Lab 1: Rinux 内核引导

1. 实验介绍

1.1 实验目的

1. 学习 操作系统的引导过程；
2. 学习 RISC-V 汇编；
3. 学习 OpenSBI 的接口调用规则；
4. 学习 Makefile 使用与编写，掌握基本的 Makefile 命令；

1.2 实验内容

编写 head.S 实现跳转到内核运行的第一个 C 函数，随后调用 OpenSBI 接口完成字符的输出，并编写 Makefile 来完成对整个工程的编译、链接管理。

2. 实验环境

Lab0中所提供的Docker镜像。

3. 实验基础知识介绍

3.1 前置知识

为了顺利完成后续的 OS 实验，我们需要了解一些 **RISC-V** 的前置知识，请同学们通过阅读以下五份 RISC-V参考手册进行学习：

- RISC-V Assembly Programmer's Manual
- RISC-V Unprivileged Spec
- RISC-V Privileged Spec
- RISC-V Supervisor Binary Interface Specification
- RISC-V 手册（中文）

> 注：RISC-V 手册（中文）中存在一些 Typo，请谨慎参考。部分文件也可从“学在浙大”中下载。

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。

RISC-V 通过设置不同的特权级别模式来管理系统资源的使用。

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

其中：

- M模式具有**最高**级别的权限，该模式下的操作被认为是安全可信的，主要为对硬件的操作。
- 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，拥有**最低**级别的权限；

3.3 Bootloader介绍

BootLoader是系统加电后运行的第一段代码，它在操作系统内核运行之前运行，可以分为Booter和Loader，Booter是初始化系统硬件使之能够运行起来；Loader是建立内存空间映射图，将操作系统镜像加载到内存中，并跳转过去运行。经过Bootloader的引导加载后，系统就会退出bootloader程序，启动并运行操作系统，此后交由内核接管。

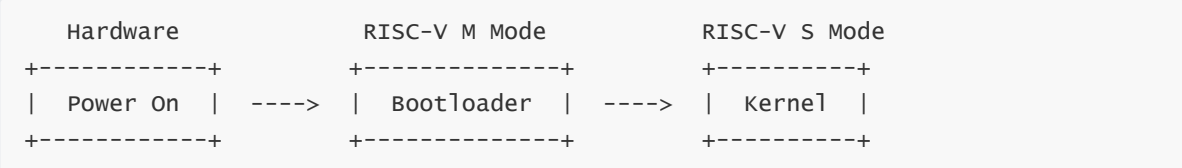
Bootloader启动可分为两个阶段：

第一阶段主要包含依赖于CPU的体系结构硬件初始化的代码，通常都用汇编语言来实现。这个阶段的任务为基本的硬件设备初始化（屏蔽所有的中断、关闭处理器内部指令/数据Cache等）、设置堆栈、跳转到第二阶段的C程序入口点。

第二阶段通常用C语言完成，以便实现更复杂的功能，也使程序有更好的可读性和可移植性。这个阶段的任务有：初始化本阶段要使用到的硬件设备、检测系统内存映射、为内核设置启动参数等，它为内核的运行完成所需的初始化和准备工作。

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，计算机随后开始运行操作系统内核。我们以最基础的嵌入式系统为例：计算机上电，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 是指运行操作系统内核之前，用于初始化硬件，加载操作系统内核的软件。

这个过程是这样的：



3.4 SBI 与 OpenSBI

SBI (Supervisor Binary Interface) 是 S-mode 下运行的操作系统内核(Kernel) 与 M-mode 执行环境之间的接口规范，而 OpenSBI 是 RISC-V SBI 规范的一种开源实现。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的硬件进行操作。

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QEMU的命令行参数 `-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。

如果你想对 RISC-V 架构的 Boot 流程有更多的了解，可以参考这份文档 bootflow 进行学习。

3.5 Makefile

Linux 有个很强大的工具make，它可以管理多模块。make工具提供灵活的机制来建立大型的软件项目。make工具依赖于一个特殊的，名字为makefile或Makefile的文件，这个文件描述了系统中各个模块之间的依赖关系。系统中部分文件改变时，make根据这些关系决定一个需要重新编译的文件的最小集合。

Makefile是一个文本形式的数据库文件，其中包含一些规则来告诉make处理哪些文件以及如何处理这些文件。这些规则主要是描述哪些文件（称为target目标文件，不要和编译时产生的目标文件相混淆）是从哪些别的文件（称为依赖文件）中产生的，以及用什么命令来执行这个过程。

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 Linux内核所提供的 Makefile 文件来管理整个工程。在阅读了 Makefile介绍 这一章节后，同学们可以学会初步编写 Makefile 文件。

3.6 内联汇编

内联汇编通常由C语言中的 `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。
有关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这些因素考虑进去。

> 注：这四部分中的后三个部分不是必须的。

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;
}
```

示例一中内联汇编的指令部分：

`%[type]`、`%[arg0]` 以及 `%[ret_val]` 代表着特定的寄存器或是内存。

在内联汇编的输出部分中：

`[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。

在内联汇编的输入部分中：

`[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。`[arg0] "r" (arg0)` 同理。

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

同学们可从 浙江大学操作系统课程学生版仓库 同步 `src/lab1` 的实验代码框架。

仿照Linux Kernel的结构，下图是本次实验需要遵循的文件结构。除了 `3.5节` 所介绍的 `Makefile` 以外，`main.c` 中包含了一个 `start_kernel` 函数，是程序最终到达的函数；`test.h`、`test.c` 中包含了一个 `test()` 函数，它是本实验的测试函数，会被 `start_kernel` 函数调用。本小结的Linux编译相关知识介绍将围绕下图中其他部分文件展开。

```
lab1
├── arch
|   └── riscv
|       ├── include
|       |   ├── defs.h           // 本次实验需要编写完善
|       |   └── sbi.h
|       ├── kernel
```

```
|       |   ├── head.S           // 本次实验需要编写完善
|       |   ├── Makefile
|       |   ├── sbi.c                // 本次实验需要编写完善
|       |   └── vmlinux.lds // 在3.7.1小节中进行介绍
|       └── Makefile
├── include
|   ├── print.h
|   └── types.h
├── init
|   ├── main.c
|   ├── Makefile
|   └── test.c
├── lib
|   ├── Makefile                        // 本次实验需要编写完善
|   └── print.c                         // 本次实验需要编写完善
├── Makefile
├── System.map                          // 本次实验将要生成，在3.7.4小节中进行介绍
└── vmlinux                             // 本次实验将要生成，在3.7.2小节中进行介绍
```

3.7.1 vmlinux.lds介绍

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` 命令查询。

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。

3.7.2 什么是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，带调试信息和符号表。在本套操作系统实验中，vmlinux 通常将指由你的代码进行编译、链接后生成的可供 QEMU (qemu-system-riscv64) 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
```

3.7.3 什么是Image

本次实验还将基于vmlinux生成一个内核的Image文件。

在Linux Kernel开发中，为了对vmlinux进行精简，通常使用objcopy去除调试信息与符号表并生成二进制文件，这就是Image。Lab0中QEMU正是使用了Image而不是vmlinux运行。

```
$ objcopy -O binary vmlinux Image --strip-all
```

此时再对Image使用**file**命令时：

```
$ file Image
image: data
```

3.7.4 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。

3.8 OpenSBI 的接口的使用方式

OpenSBI 运行在 **M** 态，并为 **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 的返回结果会放到寄存器 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。

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

更多有关Opensbi的详细内容与用法可以参考这里：RISC-V Supervisor Binary Interface Specification

4 实验步骤

4.1 准备工程

从 浙江大学操作系统课程学生版仓库 同步实验代码框架，目录树如下：

```
lab1
├── 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
├── System.map                          // 本次实验要生成的文件
└── vmlinux                             // 本次实验要生成的文件
```

本实验需要完善以下文件：

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

将工程代码映射进容器中这样就可以方便地在本地开发，同时使用容器内的工具进行编译。具体的映射方式可以通过如下方式进行：

```
### 首先新建自己本地的工作目录(比如lab1)并进入
$ mkdir lab1
$ cd lab1
$ pwd
~/.../lab1

### 查看docker已有镜像(与lab0同一个image)
$ docker image ls
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
oslab               2021                678605140682        46 hours ago
2.89GB

### 创建新的容器，同时建立volume映射。其中，“6786”取自Image ID的前4位，以实验实际ID为准
$ docker run -it -v `pwd`:/home/oslab/lab1 -u oslab -w /home/oslab 6786
/bin/bash
oslab@3c1da3906541:~$

### -v dir1 : dir2。dir1是宿主机内的目录，dir2 是容器内的目录。比如此处宿主机内的目录为
pwd=/home/user/lab1，容器内的目录为 /home/oslab/lab1

### 测试映射关系是否成功。新开一个shell终端，并进入之前~/.../lab1目录下
$ touch testfile
$ ls
testfile

### 在docker中确认是否挂载成功。切换回到第一个shell终端，
oslab@3c1da3906541:~$ pwd
/home/oslab
oslab@3c1da3906541:~$ cd lab1
oslab@3c1da3906541:~$ ls
testfile

### 确认映射关系建立成功后在本地lab1目录下继续实验
```

4.2 编写head.S

请参考RISC-V手册以及The RISC-V Instruction Set Manual来学习RISC-V的基础指令，为编写head.S文件打好基础。编写head.S，并实现如下功能：

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），并将该栈放置在`.bss.stack`段。

接下来我们只需要通过RISC-V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，以及实验代码框架已给出的 Makefile 文件，根据注释学会 Makefile 的使用规则后，补充`lib/Makefile`，实现Makefile的层级调用，使得工程得以编译。

完成此步后在工程根文件夹执行 make，将可以看到工程成功编译出`vmlinux`，`System.map`，`Image`等文件到相应的位置。

4.4 补充 `sbi.c`

学习了解完汇编与内联汇编后，请在`arch/riscv/kernel/sbi.c`中补充`sbi_ecall()`。

请参考实验手册如下章节所描述的内容：

- **内联汇编示例**: 参考实验指导3.6小节
- **OpenSBI接口规范**: 参考实验指导3.8小节

完成`sbi_ecall()`后请尝试使用`sbi_ecall()`打印单个字符，成功后进行下一步实验。

4.5 `puts()` 和 `puti()`

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。`puts()`用于打印字符串，`puti()`用于打印整型变量。

4.6 修改 defs

内联汇编的相关知识详见：实验指导**3.6小节 内联汇编示例。**

学习了解了以上知识后，补充`arch/riscv/include/defs.h`中的代码完成`read_csr`这个宏定义。

> 注：这里有相关示例。

思考题

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

5. 实验任务与要求

- 请各位同学独立完成作业，任何抄袭行为都将使本次实验判为0分。
- 请学习基础知识，并按照实验步骤指导完成实验，撰写实验报告。实验报告的要求：
  - 各实验步骤的截图以及结果分析
  - 实验结束后的心得体会
  - 对实验指导的建议（可选）

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

二、思考题

这里回答实验思考题有关的内容

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