

浙江大学

本科实验报告

课程名称：操作系统

姓 名：黄文杰

学 院：计算机科学与技术学院

系：计算机科学与技术系

专 业：软件工程

学 号：3210103379

指导教师：夏莹杰

2023 年 10 月 12 日

浙江大学操作系统实验报告

实验名称： RV64 内核引导与时钟中断处理

电子邮件地址： 3210103379@zju.edu.cn 手机： 15167970568

实验地点： 曹光彪西 503 实验日期： 2023 年 10 月 12 日

一、实验目的和要求

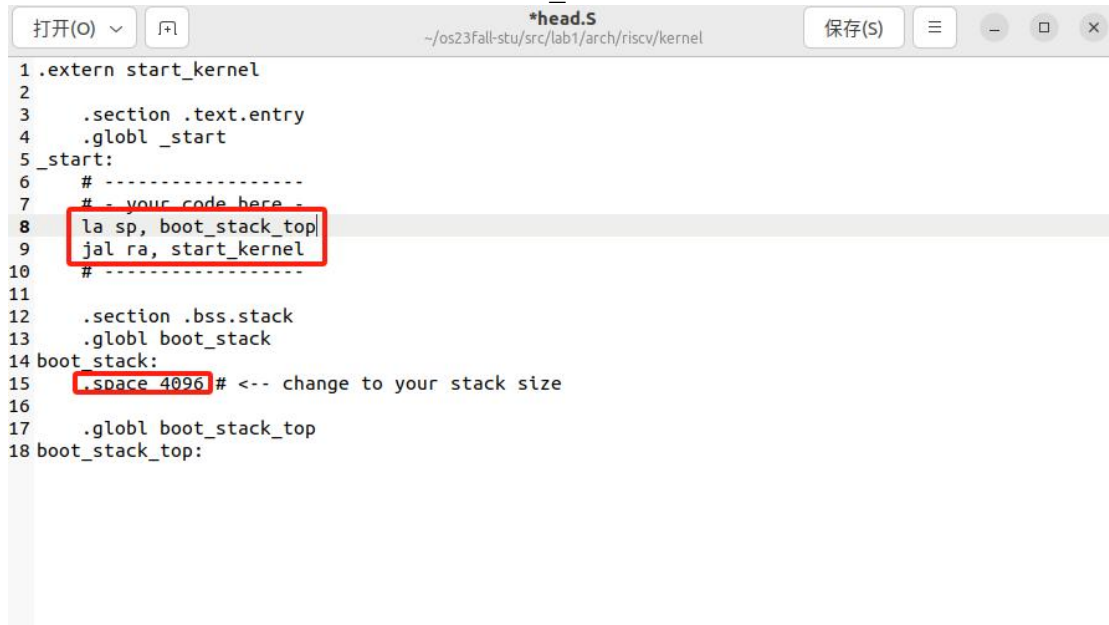
1. 学习 RISC-V 汇编，编写 head.S 实现跳转到内核运行的第一个 C 函数。
2. 学习 OpenSBI，理解 OpenSBI 在实验中所起到的作用，并调用 OpenSBI 提供的接口完成字符的输出。
3. 学习 Makefile 相关知识，补充项目中的 Makefile 文件，来完成对整个工程的管理。
4. 学习 RISC-V 的 trap 处理相关寄存器与指令，完成对 trap 处理的初始化。
5. 理解 CPU 上下文切换机制，并正确实现上下文切换功能。
6. 编写 trap 处理函数，完成对特定 trap 的处理。
7. 调用 OpenSBI 提供的接口，完成对时钟中断事件的设置。

二、实验过程

1. RV64 内核引导

1.1 编写 head.S

编写 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



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

1.2 完善 Makefile 脚本

补充 lib/Makefile，使工程得以编译：



```
1 C_SRC      = $(sort $(wildcard *.c))
2 OBJ        = $(patsubst %.c,%.o,$(C_SRC))
3
4 all:$(OBJ)
5
6 %.o:%.c
7     ${GCC} ${CFLAG} -c $<
8 clean:
9     $(shell rm *.o 2>/dev/null)
```

在工程根文件夹执行 `make`，可以看到工程成功编译出 `vmlinux`:

```
hwj@hwj-virtual-machine: ~/os23fall-stu/src/lab1$ make
make -C lib all
make[1]: 进入目录"/home/hwj/os23fall-stu/src/lab1/lib"
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/hwj/os23fall-stu/src/lab1/include -I /home/hwj/os23fall-stu/src/lab1/arch/riscv/include -c printk.c
make[1]: 离开目录"/home/hwj/os23fall-stu/src/lab1/lib"
make -C init all
make[1]: 进入目录"/home/hwj/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/hwj/os23fall-stu/src/lab1/include -I /home/hwj/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/hwj/os23fall-stu/src/lab1/include -I /home/hwj/os23fall-stu/src/lab1/arch/riscv/include -c test.c
make[1]: 离开目录"/home/hwj/os23fall-stu/src/lab1/init"
make -C arch/riscv all
make[1]: 进入目录"/home/hwj/os23fall-stu/src/lab1/arch/riscv"
make -C kernel all
make[2]: 进入目录"/home/hwj/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 -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g -I /home/hwj/os23fall-stu/src/lab1/include -I /home/hwj/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 -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g -I /home/hwj/os23fall-stu/src/lab1/include -I /home/hwj/os23fall-stu/src/lab1/arch/riscv/include -c sbi.c
make[2]: 离开目录"/home/hwj/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]: 离开目录"/home/hwj/os23fall-stu/src/lab1/arch/riscv"

Build Finished OK
```

1.3 补充 `sbi.c`

(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，用两个变量来接受这两个返回值。

```
sbi.c
~/os23fall-stu/src/lab1/arch/riscv/kernel
保存(S)

1 #include "types.h"
2 #include "sbi.h"
3
4
5 struct sbiret sbi_ecall(int ext, int fid, uint64 arg0,
6                        uint64 arg1, uint64 arg2,
7                        uint64 arg3, uint64 arg4,
8                        uint64 arg5)
9 {
10     struct sbiret result;
11     long error, value;
12     __asm__ volatile (
13         "mv a7, %[ext]\n"
14         "mv a6, %[fid]\n"
15         "mv a0, %[arg0]\n"
16         "mv a1, %[arg1]\n"
17         "mv a2, %[arg2]\n"
18         "mv a3, %[arg3]\n"
19         "mv a4, %[arg4]\n"
20         "mv a5, %[arg5]\n"
21         "ecall\n"
22         "mv %[error], a0\n"
23         "mv %[value], a1\n"
24         : [error] "=r" (error), [value] "=r" (value)
25         : [ext] "r" (ext), [fid] "r" (fid), [arg0] "r" (arg0), [arg1] "r" (arg1),
26         [arg2] "r" (arg2), [arg3] "r" (arg3), [arg4] "r" (arg4), [arg5] "r" (arg5)
27         : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7"
28     );
29     result.error=error;
30     result.value=value;
31     return result;
32 }
33
```

1.4 修改 defs

修改后的 defs.h 文件如下：

```
defs.h
~/os23fall-stu/src/lab1/arch/riscv/include
保存(S)

sbi.c
defs.h

1 #ifndef _DEFS_H
2 #define _DEFS_H
3
4 #include "types.h"
5
6 #define csr_read(csr)
7 ({
8     register uint64 __v;
9     asm volatile ("csr%[v], " #csr
10                  : [v]"=r" (__v)
11                  :
12                  : "memory");
13     __v;
14 })
15
16 #define csr_write(csr, val)
17 ({
18     uint64 __v = (uint64)(val);
19     asm volatile ("csr%[v] " #csr " " #0"
20                  : : "r" (__v)
21                  : "memory");
22 })
23
24 #endif
```

此时用 `qemu` 运行 `make` 好的内核，结果如下（可以看到在最后成功输出了 `2023 Hello RISC-V`）

```
hwj@hwj-virtual-machine:~/os23fall-stu/src/lab1$ qemu-system-riscv64 -nographic -machine virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0" \
-bios default -drive file=/home/hwj/os23fall-stu/src/lab0/rootfs.img,format=raw,id=hd0

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   : 0x0000000000000000-0xffffffffffffff (R,W,X)
Domain0 Next Address : 0x0000000080200000
Domain0 Next Arg1   : 0x0000000087000000
Domain0 Next Mode   : S-mode
```

```
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  : 0x00000000080000000-0x0000000008001ffff ()
Domain0 Region01  : 0x00000000000000000-0xffffffffffffffff (R,W,X)
Domain0 Next Address : 0x00000000080200000
Domain0 Next Arg1  : 0x00000000087000000
Domain0 Next Mode   : S-mode
Domain0 SysReset    : yes

Boot HART ID      : 0
Boot HART Domain   : root
Boot HART ISA      : rv64imafdcisu
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
2023 Hello RISC-V
```

2. RV64 时钟中断处理

2.1 修改 vmlinux.lds 以及 head.S

修改 vmlinux.lds, 在.text 段加入.text.init

```
打开(O)  vmlinux.lds  保存(S)
~/os23fall-stu/src/lab1/arch/riscv/kernel

/* 程序入口 */
5 ENTRY( _start )
6
7 /* kernel代码起始位置 */
8 BASE_ADDR = 0x80200000;
9
10 SECTIONS
11 {
12     /* . 代表当前地址 */
13     . = BASE_ADDR;
14
15     /* 记录kernel代码的起始地址 */
16     _skernel = .;
17
18     /* ALIGN(0x1000) 表示4KB对齐 */
19     /* _stext, _etext 分别记录了text段的起始与结束地址 */
20     .text : ALIGN(0x1000){
21         _stext = .;
22
23         *.text.init
24         *(.text.entry)
25         *(.text .text.*)
26
27         _etext = .;
28     }
29
```

修改 head.S, 将_start 放在.text.init 段

```
打开(O)  head.S  保存(S)
~/os23fall-stu/src/lab1/arch/riscv/kernel

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

2.2 开启 trap 处理

修改 arch/riscv/kernel/head.S, 并补全 _start 中的逻辑

(1) 设置 stvec, 将 _traps (_trap 在 4.3 中实现) 所表示的地址写入 stvec,

```
# set stvec = _traps
la t0, _traps
csrw stvec, t0
# -----
```


(2) 开启时钟中断，将 sie[STIE] 置 1。

```
# set sie[STIE] = 1
li t1,0x20
csrrs t1,sie,t1
# -----
```

(3) 设置第一次时钟中断（这里通过直接跳转到 clock_set_next_event 函数入口地址来实现）。

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

(4) 开启 S 态下的中断响应，将 sstatus[SIE] 置 1。

```
# set sstatus[SIE] = 1
li t2,0x2
csrrs t2,ssstatus,t2
# -----
```

完整的代码如下：

```
_start:
    la sp, boot_stack_top
    # -----
    # - your code here -
    # -----

    # set stvec = _traps
    la t0, _traps
    csrw stvec, t0
    # -----

    # set sie[STIE] = 1
    csrr t0, sie
    ori t0, t0, 1<<5
    csrw sie, t0
    # -----

    # set first time interrupt
    jal ra, clock_set_next_event
    #rdtime a0
    #li a1, 100000000
    #add a1,a0,a1
    #mv a0,mtimecmp
    li a7, 0x0000000000000000
    li a6, 0x0000000000000000
    rdtm a5
    li t0, 1000000000
    add a5, a5, t0
    ecall

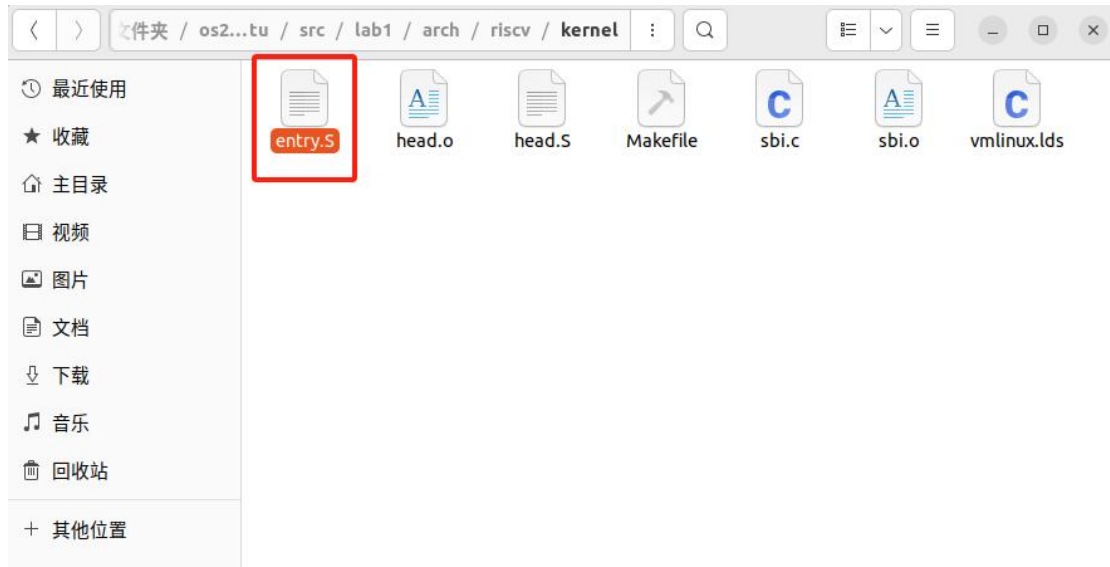
    # -----

    # set sstatus[SIE] = 1
    li t0,0x2
    csrrs t0,ssstatus ,t0
    # -----

    #la sp, boot_stack_top
    jal ra, start_kernel
    # -----
```

2.3 实现上下文切换

(1) 在 `arch/riscv/kernel/` 目录下添加 `entry.S` 文件。



(2) 保存 CPU 的寄存器（上下文，即 32 个通用寄存器以及 `sepc` 寄存器）到内存中（栈上）。

```

# 1. save 32 registers and sepc to stack
addi sp, sp, -256
sd x0, 0(sp)
sd x1, 8(sp)
sd x2, 16(sp)
sd x3, 24(sp)
sd x4, 32(sp)
sd x5, 40(sp)
sd x6, 48(sp)
sd x7, 56(sp)
sd x8, 64(sp)
sd x9, 72(sp)
sd x10, 80(sp)
sd x11, 88(sp)
sd x12, 96(sp)
sd x13, 104(sp)
sd x14, 112(sp)
sd x15, 120(sp)
sd x16, 128(sp)
sd x17, 136(sp)
sd x18, 144(sp)
sd x19, 152(sp)
sd x20, 160(sp)
sd x21, 168(sp)
sd x22, 176(sp)
sd x23, 184(sp)
sd x24, 192(sp)
sd x25, 200(sp)
sd x26, 208(sp)
sd x27, 216(sp)
sd x28, 224(sp)
sd x29, 232(sp)
sd x30, 240(sp)
sd x31, 248(sp)
addi sp, sp, -256
csrr t0, sepc
sd t0, 0(sp)

```

(3) 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 ra, trap_handler

```

(4) 在完成对 `trap` 的处理之后，我们从内存中(栈上)恢复 CPU 的寄存器(上下文)。

```

# 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
ld t0, 0(sp)
csrw sepc, t0
addi sp, sp, 8
ld x0, 0(sp)
ld x1, 8(sp)
ld x2, 16(sp)
ld x3, 24(sp)
ld x4, 32(sp)
ld x5, 40(sp)
ld x6, 48(sp)
ld x7, 56(sp)
ld x8, 64(sp)
ld x9, 72(sp)
ld x10, 80(sp)
ld x11, 88(sp)
ld x12, 96(sp)
ld x13, 104(sp)
ld x14, 112(sp)
ld x15, 120(sp)
ld x16, 128(sp)
ld x17, 136(sp)
ld x18, 144(sp)
ld x19, 152(sp)
ld x20, 160(sp)
ld x21, 168(sp)
ld x22, 176(sp)
ld x23, 184(sp)
ld x24, 192(sp)
ld x25, 200(sp)
ld x26, 208(sp)
ld x27, 216(sp)
ld x28, 224(sp)
ld x29, 232(sp)
ld x30, 240(sp)
ld x31, 248(sp)
addi sp, sp, 256

```

(5) 从 trap 中返回。

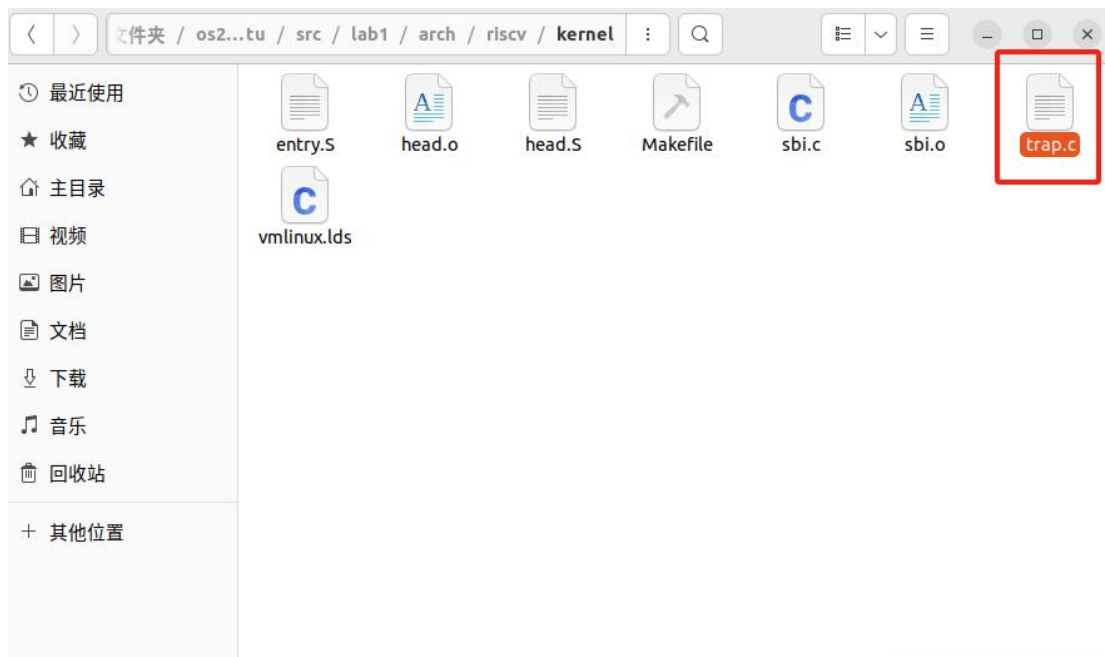
```

# 4. return from trap
sret

```

2.4 实现 trap 处理函数

(1) 在 arch/riscv/kernel/ 目录下添加 trap.c 文件



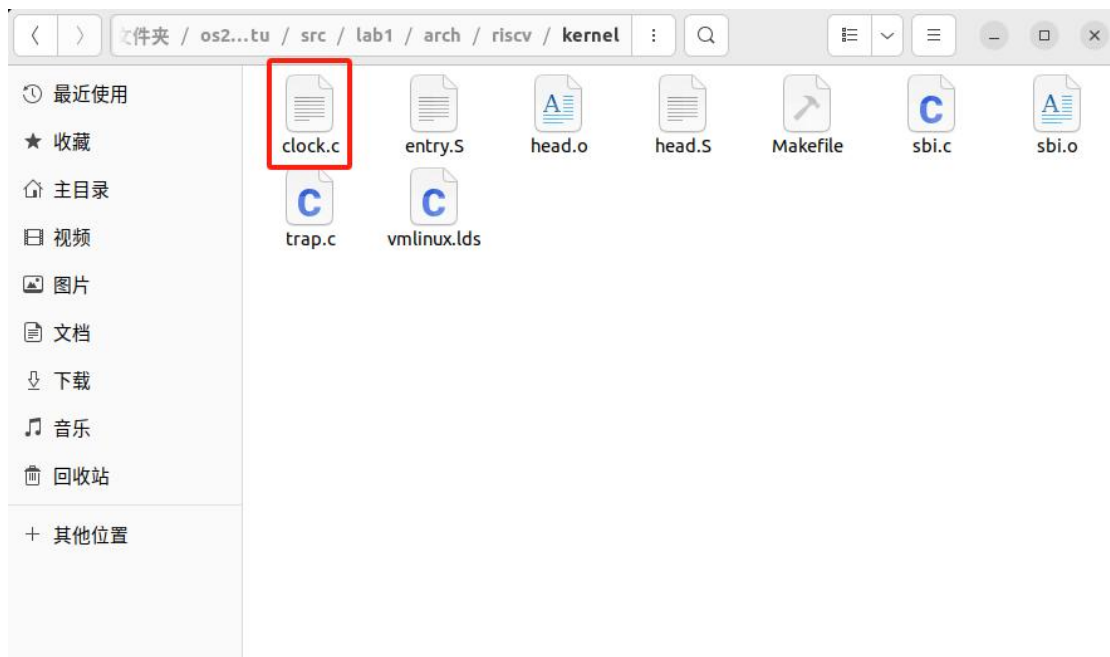
(2) 在 trap.c 中实现 trap 处理函数 trap_handler(), 其接收的两个参数分别是 scause 和 sepc 两个寄存器中的值。

A screenshot of a code editor window. The title bar shows the file name 'trap.c' and the path '~/.os23fall-stu/src/lab1/arch/riscv/kernel'. The editor has tabs for 'sbi.c', 'defs.h', 'vmlinux.lds', 'head.S', 'entry.S', and 'trap.c'. The code in the editor is as follows:

```
1 // trap.c
2 #include "printk.h"
3 #include "clock.h"
4
5 unsigned long flag1 = 0x8000000000000000;
6 unsigned long flag2 = 0x5;
7
8 void trap_handler(unsigned long scause, unsigned long sepc) {
9     // 通过 `scause` 判断trap类型
10    // 如果是interrupt 判断是否是timer interrupt
11    // 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
12    // `clock_set_next_event()` 见 4.3.4 节
13    // 其他interrupt / exception 可以直接忽略
14
15    // YOUR CODE HERE
16    if(scause&flag1){
17        scause=scause-flag1;
18        if(scause&flag2){
19            printk(" Supervisor Mode Time Interrupt!\n");
20            clock_set_next_event();
21        }else{
22            return;
23        }
24    }
25
26 }
```

2.5 实现时钟中断相关函数

(1) 在 arch/riscv/kernel/ 目录下添加 clock.c 文件。



(2) 在 clock.c 中实现 get_cycles() : 使用 rdttime 汇编指令获得当前 time 寄存器中的值。

```
unsigned long get_cycles() {
    // 编写内联汇编, 使用 rdttime 获取 time 寄存器中 (也就是mtime 寄存器) 的值并返回
    // YOUR CODE HERE

    unsigned long time;
    __asm__ volatile (
        "rdtime %0\n"
        : "=r" (time)
        :
        :
    );
    return time
}
```

(3) 在 clock.c 中实现 clock_set_next_event() : 调用 sbi_ecall, 设置下一个时钟中断事件。

```
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(0x0, 0x0, next, 0, 0, 0, 0);
    return;
}
```

完整的代码如下：

```
clock.c
~/os23fall-stu/src/lab1/arch/riscv/kernel

1 // clock.c
2 #include "clock.h"
3 #include "sbi.h"
4
5 // QEMU中时钟的频率是10MHz，也就是1秒钟相当于100000000个时钟周期。
6 unsigned long TIMECLOCK = 100000000;
7
8 unsigned long get_cycles() {
9     // 编写内联汇编，使用 rdttime 获取 time 寄存器中（也就是mtime 寄存器）的值并返回
10    // YOUR CODE HERE
11
12    unsigned long time;
13    __asm__ volatile (
14        "rdtime %0\n"
15        : "=r" (time)
16        :
17        :
18    );
19    return time;
20 }
21
22 void clock_set_next_event() {
23     // 下一次 时钟中断 的时间点
24     unsigned long next = get_cycles() + TIMECLOCK;
25
26     // 使用 sbi_ecall 来完成对下一次时钟中断的设置
27     // YOUR CODE HERE
28     sbi_ecall(0x0, 0x0, next, 0, 0, 0, 0);
29     return;
30 }
31
32 }
33
```

2.6 编译及测试

```
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x0000000000000b109
scause: 00000005
Supervisor Mode Time Interrupt!
2023 Hello RISC-V
15: 0000000f
sstatus: 00006022
sie: 00000020
prev sscratch: 00000000
later sscratch: 00000001
scause: 00000005
Supervisor Mode Time Interrupt!
scause: 00000005
Supervisor Mode Time Interrupt!
scause: 00000005
Supervisor Mode Time Interrupt!
scause: 00000005
Supervisor Mode Time Interrupt!
scause: 00000005
Supervisor Mode Time Interrupt!
```


可以看到时钟中断已经开启，每隔一秒输出一行提示语句。

3. 其他架构的交叉编译——以 Aarch64 为例

3.1 交叉编译工具链的安装

搜索包含 aarch64 的软件包：

```
hwj@hwj-virtual-machine:~$ apt-cache search aarch64
cpp-11-aarch64-linux-gnu - GNU C preprocessor
cpp-aarch64-linux-gnu - GNU C preprocessor (cpp) for the arm64 architecture
g++-11-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
g++-aarch64-linux-gnu - GNU C++ compiler for the arm64 architecture
gcc-11-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-11-aarch64-linux-gnu-base - GCC, GNU 编译器套装（基本软件包）
gcc-aarch64-linux-gnu - GNU C compiler for the arm64 architecture
binutils-aarch64-linux-gnu - GNU binary utilities, for aarch64-linux-gnu target
binutils-aarch64-linux-gnu-dbg - GNU binary utilities, for aarch64-linux-gnu target (debug symbols)
qemu-efi-aarch64 - UEFI firmware for 64-bit ARM virtual machines
qemu-system-arm - QEMU full system emulation binaries (arm)
cpp-10-aarch64-linux-gnu - GNU C 预处理器
cpp-12-aarch64-linux-gnu - GNU C 预处理器
cpp-9-aarch64-linux-gnu - GNU C 预处理器
g++-10-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
g++-12-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
g++-9-aarch64-linux-gnu - GNU C++ compiler (cross compiler for arm64 architecture)
gcc-10-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-10-aarch64-linux-gnu-base - GCC, GNU 编译器套装（基本软件包）
gcc-12-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-12-aarch64-linux-gnu-base - GCC, GNU 编译器套装（基本软件包）
gcc-9-aarch64-linux-gnu - GNU C compiler (cross compiler for arm64 architecture)
gcc-9-aarch64-linux-gnu-base - GCC, GNU 编译器套装（基本软件包）
gccgo-10-aarch64-linux-gnu - GNU Go compiler
gccgo-11-aarch64-linux-gnu - GNU Go compiler
gccgo-12-aarch64-linux-gnu - GNU Go compiler
gccgo-9-aarch64-linux-gnu - GNU Go compiler
gccgo-aarch64-linux-gnu - Go compiler (based on GCC) for the arm64 architecture
libvixl5 - ARMv8 Runtime Code Generation Library
```

下载需要的工具：

```

hwj@hwj-virtual-machine:~$ sudo apt install gcc-aarch64-linux-gnu
[sudo] hwj 的密码:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
将会同时安装下列软件:
  binutils-aarch64-linux-gnu cpp-11-aarch64-linux-gnu cpp-aarch64-linux-gnu
  gcc-11-aarch64-linux-gnu gcc-11-aarch64-linux-gnu-base gcc-11-cross-base
  gcc-12-cross-base libasan6-arm64-cross libatomic1-arm64-cross
  libc6-arm64-cross libc6-dev-arm64-cross libgcc-11-dev-arm64-cross
  libgcc-s1-arm64-cross libgomp1-arm64-cross libhwasan0-arm64-cross
  libitm1-arm64-cross liblsan0-arm64-cross libstdc++6-arm64-cross
  libtsan0-arm64-cross libubsan1-arm64-cross linux-libc-dev-arm64-cross
建议安装:
  binutils-doc gcc-11-locales cpp-doc gcc-11-doc gdb-aarch64-linux-gnu gcc-doc
下列【新】软件包将被安装:
  binutils-aarch64-linux-gnu cpp-11-aarch64-linux-gnu cpp-aarch64-linux-gnu
  gcc-11-aarch64-linux-gnu gcc-11-aarch64-linux-gnu-base gcc-11-cross-base
  gcc-12-cross-base gcc-aarch64-linux-gnu libasan6-arm64-cross
  libatomic1-arm64-cross libc6-arm64-cross libc6-dev-arm64-cross
  libgcc-11-dev-arm64-cross libgcc-s1-arm64-cross libgomp1-arm64-cross
  libhwasan0-arm64-cross libitm1-arm64-cross liblsan0-arm64-cross
  libstdc++6-arm64-cross libtsan0-arm64-cross libubsan1-arm64-cross

```

3.2 获得编译过程的中间产物

(1) 先 config

```

hwj@hwj-virtual-machine:~/Linux/linux-6.5.6$ make ARCH=arm64 CROSS_COMPILE=aarch
64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#

```

(2) 然后指定要生成的文件

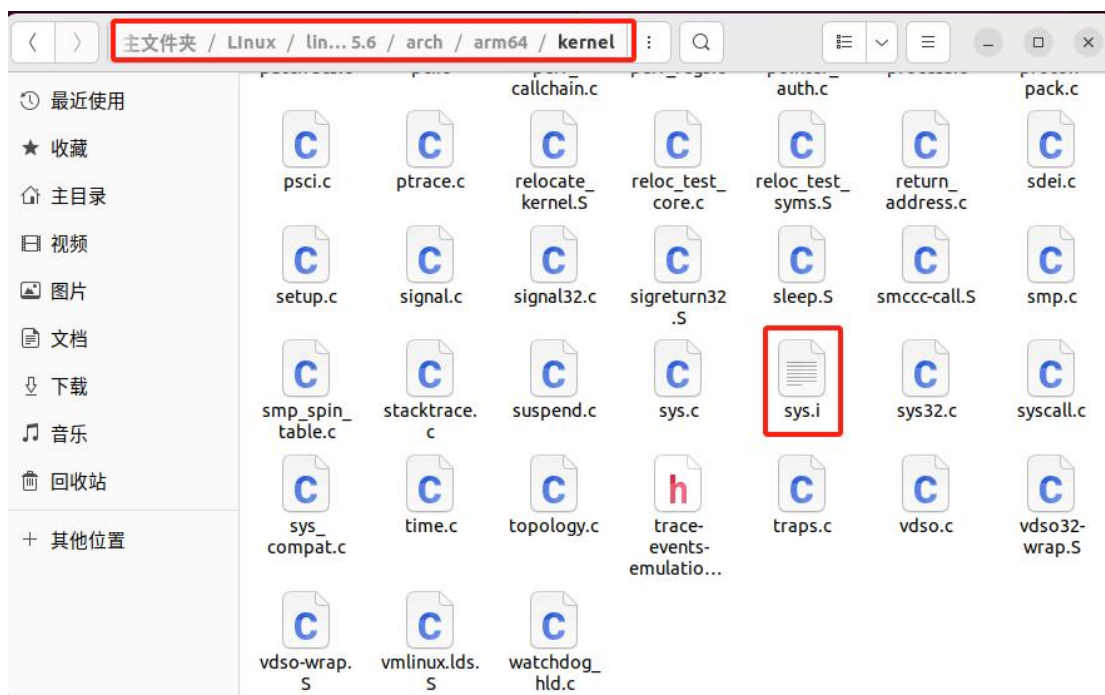
```

hwj@hwj-virtual-machine:~/Linux/linux-6.5.0$ 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
WRAP      arch/arm64/include/generated/uapi/asm/msgbuf.h
WRAP      arch/arm64/include/generated/uapi/asm/poll.h
WRAP      arch/arm64/include/generated/uapi/asm/resource.h
WRAP      arch/arm64/include/generated/uapi/asm/sembuf.h
WRAP      arch/arm64/include/generated/uapi/asm/shmbuf.h
WRAP      arch/arm64/include/generated/uapi/asm/siginfo.h
WRAP      arch/arm64/include/generated/uapi/asm/socket.h
WRAP      arch/arm64/include/generated/uapi/asm/sockios.h
WRAP      arch/arm64/include/generated/uapi/asm/stat.h
WRAP      arch/arm64/include/generated/uapi/asm/swab.h
WRAP      arch/arm64/include/generated/uapi/asm/termbits.h
WRAP      arch/arm64/include/generated/uapi/asm/termios.h
WRAP      arch/arm64/include/generated/uapi/asm/types.h
WRAP      arch/arm64/include/generated/asm/early_ioremap.h
WRAP      arch/arm64/include/generated/asm/mcs_spinlock.h
WRAP      arch/arm64/include/generated/asm/qrwlock.h
WRAP      arch/arm64/include/generated/asm/qspinlock.h
WRAP      arch/arm64/include/generated/asm/parport.h
WRAP      arch/arm64/include/generated/asm/user.h
WRAP      arch/arm64/include/generated/asm/bugs.h
WRAP      arch/arm64/include/generated/asm/delay.h
WRAP      arch/arm64/include/generated/asm/softirq_stack.h
WRAP      arch/arm64/include/generated/asm/switch_to.h
WRAP      arch/arm64/include/generated/asm/trace_clock.h
WRAP      arch/arm64/include/generated/asm/unaligned.h
WRAP      arch/arm64/include/generated/asm/vga.h
UPD      include/generated/compile.h
GEN      arch/arm64/include/generated/asm/cpucaps.h
GEN      arch/arm64/include/generated/asm/sysreg-defs.h
CC       scripts/mod/empty.o
MKELF    scripts/mod/elfconfig.h
HOSTCC   scripts/mod/modpost.o
CC       scripts/mod/devicetable-offsets.s
HOSTCC   scripts/mod/file2alias.o
HOSTCC   scripts/mod/sumversion.o
HOSTLD   scripts/mod/modpost
CC       kernel/bounds.s
UPD      include/generated/bounds.h
CC       arch/arm64/kernel/asm-offsets.s
UPD      include/generated/asm-offsets.h
CALL     scripts/checksyscalls.sh
LDS      arch/arm64/kernel/vdso/vdso.lds
CC       arch/arm64/kernel/vdso/vgettimeofday.o
AS       arch/arm64/kernel/vdso/note.o
AS       arch/arm64/kernel/vdso/sigreturn.o
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

```

(3) 查看结果，发现在指定目录下找到了 sys.i 文件



三、讨论和心得

总体来说，本次实验相较于前一次实验难度上有所提升，对我们的要求也有所提高。

通过这次实验，我学习了一些 RISC-V 汇编、内联汇编、Makefile 和时钟中断的相关知识，掌握了一点 RISC-V 汇编和内联汇编的语法，知道了如何编写 makefile 文件来链接一个项目。

当然，在做实验的过程中我碰到了不少困难，其中耗费我时间最久的莫过于在做时钟中断那部分的时候，由于在编写 Head.S 文件时，我忘记将第一部分所编写的 `la sp, boot_stack_top` 这一条汇编语句移到 `_start` 段的最上方，导致我后续的时钟中断无法成功执行（程序栈空间不够）。在用 gdb 调试的过程中，我发现我的函数总是在进入 `trap_handler` 之后，在 `_traps` 段保存上下文的时候，频繁地被系统的其他 Interrupt（M 态的 Interrupt）所打断，以致于系统会重复地执行前几句汇编代码，而不会跳出 `_traps` 段。我尝试过打印 `scause` 寄存器的值来判断这个 Interrupt 的类型，但根据 `scause` 的值只能判断出这是来自 M 态的 Interrupt，因而我调试了很久的 bug 都没调试出来，直到碰到同样问题的室友偶然间解决了

这个问题，才将我从泥淖中拉出来。

所以我诚恳地建议，可以把我碰到的问题反馈到下一届的实验指导中，提醒同学们在修改 Head.s 文件的时候一定要先将 sp 指针定义到自定义程序段的栈顶再进行后续时钟中断的处理，否则当程序跳入中断处理程序时，保存上下文这一步骤会占用大量的程序栈，从而可能会出现栈指针越界而导致 M 态 Interrupt 的问题。

```
8 _start:
9  la sp, boot_stack_top
10 # -----
11 # - your code here -
12 # -----
13
14 # set stvec = _traps
15 la t0, _traps
16 csrw stvec, t0
17 # -----
18
19 # set sie[STIE] = 1
20 csrr t0, sie
21 ori t0, t0, 1<<5
22 csrw sie, t0
23 # -----
24
25 # set first time interrupt
26 #jal ra, clock_set_next_event
27 #rdtime a0
28 #li a1, 10000000
29 #add a1,a0,a1
30 #mv a0,mtimecmp
31 li a7, 0x0000000000000000
32 li a6, 0x0000000000000000
33 rdttime a5
34 li t0, 100000000
35 add a5, a5, t0
36 ecall
37
38 # -----
39
40 # set sstatus[SIE] = 1
41 li t0,0x2
42 csrrs t0,sstatus ,t0
43 # -----
44
45 #la sp, boot_stack_top
46 jal ra, start_kernel
47 # -----
```

```
entry.S
~/os23fall-stu/src/lab1/arch/riscv/kernel
保存(S)

1 .extern trap_handler
2
3 .section .text.entry
4 .align 2
5 .globl _traps
6 traps:
7 # YOUR CODE HERE
8 # -----
9
10 # 1. save 32 registers and sepc to stack
11 addi sp, sp, -264
12 sd x0, 0(sp)
13 sd x1, 8(sp)
14 sd x2, 16(sp)
15 sd x3, 24(sp)
16 sd x4, 32(sp)
17 sd x5, 40(sp)
18 sd x6, 48(sp)
19 sd x7, 56(sp)
20 sd x8, 64(sp)
21 sd x9, 72(sp)
22 sd x10, 80(sp)
23 sd x11, 88(sp)
24 sd x12, 96(sp)
25 sd x13, 104(sp)
26 sd x14, 112(sp)
27 sd x15, 120(sp)
28 sd x16, 128(sp)
```

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？

RISC-V 的 calling convention 通常包括以下几个方面：

1. 参数传递：函数参数通常通过一些特定寄存器（如 a0-a7）传递，而不是通过堆栈。剩余的参数可能会被传递到堆栈上。
2. 返回值：函数的返回值通常存储在特定寄存器（如 a0）中，而不是堆栈上。
3. Callee Saved Registers：这些寄存器（s0-s11）在函数调用时由被调用者保存和恢复，以保护被调用者的寄存器内容。Caller 不需要保存这些寄存器。
4. Caller Saved Registers：这些寄存器（a0-a7, t0-t6）由调用者保存和恢复，以保护调用者的寄存器内容。被调用者不需要保存这些寄存器。

区别在于 Caller Saved Registers 和 Callee Saved Registers 的管理责任不同。Caller Saved Registers 是由函数调用者（Caller）来保存和恢复，而 Callee Saved Registers 是由函数被调用者（Callee）来保存和恢复。这有助于确保在函数调用过程中不会意外破坏调用者或被调用者的寄存器内容，从而维护了寄存器的一致性。

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

号的值（截图）。

```
hwj@hwj-virtual-machine:~/os23fall-stu/src/lab1$ nm vmlinux
0000000080200000 t $x
000000008020002c t $x
000000008020015c t $x
0000000080200184 t $x
00000000802001e8 t $x
00000000802002c4 t $x
0000000080200340 t $x
0000000080200384 t $x
0000000080200394 t $x
00000000802003e4 t $x
00000000802008c0 t $x
0000000080200000 A BASE_ADDR
0000000080203000 B boot_stack
0000000080204000 B boot_stack_top
0000000080200184 T clock_set_next_event
0000000080204000 B _ebss
0000000080202010 D _edata
0000000080204000 B _ekernel
0000000080201098 R _erodata
0000000080200940 T _etext
0000000080202008 D flag
000000008020015c T get_cycles
0000000080202010 d _GLOBAL_OFFSET_TABLE_
00000000802008c0 T printk
0000000080200394 T putc
00000000802001e8 T sbi_ecall
0000000080203000 B _sbss
0000000080202000 D _sdata
0000000080200000 T _skernel
0000000080201000 R srodata
0000000080200000 T _start
0000000080200340 T start_kernel
0000000080200000 T _stext
0000000080200384 T test
0000000080202000 D TIMECLOCK
00000000802002c4 T trap_handler
000000008020002c T _traps
00000000802003e4 t vprintfmt
```

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）。

先修改 main.c 的代码，通过调用 printk 来格式化输出 sstatus 寄存器的值，如下图所示：



```
1 #include "printk.h"
2 #include "sbi.h"
3 #include "defs.h"
4
5 extern void test();
6
7 int start_kernel() {
8     printk("2023");
9     printk(" Hello RISC-V\n");
10
11     uint64 result = csr_read(sstatus);
12     printk("15: %x\n",15); // Verify the usage of the printk function:
13     printk("sstatus: %x\n",result);
14
15     result = csr_read(sie);
16     printk("sie: %x\n",result);
17
18     // csr_write(sscratch, 0x1);
19
20     test(); // DO NOT DELETE !!!
21
22     return 0;
23 }
```

make run 之后结果如下图所示：

```

hwj@hwj-virtual-machine: ~/os23fall-stu/src/lab1
Domain0 HARTs      : 0*
Domain0 Region00   : 0x0000000080000000-0x000000008001ffff ( )
Domain0 Region01   : 0x0000000000000000-0xffffffffffff (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       : rv64imafdcu
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
2023 Hello RISC-V
15: 0000000f
sstatus: 00006002
sie: 00000020

```

可以看到 sstatus 寄存器的十六进制表示是 0x00006002, 查看 RISC-V 手册得知 sstatus 寄存器的格式如下:

XLLEN-1		XLLEN-2										20	19	18	17
SD		0										MXR	SUM	0	
1		XLLEN-21										1	1	1	
16	15	14	13	12	9	8	7	6	5	4	3	2	1	0	
XS[1:0]		FS[1:0]		0		SPP	0	SPIE	UPIE	0	SIE	UIE			
2		2		4		1	2	1	1	2	1	1			

由此可知此时 sstatus 寄存器的 SIE 位为 1 且 FS[1:0] 为 1, 此时表示系统会响应所有的 S 态 trap。

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)。

先修改 main.c 的代码, 先读取并打印 write 之前的 sscratch 寄存器的值, 再调用 csr_write 函数修改 sscratch 寄存器的值, 并再次读取和打印来对前后两

次的寄存器值作对比，如下图所示：



```
main.c
~/os23fall-stu/src/lab1/init

< clock.h × clock.c × sbi.c × trap.c × head.S × entry.S × main.c × printk.c × >

1 #include "printk.h"
2 #include "sbi.h"
3 #include "defs.h"
4
5 extern void test();
6
7 int start_kernel() {
8     printk("2023");
9     printk(" Hello RISC-V\n");
10
11     uint64 result = csr_read(sstatus);
12     printk("sstatus: %x\n", result);
13     printk("sstatus: %x\n", result);
14
15     result = csr_read(sie);
16     printk("sie: %x\n", result);
17
18     result = csr_read(sscratch);
19     printk("prev sscratch: %x\n", result);
20     csr_write(sscratch, 0x1);
21     result = csr_read(sscratch);
22     printk("later sscratch: %x\n", result);
23
24     test(); // DO NOT DELETE !!!
25
26     return 0;
27 }
```

输出结果如下：

```
hwj@hwj-virtual-machine: ~/os23fall-stu/src/lab1
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
2023 Hello RISC-V
15: 0000000f
sstatus: 00006002
sie: 00000020
prev sscratch: 00000000
later sscratch: 00000001
```

发现写入成功。

5. Detail your steps about how to get arch/arm64/kernel/sys.i

(1) 先为编译指定默认 config

```
hwj@hwj-virtual-machine:~/Linux/linux-6.5.6$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
```

(2) 然后指定要生成的文件 (arch/arm64/kernel/sys.i)

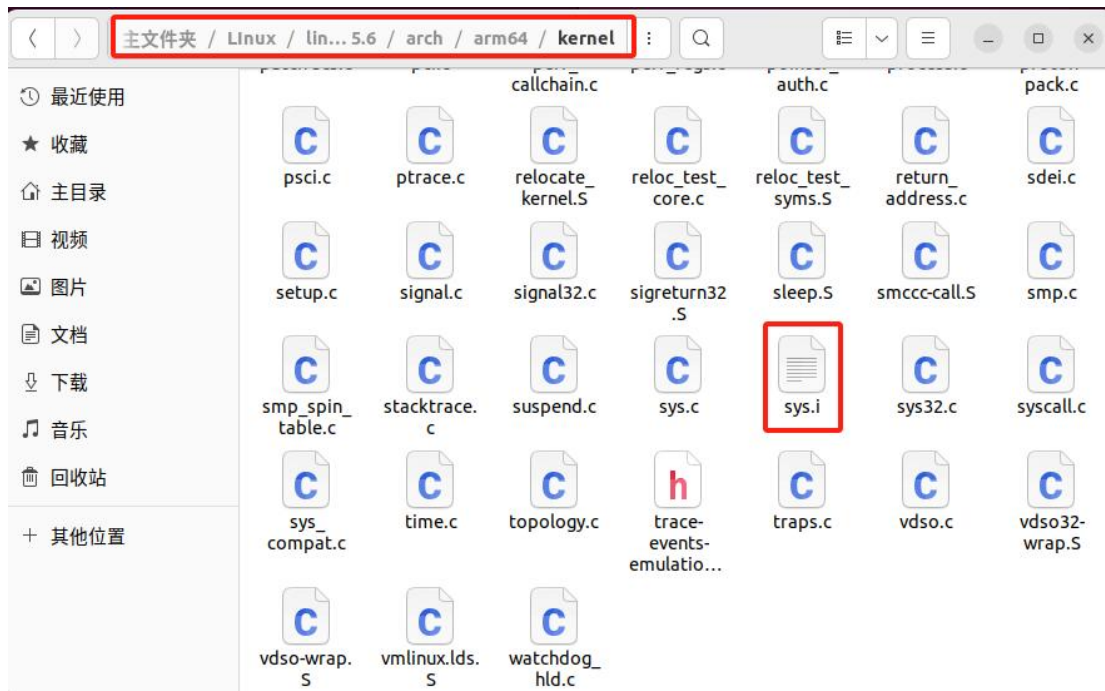
```

hwj@hwj-virtual-machine: ~/Linux/linux-6.5.0$ 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
WRAP      arch/arm64/include/generated/uapi/asm/msgbuf.h
WRAP      arch/arm64/include/generated/uapi/asm/poll.h
WRAP      arch/arm64/include/generated/uapi/asm/resource.h
WRAP      arch/arm64/include/generated/uapi/asm/smbuf.h
WRAP      arch/arm64/include/generated/uapi/asm/shmbuf.h
WRAP      arch/arm64/include/generated/uapi/asm/siginfo.h
WRAP      arch/arm64/include/generated/uapi/asm/socket.h
WRAP      arch/arm64/include/generated/uapi/asm/sockios.h
WRAP      arch/arm64/include/generated/uapi/asm/stat.h
WRAP      arch/arm64/include/generated/uapi/asm/swab.h
WRAP      arch/arm64/include/generated/uapi/asm/termbits.h
WRAP      arch/arm64/include/generated/uapi/asm/termios.h
WRAP      arch/arm64/include/generated/uapi/asm/types.h
WRAP      arch/arm64/include/generated/asm/early_ioremap.h
WRAP      arch/arm64/include/generated/asm/mcs_spinlock.h
WRAP      arch/arm64/include/generated/asm/qrwlock.h
WRAP      arch/arm64/include/generated/asm/qspinlock.h
WRAP      arch/arm64/include/generated/asm/parport.h
WRAP      arch/arm64/include/generated/asm/user.h
WRAP      arch/arm64/include/generated/asm/bugs.h
WRAP      arch/arm64/include/generated/asm/delay.h
WRAP      arch/arm64/include/generated/asm/softirq_stack.h
WRAP      arch/arm64/include/generated/asm/switch_to.h
WRAP      arch/arm64/include/generated/asm/trace_clock.h
WRAP      arch/arm64/include/generated/asm/unaligned.h
WRAP      arch/arm64/include/generated/asm/vga.h
UPD      include/generated/compile.h
GEN      arch/arm64/include/generated/asm/cpucaps.h
GEN      arch/arm64/include/generated/asm/sysreg-defs.h
CC       scripts/mod/empty.o
MKELF    scripts/mod/elfconfig.h
HOSTCC   scripts/mod/modpost.o
CC       scripts/mod/devicetable-offsets.s
HOSTCC   scripts/mod/file2alias.o
HOSTCC   scripts/mod/sumversion.o
HOSTLD   scripts/mod/modpost
CC       kernel/bounds.s
UPD      include/generated/bounds.h
CC       arch/arm64/kernel/asm-offsets.s
UPD      include/generated/asm-offsets.h
CALL     scripts/checksyscalls.sh
LDS      arch/arm64/kernel/vdso/vdso.lds
CC       arch/arm64/kernel/vdso/vgettimeofday.o
AS       arch/arm64/kernel/vdso/note.o
AS       arch/arm64/kernel/vdso/sigreturn.o
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

```

(3) 查看结果，发现在指定目录下找到了 sys.i 文件



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.

I'm very sorry I didn't complete this part, it's just that I found this part to be too time-consuming.

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) 是一种常用的二进制文件格式，用于在 Linux 和许多其他操作系统上存储可执行程序、共享库和目标文件。它是一种灵活且可扩展的文件格式，广泛用于编译、链接和运行软件。

执行 readelf 指令：

```
hwj@hwj-virtual-machine:~/Linux/linux-6.5.6$ readelf vmlinux
用法: readelf <选项> elf-文件
显示关于 ELF 格式文件内容的信息
Options are:
-a --all                Equivalent to: -h -l -S -s -r -d -V -A -I
-h --file-header        Display the ELF file header
-l --program-headers    Display the program headers
  --segments            An alias for --program-headers
-S --section-headers    Display the sections' header
  --sections            An alias for --section-headers
-g --section-groups     Display the section groups
-t --section-details    Display the section details
-e --headers            Equivalent to: -h -l -S
-s --syms               Display the symbol table
  --symbols             An alias for --syms
  --dyn-syms            Display the dynamic symbol table
  --lto-syms            Display LTO symbol tables
  --sym-base=[0|8|10|16]
                        Force base for symbol sizes. The options are
                        mixed (the default), octal, decimal, hexadecimal.
-C --demangle[=STYLE]   Decode mangled/processed symbol names
                        STYLE can be "none", "auto", "gnu-v3", "java",
                        "gnat", "dlang", "rust"
  --no-demangle         Do not demangle low-level symbol names. (default)
  --recurse-limit       Enable a demangling recursion limit. (default)
  --no-recurse-limit    Disable a demangling recursion limit
-U[d|l|e|x|h|i] --unicode=[default|locale|escape|hex|highlight|invalid]
                        Display unicode characters as determined by the current locale
                        (default), escape sequences, "<hex sequences>", highlighted
                        escape sequences, or treat them as invalid and display as
```

```
hwj@hwj-virtual-machine:~/Linux/linux-6.5.6$ readelf -h vmlinux
ELF 头:
Magic:      7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00
类别:                      ELF64
数据:                      2 补码, 小端序 (little endian)
Version:          1 (current)
OS/ABI:           UNIX - System V
ABI 版本:         0
类型:             EXEC (可执行文件)
系统架构:         RISC-V
版本:             0x1
入口点地址:       0xffffffff80000000
程序头起点:       64 (bytes into file)
Start of section headers: 18849448 (bytes into file)
标志:            0x1, RVC, soft-float ABI
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 9
Size of section headers: 64 (bytes)
Number of section headers: 31
Section header string table index: 30
```

执行 objdump 指令:


```
hwj@hwj-virtual-machine:~/Linux/linux-6.5.6$ objdump vmlinux
```

用法: objdump <选项> <文件>

显示来自目标 <文件> 的信息。

至少必须给出以下选项之一:

```
-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, --dwarf[a/=abbrev, A/=addr, r/=aranges, c/=cu_index, L/=decodedline,
    f/=frames, F/=frames-interp, g/=gdb_index, i/=info, o/=loc,
    m/=macro, p/=pubnames, t/=pubtypes, R/=Ranges, l/=rawline,
    s/=str, O/=str-offsets, u/=trace_abbrev, T/=trace_aranges,
    U/=trace_info]
    Display the contents of DWARF debug sections
-Wk,--dwarf=links        Display the contents of sections that link to
    separate debuginfo files
-WK,--dwarf=follow-links
```

```
hwj@hwj-virtual-machine:~/Linux/linux-6.5.6$ objdump -f vmlinux
```

vmlinux: 文件格式 elf64-little

体系结构: UNKNOWN!, 标志 0x00000112:

EXEC_P, HAS_SYMS, D_PAGED

起始地址 0xffffffff80000000

用 qemu 来运行 vmlinux 这个 ELF 文件:

```
hwj@hwj-virtual-machine:~$ qemu-system-riscv64 -nographic -machine virt -kernel
/home/hwj/Linux/linux-6.5.6/arch/riscv/boot/Image \
    -device virtio-blk-device,drive=hd0 -append "root=/dev/vda ro console=ttyS0"
\
    -bios default -drive file=/home/hwj/os23fall-stu/src/lab0/rootfs.img,format=
raw,id=hd0

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
```

执行 ps aux 来查看当前系统的所有进程信息:

```
hwj@hwj-virtual-machine:~$ ps aux
```

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
root	1	0.0	0.2	166756	11388	?	Ss	10月16	0:04	/sbin/init
root	2	0.0	0.0	0	0	?	S	10月16	0:00	[kthreadd]
root	3	0.0	0.0	0	0	?	I<	10月16	0:00	[rcu_gp]
root	4	0.0	0.0	0	0	?	I<	10月16	0:00	[rcu_par_gp]
root	5	0.0	0.0	0	0	?	I<	10月16	0:00	[slub_flush]
root	6	0.0	0.0	0	0	?	I<	10月16	0:00	[netns]
root	8	0.0	0.0	0	0	?	I<	10月16	0:00	[kworker/0:
root	10	0.0	0.0	0	0	?	I<	10月16	0:00	[mm_percpu_
root	11	0.0	0.0	0	0	?	I	10月16	0:00	[rcu_tasks_
root	12	0.0	0.0	0	0	?	I	10月16	0:00	[rcu_tasks_
root	13	0.0	0.0	0	0	?	I	10月16	0:00	[rcu_tasks_
root	14	0.0	0.0	0	0	?	S	10月16	0:00	[ksoftirqd/
root	15	0.0	0.0	0	0	?	I	10月16	0:03	[rcu_preemp
root	16	0.0	0.0	0	0	?	S	10月16	0:00	[migration/
root	17	0.0	0.0	0	0	?	S	10月16	0:00	[idle_injec
root	19	0.0	0.0	0	0	?	S	10月16	0:00	[cpuhp/0]

```

root      6000  0.0  0.0      0      0 ?      I   00:12  0:00 [kworker/3:1-
root      7841  0.0  0.0      0      0 ?      I   00:28  0:00 [kworker/u256
root      7854  0.0  0.0      0      0 ?      R   00:34  0:00 [kworker/u256
root      7859  0.0  0.0      0      0 ?      I   00:35  0:00 [kworker/2:1-
hwj      7876  0.0  1.7 2909348 68356 ?      Sl  00:35  0:01 gjs /usr/shar
root      7974  0.1  0.0      0      0 ?      I   00:40  0:02 [kworker/1:2-
root      7986  0.0  0.0      0      0 ?      I   00:43  0:00 [kworker/0:0-
root      8788  0.0  0.0      0      0 ?      I   00:45  0:00 [kworker/2:0-
root      9055  0.0  0.0      0      0 ?      I   00:54  0:00 [kworker/3:2-
root      9236  0.1  0.0      0      0 ?      I   01:01  0:00 [kworker/1:1-
root      9239  0.0  0.0      0      0 ?      I   01:03  0:00 [kworker/0:1-
root      9252  0.0  0.0      0      0 ?      I   01:03  0:00 [kworker/2:2-
root      9256  0.0  0.0      0      0 ?      I   01:03  0:00 [kworker/u256
root      9295  0.0  0.0      0      0 ?      I   01:03  0:00 [kworker/3:0-
root      9296  0.0  0.0      0      0 ?      I   01:03  0:00 [kworker/2:3-
root      9312  0.0  0.0      0      0 ?      I   01:06  0:00 [kworker/1:0-
hwj      9402  6.2  1.7 649780 69636 ?      Rsl 01:07  0:02 /usr/libexec/
hwj      9442  0.0  0.1 14360  5120 pts/1   Ss  01:07  0:00 bash
root      9450  0.0  0.0      0      0 ?      I   01:07  0:00 [kworker/3:3]
root      9459  0.1  0.0      0      0 ?      D   01:07  0:00 [kworker/1:3+
hwj      9460  0.0  0.1 14360  5120 pts/0   Ss  01:07  0:00 bash
hwj      9468 35.5  3.7 1026548 150060 pts/0   Sl+ 01:07  0:04 qemu-system-r
hwj      9473  0.0  0.0 15776  3328 pts/1   R+  01:07  0:00 ps aux

```

```
qemu-system-riscv64 -nographic -machine virt -kernel vmlinux -bios default
```

（找到对应的 PID 为 9468）

执行 `cat /proc/9468/maps` 来查看 ELF 文件的内存布局；


```

hwj@hwj-virtual-machine:~$ cat /proc/9468/maps
55a936b08000-55a936ce000 r--p 00000000 08:03 1236549 /usr/bin/qemu-system-riscv64
55a936ce000-55a93746c000 r-xp 003c6000 08:03 1236549 /usr/bin/qemu-system-riscv64
55a93746c000-55a937704000 r--p 00964000 08:03 1236549 /usr/bin/qemu-system-riscv64
55a937704000-55a937864000 r--p 00bfb000 08:03 1236549 /usr/bin/qemu-system-riscv64
55a937864000-55a937977000 rw-p 00d5b000 08:03 1236549 /usr/bin/qemu-system-riscv64
55a937977000-55a9379a9000 rw-p 00000000 00:00 0
55a938171000-55a9388f4000 rw-p 00000000 00:00 0 [heap]
7f35ac000000-7f35ac021000 rw-p 00000000 00:00 0
7f35ac021000-7f35b0000000 ---p 00000000 00:00 0
7f35b09e2000-7f35b09e3000 ---p 00000000 00:00 0
7f35b09e3000-7f35b0ae3000 rw-p 00000000 00:00 0
7f35b0ae3000-7f35b0ae4000 ---p 00000000 00:00 0
7f35b0ae4000-7f35b0be4000 rw-p 00000000 00:00 0
7f35b0be4000-7f35b0be5000 ---p 00000000 00:00 0
7f35b0be5000-7f35b0ce5000 rw-p 00000000 00:00 0
7f35b0ce5000-7f35b0ce6000 ---p 00000000 00:00 0
7f35b0ce6000-7f35b0de6000 rw-p 00000000 00:00 0
7f35b0de6000-7f35b0de7000 ---p 00000000 00:00 0
7f35b0de7000-7f35b0ee7000 rw-p 00000000 00:00 0
7f35b0ee7000-7f35b0ee8000 ---p 00000000 00:00 0
7f35b0ee8000-7f35b0fe8000 rw-p 00000000 00:00 0
7f35b0fe8000-7f35b0fe9000 ---p 00000000 00:00 0
7f35b0fe9000-7f35b10e9000 rw-p 00000000 00:00 0
7f35b10e9000-7f35b10ea000 ---p 00000000 00:00 0
7f35b10ea000-7f35b11ea000 rw-p 00000000 00:00 0
7f35b11ea000-7f35b11eb000 ---p 00000000 00:00 0

```

```

7f35ea115000-7f35ea120000 r--p 0002e000 08:03 1205404 /usr/lib/x86_64-linux-gnu/libpng16.so.16.37.0
7f35ea120000-7f35ea121000 r--p 00038000 08:03 1205404 /usr/lib/x86_64-linux-gnu/libpng16.so.16.37.0
7f35ea121000-7f35ea122000 rw-p 00039000 08:03 1205404 /usr/lib/x86_64-linux-gnu/libpng16.so.16.37.0
7f35ea122000-7f35ea124000 r--p 00000000 08:03 1205864 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
7f35ea124000-7f35ea135000 r-xp 00002000 08:03 1205864 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
7f35ea135000-7f35ea13b000 r--p 00013000 08:03 1205864 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
7f35ea13b000-7f35ea13c000 ---p 00019000 08:03 1205864 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
7f35ea13c000-7f35ea13d000 r--p 00019000 08:03 1205864 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
7f35ea13d000-7f35ea13e000 rw-p 0001a000 08:03 1205864 /usr/lib/x86_64-linux-gnu/libz.so.1.2.11
7f35ea13e000-7f35ea148000 r--p 00000000 08:03 1205386 /usr/lib/x86_64-linux-gnu/libpixman-1.so.0.40.0
7f35ea148000-7f35ea1ce000 r-xp 0000a000 08:03 1205386 /usr/lib/x86_64-linux-gnu/libpixman-1.so.0.40.0
7f35ea1ce000-7f35ea1e0000 r--p 00090000 08:03 1205386 /usr/lib/x86_64-linux-gnu/libpixman-1.so.0.40.0
7f35ea1e0000-7f35ea1e8000 r--p 000a1000 08:03 1205386 /usr/lib/x86_64-linux-gnu/libpixman-1.so.0.40.0
7f35ea1e8000-7f35ea1e9000 rw-p 000a9000 08:03 1205386 /usr/lib/x86_64-linux-gnu/libpixman-1.so.0.40.0
7f35ea1e9000-7f35ea1eb000 r--p 00000000 08:03 1236523 /usr/lib/x86_64-linux-gnu/libfdt-1.6.1.so
7f35ea1eb000-7f35ea1f0000 r-xp 00002000 08:03 1236523 /usr/lib/x86_64-linux-gnu/libfdt-1.6.1.so
7f35ea1f0000-7f35ea1f2000 r--p 00007000 08:03 1236523 /usr/lib/x86_64-linux-gnu/libfdt-1.6.1.so
7f35ea1f2000-7f35ea1f3000 r--p 00008000 08:03 1236523 /usr/lib/x86_64-linux-gnu/libfdt-1.6.1.so
7f35ea1f3000-7f35ea1f4000 rw-p 00009000 08:03 1236523 /usr/lib/x86_64-linux-gnu/libfdt-1.6.1.so
7f35ea204000-7f35ea206000 rw-p 00000000 00:00 0
7f35ea206000-7f35ea208000 r--p 00000000 08:03 1233269 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f35ea208000-7f35ea232000 r-xp 00002000 08:03 1233269 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f35ea232000-7f35ea23d000 r--p 0002c000 08:03 1233269 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f35ea23e000-7f35ea240000 r--p 00037000 08:03 1233269 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f35ea240000-7f35ea242000 rw-p 00039000 08:03 1233269 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffffebcd000-7ffffebef000 rw-p 00000000 00:00 0 [stack]
7ffffebf2000-7ffffebf6000 r--p 00000000 00:00 0 [vvar]
7ffffebf6000-7ffffebf8000 r-xp 00000000 00:00 0 [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]

```

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

```

1      OpensBI v0.9
2
3      _____
4      /  _  \           /  _  |  _  \  _  |
5      | | | | | _  _  _  | ( _  | | ) | | |
6      | | | | | ' _ \ / _ \ ' _ \ \ _ \ | | |
7      | | _ | | | ) | _ / | | | _ ) | | ) | | |
8      \ _ _ / | . _ / \ _ | | | | _ _ / | _ _ / | _ _ /
9
10     | |
11     | |
12
13     Boot HART MIDELEG      : 0x00000000000000222
14     Boot HART MEDELEG     : 0x0000000000000b109
15
16     .....

```

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

```

hwj@hwj-virtual-machine:~/os23fall-stu/src/lab1$ make run
make -C lib all
make[1]: 进入目录"/home/hwj/os23fall-stu/src/lab1/lib"
make[1]: 对"all"无需做任何事。
make[1]: 离开目录"/home/hwj/os23fall-stu/src/lab1/lib"
make -C init all
make[1]: 进入目录"/home/hwj/os23fall-stu/src/lab1/init"
make[1]: "all"已是最新。
make[1]: 离开目录"/home/hwj/os23fall-stu/src/lab1/init"
make -C arch/riscv all
make[1]: 进入目录"/home/hwj/os23fall-stu/src/lab1/arch/riscv"
make -C kernel all
make[2]: 进入目录"/home/hwj/os23fall-stu/src/lab1/arch/riscv/kernel"
make[2]: 对"all"无需做任何事。
make[2]: 离开目录"/home/hwj/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]: 离开目录"/home/hwj/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    : 0x0000000000000000-0xffffffffffff (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       : rv64inafdcsu
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

```

MIDELEG 是 RISC-V 架构中的一个机器级别的寄存器，用于控制将哪些中断委托给当前执行上下文的特权级（Machine mode）处理。它是一个 64 位的寄存器，每个位对应一种特定的异常或中断。

该寄存器的每个位控制特定类型的异常或中断是否被委托给低特权级（例如 Supervisor mode）处理。如果相应位设置为 1，则将该类型的异常或中断委托给低特权级处理。

具体来说，这个二进制数 0x0000000000000222 可以解释为：

Bit 1（LSB）：设为 1，表示将指令地址错误异常（Instruction Address Misaligned）委托给低特权级处理。

Bit 5：设为 1，表示将环境调用异常（Environment Call）委托给低特权级处理。

Bit 9：设为 1，表示将系统调用异常（Supervisor Call）委托给低特权级处理。其他位通常设为 0，表示不委托其他异常给低特权级处理。

所以，该 MIDELEG 值的含义是将指令地址错误异常、环境调用异常和系统调用异常委托给低特权级处理，其他异常保持在机器模式处理。

五、附录

Interrupt	Exception Code	Description
1	0	<i>Reserved</i>
1	1	Supervisor software interrupt
1	2	<i>Reserved</i>
1	3	Machine software interrupt
1	4	<i>Reserved</i>
1	5	Supervisor timer interrupt
1	6	<i>Reserved</i>
1	7	Machine timer interrupt
1	8	<i>Reserved</i>
1	9	Supervisor external interrupt
1	10	<i>Reserved</i>
1	11	Machine external interrupt
1	12–15	<i>Reserved</i>
1	≥ 16	<i>Designated for platform use</i>
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	<i>Reserved</i>
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	<i>Reserved</i>
0	15	Store/AMO page fault
0	16–23	<i>Reserved</i>
0	24–31	<i>Designated for custom use</i>
0	32–47	<i>Reserved</i>
0	48–63	<i>Designated for custom use</i>
0	≥ 64	<i>Reserved</i>

Table 3.6: Machine cause register (`mcause`) values after trap.