Lab 1: RV64 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

Docker 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](https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md)
* [RISC-V Unprivileged Spec](https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf)
* [RISC-V Privileged Spec](https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMFDQC-and-Priv-v1.11/riscv-privileged-20190608.pdf)

**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。mstatus寄存器的MPP位（共 2 bits），反映了当前(Machine Previous Privilege)特权模式：

| **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

计算机电源开启，进行初始化时，系统处于M模式。

**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。

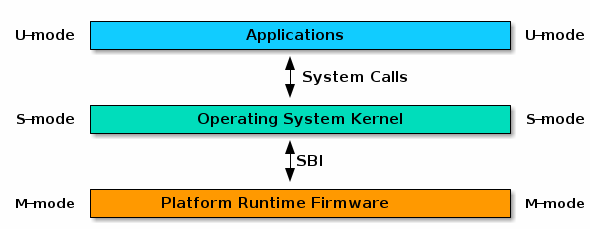
这个过程简单而言就是这样：

|  |  |
| --- | --- |
| 1  2  3  4 | 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](https://riscv.org/wp-content/uploads/2019/12/Summit_bootflow.pdf)。

**3.5 Makefile**

Makefile 可以简单地认为是一个编译一个project文件的编译规则，描述了整个project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project。在阅读了[Makefile介绍](https://seisman.github.io/how-to-write-makefile/introduction.html)这一章节后，同学们可以根据project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](https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html)。

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段、.rodata段、.data段和.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](https://sourceware.org/binutils/docs/ld/Scripts.html)。

**vmlinux**

vmlinux 通常指Linux Kernel编译生成的可执行文件(ELF，即Executable and Linkable Format)，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**

基于Lab0已经建立的环境。

从[repo](https://gitee.com/zjusec/os21fall)同步实验代码框架，参考Lab0中，将project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。

$ git clone https://gitee.com/zjusec/os21fall

$ cd os21fall/src/lab1

$ ls -l # 已经构建完成的根文件系统的镜像

drwxrwxr-x 3 lisp lisp 4096 8月 3 15:44 arch

drwxrwxr-x 2 lisp lisp 4096 8月 3 15:44 include

drwxrwxr-x 2 lisp lisp 4096 8月 3 15:44 init

drwxrwxr-x 2 lisp lisp 4096 8月 3 15:44 lib

-rw-rw-r-- 1 lisp lisp 971 8月 3 15:44 Makefile

|  |
| --- |
| ├── 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 |

不妨在家目录下设立lab1。

创建、进入docker容器：

$ cd ~/lab1

$ cp -r ~/os21fall/src/lab1 ~/.

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

**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），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。

示例代码：

.extern start\_kernel

.section .text.entry

.globl \_start

\_start:

la sp, boot\_stack\_top

j start\_kernel

.section .bss.stack

.globl boot\_stack

boot\_stack:

.space 4096

.globl boot\_stack\_top

boot\_stack\_top:

**4.3 完善 Makefile 脚本**

阅读文档中关于[Makefile](https://zjusec.gitee.io/os21fall/lab1/" \l "35-makefile)的章节，以及project文件中的Makefile文件，根据注释学会Makefile的使用规则后，补充lib/Makefile，使得以编译。

示例代码：

C\_SRC = $(sort $(wildcard \*.c))

OBJ = $(patsubst %.c,%.o,$(C\_SRC))

all:$(OBJ)

%.o:%.c

${GCC} ${CFLAG} -c $<

clean:

$(shell rm \*.o 2>/dev/null)

完成此步后在Lab1根文件夹执行make，可以看到成功编译生成了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：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | 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。

示例代码：

#include "types.h"

#include "sbi.h"

struct sbiret sbi\_ecall(int ext, int fid, unsigned long arg0,

unsigned long arg1, unsigned long arg2,

unsigned long arg3, unsigned long arg4,

unsigned long arg5) {

struct sbiret ret;

register uint64 a0 asm ("a0") = (uint64)(arg0);

register uint64 a1 asm ("a1") = (uint64)(arg1);

register uint64 a2 asm ("a2") = (uint64)(arg2);

register uint64 a3 asm ("a3") = (uint64)(arg3);

register uint64 a4 asm ("a4") = (uint64)(arg4);

register uint64 a5 asm ("a5") = (uint64)(arg5);

register uint64 a6 asm ("a6") = (uint64)(fid);

register uint64 a7 asm ("a7") = (uint64)(ext);

asm volatile("ecall"

: "+r" (a0), "+r" (a1)

: "r" (a0),"r" (a1),"r" (a2),"r" (a3),"r" (a4),"r" (a5),"r" (a6),"r" (a7)

: "memory");

ret.error = a0;

ret.value = a1;

return ret;

}

void sbi\_putchar(unsigned c) {

sbi\_ecall(SBI\_PUTCHAR, 0, c, 0, 0, 0, 0, 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。

示例代码：

#include "print.h"

#include "sbi.h"

void putc(char c) {

sbi\_putchar(c);

}

void puts(char \*s) {

while((\*s) != '\0') {

sbi\_putchar(\*s);

s++;

}

}

void puti(int x) {

int digit = 1;

int tmp = x;

while (tmp >= 10) {

digit \*= 10;

tmp /= 10;

}

while (digit >= 1) {

sbi\_putchar('0' + x/digit);

x %= digit;

digit /= 10;

}

}

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](https://zjusec.gitee.io/os21fall/lab1/" \l "36)。

学习了解了以上知识后，补充arch/riscv/include/defs.h中的代码，补充完整read\_csr这个宏定义。

[示例](https://zjusec.gitee.io/os21fall/lab1/" \l "_2)代码：

#ifndef \_DEFS\_H

#define \_DEFS\_H

#include "types.h"

#define csr\_read(csr) \

({ \

register uint64 \_\_v; \

asm volatile ("csrr %0, " #csr \

: "=r" (\_\_v) : \

: "memory"); \

\_\_v; \

})

#define csr\_write(csr, val) \

({ \

uint64 \_\_v = (uint64)(val); \

asm volatile ("csrw " #csr ", %0" \

: : "r" (\_\_v) \

: "memory"); \

})

#endif

**4.7 编译内核**

创建docker容器。建议在虚拟机平台上启用2个Terminal联动，编译调试时一个在容器外，一个在容器内。

$ cd ~/lab1

$ docker run --name oslab -it -v `pwd`:/home/lab1 -w /home/lab1 alphavake/oslab bash

root@8af22831a5d5:/home/lab1#

建议先直接编译、运行一次，如果存在bug则逐个消灭：

# make run

编译成功后，再执行调试方式：

（注意：必须在编译开关CF中定义“-g”这个option，才能让程序带上符号信息）

### Terminal 1

#make debug

### Terminal 2

# riscv64-unknown-linux-gnu-gdb vmlinux

(gdb) target remote:1234 # 连接 qemu

(gdb) b start\_kernel # 设置断点

(gdb) continue # 继续执行

(gdb) quit # 退出 gdb

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

作业提交

同学需要提交实验报告以及整个工程代码。在提交前请使用 make clean 清除所有构建产物。