浙江大学

本科实验报告

课程名称：操作系统

姓　名：

学　院：计算机科学与技术学院

系：计算机科学与技术系

专　业：计算机科学与技术

学　号：

指导教师：夏莹杰

2022 年 10 月 19 日

浙江大学操作系统实验报告

实验名称：RV64 内核引导

电子邮件地址：　　手机：

实验地点： 曹西 503　　实验日期：2022 年 10 月 19 日

一、实验目的和要求

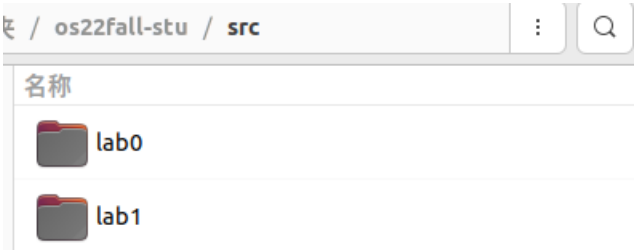
1. 配置 Linux 环境学习 RISC-V 汇编， 编写 head.S 实现跳转到内核运行的第一个 C 函数。
2. 学习 OpenSBI，理解 OpenSBI 在实验中所起到的作用，并调用 OpenSBI 提供的接口完成字符的输出。
3. 学习 Makefile 相关知识， 补充项目中的 Makefile 文件， 来完成对整个工程的管理。

二、实验过程

1. 准备工程
   ① 进入本地的项目文件夹，利用 git fetch 和 git reset 合并本地分支和更新后的 master 分支。

```
zarin@ZaricCarpathia:~/os22fall-stu$ git fetch --all
正在获取 origin
zarin@ZaricCarpathia:~/os22fall-stu$ git reset --hard origin/master
HEAD 现在位于 332730c add lab2 ddl
```

   ② 可以从文件系统进入项目文件夹，确认合并已成功。



2. 编写 head.S
   ① 使用.space 指示符指示栈所需的内存大小，使用 la 伪指令让 sp 指向这段内存的结束（即 boot_stack_top）处即可设置程序栈。之后使用 j 伪指令让内核从 start_kernel 开始运行即可。代码如下：

```
.extern start_kernel

    .section .text.entry
    .globl _start
_start:
    # ------------------
    # - your code here -
    # ------------------
    la sp, boot_stack_top
    j start_kernel

    .section .bss.stack
    .globl boot_stack
boot_stack:
    .space 4096 * 4 # <-- change to your stack size

    .globl boot_stack_top
boot_stack_top:
```

3. 完善 makefile 脚本

① 参考其他目录的 makefile 文件，编写脚本如下：

```
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)
```

② 在 docker 容器中进入 lab1 目录，运行 make 命令，即可看到编译完成的 vmlinux。

4. 补充 sbi.c

① 参考实验指导书中有关内嵌汇编的章节，编写代码如下。

```
#include "types.h"
#include "sbi.h"


struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
                                    uint64 arg1, uint64 arg2,
                                    uint64 arg3, uint64 arg4,
                                    uint64 arg5)
{
    struct sbiret ret;
    uint64 uext = ext;
    uint64 ufid = fid;
    __asm__ volatile(
        "mv a7, %[uext]\n"
        "mv a6, %[ufid]\n"
        "mv a0, %[arg0]\n"
        "mv a1, %[arg1]\n"
        "mv a2, %[arg2]\n"
        "mv a3, %[arg3]\n"
        "mv a4, %[arg4]\n"
        "mv a5, %[arg5]\n"
        "ecall\n"
        "mv %[error], a0\n"
        "mv %[value], a1\n"
        : [error] "=r" (ret.error), [value] "=r" (ret.value)
        : [uext] "r" (uext), [ufid] "r" (ufid), [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2), [arg3] "r" (arg3), [arg4] "r" (arg4), [arg5] "r" (arg5)
        : "memory"
    );
    return ret;
}
```

5. puts()和 puti()

① 编写完 sbi_ecall()后，可在思路上将调用等效为 putchar()，编写 puts()。

```
4 void puts(char *s) {
5     for(int i = 0; s[i]; ++i){
6         sbi_ecall(0x1, 0x0, s[i], 0, 0, 0, 0, 0);
7     }
8 }
```

② 同理，编写 puti()。

```
10 void puti(int x) {
11     if(x==0) sbi_ecall(0x1, 0x0, '0', 0, 0, 0, 0, 0);
12     else{
13         int stack[10];
14         int num=0;
15         if(x<0) sbi_ecall(0x1, 0x0, '-', 0, 0, 0, 0, 0);
16         while(x){
17             stack[num++]=x%10;
18             x/=10;
19         }
20         for(int i = num - 1; i >= 0; --i){
21             sbi_ecall(0x1, 0x0, '0' + stack[i], 0, 0, 0, 0, 0);
22         }
23     }
24 }
```

6. 修改 defs

① 参考 csr_write 的编写方式，编写宏 csr_read 如下。

```
6 #define csr_read(csr)                           \
7 ({                                              \
8     register uint64 __v;                        \
9     asm volatile ("csrr %0, " #csr              \
10                   : "=r" (__v)                 \
11                   : : "memory");               \
12     __v;                                        \
13 })
```

② 保存后回到 lab1 目录，运行 make 和 make run 指令。效果如下，可见实验取得成功。

```
Domain0 SysReset          : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
2022 Hello RISC-V
This is a message from 3200105646.
```

三、讨论和心得

本次实验似乎没有什么太多需要讨论的。

问题主要是：对 riscv 汇编的知识不足，不太理解进行指示符的作用；同时也不太知道计组和体系课上用不到的伪指令，需要通过手册和其他途径进行一定程度的学习。

四、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？
   ① RISC-V 保持 c 数据的类型的自然对齐和堆栈的自然对齐；在寄存器上主要把 register 分为 caller 和 callee saved。
   ② Caller saved registers 需要调用函数在函数运行结束时恢复原值，callee 则不需要（由调用者保存）。
2. 编译之后，通过 System.map 查看 vmlinux.lds 中自定义符号的值
   ① 如下。

```
1 0000000080200000 A BASE_ADDR
2 0000000080206000 B _ebss
3 0000000080202000 R _edata
4 0000000080206000 B _ekernel
5 0000000080201033 R _erodata
6 0000000080200314 T _etext
7 0000000080202000 B _sbss
8 0000000080202000 R _sdata
9 0000000080200000 T _skernel
10 0000000080201000 R _srodata
11 0000000080200000 T _start
12 0000000080200000 T _stext
13 0000000080202000 B boot_stack
14 0000000080206000 B boot_stack_top
15 00000000802001c4 T puti
16 000000008020013c T puts
17 000000008020000c T sbi_ecall
18 00000000802000e0 T start_kernel
19 000000008020012c T test
```

五、附录

用到的参考资料：

RISC-V 汇编快速入门 | Half Coder (lgl88911.github.io)