecture is set to "riscv:rv64".
Remote debugging using :1234
0x0000000000001000 in ?? ()
(gdb)
```

输入指令`x/10i $pc`让GDB从当前PC所指的位置开始，打印即将执行的10条指令。

```
0x1000: auipc   t0,0x0  #把当前 PC 的高 20 位原样装进 t0，为下一步 PC读数做准备。
0x1004: addi    a1,t0,32  #计算“t0+32”的地址并放入 a1，后面用它当参数或指针基址。
0x1008: csrr    a0,mhartid  #把当前硬件线程编号读到 a0，作为启动参数。
0x100c: ld  t0,24(t0)  #从“t0+24”处读出一个 8 字节量，放进 t0（这里读到的值就是
0x8000_0000）。
0x1010: jr  t0  #无条件跳转到刚读出的地址，CPU 离开0x1000，正式进入 OpenSBI
（0x8000_0000）。
0x1014: unimp
0x1016: unimp
0x1018: unimp
0x101a: 0x8000
0x101c: unimp
```

输入`si`指令来单步执行上面的汇编指令，同时利用`info r t0`来观察t0寄存器值的变化。

```
(gdb) si                                    #auipc t0,0x0
0x0000000000001004 in ?? ()
(gdb) info r t0
t0             0x1000   4096
(gdb) si                                    #addi  a1,t0,32
0x0000000000001008 in ?? ()
(gdb) info r t0
t0             0x1000   4096                #指令只写a1，t0不变
(gdb) si                                    #csrr  a0,mhartid
0x000000000000100c in ?? ()
(gdb) info r t0
t0             0x1000   4096                #读出当前Hart ID到a0，t0仍未变化
(gdb) si                                    #ld t0,24(t0)
0x0000000000001010 in ?? ()
```

```
(gdb) info r t0
t0             0x80000000   2147483648        #从地址t0+24读出0x8000 0000，确认t0发生
变化
(gdb) si                                      #jr    t0
0x0000000080000000 in ?? ()                   #无条件跳转，正式进入OpenSBI
```

现在已经完成了硬件到固件的交接，输入`x/10i 0x80000000`，可以看到接下来的10条汇编指令。

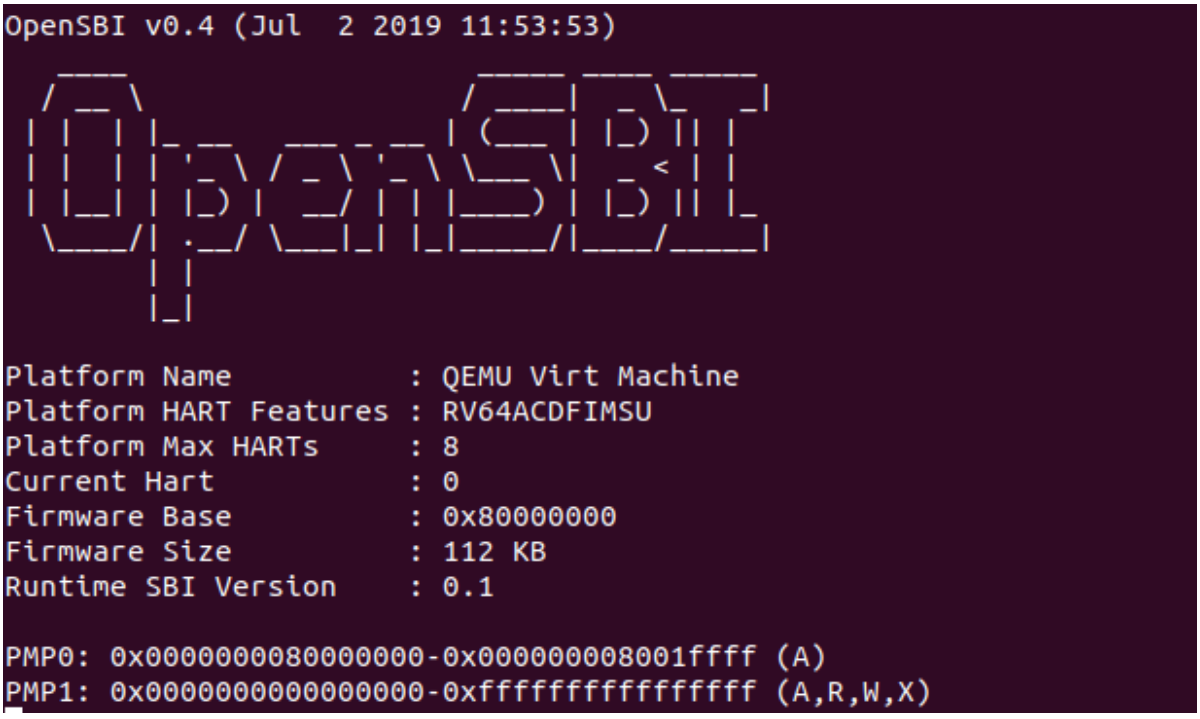
```
0x80000000: csrr    a6,mhartid    #把当前 Hart 的硬件线程ID读进a6
0x80000004: bgtz    a6,0x80000108 #如果a6>0,就跳到0x80000108
0x80000008: auipc   t0,0x0        #t0 ← PC高20位+0，为下一条PC存数准备基址
0x8000000c: addi    t0,t0,1032    #t0 = t0 + 1032 → 算出0x80000408
0x80000010: auipc   t1,0x0        #t1 ← PC高20位+0，同样拿当前PC当基址。
0x80000014: addi    t1,t1,-16     #t1 = t1-16 → 得到0x80000000
0x80000018: sd  t1,0(t0)          #把0x80000000写入0x80000408
0x8000001c: auipc   t0,0x0        #再次以 PC 为基址，准备读跳转表
0x80000020: addi    t0,t0,1020    #t0 = t0 + 1020 → 指向0x80000400
0x80000024: ld  t0,0(t0)          #从0x80000400读出一个8字节地址到t0
```

为了捕捉内核开始执行的时刻，我们在函数kern_entry第一条指令处设断点，输入指令`break kern_entry`，可以看到如下内容。

```
(gdb) break kern_entry
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
```

可以看到我们已经成功设置好了断点，对应地址0x80200000，接下来我们输入指令`continue`让程序执行到断点停下。

我们发现在第一个终端页面出现了如图所示的内容，说明OpenSBI已经启动。



接着我们回到第二个终端输入指令`x/5i 0x80200000`,看一下接下来的五条汇编代码。

```
0x80200000 <kern_entry>:    auipc   sp,0x3   #把PC高20位+0x3装进sp
0x80200004 <kern_entry+4>:  mv  sp,sp        #不改动sp的值，给链接器留插桩位，保证4字节
对齐
0x80200008 <kern_entry+8>:  j   0x8020000a <kern_init>  #跳转到kern_init
0x8020000a <kern_init>: auipc   a0,0x3       #kern_init第一条指令，a0 ← PC高20位+0x3
0x8020000e <kern_init+4>:   addi    a0,a0,-2 #把a0减2
```

我们再在kern_init函数处，设置断点，输入`break kern_init`,可以看到输出的地址与上面的跳转地址也是对应的。

```
(gdb) break kern_init
Breakpoint 2 at 0x8020000a: file kern/init/init.c, line 8.
```

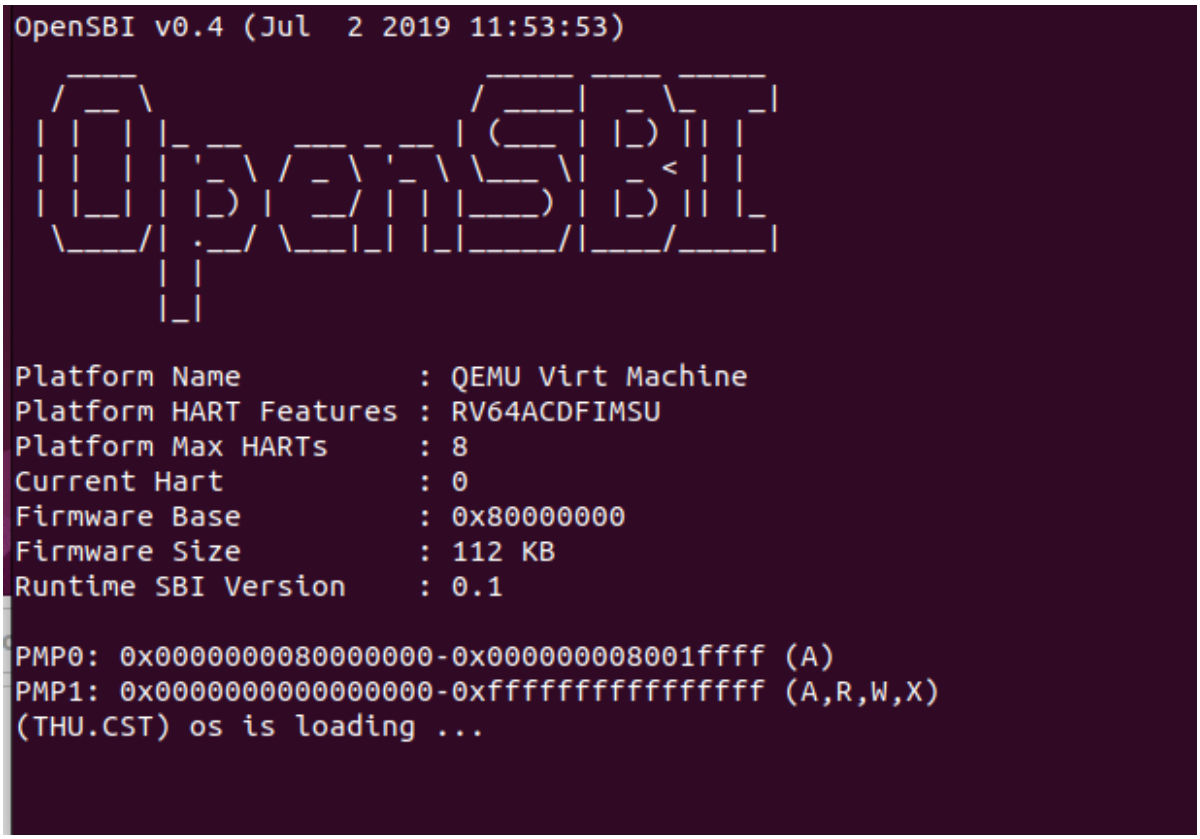
输入`continue`，执行到断点处停下。

```
Breakpoint 2, kern_init () at kern/init/init.c:8
8       memset(edata, 0, end - edata);
```

输入指令`x/15i 0x8020000c`查看接下来的15条汇编指令。

```
0x8020000c <kern_init+2>:   unimp
0x8020000e <kern_init+4>:   addi    a0,a0,-2            #把auipc算出的地址减2
0x80200012 <kern_init+8>:   auipc   a2,0x3              #给a2算高20位基址
0x80200016 <kern_init+12>:  addi    a2,a2,-10           #a2减10
0x8020001a <kern_init+16>:  addi    sp,sp,-16           #给当前函数开16字节栈帧
0x8020001c <kern_init+18>:  li  a1,0                    #a1=0
0x8020001e <kern_init+20>:  sub a2,a2,a0                #a2=a2-a0，得到.bss长度
0x80200020 <kern_init+22>:  sd  ra,8(sp)                #把返回地址存入栈中
0x80200022 <kern_init+24>:  jal ra,0x802004b6 <memset>  #调用memset函数
0x80200026 <kern_init+28>:  auipc   a1,0x0
0x8020002a <kern_init+32>:  addi    a1,a1,1186          #a1=a1+1186
0x8020002e <kern_init+36>:  auipc   a0,0x0
0x80200032 <kern_init+40>:  addi    a0,a0,1210          #a1=a1+1210
0x80200036 <kern_init+44>:  jal ra,0x80200056 <cprintf>  #调用cprintf函数并设置返回
地址
0x8020003a <kern_init+48>:  j   0x8020003a <kern_init+48>#跳转到地址0x8020003a
```

我们可以看到最后一行指令是跳转到指令本身的地址，会进行无限自循环。我们输入`continue`，可以看到第一个终端页面处多了下面一行。



证明已经执行过了cprintf函数并进入了无限自循环。

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？

地址位于0x1000~0x1010，实现功能如下：

```
0x1000: auipc   t0,0x0  #把当前 PC 的高 20 位原样装进 t0，为下一步 PC读数做准备。
0x1004: addi    a1,t0,32  #计算“t0+32”的地址并放入 a1，后面用它当参数或指针基址。
0x1008: csrr    a0,mhartid  #把当前硬件线程编号读到 a0，作为启动参数。
0x100c: ld  t0,24(t0)  #从“t0+24”处读出一个 8 字节量，放进 t0（这里读到的值就是
0x8000_0000）。
0x1010: jr  t0  #无条件跳转到刚读出的地址，CPU 离开0x1000，正式进入 OpenSBI
（0x8000_0000）。
```

重要知识点

实验中涉及

1. **内核初始化**
   - 实验：`la sp,bootstacktop`,通过`la`伪指令将内核栈顶符号bootstacktop的地址加载到栈指针寄存器sp，为内核 C 代码执行准备栈环境。栈在 RISC-V 架构中从高地址向低地址生长，bootstack到bootstacktop之间的KSTACKSIZE空间作为内核栈，支持函数调用、局部变量存储和寄存器压栈。
   - OS：进程 / 线程的栈管理，操作系统必须为内核自身及用户进程维护栈空间。
   - 理解：实验中的栈初始化是 OS 原理中 “栈管理” 在**内核启动阶段**的具体实现，是内核能够执行 C 语言代码的必要条件。
2. **控制权转移**

- 实验：使用`tail`指令从汇编入口kern_entry跳转到 C 函数kern_init，不保存返回地址，标志着执行流从汇编阶段过渡到 C 语言阶段，内核初始化逻辑正式开始。
- OS：操作系统启动流程中的控制权交接。
- 理解：`tail`指令是控制权交接的指令级实现，但原理中的控制权转移可能涉及多个层级，实验中只是从汇编到C的直接转移。

3. **硬件加电初始执行流程**
   - 实验：RISC-V硬件加电后从地址0x1000开始执行，通过auipc、addi、csrr等指令，读取当前硬件线程ID，获取OpenSBI的入口地址并跳转，完成从硬件到固件的交接。
   - OS：对应计算机的启动流程，加电自检→固件初始化→引导程序→内核加载
   - 理解：原理中的流程适用于所有架构，实验是RISC-V架构的实现。
4. **OpenSBI的作用**
   - 实验：硬件初始指令跳转至0x80000000进入 OpenSBI，OpenSBI 执行后将控制权移交内核入口kern_entry，是硬件与内核之间的中间层。
   - OS：固件负责初始化关键硬件、提供底层硬件抽象接口，为操作系统内核运行提供基础环境，是硬件与 OS 之间的适配层。
   - 理解：OpenSBI是固件在RISC-V架构下的具体实现，聚焦于它的启动交接功能。
5. **调试工具的使用**
   - 通过QEMU模拟硬件并挂起，GDB连接调试，使用`break`、`si`、`x/10i`、`info r`等指令，可以观察启动流程和指令执行过程。

实验中未涉及

1. **进程的管理与调度**

   进程是 OS 进行资源分配和调度的基本单位，当前实验仅完成内核启动初始化，未进入多进程 / 线程环境。

2. **虚拟内存与地址转换**

   实验聚焦内核启动流程，内核代码直接运行在物理地址空间，没有涉及页表建立、虚拟地址映射等环节，也没有体现用户进程的虚拟地址空间隔离。

3. **中断与异常处理**

   中断和异常是 OS 与硬件交互、响应外部事件的关键机制，实验阶段没有涉及。

4. **用户态与内核态切换**

   OS通过特权级划分隔离用户程序和内核，而实验只是运行内核态的代码，不涉及用户态与内核态的切换场景。

5. **并发控制与同步机制**

   当多线程/进程并发访问、共享资源时，需要通过同步工具避免竞争，保证数据一致，实验中没有对应的场景，也就不涉及同步问题。