Lab 1: RISC-V 移植

1. 实验简介

学习RISC-V相关知识，Makefile相关知识，编写head.S实现bootloader的功能，并利用Makefile来完成对整个工程的管理。

2. 实验环境

- Docker in Lab0

3. 实验基础知识介绍

3.1 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Linux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。本实验中对应为head.S的编写，操作相关寄存器，实现模式切换、中断关闭、程序跳转等。

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。

3.2 寄存器介绍

寄存器是计算机中最基本的概念，是中央处理器用来存放数据，地址以及指令的部件。一般而言，寄存器都有各自的功能，比如数据寄存器一般用来保存操作数和运算结果等，指针寄存器一般用来存放堆栈内存存储的偏移量等。而RISC-V中寄存器的种类以及功能详细的信息可以参考RISC-V手册以及The RISC-V Instruction Set Manual。接下来我们对实验中需要用到的几个寄存器进行介绍。

mstatus寄存器

mstatus寄存器，即Machine Status Register，是一种CSRs(Control and Status Registers)，其中m代表machine mode（参考3.6 Mode介绍），此寄存器中保持跟踪以及控制hart(hardware thread)的运算状态。比如mie和mip都对应mstatus上的某些bit位，所以通过对mstatus进行一下按位运算，可以实现对不同bit位的设置，从而控制不同运算状态（具体的mstatus位的布局请参考The RISC-V Instruction Set Manual 3.1.6节）。

同时注意此处的mie和mip指的是mstatus上的状态位。

mie以及mip寄存器

mie以及mip寄存器是Machine Interrup Registers，用来保存中断相关的一些信息，通过mstatus上mie以及mip位的设置，以及mie和mip本身两个寄存器的设置可以实现对硬件中断的控制。mie以及mip具体的布局及各个位的解释请参考The RISC-V Instruction Set Manual 3.1.9节。

mtvec寄存器

mtvec(即Machine Trap-Vector Base-Address Register)寄存器，主要保存machine mode下的trap vector（可理解为中断向量）的设置，其包含一个基地址以及一个mode。具体布局以及介绍请参考The RISC-V Instruction Set Manual 3.1.7节。

stvec寄存器

stvec(即Supervisor Trap Vector Bass Address Register)寄存器，其作用与mstatus类似，区别是保存的是supervisor mode对应的base和mode。具体布局以及介绍请参考The RISC-V Instruction Set Manual 4.1.2节。

sp寄存器

sp寄存器即栈顶指针寄存器，栈是程序运行中一个基本概念，栈顶指针寄存器是用来保存当前栈的栈顶地址的寄存器。

3.3 Makefile介绍

Makefile是一种实现关系整个工程编译规则的文件，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，那么什么是编译，以及Makefile的作用是什么，如何编写Makefile，请参考Makefile介绍进行学习，为之后实验步骤打好基础。

3.4 Linux Basic

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**3.3**介绍的Makefile，图中head.S是本实验最主要的部分，承担了BootLoader的工作；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`os_test`函数，它是本实验的测试函数，会被`start_kernel`函数调用。本部分的Linux基础知识将围绕下图中其他的文件展开。

```
lab1
├── arch
│   └── riscv
│       ├── boot
│       │   └── Image
│       ├── include
│       ├── kernel
│       │   ├── head.S
│       │   ├── Makefile
│       │   └── vmlinux.lds
│       └── Makefile
├── include
│   └── test.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── Makefile
├── System.map
```

```
└── vmlinux
```

什么是**vmlinux**

vmlinux通常指Linux Kernel编译出的可执行文件（Executable and Linkable Format， ELF），特点是未压缩的，带调试信息和符号表的。在本实验中，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RISC-V 64-bit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
```

什么是**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。

什么是**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 )
/* 程序起始地址 */
BASE_ADDR = 0x80000000;
SECTIONS
{
  /* . 代表当前地址 */
  . = BASE_ADDR;
  /* code 段 */
  .text : { *(.text) }
  /* data 段 */
```

```
  .rodata : { *(.rodata) }
  .data : { *(.data) }
  .bss : { *(.bss) }
  . += 0x8000;
  /* 栈顶 */
  stack_top = .;
  /* 程序结束地址 */
  _end = .;
}
```

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0x80000000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stack_top与_end都是我们自己定义的，其中stack_top与程序调用栈有关。

更多有关链接脚本语法可以参考这里。

什么是Image

在Linux Kernel开发中，为了对vmlinux进行精简，通常使用objcopy丢弃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
```

3.5 Driver介绍

本部分主要是介绍QEMU模拟的 RISC-V Virt 计算机访问外设的方法。简要的说，RISC-V使用MMIO（Memory Mapped I/O）技术，将外设映射到各个物理地址，这样我们可以通过访问物理地址来操作外设。在virt.c的virt_memmap[]中有各个外设的物理地址。本次实验只需要关注**0x10000000**地址即VIRT_UART0，它是串口，通过向串口写入字符我们可以将它们打印出来。

在test.c中有一个`putChar`函数使用了串口，打印所需的字符串。这只是一个最简单的版本，最终我们可以利用这个串口实现`printf`函数。

3.6 Mode介绍

RISC-V有三个特权模式：U（user）模式、S（supervisor）模式和M（machine）模式。它通过设置不同的特权级别模式来管理系统资源的使用。其中M模式是最高级别，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；U模式是最低级别，该模式主要执行用户程序，操作系统中对应于用户态；S模式介于M模式和U模式之间，操作系统中对应于内核态，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。

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

4. 实验步骤

4.1 搭建实验环境

实验环境仍为docker镜像，请下载最新镜像，按以下方法创建新的容器，并建立volumn映射(参考资料)，在本地编写代码，并在容器内进行编译检查。

```
### 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               2020                678605140682        46 hours ago        2.89GB

### 创建新的容器，同时建立volumn映射
$ docker run -it -v `pwd`:/home/oslab/lab1 -u oslab -w /home/oslab 6786 /bin/bash
oslab@3c1da3906541:~$


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

### 在docker中确认是否挂载成功
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Makefile

在了解了make的使用以及Makefile的基本知识之后（参考**3.3** Makefile介绍），并参照**3.4**节文件结构介绍。实验给出以下文件(https://gitee.com/zjuicsr/lab20fall-stu)

```
main.c
test.h
test.c
vmlinux.lds
```

4.2.1 组织文件结构

请参照3.4节文件结构，并组织形成对应的目录结构。其中main.c,test.c,test.h以及vmlinux.lds已经给出，在对应目录下添加空的Makefile文件，并添加空的head.S文件，形成如下目录结构，并确认容器中对应目录下同步成功。

```
lab1
├── arch
|   └── riscv
|       ├── boot
|       ├── include
|       ├── kernel
|       |   ├── head.S
|       |   ├── Makefile
|       |   └── vmlinux.lds
|       └── Makefile
├── include
|   └── test.h
├── init
|   ├── main.c
|   ├── Makefile
|   └── test.c
└── Makefile
```

4.2.2 编写各级目录Makefile

需要你书写各级目录下的Makefile，用来控制整个工程，需要实现的功能有：

- 能通过lab1目录（即工程中顶级目录）来实现Makefile的层级调用。
- 能够通过(main.c，vmlinux.lds，以及head.S)来生成vmlinux, System.map, Image等文件到相应位置。
- 实现make clean来清除文件，实现make run来直接运行(参考**4.4**)。

基本思路如下（为了描述方便，文档中Makefile_dir表示dir目录下的Makefile）：首先清楚Makefile的层级调用关系如下，其中顶级目录下的Makefile_lab1控制Makefile_init和Makefile_riscv，Makefile_riscv控制Makefile_kernel。

```
lab1
├── arch
|   └── riscv
|       ├── kernel
|       |   ├── Makefile_kernel
|       └── Makefile_riscv
├── init
|   ├── Makefile_init
└── Makefile_lab1
```

接下来每个Makefile需要实现的功能如下：

- Makefile_lab1：设置编译需要的变量，并利用make -C调用其需要控制的其他Makefile
- Makefile_init：利用main.c和test.c生成main.o和test.o
- Makefile_kernel：利用head.S生成head.o
- Makefile_riscv：使用ld并将main.o，test.o，head.o等目标文件以及lds文件（利用ld的-T选项）生成vmlinux文件到指定目录下，并使用OBJCOPY利用vmlinux生成Image文件到指定目录。

其中顶层Makefile中的变量可以参考以下设置：

```
export
CROSS_= riscv64-unknown-elf-
AR=${CROSS_}ar
GCC=${CROSS_}gcc
LD=${CROSS_}ld
OBJCOPY=${CROSS_}objcopy

ISA ?= rv64imafd
ABI ?= lp64

INCLUDE = -I ../include
CF =  -O3 -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections
CFLAG = ${CF} ${INCLUDE}
```

4.3 编写head.S

4.3.1 学习汇编指令

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。

4.3.2 学习mode切换

请参考寄存器介绍部分，如何设置mtvec可以使得CPU从machine mode切换到supervisor mode，切换后如何设置stvec。

4.3.3 完成head.S

编写head.S，并且实现以下功能：

- 设置mstatus寄存器，关闭mie和mip。
- 设置mtvec，使cpu从machine mode切换到supervisor mode，然后设置stvec。

- 设置sp寄存器
- 跳转到main.c中给出的start kernel函数

4.4 编译及测试

利用make调用Makefile文件完成整个工程的编译过程，利用qemu启动kernel（参考lab0），如果kernel能正常运行，并成功执行了start_kernel中的函数（即打印字符串），那么实验成功。

```
### 此时应提前设置好path等环境变量（export PATH=$PATH:/opt/riscv/bin）
# QEMU RUN (即make run)
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux
# QEMU DEBUG (可选配置为make debug)
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -S -s
```

预期的实验结果：

```
oslab@3c1da3906541:~/lab1$ make run
qemu-system-riscv64: warning: No -bios option specified. Not loading a firmware.
qemu-system-riscv64: warning: This default will change in a future QEMU release. Please use
the -bios option to avoid breakages when this happens.
qemu-system-riscv64: warning: See QEMU's deprecation documentation for details.
Hello RISC-V!
```

5. 实验任务与要求

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）