1. 实验主题

比麻雀更小的麻雀（最小可执行内核）

2. 实验目的

- 使用 链接脚本 描述内存布局
- 进行 交叉编译 生成可执行文件，进而生成内核镜像
- 使用 OpenSBI 作为 bootloader 加载内核镜像，并使用 Qemu 进行模拟
- 使用 OpenSBI 提供的服务，在屏幕上格式化打印字符串用于以后调试

3. 实验过程

3.1 理解内核启动中的程序入口操作

这其中重要的代码

```
kern_entry:
    la sp, bootstacktop
    tail kern_init
```

- 首先对于`la sp, bootstacktop`

`la`(Load Address) 是 RISC-V 的伪指令，用于加载地址，整个的作用是将`bootstacktop`的地址加载到栈指针`sp`中, 其中`bootstacktop`的指向内核栈的顶部. 这条指令的目的明确, 即为**初始化内核栈,** 即在内核开始执行 C 代码之前，必须先设置好栈环境, 这是函数执行的基础.

- 然后是指令`tail kern_init`

`tail`是RISC-V的尾调用指令, 用于跳转到`kern_init`函数, 但是**不保存返回地址, 实际等价于`j kern_init`.** 这里不使用`call`是因为调用内核初始化函数后系统不需要再次返回这里, 所以没有必要再保存返回地址, 而`call`会将返回地址写入`ra`寄存器. 由于这里没有保存返回地址, 所以同时避免了不必要的栈帧创建.

3.2 使用GDB验证启动流程

使用命令`make debug`和`make gdb`连接QEMU成功后, 查看PC输出

```
pc             0x1000   0x1000
```

说明现在PC停留在BIOS内置的复位地址0x1000上. 然后使用命令`x/10i $pc`查看接下来要执行的十条命令, 解释一并如下

```
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0          # t0 = PC + 0 = 0x1000（当前地址）
   0x1004:      addi    a1,t0,32        # a1 = t0 + 32 = 0x1000 + 32 = 0x1020
   0x1008:      csrr    a0,mhartid      # a0 = 当前 hart ID
   0x100c:      ld      t0,24(t0)       # t0 = memory[t0 + 24] = memory[0x80000000]
   0x1010:      jr      t0              # 跳转到 t0 指向的地址
   0x1014:      unimp
   0x1016:      unimp
   0x1018:      unimp
   0x101a:      .insn   2, 0x8000
   0x101c:      unimp
```

这其中后面的unimp是无效指令. 然后单步执行至0x1010的位置后使用`i r $t0`查看t0指向的地址

```
(gdb) si
0x0000000000001010 in ?? ()
(gdb) i r $t0
t0             0x80000000       2147483648
(gdb) si
0x0000000080000000 in ?? ()  # PC的值
```

可以看到 PC 成功跳转到了0x80000000的位置, 接下来再次查看之后的10个指令.

```
(gdb) x/10i $pc
=> 0x80000000: csrr    a6, mhartid      # 读取当前 hart ID（CPU 核编号）
   0x80000004: bgtz    a6, 0x80000108   # 如果 hart ID > 0，跳转到 secondary_hart_entry
（其他核等待）
   0x80000008: auipc   t0, 0x0          # t0 = 0x80000008
   0x8000000c: addi    t0, t0, 1032     # t0 = 0x80000008 + 1032 = 0x80000410
   0x80000010: auipc   t1, 0x0          # t1 = 0x80000010
   0x80000014: addi    t1, t1, -16      # t1 = 0x80000000
   0x80000018: sd      t1, 0(t0)        # 将启动地址存入内存
   0x8000001c: auipc   t0, 0x0          # t0 = 0x8000001c
   0x80000020: addi    t0, t0, 1020     # t0 = 0x8000001c + 1020 = 0x80000418
   0x80000024: ld      t0, 0(t0)        # 从 0x80000418 读取一个地址
```

接下来在`kern_entry`打上断点,

```
b* kern_entry
```

输出

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

查看`0x80200000`处的后5条指令(因为代码很短, 所以这里在之后再查看后续代码)

```
(gdb) x/5i 0x80200000
   0x80200000 <kern_entry>:     auipc   sp,0x3   # sp = PC + 0x3000 = 0x80203000
   0x80200004 <kern_entry+4>:   mv      sp,sp    # 看似无用，实为对齐或编译器填充
   0x80200008 <kern_entry+8>:   j       0x8020000a <kern_init> # 跳转到 kern_init, 这里
的 j 正好对应 tail 跳转指令
   0x8020000a <kern_init>:      auipc   a0,0x3
   0x8020000e <kern_init+4>:    addi    a0,a0,-2
```

然后使用`c`继续执行至断点, 此时在左侧看到成功启动了OpenSBI

```
root@Martexz:~/labcode/lab1# make debug

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)
```

然后再对`kern_init`打上断点并执行至此处, 使用命令`disassemble kern_init`查看反汇编的代码

```
(gdb) disassemble kern_init
Dump of assembler code for function kern_init:
0x8020000a <+0>:    auipc   a0, 0x3          # a0 = &edata (高20位)
0x8020000e <+4>:    addi    a0, a0, -2       # a0 = edata 地址 (0x80203008)
0x80200012 <+8>:    auipc   a2, 0x3          # a2 = &end (高20位)
0x80200016 <+12>:   addi    a2, a2, -10      # a2 = end 地址 (0x80203008)
0x8020001a <+16>:   addi    sp, sp, -16      # 分配16字节栈空间
0x8020001c <+18>:   li      a1, 0            # a1 = 0 (memset 填充值)
0x8020001e <+20>:   sub     a2, a2, a0       # a2 = end - edata = 0 (.bss 长度)
0x80200020 <+22>:   sd      ra, 8(sp)        # 保存返回地址（虽不返回）
0x80200022 <+24>:   jal     0x80200490       # 调用 memset(edata, 0, 0)
0x80200026 <+28>:   auipc   a1, 0x0          # a1 = PC 高20位
0x8020002a <+32>:   addi    a1, a1, 1154     # a1 = 消息字符串地址 (0x802004a8)
```

```
0x8020002e <+36>:  auipc   a0, 0x0          # a0 = PC 高20位
0x80200032 <+40>:  addi    a0, a0, 1178     # a0 = 格式串地址 ("%s\n\n", 0x802004c8)
0x80200036 <+44>:  jal     0x80200054       # 调用 cprintf("%s\n\n", message)
0x8020003a <+48>:  j       0x8020003a       # 无限循环 (while(1))
```

然后再输入 `c` 后, gdb 一直显示Continuing. 说明正在执行死循环, 此时左侧也成功输出(THU.CST) os is loading ... 说明此时内核成功启动.

3.3 回答问题

在 QEMU 模拟的 RISC-V `virt` 机器上，硬件加电（复位）后，CPU 最初从物理地址 `0x1000` 开始执行指令。

这段位于 `0x1000` 的代码是 QEMU 内置的 启动固件 stub（由 `-bios default` 提供），其主要功能包括:

1. 获取当前代码地址（通过 `auipc t0, 0`）;
2. 读取当前 hart ID（通过 `csrr a0, mhartid`），用于多核系统中区分 CPU 核心;
3. 设置设备树地址（`a1 = 0x1020`），指向 QEMU 自动生成的设备树 blob（DTB）;
4. 从固定偏移（`0x1018`）加载内核入口地址（如 `0x80200000`）;
5. 跳转到操作系统内核入口，将控制权交给内核。