浙江大学

本科实验报告

课程名称：	操作系统
实验名称：	RV64 内核引导与时钟中断处理
姓　　名：	蔡雨谦
学　　院：	计算机学院
系：	计算机系
专　　业：	计算机科学与技术
学　　号：	3210102466
指导教师：	寿黎但

2023 年　　9 月　　28 日

浙江大学实验报告

一、 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。
- 学习 RISC-V 的 trap 处理相关寄存器与指令，完成对 trap 处理的初始化。
- 理解 CPU 上下文切换机制，并正确实现上下文切换功能。
- 编写 trap 处理函数，完成对特定 trap 的处理。
- 调用 OpenSBI 提供的接口，完成对时钟中断事件的设置。

二、 实验环境

- Ubuntu 22.04.2 LTS Windows Subsystem for Linux 2

三、 操作方法与实验步骤

1. 准备工程

从课程提供的 github 仓库获取实验代码框架，准备进行后续实验。

2. RV64 内核引导

实现 RV64 内核引导需要完善的文件：

- arch/riscv/kernel/head.S
- lib/Makefile
- arch/riscv/kernel/sbi.c
- arch/riscv/include/defs.h

2.1 编写 head.S

head.S 的作用是引导内核的启动。设置一个大小为 4kb(4096)的栈，并通过跳转指令跳转到 start_kernel 函数。

arch > riscv > kernel > head.S

```
1  .extern start_kernel
2
3      .section .text.entry
4      .globl _start
5  _start:
6      # ------------------
7      la sp, boot_stack_top
8      jal x0, start_kernel
9      # ------------------
10
11     .section .bss.stack
12     .globl boot_stack
13 boot_stack:
14     .space 4096 # <-- change to your stack size
15
16     .globl boot_stack_top
17 boot_stack_top:
```

2.2 完善 makefile 脚本

参照已经给出的其他 makefile 文件，将目录下的全部.c 文件编译成.o 文件。

lib > Makefile

```
C_SRC       = $(sort $(wildcard *.c))
OBJ         = $(patsubst %.c,%.o,$(C_SRC))

file = printk.o
all: $(OBJ)

%.o: %.c
    ${GCC}  ${CFLAG} -c $<
clean:
    $(shell rm *.o 2>/dev/null)
'
```

运行 make all 指令可以进行编译。

```
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ make all
make -C lib all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/lib'
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-se
-nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab
rch/riscv/include -c printk.c
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/lib'
make -C init all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/init'
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sec
nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1
ch/riscv/include -c main.c
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sec
nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1
ch/riscv/include -c test.c
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/init'
make -C arch/riscv all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/arch/riscv'
make -C kernel all
make[2]: Entering directory '/home/icey/os23fall-stu/src/lab1/arch/riscv/kernel'
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-se
-nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab
rch/riscv/include -c head.S
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-se
-nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab
rch/riscv/include -c sbi.c
make[2]: Leaving directory '/home/icey/os23fall-stu/src/lab1/arch/riscv/kernel'
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o -o ../../v
riscv64-linux-gnu-objcopy -O binary ../../vmlinux ./boot/Image
nm ../../vmlinux >  ../../System.map
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/arch/riscv'

Build Finished OK
```

编译后会生成 vmlinux 内核文件，使用 file vmlinux 指令查看：

```
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ file vmlinux
vmlinux: ELF 64-bit LSB executable, UCB RISC-V, soft-float ABI, version 1 (SYSV), statically linked, with debug_info, not stripped
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$
```

2.3 补充 sbi.c

参照示例一，使用 asm volitile 向 C 代码中插入内联汇编代码。将 ext 放入寄存器 a7 中，fid 放入寄存器 a6 中，将 arg0 - arg5 放入寄存器 a0 - a5 中。使用 ecall 指令。ecall 之后系统会进入 M 模式。返回结果存放在寄存器 a0、a1 中，其中 a0 为 error，a1 为返回值。将 a0、a1 的值放进一个 sbiret 变量中作为 sbi_ecall 的返回值。

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

2.4 修改 defs.h

参照示例二，使用 asm volitile 插入内联汇编代码，完善 csr_read 宏，将 csr 寄存器的值读到变量_V 中。

```
#define csr_read(csr)                       \
({                                          \
    register uint64 __v;                    \
    /* unimplemented */                     \
    asm volatile (                          \
        "csrr %[v], " #csr                  \
        : [v] "=r" (__v)                     \
        :                                   \
        : "memory"                          \
    );                                      \
    __v;                                    \
})
```

代码框架中给出的 csr_write 宏则是将 val 的值写进 csr 寄存器中。

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

#endif
```

2.5 运行 RV64 内核。

使用 make all 指令编译项目文件夹得到的内核可以通过 make

run 指令运行。

```
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ make run
make -C lib all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/lib'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/lib'
make -C init all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/init'
make[1]: 'all' is up to date.
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/init'
make -C arch/riscv all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/arch/riscv'
make -C kernel all
make[2]: Entering directory '/home/icey/os23fall-stu/src/lab1/arch/riscv/kernel'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/home/icey/os23fall-stu/src/lab1/arch/riscv/kernel'
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o -o ../../vmlinux
riscv64-linux-gnu-objcopy -O binary ../../vmlinux ./boot/Image
nm ../../vmlinux >  ../../System.map
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/arch/riscv'

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

OpenSBI v0.9
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : timer,mfdeleg
Platform HART Count       : 1
Firmware Base             : 0x80000000
Firmware Size             : 100 KB
Runtime SBI Version       : 0.2

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
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
```

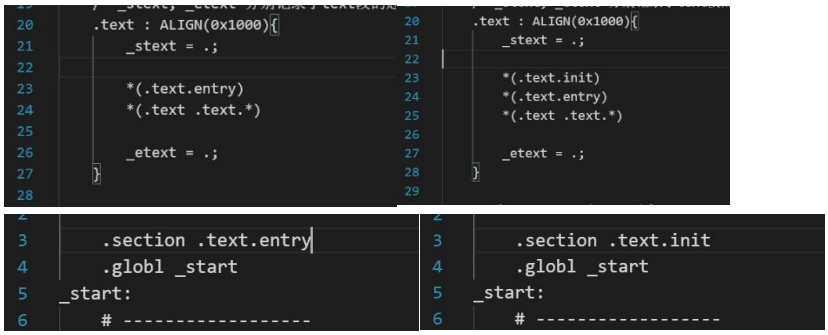
3. RV64 时钟中断处理

实现 RV64 时钟中断处理需要完善或添加的文件：

- arch/riscv/kernel/head.S
- arch/riscv/kernel/entry.S
- arch/riscv/kernel/trap.c
- arch/riscv/kernel/clock.c

3.1 准备工作

vmlinux.lds 与 head.S 文件中有需要修改的部分如下(左为修改前，右为修改后)：



3.2 开启 trap 处理

在运行 start_kernel 前需要在 head.S 中对一些寄存器进行初始化：

设置 stvec,将_traps 所表示的地址写入 stvec

```
# set stvec = _traps
la a0, _traps
csrw stvec, a0
```

开启时钟中断，将 sie[STIE]置 1，STIE 是第五位：

```
# set sie[STIE] = 1
li a0, 1<<5
csrw sie, a0
```

设置第一次时钟中断

```
# set first time interrupt
jal ra, clock_set_next_event
```

开启 S 态下的中断响应，将 sstatus[SIE]置 1，SIE 是第一位：

```
# set sstatus[SIE] = 1
li a0, 1<<1
csrw sstatus, a0
```

3.3 实现上下文切换

在 arch/riscv/kernel/ 目录下添加 entry.S 文件，文件包含以下内容：

保存 CPU 的寄存器到内存中

```
# 1. save 32 registers and sepc to stack
addi sp, sp, -256
sd x1, 0(sp)
sd x2, 8(sp)
sd x3, 16(sp)
sd x4, 24(sp)
sd x5, 32(sp)
sd x6, 40(sp)
sd x7, 48(sp)
sd x8, 56(sp)
sd x9, 64(sp)
sd x10, 72(sp)
sd x11, 80(sp)
sd x12, 88(sp)
sd x13, 96(sp)
sd x14, 104(sp)
sd x15, 112(sp)
sd x16, 120(sp)
sd x17, 128(sp)
sd x18, 136(sp)
sd x19, 144(sp)
sd x20, 152(sp)
sd x21, 160(sp)
sd x22, 168(sp)
sd x23, 176(sp)
sd x24, 184(sp)
sd x25, 192(sp)
sd x26, 200(sp)
sd x27, 208(sp)
sd x28, 216(sp)
sd x29, 224(sp)
sd x30, 232(sp)
sd x31, 240(sp)
csrr t0, sepc
sd t0, 248(sp)
```

将 scause 和 sepc 中的值传入 trap 处理函数 trap_handler

```
# 2. call trap_handler
csrr a0, scause
csrr a1, sepc
jal x1, trap_handler
```

从内存中恢复 CPU 的寄存器

```
# 3. restore sepc and 32 registers (x2(sp) should be restore
ld t0, 248(sp)
csrw sepc, t0
ld x1, 0(sp)
ld x3, 16(sp)
ld x4, 24(sp)
ld x5, 32(sp)
ld x6, 40(sp)
ld x7, 48(sp)
ld x8, 56(sp)
ld x9, 64(sp)
ld x10, 72(sp)
ld x11, 80(sp)
ld x12, 88(sp)
ld x13, 96(sp)
ld x14, 104(sp)
ld x15, 112(sp)
ld x16, 120(sp)
ld x17, 128(sp)
ld x18, 136(sp)
ld x19, 144(sp)
ld x20, 152(sp)
ld x21, 160(sp)
ld x22, 168(sp)
ld x23, 176(sp)
ld x24, 184(sp)
ld x25, 192(sp)
ld x26, 200(sp)
ld x27, 208(sp)
ld x28, 216(sp)
ld x29, 224(sp)
ld x30, 232(sp)
ld x31, 240(sp)
ld x2, 8(sp)
addi sp, sp, 256
```

从 trap 中返回

```
# 4. return from trap
sret
```

3.4 实现 trap 处理函数

在 arch/riscv/kernel/ 目录下添加 trap.c 文件，文件包含 trap 处理函数 trap_handler(), scause 和 sepc 寄存器的值作为参数。scause 寄存器最高位为 1 时为 interrupt， 剩余位表示 5 时为 timer interrupt。在 arch/riscv/include/ 目录下添加 对应 trap.h 文件

```c
void trap_handler(unsigned long scause, unsigned long sepc) {
    // 通过 `scause` 判断trap类型
    // 如果是interrupt 判断是否是timer interrupt
    // 如果是timer interrupt 则打印输出相关信息, 并通过 `clock_set_next_event()` 设置下一次时钟中断
    // `clock_set_next_event()` 见 4.3.4 节
    // 其他interrupt / exception 可以直接忽略

    // YOUR CODE HERE
    if (scause >> 63){ // 最高位为1就是interrupt
        unsigned long rest = scause - 0x8000000000000000;
        if (rest == 5){ // 剩余位表示5时是timer interrupt
            printk("[S] Supervisor Mode Timer Interrupt\n");
            clock_set_next_event();
        }
    }
}
```

arch > riscv > include > C trap.h > ...

```c
#ifndef _TRAP_H
#define _TRAP_H

void trap_handler(unsigned long scause, unsigned long sepc);
#endif
```

3.5 实现时钟中断相关函数

在 arch/riscv/kernel/ 目录下添加 clock.c 文件。文件包含两个函数：get_cycles（）：使用 rdtime 汇编指令获得当前 time 寄存器中的值；clock_set_next_event（）：调用 sbi_ecall，设置下一个时钟中断事件。

```c
unsigned long get_cycles() {
    // 编写内联汇编，使用 rdtime 获取 time 寄存器中 (也就是mtime 寄存器 )的值并返回
    // YOUR CODE HERE
    unsigned long ret;
    __asm__ volatile (
        "rdtime %[ret]\n"
        : [ret] "=r" (ret)
        :
        : "memory"
    );
    return ret;
}
```

```c
void clock_set_next_event() {
    // 下一次 时钟中断 的时间点
    unsigned long next = get_cycles() + TIMECLOCK;

    // 使用 sbi_ecall 来完成对下一次时钟中断的设置
    // YOUR CODE HERE
    sbi_ecall(0,0,next,0,0,0,0,0);
}
```

3.6 编译测试

虽然新添了.c 文件，但是在 RV64 内核引导部分写的 makefile

文件将全部.c 文件一起处理了，所以不用修改。

```
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ make all
make -C lib all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/lib'
make[1]: Nothing to be done for 'all'.
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/lib'
make -C init all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/init'
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static
 -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1/include -I /home/icey/os23fall-stu/src/lab1/arch/riscv/include -c main.c
riscv64-linux-gnu-gcc -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static
 -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1/include -I /home/icey/os23fall-stu/src/lab1/arch/riscv/include -c test.c
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/init'
make -C arch/riscv all
make[1]: Entering directory '/home/icey/os23fall-stu/src/lab1/arch/riscv'
make -C kernel all
make[2]: Entering directory '/home/icey/os23fall-stu/src/lab1/arch/riscv/kernel'
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -stati
c -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1/include -I /home/icey/os23fall-stu/src/lab1/arch/riscv/include -c entry.S
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -stati
c -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1/include -I /home/icey/os23fall-stu/src/lab1/arch/riscv/include -c head.S
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -stati
c -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1/include -I /home/icey/os23fall-stu/src/lab1/arch/riscv/include -c clock.c
riscv64-linux-gnu-gcc  -march=rv64imafd -mabi=lp64 -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -stati
c -lgcc -Wl,--nmagic -Wl,--gc-sections -g  -I /home/icey/os23fall-stu/src/lab1/include -I /home/icey/os23fall-stu/src/lab1/arch/riscv/include -c trap.c
make[2]: Leaving directory '/home/icey/os23fall-stu/src/lab1/arch/riscv/kernel'
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o -o ../../vmlinux
riscv64-linux-gnu-objcopy -O binary ../../vmlinux ./boot/Image
nm ../../vmlinux >  ../../System.map
make[1]: Leaving directory '/home/icey/os23fall-stu/src/lab1/arch/riscv'

Build Finished OK
```

运行 make run 指令，有如下输出：

```
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
sstatus: 2
sscratch: 0
sscratch after write: 9
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
```

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？

calling convention(函数调用规范)：

1）将参数存储到函数能够访问到的位置；

2）跳转到函数开始位置（使用 jal 指令）

3）获取函数需要的局部存储资源，按需保留寄存器

4）执行函数中的指令

5）将返回值存储到调用者能够访问到的位置，回复寄存器，释放局部存储资源

6）返回调用函数的位置（使用 ret 指令）

Caller / Callee Saved Register 的区别：

caller saved registers（调用者保存寄存器）也叫易失性寄存器，在程序调用的过程中，这些寄存器中的值不需要被保存，如果某一个程序需要保存这个寄存器的值，需要调用者自己压入栈；

callee saved registers（被调用者保存寄存器）也叫非易失性寄存器，在程序调用过程中，

这些寄存器中的值需要被保存，不能被覆盖；当某个程序调用这些寄存器，被调用寄存器会先保存这些值然后再进行调用，且在调用结束后恢复被调用之前的值；

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

```
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ nm vmlinux
0000000080200000 t $x
000000008020000c t $x
00000000802000f0 t $x
0000000080200134 t $x
0000000080200144 t $x
0000000080200194 t $x
0000000080200670 t $x
0000000080200000 A BASE_ADDR
0000000080202000 d _GLOBAL_OFFSET_TABLE_
0000000080204000 B _ebss
0000000080202000 D _edata
0000000080204000 B _ekernel
0000000080201070 R _erodata
00000000802006f0 T _etext
0000000080203000 B _sbss
0000000080202000 D _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080203000 B boot_stack
0000000080204000 B boot_stack_top
0000000080200670 T printk
0000000080200144 T putc
000000008020000c T sbi_ecall
00000000802000f0 T start_kernel
0000000080200134 T test
0000000080200194 t vprintfmt
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$
```

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

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

3 and 4：

要输出寄存器的值，需要一个输出函数，负责输出 int 类型的值：

```c
void printi(int x)
{
    char X[16];
    int i = 0;
    if (x == 0) {
        putc('0');
        return;
    }

    while (x) {
        X[i++] = '0' + x % 10;
        x /= 10;
    }

    while (i) {
        putc(X[--i]);
    }
}
```

在 main.c 中添加代码用来读写寄存器：

```c
    register int val;
    val = csr_read(sstatus);
    printk("sstatus: ");
    printi(val);
    printk("\n");

    val = csr_read(sscratch);
    printk("sscratch: ");
    printi(val);
    printk("\n");
    csr_write(sscratch,9);
    val = csr_read(sscratch);
    printk("sscratch after write: ");
    printi(val);
    printk("\n");
```

输出如下：

```
Boot HART MEDELEG         : 0x000000000000b
2022 Hello RISC-V
sstatus: 2
sscratch: 0
sscratch after write: 9
```

sstatus 寄存器的第 1 位(SIE)为 1 说明允许 S 模式的中断。
sscratch 寄存器前后值的变化说明写入成功。

5. Detail your steps about how to get arch/arm64/kernel/sys.i
进入 linux 文件夹：

```
icey@LAPTOP-RKUCA6PI:/mnt/d/文档/course/OS/lab/lab0$ cd linux-6.6-rc2/
```

配置 defconfig:

```
icey@LAPTOP-RKUCA6PI:/mnt/d/文档/course/OS/lab/lab0/linux-6.6-rc2$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defc
onfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
```

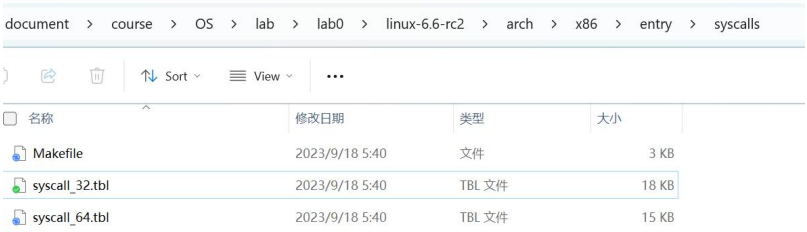
生成 sys.i 文件：

```
#
icey@LAPTOP-RKUCA6PI:/mnt/d/文档/course/OS/lab/lab0/linux-6.6-rc2$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch
/arm64/kernel/sys.i
  SYNC    include/config/auto.conf.cmd
  WRAP    arch/arm64/include/generated/uapi/asm/kvm_para.h
  WRAP    arch/arm64/include/generated/uapi/asm/errno.h
  WRAP    arch/arm64/include/generated/uapi/asm/ioctl.h
  WRAP    arch/arm64/include/generated/uapi/asm/ioctls.h
  WRAP    arch/arm64/include/generated/uapi/asm/ipcbuf.h
```

```
  LD      arch/arm64/kernel/vdso/vdso.so.dbg
  VDSOSYM include/generated/vdso-offsets.h
  OBJCOPY arch/arm64/kernel/vdso/vdso.so
  CPP     arch/arm64/kernel/sys.i
```

6. Find system call table of Linux v6.0 for ARM32, RISC-V(32 bit), RISC-V(64 bit), x86(32 bit), x86_64 List source code file, the whole system call table with macro expanded, screenshot every step.

System call table for x86:



如图所示：

```
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> <compat entry point>
#
# The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
# sys_*() system calls and compat_sys_*() compat system calls if
# IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
# parameter.
#
# The abi is always "i386" for this file.
#
0      i386    restart_syscall           sys_restart_syscall
1      i386    exit                      sys_exit
2      i386    fork                      sys_fork
3      i386    read                      sys_read
4      i386    write                     sys_write
5      i386    open                      sys_open                  compat_sys_open
6      i386    close                     sys_close
7      i386    waitpid                         sys_waitpid
8      i386    creat                     sys_creat
9      i386    link                      sys_link
10     i386    unlink                    sys_unlink
11     i386    execve                    sys_execve
       compat_sys_execve
12     i386    chdir                     sys_chdir
13     i386    time                      sys_time32
14     i386    mknod                     sys_mknod
15     i386    chmod                     sys_chmod
16     i386    lchown                    sys_lchown16
17     i386    break
18     i386    oldstat                         sys_stat
19     i386    lseek                     sys_lseek                 compat_sys_lseek
20     i386    getpid                    sys_getpid
21     i386    mount                     sys_mount
22     i386    umount                    sys_oldumount
23     i386    setuid                    sys_setuid16
```

7. Explain what is ELF file? Try readelf and objdump command on an ELF file, give screenshot of the output. Run an ELF file and cat /proc/PID/maps to give its memory layout.

ELF(executable and linkable format)是一种可执行文件格式，主要用于 linux 平台，windows 平台下相对应的文件为 coff 格式。

```
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ readelf -a vmlinux
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x80200000
  Start of program headers:          64 (bytes into file)
  Start of section headers:          22280 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         4
  Size of section headers:           64 (bytes)
  Number of section headers:         20
  Section header string table index: 19
```

```
Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS        0000000080200000  00001000
       0000000000000934  0000000000000000  AX       0     0     4096
  [ 2] .rodata           PROGBITS        0000000080201000  00002000
       0000000000000098  0000000000000000   A       0     0     4096
  [ 3] .data             PROGBITS        0000000080202000  00003000
       0000000000000008  0000000000000000  WA       0     0     4096
  [ 4] .got              PROGBITS        0000000080202008  00003008
       0000000000000018  0000000000000008  WA       0     0     8
  [ 5] .got.plt          PROGBITS        0000000080202020  00003020
       0000000000000010  0000000000000008  WA       0     0     8
  [ 6] .bss              NOBITS          0000000080203000  00003030
       0000000000001000  0000000000000000  WA       0     0     4096
  [ 7] .debug_info       PROGBITS        0000000000000000  00003030
```

```
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), p (processor specific)

There are no section groups in this file.

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  RISCV_ATTRIBUT 0x0000000000004cc5 0x0000000000000000 0x0000000000000000
                 0x0000000000000032 0x0000000000000000  R      0x1
  LOAD           0x0000000000001000 0x0000000080200000 0x0000000080200000
                 0x0000000000001098 0x0000000000001098  R E    0x1000
  LOAD           0x0000000000003000 0x0000000080202000 0x0000000080202000
                 0x0000000000000030 0x0000000000002000  RW     0x1000
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10

 Section to Segment mapping:
  Segment Sections...
   00     .riscv.attributes
   01     .text .rodata
   02     .data .got .got.plt .bss
   03
```

```
Symbol table '.symtab' contains 63 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000080200000     0 SECTION LOCAL  DEFAULT    1 .text
     2: 0000000080201000     0 SECTION LOCAL  DEFAULT    2 .rodata
     3: 0000000080202000     0 SECTION LOCAL  DEFAULT    3 .data
     4: 0000000080202008     0 SECTION LOCAL  DEFAULT    4 .got
     5: 0000000080202020     0 SECTION LOCAL  DEFAULT    5 .got.plt
     6: 0000000080203000     0 SECTION LOCAL  DEFAULT    6 .bss
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    7 .debug_info
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    8 .debug_abbrev
     9: 0000000000000000     0 SECTION LOCAL  DEFAULT    9 .debug_aranges
    10: 0000000000000000     0 SECTION LOCAL  DEFAULT   10 .debug_rnglists
    11: 0000000000000000     0 SECTION LOCAL  DEFAULT   11 .debug_line
    12: 0000000000000000     0 SECTION LOCAL  DEFAULT   12 .debug_str
    13: 0000000000000000     0 SECTION LOCAL  DEFAULT   13 .debug_line_str
    14: 0000000000000000     0 SECTION LOCAL  DEFAULT   14 .comment
    15: 0000000000000000     0 SECTION LOCAL  DEFAULT   15 .riscv.attributes
    16: 0000000000000000     0 SECTION LOCAL  DEFAULT   16 .debug_frame
    17: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS head.o
    18: 0000000080200000     0 NOTYPE  LOCAL  DEFAULT    1 $x
    19: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS entry.o
    20: 0000000080200028     0 NOTYPE  LOCAL  DEFAULT    1 $x
    21: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS clock.c
    22: 0000000080200148     0 NOTYPE  LOCAL  DEFAULT    1 $x
```

```
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ objdump -v vmlinux
GNU objdump (GNU Binutils for Ubuntu) 2.38
Copyright (C) 2022 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or (at your option) any later version.
This program has absolutely no warranty.
```

```
-bash: ./System.map: Permission denied
icey@LAPTOP-RKUCA6PI:~/os23fall-stu/src/lab1$ cat System.map
0000000080200000 t $x
000000008020002c t $x
000000008020014c t $x
0000000080200174 t $x
00000000802001d8 t $x
00000000802002a4 t $x
0000000080200308 t $x
00000000802003f4 t $x
0000000080200404 t $x
0000000080200454 t $x
0000000080200930 t $x
00000000802009b0 t $x
0000000080200000 A BASE_ADDR
0000000080202000 D TIMECLOCK
0000000080202008 d _GLOBAL_OFFSET_TABLE_
0000000080204000 B _ebss
0000000080202008 D _edata
0000000080204000 B _ekernel
00000000802010e8 R _erodata
0000000080200a80 T _etext
0000000080203000 B _sbss
0000000080202000 D _sdata
0000000080200000 T _skernel
0000000080201000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
000000008020002c T _traps
0000000080203000 B boot_stack
0000000080204000 B boot_stack_top
0000000080200174 T clock_set_next_event
000000008020014c T get_cycles
```

8. 在我们使用 make run 时， OpenSBI 会产生如下输出 ：

```
OpenSBI v0.9
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

......

Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109

......
```

通过查看 RISC-V Privileged Spec 中的 medeleg 和 mideleg ，解释上面 MIDELEG 值的含义。

MIDELEG 值为 0x222，即第 1、5、9 位为 1，也就是说这些位上的中断由 S 模式处理。同样，MEDELEG 寄存器的对应位的异常由 S 模式处理。

五、讨论心得

1、编写 makefile 文件时，可以将文件夹下的所有.c 文件用 all 来代替，这样添加新的.c 文件时不用修改原 makefile 文件。

2、本实验最重要的部分还是对扩展知识的理解上，无论是 makefile 的编写还是汇编语言的使用都是以前没有深入了解过的内容。虽然本次实验只使用到较小部分的扩展知识，按照实验指导进行实验就很顺利，但为了后续实验也能顺利进行，必须更全面地学习这些知识。

3、内核运行后出现的界面包含了许多信息，做 lab0 时没有仔细看，做 lab1 的思考题时才注意到这点。包括 OpenSBI 的版本、平台的名称、一些寄存器的值等等。

```
OpenSBI v0.9
   ____                   _____ ____ _____
  / __ \                 / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : timer,mfdeleg
Platform HART Count       : 1
Firmware Base             : 0x80000000
Firmware Size             : 100 KB
Runtime SBI Version       : 0.2

Domain0 Name              : root
Domain0 Boot HART         : 0
Domain0 HARTs             : 0*
Domain0 Region00          : 0x0000000080000000-0x000000008001ffff ()
Domain0 Region01          : 0x0000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087000000
Domain0 Next Mode         : S-mode
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
```

4、实现时钟中断的实验中，在往 head.S 文件增添代码时，要添在加载栈地址的代码之后。

```
_start:

    # 内核引导部分的代码

    la sp, boot_stack_top

    # 时钟中断部分的代码
    # ------------------

        # set stvec = _traps
        la a0, _traps
        csrw stvec, a0

    # ------------------

        # set sie[STIE] = 1
        li a0, 1<<5
        csrw sie, a0
```