Lab 0.5 最小可执行内核

小组成员：于成俊、楚乾靖、李帅东

练习：使用GDB验证启动流程

【启动流程简述】

在riscv上电时，会进行CPU自检，然后跳转到bootloader处执行。bootloader设置好kernel的运行环境后，从硬盘加载kernel到内存，最后再跳转到kernel入口地址。

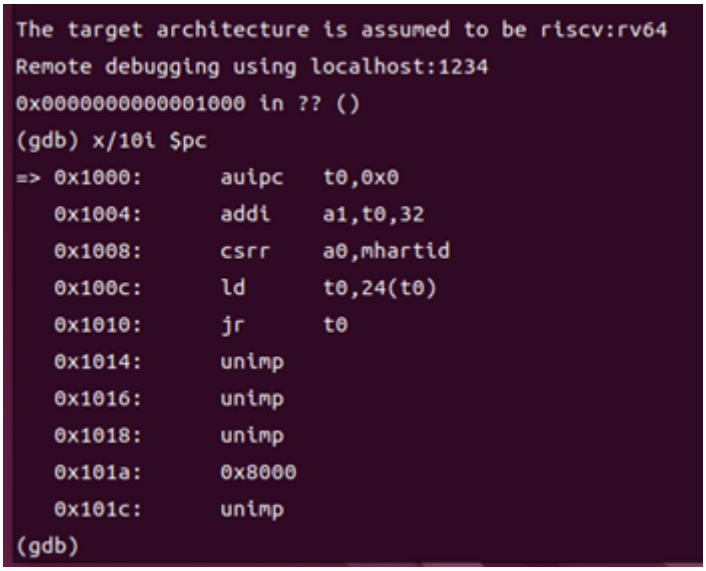
我们采用的bootloader为OpenSBI，被加载到0x80000000地址，OpenSBI探测好外设并初始化内核的环境变量后，加载内核到0x80200000地址，最后再跳转到0x80200000地址。从上文我们知道，我们的入口点*start*地址正好为*0x80200000*，也就是*OpenSBI*会调用我们*kernel*的start函数。

【验证】

在lab0的实验路径下，依次执行以下指令，开启GDB

```
make debug
make gdb
```

1.系统初始化



发现系统从0x1000的位置开始执行。0x1000是该系统的复位地址，每次系统都会从复位地址开始运行，以执行相关的初始化和准备工作。

输入指令`l`，得到执行的源码

```
0x0000000000001000 in ?? ()
(gdb) l
1       #include <mmu.h>
2       #include <memlayout.h>
3
4           .section .text,"ax",%progbits
5           .globl kern_entry
6       kern_entry:
7           la sp, bootstacktop
8
9           tail kern_init
10
(gdb)
```

这些指令为系统刚刚上电时所执行的命令。我们可以发现，在entry/S当中，OpenSBI启动后进行内核栈的分配（执行`la sp, bootstacktop`）,再调用C语言编写的`kern_init`进行内核初始化。

2.我们执行到`0x1010 jr t0`跳转至0x80000000地址处。根据理论知识，这里就开始执行bootloader了。可以发现，跳转之后程序在分配内核栈。

```
(gdb) x/10i $pc
=> 0x80000000:  csrr    a6,mhartid
   0x80000004:  bgtz    a6,0x80000108
   0x80000008:  auipc   t0,0x0
   0x8000000c:  addi    t0,t0,1032
   0x80000010:  auipc   t1,0x0
   0x80000014:  addi    t1,t1,-16
   0x80000018:  sd      t1,0(t0)
   0x8000001c:  auipc   t0,0x0
   0x80000020:  addi    t0,t0,1020
   0x80000024:  ld      t0,0(t0)
(gdb)
```

```
0x0000000080000000 in ?? ()
(gdb) l
11      .section .data
12          # .align 2^12
13          .align PGSHIFT
14          .global bootstack
15      bootstack:
16          .space KSTACKSIZE
17          .global bootstacktop
18      bootstacktop:(gdb)
```

（`.align PGSHIFT`按照2^PGSHIFT进行地址对齐, 也就是对齐到下一页 PGSHIFT在 mmu.h定义）

(`.global bootstack`定义内核栈)

(`.space KSTACKSIZE`留出KSTACKSIZE字节的内存)

(`.global bootstacktop`之后内核栈将要从高地址向低地址增加，初始时内核栈为空)

3.内核入口！kern_init

`break *0x80200000`在内核初始化函数处打断点。我们发现将要执行`la sp,bootstacktop`。此时仍然处于`entry.S`文件当中而并非`init.c`文件当中。所以此处并不是内核真正的入口点！

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

Breakpoint 1, kern_entry () at kern/init/entry.S:7
7           la sp, bootstacktop
```

继续执行，发现程序进入`init.c`文件当中的`kern_init()`函数。进入程序内核，开始准备打印字符`'（THU.CST） os is loading ... \n ’`

```
(gdb) si
9           tail kern_init
(gdb) si
kern_init () at kern/init/init.c:8
8           memset(edata, 0, end - edata);
(gdb) l
3       #include <sbi.h>
4       int kern_init(void) __attribute__((noreturn));
5
6       int kern_init(void) {
7           extern char edata[], end[];
8           memset(edata, 0, end - edata);
9
10          const char *message = "(THU.CST) os is loading ...\n";
11          cprintf("%s\n\n", message);
12          while (1)
(gdb)
```

Help

知识点总结

1.面向Ucore的操作系统上电初步理解

qemu模拟risc-v计算机硬件。

复位向量地址0x1000，初始化PC为改地址。

OpenSBI(bootloader)加载操作系统到内存当中。先将作为BootLoader的OpenSBI.bin加载到0x80000000地址，将内核镜像os.bin 加载至0x80200000中(kern/init/entry.S)，操作系统再开始运行。

Q:0x80000000这个地址是怎么来的?

A：看汇编代码，`jr t0`跳转至0x80000000。此时，`t0`存入的是0x1018，因此复位地址处的程序决定了0x8000000

0x80200000这个地址由QEMU（CPU）决定。

地址无关代码：在主存储器中的任意位置都可以正确运行，不受绝对地址影响的机器码

地址相关代码与之相反。

2.Ucore文件梳理

（1）kern

init/entry.S:OpenSBI启动后跳转到的汇编代码（用以启动操作系统的第一段代码，or操作系统初始化代码的入口文件）

init/init.c:操作系统初始化代码文件

driver/console.c(h):控制台（输入输出）驱动相关，调用了sbi.c(h)

libs/stdio.c&readline.c&printfmt.c:实现标准输入输出

errors.h:定义内核错误类型的宏

（2）libs

riscv.h:宏定义riscv指令集的寄存器和指令，实现了在C程序中类似函数般使用内链汇编

sbi.c(h):封装OpenSBI接口为函数，利于C程序使用之

defs.h:定义常用类型和宏

string.c(h):用于操作字符数组的函数，类似于C中的string.h

（3）tools

kernel.ld:链接脚本

function.mk:定义makefile中使用的一些函数

（4）makefile:GNU make编译脚本

3.内存布局

elf文件和bin文件为两种不同的可执行文件格式。其中bin文件头会简单解释自己应该被加载到什么起始位置，而ELF文件包含冗余的调试信息，指定程序每个section的内存布局。因此，我们可以将内存布局合适的elf文件转化为bin文件，然后再加载到qemu中运行，就可以节省部分内存。

-
  - 一个程序包含多个段，如.text 段，即代码段，存放汇编代码； .rodata 段，即只读数据段，； .data 段，存放被初始化的可读写数据，通常保存程序中的全局变量； .bss 段，存放被初始化为 00 的可读写数据。stack栈负责函数调用和局部变量的控制，heap堆负责动态内存的分配。
  - 链接器，把输入文件的各个节链接在一起，链接脚本(linker script)的作用，就是描述怎样把输入文件的section映射到输出文件的section, 同时规定这些section的内存布局。
  - 链接脚本里把程序入口点定义为 kern_entry ，在kernel/init/entry.S中，调用kern_init作为真正的入口点。
- “真正的”入口点

  ```
  kern/init/init.c
  ```

  中，存在kern_init函数，这个函数对内存进行了初始化，然后输出了os正在loading

4.零散小知识

（1）三种状态/环境（逻辑从低到高，权限从高到底）：M--S--U

User模式：该特权模式为权限最小的模式，在linux系统中用户态就运行在该特权级；

Supervisor模式：该特权级时linux操作系统运行的模式，特权级别比User模式高；

Machine模式：CPU上电启动后运行在该特权模式，该特权比Supervisor更高。

从 U 到 S 再到 M，权限不断提高，这意味着你可以使用更多的特权指令，访需求权限更高的寄存器等等。我们可以使用一些指令来修改 CPU 的**当前特权级**

（2）状态转换指令：ecall指令（environment call），C语言通过内联汇编调用该指令

（3）文件调用流：sbi.c（内链汇编ecall指令）定义函数sbi_call、函数sbi_console_putchar调用之->console.c定义函数cons_putc以封装sbi_console_putchar->stdio.c定义函数cputch以调用cons_putc、函数cputs/cprintf调用之

5.makefile初步

target:目标文件（object or executable）

prerequisites:生成target所需文件或目标

command:make需要执行的命令

```
target  ...: prerequisites ...
    command
    ...
    ...
```

```
.PHONY: qemu
    qemu:$(UCOREIMG) $(SWAPIMG) $(SFSIMG)
#   $(V)$(QEMU) -kernel $(UCOREIMG) -nographic
    $(V)$(QEMU) \
        -machine virt \
        -nographic \
        -bios default \
        -device loader,file=$(UCOREIMG),addr=0x80200000
```

问题讨论

1.谁引导了BootLoader?

2.源程序的编译、汇编、链接等是被谁执行的?

3.为啥复位地址0x1000对应的源程序是这样子的?