LAB 4

姓名：张云策 学号：3200105787 学院：计算机科学与技术学院

课程名称：计算机系统II 同组学生姓名：/

实验时间：2021.12 实验地点：紫金港机房 指导老师：申文博

一、 实验目的和要求

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。

二、 主要仪器设备

Docker in Lab3

三、 操作方法与实验步骤

4.1 操作方法

补充文件，使得 Linux 打印出字符串。

4.2 实验步骤

1. 准备 Haed.S

```
.extern start_kernel
.extern stack_top
_start:
    li t1, 0x8              #t1=1000
    csrc mstatus, t1        #将mstatus寄存器的第三位置0（1的对应位清零）
    li t1, 0x888            #t1=1000 1000 1000
    csrc mie, t1            #将mie寄存器的第11、7、3位置0（1的对应位清零）

    la t1, _mtrap           #t1=_mtrap
    slli t1, t1, 2          #t1左移2位
    csrw mtvec, t1          #mtvec的第0、1位置0（所有异常均跳转到一个pc地址），高30位为pc地址


    li t1, 0x800            #t1=1000 0000 0000  supervisor mode = 01
    csrw mstatus, t1        #将mstatus的MPP域改为01

    la t1, _supervisor      #t1=_supervisor
    csrw mepc, t1           #将t1的值写入mepc（出现异常的返回地址，用mret必须要有的）
    mret

_supervisor:

    la t1, _strap           #t1=_strap
    slli t1, t1, 2          #t1左移2位
    csrw stvec, t1          #stvec的第0、1位置0（所有异常均跳转到一个pc地址），高30位为pc地址

    la sp, stack_top        #sp=stack_top的地址（设置栈环境）
    j start_kernel          #跳转到main.c中的start_kernel函数
```

2.准备 lib/Makefile

```
# makefile_lab4
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 -nos
CFLAG = ${CF} ${INCLUDE}

.PHONY: run debug clean
run:
    @make -C init -s
    @make -C arch/riscv -s
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux


debug:
    @qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -S -s

clean:
    @make clean -C init -s
    @make clean -C arch/riscv -s
```

编译 vmLinux 成功

3. 完成 sbi_ecall

```c
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 uintptr_t a0 asm ("a0") = (uintptr_t)(arg0);
    register uintptr_t a1 asm ("a1") = (uintptr_t)(arg1);
    register uintptr_t a2 asm ("a2") = (uintptr_t)(arg2);
    register uintptr_t a3 asm ("a3") = (uintptr_t)(arg3);
    register uintptr_t a4 asm ("a4") = (uintptr_t)(arg4);
    register uintptr_t a5 asm ("a5") = (uintptr_t)(arg5);
    register uintptr_t a6 asm ("a6") = (uintptr_t)(fid);
    register uintptr_t a7 asm ("a7") = (uintptr_t)(ext);
    asm volatile ("ecall"
                : "+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;
}
```

4. 实现 print.c 中的 puts（）和 puti（）

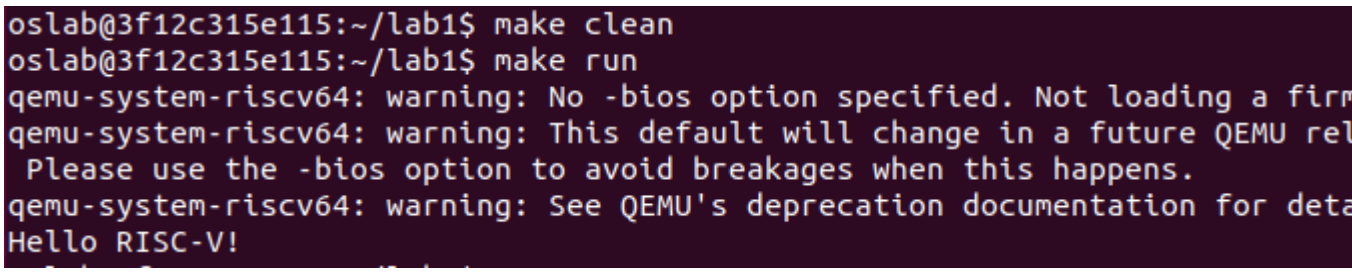
```c
#include "print.h"
#include "sbi.h"

void puts(char *s) {
    do{
        sbi_call(1, (uint64_t)*s, 0, 0);
        s = s + 1;
    }while (*s != '\0');

    return 0;
}

void puti(int x) {
    char str[100];
    char ustr[100];
    int m;
    int i = 0;
    int j = 0;
    do{
        m = n%10;
        n = n/10;
        ustr[i] = '0'+m;
        i++;
    }while (x != 0);
    for (i = i - 1; i >= 0; i--){
        str[j] = ustr[i];
        j++;
    }
    str[j] = '\0';
    puts(str);
    return 0;
    }

```

5.修改 defs.h

```c
#ifndef _DEFS_H
#define _DEFS_H

#include "types.h"

#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    typedef unsigned long long uint64_t;    \
    __v;                                    \
})

#define csr_write(csr, val)                         \
({                                                  \
    uint64 __v = (uint64)(val);                     \
    asm volatile ("csrw " #csr ", %0"               \
                  : : "r" (__v)                     \
                  : "memory");                      \
})

#endif

```

四、 实验结果与分析

将 main.c 中的内容改为 Hello RISC-V，经 make clean/ make run 操作后，可以看到输出字符串。



五、 讨论、心得

思考题 1：

RISC v 的调用规范:

RISC v 有 8 个整型寄存器（a0-a7），8 个浮点数寄存器（fa0-fa7），c 语言在 RISC v 平台上，除了指针变量和 long 变量之外，其余都不受指令长度影响；在函数调用中，结构体最多只有八个可以储存在寄存器中，其他放置在栈内，栈指针指向第一个放置在栈内的变量，而放置何种类型寄存器内由数据类型决定；小于一指针字节长度的参数被存在寄存器的低位，栈存放同理，而且参数如果字长是指针两倍，则分开存放，如果大于两倍，则通过内存引用传递参数；函数结束的返回值储存在相应的寄存器中，如果大于两倍指针则存放在内存中。

Caller saved register and Callee saved register

Caller 主要为调用者自行保存，在调用其他进程时，必须保存点用着所要保存到寄存器，便于后期再次访问调用者。而 Callee 为被调用者自行保存，被调用的进程保存在所要被调用的寄存器中，在进程结束后返回，恢复寄存器内容。

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

Table 18.2: RISC-V calling convention register usage.

思考题 2