Operating System Lab 1

成员：禹相祐（2312900） 查科言（2312189） 董丰瑞 （2311973）

练习一

> 在操作系统内核启动流程中，`kern/init/entry.S`是内核的入口汇编代码，负责完成从底层硬件到内核C代码执行的过渡

（1）`la sp,bootstacktop`完成了什么操作，目的是什么？

- **作用：初始化内核栈**
  - 将`bootstacktop`的地址加载到 $RISC-V$ 的栈指针寄存器`sp`（指向栈顶部）中
- **目的：**
  - 在内核启动的早期即汇编阶段，首要任务是为后续的C代码执行准备运行环境，栈初始化是基础的一步，C语言的函数调用、局部变量的存储，返回地址保存都依赖栈来实现。**内核启动时，`sp`寄存器的值不确定，直接运行C代码会导致栈访问错误。**

（2）`tail kern_init`完成了什么操作，目的是什么？

- **作用：无返回的跳转到`kern_init`函数**
  - `tail`是 $RISC-V$ 汇编中的伪指令（实际会被编译为`jr ra`等指令）
    - `tail`与普通跳转指令（如`j`）的区别：`tail`会优化函数调用的栈帧，可以释放调用者的栈空间
  - 本质是将`kern_init`的地址写入程序计数器`pc`，且不保留当前函数的返回信息。
- **目的：**
  - 完成阶段的初始化工作后，将执行流从汇编代码切换到 C 语言代码，进入内核的正式初始化流程，如初始化内存管理、设备、进程调度等。

练习二

（1） Running ucore

> 根据知道书中的内容可以学习到：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
> ```

我们首先在根目录下执行命令`make qemu`，可以看到

输出一行`(THU.CST) os is loading`，进入了死循环，和我们的C代码相符

（2）GDB调试

我们接下来逐步验证：

我们使用`tmux`工具同时查看两个终端，一个用来运行`qemu`，一个用来运行`GDB`

在左侧运行指令`make debug`，右侧运行`make gdb`，执行完如下：

- 左侧：让`qemu`监听1234端口，等待`gdb`连接
- 右侧：启动`GDB`，加载内核镜像并连接到`qemu`

  我们可以看到右侧输出了一些内容：
- 首先是：GDB版本与配置，运行在`x86_64 Linux`主机，调试`riscv64-unknown-elf`目标
- 之后是：`Reading symbols from bin/kernel...`GDB正从`bin/kernel`中读取调试符号，并确认目标结构为$RISC-V$ 64位
- 最后是：连接到了`qemu`的1234端口，进入远程调试模式；`0x0000000000001000 in ?? ()`显示当前`PC`指向地址`0x1000`，`??`表示无法识别函数名，但此处没有调试符号。

> 在`QEMU`模拟的这款$RISC-V$处理器中，将复位向量地址初始化为`0x1000`，再将PC初始化为该复位地址，因此处理器将从此处开始执行复位代码

---

① CPU从0x1000进入MROM

我们输入命令`x/10i $pc`可以查看当前程序计数器`$pc`指向的内存区域，知道正在执行和以后要执行的10条指令。我们写一些注释：

```
(gdb) x/10i $pc
=> 0x1000:  auipc   t0,0x0 #将pc高20位与0x0拼接存入t0,即t0=pc= 0x1000
   0x1004:  addi    a1,t0,32   #a1=t0+32=0x1020
   0x1008:  csrr    a0,mhartid  #读取机器模式hart ID,存入寄存器a0=0x0
   0x100c:  ld      t0,24(t0)#从t0+24处加载8字节数据,t0=0x80000000
   0x1010:  jr      t0 #跳转到t0存储的地址
   ...
(gdb)
```

之后，我们进行单步调试，输入`si`，并且每输入一次，就查看一次寄存器中的值`info r t0`。

② 跳转到0x80000000

我们再输入`x/10i 0x80000000`，对这个地址处处的代码进行反汇编，显示10条指令。该地址处加载的是作为`bootloader`的`OpenSBI.bin`，如下,我们写一写注释来理解：

```
(gdb) x/10i 0x80000000
=> 0x80000000:  csrr    a6,mhartid  #读取机器模式hart ID,并存入a6
   0x80000004:  bgtz    a6,0x80000108 #若a6大于0，跳转，适用于多核场景
   0x80000008:  auipc   t0,0x0    #t0=pc,获取当前地址
   0x8000000c:  addi    t0,t0,1032  #t0=t0+1032
   0x80000010:  auipc   t1,0x0    #t1=pc
   0x80000014:  addi    t1,t1,-16  # t1=t1=16
   0x80000018:  sd      t1,0(t0)  #将t1的值存储到t0+0(即t0)指向的地址
   0x8000001c:  auipc   t0,0x0  #将PC存入t0
   0x80000020:  addi    t0,t0,1020 #t0=t-+1020
   0x80000024:  ld      t0,0(t0) #从内存地址t0+0处加载8字节数据，存入t0
(gdb)
```

- 这些指令是**RISC-V 平台的底层固件（OpenSBI）**的初始化逻辑，判断是否是多核场景，计算并操作内存地址，用于加载或存储数据，为内核启动做准备

---

③ OpenSBI初始化并加载内核到0x80200000,进入kern_entry

我们执行指令`break kern_entry`，在此处打下断点

接着执行指令`x/10i 0x80200000`，查看此处的汇编指令，这是内核入口相关的代码

```
(gdb) x/10i 0x80200000
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
(gdb) c
Continuing.
```

- 其中`0x80200000 <kern_entry>:  auipc   sp,0x3`**就是我们的练习一中**`la sp, bootstacktop`**实际执行的机器指令**，将当前PC的高20位与`0x3`拼接，结果存入栈指针寄存器`sp`，即初始化内核栈的栈顶。
- 其中`0x80200008 <kern_entry+8>:        j       0x8020000a <kern_init>`对应的是`tail kern_init`，无条件跳转，且不修改`ra`即返回寄存器的值。

我们通过`si`单步执行和`disassemble`也可以验证出来：

下图是在`la sp, bootstacktop`反汇编：

下图是在`tail kern_init`处反汇编：

- 我们可以看到调试器在此处，`memset(edata, 0, end - edata)`是将已初始化数据短的结束地址`（edata）`到未初始化数据段的结束地址`(end)`之间清零，即初始化`BSS`段。

接着我们输入`c`，执行到断点处，可以看到左侧终端输出，说明这时候`OpenSBI`已经启动

> 补充：

我们也可以用`x/10x $sp`来查看`sp`指向的内存区域即栈顶地址`bootstacktop`，可以看到，在未初始化之前，是杂乱的数据，肯是上一次被使用后的数据，或者是硬件上电后的随机值。

我们单步执行，发现执行过`la sp,bootstacktop`之后，对栈顶附近的内存进行了初始化清零，位后续函数调用提高干净的栈空间，用实验验证出来我们的练习一。

---

④ 调用`kern_init()`，并输出信息

我们输入`break kern_init`在此处打下断点，并`continue`执行到断点处，可以看到输出`(THU.CST) os is loading`

在`kern_init`的汇编代码的最后，调用`cprintf`进行打印，跳转到自身地址，进入死循环，内核初始化完成！