

浙江大学

操作系统实验一

课程名称:	操作系统
作业名称:	RV64 内核引导与时钟中断处理
姓 名:	仇国智
学 号:	3220102181
电子邮箱:	3220102181@zju.edu.cn
联系电话:	13714176104
指导教师:	申文博

2024 年 9 月 24 日

目录

1	实验内容及原理	1
1.1	实验内容	1
1.2	实验原理	1
2	实验具体过程与代码实现	3
2.1	RV64 内核引导	3
2.2	RV64 时钟中断处理	7
2.3	实现时钟中断相关函数	13
3	实验结果与分析	14
4	遇到的问题及解决方法	14
5	总结与心得	15
6	思考题	15
6.1	请总结一下 RISC-V 的 calling convention, 并解释 Caller / Callee Saved Register 有什么区别?	15
6.2	编译之后, 通过 System.map 查看 vmlinux.lds 中自定义符号的值并截图.	16
6.3	用 csr_read 宏读取 sstatus 寄存器的值, 对照 RISC-V 手册解释其含义并截图.	17
6.4	用 csr_write 宏向 sscratch 寄存器写入数据, 并验证是否写入成功并截图.	18
6.5	详细描述你可以通过什么步骤来得到 arch/arm64/kernel/sys.i, 给出过程以及截图.	19
6.6	寻找 Linux v6.0 中 ARM32 RV32 RV64 x86_64 架构的系统调用表; 请列出源代码文件, 展示完整的系统调用表 (宏展开后), 每一步都需要截图.	21
6.7	阐述什么是 ELF 文件? 尝试使用 readelf 和 objdump 来查看 ELF 文件, 并给出解释和截图. 运行一个 ELF 文件, 然后通过 cat /proc/PID/maps 来给出其内存布局并截图.	29

6.8	在我们使用 <code>make run</code> 时,OpenSBI 会产生如下输出. 通过查看 RISC-V Privileged Spec 中的 <code>medeleg</code> 和 <code>mideleg</code> 部分, 解释上面 <code>MIDELEG</code> 和 <code>MEDELEG</code> 值的含义.	33
-----	--	----

1 实验内容及原理

1.1 实验内容

- 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.

1.2 实验原理

RISC-V 特权级别

RISC-V 有三个特权模式:U(user) 模式,S(supervisor) 模式和 M(machine) 模式.

Level	Encoding	Name
0	00	User/Application
1	01	Supervisor
2	10	
3	11	Machine

表 1: RISC-V 特权级别

其中:

M 模式是对硬件操作的抽象, 有最高级别的权限; S 模式介于 M 模式和 U 模式之间, 在操作系统中对应于内核态 (kernel). 当用户需要内核资源时, 向内核申请, 并切换到内核态进行处理; U 模式用于执行用户程序, 在操作系统中对应于用户态, 有最低级别的权限.

我们以最基础的嵌入式系统为例, 计算机上电后, 首先硬件进行一些基础的初始化后, 将 CPU 的 Program Counter 移动到内存中 bootloader 的起始地址.

Bootloader 是操作系统内核运行之前, 用于初始化硬件, 加载操作系统内核.

在 RISC-V 架构里,bootloader 运行在 M 模式下.Bootloader 运行完毕后就会把当前模式切换到 S 模式下, 机器随后开始运行 kernel.

OpenSBI

SBI(Supervisor Binary Interface) 是 S-mode 的 Kernel 和 M-mode 执行环境之间的接口规范, 而 OpenSBI 是一个 RISC-V SBI 规范的开源实现.RISC-V 平台和 SoC 供应商可以自主扩展 OpenSBI 实现, 以适应特定的硬件配置.

简单的说, 为了使操作系统内核适配不同硬件,OpenSBI 提出了一系列规范对 M-mode 下的硬件进行了统一定义, 运行在 S-mode 下的内核可以按照这些规范对不同硬件进行操作.

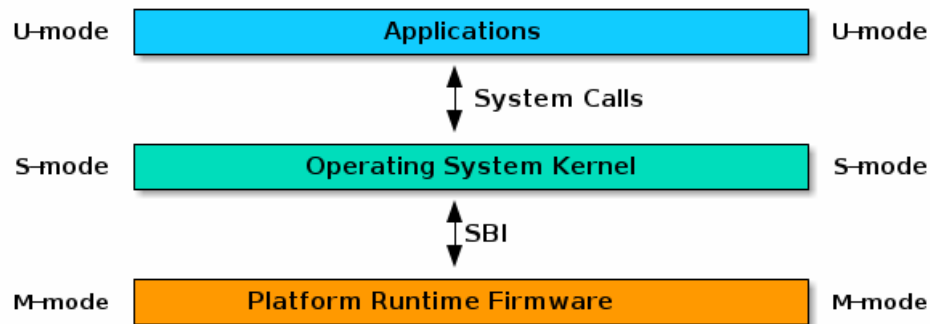


图 1: OpenSBI

异常寄存器

除了 32 个通用寄存器之外,RISC-V 架构还有大量的控制状态寄存器 (Control and Status Registers,CSRs), 下面将介绍几个和 trap 机制相关的重要寄存器.

Supervisor Mode 下 trap 相关寄存器:

- sstatus(Supervisor Status Register) 中存在一个 SIE(Supervisor Interrupt Enable) 比特位, 当该比特位设置为 1 时, 会响应所有的 S 态 trap, 否则将会禁用所有 S 态 trap.
- sie(Supervisor Interrupt Eable Register), 在 RISC-V 中,interrupt 被划分为三类 software interrupt,timer interrupt,external interrupt. 在开启了 sstatus[SIE] 之后, 系统会根据 sie 中的相关比特位来决定是否对该 interrupt 进行处理.
- stvec(Supervisor Trap Vector Base Address Register) 即所谓的”中断向量表基址”.stvec 有两种模式:Direct 模式, 适用于系统中只有一个中断处理程序, 其指向

中断处理入口函数 (本次实验中我们所用的模式).Vectored 模式, 指向中断向量表, 适用于系统中有多中断处理程序 (该模式可以参考 RISC-V 内核源码).

- scause(Supervisor Cause Register), 会记录 trap 发生的原因, 还会记录该 trap 是 interrupt 还是 exception.
- sepc(Supervisor Exception Program Counter), 会记录触发 exception 的那条指令的地址.

Machine Mode 异常相关寄存器: 类似于 Supervisor Mode, Machine Mode 也有相对应的寄存器, 但由于本实验同学不需要操作这些寄存器, 故不在此作介绍.

2 实验具体过程与代码实现

2.1 RV64 内核引导

完善 Makefile 脚本

```
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)
```

参考同级文件夹 init 下的 Makefile 脚本, 完善 lib 文件夹下的 Makefile 脚本, 收集所有的.c 文件, 并将其编译为.o 文件. 并添加 clean 指令, 清除所有.o 文件.

编写 head.S

```
1     .extern start_kernel
2     .extern TIMECLOCK
3     .extern sbi_ecall
4     .extern _traps
```

```

5     .section .text.init
6     .globl _start
7     STIE = 0x01 << 5
8     SIE = 0x01 << 1
9     _start:
10    la sp, boot_stack_top
11
12    la a0, _traps
13    csrw stvec, a0
14
15    li a0, STIE
16    csrs sie, a0
17
18    rdttime a0
19    la a1, TIMECLOCK
20    ld a1, 0(a1)
21    add a0, a0, a1
22    li a6, 0
23    li a7, 0x54494d45
24    ecalls # SBI_SET_TIMER
25
26    li a0, SIE
27    csrs sstatus, a0
28    call start_kernel
29
30    .section .bss.stack
31    .globl boot_stack
32    boot_stack:
33    .space 0x1000 # <-- change to your stack size
34
35    .globl boot_stack_top
36    boot_stack_top:

```

在 head.S 中, 首先设置栈指针, 然后设置中断处理函数的地址, 并将其写入 stvec 寄存器. 接着设置 STIE 位, 使能时钟中断, 并设置时钟中断的时间. 最后设置 SIE 位, 使能

中断, 并调用 start_kernel 函数.

编写 sbi.c

```
1 struct sbiret sbi_ecall(uint64_t eid, uint64_t fid,
2                          uint64_t arg0, uint64_t arg1, uint64_t arg2,
3                          uint64_t arg3, uint64_t arg4, uint64_t arg5)
4 {
5     struct sbiret ret;
6     __asm__ volatile(
7         " move a0, %2 \n"
8         " move a1, %3 \n"
9         " move a2, %4 \n"
10        " move a3, %5 \n"
11        " move a4, %6 \n"
12        " move a5, %7 \n"
13        " move a6, %8 \n"
14        " move a7, %9 \n"
15        " ecall \n"
16        " move %0, a0 \n"
17        " move %1, a1 \n"
18        : "=r"(ret.error), "=r"(ret.value)
19        : "r"(arg0), "r"(arg1), "r"(arg2), "r"(arg3), "r"(arg4),
20          "r"(arg5), "r"(fid), "r"(eid)
21        : "memory");
22    return ret;
23 }
24 struct sbiret sbi_set_timer(uint64_t stime_value)
25 {
26     struct sbiret ret;
27     ret = sbi_ecall(0x54494d45, 0x0, stime_value, 0, 0, 0, 0, 0);
28     return ret;
29 }
30 struct sbiret sbi_system_reset(uint32_t reset_type, uint32_t
    reset_reason)
```



```

31 {
32     struct sbiret ret;
33     ret = sbi_ecall(0x53525354, 0x0, reset_type, reset_reason, 0, 0,
34                     0, 0);
35     return ret;
36 }
37 struct sbiret sbi_debug_console_write(unsigned long
38     num_bytes, unsigned long base_addr_lo, unsigned long base_addr_hi)
39 {
40     struct sbiret ret;
41     ret = sbi_ecall(0x4442434e, 0x0, num_bytes, base_addr_lo,
42                     base_addr_hi, 0, 0, 0);
43     return ret;
44 }
45 struct sbiret sbi_debug_console_read(unsigned long
46     num_bytes, unsigned long base_addr_lo, unsigned long base_addr_hi)
47 {
48     struct sbiret ret;
49     ret = sbi_ecall(0x4442434e, 0x1, num_bytes, base_addr_lo,
50                     base_addr_hi, 0, 0, 0);
51     return ret;
52 }
53 struct sbiret sbi_debug_console_write_byte(uint8_t byte)
54 {
55     struct sbiret ret;
56     ret = sbi_ecall(0x4442434e, 0x2, byte, 0, 0, 0, 0, 0);
57     return ret;
58 }

```

在 sbi.c 中, 定义了 sbi_ecall 函数, 在该函数中, 将参数传递给寄存器, 并调用 ecall 指令, 这样在后面几个函数中, 只需要传递参数即可, 无需和汇编交互. 之后定义了 sbi_set_timer, 用于设置时钟中断的时间; 定义了 sbi_system_reset, 用于重置系统; 定义了 sbi_debug_console_write, 用于向控制台输出; 定义了 sbi_debug_console_read, 用于从控制台读取; 定义了 sbi_debug_console_write_byte, 用于向控制台输出一个字节.

修改 defs

```
1  #ifndef __DEFS_H__
2  #define __DEFS_H__
3
4  #include "stdint.h"
5
6  #define csr_read(csr) \
7      ({ \
8          uint64_t __v; \
9          asm volatile("csrr %0, " #csr \
10                      : "=r"(__v) \
11                      : \
12                      : "memory"); \
13          __v; \
14      })
15
16  #define csr_write(csr, val) \
17      ({ \
18          uint64_t __v = (uint64_t)(val); \
19          asm volatile("csrw " #csr ", %0" : : "r"(__v) : "memory"); \
20      })
21
22  #endif
```

在 defs.h 中, 定义了 csr_read 和 csr_write 两个宏, 用于读取和写入 csr 寄存器.

2.2 RV64 时钟中断处理

修改 vmlinux.lds

```
1  .text : ALIGN(0x1000) {
2      _stext = .;
3
4      *(.text .text.init)
5      *(.text.entry)
```

```

6          *(.text .text.*)
7          ...
8      }
9      ...

```

将.text.init 段用于存放启动代码, 并将.text.entry 段改用于存放中断处理函数.

实现上下文切换

```

1      .extern trap_handler
2      .section .text.entry
3      .align 2
4      .globl _traps
5  _traps:
6
7      sd x2, -240(sp)
8      addi sp, sp, -256
9      sd x1, 8(sp)
10     csrr x1, sepc
11     sd x1, 0(sp)
12     sd x3, 24(sp)
13     sd x4, 32(sp)
14     sd x5, 40(sp)
15     sd x6, 48(sp)
16     sd x7, 56(sp)
17     sd x8, 64(sp)
18     sd x9, 72(sp)
19     sd x10, 80(sp)
20     sd x11, 88(sp)
21     sd x12, 96(sp)
22     sd x13, 104(sp)
23     sd x14, 112(sp)
24     sd x15, 120(sp)
25     sd x16, 128(sp)
26     sd x17, 136(sp)
27     sd x18, 144(sp)

```

```

28     sd x19, 152(sp)
29     sd x20, 160(sp)
30     sd x21, 168(sp)
31     sd x22, 176(sp)
32     sd x23, 184(sp)
33     sd x24, 192(sp)
34     sd x25, 200(sp)
35     sd x26, 208(sp)
36     sd x27, 216(sp)
37     sd x28, 224(sp)
38     sd x29, 232(sp)
39     sd x30, 240(sp)
40     sd x31, 248(sp)
41
42     csrr a0, scause
43     csrr a1, sepc
44     call trap_handler
45
46     ld a0, 0(sp)
47     csrwrw sepc, a0
48     ld x1, 8(sp)
49     ld x3, 24(sp)
50     ld x4, 32(sp)
51     ld x5, 40(sp)
52     ld x6, 48(sp)
53     ld x7, 56(sp)
54     ld x8, 64(sp)
55     ld x9, 72(sp)
56     ld x10, 80(sp)
57     ld x11, 88(sp)
58     ld x12, 96(sp)
59     ld x13, 104(sp)
60     ld x14, 112(sp)
61     ld x15, 120(sp)
62     ld x16, 128(sp)

```

```

63     ld x17, 136(sp)
64     ld x18, 144(sp)
65     ld x19, 152(sp)
66     ld x20, 160(sp)
67     ld x21, 168(sp)
68     ld x22, 176(sp)
69     ld x23, 184(sp)
70     ld x24, 192(sp)
71     ld x25, 200(sp)
72     ld x26, 208(sp)
73     ld x27, 216(sp)
74     ld x28, 224(sp)
75     ld x29, 232(sp)
76     ld x30, 240(sp)
77     ld x31, 248(sp)
78     ld x2, 16(sp)
79
80
81     sret
82
83     # 1. save 32 registers and sepc to stack
84     # 2. call trap_handler
85     # 3. restore sepc and 32 registers (x2(sp) should be restore
        last) from stack
86     # 4. return from trap

```

在 `_traps` 中, 首先保存 32 个寄存器和 `sepc` 到栈中, 注意 `x2` 寄存器应该最先保存, 且栈的扩张是向下的, 每个寄存器为 8 字节, 所以栈指针应该减去 256, 依次保存 32 个寄存器和 `sepc`. 接着设置 `trap_handler` 的参数, 并调用 `trap_handler` 函数. 最后恢复 `sepc` 和 32 个寄存器, 注意 `x2` 寄存器应该最后恢复, 并返回, 注意需要调用 `sret` 指令, 以恢复中断使能和先前的特权级别.

实现 `trap_handler`

```

1 void trap_handler(uint64_t scause, uint64_t sepc) {
2

```

```

3  if ((scause >> 63) == 0) {
4      // Handle exceptions
5      switch (scause) {
6          case 0x0:
7              printk("Instruction address misaligned\n");
8              break;
9          case 0x1:
10             printk("Instruction access fault\n");
11             break;
12          case 0x2:
13             printk("Illegal instruction\n");
14             break;
15          case 0x3:
16             printk("Breakpoint\n");
17             break;
18          case 0x4:
19             printk("Load address misaligned\n");
20             break;
21          case 0x5:
22             printk("Load access fault\n");
23             break;
24          case 0x6:
25             printk("Store/AMO address misaligned\n");
26             break;
27          case 0x7:
28             printk("Store/AMO access fault\n");
29             break;
30          case 0x8:
31             printk("Environment call from U-mode\n");
32             break;
33          case 0x9:
34             printk("Environment call from S-mode\n");
35             break;
36          case 0xC:
37             printk("Instruction page fault\n");

```

```

38         break;
39     case 0xD:
40         printk("Load page fault\n");
41         break;
42     case 0xF:
43         printk("Store/AMO page fault\n");
44         break;
45     case 0x12:
46         printk("Software check\n");
47         break;
48     case 0x13:
49         printk("Hardware error\n");
50         break;
51     default:
52         printk("Unknown exception code: %p\n", scause);
53     }
54 } else {
55     // Handle interrupts
56     uint64_t interrupt_code = scause & (~(uint64_t)0x1<<63));
57     switch (interrupt_code) {
58     case 0x1:
59         printk("Supervisor software interrupt\n");
60         break;
61     case 0x5:
62         clock_set_next_event();
63         printk("Supervisor timer interrupt\n");
64         break;
65     case 0x9:
66         printk("Supervisor external interrupt\n");
67         break;
68     case 0xD:
69         printk("Counter-overflow interrupt\n");
70         break;
71     default:
72         printk("Unknown interrupt code: %p\n", interrupt_code);

```

```

73     }
74 }
75 printk("sepc: %p\n", sepc);
76 return;
77 }

```

在 `trap_handler` 中, 首先判断 `scause` 的最高位, 如果为 0, 则表示异常, 否则表示中断. 接着根据 `scause` 的值, 判断异常或中断的类型, 并输出相应的信息, 最后输出 `sepc` 的值.

2.3 实现时钟中断相关函数

```

1 #include "stdint.h"
2 #include "../include/sbi.h"
3
4 // QEMU 中时钟的频率是 10MHz, 也就是 1 秒钟相当于 100000000 个时钟周期
5 uint64_t TIMECLOCK = 100000000;
6
7 uint64_t get_cycles() {
8     // 编写内联汇编, 使用 rdttime 获取 time 寄存器中(也就是 mtime
9     // 寄存器)的值并返回
10    uint64_t ret;
11    __asm__ volatile("rdtime %0" : "=r"(ret));
12    return ret;
13 }
14
15 void clock_set_next_event() {
16     // 下一次时钟中断的时间点
17     uint64_t next = get_cycles() + TIMECLOCK;
18
19     // 使用 sbi_set_timer 来完成对下一次时钟中断的设置
20     sbi_set_timer(next);
21 }

```

在 `clock.c` 中, 定义了 `TIMECLOCK` 变量, 用于时钟中断的间隔时间. 定义了 `get_cycles` 函数, 用于获取当前时钟周期数. 定义了 `clock_set_next_event` 函数, 用于设置下一次

时钟中断的时间.

3 实验结果与分析

实验结果如下:

```
Boot HART ID           : 0
Boot HART Domain       : root
Boot HART Priv Version  : v1.12
Boot HART Base ISA     : rv64imafdc
Boot HART ISA Extensions : sstc,zicntr,zihpm,zicboz,zicbom,sdtrig,svadu
Boot HART PMP Count     : 16
Boot HART PMP Granularity : 2 bits
Boot HART PMP Address Bits: 54
Boot HART MHPM Info     : 16 (0x0007fff8)
Boot HART Debug Triggers : 2 triggers
Boot HART MIDELEG       : 0x0000000000001666
Boot HART MEDELEG       : 0x0000000000f0b509
2024 ZJU Operating System
kernel is running!
kernel is running!
kernel is running!
kernel is running!
Supervisor timer interrupt
sepc: 0x80200850
kernel is running!
kernel is running!
kernel is running!
kernel is running!
kernel is running!
Supervisor timer interrupt
sepc: 0x80200850
```

图 2: 实验结果

可以看到 printk 正常输出字符, 且每个一段时间触发时间中断, 符合实验预期结果.

4 遇到的问题及解决方法

问题 1: 在编写 entry.S 时, 没有注意到是 64 位的汇编代码, 误以为是 32 位的, 导致编写错误. 解决办法: 将 lw,sw 改为 ld,sd, 并栈上的储存间隔改为 8 字节.

问题 2: 在依照网站<https://wiki.qemu.org/Hosts/Linux>的指导, 在 Ubuntu 22.04 上安装 qemu, 但是执行 `../.././configure --enable-debug` 时, 出现错误 `qemu keycodemapdb has no meson.build file` 解决办法: 下载 git 仓库的子模块, 使用 `git submodule update --init` 命令, 再次执行 `../.././configure --enable-debug` 即可.

5 总结与心得

通过本次实验, 我学会了如何使用 makefile 和如何链接汇编代码和 C 代码, 同时也学会了如何使用 OpenSBI 接口来进行输出.

6 思考题

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

calling convention 总结:

参数传递与返回值: 使用 8 个整数参数寄存器 a0-a7 传递参数, 其中 a0 和 a1 也用于传递返回值, 特殊的参数或返回值 (过大/寄存器不足/...) 可以通过栈传递. 寄存器保存: 寄存器分为调用者保存 (Caller-Saved) 和被调用者保存 (Callee-Saved). 栈使用: 栈是从高地址向低地址生长的, 参数在栈上传递时, 依次存放在栈指针之上的更高地址处.

Caller / Callee Saved Register 区别:

Callee-Saved Registers: 在函数调用过程前后, 寄存器的值不会发生改变, 如子函数需要使用该寄存器, 则需要在栈上保存该寄存器的值, 并在函数结束后恢复. Caller-Saved Registers: 在函数调用过程中, 寄存器的值不保证不变, 如子函数需要使用该寄存器, 则不需要在栈上保存该寄存器的值, 而是直接使用, 如父函数需要在函数调用前后保持该寄存器的值, 则需要在栈上保存该寄存器的值, 并在函数结束后恢复.

6.2 编译之后, 通过 System.map 查看 vmlinux.lds 中自定义符号的值并截图.

```
0000000080200000 A BASE_ADDR
0000000000000002 a SIE|
0000000000000020 a STIE
0000000080203000 D TIMECLOCK
0000000080203008 d _GLOBAL_OFFSET_TABLE_
0000000080205000 B _ebss
0000000080203008 D _edata
0000000080205000 B _kernel
0000000080202321 R _erodata
00000000802017a0 T _etext
0000000080204000 B _sbss
0000000080203000 D _sdata
0000000080200000 T _skernel
0000000080202000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
000000008020004c T _traps
0000000080204000 B boot_stack
0000000080205000 B boot_stack_top
```

图 3: System.map 查看 vmlinux.lds 中自定义标签的值

```

0000000080200190 T clock_set_next_event
0000000080200168 T get_cycles
00000000802008d4 T isspace
0000000080202310 r lowerxdigits.0
0000000080200c28 t print_dec_int
0000000080201720 T printk
000000008020088c T putc
0000000080200ba0 t puts_wo_nl
0000000080200480 T sbi_debug_console_read
00000000802003dc T sbi_debug_console_write
0000000080200524 T sbi_debug_console_write_byte
00000000802001d8 T sbi_ecall
0000000080200294 T sbi_set_timer
0000000080200330 T sbi_system_reset
00000000802007f8 T start_kernel
0000000080200934 T strtol
000000008020083c T test
00000000802005c4 T trap_handler
00000000802022f8 r upperxdigits.1
0000000080200f30 T vprintfmt

```

图 4: System.map 查看 vmlinux.lds 中自定义函数的位置

6.3 用 csr_read 宏读取 sstatus 寄存器的值, 对照 RISC-V 手册解释其含义并截图.

在 main 函数中加入如下代码:

```

1   printk("sstatus: 0x%lx\n", csr_read(sstatus));

```

得到输出:

```
Boot HART MIDELEG      : 0x0000000000001666
Boot HART MEDELEG      : 0x0000000000f0b509
2024 ZJU Operating System
sstatus: 0x8000000200006002
kernel is running!
kernel is running!
kernel is running!
```

图 5: 读取 sstatus 寄存器的值

由输出知 sstatus 值为 0x8000000200006002, 换算为二进制如下表所示

位位置	字段名	位宽	采样值	说明
63	SD	1	1	浮点扩展状态寄存器或向量扩展状态寄存器或者用户模式扩展状态寄存器指示为脏状态
62-34	WPRI	29	0	保留位
33-32	UXL	2	2	用户模式下 XLEN 的值为 64
31-20	WPRI	12	0	保留位
19	MXR	1	0	不允许加载读取可执行页面的数据
18	SUM	1	0	不允许在 S 模式下访问用户模式的页面
17	WPRI	1	0	保留位
16-15	XS	2	0	无用户扩展开启
14-13	FS	2	3	浮点扩展有脏位
12-11	WPRI	2	0	保留位
10-9	VS	2	0	向量扩展初始化
8	SPP	1	0	上一个特权级别不为 S 模式 (刚从 m 模式退出)
7	WPRI	1	0	保留位
6	UBE	1	0	用户模式为小端
5	SPIE	1	0	先前的 S 模式中断被启用
4-2	WPRI	3	0	保留位
1	SIE	1	1	S 模式中断被启用
0	WPRI	1	0	保留位

表 2: RISC-V sstatus 寄存器字段说明

6.4 用 csr_write 宏向 sscratch 寄存器写入数据, 并验证是否写入成功并截图.

在 main 函数中加入如下代码:

```
1  printk("sstatus: 0x%lx\n", csr_read(sstatus));
2  printk("previous sscratch: 0x%lx\n", csr_read(sscratch));
3  csr_write(sscratch, 0x1234567890abcdef);
4  printk("current sscratch: 0x%lx\n", csr_read(sscratch));
```

结果如下图:

```
Boot HART MIDELEG      : 0x0000000000001666
Boot HART MEDELEG      : 0x0000000000f0b509
2024 ZJU Operating System
sstatus: 0x8000000200006002
previous sscratch: 0x0
current sscratch: 0x1234567890abcdef
```

图 6: 写入 sscratch 寄存器的值

6.5 详细描述你可以通过什么步骤来得到
arch/arm64/kernel/sys.i, 给出过程以及截图.

进入 linux 源文件文件夹, 依次在终端输入如下命令:

```
1 apt install aarch64-linux-gnu-gcc
2 make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
3 make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu-
  arch/arm64/kernel/sys.i
4 ls arch/arm64/kernel | grep sys.i
```

首先安装 aarch64-linux-gnu-gcc 交叉编译器, 然后配置 linux 编译参数, 生成 sys.i 文件, 最后查看生成的 sys.i 文件. 整个过程截图如下:

```

~/course/OSZJU/linux-6.10.9 3s Py base 20:12:48
● > sudo apt install gcc-aarch64-linux-gnu
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following packages were automatically installed and are no longer
required:
  cpu-checker golang-1.18 golang-1.18-doc golang-1.18-go
  golang-1.18-src golang-src ipxe-qemu
  ipxe-qemu-256k-compat-efi-roms libcacard0 libgfat0 libgfrpc0
  libgfxdr0 libglusterfs0 libnspr4 libnss3 libpcsc-lite1 libslirp0
  libspice-server1 liburing2 libusbredirparser1 libvirglrenderer1
  msr-tools ovmf qemu-block-extra qemu-system-common
  qemu-system-data qemu-system-gui qemu-system-x86 qemu-utils
  seabios
Use 'sudo apt autoremove' to remove them.
The following additional packages will be installed:
  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

```

图 7: 安装 aarch64-linux-gnu-gcc 交叉编译器

```

~/course/OSZJU/linux-6.10.9 13s Py base 20:13:22
● > make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#

```

图 8: 配置 linux 编译参数

```

~/course/OSZJU/linux-6.10.9 Py base 20:13:44
● > make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/s
ys.i
HOSTCC scripts/dtc/dtc.o
HOSTCC scripts/dtc/flattree.o
HOSTCC scripts/dtc/fstree.o
HOSTCC scripts/dtc/data.o
HOSTCC scripts/dtc/livetree.o
HOSTCC scripts/dtc/treesource.o
HOSTCC scripts/dtc/srcpos.o
HOSTCC scripts/dtc/checks.o
HOSTCC scripts/dtc/util.o
LEX scripts/dtc/dtc-lexer.lex.c
YACC scripts/dtc/dtc-parser.tab.[ch]

```

图 9: 生成 sys.i 文件

```
~/course/OSZJU/linux-6.10.9 5s Py base 20:15:50
> ls arch/arm64/kernel | grep sys.i
sys.i
```

图 10: 查看生成的 sys.i 文件

6.6 寻找 Linux v6.0 中 ARM32 RV32 RV64 x86_64 架构的系统调用表; 请列出源代码文件, 展示完整的系统调用表 (宏展开后), 每一步都需要截图.

首先安装编译器

```
1 sudo apt install gcc # x86_64
2 sudo apt install gcc-arm-linux-gnueabi # ARM32
3 sudo apt install gcc-riscv64-linux-gnu # RV64 RV32
```

截图如下:

```
~/course/OSZJU/os24fall-stu main !7 78 10:09:59
> sudo apt install gcc # x86_64
sudo apt install gcc-arm-linux-gnueabi # ARM32
sudo apt install gcc-riscv64-linux-gnu # RV64 RV32
[sudo] password for morretti:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
gcc is already the newest version (4:11.2.0-1ubuntu1).
The following packages were automatically installed and are no longer required:
  cpu-checker golang-1.18 golang-1.18-doc golang-1.18-go golang-1.18-src golang-src ipxe-qemu
  ipxe-qemu-256k-compat-efi-roms libcacard0 libgfapi0 libgfrpc0 libgfxdr0 libglusterfs0 libnspr4 libnss3 libpcsc-lite1
  libslirp0 libspice-server1 liburing2 libusbredirparser1 libvirglrenderer1 msr-tools ovmf qemu-block-extra
  qemu-system-common qemu-system-data qemu-system-gui qemu-system-x86 qemu-utils seabios
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
gcc-arm-linux-gnueabi is already the newest version (4:11.2.0-1ubuntu1).
The following packages were automatically installed and are no longer required:
  cpu-checker golang-1.18 golang-1.18-doc golang-1.18-go golang-1.18-src golang-src ipxe-qemu
  ipxe-qemu-256k-compat-efi-roms libcacard0 libgfapi0 libgfrpc0 libgfxdr0 libglusterfs0 libnspr4 libnss3 libpcsc-lite1
  libslirp0 libspice-server1 liburing2 libusbredirparser1 libvirglrenderer1 msr-tools ovmf qemu-block-extra
  qemu-system-common qemu-system-data qemu-system-gui qemu-system-x86 qemu-utils seabios
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
gcc-riscv64-linux-gnu is already the newest version (4:11.2.0-1ubuntu1).
The following packages were automatically installed and are no longer required:
  cpu-checker golang-1.18 golang-1.18-doc golang-1.18-go golang-1.18-src golang-src ipxe-qemu
  ipxe-qemu-256k-compat-efi-roms libcacard0 libgfapi0 libgfrpc0 libgfxdr0 libglusterfs0 libnspr4 libnss3 libpcsc-lite1
  libslirp0 libspice-server1 liburing2 libusbredirparser1 libvirglrenderer1 msr-tools ovmf qemu-block-extra
  qemu-system-common qemu-system-data qemu-system-gui qemu-system-x86 qemu-utils seabios
Use 'sudo apt autoremove' to remove them.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

图 11: 安装编译器

几个架构的系统调用表文件如下:

架构	系统调用表文件
x86_64	arch/x86/entry/syscalls/syscall_64.tbl
ARM32	arch/arm/tools/syscall.tbl
RV32	arch/riscv/kernel/syscall_table.c
RV64	arch/riscv/kernel/syscall_table.c

表 3: 系统调用表文件

查看 x86_64 架构的系统调用表, 如下图:

```

syscall_64.tbl x syscall.tbl C syscall_table.c 5 C syscall ...
home > morretti > course > OSZJU > linux-6.10.9 > arch > x86 > entry > syscalls > sys
25 #
24 # 64-bit system call numbers and entry vectors
23 #
22 # The format is:
21 # <number> <abi> <name> <entry point> [<compat entry point>]
20 #
19 # The __x64_sys_*() stubs are created on-the-fly for sys_*(
18 #
17 # The abi is "common", "64" or "x32" for this file.
16 #
15 0 common read sys_read
14 1 common write sys_write
13 2 common open sys_open
12 3 common close sys_close
11 4 common stat sys_newstat
10 5 common fstat sys_newfstat
9 6 common lstat sys_newlstat
8 7 common poll sys_poll
7 8 common lseek sys_lseek
6 9 common mmap sys_mmap
5 10 common mprotect sys_mprotect
4 11 common munmap sys_munmap
3 12 common brk sys_brk
2 13 64 rt_sigaction sys_rt_sigaction
1 14 common rt_sigprocmask sys_rt_sigprocmask
26 15 64 rt_sigreturn sys_rt_sigreturn
1 16 64 ioctl sys_ioctl
2 17 common pread64 sys_pread64
3 18 common pwrite64 sys_pwrite64
4 19 64 readv sys_readv
5 20 64 writev sys_writev
6 21 common access sys_access
7 22 common pipe sys_pipe
8 23 common select sys_select
9 24 common sched_yield sys_sched_yield
10 25 common mremap sys_mremap
11 26 common msync sys_msync
12 27 common mincore sys_mincore
13 28 common madvise sys_madvise
14 29 common shmget sys_shmget
15 30 common shmat sys_shmat
16 31 common shmctl sys_shmctl
17 32 common dup sys_dup
18 33 common dup2 sys_dup2
19 34 common pause sys_pause
20 35 common nanosleep sys_nanosleep
21 36 common getitimer sys_getitimer
22 37 common alarm sys_alarm
23 38 common setitimer sys_setitimer
24 39 common getpid sys_getpid

```

图 12: x86_64 架构的系统调用表

查看 ARM32 架构的系统调用表, 如下图:

```

1  # Linux system call numbers and entry vectors
2  #
3  # The format is:
4  # <num> <abi>  <name>          [<entry point>          [<oabi name>
5  #
6  # Where abi is:
7  # common - for system calls shared between oabi and eabi (may have compat)
8  # oabi   - for oabi-only system calls (may have compat)
9  # eabi   - for eabi-only system calls
10 #
11 # For each syscall number, "common" is mutually exclusive with "oabi" and "eabi"
12 #
13 0  common restart_syscall    sys_restart_syscall
14 1  common exit              sys_exit
15 2  common fork             sys_fork
16 3  common read             sys_read
17 4  common write            sys_write
18 5  common open             sys_open
19 6  common close            sys_close
20 # 7 was sys_waitpid
21 8  common creat            sys_creat
22 9  common link             sys_link
23 10 common unlink           sys_unlink
24 11 common execve           sys_execve
25 12 common chdir            sys_chdir
26 13 oabi  time              sys_time32
27 14 common mknod            sys_mknod
28 15 common chmod            sys_chmod
29 16 common lchown           sys_lchown16
30 # 17 was sys_break
31 # 18 was sys_stat
32 19 common lseek            sys_lseek
33 20 common getpid           sys_getpid
34 21 common mount            sys_mount
35 22 oabi  umount            sys_oldumount
36 23 common setuid           sys_setuid16
37 24 common getuid           sys_getuid16
38 25 oabi  stime              sys_stime32
39 26 common ptrace           sys_ptrace
40 27 oabi  alarm              sys_alarm
41 # 28 was sys_fstat
42 29 common pause            sys_pause
43 30 oabi  utime              sys_utime32
44 # 31 was sys_stty
45 # 32 was sys_gtty
46 33 common access           sys_access
47 34 common nice              sys_nice

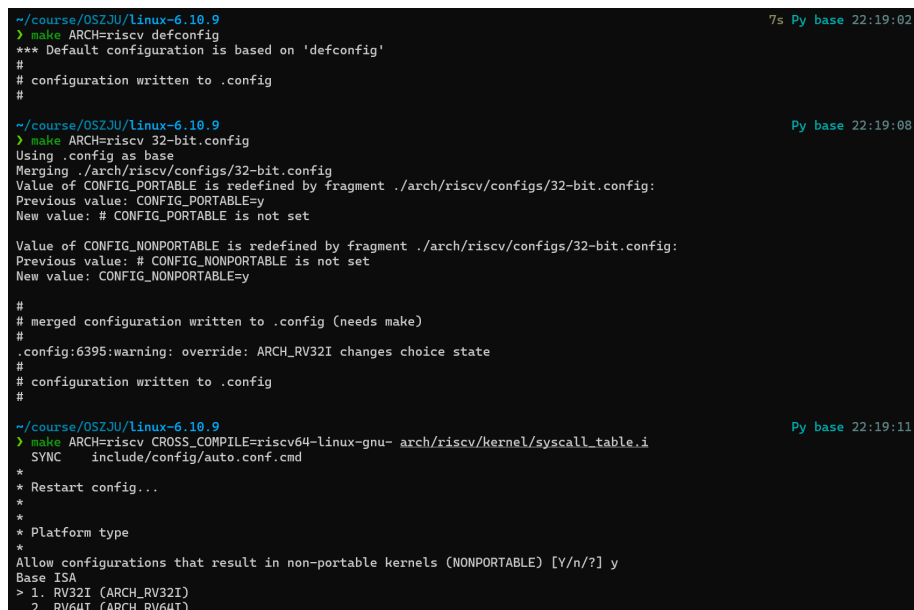
```

图 13: ARM32 架构的系统调用表

RV32 和 RV64 架构的系统调用表需要交叉编译, 去除宏定义.
首先获取 RV32 的系统调用表, 执行如下命令:

```
1 make ARCH=riscv defconfig
2 make ARCH=riscv 32-bit.config
3 make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-
  arch/riscv/kernel/syscall_table.i
4 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-
  arch/arm/tools/syscall.tbl
```

截图如下:



```
~/course/OSZJU/Linux-6.10.9 7s Py base 22:19:02
> make ARCH=riscv defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#

~/course/OSZJU/Linux-6.10.9 Py base 22:19:08
> make ARCH=riscv 32-bit.config
Using .config as base
Merging ./arch/riscv/configs/32-bit.config
Value of CONFIG_PORTABLE is redefined by fragment ./arch/riscv/configs/32-bit.config:
Previous value: CONFIG_PORTABLE=y
New value: # CONFIG_PORTABLE is not set
Value of CONFIG_NONPORTABLE is redefined by fragment ./arch/riscv/configs/32-bit.config:
Previous value: # CONFIG_NONPORTABLE is not set
New value: CONFIG_NONPORTABLE=y
#
# merged configuration written to .config (needs make)
#
.config:6395:warning: override: ARCH_RV32I changes choice state
#
# configuration written to .config
#

~/course/OSZJU/Linux-6.10.9 Py base 22:19:11
> make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i
SYNC include/config/auto.conf.cmd
*
* Restart config...
*
* Platform type
*
Allow configurations that result in non-portable kernels (NONPORTABLE) [Y/n/?] y
Base ISA
> 1. RV32I (ARCH_RV32I)
> 2. RV64I (ARCH_RV64I)
```

图 14: 预编译 RV32 的系统调用表

然后查看生成的 syscall_table.i 文件, 如下图:

```

67342 void * const sys_call_table[463] = {
67343 | [0 ... 463 - 1] = __riscv_sys_ni_syscall,
67344 # 1 "./arch/riscv/include/asm/unistd.h" 1
67345 # 24 "./arch/riscv/include/asm/unistd.h"
67346 # 1 "./arch/riscv/include/uapi/asm/unistd.h" 1
67347 # 26 "./arch/riscv/include/uapi/asm/unistd.h"
67348 # 1 "./include/uapi/asm-generic/unistd.h" 1
67349 # 34 "./include/uapi/asm-generic/unistd.h"
67350 [0] = __riscv_sys_io_setup,
67351
67352 [1] = __riscv_sys_io_destroy,
67353
67354 [2] = __riscv_sys_io_submit,
67355
67356 [3] = __riscv_sys_io_cancel,
67357
67358
67359
67360
67361
67362
67363
67364 [5] = __riscv_sys_setxattr,
67365
67366 [6] = __riscv_sys_lsetxattr,
67367
67368 [7] = __riscv_sys_fsetxattr,
67369
67370 [8] = __riscv_sys_getxattr,
67371
67372 [9] = __riscv_sys_lgetxattr,
67373
67374 [10] = __riscv_sys_fgetxattr,
67375
67376 [11] = __riscv_sys_listxattr,
67377
67378 [12] = __riscv_sys_llistxattr,
67379
67380 [13] = __riscv_sys_flistxattr,
67381
67382 [14] = __riscv_sys_removexattr,
67383
67384 [15] = __riscv_sys_lremovexattr,
67385
67386 [16] = __riscv_sys_fremovexattr,

```

图 15: RV32 架构的系统调用表

然后获取 RV64 的系统调用表, 执行如下命令:

```

1 make ARCH=riscv defconfig
2 make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-
  arch/riscv/kernel/syscall_table.i

```

截图如下:

```

~/course/OS2.0/Linux-6.10.9
> make ARCH=riscv defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#

~/course/OS2.0/Linux-6.10.9
> make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i
SYMC include/config/auto.conf.cmd
*
* Restart config...
*
*
* Platform type
*
Allow configurations that result in non-portable kernels (NONPORTABLE) [N/y/?] n
Base ISA
> 1. RV64I (ARCH_RV64I)
choice[1]: 1
Hermel Code Model
1. medium low code model (CMODEL_MEDLOW)
> 2. medium any code model (CMODEL_MEDANY)
choice[1-2]: 2
Symmetric Multi-Processing (SMP) [Y/n/?] y
Multi-core scheduler support (SCHED_MC) [N/y/?] n
Maximum number of CPUs (2-512) (NR_CPUS) [64] 64
Support for hot-pluggable CPUs (HOTPLUG_CPU) [Y/?] y
cpu:uninon

```

图 16: 预编译 RV64 的系统调用表

然后查看生成的 syscall_table.i 文件, 如下图:

```

767 void * const sys_call_table[463] = {
766 | [0 ... 463 - 1] = __riscv_sys_ni_syscall,
765 # 1 "./arch/riscv/include/asm/unistd.h" 1
764 # 24 "./arch/riscv/include/asm/unistd.h"
763 # 1 "./arch/riscv/include/uapi/asm/unistd.h" 1
762 # 26 "./arch/riscv/include/uapi/asm/unistd.h"
761 # 1 "./include/uapi/asm-generic/unistd.h" 1
760 # 34 "./include/uapi/asm-generic/unistd.h"
759 [0] = __riscv_sys_io_setup,
758
757 [1] = __riscv_sys_io_destroy,
756
755 [2] = __riscv_sys_io_submit,
754
753 [3] = __riscv_sys_io_cancel,
752
751
750
749 [4] = __riscv_sys_io_getevents,
748
747
746
745 [5] = __riscv_sys_setxattr,
744
743 [6] = __riscv_sys_lsetxattr,
742
741 [7] = __riscv_sys_fsetxattr,
740
739 [8] = __riscv_sys_getxattr,
738
737 [9] = __riscv_sys_lgetxattr,
736
735 [10] = __riscv_sys_fgetxattr,

```

图 17: RV64 架构的系统调用表

6.7 阐述什么是 ELF 文件? 尝试使用 readelf 和 objdump 来查看 ELF 文件, 并给出解释和截图. 运行一个 ELF 文件, 然后通过 cat /proc/PID/maps 来给出其内存布局并截图.

ELF(Executable and Linkable Format) 文件是一种标准的文件格式, 用于存储可执行文件, 目标代码, 共享库等文件的二进制格式. ELF 文件包含了程序的代码, 数据, 符号表, 动态链接信息等. ELF 主要包括 ELF 头, 程序头表, 节头表, 数据区等部分. 文件头是 ELF 文件的起始部分, 它提供了关于整个文件的元数据. 该部分定义了 ELF 文件的类型, 结构, ELF Header 大小, 程序头表偏移, 段头表偏移, 入口点地址等元信息. 程序头表内每个条目描述了一个段的信息, 包括段的类型, 文件偏移, 段的文件大小, 段的内存大小, 段的内存加载等, 一个段包含多个节. 节头表内每个条目描述了一个节的信息, 包括节的名称, 节的类型, 节的偏移, 节的大小等. 数据区包含了程序的代码, 数据, 符号表, 动态链接信息等即节的内容. 一般程序头表用于加载, 节头表用于链接. 运行 `readelf -e vmlinux` 和 `riscv64-linux-gnu-objdump -d vmlinux` 查看 ELF 文件, 如下图:

```
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:             33032 (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
```

图 18: readelf 查看文件头


```

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz             MemSiz             Flags   Align
RISCV_ATTRIBUT 0x0000000000007231 0x0000000000000000 0x0000000000000000
                 0x0000000000000032 0x0000000000000000  R       0x1
LOAD            0x0000000000000100 0x0000000000002000 0x0000000000002000
                 0x00000000000002379 0x00000000000002379  R E     0x1000
LOAD            0x0000000000000400 0x0000000000002030 0x0000000000002030
                 0x0000000000000038 0x0000000000000200  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

```

图 19: readelf 查看程序头表

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
	000000000000180c	0000000000000000	AX 0 0	4096
[2]	.rodata	PROGBITS	0000000080202000	00003000
	0000000000000379	0000000000000000	A 0 0	4096
[3]	.data	PROGBITS	0000000080203000	00004000
	0000000000000008	0000000000000000	WA 0 0	4096
[4]	.got	PROGBITS	0000000080203008	00004008
	0000000000000020	0000000000000008	WA 0 0	8
[5]	.got.plt	PROGBITS	0000000080203028	00004028
	0000000000000010	0000000000000008	WA 0 0	8
[6]	.bss	NOBITS	0000000080204000	00004038
	0000000000001000	0000000000000000	WA 0 0	4096
[7]	.debug_info	PROGBITS	0000000000000000	00004038
	0000000000000de2	0000000000000000	0 0	1
[8]	.debug_abbrev	PROGBITS	0000000000000000	00004e1a
	00000000000005b6	0000000000000000	0 0	1
[9]	.debug_aranges	PROGBITS	0000000000000000	000053d0
	0000000000000240	0000000000000000	0 0	16
[10]	.debug_rnglists	PROGBITS	0000000000000000	00005610
	00000000000001a3	0000000000000000	0 0	1
[11]	.debug_line	PROGBITS	0000000000000000	000057b3
	0000000000001502	0000000000000000	0 0	1
[12]	.debug_str	PROGBITS	0000000000000000	00006cb5
	00000000000003b2	0000000000000001	MS 0 0	1
[13]	.debug_line_str	PROGBITS	0000000000000000	00007067
	000000000000019f	0000000000000001	MS 0 0	1
[14]	.comment	PROGBITS	0000000000000000	00007206
	000000000000002b	0000000000000001	MS 0 0	1
[15]	.riscv.attributes	RISCV_ATTRIBUTE	0000000000000000	00007231
	0000000000000032	0000000000000000	0 0	1
[16]	.debug_frame	PROGBITS	0000000000000000	00007268
	00000000000003e8	0000000000000000	0 0	8
[17]	.symtab	SYMTAB	0000000000000000	00007650
	00000000000007f8	0000000000000018	18 53	8
[18]	.strtab	STRTAB	0000000000000000	00007e48
	00000000000001f6	0000000000000000	0 0	1
[19]	.shstrtab	STRTAB	0000000000000000	0000803e
	00000000000000ca	0000000000000000	0 0	1

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)

图 20: readelf 查看节头表

```

src > lab1 > 1.txt
1584
1583   vmlinux:      file format elf64-littleriscv
1582
1581
1580   Disassembly of section .text:
1579
1578   0000000080200000 <_skernel>:
1577       80200000: 00003117      auipc  sp,0x3
1576       80200004: 01813103      ld     sp,24(sp) # 80203018 <_GLOBAL_OFFSET_TABLE_+0x10>
1575       80200008: 00003517      auipc  a0,0x3
1574       8020000c: 01853503      ld     a0,24(a0) # 80203020 <_GLOBAL_OFFSET_TABLE_+0x18>
1573       80200010: 10551073      csrw   stvec,a0
1572       80200014: 02000513      li     a0,32
1571       80200018: 10452073      csrs   sie,a0
1570       8020001c: c0102573      rdtime a0
1569       80200020: 00003597      auipc  a1,0x3
1568       80200024: ff05b583      ld     a1,-16(a1) # 80203010 <_GLOBAL_OFFSET_TABLE_+0x8>
1567       80200028: 0000b583      ld     a1,0(a1)
1566       8020002c: 00b50533      add    a0,a0,a1
1565       80200030: 00000813      li     a6,0
1564       80200034: 544958b7      lui    a7,0x54495
1563       80200038: d458889b      addiw  a7,a7,-699 # 54494d45 <STIE+0x54494d25>
1562       8020003c: 00000073      ecall
1561       80200040: 00200513      li     a0,2
1560       80200044: 10052073      csrs   sstatus,a0
1559       80200048: 7b0000ef      jal    ra,802007f8 <start_kernel>
1558
1557   000000008020004c <_traps>:
1556       8020004c: f0213823      sd     sp,-240(sp)
1555       80200050: f0010113      addi   sp,sp,-256
1554       80200054: 00113423      sd     ra,8(sp)
1553       80200058: 141020f3      csrr   ra,sepc
1552       8020005c: 00113023      sd     ra,0(sp)
1551       80200060: 00313c23      sd     gp,24(sp)
1550       80200064: 02413023      sd     tp,32(sp)
1549       80200068: 02513423      sd     t0,40(sp)
1548       8020006c: 02613823      sd     t1,48(sp)
1547       80200070: 02713c23      sd     t2,56(sp)
1546       80200074: 04813023      sd     s0,64(sp)
1545       80200078: 04913423      sd     s1,72(sp)
1544       8020007c: 04a13823      sd     a0,80(sp)
1543       80200080: 04b13c23      sd     a1,88(sp)
1542       80200084: 06c13023      sd     a2,96(sp)
1541       80200088: 06d13423      sd     a3,104(sp)
1540       8020008c: 06e13823      sd     a4,112(sp)
1539       80200090: 06f13c23      sd     a5,120(sp)
1538       80200094: 09013023      sd     a6,128(sp)
1537       80200098: 09113423      sd     a7,136(sp)
1536       8020009c: 09213823      sd     s2,144(sp)
1535       802000a0: 09313c23      sd     s3,152(sp)
1534       802000a4: 0b413023      sd     s4,160(sp)

```

图 21: objdump 查看数据区

接下来我编写了一个简单的 c 文件

```

1 int main()
2 {
3     unsigned int i=0;
4     while(i>=0)
5     {
6         i++;
7     }
8 }

```

用 gcc 编译为名为 a.out 的 ELF 可执行文件, 然后在一个终端运行该文件, 在另一个终端输入如下命令:

```
1 ps aux | grep a.out //查找PID
2 cat /proc/23049/maps
```

其中 23049 为 a.out 的 PID, 得到的内存布局如下:

```
~/course/OS2JU/os24fall-stu lab1 P1 x INT Py base 16:28:23
> ps aux | grep a.out
morretti 23049 98.1 0.0 2648 936 pts/15 R+ 16:28 0:38 ./a.out
morretti 23296 0.0 0.0 4836 2104 pts/2 S+ 16:28 0:00 grep --color=auto --exclude-dir=.bzr --exclude-dir=CVS --exclude-dir=.git --exclude-dir=.hg --excl
ude-dir=.svn --exclude-dir=.idea --exclude-dir=.tox a.out

~/course/OS2JU/os24fall-stu lab1 P1 Py base 16:28:43
> cat /proc/23049/maps
55579a4f000-55579a4f000 r--p 00000000 08:20 545521 /home/morretti/course/OS2JU/os24fall-stu/src/lab1/a.out
55579a4f000-55579a4f1000 r-xp 00001000 08:20 545521 /home/morretti/course/OS2JU/os24fall-stu/src/lab1/a.out
55579a4f1000-55579a4f2000 r--p 00002000 08:20 545521 /home/morretti/course/OS2JU/os24fall-stu/src/lab1/a.out
55579a4f2000-55579a4f3000 r--p 00003000 08:20 545521 /home/morretti/course/OS2JU/os24fall-stu/src/lab1/a.out
55579a4f3000-55579a4f4000 r-wp 00004000 08:20 545521 /home/morretti/course/OS2JU/os24fall-stu/src/lab1/a.out
7f1f15002000-7f1f15005000 rw-p 00000000 00:00 0
7f1f15005000-7f1f1502d000 r--p 00000000 08:20 31675 /usr/lib/x86_64-linux-gnu/libc.so.6
7f1f1502d000-7f1f151c2000 r-xp 00028000 08:20 31675 /usr/lib/x86_64-linux-gnu/libc.so.6
7f1f151c2000-7f1f1521a000 r--p 001bd000 08:20 31675 /usr/lib/x86_64-linux-gnu/libc.so.6
7f1f1521a000-7f1f1521b000 ---p 00215000 08:20 31675 /usr/lib/x86_64-linux-gnu/libc.so.6
7f1f1521b000-7f1f1521f000 r--p 00215000 08:20 31675 /usr/lib/x86_64-linux-gnu/libc.so.6
7f1f1521f000-7f1f15221000 rw-p 00219000 08:20 31675 /usr/lib/x86_64-linux-gnu/libc.so.6
7f1f15221000-7f1f1522a000 rw-p 00000000 00:00 0
7f1f1522a000-7f1f1523d000 rw-p 00000000 00:00 0
7f1f1523d000-7f1f1523f000 r--p 00000000 08:20 31669 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f1f1523f000-7f1f15239000 r-xp 00002000 08:20 31669 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f1f15239000-7f1f15274000 r--p 0002c000 08:20 31669 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f1f15274000-7f1f15277000 r--p 00037000 08:20 31669 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f1f15277000-7f1f15279000 rw-p 00039000 08:20 31669 /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffdcb098000-7ffdcb0ba000 rw-p 00000000 00:00 0 [stack]
7ffdcb0c7000-7ffdcb0cb000 r--p 00000000 00:00 0 [vvar]
7ffdcb0cb000-7ffdcb0cd000 r-xp 00000000 00:00 0 [vdso]
```

图 22: a.out 的内存布局

6.8 在我们使用 make run 时, OpenSBI 会产生如下输出. 通过查看 RISC-V Privileged Spec 中的 medeleg 和 mideleg 部分, 解释上面 MIDELEG 和 MEDELEG 值的含义.

输出如下:

OpenSBI v1.5.1

```

-----
/  __ \      /  ____|  _ \   _ |
| | | | | _ _ _ _ _ _ _ _ _ | ( _ _ | | ) | | | | | |
| | | | | ' \ / _ \ ' \ \ _ _ \ |  _ < | |
| | _ | | | ) | _ _ / | | | _ _ ) | | ) | | |
\ _ _ / | . _ / \ _ _ | | | | _ _ / | _ _ / _ _ |
  | |
  | _ |
.....
```

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

.....

回答:medeleg 寄存器用于控制哪些异常可以被委托到 S 模式,mideleg 寄存器用于控制哪些中断可以被委托给 s 模式处理, 每位对应的异常为相应 Exception Code 的异常. 当对应位未置为 1 时, 异常或中断将默认情况下给 M 模式处理. 根据输出,MEDELEG 的值为 0x000000000000b109, 对应的二进制为... 0000 0000 0000 0000 0000 1011 0000 1001, 根据 RISC-V Privileged Spec 中的 medeleg 部分, 委托给 S 模式处理的异常有:Instruction address misaligned, Breakpoint, Environment call from U-mode, Environment call from S-mode, Environment call from M-mode; 根据输出,MIDELEG 的值为 0x0000000000000222, 对应的二进制为 0000 0000 0000 0000 0010 0010 0010, 根据 RISC-V Privileged Spec 中的 mideleg 部分, 委托给 S 模式处理的中断有:Supervisor software interrupt, Supervisor timer interrupt, Supervisor external interrupt.

Table 14. Machine cause register (*mcause*) values after trap.

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	<i>Reserved</i>
1	13	Counter-overflow interrupt
1	14-15	<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	16-17	<i>Reserved</i>
0	18	Software check
0	19	Hardware error
0	20-23	<i>Reserved</i>
0	24-31	<i>Designated for custom use</i>
	32-47	<i>Reserved</i>
	48-63	<i>Designated for custom use</i>
	≥ 64	<i>Reserved</i>

图 23: RISC-V 异常码表