浙江大學

课程名称：	操作系统原理与实践
题　　目：	Lab 1：RV64 内核引导与时钟中断处理
授课教师：	申文博
助　　教：	王鹤翔、陈淦豪、许昊瑞
姓　　名：	潘潇然
学　　号：	3220106049
地　　点：	32舍367

一、实验过程与步骤

RV64内核引导

1. 完善Makefile脚本，补充 `lib/Makefile` 。这里我们直接使用 `init` 目录下的Makefile。接下来将对Makefile的内容做出解释

- `$(wildcard *.c)`：获取当前目录下所有 `.c` 文件
- `$(sort ...)`：对传入文件列表按字母序排列，并去除重复项。因此第一行获取当前目录下所有 `.c` 文件并排序
- `$(patsubst %.c,%.o,$(C_SRC))`：将上述的 `.c` 文件名转成 `.o` 文件名，即我们需要生成的目标
- `${GCC}`：以下内容在根目录的Makefile可以找到对应定义，指代 `riscv64-linux-gnu-gcc`
- `CFLAG = ${CF} ${INCLUDE}`
  - 其中 `INCLUDE = -I $(shell pwd)/include -I $(shell pwd)/arch/riscv/include`，将当前目录下两个指定路径的文件作为头文件
  - `CF = -march=$(ISA) -mabi=$(ABI) -mcmodel=medany -fno-builtin -ffunction-sections -fdata-sections -nostartfiles -nostdlib -nostdinc -static -lgcc -Wl,--nmagic -Wl,--gc-sections -g`，包含了一系列编译选项，包括指定目标架构，指定应用二进制接口，禁用内置函数，不使用标准启动文件、标准库、头文件路径，使用静态链接，生成调试信息等等

综上，以上Makefile获取当前目录所有 `.c` 文件进行编译，因此在后续过程中即使增删文件，也不需要对Makefile进行修改

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

all:$(OBJ)

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

2. 编写 `head.S`

- 将 `.space` 设为4096，即4KB
- 之后将栈指针指向 `boot_stack_top`，并跳转到 `start_kernel`

```
    .extern start_kernel
    .section .text.init
    .globl _start
_start:

    la a0, boot_stack_top
    mv sp, a0
    jal start_kernel

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

    .globl boot_stack_top
boot_stack_top:
```

3. 补充 `sbi.c`，在此部分补充完成了 `sbi_ecall`，`sbi_set_timer`，`sbi_debug_console_write_byte`，`sbi_system_reset` 这四个函数

- `sbi_ecall`：这里使用内联汇编，依次存储 `eid`，`fid`，`arg[0~5]` 到 `a0~a7`，之后调用 `ecall` 进入M模式，让OpenSBI完成相关操作。之后从 `a0`，`a1` 取出 `error code` 和 `value` 作为函数的返回结果
    - 其中 `%0` 表示输入输出操作数部分的第1个，从输出开始计算，其余同理
- 其他函数：直接根据不同的Extension ID、Function ID和输入调用 `sbi_ecall` 即可

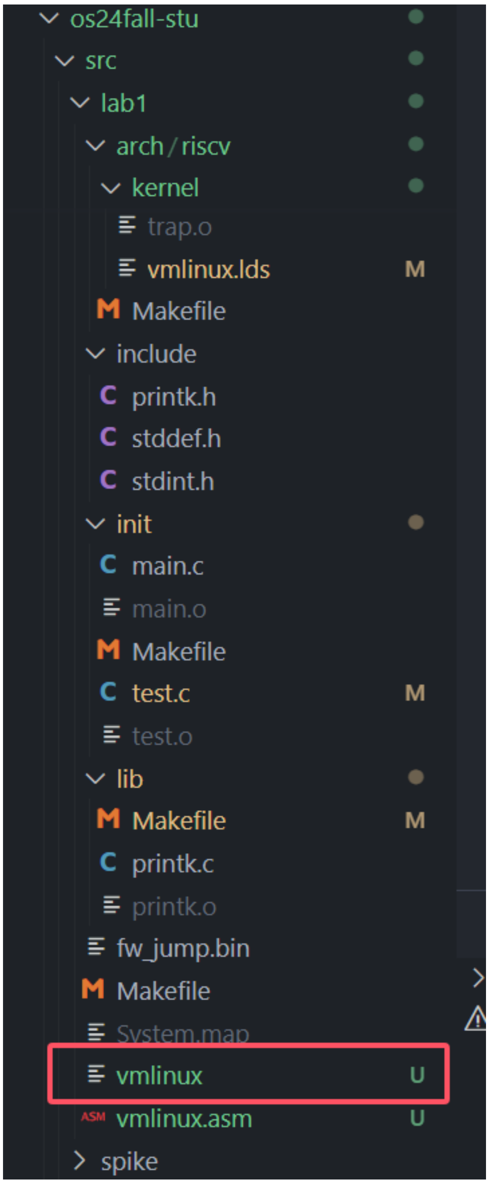
```
#include "stdint.h"

struct sbiret sbi_ecall(uint64_t eid, uint64_t fid, uint64_t arg0,
                        uint64_t arg1, uint64_t arg2, uint64_t arg3,
                        uint64_t arg4, uint64_t arg5) {
  struct sbiret ret;

  __asm__ volatile(
      "mv a7, %2\n"
      "mv a6, %3\n"
      "mv a0, %4\n"
      "mv a1, %5\n"
      "mv a2, %6\n"
      "mv a3, %7\n"
      "mv a4, %8\n"
```

```c
      "mv a5, %9\n"
      "ecall\n"
      "mv %0, a0\n"
      "mv %1, a1\n"
      : "=r"(ret.error), "=r"(ret.value)
      : "r"(eid), "r"(fid), "r"(arg0), "r"(arg1), "r"(arg2), "r"(arg3),
        "r"(arg4), "r"(arg5)
      : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7");

  return ret;
}

struct sbiret sbi_set_timer(uint64_t stime_value) {
  struct sbiret ret;

  sbi_ecall(0x54494d45, 0, stime_value, 0, 0, 0, 0, 0);

  return ret;
}

struct sbiret sbi_debug_console_write_byte(uint8_t byte) {
  struct sbiret ret;

  sbi_ecall(0x4442434e, 2, byte, 0, 0, 0, 0, 0);

  return ret;
}

struct sbiret sbi_system_reset(uint32_t reset_type, uint32_t reset_reason) {
  struct sbiret ret;

  sbi_ecall(0x53525354, 0, reset_type, reset_reason, 0, 0, 0, 0);

  return ret;
}
```

4. 修改 `defs` ，参考 `csr_write` 的宏定义对 `csr_read` 进行宏定义

```c
#define csr_read(csr)                                       \
  ({                                                        \
    uint64_t __v;                                           \
    asm volatile("csrr %0, " #csr : "=r"(__v) : : "memory"); \
    __v;                                                    \
  })
```

5. 运行 `make`，发现根目录下成功生成了 `vmlinux`



6. 运行 `make run`，正确启动并显示了 `2024 ZJU operating system`

终端

```
make[2]: Leaving directory '/usr/os24fall-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
riscv64-linux-gnu-objdump -S ../../vmlinux > ../../vmlinux.asm
nm ../../vmlinux >  ../../System.map
make[1]: Leaving directory '/usr/os24fall-stu/src/lab1/arch/riscv'

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

OpenSBI v1.5.1
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Platform Name             : riscv-virtio,qemu
Platform Features         : medeleg
Platform HART Count       : 1
Platform IPI Device       : aclint-mswi
Platform Timer Device     : aclint-mtimer @ 10000000Hz
Platform Console Device   : uart8250
```

```
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
```

RV64 时钟中断处理

1. 修改 `vmlinux.lds` 以及 `head.S`，这一部分文档中已经提供了修改内容，不再赘述
2. 开启trap处理

- 首先利用 `la` 指令将 `_traps` 所表示的地址写入 `a0`，之后利用csr指令 `csrw` 将 `a0` 的值写入 `stvec`
- 之后我们要设置 `sie` 寄存器的 `STIE` 位为1，查询可知对应 `sie[5]`，因此对应十六进制为 `0x20`，因此我们先利用 `csrr` 指令将 `sie` 值取出，同时将 `a0` 通过 `ori` 指令设置为 `0x20`，使用位运算可以提高运算效率，最后再将 `a0` 存回 `sie`
- 设置第一次时钟中断，这里我们调用 `sbi_set_timer` 完成。即首先利用 `rdtime` 获取当前时间，之后加上1秒钟(由于QEMU时钟频率是10MHz，因此1秒钟相当于10000000个时钟周期)。之后将 `a6` 和 `a7` 设置为 `sbi_set_timer` 对应的Function ID和Extension ID，最后调用 `ecall` 就相当于调用 `sbi_set_timer`
- 之后，类似第二步，查询可知 `SIE` 对应 `sstatus[1]`，对应 `0x2`

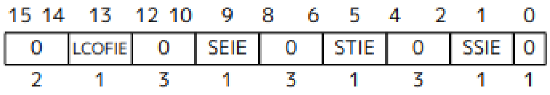


Figure 48. Standard portion (bits 15:0) of **sie**.

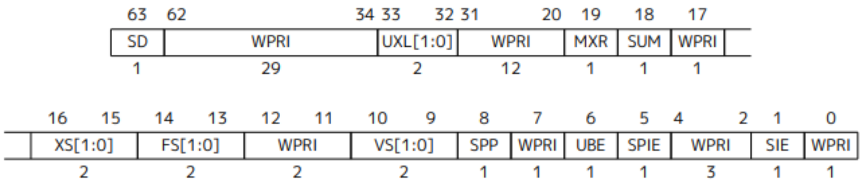


*Figure 43. Supervisor-mode status register (***sstatus***) when SXLEN=64.*

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

# set sie[STIE] = 1
csrr a0, sie
ori a0, a0, 0x20
csrw sie, a0

# set first time interrupt
rdtime a0
la t0, 10000000
add a0, a0, t0
la a6, 0x0
la a7, 0x54494d45
ecall

# set sstatus[SIE] = 1
csrr a0, sstatus
ori a0, a0, 0x2
csrw sstatus, a0
```

3. 实现上下文切换

- 首先将31个寄存器(`x0` 不需要保存)和 `sepc` 保存到栈上，这里我通过 `csrr` 指令将 `scause` 和 `sepc` 存储到 `a0` 和 `a1` ，因此我先存储 `a0` 和 `a1` 原本的值，再获取这两个CSR寄存器的值。值得注意的是，由于是64位，因此每个寄存器大小8字节
- 接下来调用 `trap_handler` 函数
- 接下来从栈中读取31个寄存器和 `sepc` 的值，这里我同样先将 `sepc` 取出再取出 `a1` ，同时需要注意的是，由于 `x2` 即为 `sp` ，因此需要最后取出
- 最后调用 `sret` 从trap中返回，注意我们这里是Supervisor Mode，不能使用 `mret`

```
.extern trap_handler
.section .text.entry
.align 2
```

```
    .globl _traps
_traps:
    # 1. save 32 registers and sepc to stack
    addi sp, sp, -256
    sd x1, 248(sp)
    sd x3, 240(sp)
    sd x4, 232(sp)
    sd x5, 224(sp)
    sd x6, 216(sp)
    sd x7, 208(sp)
    sd x8, 200(sp)
    sd x9, 192(sp)
    sd x10, 184(sp)
    sd x11, 176(sp)
    sd x12, 168(sp)
    sd x13, 160(sp)
    sd x14, 152(sp)
    sd x15, 144(sp)
    sd x16, 136(sp)
    sd x17, 128(sp)
    sd x18, 120(sp)
    sd x19, 112(sp)
    sd x20, 104(sp)
    sd x21, 96(sp)
    sd x22, 88(sp)
    sd x23, 80(sp)
    sd x24, 72(sp)
    sd x25, 64(sp)
    sd x26, 56(sp)
    sd x27, 48(sp)
    sd x28, 40(sp)
    sd x29, 32(sp)
    sd x30, 24(sp)
    sd x31, 16(sp)
    csrr a0, scause
    csrr a1, sepc
    sd a1, 8(sp)
    sd x2, 0(sp)

    # 2. call trap_handler
    call trap_handler

    # 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
    ld a1, 8(sp)
    csrw sepc, a1
    ld x1, 248(sp)
    ld x3, 240(sp)
```

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

    # 4. return from trap
    sret
```

4. 实现trap处理函数

- `scause` 最高位若为1则表示位interrupt，因此 `scause` 输入与 `flag` 进行与运算后若不为0，则说明为interrupt
- supervisor timer interrupt的exception code为5，因此将 `scause` 与 `~flag` 进行与运算就可以将最高位的1变成0，之后再和 `0x5` 进行比较，若相同则说明是timer interrupt，输出 `[S] Supervisor Mode Timer Interrupt`
- 若不为timer interrupt则输出 `[S] Supervisor Mode Other Interrupt`，并输出 `scause` 和 `sepc`
- 若不为interrupt则输出 `[S] Supervisor Mode Exception`，并输出 `scause` 和 `sepc`

Figure 52. Supervisor Cause register `scause`.

Table 22. Supervisor cause register (`scause`*) values after trap. Synchronous exception priorities are given by Table 15.*

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2-4	*Reserved*
1	5	Supervisor timer interrupt
1	6-8	*Reserved*
1	9	Supervisor external interrupt
1	10-12	*Reserved*
1	13	Counter-overflow interrupt
1	14-15	*Reserved*
1	≥16	*Designated for platform use*

```
#include "printk.h"
#include "stdint.h"

void trap_handler(uint64_t scause, uint64_t sepc) {
  // 通过 `scause` 判断 trap 类型
  // 如果是 interrupt 判断是否是 timer interrupt
  // 如果是 timer interrupt 则打印输出相关信息, 并通过 `clock_set_next_event()`设置下
一次时钟中断
  // `clock_set_next_event()` 见 4.3.4 节
  // 其他 interrupt /exception 可以直接忽略, 推荐打印出来供以后调试
  uint64_t flag = 0x8000000000000000;  // 第一位是1
  uint64_t exception_code = 0x5;       // exception code for timer interrupt
  if (scause & flag)                   // if interrupt
    if ((scause & ~flag) == exception_code) {  // if timer interrupt
      printk("[S] Supervisor Mode Timer Interrupt\n");
      clock_set_next_event();
    } else
      printk("[S] Supervisor Mode Other Interrupt (scause: %lx, sepc: %lx).\n",
scause, sepc);
  else
    printk("[S] Supervisor Mode Exception (scause: %lx, sepc: %lx).\n", scause,
sepc);
}
```

5. 实现时钟中断相关函数

- `get_cycles` 直接调用 `rdtime` 当前 `cycle` 数即可
- `clock_set_next_event` 同样和之前在开启trap处理中进行类似的操作, 将Function ID, Extension ID和 `stime_value` 设置好后 `ecall` 即可

```
#include "stdint.h"
```

```c
// QEMU 中时钟的频率是 10MHz, 也就是 1 秒钟相当于 10000000 个时钟周期
uint64_t TIMECLOCK = 10000000;

uint64_t get_cycles() {
  // 编写内联汇编, 使用 rdtime 获取 time 寄存器中 (也就是 mtime
  // 寄存器) 的值并返回
  uint64_t cycles;
  // 使用 rdtime 获取 time 寄存器中的值
  __asm__ volatile("rdtime %0" : "=r"(cycles));
  return cycles;
}

void clock_set_next_event() {
  // 下一次时钟中断的时间点
  uint64_t next = get_cycles() + TIMECLOCK;

  // 使用 sbi_set_timer 来完成对下一次时钟中断的设置
  __asm__ volatile(
      "la a6, 0x0\n"
      "la a7, 0x54494d45\n"
      "mv a0, %0\n"
      "ecall\n"
      :
      : "r"(next)
      : "a0", "a7");
}
```

6. 修改test函数成文档中的即可
7. 正如之前在Makefile部分提到的，此处Makefile不需进行任何修改
8. 编译测试：依次运行 `make` 和 `make run` 后出现以下输出

终端

```
[S] Supervisor Mode Timer Interrupt
kernel is running!
kernel is running!
kernel is running!
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
kernel is running!
kernel is running!
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
kernel is running!
kernel is running!
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
kernel is running!
kernel is running!
kernel is running!
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
kernel is running!
kernel is running!
kernel is running!
[S] Supervisor Mode Timer Interrupt
```

二、实验心得与体会

感觉这次实验接受的新知识还是比较多的，学习了内联汇编、时钟中断等，也复习了之前的Makefile以及计组的汇编，总体来讲收获很大。整个过程其实最不适应的就是从计组的32位到现在的64位，导致一开始栈的大小设置错误。然后一开始对不同mode的理解也出了问题，导致使用了 `mret` 命令产生了bug。

三、思考题

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

- calling convention
  - 将函数参数存储到函数能访问的对应位置(`a0-a7, fa0-fa7`)
  - 利用 `jal` 指令跳转到函数开始位置
  - 获取函数需要的局部存储资源，按需保存寄存器
  - 运行函数中的指令
  - 将返回值存储到调用者能够访问到的位置，恢复寄存器，释放局部存储资源
  - 使用 `ret` 指令返回调用函数的位置
- Caller/ Callee Saved Register之间的区别在于当寄存器在函数中被修改时，如何保存该寄存器的值。我们假设函数F1调用函数F2。

- Caller Saved Register是调用者保存寄存器，指的是函数在调用另一个函数之前需要保存的寄存器，如函数F1在调用函数F2之前先保存寄存器的值，再在函数F2调用完毕后恢复寄存器的值，如 `t0-t6, a0-a7`
- Callee Saved Register是被调用者保存寄存器，指的是被调用的函数在使用这些寄存器之前，必须保存它们的当前值，并在函数返回前恢复，如函数F2在使用对应寄存器前要先保存该寄存器值，并在函数F2返回前恢复值，如 `s0-s11`

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

Table 18.2: RISC-V calling convention register usage.

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

- 编译后产生以下System.map，可以观察到我们的自定义符号，如 `boot_stack`，`boot_stack_top`，`sbi_ecall` 等等

```
0000000080200000 t $x
0000000080200054 t $x
0000000080200170 t $x
0000000080200198 t $x
00000000802001f0 t $x
00000000802002c4 t $x
0000000080200350 t $x
00000000802003e0 t $x
000000008020047c t $x
0000000080200524 t $x
0000000080200568 t $x
00000000802005b8 t $x
0000000080200600 t $x
0000000080200660 t $x
00000000802008cc t $x
0000000080200954 t $x
0000000080200c5c t $x
000000008020144c t $x
0000000080200000 A BASE_ADDR
0000000080203000 D TIMECLOCK
```

```
0000000080203008 d _GLOBAL_OFFSET_TABLE_
0000000080205000 B _ebss
0000000080203008 D _edata
0000000080205000 B _ekernel
0000000080202129 R _erodata
00000000802014cc T _etext
0000000080204000 B _sbss
0000000080203000 D _sdata
0000000080200000 T _skernel
0000000080202000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
0000000080200054 T _traps
0000000080204000 B boot_stack
0000000080205000 B boot_stack_top
0000000080200198 T clock_set_next_event
0000000080200170 T get_cycles
0000000080200600 T isspace
0000000080202118 r lowerxdigits.0
0000000080200954 t print_dec_int
000000008020144c T printk
00000000802005b8 T putc
00000000802008cc t puts_wo_nl
0000000080200350 T sbi_debug_console_write_byte
00000000802001f0 T sbi_ecall
00000000802002c4 T sbi_set_timer
00000000802003e0 T sbi_system_reset
0000000080200524 T start_kernel
0000000080200660 T strtol
0000000080200568 T test
000000008020047c T trap_handler
0000000080202100 r upperxdigits.1
0000000080200c5c T vprintfmt
```

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

- 我们在 `test.c` 中加入以下代码，重新 `make` 后 `make run` ，即可得到 `sstatus value: 8000000200006002`

```
...
#define csr_read(csr)                          \
  ({                                           \
    unsigned long __tmp;                       \
    __asm__ volatile("csrr %0, " #csr : "=r"(__tmp)); \
    __tmp;                                     \
  })
void test() {
  ...
  unsigned long sstatus_value = csr_read(sstatus);
  printk("sstatus value: %lx\n", sstatus_value);
  ...
}
```

- 对照手册可以发现以下信息
  - `SPP` 位为0，说明trap来自user mode
  - `SIE` 位为1，即supervisor mode下的全局中断使能位，即hart于user mode和supervisor mode运行时都打开中断全局
  - `SPIE` 位为0，SPIE位记录的是在进入S-Mode之前S-Mode中断是否开启。进入trap时，系统会自动将SPIE位设置为SIE位，SIE设置为0；执行 `sret` 后，SPIE的值会重新放置到SIE位上来恢复原先的值，并且将SPIE的值置为1。

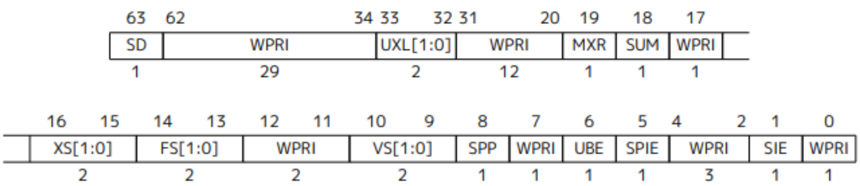


Figure 43. Supervisor-mode status register (`sstatus`) when SXLEN=64.

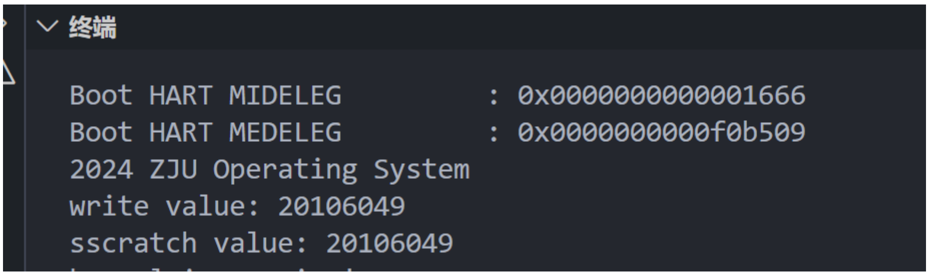
4. 用 `csr_write` 宏向 `sscratch` 寄存器写入数据，并验证是否写入成功并截图

- 对 `test.c` 添加如下内容，写入学号后八位

```
...
#define csr_read(csr)                          \
  ({                                           \
    unsigned long __tmp;                       \
    __asm__ volatile("csrr %0, " #csr : "=r"(__tmp)); \
    __tmp;                                     \
  })

#define csr_write(csr, value) \
  __asm__ volatile("csrw " #csr ", %0" : : "r"(value))

void test() {
  ...
```

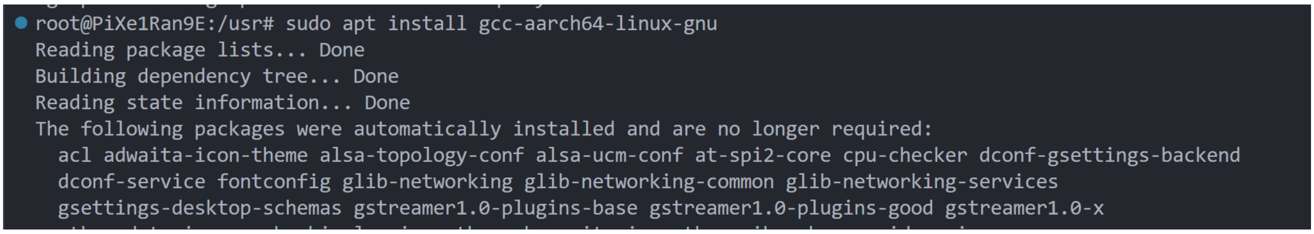
```c
    unsigned long write_value = 0x20106049;
    csr_write(sscratch, write_value);
    unsigned long read_value = csr_read(sscratch);
    printk("write value: %lx\n", write_value);
    printk("sscratch value: %lx\n", read_value);
    ...
}
```

- 运行后发现成功写入

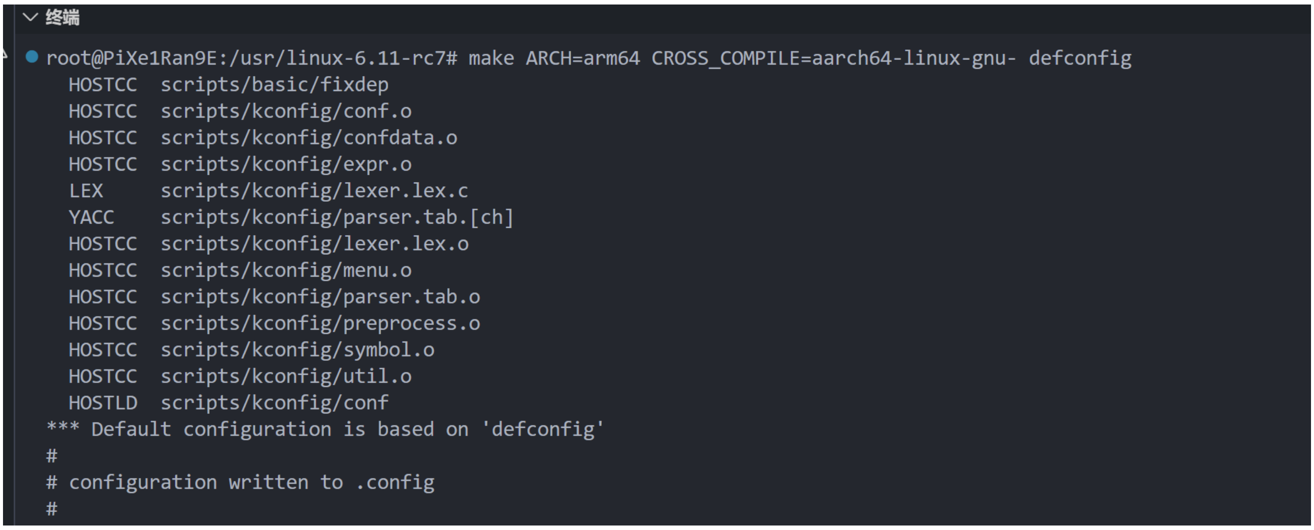


5. 详细描述你可以通过什么步骤来得到 `arch/arm64/kernel/sys.i`，给出过程以及截图

- 首先安装 `gcc-aarch64-linux-gnu`



- 之后在Linux内核根目录修改make的 `defconfig` 为 `arm64`



- 之后编译 `arch/arm64/kernel/sys.c` 为 `sys.i`

```
root@PiXe1Ran9E:/usr/linux-6.11-rc7# make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i
  SYNC    include/config/auto.conf.cmd
  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]
  HOSTCC  scripts/dtc/dtc-lexer.lex.o
  HOSTCC  scripts/dtc/dtc-parser.tab.o
  HOSTLD  scripts/dtc/dtc
  HOSTCC  scripts/dtc/libfdt/fdt.o
  HOSTCC  scripts/dtc/libfdt/fdt_ro.o
```

- 之后我们切换到对应目录可以观察到有对应文件

```
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch/arm64/kernel# ls
Makefile                 efi-rt-wrapper.S  jump_label.c          proton-pack.c       sys.i
Makefile.syscalls        efi.c             kaslr.c               psci.c              sys32.c
acpi.c                   elfcore.c         kexec_image.c         ptrace.c            sys_compat.c
acpi_numa.c              entry-common.c    kgdb.c                reloc_test_core.c   syscall.c
acpi_parking_protocol.c  entry-fpsimd.S    kuser32.S             reloc_test_syms.S   time.c
alternative.c            entry-ftrace.S    machine_kexec.c       relocate_kernel.S   topology.c
armv8_deprecated.c       entry.S           machine_kexec_file.c  return_address.c    trace-events-emulation.h
asm-offsets.c            fpsimd.c          module-plts.c         sdei.c              traps.c
asm-offsets.s            ftrace.c          module.c              setup.c             vdso
cacheinfo.c              head.S            mte.c                 signal.c            vdso-wrap.S
compat_alignment.c       hibernate-asm.S   paravirt.c            signal32.c          vdso.c
cpu-reset.S              hibernate.c       patching.c            sigreturn32.S       vdso32
cpu_errata.c             hw_breakpoint.c   pci.c                 sleep.S             vdso32-wrap.S
cpu_ops.c                hyp-stub.S        perf_callchain.c      smccc-call.S        vmcore_info.c
cpufeature.c             idle.c            perf_regs.c           smp.c               vmlinux.lds.S
```

6. 寻找Linux v6.0中 `ARM32` `RV32` `RV64` `x86_64` 架构的系统调用表

- 这里由于我安装的是Linux6.11-rc7，因此以下系统调用表都来自此版本
- `ARM32`：切换到文件夹 `/usr/linux-6.11-rc7/arch/arm/tools`，打开文件 `syscall.tbl`，可以观察到系统调用表

```
root@PiXe1Ran9E:/usr# ls
aarch64-linux-gnu  arm-linux-gnueabi  bin  games  include  lib  lib32  lib64  libexec  libx32  linux-6.11-rc7  local  os-lab  os24fall-stu  qemu  riscv64-linux-gnu  sbin  share  src
root@PiXe1Ran9E:/usr# cd linux-6.11-rc7
root@PiXe1Ran9E:/usr/linux-6.11-rc7# ls
COPYING  Documentation  Kconfig   MAINTAINERS  Module.symvers  arch   certs   drivers  include  io_uring  kernel  mm   rust     scripts   sound  usr
CREDITS  Kbuild         LICENSES  Makefile     README          block  crypto  fs       init     ipc       lib     net  samples  security  tools  virt
root@PiXe1Ran9E:/usr/linux-6.11-rc7# cd arch
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch# ls
Kconfig  alpha  arc  arm  arm64  csky  hexagon  loongarch  m68k  microblaze  mips  nios2  openrisc  parisc  powerpc  riscv  s390  sh  sparc  um  x86  xtensa
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch# cd arm
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch/arm# ls
Kbuild            Kconfig.platforms  crypto        mach-alpine  mach-bcm        mach-dove        mach-highbank  mach-keystone  mach-milbeaut  mach-mxs      mach-orion5x   mach-rpc       mach-socfpga  mach-tegra      mm           tools
Kconfig           Makefile           include       mach-artpec  mach-berlin     mach-ep93xx      mach-hisi      mach-lpc18xx   mach-mmp       mach-nomadik  mach-pxa       mach-s3c       mach-spear    mach-ux500      net          vdso
Kconfig-nommu     boot               kernel        mach-aspeed  mach-clps711x   mach-exynos      mach-hpe       mach-lpc32xx   mach-mstar     mach-npcm     mach-qcom      mach-s5pv210   mach-sti      mach-versatile  nwfpe        vfp
Kconfig.assembler common             lib           mach-at91    mach-davinci    mach-footbridge  mach-imx       mach-mediatek  mach-mv78xx0   mach-omap1    mach-realtek   mach-sa1100    mach-stm32    mach-vt8500     plat-orion   xen
Kconfig.debug     configs            mach-actions  mach-axxia   mach-digicolor  mach-gemini      mach-ixp4xx    mach-meson     mach-mvebu     mach-omap2    mach-rockchip  mach-shmobile  mach-sunxi    mach-zynq       probes
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch/arm# cd tools
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch/arm/tools# ;s
bash: syntax error near unexpected token `;'
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch/arm/tools# ls
Makefile  gen-mach-types  mach-types  syscall.tbl  syscallnr.sh
root@PiXe1Ran9E:/usr/linux-6.11-rc7/arch/arm/tools# vim syscall.tbl
```

```
∨ 终端

# SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
#
# Linux system call numbers and entry vectors
#
# The format is:
# <num> <abi>   <name>                  [<entry point>]                 [<oabi compat entry point>]]
#
# Where abi is:
#  common - for system calls shared between oabi and eabi (may have compat)
#  oabi   - for oabi-only system calls (may have compat)
#  eabi   - for eabi-only system calls
#
# For each syscall number, "common" is mutually exclusive with oabi and eabi
#
0       common  restart_syscall         sys_restart_syscall
1       common  exit                    sys_exit
2       common  fork                    sys_fork
3       common  read                    sys_read
4       common  write                   sys_write
5       common  open                    sys_open
6       common  close                   sys_close
# 7 was sys_waitpid
8       common  creat                   sys_creat
9       common  link                    sys_link
10      common  unlink                  sys_unlink
11      common  execve                  sys_execve
12      common  chdir                   sys_chdir
13      oabi    time                    sys_time32
14      common  mknod                   sys_mknod
15      common  chmod                   sys_chmod
16      common  lchown                  sys_lchown16
# 17 was sys_break
# 18 was sys_stat
19      common  lseek                   sys_lseek
20      common  getpid                  sys_getpid
"syscall.tbl" 479L, 17510B
```

- `RV32`：系统调用表文件 `sys_call_table.c` 在目录 `arch/riscv/kernel` 下。这里首先需要把默认编译设置更改为32位，通过命令 `make ARCH=riscvCROSS COMPILE=riscv64-linux-gnu-rv32 defconfig`

```
root@PiXe1Ran9E:/usr/linux-6.11-rc7# make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- rv32_defconfig
*** Default configuration is based on 'defconfig'
#
# No change to .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)
#
#
# configuration written to .config
#
```

之后编译 `sys_call_table.c`

```
root@PiXe1Ran9E:/usr/linux-6.11-rc7# make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i
  SYNC    include/config/auto.conf.cmd
  HOSTCC  scripts/selinux/genheaders/genheaders
  HOSTCC  scripts/selinux/mdp/mdp
  SYSHDR  arch/riscv/include/generated/uapi/asm/unistd_32.h
  SYSHDR  arch/riscv/include/generated/uapi/asm/unistd_64.h
  SYSTBL  arch/riscv/include/generated/asm/syscall_table_32.h
  SYSTBL  arch/riscv/include/generated/asm/syscall_table_64.h
  UPD     include/generated/compile.h
  CC      scripts/mod/empty.o
  MKELF   scripts/mod/elfconfig.h
  HOSTCC  scripts/mod/modpost.o
  CC      scripts/mod/devicetable-offsets.s
  UPD     scripts/mod/devicetable-offsets.h
  HOSTCC  scripts/mod/file2alias.o
  HOSTCC  scripts/mod/sumversion.o
  HOSTCC  scripts/mod/symsearch.o
  HOSTLD  scripts/mod/modpost
  CC      kernel/bounds.s
  UPD     include/generated/bounds.h
  CC      arch/riscv/kernel/asm-offsets.s
  UPD     include/generated/asm-offsets.h
  CALL    scripts/checksyscalls.sh
  LDS     arch/riscv/kernel/vdso/vdso.lds
  AS      arch/riscv/kernel/vdso/rt_sigreturn.o
  AS      arch/riscv/kernel/vdso/getcpu.o
  AS      arch/riscv/kernel/vdso/flush_icache.o
  CC      arch/riscv/kernel/vdso/hwprobe.o
  AS      arch/riscv/kernel/vdso/sys_hwprobe.o
  AS      arch/riscv/kernel/vdso/note.o
  VDSOLD  arch/riscv/kernel/vdso/vdso.so.dbg
  VDSOSYM include/generated/vdso-offsets.h
  CPP     arch/riscv/kernel/syscall_table.i
```

之后使用Vim查看并搜索关键词 `sys_call_table` ，可以找到对应的内容

```
void * const sys_call_table[463] = {
 [0 ... 463 - 1] = __riscv_sys_ni_syscall,
# 1 "./arch/riscv/include/asm/syscall_table.h" 1





# 1 "./arch/riscv/include/generated/asm/syscall_table_32.h" 1
[0] = __riscv_sys_io_setup,
[1] = __riscv_sys_io_destroy,
[2] = __riscv_sys_io_submit,
[3] = __riscv_sys_io_cancel,
[4] = __riscv_sys_ni_syscall,
[5] = __riscv_sys_setxattr,
[6] = __riscv_sys_lsetxattr,
[7] = __riscv_sys_fsetxattr,
[8] = __riscv_sys_getxattr,
[9] = __riscv_sys_lgetxattr,
[10] = __riscv_sys_fgetxattr,
[11] = __riscv_sys_listxattr,
[12] = __riscv_sys_llistxattr,
[13] = __riscv_sys_flistxattr,
[14] = __riscv_sys_removexattr,
[15] = __riscv_sys_lremovexattr,
[16] = __riscv_sys_fremovexattr,
[17] = __riscv_sys_getcwd,
[18] = __riscv_sys_ni_syscall,
[19] = __riscv_sys_eventfd2,
[20] = __riscv_sys_epoll_create1,
[21] = __riscv_sys_epoll_ctl,
[22] = __riscv_sys_epoll_pwait,
[23] = __riscv_sys_dup,
[24] = __riscv_sys_dup3,
[25] = __riscv_sys_fcntl64,
```

- `RV64`：同样地，我们首先恢复默认配置

```
root@PiXe1Ran9E:/usr/linux-6.11-rc7# make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
```

之后编译 `sys_call_table.c`

```
root@PiXe1Ran9E:/usr/linux-6.11-rc7# make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i
  SYNC    include/config/auto.conf.cmd
  UPD     include/generated/compile.h
  CC      scripts/mod/empty.o
  MKELF   scripts/mod/elfconfig.h
  HOSTCC  scripts/mod/modpost.o
  CC      scripts/mod/devicetable-offsets.s
  UPD     scripts/mod/devicetable-offsets.h
  HOSTCC  scripts/mod/file2alias.o
  HOSTCC  scripts/mod/sumversion.o
  HOSTCC  scripts/mod/symsearch.o
  HOSTLD  scripts/mod/modpost
  CC      kernel/bounds.s
  UPD     include/generated/bounds.h
  CC      arch/riscv/kernel/asm-offsets.s
  UPD     include/generated/asm-offsets.h
  CALL    scripts/checksyscalls.sh
  LDS     arch/riscv/kernel/vdso/vdso.lds
  AS      arch/riscv/kernel/vdso/rt_sigreturn.o
  AS      arch/riscv/kernel/vdso/getcpu.o
  AS      arch/riscv/kernel/vdso/flush_icache.o
  CC      arch/riscv/kernel/vdso/hwprobe.o
  AS      arch/riscv/kernel/vdso/sys_hwprobe.o
  AS      arch/riscv/kernel/vdso/note.o
  CC      arch/riscv/kernel/vdso/vgettimeofday.o
  VDSOLD  arch/riscv/kernel/vdso/vdso.so.dbg
  VDSOSYM include/generated/vdso-offsets.h
  LDS     arch/riscv/kernel/compat_vdso/compat_vdso.lds
  VDSOAS  arch/riscv/kernel/compat_vdso/rt_sigreturn.o
  VDSOAS  arch/riscv/kernel/compat_vdso/getcpu.o
  VDSOAS  arch/riscv/kernel/compat_vdso/flush_icache.o
  VDSOAS  arch/riscv/kernel/compat_vdso/note.o
  VDSOLD  arch/riscv/kernel/compat_vdso/compat_vdso.so.dbg
  VDSOSYM include/generated/compat_vdso-offsets.h
  CPP     arch/riscv/kernel/syscall_table.i
```

用Vim查看并搜索 `sys_call_table`，可以观察到对应内容

```
void * const sys_call_table[463] = {
 [0 ... 463 - 1] = __riscv_sys_ni_syscall,
# 1 "./arch/riscv/include/asm/syscall_table.h" 1



# 1 "./arch/riscv/include/generated/asm/syscall_table_64.h" 1
[0] = __riscv_sys_io_setup,
[1] = __riscv_sys_io_destroy,
[2] = __riscv_sys_io_submit,
[3] = __riscv_sys_io_cancel,
[4] = __riscv_sys_io_getevents,
[5] = __riscv_sys_setxattr,
[6] = __riscv_sys_lsetxattr,
[7] = __riscv_sys_fsetxattr,
[8] = __riscv_sys_getxattr,
[9] = __riscv_sys_lgetxattr,
[10] = __riscv_sys_fgetxattr,
[11] = __riscv_sys_listxattr,
[12] = __riscv_sys_llistxattr,
[13] = __riscv_sys_flistxattr,
[14] = __riscv_sys_removexattr,
[15] = __riscv_sys_lremovexattr,
[16] = __riscv_sys_fremovexattr,
[17] = __riscv_sys_getcwd,
[18] = __riscv_sys_ni_syscall,
[19] = __riscv_sys_eventfd2,
[20] = __riscv_sys_epoll_create1,
[21] = __riscv_sys_epoll_ctl,
[22] = __riscv_sys_epoll_pwait,
[23] = __riscv_sys_dup,
[24] = __riscv_sys_dup3,
[25] = __riscv_sys_fcntl,
[26] = __riscv_sys_inotify_init1,
[27] = __riscv_sys_inotify_add_watch,
```

- `x86-64`：系统调用表在 `arch/x86/entry/syscalls/syscall_64.tbl`，打开即可查看

```
# SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> [<compat entry point> [noreturn]]
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
4       common  stat                    sys_newstat
5       common  fstat                   sys_newfstat
6       common  lstat                   sys_newlstat
7       common  poll                    sys_poll
8       common  lseek                   sys_lseek
9       common  mmap                    sys_mmap
10      common  mprotect                sys_mprotect
11      common  munmap                  sys_munmap
12      common  brk                     sys_brk
13      64      rt_sigaction            sys_rt_sigaction
14      common  rt_sigprocmask          sys_rt_sigprocmask
15      64      rt_sigreturn            sys_rt_sigreturn
16      64      ioctl                   sys_ioctl
17      common  pread64                 sys_pread64
18      common  pwrite64                sys_pwrite64
19      64      readv                   sys_readv
20      64      writev                  sys_writev
21      common  access                  sys_access
22      common  pipe                    sys_pipe
23      common  select                  sys_select
"syscall_64.tbl" 433L, 15472B
```

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）文件是一种用于存储可执行文件、目标代码和共享库的文件格式，由Header，Program Header Table和Section Header Table等几部分构成，其中Header中的Magic代表文件格式。ELF文件可以被操作系统直接执行，包含了程序运行所需的所有信息，如代码、数据和动态链接信息。ELF文件常用于Unix及Unix-like操作系统中。
- 接下来我们查看 `test.o` 相关信息， `readelf -a test.o` ，Header中的Magic `7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00` 即为标识文件格式的部分，同时我们可以看到Header还包括了Class，Data，OS/ABI，Machine等标识了文件基本信息的内容。同时Section Header列出了文件每个节的详细信息，包括编号、名称、类型、地址、偏移量、大小等信息。同时我们可以看到此 `.o` 文件无Program Header。

```
root@PiXe1Ran9E:/usr/os24fall-stu/src/lab1/init# readelf -a test.o
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:                              REL (Relocatable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          0 (bytes into file)
  Start of section headers:          5552 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           0 (bytes)
  Number of program headers:         0
  Size of section headers:           64 (bytes)
  Number of section headers:         26
  Section header string table index: 25

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         0000000000000000  00000040
       0000000000000000  0000000000000000  AX       0     0     4
  [ 2] .data             PROGBITS         0000000000000000  00000040
       0000000000000000  0000000000000000  WA       0     0     1
  [ 3] .bss              NOBITS           0000000000000000  00000040
       0000000000000000  0000000000000000  WA       0     0     1
```

```
There are no section groups in this file.

There are no program headers in this file.

There is no dynamic section in this file.

Relocation section '.rela.text.test' at offset 0xcd8 contains 20 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
00000000003c  002700000017 R_RISCV_PCREL_HI2 0000000000000000 .LC0 + 0
00000000003c  000000000033 R_RISCV_RELAX                        0
000000000040  002800000018 R_RISCV_PCREL_LO1 000000000000003c .L0  + 0
000000000040  000000000033 R_RISCV_RELAX                        0
000000000044  004500000013 R_RISCV_CALL_PLT  0000000000000000 printk + 0
000000000044  000000000033 R_RISCV_RELAX                        0
000000000050  002900000017 R_RISCV_PCREL_HI2 0000000000000018 .LC1 + 0
000000000050  000000000033 R_RISCV_RELAX                        0
000000000054  002a00000018 R_RISCV_PCREL_LO1 0000000000000050 .L0  + 0
000000000054  000000000033 R_RISCV_RELAX                        0
000000000058  004500000013 R_RISCV_CALL_PLT  0000000000000000 printk + 0
000000000058  000000000033 R_RISCV_RELAX                        0
000000000084  002d00000010 R_RISCV_BRANCH    0000000000000060 .L3 + 0
000000000088  002b00000017 R_RISCV_PCREL_HI2 0000000000000030 .LC2 + 0
000000000088  000000000033 R_RISCV_RELAX                        0
00000000008c  002c00000018 R_RISCV_PCREL_LO1 0000000000000088 .L0  + 0
00000000008c  000000000033 R_RISCV_RELAX                        0
000000000090  004500000013 R_RISCV_CALL_PLT  0000000000000000 printk + 0
000000000090  000000000033 R_RISCV_RELAX                        0
00000000009c  002d00000011 R_RISCV_JAL       0000000000000060 .L3 + 0
```

- 接下来我们利用 `objdump` 查看此文件相关信息。首先可以看到此文件是适用于 64 位小端RISC-V架构的ELF文件，文件的标志位位于 `0x00000011` ，起始地址为 `0x0` ；其次我们看到 `.text` 段的反汇编，每行包括了指令的起始位置，指令的机器码，以及对应的汇编指令；接下来， `riscv64-linux-gnu-objdump -h test.o` 返回了文件各个段的详

细信息，包括对齐要求，文件偏移量，虚拟内存地址以及段特性等等；最后，`riscv64-linux-gnu-objdump -t test.o` 可以查看文件符号表的信息，包括 `.text` ， `.data` ， `.bss` ， `.rodata` 等符号，同时 `l` 表示这些符号仅在本文件中可见， `F` 表示这是一个函数符号。

```
root@PiXe1Ran9E:/usr/os24fall-stu/src/lab1/init# riscv64-linux-gnu-objdump -f test.o

test.o:     file format elf64-littleriscv
architecture: riscv:rv64, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x0000000000000000

root@PiXe1Ran9E:/usr/os24fall-stu/src/lab1/init# riscv64-linux-gnu-objdump -d test.o

test.o:     file format elf64-littleriscv


Disassembly of section .text.test:

0000000000000000 <test>:
   0:   fd010113                addi    sp,sp,-48
   4:   02113423                sd      ra,40(sp)
   8:   02813023                sd      s0,32(sp)
   c:   03010413                addi    s0,sp,48
  10:   fe042623                sw      zero,-20(s0)
  14:   201067b7                lui     a5,0x20106
  18:   04978793                addi    a5,a5,73 # 20106049 <.LASF5+0x20105f89>
  1c:   fef43023                sd      a5,-32(s0)
  20:   fe043783                ld      a5,-32(s0)
  24:   14079073                csrw    sscratch,a5

0000000000000028 <.LBB2>:
  28:   140027f3                csrr    a5,sscratch
  2c:   fcf43c23                sd      a5,-40(s0)
  30:   fd843783                ld      a5,-40(s0)

0000000000000034 <.LBE2>:
  34:   fcf43823                sd      a5,-48(s0)
  38:   fe043583                ld      a1,-32(s0)
  3c:   00000517                auipc   a0,0x0
```

```
root@PiXe1Ran9E:/usr/os24fall-stu/src/lab1/init# riscv64-linux-gnu-objdump -h test.o

test.o:     file format elf64-littleriscv

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000000  0000000000000000  0000000000000000  00000040  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000040  2**0
                  ALLOC
  3 .rodata       00000044  0000000000000000  0000000000000000  00000040  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .text.test    000000a0  0000000000000000  0000000000000000  00000084  2**2
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  5 .debug_info   000000cd  0000000000000000  0000000000000000  00000124  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  6 .debug_abbrev 000000a0  0000000000000000  0000000000000000  000001f1  2**0
                  CONTENTS, READONLY, DEBUGGING, OCTETS
  7 .debug_aranges 00000030  0000000000000000  0000000000000000  00000291  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  8 .debug_rnglists 0000001e  0000000000000000  0000000000000000  000002c1  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
  9 .debug_line   000000bb  0000000000000000  0000000000000000  000002df  2**0
                  CONTENTS, RELOC, READONLY, DEBUGGING, OCTETS
 10 .debug_str    000000c7  0000000000000000  0000000000000000  0000039a  2**0
```

```
root@PiXe1Ran9E:/usr/os24fall-stu/src/lab1/init# riscv64-linux-gnu-objdump -t test.o

test.o:     file format elf64-littleriscv

SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 test.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .rodata        0000000000000000 .rodata
0000000000000000 l    d  .text.test     0000000000000000 .text.test
0000000000000000 l    d  .debug_info    0000000000000000 .debug_info
0000000000000000 l    d  .debug_abbrev  0000000000000000 .debug_abbrev
0000000000000000 l    d  .debug_aranges 0000000000000000 .debug_aranges
0000000000000000 l    d  .debug_rnglists        0000000000000000 .debug_rnglists
0000000000000000 l    d  .debug_line    0000000000000000 .debug_line
0000000000000000 l    d  .debug_str     0000000000000000 .debug_str
0000000000000000 l    d  .debug_line_str        0000000000000000 .debug_line_str
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 l    d  .debug_frame   0000000000000000 .debug_frame
0000000000000000 l    d  .riscv.attributes      0000000000000000 .riscv.attributes
0000000000000000 g     F .text.test     00000000000000a0 test
0000000000000000         *UND*  0000000000000000 printk
```

- 最后我们对整个工程执行 `make run` ，即运行 `vmlinux` 。之后开启另一个终端，输入 `echo $$` ，输出进程号 `71918` 。之后再 `cat /proc/71918/maps` 即可得到内存布局。我们可以观察到内存布局最上方是关于zsh本身二进制文件的映射，之后内容则显示了其他内容。

```
> cat /proc/71918/maps
5610f9bfc000-5610f9c13000 r--p 00000000 08:20 168062                     /usr/bin/zsh
5610f9c13000-5610f9cd1000 r-xp 00017000 08:20 168062                     /usr/bin/zsh
5610f9cd1000-5610f9cec000 r--p 000d5000 08:20 168062                     /usr/bin/zsh
5610f9cec000-5610f9cee000 r--p 000ef000 08:20 168062                     /usr/bin/zsh
5610f9cee000-5610f9cf4000 rw-p 000f1000 08:20 168062                     /usr/bin/zsh
5610f9cf4000-5610f9d08000 rw-p 00000000 00:00 0
5610fa611000-5610faa0f000 rw-p 00000000 00:00 0                          [heap]
7faa993dc000-7faa993f5000 r--s 00000000 08:20 167943                     /usr/share/zsh/functions/Zle.zwc
7faa9942c000-7faa9947c000 rw-p 00000000 00:00 0
7faa9947d000-7faa99495000 rw-p 00000000 00:00 0
7faa99495000-7faa99496000 r--p 00000000 08:20 168086                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/net/soc
ket.so
7faa99496000-7faa99497000 r-xp 00001000 08:20 168086                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/net/soc
ket.so
7faa99497000-7faa99498000 r--p 00002000 08:20 168086                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/net/soc
ket.so
7faa99498000-7faa99499000 r--p 00002000 08:20 168086                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/net/soc
ket.so
7faa99499000-7faa9949a000 rw-p 00003000 08:20 168086                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/net/soc
ket.so
7faa9949a000-7faa9949b000 r--p 00000000 08:20 168098                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/termcap
.so
7faa9949b000-7faa9949c000 r-xp 00001000 08:20 168098                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/termcap
.so
7faa9949c000-7faa9949d000 r--p 00002000 08:20 168098                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/termcap
.so
7faa9949d000-7faa9949e000 r--p 00002000 08:20 168098                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/termcap
.so
7faa9949e000-7faa9949f000 rw-p 00003000 08:20 168098                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/termcap
.so
7faa9949f000-7faa994a1000 r--p 00000000 08:20 168083                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/mathfun
c.so
7faa994a1000-7faa994a3000 r-xp 00002000 08:20 168083                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/mathfun
c.so
7faa994a3000-7faa994a4000 r--p 00004000 08:20 168083                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/mathfun
c.so
7faa994a4000-7faa994a5000 r--p 00004000 08:20 168083                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/mathfun
c.so
7faa994a5000-7faa994a6000 rw-p 00005000 08:20 168083                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/mathfun
c.so
7faa994a6000-7faa994aa000 rw-p 00000000 00:00 0
7faa994aa000-7faa994ac000 r--p 00000000 08:20 168080                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/files.s
o
7faa994ac000-7faa994af000 r-xp 00002000 08:20 168080                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/files.s
o
7faa994af000-7faa994b0000 r--p 00005000 08:20 168080                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/files.s
o
7faa994b0000-7faa994b1000 r--p 00005000 08:20 168080                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/files.s
o
7faa994b1000-7faa994b2000 rw-p 00006000 08:20 168080                     /usr/lib/x86_64-linux-gnu/zsh/5.8.1/zsh/files.s
```

8. 解释运行 `make run` 后OpenSBI输出中的 `MIDELEG` 和 `MEDELEG` 值的含义。

```
OpenSBI v1.5.1
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

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

- `MIDELEG` 指的是machine interrupt delegation register，即机器中断委托寄存器； `MEDELEG` 指的是machine exception delegation register，即机器异常委托寄存器。这两个寄存器可以通过置位将S或U态的trap转交给S态的trap处理程序
- 其中 `MIDELEG` 寄存器的值的含义与 `mip` 寄存器的一致，值 `0x222` 即 `0010 0010 0010` 表示将1,5,9位设为1

- 1表示 `SSIP` ，将软件中断委托给S模式
- 5表示 `STIP` ，将时钟中断委托给S模式
- 9表示 `SEIP` ，将外部中断委托给S模式

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WIRI	MEIP	HEIP	SEIP	UEIP	MTIP	HTIP	STIP	UTIP	MSIP	HSIP	SSIP	USIP
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.10: Machine interrupt-pending register (`mip`).

XLEN-1 12	11	10	9	8	7	6	5	4	3	2	1	0
WPRI	MEIE	HEIE	SEIE	UEIE	MTIE	HTIE	STIE	UTIE	MSIE	HSIE	SSIE	USIE
XLEN-12	1	1	1	1	1	1	1	1	1	1	1	1

Figure 3.11: Machine interrupt-enable register (`mie`).

- `MEDELEG` 寄存器每一位的含义如下表。则值 `0xb109` 即 `1011 0001 0000 1001` 表示将0,3,8,12,13,15位设为1，分别表示：
  - 0表示委托指令访问未对齐异常
  - 3表示委托断点异常
  - 8表示委托来自U-Mode的环境调用
  - 12表示委托Instruction Page Fault
  - 13表示委托Load Page Fault
  - 15表示委托Store/AMO Page Fault异常

Machine Exception Delegation Register (medeleg)		
CSR	0x302	
Bits	**Attr.**	**Description**
0	RW	Delegate Instruction Access Misaligned Exception
1	RW	Delegate Instruction Access Fault Exception
2	RW	Delegate Illegal Instruction Exception
3	RW	Delegate Breakpoint Exception
4	RW	Delegate Load Access Misaligned Exception
5	RW	Delegate Load Access Fault Exception
6	RW	Delegate Store/AMO Address Misaligned Exception
7	RW	Delegate Store/AMO Access Fault Exception
8	RW	Delegate Environment Call from U-Mode
9	RW	Delegate Environment Call from S-Mode
[11:0]	WARL	Reserved
12	RW	Delegate Instruction Page Fault
13	RW	Delegate Load Page Fault
14	WARL	Reserved
15	RW	Delegate Store/AMO Page Fault Exception
[63:16]	WARL	Reserved

Table 109: Machine Exception Delegation Register