《操作系统》实验报告

lab1：最小可执行内核

1. 实验内容

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

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。

2. 练习1

1. `la sp,bootstacktop`：将`bootstacktop`对应的地址赋值给`sp`寄存器，目的是初始化栈，为栈分配内存空间。
2. `tail kern_init`：尾调用，在函数`kern_init`的位置继续执行，目的是进入操作系统的入口。

3. 练习2

> **完整流程**
>
> 最小可执行内核的完整启动流程为:
>
> ```
> 加电复位 → CPU从0x1000进入MROM → 跳转到0x80000000(OpenSBI) → OpenSBI初始化并加载内核到0x80200000 → 跳转到entry.S → 调用kern_init() → 输出信息 → 结束
> ```
>
> **详细步骤**
>
> 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。

`ucore.img`文件获取

我们先在根目录进行`make`操作，得到输出：

```
wey@wey:/mnt/d/os-riscv/labcode/lab1$ make # 编译所有源代码并生成内核镜像

+ 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
```

成功获得`ucore.img`和`kernel`文件。

环境准备与连接

- 启动QEMU：`qemu-system-riscv64 -machine virt -kernel bin/kernel -nographic -s -S`
- 启动GDB：`riscv64-unknown-elf-gdb bin/kernel`
- 设置目标架构：`set architecture riscv:rv64`
- 连接：`target remote :1234`

输出结果如下：

```
(gdb) set architecture riscv:rv64
The target architecture is set to "riscv:rv64".
(gdb) target remote :1234
Remote debugging using :1234
0x0000000000001000 in ?? ()
```

关键断点设置

```
(gdb) b *0x1000           # 复位地址断点
(gdb) b *0x80200000       # 内核入口断点
```

输出结果如下：

```
(gdb) b *0x1000
Breakpoint 1 at 0x1000
(gdb) b *0x80200000
Breakpoint 2 at 0x80200000: file kern/init/entry.S, line 7.
```

复位地址代码分析

连接后立即停在`0x1000`，查看前6条指令：

```
(gdb) x/10i $pc
=> 0x1000:      auipc   t0,0x0
   0x1004:      addi    a2,t0,40
   0x1008:      csrr    a0,mhartid
   0x100c:      ld      a1,32(t0)
   0x1010:      ld      t0,24(t0)
   0x1014:      jr      t0
   0x1018:      unimp
   0x101a:      .insn   2, 0x8000
   0x101c:      unimp
   0x101e:      unimp
```

所以最初执行的几条指令地址位于0x1000~0x101e，具体分析功能如下：

1. `auipc t0,0x0`
   将当前PC值（0x1000）存入t0寄存器，建立地址基准用于后续相对寻址。
2. `addi a2,t0,40`
   计算地址0x1028（0x1000 + 40）并存入a2寄存器，准备设备树或其他配置信息的地址参数。
3. `csrr a0,mhartid`
   读取当前硬件线程ID到a0寄存器，识别正在执行的CPU核心。
4. `ld a1,32(t0)`
   从地址0x1020（0x1000 + 32）加载数据到a1寄存器，获取设备树大小或其他配置参数。
5. `ld t0,24(t0)`
   从地址0x1018（0x1000 + 24）加载跳转地址到t0寄存器，获取OpenSBI固件的入口地址0x80000000。
6. `jr t0`
   跳转到t0寄存器指定的地址0x80000000，将控制权完全转移给OpenSBI固件进行后续初始化。

我们通过`si`命令逐步运行，最后跳转到地址0x80000000进行OpenSBI固件初始化。

```
(gdb) si
0x0000000000001004 in ?? ()
(gdb) si
0x0000000000001008 in ?? ()
(gdb) si
0x000000000000100c in ?? ()
(gdb) si
0x0000000000001010 in ?? ()
(gdb) i r $t0
t0             0x1000   4096
(gdb) si
```

```
0x0000000000001014 in ?? ()
(gdb) i r $t0
t0             0x80000000       2147483648
(gdb) si
0x0000000080000000 in ?? ()
```

监控内核加载

设置观察点监控内核加载：

```
(gdb) watch *0x80200000
(gdb) c
```

到达内核入口

观察点触发后继续执行，最终到达内核入口：

```
Breakpoint 2, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
```

OpenSBI输出信息

从另一个终端的输出可以看到：

- 平台信息：RISC-V VirtIO平台
- 固件基地址：`0x80000000`
- 内核加载地址：`0x80200000`
- 启动模式：S-mode（监管者模式）

输出结果如下：

```
OpenSBI v1.0


____                   _____ ____ _____

  / __ \                 / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : medeleg
Platform HART Count       : 1
Platform IPI Device       : aclint-mswi
Platform Timer Device     : aclint-mtimer @ 10000000Hz
Platform Console Device   : uart8250
Platform HSM Device       : ---
Platform Reboot Device    : sifive_test
```

```
Platform Shutdown Device  : sifive_test
Firmware Base             : 0x80000000
Firmware Size             : 252 KB
Runtime SBI Version       : 0.3

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000002000000-0x000000000200ffff (I)
Domain0 Region01          : 0x0000000080000000-0x0000000008003ffff ()
Domain0 Region02          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsuh
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
```

最后我们输入`c`使程序继续运行，发现输出

```
(gdb) c
Continuing.
```

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

gdb 一直显示`Continuing`，说明正在执行死循环,，此时另一个终端也成功输出`(THU.CST) os is loading ...`说明此时内核成功启动.