

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

Environment in Lab0

3 实验步骤

3.1 环境搭建

从 repo 同步实验代码框架，参考 Lab0 中，将工程代码映射进容器中。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。

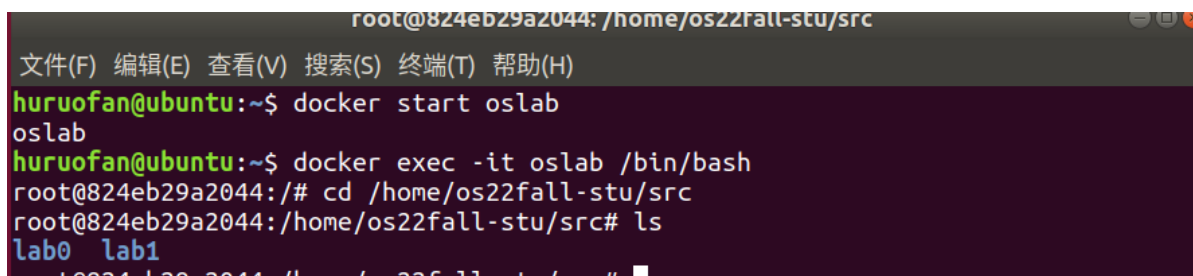
用Lab0的方法打开容器并从终端连入容器。

```
docker start oslab
docker exec -it oslab /bin/bash
export RISCVC=/opt/riscv
export PATH=$PATH:$RISCVC/bin
```

使用git命令时，用fetch命令进行同步更新更为安全。

```
git remote -v
git fetch origin master
git log -p master.. origin/master
git merge origin/master
```

操作后，如图

A terminal window screenshot showing the setup of a Docker container named 'oslab'. The user 'huruofan@ubuntu' runs 'docker start oslab' and then 'docker exec -it oslab /bin/bash' to enter the container. Inside the container, the user runs 'cd /home/os22fall-stu/src' and 'ls', which shows files 'lab0' and 'lab1'. The prompt changes from '~\$' to 'root@824eb29a2044: /home/os22fall-stu/src#'.

```
root@824eb29a2044: /home/os22fall-stu/src
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
huruofan@ubuntu:~$ docker start oslab
oslab
huruofan@ubuntu:~$ docker exec -it oslab /bin/bash
root@824eb29a2044:/# cd /home/os22fall-stu/src
root@824eb29a2044:/home/os22fall-stu/src# ls
lab0 lab1
root@824eb29a2044:/home/os22fall-stu/src#
```

3.2 编写head.S

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。

这里具体实现，就是la存储，jal跳转，以及4KB的4096设置

```

.extern start_kernel

        .section .text.entry
        .globl _start
_start:
    la sp, boot_stack_top
    jal start_kernel

        .section .bss.stack
        .globl boot_stack
boot_stack:
    .space 4096

        .globl boot_stack_top
boot_stack_top:

```

3.3 完善 Makefile 脚本

补充lib/Makefile

```

C_SRC = $(sort $(wildcard *.c))
#将所有的工作目录下的.C文件排序
OBJ = $(patsubst %.c,%.o,$(C_SRC))
#模式替换, 将所有.c转换到.o
all:$(OBJ)

%.o:%.c
    ${GCC} ${CFLAG} -c $<
#把源码文件编译成 .o 对象文件, 不链接
#"$<"表示所有的依赖目标集
clean:
    $(shell rm *.o 2>/dev/null)
#错误定向

```

3.4 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);

```

修改后的代码如下:

```

#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 a6 asm ("a6") = (uint64)(fid);
    register uint64 a7 asm ("a7") = (uint64)(ext);
    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);
    //按要求放入寄存器

    asm volatile("ecall"
        : "+r" (a0), "+r" (a1)//输出操作数
        : "r" (a0), "r" (a1), "r" (a2), "r" (a3), "r" (a4), "r" (a5), "r" (a6), "r"
(a7)//输入操作数
        : "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);
    //按要求调用sbi_ecall(0x1, 0x0, 0x30, 0, 0, 0, 0, 0)
}

```

3.5 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。

```

#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;
}
//进行int的转换
while (digit >= 1) {
    sbi_putchar('0' + x/digit);
    x %= digit;
    digit /= 10;
}
}

```

3.6 修改 defs

```

#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 ("csw " #csr ", %0" \
                  : : "r" (__v) \
                  : "memory"); \
})

#endif

```

3.7 展示结果

```
root@824eb29a2044: /home/os22fall-stu/src/lab1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
root@824eb29a2044:/home/os22fall-stu/src/lab1# export RISCVM=/opt/riscv
root@824eb29a2044:/home/os22fall-stu/src/lab1# export PATH=$PATH:$RISCVM/bin
root@824eb29a2044:/home/os22fall-stu/src/lab1# make run
make -C lib all
make[1]: Entering directory '/home/os22fall-stu/src/lab1/lib'
make[1]: 'all' is up to date.
make[1]: Leaving directory '/home/os22fall-stu/src/lab1/lib'
make -C init all
make[1]: Entering directory '/home/os22fall-stu/src/lab1/init'
make[1]: 'all' is up to date.
make[1]: Leaving directory '/home/os22fall-stu/src/lab1/init'
make -C arch/riscv all
make[1]: Entering directory '/home/os22fall-stu/src/lab1/arch/riscv'
make -C kernel all
make[2]: Entering directory '/home/os22fall-stu/src/lab1/arch/riscv/kernel'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/os22fall-stu/src/lab1/arch/riscv/kernel'
riscv64-unknown-elf-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o -o ../../vmlinux
riscv64-unknown-elf-objcopy -O binary ../../vmlinux ./boot/Image
nm ../../vmlinux > ../../System.map
make[1]: Leaving directory '/home/os22fall-stu/src/lab1/arch/riscv'

Build Finished OK
Launch the qemu .....
```

```
root@824eb29a2044: /home/os22fall-stu/src/lab1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)

Build Finished OK
Launch the qemu .....

OpenSBI v0.6

      _ _ _ _ _
     / / / / /
    / / / / /
   / / / / /
  / / / / /
 / / / / /
/_/_/_/_/_

Platform Name       : QEMU Virt Machine
Platform HART Features : RV64ACDFIMSU
Platform Max HARTs   : 8
Current Hart        : 0
Firmware Base       : 0x80000000
Firmware Size       : 120 KB
Runtime SBI Version  : 0.2

MIDELEG : 0x0000000000000222
MEDELEG : 0x0000000000000b109
PMP0    : 0x0000000080000000-0x000000008001ffff (A)
PMP1    : 0x0000000000000000-0xffffffffffff (A,R,W,X)
2022
Hello RISC-V
```

4 心得与思考

1. 请总结一下 RISC-V 的 calling convention，并解释 Caller / Callee Saved Register 有什么区别？

函数调用规范(Calling convention)如下：

- 把函数参数放到函数能访问的地方
- 把控制权给函数（使用 `jal` 指令）
- 拿到 memory 中的资源（获取函数需要的局部存储资源，按需保存寄存器）
- 运行函数中的指令
- 把值写到 memory/register 中（将返回值存储到调用者能够访问到的位置，恢复寄存器，释放局部存储资源）
- 返回（`ret` 指令）

Caller-saved register 用于保存不需要在各个调用之间保留的临时数据。

Callee-saved register 用于保存应在每次调用中保留的长寿命值。

由于有些临时变量是不需要存储的，所以不需要保存，这就是区分这两个寄存器的意义。

2. 编译之后，通过 System.map 查看 vmlinux.lds 中自定义符号的值。

```
root@824eb29a2044: /home/os22fall-stu/src/lab1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
0000000080200000 A BASE_ADDR
0000000080203000 B _ebss
0000000080202000 R _edata
0000000080203000 B _kernel
000000008020100f R _erodata
000000008020033c T _etext
0000000080202000 B _sbss
0000000080202000 R _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080202000 B boot_stack
0000000080203000 B boot_stack_top
0000000080200204 T puti
0000000080200138 T puts
000000008020000c T sbi_ecall
00000000802000e8 T start_kernel
0000000080200128 T test
~
~
~
~
"System.map" 19L, 518C 1,1 All
```

可以看见T,B,D,R四类主要的变量的地址和puti, puts, sbi_ecall, sbi_putchar, start_kernel等的值

3. 用 read_csr 宏读取 sstatus 寄存器的值，对照 RISC-V 手册解释其含义。

打印，值为24576

4. 用 write_csr 宏向 sscratch 寄存器写入数据，并验证是否写入成功。

写入0x222222，打印，发现十进制表示的数字与其相等

```
#include "print.h"
#include "sbi.h"
#include "defs.h"

extern void test();

int start_kernel() {
    puti(2022);
    puts("Hello RISC-V\n");

    uint64 t = csr_read(sstatus);
    puti(t);

    csr_write(sscratch, 0x222222);

    test(); // DO NOT DELETE !!!

    return 0;
}
```



```
Usage: riscv64-unknown-linux-gnu-objdump <option(s)> <file(s)>
Display information from object <file(s)>.
At least one of the following switches must be given:
-a, --archive-headers    Display archive header information
-f, --file-headers       Display the contents of the overall file header
-p, --private-headers    Display object format specific file header contents
-P, --private=OPT,OPT... Display object format specific contents
-h, --[section-]headers  Display the contents of the section headers
-x, --all-headers        Display the contents of all headers
-d, --disassemble       Display assembler contents of executable sections
-D, --disassemble-all   Display assembler contents of all sections
--disassemble=<sym>     Display assembler contents from <sym>
-S, --source             Intermix source code with disassembly
--source-comment[=<txt>] Prefix lines of source code with <txt>
-s, --full-contents      Display the full contents of all sections requested
-g, --debugging          Display debug information in object file
-e, --debugging-tags     Display debug information using ctags style
-G, --stabs              Display (in raw form) any STABS info in the file
-W[LLIaprmFFsoORtUuTgAckK] or
--dwarf[=rawline,=decodedline,=info,=abbrev,=pubnames,=aranges,=macro,=frames,
=frames-interp,=str,=str-offsets,=loc,=Ranges,=pubtypes,
=gdb_index,=trace_info,=trace_abbrev,=trace_aranges,
=addr,=cu_index,=links,=follow-links]
--ctf=SECTION           Display CTF info from SECTION
-t, --syms              Display the contents of the symbol table(s)
-T, --dynamic-syms      Display the contents of the dynamic symbol table
-r, --reloc             Display the relocation entries in the file
-R, --dynamic-reloc     Display the dynamic relocation entries in the file
@<file>                 Read options from <file>
-v, --version           Display this program's version number
-i, --info              List object formats and architectures supported
-H, --help              Display this information
```

```
root@824eb29a2044:/home/os22fall-stu/src/lab1# riscv64-unknown-linux-gnu-objdump -x vmlinux

vmlinux:      file format elf64-littleriscv
vmlinux
architecture: riscv:rv64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0000000000200000

Program Header:
  LOAD off 0x0000000000001000 vaddr 0x0000000000200000 paddr 0x0000000000200000 align 2**12
    filesz 0x000000000000100f memsz 0x000000000000100f flags r-x
  LOAD off 0x0000000000000000 vaddr 0x0000000000202000 paddr 0x0000000000202000 align 2**12
    filesz 0x0000000000000000 memsz 0x0000000000001000 flags rw-

Sections:
Idx Name      Size      VMA           LMA           File off  Algn
0 .text      0000033c  0000000000200000 0000000000200000 00001000  2**12
CONTENTS, ALLOC, LOAD, READONLY, CODE
1 .rodata    0000000f  0000000000201000 0000000000201000 00002000  2**12
CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .bss       00001000  0000000000202000 0000000000202000 00003000  2**12
ALLOC
3 .riscv.attributes 00000030 0000000000000000 0000000000000000 0000200f  2**0
CONTENTS, READONLY
4 .comment   00000012 0000000000000000 0000000000000000 0000203f  2**0
CONTENTS, READONLY
5 .debug_line 000002c6 0000000000000000 0000000000000000 00002051  2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
6 .debug_info 000002dc 0000000000000000 0000000000000000 00002317  2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
7 .debug_abbrev 000001c6 0000000000000000 0000000000000000 000025f3  2**0
CONTENTS, READONLY, DEBUGGING, OCTETS
8 .debug_aranges 00000100 0000000000000000 0000000000000000 000027c0  2**4
CONTENTS, READONLY, DEBUGGING, OCTETS
9 .debug_str 0000019f 0000000000000000 0000000000000000 000028c0  2**0
```

- 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。

```
root@824eb29a2044:/home/os22fall-stu/src/lab1# file vmlinux
vmlinux: ELF 64-bit LSB executable, UCB RISC-V, version 1 (SYSV), statically linked, with debug_info, not stripped
```

vmlinux 是ELF 文件，是编译出来最原始的文件，ELF 文件由4 部分组成，分别是ELF 头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。

vmlinux 与实验所用连接脚本(vmlinux.lds)之间的联系:

Linux 启动后，会启动内核编译后的文件vmlinux，vmlinux 会按照vmlinux.lds 设定的规则，进行链接，vmlinux.lds 是vmlinux.lds.S 编译之后生成的，为了确定vmlinux 内核的起始地址，首先通过vmlinux.lds.S 链接脚本进行分析

- 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

ARM32:/lab0/linux-5.19.8/arch/arm/tools/syscall.tbl

RISC-V(32 bit),RISC-V(64 bit):/lab0/linux-5.19.8/arch/riscv/include/asm/unistd.h

x86(32 bit):/lab0/linux-5.19.8/arch/x86/entry/syscalls/syscall_32.tbl

x86_64:/lab0/linux-5.19.8/arch/x86/entry/syscalls/syscall_64.tblhttps://elixir.bootlin.com/linux/v6.0-rc6/source/arch/riscv/kernel/syscall_table.c

心得：在本次实验中，有几个地方比较困难。首先难住我的就是更新仓库，由于要把lab1拷贝进来，查找了许多语句。然后就是这次涉及到的很多知识，虽然程序代码都不是很长，但是背后要记要学的东西却十分多。比如riscv指令的head.S，要写的十分少，但是理解那些.*一类却很复杂，特别是空格有关的规范，一不小心就会一直报错不能通过。Makefile的话看了操作手册，虽然也觉得很困难，但写起来还算有数。另外的.c设计倒是容易了不少，主要这和平时写的代码也接近。内联汇编也是一个很新的知识，主要要注意其中的格式。