操作系统Lab0.5实验报告

学号：2210620　姓名：何畅

学号：2212117　姓名：胡雨欣

实验目的

实验0.5主要讲解最小可执行内核和启动流程。我们的内核主要在 Qemu 模拟器上运行，它可以模拟一台 64 位 RISC-V 计算机。为了让我们的内核能够正确对接到 Qemu 模拟器上，需要了解 Qemu 模拟器的启动流程，还需要一些程序内存布局和编译流程（特别是链接）相关知识。

本章将学到：

- 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

实验内容

根据资料，最小可执行内核的执行流为：
加电 -> OpenSBI启动 -> 跳转到 0x80200000 (kern/init/entry.S) ->进入 kern_init() 函数 (kern/init/init.c) ->调用 cprintf() 输出一行信息->结束。

1.显示即将执行的15条汇编指令

在make debug和make gdb后，运行如下指令：

```
x/15i $pc
```

得到的输出结果如下：

```
0x1000: auipc   t0,0x0
0x1004: addi    a1,t0,32
0x1008: csrr    a0,mhartid
0x100c: ld  t0,24(t0)
0x1010: jr  t0
0x1014: unimp
0x1016: unimp
0x1018: unimp
0x101a: .insn   2, 0x8000
0x101c: unimp
0x101e: unimp
0x1020: addi    a2,sp,724
0x1022: sd  t6,216(sp)
0x1024: unimp
0x1026: addiw   a2,a2,3
```

`0x1000: auipc t0,0x0`

- `AUIPC`（Add Upper Immediate to PC）指令，将当前程序计数器（PC）加上一个立即数的高位部分（这里是`0x0`），结果存储到寄存器`t0`中。这条指令通常用于生成一个基地址。

`0x1004: addi a1,t0,32`

- `ADDI`（Add Immediate）指令，取寄存器`t0`中的值加上立即数`32`，并将结果存储到寄存器`a1`中。也就是`a1 = t0 + 32`。

`0x1008: csrr a0,mhartid`

- `CSRR`（Control and Status Register Read）指令，读取机器级硬件线程ID（`mhartid`）寄存器的值并将其存储到`a0`中。`mhartid`寄存器通常保存当前处理器核心的ID。

`0x100c: ld t0,24(t0)`

- `LD`（Load Doubleword）指令，从`t0`寄存器中地址偏移`24`的内存中加载一个64位数据，并将其存储到`t0`寄存器中。即从`t0+24`的内存地址读取数据。

`0x1010: jr t0`

- `JR`（Jump Register）指令，跳转到`t0`寄存器中的地址。

`0x1014: unimp`

- `UNIMP`指令表示"未实现的操作"，这通常是用于占位或用于调试的非法指令。

`0x101a: .insn 2, 0x8000`

- 这是一个自定义指令，它使用了`.insn`伪指令，它允许程序员手动编码非标准的指令。

`0x1020: addi a2,sp,724`

- `ADDI`指令，取寄存器`sp`（堆栈指针）的值加上立即数`724`，并将结果存储到寄存器`a2`中。也就是`a2 = sp + 724`。

`0x1022: sd t6,216(sp)`

- `SD`（Store Doubleword）指令，将寄存器`t6`的值存储到`sp+216`地址的内存中。也就是将`t6`的值存到堆栈偏移为`216`的地址处。

`0x1026: addiw a2,a2,3`

- `ADDIW`（Add Immediate Word）指令，取寄存器`a2`中的值加上立即数`3`，并将结果存储到寄存器`a2`中。

2.显示当前所有寄存器信息

在运行指令前需要检查一下当前寄存器的值，使用如下指令：

```
info register
```

得到结果如下：

```
ra             0x0  0x0
sp             0x0  0x0
gp             0x0  0x0
tp             0x0  0x0
t0             0x0  0
t1             0x0  0
```

```
t2              0x0   0
fp              0x0   0x0
s1              0x0   0
a0              0x0   0
a1              0x0   0
a2              0x0   0
a3              0x0   0
a4              0x0   0
a5              0x0   0
a6              0x0   0
a7              0x0   0
s2              0x0   0
s3              0x0   0
s4              0x0   0
s5              0x0   0
s6              0x0   0
s7              0x0   0
s8              0x0   0
s9              0x0   0
s10             0x0   0
s11             0x0   0
t3              0x0   0
t4              0x0   0
t5              0x0   0
t6              0x0   0
pc              0x1000    0x1000
```

可以看出程序运行前，`PC`值为`0x1000`，`t0`值为`0`，按照第一条指令的解释，运行第一条指令后，`t0`值应变为`0x1000`。

3.运行0x1000指令

运行`0x1000`指令后，`t0`值如下：

```
t0              0x1000    4096
```

符合上一步猜想。

4.运行0x1004指令

根据指令，运行后应为`a1=t0+32=0x1000+32=0x1020`，运行这条指令，寄存器值为：

```
a1              0x1020    4128
```

符合猜想。

5.运行0x1008指令

在运行这条指令前，先记录`mhartid`和`a0`寄存器的值：

```
mhartid         0x0   0
a0              0x0   0
```

运行后，`a0` 寄存器的值依然为 `0`。

6.运行0x100C和0x1010指令

`0x100C` 是从 `t0+24` 的内存地址读取数据，并将其存储到 `t0` 寄存器中。这一步运行后 `t0` 的值是 `0x80000000`。

接下来的0x1000指令是一个跳转指令，即程序会跳转到 `0x80000000`。

这一步运行结束后，观察各寄存器的值，其中：

```
pc             0x80000000   0x80000000
```

证明程序跳转成功，接下来运行 `0x80000000` 处的指令。 而我们知道，`bootloader` 的 `OpenSBI.bin` 被加载到物理内存以物理地址 `0x80000000` 开头的区域上。

7.阶段小结

不难看出，上面几条指令首先将程序计数器的当前值存储到 `t0` 寄存器中，然后在该值上加 `32` 并存储到 `a1` 寄存器中。接着，它读取当前硬件线程的ID并存储到 `a0` 寄存器中。之后，它从内存中加载一个64位数到 `t0` 寄存器中，实现跳转到 `0x80000000`。

其中把硬件线程的ID储存是因为主核心会负责引导操作系统，而其他核心可能会等待或执行不同的初始化。当主核心完成某些初始化工作时，可以通知其他核心继续运行特定的任务。因此，读取并存储线程ID，是为了正确区分、管理和协调多核处理器的启动流程。

8.分析kern/init/entry.S

我们知道，内核镜像 `os.bin` 被加载到以物理地址 `0x80200000` 开头的区域上。

因此我们对 `0x80200000` 设置断点：

```
break *0x80200000
```

运行后可以看到显示：

```
Breakpoint 1 at 0x80200000: file kern/init/entry.S, line 7.
```

表示在 `kern/init/entry.S` 文件中的第7行设置了这个断点。

接下来我们看一看 `kern/init/entry.S` 有什么：

```
#include <mmu.h>
#include <memlayout.h>

    .section .text,"ax",%progbits
    .globl kern_entry
kern_entry:
    la sp, bootstacktop

    tail kern_init

.section .data
    # .align 2^12
```

```
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

8.1头文件

```
#include <mmu.h>
#include <memlayout.h>
```

这些是预处理器指令。这些头文件可能定义了一些内存管理单元（MMU）相关的常量、内存布局以及其他与内存地址相关的定义，如 `PGSHIFT` 和 `KSTACKSIZE` 等宏。

8.2定义

```
.section .text,"ax",%progbits
.globl kern_entry
```

第一条指令定义了 `.text` 段，通常用于存放可执行代码。`"ax"` 表示该段具有可执行（`x`）和可读（`a`）权限，`%progbits` 表示这是包含程序代码的部分。

`.globl` 表示 `kern_entry` 是一个全局符号，可以被其他文件访问。它定义了内核启动的入口点，即所有内核初始化操作都从这里开始。

8.3kern_entry

```
kern_entry:
    la sp, bootstacktop

    tail kern_init
```

`la` 是 "load address" 指令，它将 `bootstacktop` 的地址加载到栈指针寄存器（`sp`）中。`bootstacktop` 是内核栈的栈顶位置。

`tail` 是汇编中的一个优化指令，它相当于一个无条件跳转，并且在跳转时不保留当前函数的调用栈。此指令会直接跳转到 `kern_init` 函数，并将控制权交给它。

8.4设置内核的初始状态

```
.section .data
    # .align 2^12
    .align PGSHIFT
    .global bootstack
bootstack:
    .space KSTACKSIZE
    .global bootstacktop
bootstacktop:
```

第一条指令这定义了 `.data` 段，存放的是程序中的已初始化数据。这里面存放的是与内核栈相关的数据。

`.align` 用于调整段的对齐方式。`PGSHIFT` 可能定义为页大小的移位数（通常是 `12`，即4KB对齐），因此这条指令将数据对齐到一个页边界。

`.global` 指令将 `bootstack` 声明为全局符号，以便其他文件可以访问它。`bootstack` 是内核栈的起始地址。

`.space` 用于在内存中分配一块空间，这里分配的空间大小为 `KSTACKSIZE`，即内核栈的大小。

`.global` 指令将 `bootstacktop` 声明为全局符号。

这段代码确保了内核启动时栈的正确初始化，并且为内核初始化代码 `kern_init` 的执行做好了准备。

9.运行

运行指令`continue`后，程序在断点处停下，观察当前`PC`下的接下来10条指令：

```
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
```

`AUIPC` 是 "Add Upper Immediate to PC" 的缩写，作用是将当前的程序计数器（PC）值加上立即数的高位部分（0x3），并将结果存入指定的寄存器，这里是栈指针寄存器（`sp`）。即`sp`的值是`0x80200000 + 0x3000 = 0x80203000`。

`MV` 是 "move" 指令，作用是将一个寄存器的值复制到另一个寄存器中。这里是将 `sp` 寄存器的值复制回 `sp` 自身。

`J` 是 "jump" 指令，用于无条件跳转到一个指定的地址。这里跳转到 `0x8020000A` 地址处，也就是 `kern_init` 函数的位置，开始执行内核初始化代码。

10.分析`kern_init`

因为`kern_init` 函数在`init.c`文件里，所以我们进入`init.c`。

```c
#include <stdio.h>
#include <string.h>
#include <sbi.h>
int kern_init(void) __attribute__((noreturn));

int kern_init(void) {
    extern char edata[], end[];
    memset(edata, 0, end - edata);

    const char *message = "(THU.CST) os is loading ...\n";
    cprintf("%s\n\n", message);
   while (1)
        ;
}
```

`extern char edata[], end[];` 声明了两个外部符号 `edata` 和 `end`

`memset(edata, 0, end - edata);` 使用 `memset` 函数将从 `edata` 到 `end` 之间的内存区域清零。这段区域通常是BSS段（未初始化的全局变量区域）。

`const char \*message = "(THU.CST) os is loading ...\n";` 定义了一个字符串常量 `message`，内容为 `"(THU.CST) os is loading ...\n"`。这个消息是操作系统启动时显示给用户的提示信息。

`cprintf("%s\n\n", message);` 调用 `cprintf` 函数将 `message` 内容输出到控制台。

`while (1)` 这是一个无限循环，程序进入这个循环后永不退出。

在控制台运行 `make qemu`，使用 `qemu` 语句进行 `qemu` 启动加载内核可以看到：

```
hechang@hechang-virtual-machine:/mnt/hgfs/Ubuntu/riscv64-ucore-labcodes/lab0$ make qemu

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

说明 `ucore` 成功执行。

实验感想

在本次实验中，我深入了解了操作系统内核启动过程中的关键环节。本实验通过对上电后 `OpenSBI` 启动过程的分析以及跳转到 `0x80200000` 后对内核初始化代码的调试，不仅让我对硬件启动和操作系统的交互有了更深的认识，还增强了我对低层系统结构的理解，让我知道启动操作系统需要一个复杂的过程。