实验1: Rinux 内核引导

1 实验目的

- 学习 RISC-V 汇编， 编写 head.S 实现跳转到内核运行的第一个 C 函数。
- 学习 OpenSBI，理解 OpenSBI 在实验中所起到的作用，并调用 OpenSBI 提供的接口完成字符的输出。
- 学习 Makefile 相关知识， 补充项目中的 Makefile 文件， 来完成对整个工程的管理。

2 实验环境

- Environment in Lab0

3 实验基础知识介绍

3.1 前置知识

为了顺利完成 OS 实验，我们需要一些前置知识和较多调试技巧。在 OS 实验中我们需要 **RISC-V汇编** 的前置知识，课堂上不会讲授，请同学们通过阅读以下四份文档自学：

- RISC-V Assembly Programmer's Manual
- RISC-V Unprivileged Spec
- RISC-V Privileged Spec
- RISC-V 手册（中文）

> 注：RISC-V 手册（中文）中有一些 Typo，请谨慎参考。

3.2 RISC-V 的三种特权模式

RISC-V 有三个特权模式：U (user) 模式、S (supervisor) 模式和 M (machine) 模式。

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

其中：

- M 模式是对硬件操作的抽象，有**最高**级别的权限
- S 模式介于 M 模式和 U 模式之间，在操作系统中对应于内核态 (Kernel)。当用户需要内核资源时，向内核申请，并切换到内核态进行处理
- U 模式用于执行用户程序，在操作系统中对应于用户态，有**最低**级别的权限

3.3 从计算机上电到 OS 运行

我们以最基础的嵌入式系统为例，计算机上电后，首先硬件进行一些基础的初始化后，将 CPU 的 Program Counter 移动到内存中 Bootloader 的起始地址。
Bootloader 是操作系统内核运行之前，用于初始化硬件，加载操作系统内核。
在 RISC-V 架构里，Bootloader 运行在 M 模式下。Bootloader 运行完毕后就会把当前模式切换到 S 模式下，机器随后开始运行 Kernel。

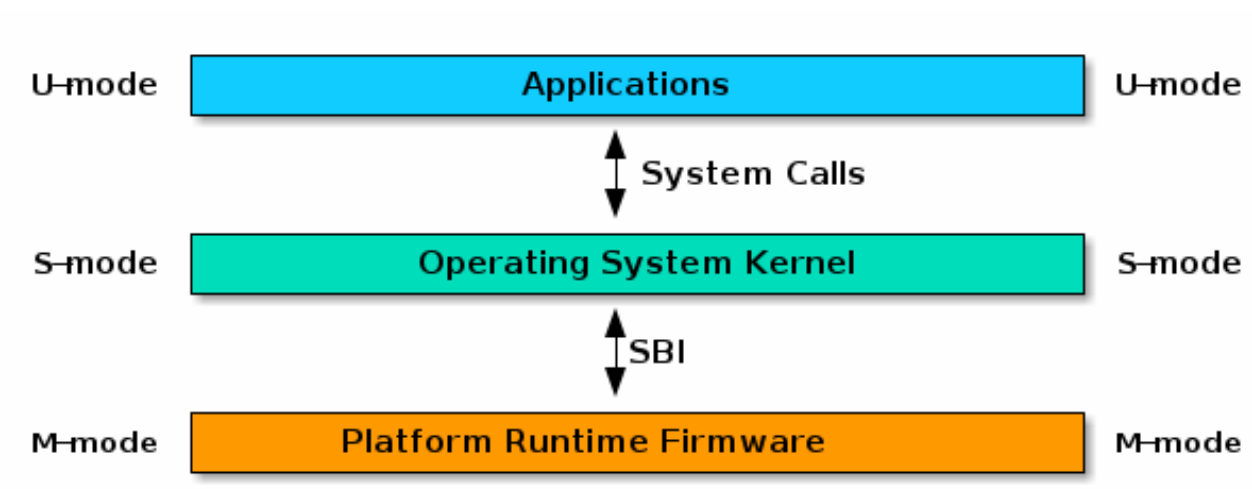
该过程简单演示如下：

```
   Hardware              RISC-V M Mode            RISC-V S Mode
+------------+         +--------------+         +----------+
|  Power On  |  ---->  |  Bootloader  |  ---->  |  Kernel  |
+------------+         +--------------+         +----------+
```

3.4 SBI 与 OpenSBI

SBI (Supervisor Binary Interface) 是 S-mode 的 Kernel 和 M-mode 执行环境之间的接口规范，而 OpenSBI 是一个 RISC-V SBI 规范的开源实现。RISC-V 平台和 SoC 供应商可以自主扩展 OpenSBI 实现，以适应特定的硬件配置。

简单的说，为了使操作系统内核适配不同硬件，OpenSBI 提出了一系列规范对 M-mode 下的硬件进行了统一定义，运行在 S-mode 下的内核可以按照这些规范对不同硬件进行操作。



为降低实验难度，我们选择 OpenSBI 作为 Bootloader 来完成机器启动时 M-mode 下的硬件初始化与寄存器设置，并使用 OpenSBI 所提供的接口完成诸如字符打印的操作。

在实验中，QEMU 已经内置了 OpenSBI 作为 Bootloader，我们可以使用 `-bios default` 启用。如果启用，QEMU 会将 OpenSBI 代码加载到 0x80000000 起始处。OpenSBI 初始化完成后，会跳转到 0x80200000 处（也就是 Kernel 的起始地址）。因此，我们所编译的代码需要放到 0x80200000 处。

如果你对 RISC-V 架构的 Boot 流程有更多的好奇，可以参考这份 bootflow。

3.5 Makefile

Makefile 可以简单的认为是一个工程文件的编译规则，描述了整个工程的编译和链接流程。在 Lab0 中我们已经使用了 make 工具利用 Makefile 文件来管理整个工程。在阅读了 Makefile介绍 这一章节后，同学们可以根据工程文件夹里 Makefile 的代码来掌握一些基本的使用技巧。

3.6 内联汇编

内联汇编（通常由 `asm` 或者 `__asm__` 关键字引入）提供了将汇编语言源代码嵌入 C 程序的能力。
内联汇编的详细介绍请参考 Assembler Instructions with C Expression Operands 。
下面简要介绍一下这次实验会用到的一些内联汇编知识：

内联汇编基本格式为：

```
    __asm__ volatile (
        "instruction1\n"
        "instruction2\n"
        ......
        ......
        "instruction3\n"
        : [out1] "=r" (v1),[out2] "=r" (v2)
        : [in1] "r" (v1), [in2] "r" (v2)
        : "memory"
    );
```

其中，三个 `:` 将汇编部分分成了四部分：

- 第一部分是汇编指令，指令末尾需要添加 '\n'。
- 第二部分是输出操作数部分。
- 第三部分是输入操作数部分。
- 第四部分是可能影响的寄存器或存储器，用于告知编译器当前内联汇编语句可能会对某些寄存器或内存进行修改，使得编译器在优化时将其因素考虑进去。

这四部分中的后三部分不是必须的。

示例一

```
unsigned long long s_example(unsigned long long type,unsigned long long arg0) {
    unsigned long long ret_val;
    __asm__ volatile (
        "mv x10, %[type]\n"
        "mv x11, %[arg0]\n"
        "mv %[ret_val], x12"
        : [ret_val] "=r" (ret_val)
        : [type] "r" (type), [arg0] "r" (arg0)
        : "memory"
    );
    return ret_val;
}
```

示例一中指令部分，`%[type]`、`%[arg0]` 以及 `%[ret_val]` 代表着特定的寄存器或是内存。

输入输出部分中，`[type] "r" (type)`代表着将 `()` 中的变量 `type` 放入寄存器中（`"r"` 指放入寄存器，如果是 `"m"` 则为放入内存），并且绑定到 `[]` 中命名的符号中去。`[ret_val] "=r" (ret_val)` 代表着将汇编指令中 `%[ret_val]` 的值更新到变量 `ret_val`中。

示例二

```
#define write_csr(reg, val) ({
    __asm__ volatile ("csrw " #reg ", %0" :: "r"(val)); })
```

示例二定义了一个宏，其中 `%0` 代表着输出输入部分的第一个符号，即 `val`。

`#reg` 是c语言的一个特殊宏定义语法，相当于将reg进行宏替换并用双引号包裹起来。

例如 `write_csr(sstatus,val)` 经宏展开会得到：

```
({
    __asm__ volatile ("csrw " "sstatus" ", %0" :: "r"(val)); })
```

3.7 编译相关知识介绍

vmlinux.lds

GNU ld 即链接器，用于将 `*.o` 文件（和库文件）链接成可执行文件。在操作系统开发中，为了指定程序的内存布局，ld 使用链接脚本（Linker Script）来控制，在 Linux Kernel 中链接脚本被命名为 vmlinux.lds。更多关于 ld 的介绍可以使用 `man ld` 命令。

下面给出一个 vmlinux.lds 的例子：

```
/* 目标架构 */
OUTPUT_ARCH( "riscv" )

/* 程序入口 */
ENTRY( _start )

/* kernel代码起始位置 */
BASE_ADDR = 0x80200000;

SECTIONS
{
    /* . 代表当前地址 */
    . = BASE_ADDR;

    /* 记录kernel代码的起始地址 */
    _skernel = .;

    /* ALIGN(0x1000) 表示4KB对齐 */
    /* _stext, _etext 分别记录了text段的起始与结束地址 */
    .text : ALIGN(0x1000){
        _stext = .;

        *(.text.entry)
        *(.text .text.*)

        _etext = .;
    }

    .rodata : ALIGN(0x1000){
        _srodata = .;

        *(.rodata .rodata.*)

        _erodata = .;
    }

    .data : ALIGN(0x1000){
        _sdata = .;

        *(.data .data.*)

        _edata = .;
    }

    .bss : ALIGN(0x1000){
        _sbss = .;

        *(.bss.stack)
        sbss = .;
        *(.bss .bss.*)

        _ebss = .;
    }

    /* 记录kernel代码的结束地址 */
    _ekernel = .;
}
```

首先我们使用 OUTPUT_ARCH 指定了架构为 RISC-V ，之后使用 ENTRY 指定程序入口点为 `_start` 函数，程序入口点即程序启动时运行的函数，经过这样的指定后在head.S中需要编写 `_start` 函数，程序才能正常运行。

链接脚本中有`.` `*`两个重要的符号。单独的`.`在链接脚本代表当前地址，它有赋值、被赋值、自增等操作。而`*`有两种用法，其一是`*()`在大括号中表示将所有文件中符合括号内要求的段放置在当前位置，其二是作为通配符。

链接脚本的主体是SECTIONS部分，在这里链接脚本的工作是将程序的各个段按顺序放在各个地址上，在例子中就是从0x80200000地址开始放置了 `.text` ， `.rodata` ， `.data` 和 `.bss` 段。各个段的作用可以简要概括成：

段名	主要作用
.text	通常存放程序执行代码
.rodata	通常存放常量等只读数据
.data	通常存放已初始化的全局变量、静态变量
.bss	通常存放未初始化的全局变量、静态变量

在链接脚本中可以自定义符号，例如以上所有 `_s` 与 `_e` 开头的符号都是我们自己定义的。

更多有关链接脚本语法可以参考这里。

vmlinux

vmlinux 通常指 Linux Kernel 编译出的可执行文件 (Executable and Linkable Format / ELF)，特点是未压缩的，带调试信息和符号表的。在整套 OS 实验中，vmlinux 通常指将你的代码进行编译，链接后生成的可供 QEMU 运行的 RV64 架构程序。如果对 vmlinux 使用 `file` 命令，你将看到如下信息：

```
$ file vmlinux
vmlinux: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, not stripped
```

System.map

System.map是内核符号表（Kernel Symbol Table）文件，是存储了所有内核符号及其地址的一个列表。“符号”通常指的是函数名，全局变量名等等。使用 `nm vmlinux` 命令即可打印vmlinux的符号表，符号表的样例如下：

```
0000000000000800 A __vdso_rt_sigreturn
ffffffe000000000 T __init_begin
ffffffe000000000 T _sinittext
ffffffe000000000 T _start
ffffffe000000040 T _start_kernel
ffffffe000000076 t clear_bss
ffffffe000000080 t clear_bss_done
ffffffe0000000c0 t relocate
ffffffe00000017c t set_reset_devices
ffffffe000000190 t debug_kernel
```

使用 System.map 可以方便地读出函数或变量的地址，为 Debug 提供了方便。

4 实验步骤

4.1 准备工程

从 源代码仓库: os22fall-stu 同步实验代码框架

```
├── arch
│   └── riscv
│       ├── include
│       │   ├── defs.h
│       │   └── sbi.h
│       ├── kernel
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sbi.c
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   ├── print.h
│   └── types.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── print.c
└── Makefile
```

需要完善以下文件：

- arch/riscv/kernel/head.S
- lib/Makefile
- arch/riscv/kernel/sbi.c
- lib/print.c
- arch/riscv/include/defs.h

参考 Lab0 中，将上述工程代码映射进容器目录 `/home/oslab` 中，这样就可以方便的在本地开发，同时使用容器内的工具进行编译。

```
### 将用户的 home 目录映射到 docker 镜像内的 `/home/oslab`
$ docker run --name oslab22 -it -v ${HOME}:/home/oslab oslab:2022 /bin/bash
### --name 生成带有外部目录挂载的Docker容器名称
### -v 本地目录:容器内目录

### 在容器中检查外部目录是否挂载成功
# ls /home/oslab/
```

4.2 编写head.S

学习riscv的汇编。

完成 arch/riscv/kernel/head.S 。我们首先为即将运行的第一个 C 函数设置程序栈（栈的大小可以设置为4KB，留意栈的增长方向），并将该栈放置在 `.bss.stack` 段。接下来我们只需要通过跳转指令，跳转至 main.c 中的 `start_kernel` 函数即可。

4.3 完善 Makefile 脚本

阅读文档中关于 Makefile 的章节，以及工程文件中的 Makefile 文件，根据注释学会 Makefile 的使用规则后，补充 `lib/Makefile`，使工程得以编译。

完成此步后在工程根文件夹执行 make，可以看到工程成功编译出 vmlinux。

4.4 补充 `sbi.c`

OpenSBI 在 M 态，为 S 态提供了多种接口，比如字符串输入输出。因此我们需要实现调用 OpenSBI 接口的功能。给出函数定义如下：

```
struct sbiret {
  long error;
  long value;
};

struct sbiret sbi_ecall(int ext, int fid,
                        uint64 arg0, uint64 arg1, uint64 arg2,
                        uint64 arg3, uint64 arg4, uint64 arg5);
```

sbi_ecall 函数中，需要完成以下内容：

1. 将 ext (Extension ID) 放入寄存器 a7 中，fid (Function ID) 放入寄存器 a6 中，将 arg0 ~ arg5 放入寄存器 a0 ~ a5 中。
2. 使用 `ecall` 指令。`ecall` 之后系统会进入 M 模式，之后 OpenSBI 会完成相关操作。
3. OpenSBI 的返回结果会存放在寄存器 a0 ， a1 中，其中 a0 为 error code， a1 为返回值， 我们用 sbiret 来接受这两个返回值。

同学们可以参照内联汇编的示例一完成该函数的编写。
编写成功后，调用 `sbi_ecall(0x1, 0x0, 0x30, 0, 0, 0, 0, 0)` 将会输出字符'0'。其中`0x1`代表 `sbi_console_putchar` 的ExtensionID， `0x0`代表FunctionID, 0x30代表'0'的ascii值，其余参数填0。

请在 `arch/riscv/kernel/sbi.c` 中补充 `sbi_ecall()`。

下面列出了一些在后续的实验中可能需要使用的功能。

Function Name	Function ID	Extension ID
sbi_set_timer （设置时钟相关寄存器）	0	0x00
sbi_console_putchar （打印字符）	0	0x01
sbi_console_getchar （接收字符）	0	0x02
sbi_shutdown （关机）	0	0x08

4.5 `puts()` 和 `puti()`

调用以上完成的 `sbi_ecall`，完成 `puts()` 和 `puti()` 的实现。
`puts()` 用于打印字符串， `puti()` 用于打印整型变量。

请编写 `lib/print.c` 中的 `puts()` 和 `puti()`， 函数的相关定义已经写在了 `print.h` 文件。

4.6 修改 defs

内联汇编的相关知识见内联汇编。

学习了解了以上知识后，补充 `arch/riscv/include/defs.h` 中的代码完成：

补充完 `read_csr` 这个宏定义。这里有相关示例。

思考题

实验1： Rinux 内核引导

1. 请总结一下 RISC-V 的 calling convention，并解释 Caller / Callee Saved Register 有什么区别?
2. 编译之后，通过 System.map 查看 vmlinux.lds 中自定义符号的值。
3. 用 `read_csr` 宏读取 `sstatus` 寄存器的值，对照 RISC-V 手册解释其含义。
4. 用 `write_csr` 宏向 `sscratch` 寄存器写入数据，并验证是否写入成功。
5. 尝试使用本实验环境中提供的 `riscv64-unknown-elf-readelf` 和 `riscv64-unknown-linux-gnu-objdump` 工具，通过选择添加不同的命令行参数，解析本次实验编译所得 vmlinux文件，并截图说明。
6. 查看本次实验编译所得 vmlinux 文件的类型，简要说明该类型文件的构成，以及它与本实验所用链接脚本（vmlinux.lds）之间的联系。
7. 探索Linux 内核版本 v6.0 源代码，从中找到`ARM32`，`RISC-V(32 bit)`，`RISC-V(64 bit)`，`x86(32 bit)`，`x86_64`这些不同架构中的系统调用表（System Call Table），并列出具体的文件路径。

作业提交

1. 对于每个实验，请每位同学提交电子文档的实验报告，报告文件名：姓名+实验编号.pdf
2. 将本次实验源代码树打包构成一个压缩文件：姓名+实验编号.zip
3. 将每个实验的上述两个文件上传到“学在浙大”中

浙江大学实验报告

课程名称： 操作系统

实验项目名称：

学生姓名： 学号：

电子邮件地址：

实验日期： 年 月 日

一、实验内容

记录实验过程并截图，对每一步的命令以及结果进行必要的解释

三、讨论、心得（20分）

在这里写：实验过程中遇到的问题及解决的方法，你做本实验体会