操作系统lab1：RV64 内核引导与时钟中断处理

教师：李环、柳晴
学号：3220104519
姓名：蔡佳伟

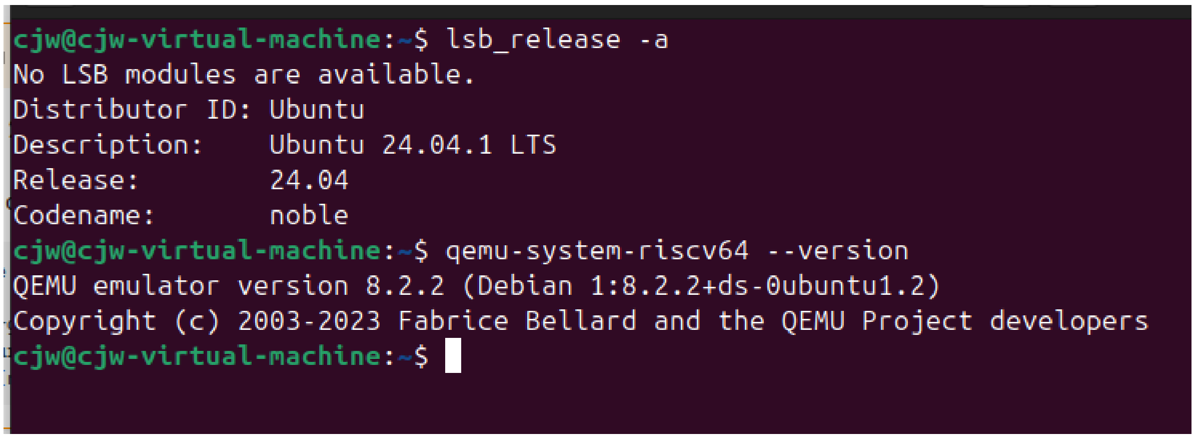
一、实验目的和要求

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

二、实验过程

2.0 更新qemu版本

实验过程中，做到一大半，完成修改RV64内核引导的所有步骤后，发现无论如何都无法输出2024那一段提示语（make显示正常，修改了很多发现没有问题）。然后就放弃休息了一段时间，再接着就发现实验指导更新了同时qemu需要更新。查看版本后发现是6.2，于是更新。

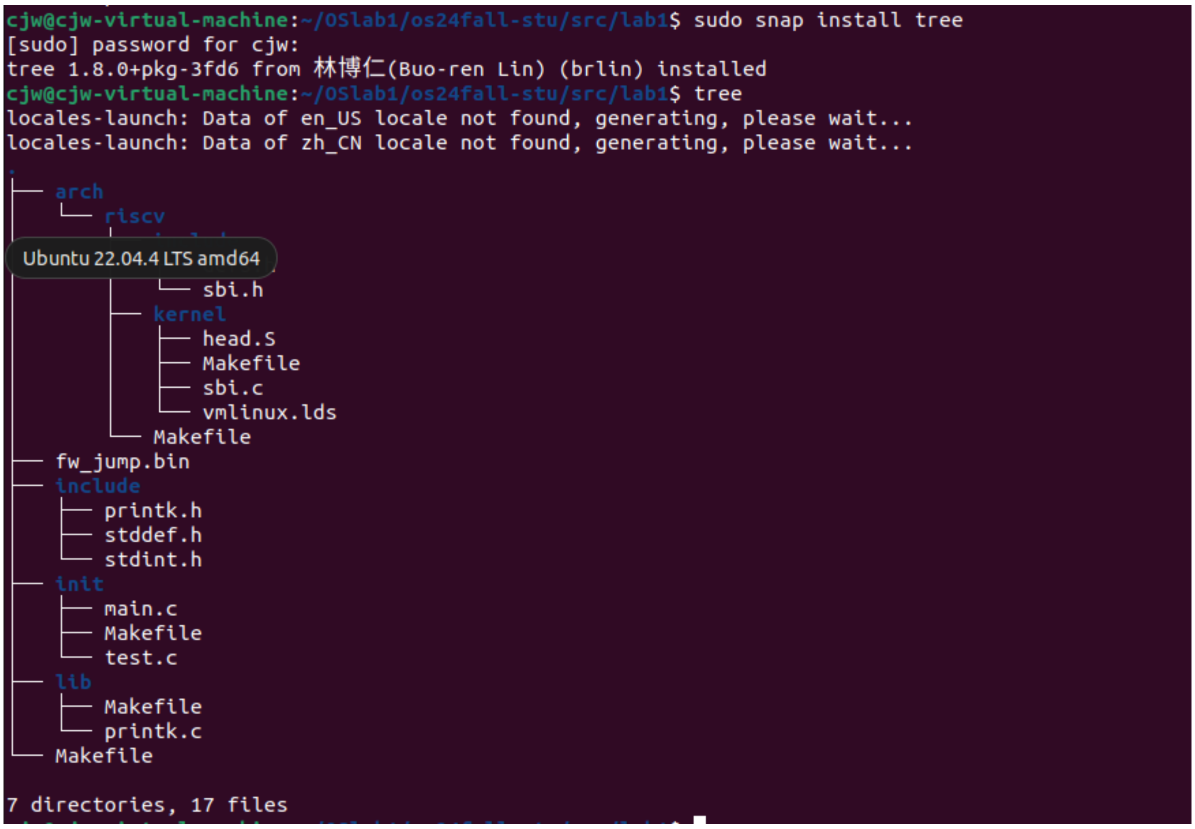

```
cjw@cjw-virtual-machine:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 24.04.1 LTS
Release:        24.04
Codename:       noble
cjw@cjw-virtual-machine:~$ qemu-system-riscv64 --version
QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.2)
Copyright (c) 2003-2023 Fabrice Bellard and the QEMU Project developers
cjw@cjw-virtual-machine:~$ 
```

直接更新了ubuntu版本到24.04，结果如上。

2.1 RV64内核引导

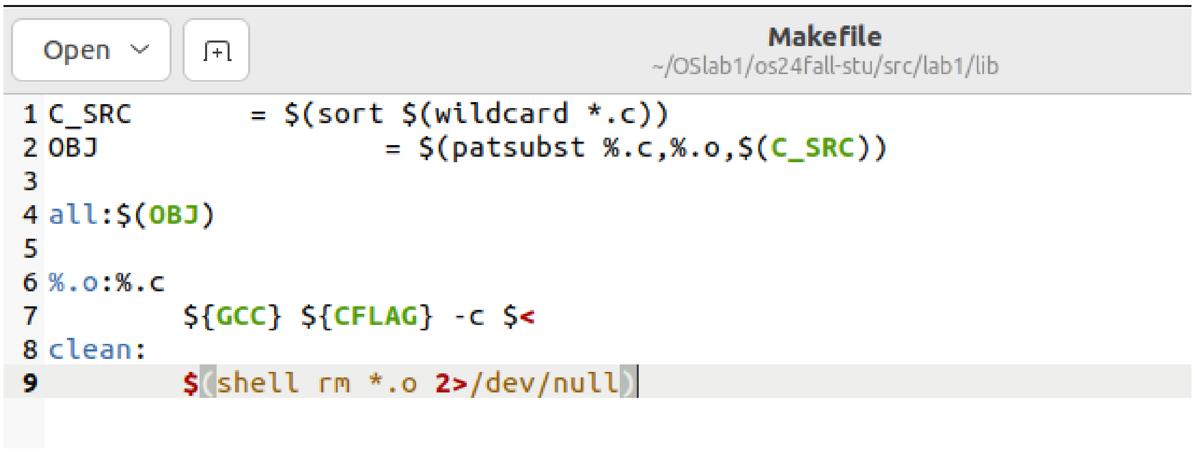
2.1.1 准备工作

clone整个文件夹后，安装tree所需要的文件，显示如下



```
cjw@cjw-virtual-machine:~/OSlab1/os24fall-stu/src/lab1$ sudo snap install tree
[sudo] password for cjw:
tree 1.8.0+pkg-3fd6 from 林博仁(Buo-ren Lin) (brlin) installed
cjw@cjw-virtual-machine:~/OSlab1/os24fall-stu/src/lab1$ tree
locales-launch: Data of en_US locale not found, generating, please wait...
locales-launch: Data of zh_CN locale not found, generating, please wait...
.
├── arch
│   └── riscv
│       │
Ubuntu 22.04.4 LTS amd64
│       │   └── sbi.h
│       ├── kernel
│       │   ├── head.S
│       │   ├── Makefile
│       │   ├── sbi.c
│       │   └── vmlinux.lds
│       └── Makefile
├── fw_jump.bin
├── include
│   ├── printk.h
│   ├── stddef.h
│   └── stdint.h
├── init
│   ├── main.c
│   ├── Makefile
│   └── test.c
├── lib
│   ├── Makefile
│   └── printk.c
└── Makefile

7 directories, 17 files
```

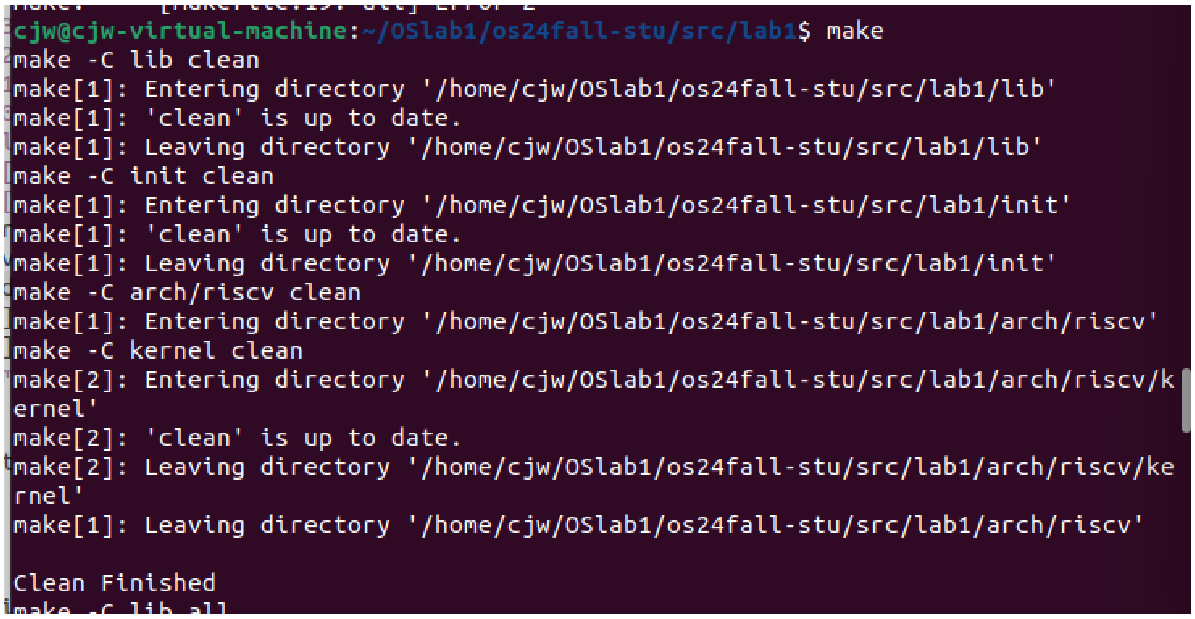
2.1.2 完善Makefile脚本



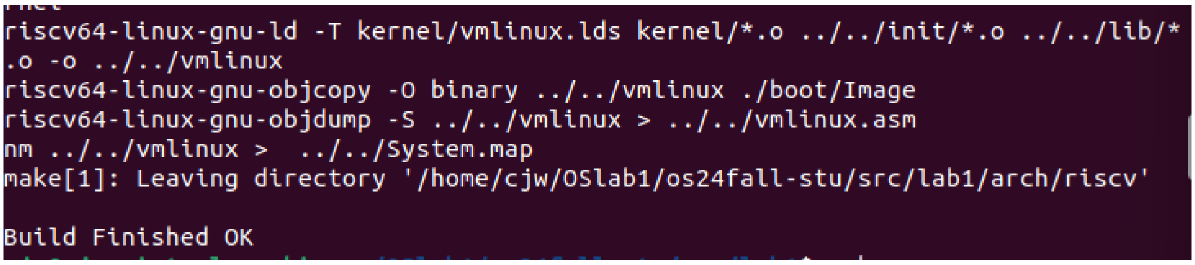
Open | Makefile ~/OSlab1/os24fall-stu/src/lab1/lib

```
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)
```

完成如上，直接照抄了/init/Makefile的文件



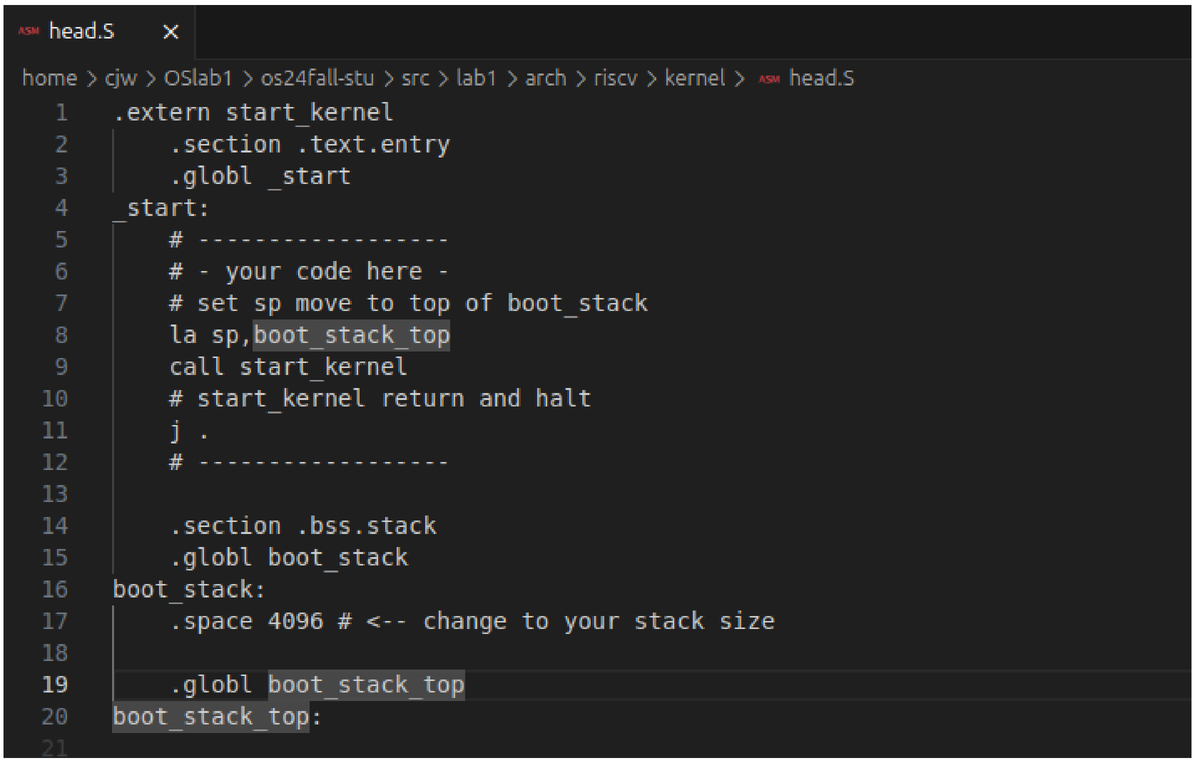
```
make:     [Makefile:19: all] Error 2
cjw@cjw-virtual-machine:~/OSlab1/os24fall-stu/src/lab1$ make
make -C lib clean
make[1]: Entering directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/lib'
make[1]: 'clean' is up to date.
make[1]: Leaving directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/lib'
make -C init clean
make[1]: Entering directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/init'
make[1]: 'clean' is up to date.
make[1]: Leaving directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/init'
make -C arch/riscv clean
make[1]: Entering directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/arch/riscv'
make -C kernel clean
make[2]: Entering directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/arch/riscv/k
ernel'
make[2]: 'clean' is up to date.
make[2]: Leaving directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/arch/riscv/ke
rnel'
make[1]: Leaving directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/arch/riscv'

Clean Finished
make -C lib all
```



```
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 '/home/cjw/OSlab1/os24fall-stu/src/lab1/arch/riscv'

Build Finished OK
```

可以看到不会再提示 Makefile 的错误，而是 C 或汇编代码中的 #error 错误。

2.1.3 编写head.S

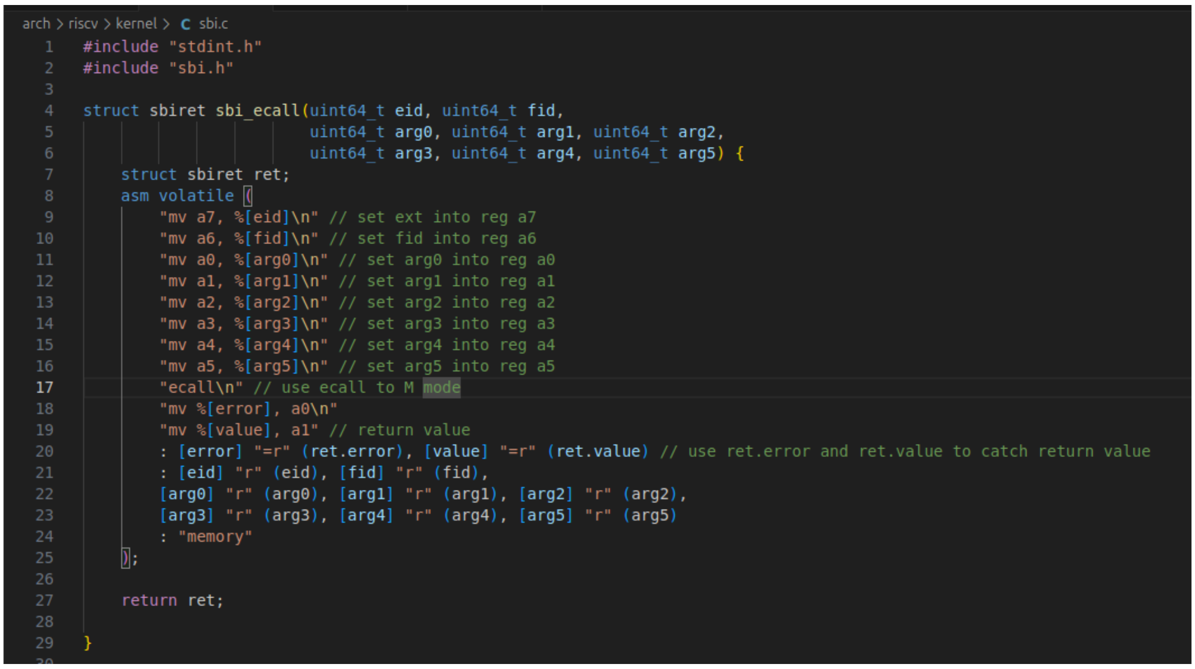


head.S
home > cjw > OSlab1 > os24fall-stu > src > lab1 > arch > riscv > kernel > head.S

```
.extern start_kernel
    .section .text.entry
    .globl _start
_start:
    # ------------------
    # - your code here -
    # set sp move to top of boot_stack
    la sp,boot_stack_top
    call start_kernel
    # start_kernel return and halt
    j .
    # ------------------

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

    .globl boot_stack_top
boot_stack_top:
```

补充如上

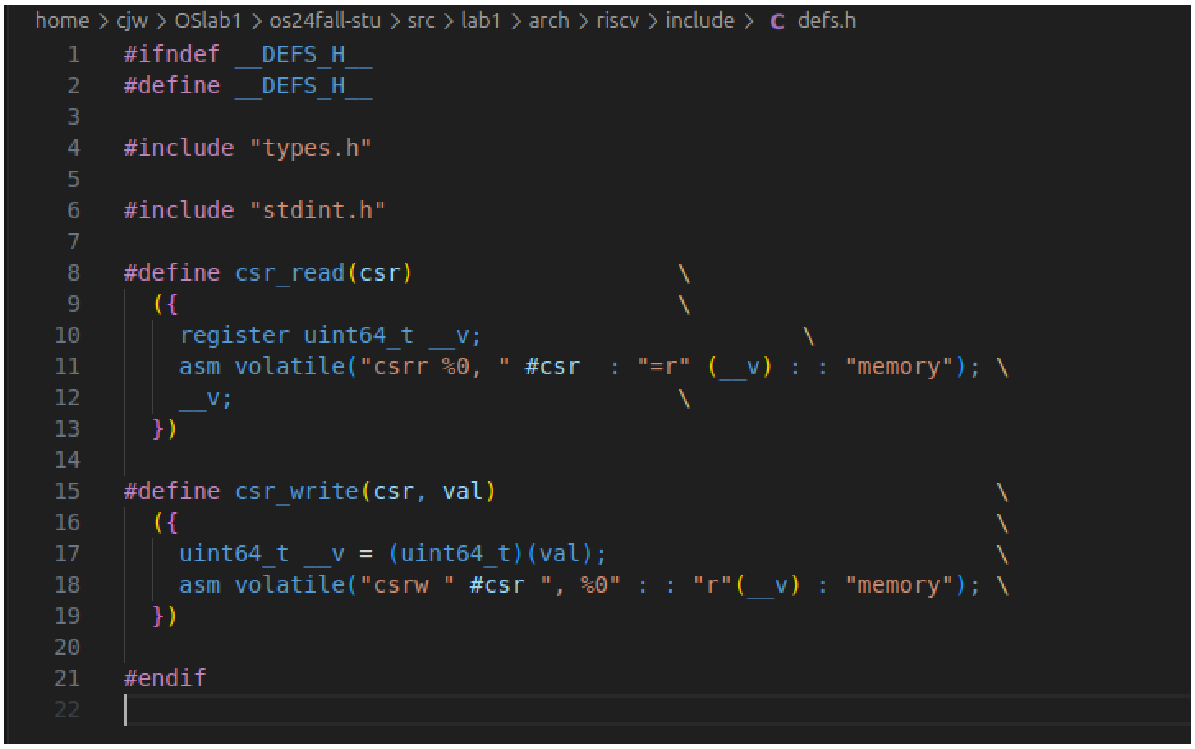
2.1.4 补充sbi.c



```c
#include "stdint.h"
#include "sbi.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, %[eid]\n" // set ext into reg a7
        "mv a6, %[fid]\n" // set fid into reg a6
        "mv a0, %[arg0]\n" // set arg0 into reg a0
        "mv a1, %[arg1]\n" // set arg1 into reg a1
        "mv a2, %[arg2]\n" // set arg2 into reg a2
        "mv a3, %[arg3]\n" // set arg3 into reg a3
        "mv a4, %[arg4]\n" // set arg4 into reg a4
        "mv a5, %[arg5]\n" // set arg5 into reg a5
        "ecall\n" // use ecall to M mode
        "mv %[error], a0\n"
        "mv %[value], a1" // return value
        : [error] "=r" (ret.error), [value] "=r" (ret.value) // use ret.error and ret.value to catch return value
        : [eid] "r" (eid), [fid] "r" (fid),
        [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2),
        [arg3] "r" (arg3), [arg4] "r" (arg4), [arg5] "r" (arg5)
        : "memory"
    );

    return ret;

}
```

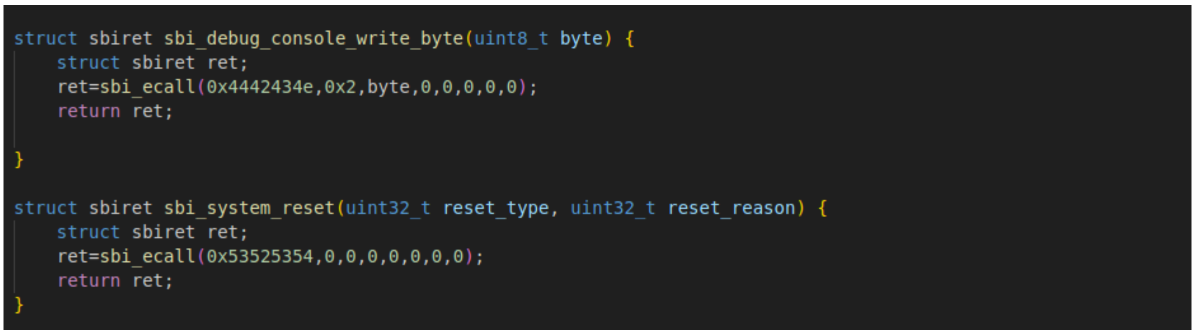
先将参数ext、fid和arg0到arg5放入相应的寄存器中，然后使用ecall指令进入M模式，等待OpenSBI完成相关操作。最后，它将返回的结果存储在ret结构体中，并返回该结构体。

2.1.5 修改defs.h



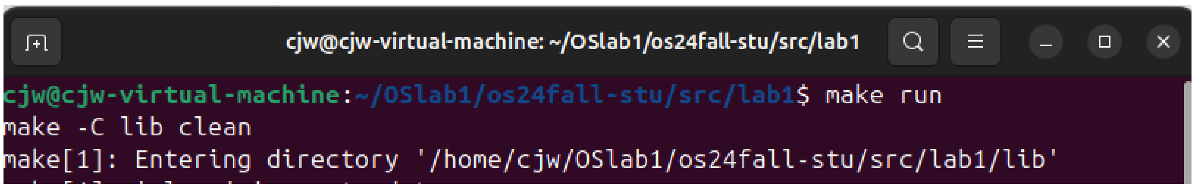
home > cjw > OSlab1 > os24fall-stu > src > lab1 > arch > riscv > include > defs.h

```c
#ifndef __DEFS_H__
#define __DEFS_H__

#include "types.h"

#include "stdint.h"

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

#define csr_write(csr, val)                                     \
  ({                                                            \
    uint64_t __v = (uint64_t)(val);                             \
    asm volatile("csrw " #csr ", %0" : : "r"(__v) : "memory"); \
  })

#endif

```



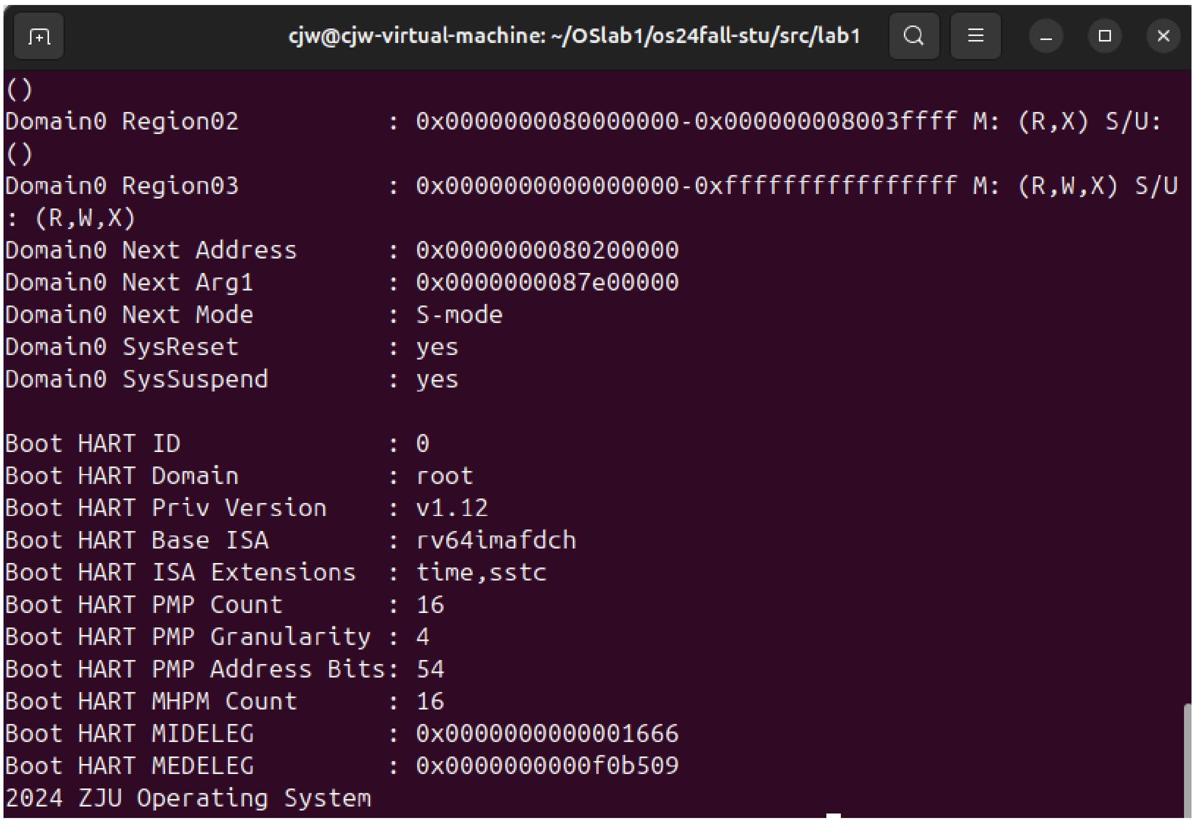
```c
struct sbiret sbi_debug_console_write_byte(uint8_t byte) {
    struct sbiret ret;
    ret=sbi_ecall(0x4442434e,0x2,byte,0,0,0,0,0);
    return ret;

}

struct sbiret sbi_system_reset(uint32_t reset_type, uint32_t reset_reason) {
    struct sbiret ret;
    ret=sbi_ecall(0x53525354,0,0,0,0,0,0,0);
    return ret;
}
```

其中退出函数参数先写了0（因为这里只会用到0），后续我再进行修改。



cjw@cjw-virtual-machine: ~/OSlab1/os24fall-stu/src/lab1

```
cjw@cjw-virtual-machine:~/OSlab1/os24fall-stu/src/lab1$ make run
make -C lib clean
make[1]: Entering directory '/home/cjw/OSlab1/os24fall-stu/src/lab1/lib'
```



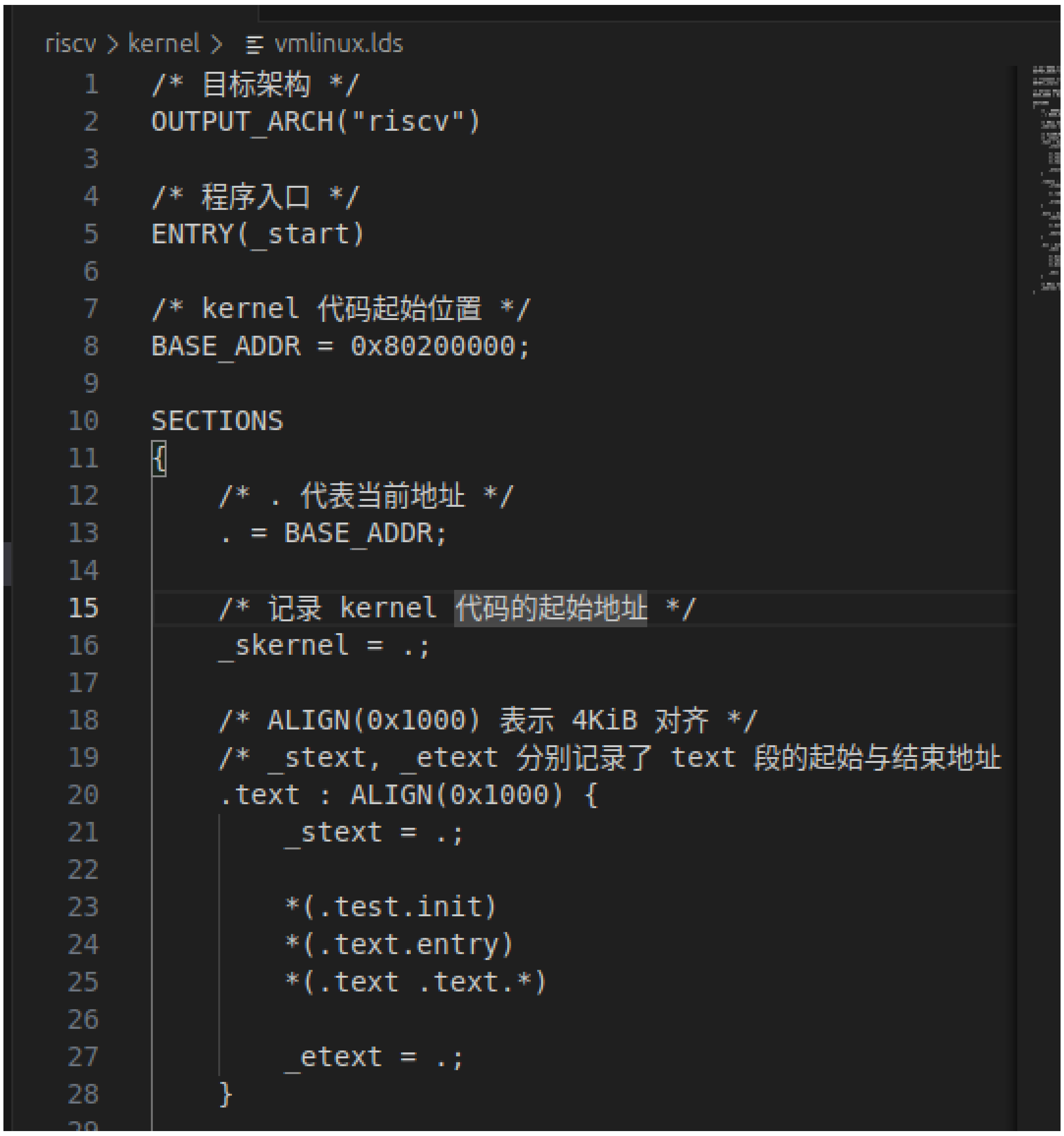
```
()
Domain0 Region02          : 0x0000000080000000-0x000000008003ffff M: (R,X) S/U:
()
Domain0 Region03          : 0x0000000000000000-0xffffffffffffffff M: (R,W,X) S/U
: (R,W,X)
Domain0 Next Address      : 0x0000000080200000
Domain0 Next Arg1         : 0x0000000087e00000
Domain0 Next Mode         : S-mode
Domain0 SysReset          : yes
Domain0 SysSuspend        : yes

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : time,sstc
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
2024 ZJU Operating System
```

这里执行make run，发现正确打印了欢迎信息并正常退出。

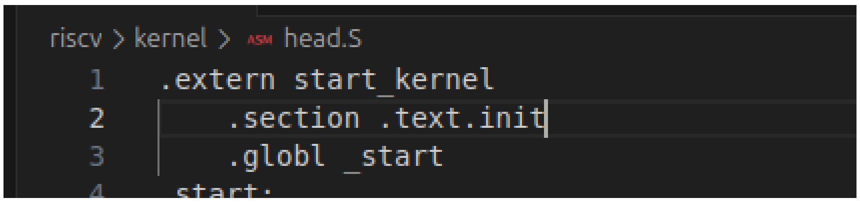
2.2 RV64时钟中断处理

2.2.1 准备工作

修改vmlinux.lds和head.S


riscv > kernel > vmlinux.lds

```
/* 目标架构 */
OUTPUT_ARCH("riscv")

/* 程序入口 */
ENTRY(_start)

/* kernel 代码起始位置 */
BASE_ADDR = 0x80200000;

SECTIONS
{
    /* . 代表当前地址 */
    . = BASE_ADDR;

    /* 记录 kernel 代码的起始地址 */
    _skernel = .;

    /* ALIGN(0x1000) 表示 4KiB 对齐 */
    /* _stext, _etext 分别记录了 text 段的起始与结束地址
    .text : ALIGN(0x1000) {
        _stext = .;

        *(.test.init)
        *(.text.entry)
        *(.text .text.*)

        _etext = .;
    }
```


riscv > kernel > head.S

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

2.2.2 初始化CSR

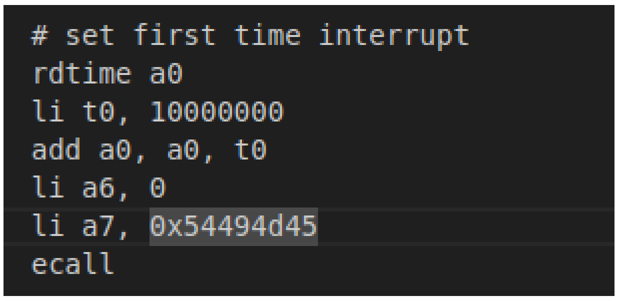
将 *traps*（traps 在 4.3.2 中实现）所表示的地址写入 stvec：

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

开启时钟中断，将sie[STIE]置1

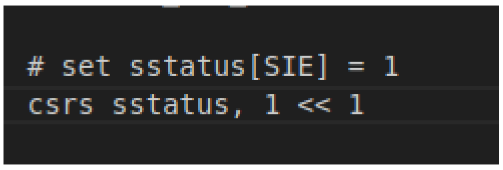
```
# set sie[STIE] = 1
li a0,0x20
csrs sie, a0
```

设置第一次时钟中断


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

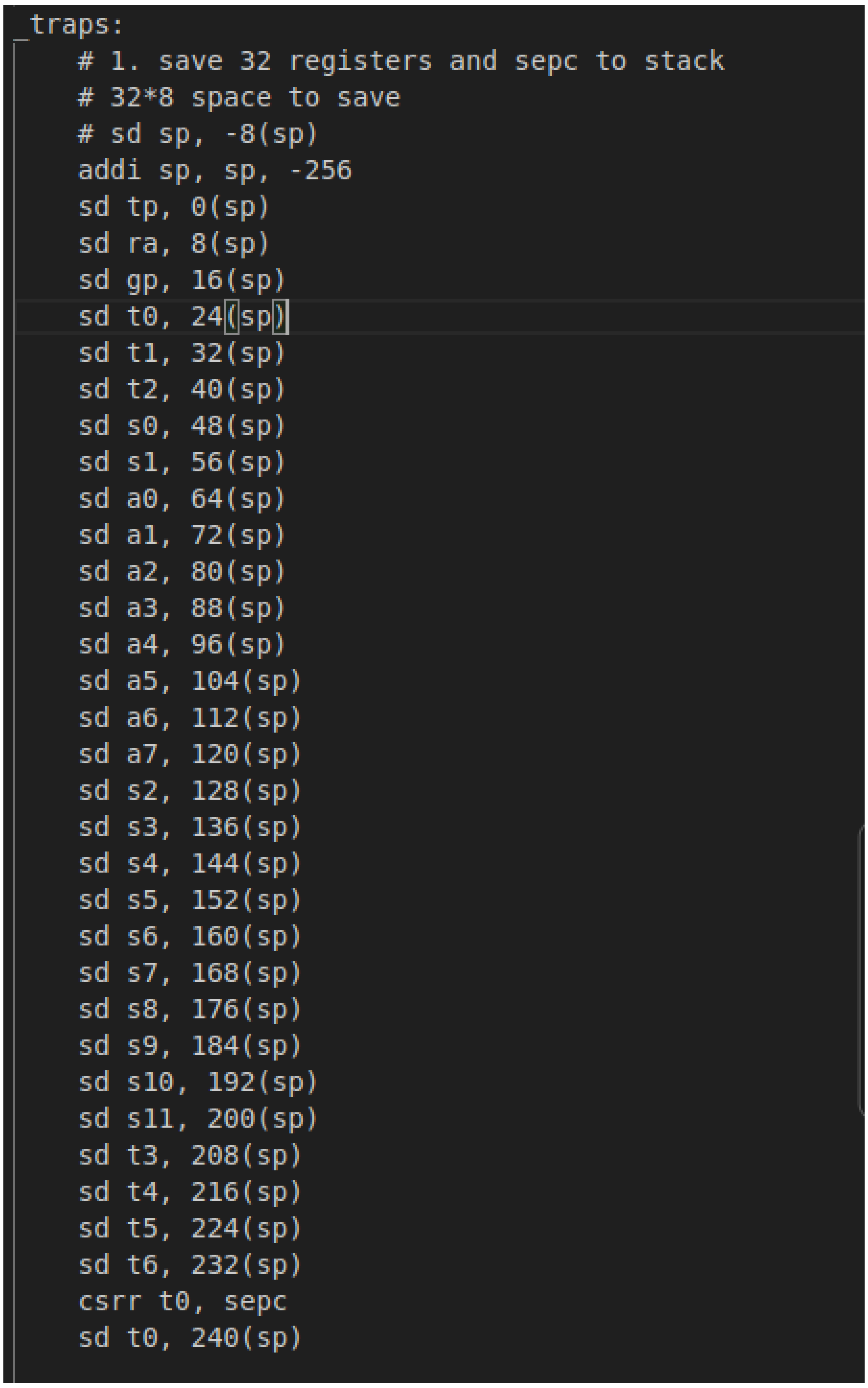
注意这里设置a6为0，a7为0x54494d45，因为调用ecall时直接使用了a6,a7寄存器里存储的值。这两个参数是extension id和function id。

将sstatus[SIE]置1


```
# set sstatus[SIE] = 1
csrs sstatus, 1 << 1
```

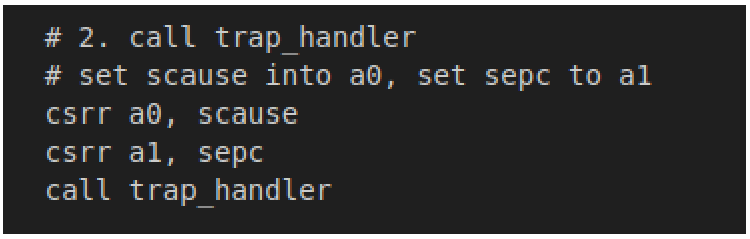
2.2.3 实现上下文切换

在 arch/riscv/kernel/ 目录下添加 entry.S 文件
保存 CPU 的寄存器（上下文）到内存中（栈上）



```
_traps:
    # 1. save 32 registers and sepc to stack
    # 32*8 space to save
    # sd sp, -8(sp)
    addi sp, sp, -256
    sd tp, 0(sp)
    sd ra, 8(sp)
    sd gp, 16(sp)
    sd t0, 24(sp)
    sd t1, 32(sp)
    sd t2, 40(sp)
    sd s0, 48(sp)
    sd s1, 56(sp)
    sd a0, 64(sp)
    sd a1, 72(sp)
    sd a2, 80(sp)
    sd a3, 88(sp)
    sd a4, 96(sp)
    sd a5, 104(sp)
    sd a6, 112(sp)
    sd a7, 120(sp)
    sd s2, 128(sp)
    sd s3, 136(sp)
    sd s4, 144(sp)
    sd s5, 152(sp)
    sd s6, 160(sp)
    sd s7, 168(sp)
    sd s8, 176(sp)
    sd s9, 184(sp)
    sd s10, 192(sp)
    sd s11, 200(sp)
    sd t3, 208(sp)
    sd t4, 216(sp)
    sd t5, 224(sp)
    sd t6, 232(sp)
    csrr t0, sepc
    sd t0, 240(sp)
```

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

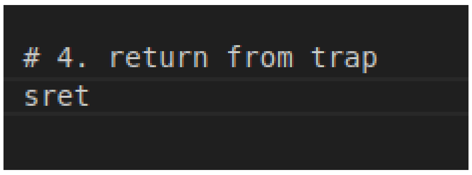


```
# 2. call trap_handler
# set scause into a0, set sepc to a1
csrr a0, scause
csrr a1, sepc
call trap_handler
```

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）

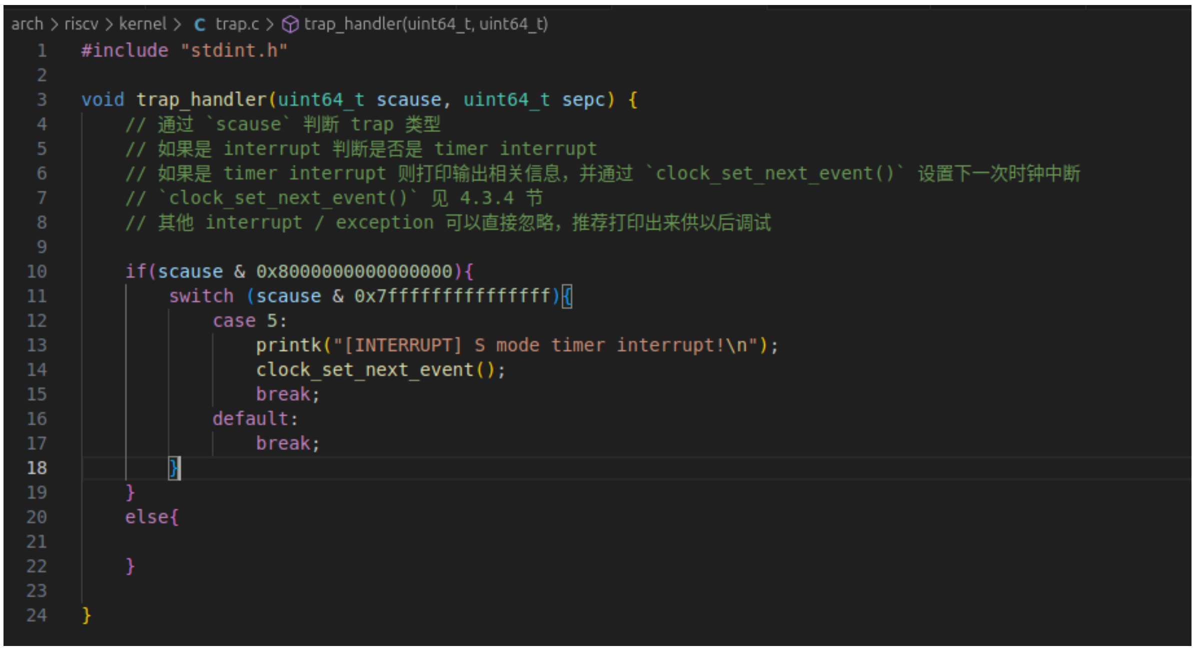
```
# 3. restore sepc and 32 registers (x2(sp) should be restore last) from stack
ld t0, 240(sp)
csrw sepc, t0
ld tp, 0(sp)
ld ra, 8(sp)
ld gp, 16(sp)
ld t0, 24(sp)
ld t1, 32(sp)
ld t2, 40(sp)
ld s0, 48(sp)
ld s1, 56(sp)
ld a0, 64(sp)
ld a1, 72(sp)
ld a2, 80(sp)
ld a3, 88(sp)
ld a4, 96(sp)
ld a5, 104(sp)
ld a6, 112(sp)
ld a7, 120(sp)
ld s2, 128(sp)
ld s3, 136(sp)
ld s4, 144(sp)
ld s5, 152(sp)
ld s6, 160(sp)
ld s7, 168(sp)
ld s8, 176(sp)
ld s9, 184(sp)
ld s10, 192(sp)
ld s11, 200(sp)
ld t3, 208(sp)
ld t4, 216(sp)
ld t5, 224(sp)
ld t6, 232(sp)
addi sp, sp, 256
```

从trap中返回



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

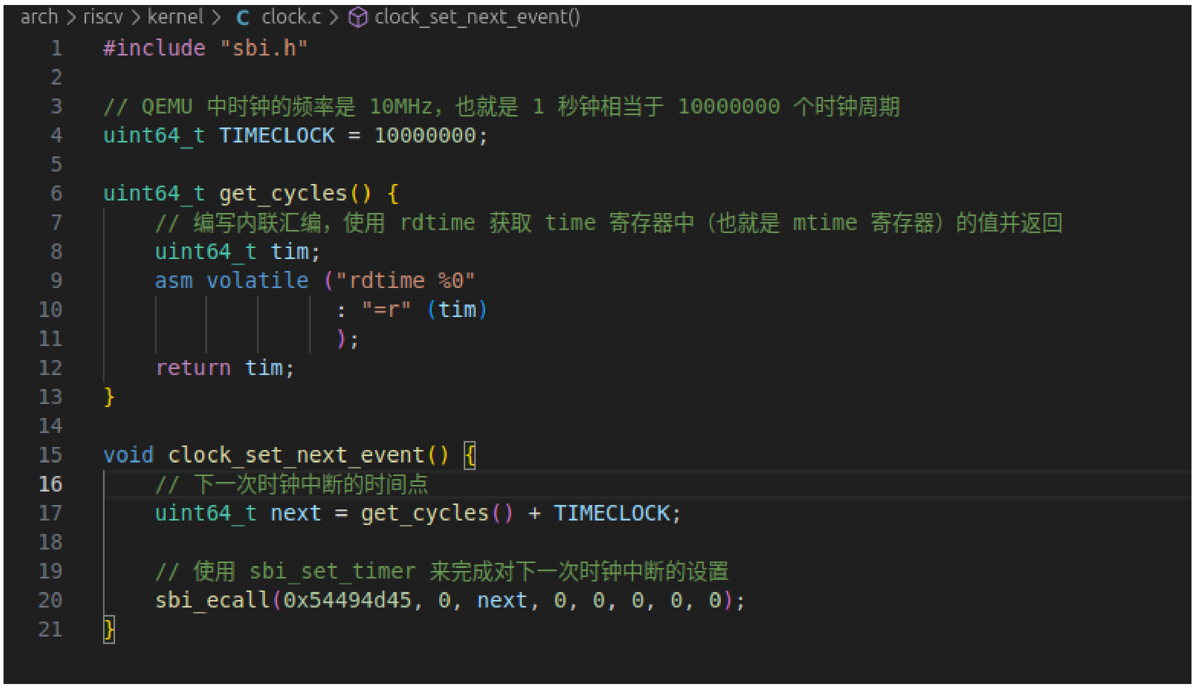
2.2.4 实现trap处理函数


arch > riscv > kernel > C trap.c > trap_handler(uint64_t, uint64_t)

```c
#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

    if(scause & 0x8000000000000000){
        switch (scause & 0x7fffffffffffffff){
            case 5:
                printk("[INTERRUPT] S mode timer interrupt!\n");
                clock_set_next_event();
                break;
            default:
                break;
        }
    }
    else{

    }

}
```

一开始先判断是否是中断，如果是中断，判断是否是5（时钟类型），如果是就调用clock_set_next_event()。

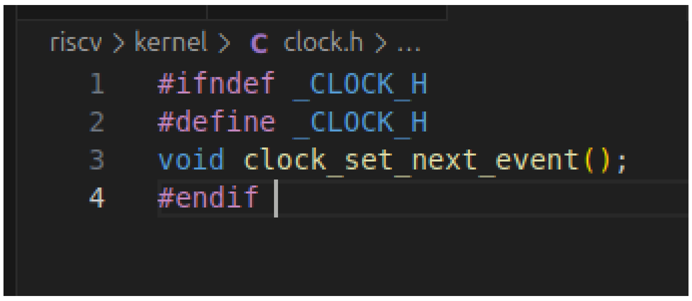
2.2.5 实现时钟中断相关函数



arch > riscv > kernel > clock.c > clock_set_next_event()

```c
#include "sbi.h"

// 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 tim;
    asm volatile ("rdtime %0"
                : "=r" (tim)
                );
    return tim;
}

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
    sbi_ecall(0x54494d45, 0, next, 0, 0, 0, 0, 0);
}
```

注意调用sbi_ecall需要设置第一个参数为0x54494d45，这个是设置时钟相关寄存器的extension id，functuon id是0。



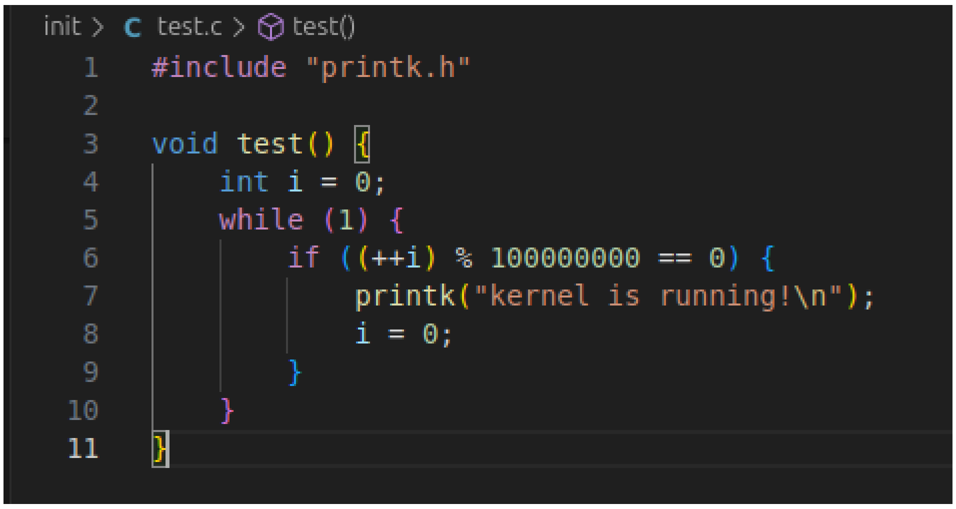
riscv > kernel > clock.h > ...

```c
#ifndef _CLOCK_H
#define _CLOCK_H
void clock_set_next_event();
#endif
```

此外，为了正常调用clock_set_next_event()，我完成了clock.h文件。

2.2.6 修改test函数

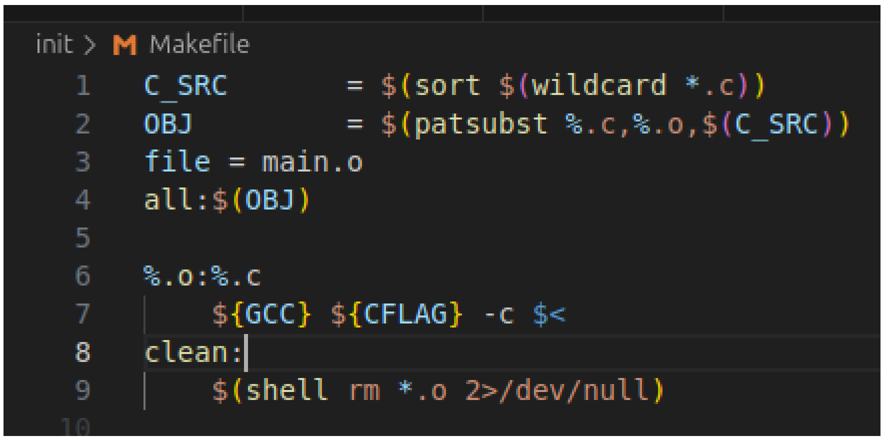
为了让 kernel 持续运行，并使中断更容易观察，将 test.c 中的 test() 函数修改为如下内容：



init > test.c > test()

```c
#include "printk.h"

void test() {
    int i = 0;
    while (1) {
        if ((++i) % 100000000 == 0) {
            printk("kernel is running!\n");
            i = 0;
        }
    }
}
```

我还修改了init文件中的makefile文件，添加了file=main.o(试了一下 不改也行)

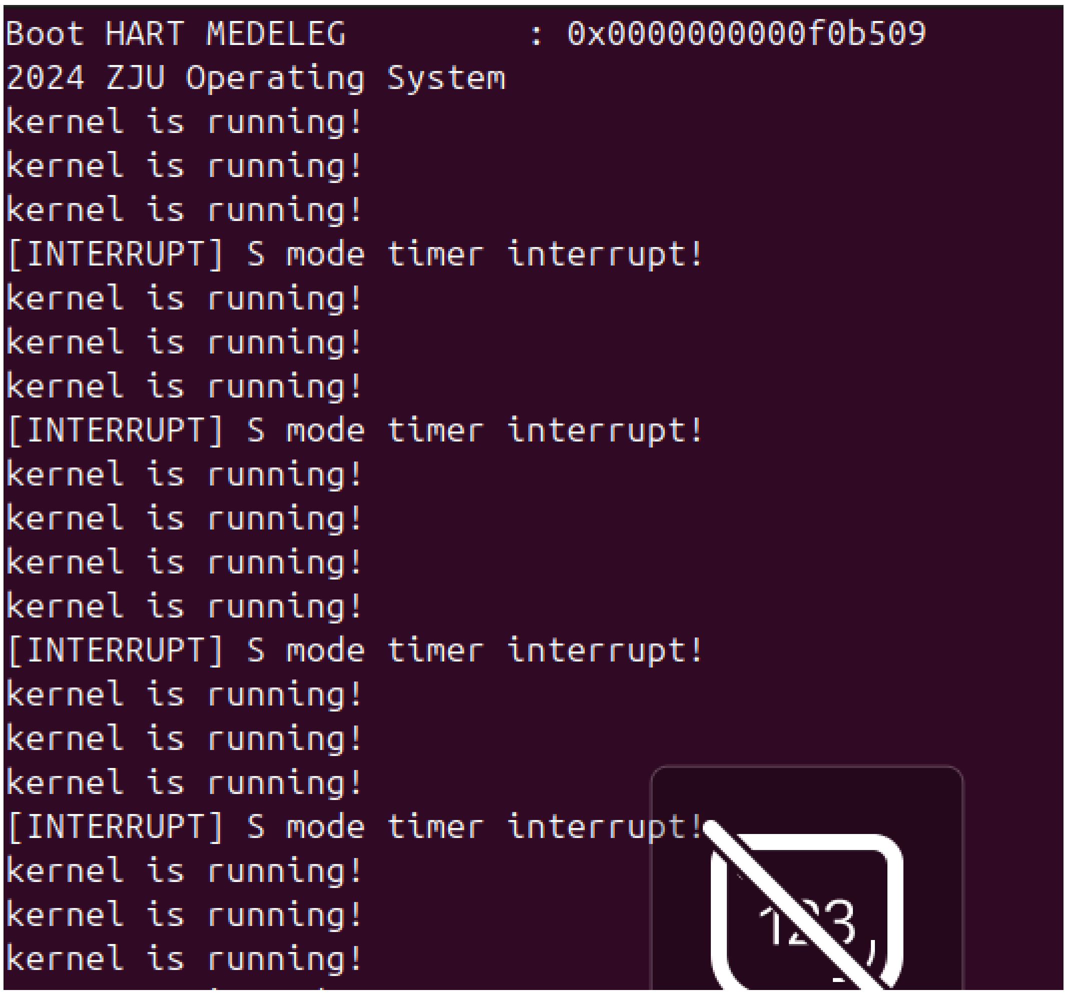


```
init > Makefile
C_SRC       = $(sort $(wildcard *.c))
OBJ         = $(patsubst %.c,%.o,$(C_SRC))
file = main.o
all:$(OBJ)

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

运行结果:

在根目录下使用make run

正常运行。



```
Boot HART MEDELEG          : 0x0000000000f0b509
2024 ZJU Operating System
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
kernel is running!
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
kernel is running!
kernel is running!
kernel is running!
```

三、讨论和心得

本次遇到的第一个问题是版本更新，一开始更新qemu一直没有成功，后来拖了两天，发现ubuntu也要更新，且更新完ubuntu后qemu也更新了，那就算解决了这个问题，本来一直更新很奇怪的。

第二个问题是在第一个小实验里一直输出不了欢迎信息。修改尝试了很多次，最后居然是sbi.c里面mv a0到a5的顺序是有影响的，测试中是a0在最后面就不行，0-5的顺序可以，稍微交换一点也可以。不是很理解这一段，因为讲道理什么顺序应该是没有影响的才对。

第三个问题是最抽象的，因为vmlinux.asm可以看到汇编的执行顺序嘛，我发现我先执行的是trap，然后怎么修改都不对，找了很多地方都没有发现bug。然后我复制了一份一模一样的vmlinux.lds，就好了？！就好了？！真的很离谱。如果不是亲眼所见，真的完全不敢相信的。助教gg肯定觉得是有哪里没有修改，但真的完全一样。

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?

调用过程：

1. 将参数存储到函数能够访问到的位置
2. 使用jal指令跳转到函数开始位置
3. 获取函数需要的局部存储资源，按需保存寄存器
4. 执行函数中的指令
5. 将返回值存储到调用者能够访问到的位置，恢复寄存器，释放局部存储资源
6. 返回调用函数的位置（ret指令）

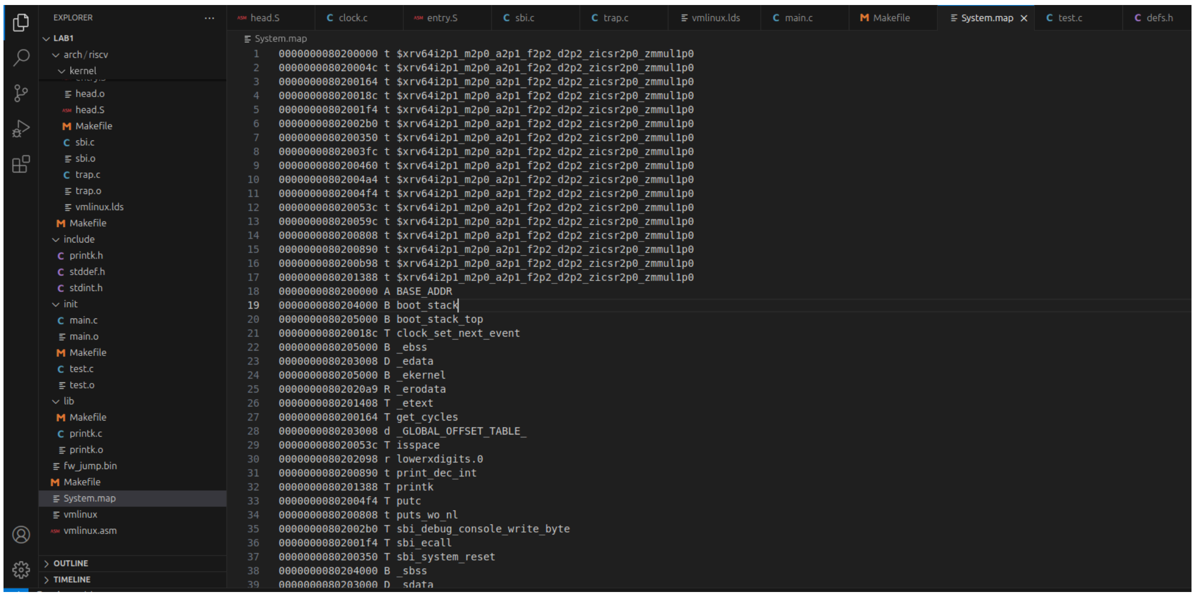
规范：

1. 变量应该尽量放在寄存器而不是内存中
2. 避免频繁地保存和恢复进村器
3. 整型参数保存在a0-a7寄存器中，浮点型参数存在f0到f7寄存器中
4. 整型返回值存在a0寄存器中，浮点返回值存在f0寄存器中
5. 函数调用中其它的寄存器，要么被当做保存寄存器来使用，在函数调用前后值不变，要么被当做淋湿寄存器使用，在函数调用中不保留

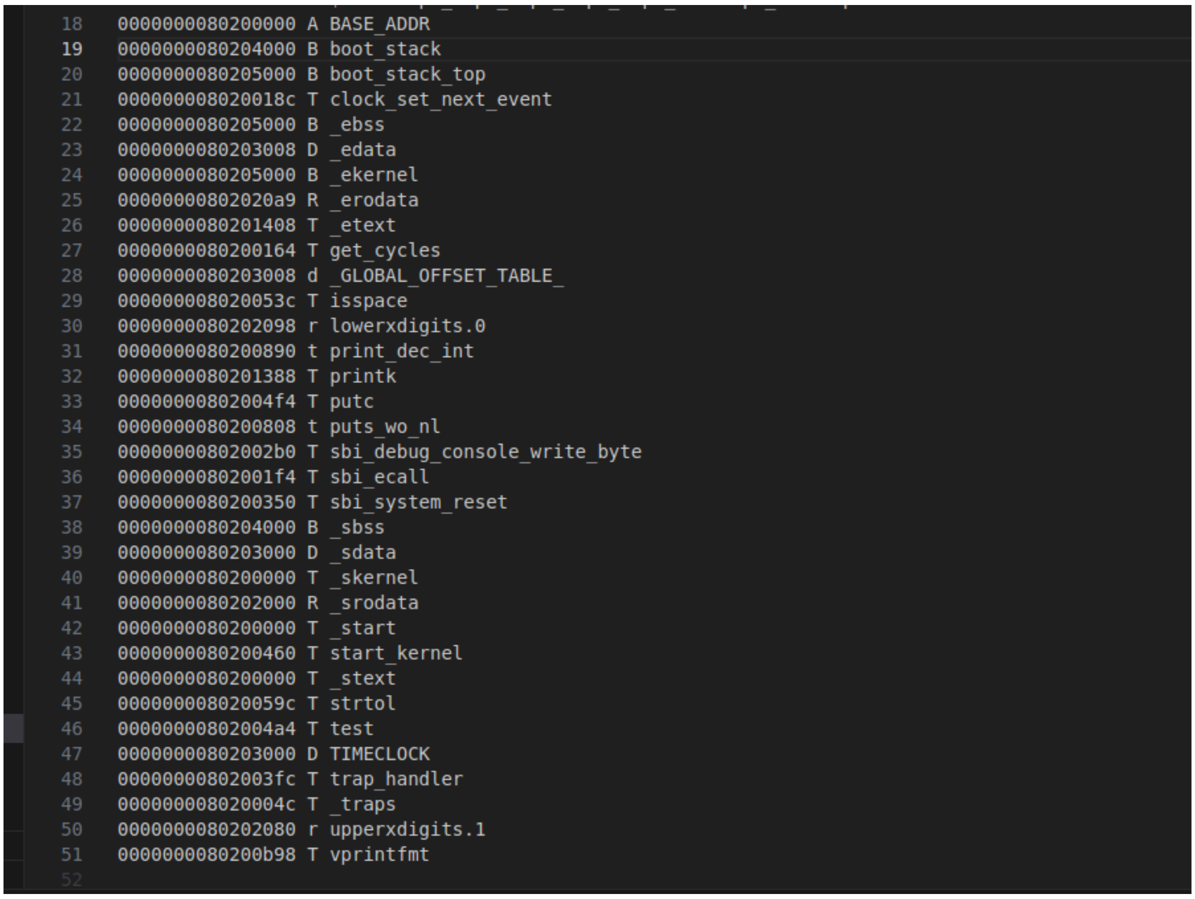
Caller / Callee Saved Register 区别：

1. Caller-Saved寄存器是在函数调用前由调用者保存，如果需要在函数调用之后恢复，则需要调用者自行保存和恢复。用于保存在调用之间不用保留的临时数据。
2. Callee-Saved寄存器是被调用的函数保存和恢复的寄存器，被调用的函数负责保存和恢复这些寄存器。用于保存长期使用的寄存器值。

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



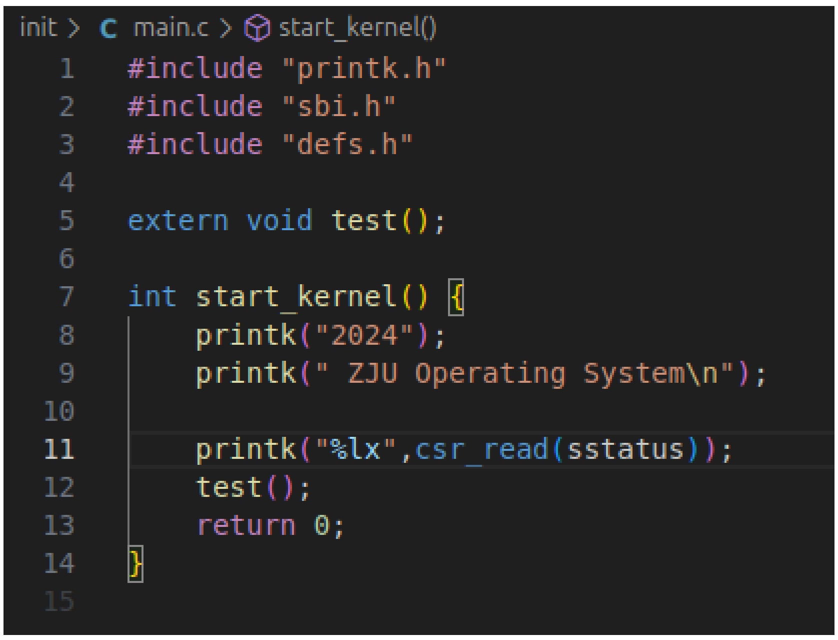
```
0000000080200000 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
000000008020004c t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080200164 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
000000008020018c t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
00000000802001f4 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
00000000802002b0 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080200350 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
00000000802003fc t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080200460 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
00000000802004a4 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
00000000802004f4 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
000000008020053c t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
000000008020059c t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080200808 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080200890 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080200b98 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080201388 t $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_zicsr2p0_zmmul1p0
0000000080200000 A BASE_ADDR
0000000080204000 B boot_stack
0000000080205000 B boot_stack_top
000000008020018c T clock_set_next_event
0000000080205000 B _ebss
0000000080203008 D _edata
0000000080205000 B _ekernel
00000000802020a9 R _erodata
0000000080201408 T _etext
0000000080200164 T get_cycles
0000000080203008 d _GLOBAL_OFFSET_TABLE_
000000008020053c T isspace
0000000080202098 r lowerxdigits.0
0000000080200890 t print_dec_int
0000000080201388 T printk
00000000802004f4 T putc
0000000080200808 t puts_wo_nl
00000000802002b0 T sbi_debug_console_write_byte
00000000802001f4 T sbi_ecall
0000000080200350 T sbi_system_reset
0000000080204000 B _sbss
0000000080203000 D _sdata
```



```
0000000080200000 A BASE_ADDR
0000000080204000 B boot_stack
0000000080205000 B boot_stack_top
000000008020018c T clock_set_next_event
0000000080205000 B _ebss
0000000080203008 D _edata
0000000080205000 B _ekernel
00000000802020a9 R _erodata
0000000080201408 T _etext
0000000080200164 T get_cycles
0000000080203008 d _GLOBAL_OFFSET_TABLE_
000000008020053c T isspace
0000000080202098 r lowerxdigits.0
0000000080200890 t print_dec_int
0000000080201388 T printk
00000000802004f4 T putc
0000000080200808 t puts_wo_nl
00000000802002b0 T sbi_debug_console_write_byte
00000000802001f4 T sbi_ecall
0000000080200350 T sbi_system_reset
0000000080204000 B _sbss
0000000080203000 D _sdata
0000000080200000 T _skernel
0000000080202000 R _srodata
0000000080200000 T _start
0000000080200460 T start_kernel
0000000080200000 T _stext
000000008020059c T strtol
00000000802004a4 T test
0000000080203000 D TIMECLOCK
00000000802003fc T trap_handler
000000008020004c T _traps
0000000080202080 r upperxdigits.1
0000000080200b98 T vprintfmt
```

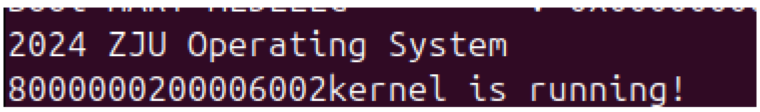
可以查看T,B,D,R变量的地址和sbi_ecall,start_kernel等符号的值

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。



init > main.c > start_kernel()

```c
#include "printk.h"
#include "sbi.h"
#include "defs.h"

extern void test();

int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");

    printk("%lx",csr_read(sstatus));
    test();
    return 0;
}

```



```
2024 ZJU Operating System
8000000200006002kernel is running!
```

sstatus值为80000002 00006002 由手册可知SIE目前是1。

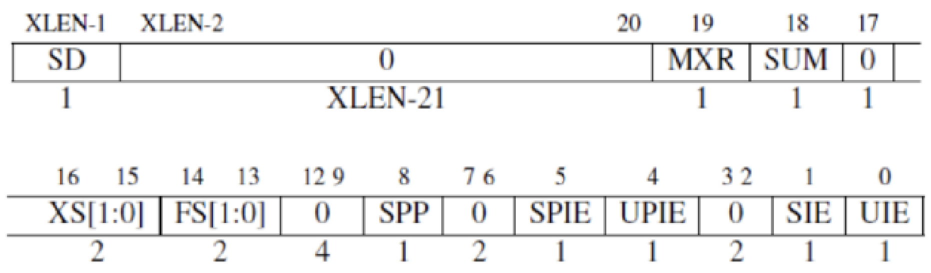
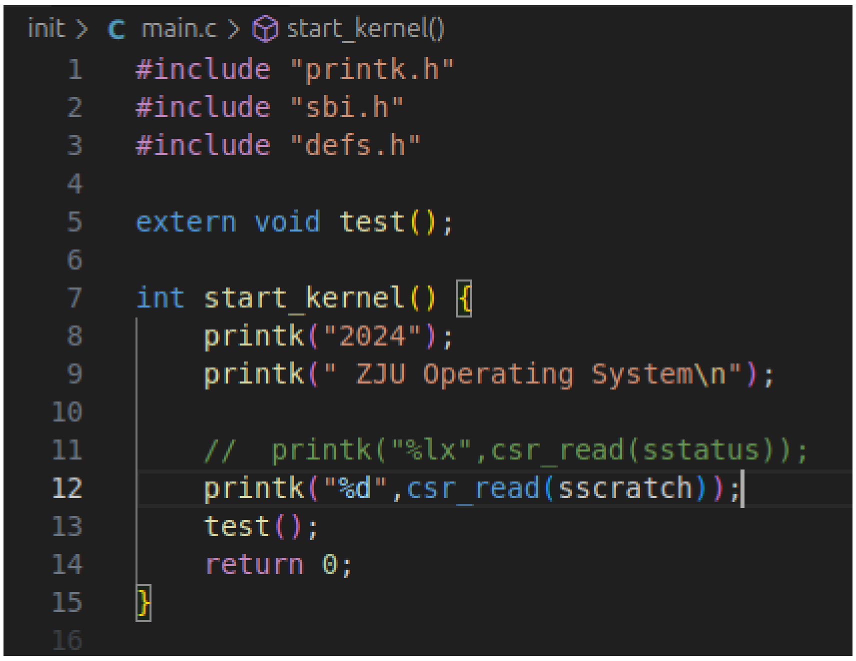


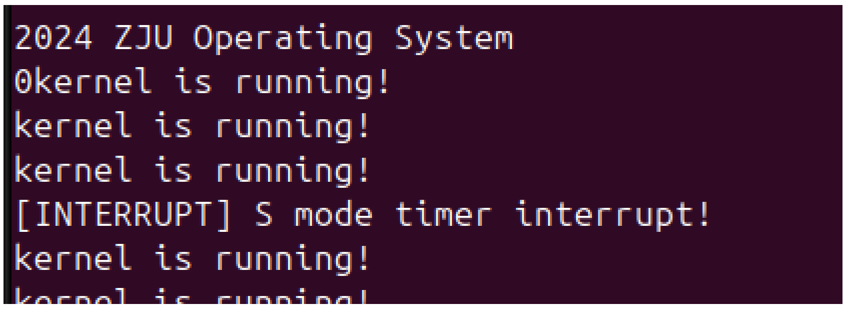
图 10.9：sstatus CSR。sstatus 是 mstatus（图 10.4）的一个子集，因此它们的布局类似。SIE 和 SPIE 中分别保存了当前的和异常发生之前的中断使能，类似于 mstatus 中的 MIE 和 MPIE。RV32 的 XLEN 为 32，RV64 为 40。（来自[Waterman and Asanovic 2017]中的图 4.2；有关其他域的说明请参见该文档的第 4.1 节。）

由手册可知，SIE=1时中断启用，进入中断时SIE=0防止其他中断产生，这时SPP=1表明此时与普通的禁止中断状态不同

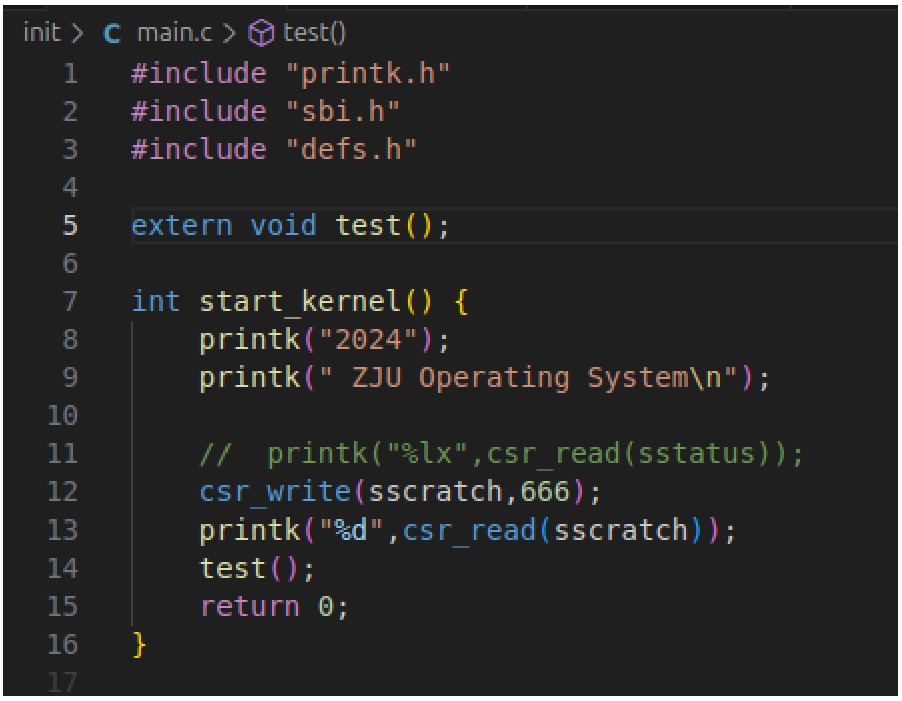
4.用 `csr_write` 宏向 `sscratch` 寄存器写入数据，并验证是否写入成功并截图。



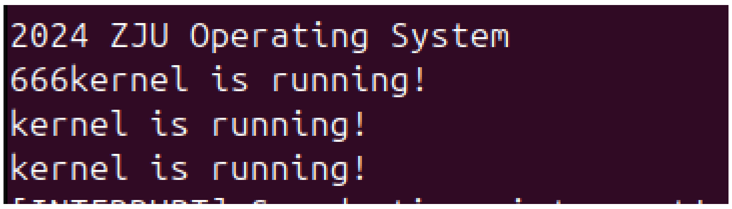
```c
#include "printk.h"
#include "sbi.h"
#include "defs.h"

extern void test();

int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");

    //  printk("%lx",csr_read(sstatus));
    printk("%d",csr_read(sscratch));
    test();
    return 0;
}
```



```
2024 ZJU Operating System
0kernel is running!
kernel is running!
kernel is running!
[INTERRUPT] S mode timer interrupt!
kernel is running!
kernel is running!
```


init > main.c > test()

```c
#include "printk.h"
#include "sbi.h"
#include "defs.h"

extern void test();

int start_kernel() {
    printk("2024");
    printk(" ZJU Operating System\n");

    //  printk("%lx",csr_read(sstatus));
    csr_write(sscratch,666);
    printk("%d",csr_read(sscratch));
    test();
    return 0;
}
```



```
2024 ZJU Operating System
666kernel is running!
kernel is running!
kernel is running!
```

说明写入成功。

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。

1. 安装交叉编译工具（在linux-6.11-rc7目录下）：

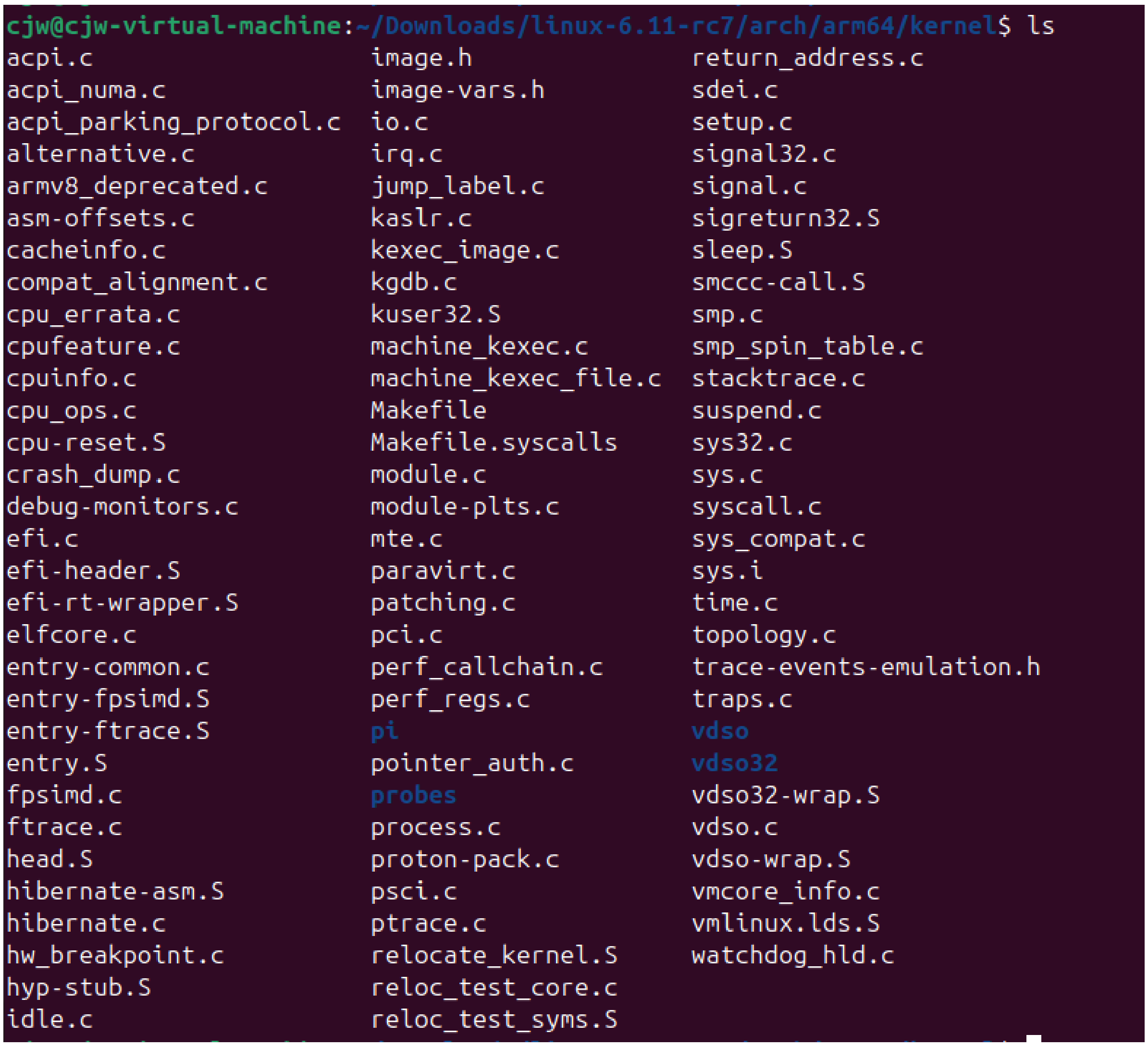
   `sudo apt install gcc-aarch64-linux-gnu`

2. 生成config

   `make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- defconfig`

3. make获得文件sys.i

   `make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/arm64/kernel/sys.i`



```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/arm64/kernel$ ls
acpi.c                    image.h                return_address.c
acpi_numa.c               image-vars.h           sdei.c
acpi_parking_protocol.c   io.c                   setup.c
alternative.c             irq.c                  signal32.c
armv8_deprecated.c        jump_label.c           signal.c
asm-offsets.c             kaslr.c                sigreturn32.S
cacheinfo.c               kexec_image.c          sleep.S
compat_alignment.c        kgdb.c                 smccc-call.S
cpu_errata.c              kuser32.S              smp.c
cpufeature.c              machine_kexec.c        smp_spin_table.c
cpuinfo.c                 machine_kexec_file.c   stacktrace.c
cpu_ops.c                 Makefile               suspend.c
cpu-reset.S               Makefile.syscalls      sys32.c
crash_dump.c              module.c               sys.c
debug-monitors.c          module-plts.c          syscall.c
efi.c                     mte.c                  sys_compat.c
efi-header.S              paravirt.c             sys.i
efi-rt-wrapper.S          patching.c             time.c
elfcore.c                 pci.c                  topology.c
entry-common.c            perf_callchain.c       trace-events-emulation.h
entry-fpsimd.S            perf_regs.c            traps.c
entry-ftrace.S            pi                     vdso
entry.S                   pointer_auth.c         vdso32
fpsimd.c                  probes                 vdso32-wrap.S
ftrace.c                  process.c              vdso.c
head.S                    proton-pack.c          vdso-wrap.S
hibernate-asm.S           psci.c                 vmcore_info.c
hibernate.c               ptrace.c               vmlinux.lds.S
hw_breakpoint.c           relocate_kernel.S      watchdog_hld.c
hyp-stub.S                reloc_test_core.c
idle.c                    reloc_test_syms.S
```

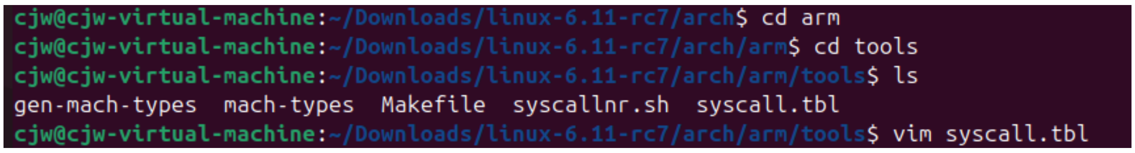
可以看到出现sys.i

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

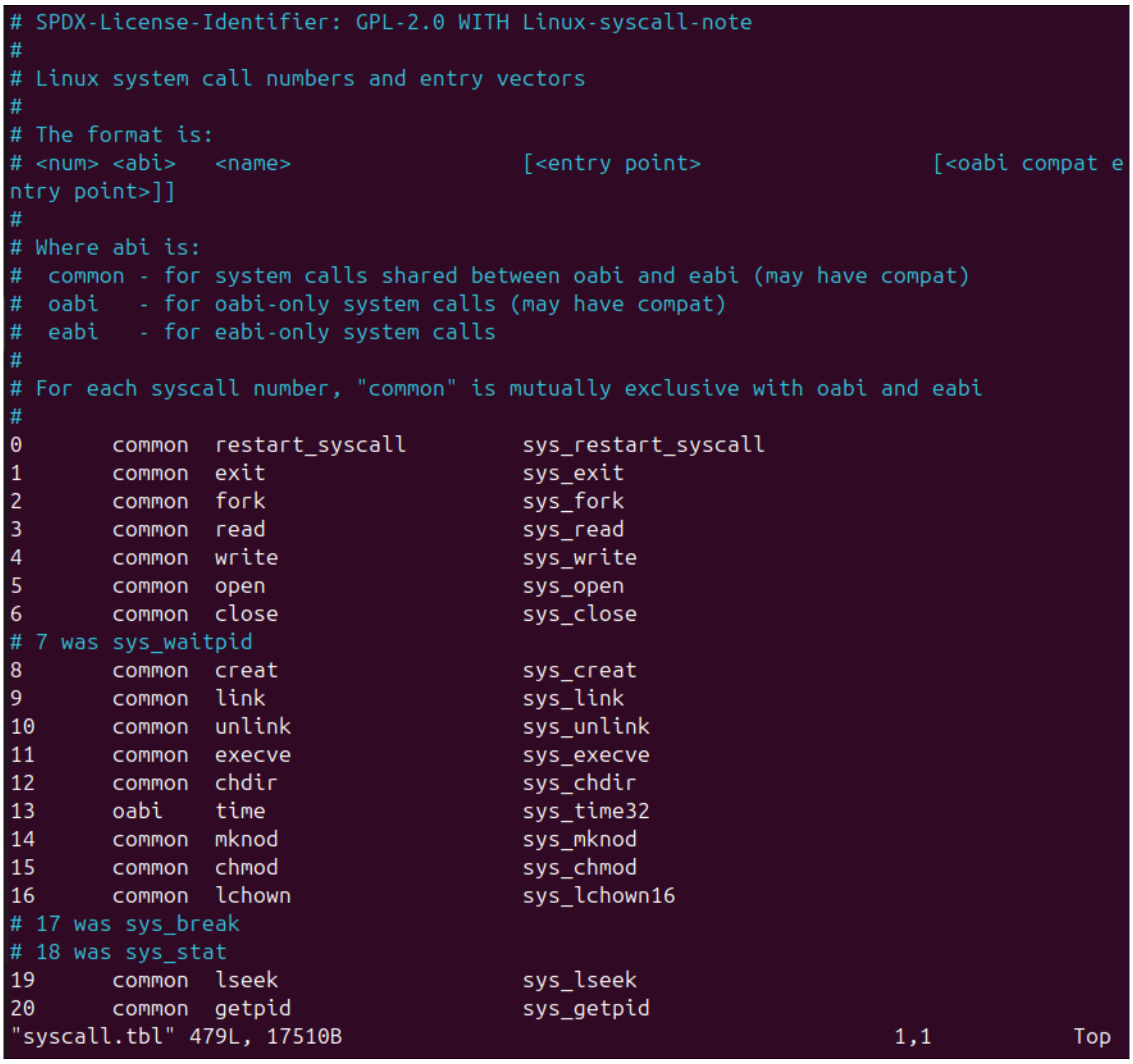
- **请列出源代码文件，展示完整的系统调用表（宏展开后），每一步都需要截图。**

1. ARM32

   进入linux-6.11-rc7/arch/arm/tools 使用vim syscall.tbl查看

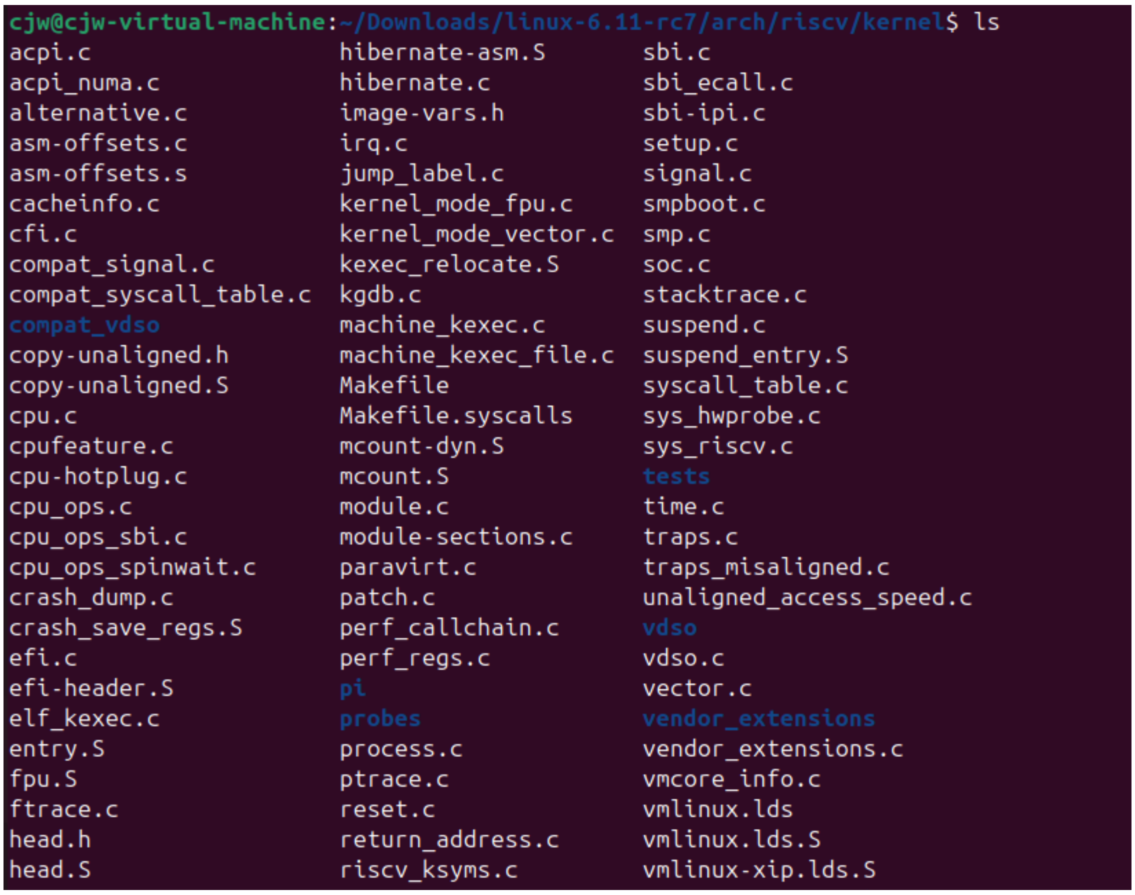


```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch$ cd arm
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/arm$ cd tools
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/arm/tools$ ls
gen-mach-types  mach-types  Makefile  syscallnr.sh  syscall.tbl
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/arm/tools$ vim syscall.tbl
```



```
# 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 e
ntry 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                                            1,1           Top
```

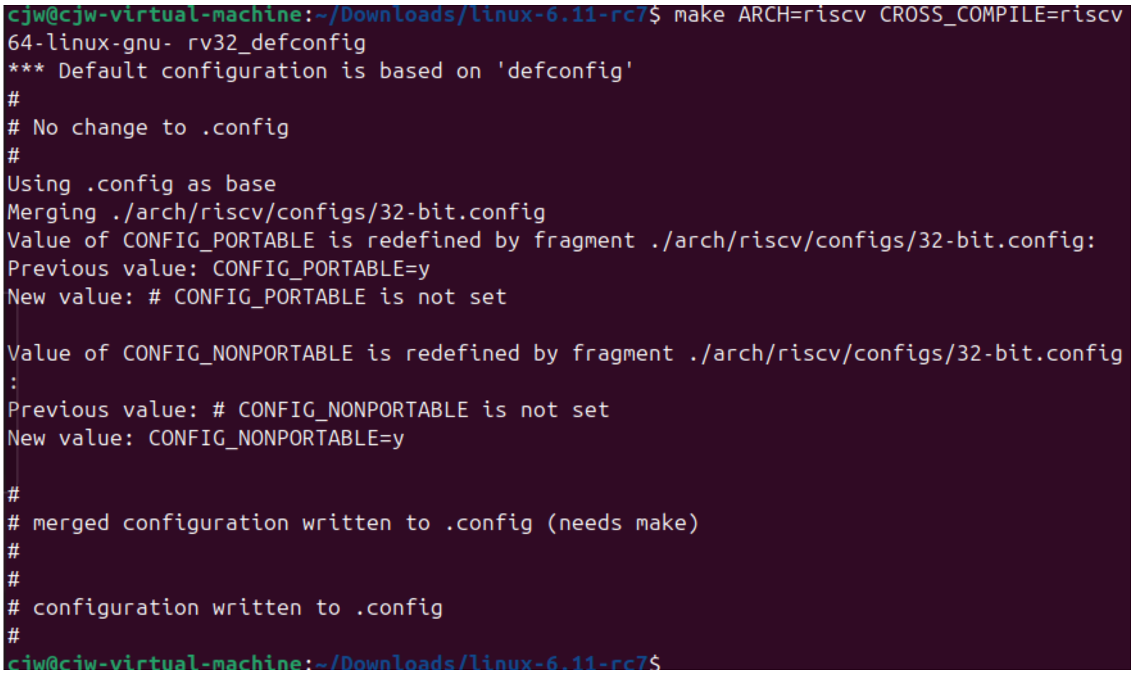
2. RV32



```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/riscv/kernel$ ls
acpi.c                  hibernate-asm.S         sbi.c
acpi_numa.c             hibernate.c             sbi_ecall.c
alternative.c           image-vars.h            sbi-ipi.c
asm-offsets.c           irq.c                   setup.c
asm-offsets.s           jump_label.c            signal.c
cacheinfo.c             kernel_mode_fpu.c       smpboot.c
cfi.c                   kernel_mode_vector.c    smp.c
compat_signal.c         kexec_relocate.S        soc.c
compat_syscall_table.c  kgdb.c                  stacktrace.c
compat_vdso             machine_kexec.c         suspend.c
copy-unaligned.h        machine_kexec_file.c    suspend_entry.S
copy-unaligned.S        Makefile                syscall_table.c
cpu.c                   Makefile.syscalls       sys_hwprobe.c
cpufeature.c            mcount-dyn.S            sys_riscv.c
cpu-hotplug.c           mcount.S                tests
cpu_ops.c               module.c                time.c
cpu_ops_sbi.c           module-sections.c       traps.c
cpu_ops_spinwait.c      paravirt.c              traps_misaligned.c
crash_dump.c            patch.c                 unaligned_access_speed.c
crash_save_regs.S       perf_callchain.c        vdso
efi.c                   perf_regs.c             vdso.c
efi-header.S            pi                      vector.c
elf_kexec.c             probes                  vendor_extensions
entry.S                 process.c               vendor_extensions.c
fpu.S                   ptrace.c                vmcore_info.c
ftrace.c                reset.c                 vmlinux.lds
head.h                  return_address.c        vmlinux.lds.S
head.S                  riscv_ksyms.c           vmlinux-xip.lds.S
```

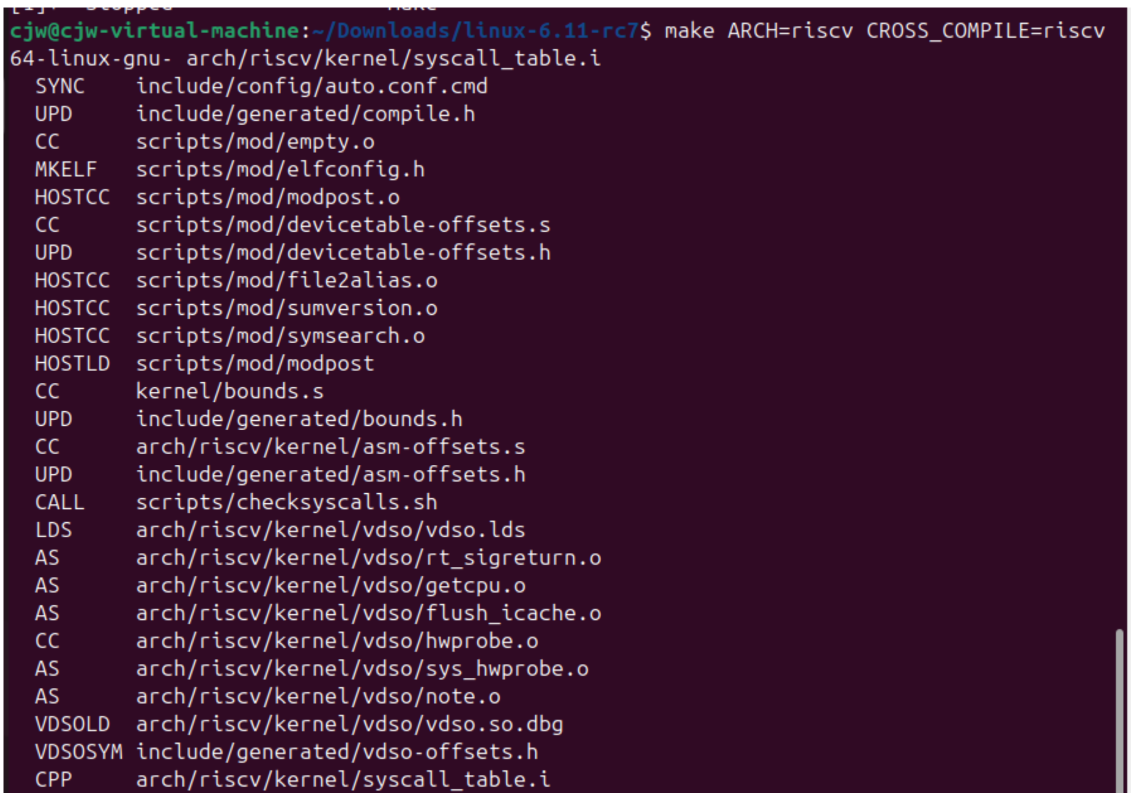
进入文件夹以后找不到syscall_table.i。

先进行defconfig编译，需要注意，使用rv32_defconfig



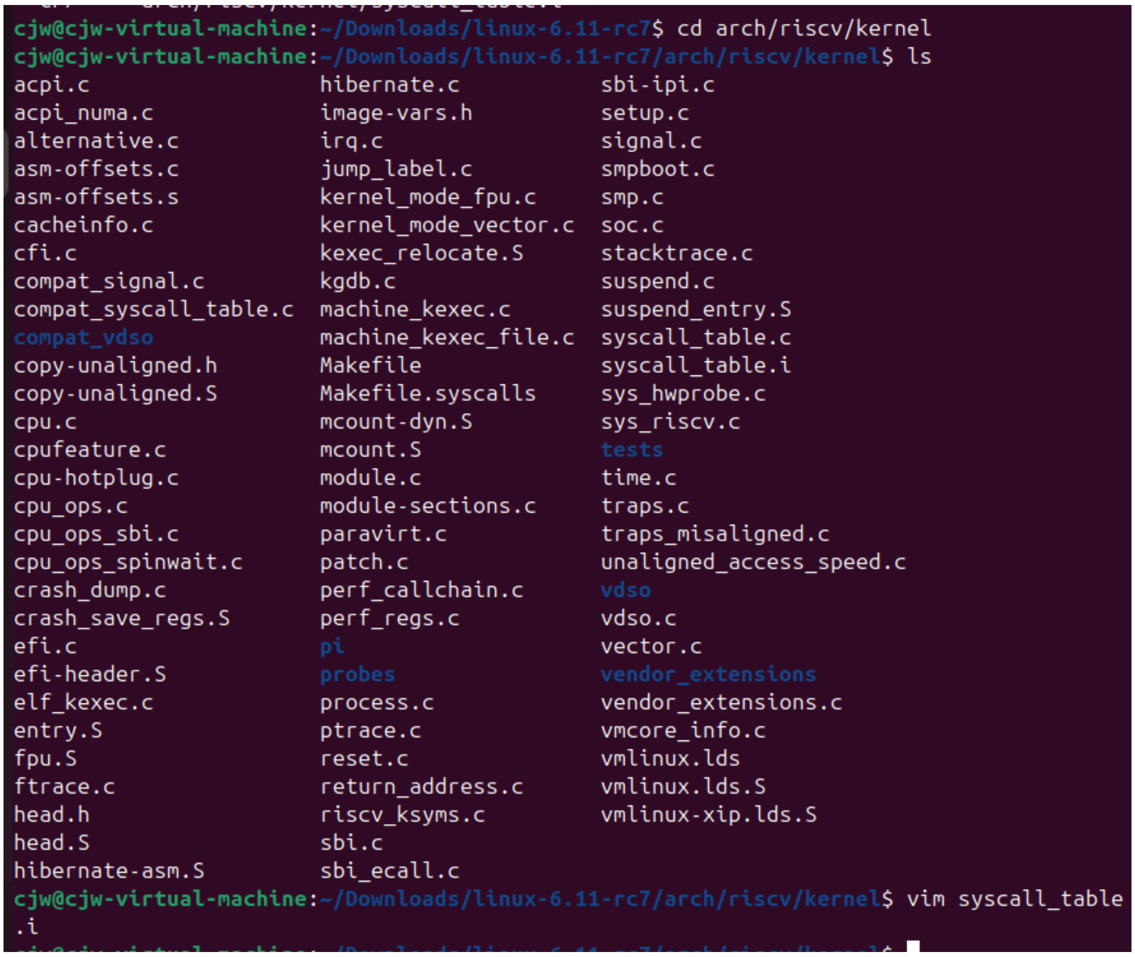
```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7$ make ARCH=riscv CROSS_COMPILE=riscv
64-linux-gnu- rv32_defconfig
*** Default configuration is based on 'defconfig'
#
# No change to .config
#
Using .config as base
Merging ./arch/riscv/configs/32-bit.config
Value of CONFIG_PORTABLE is redefined 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
#
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7$
```

进行make

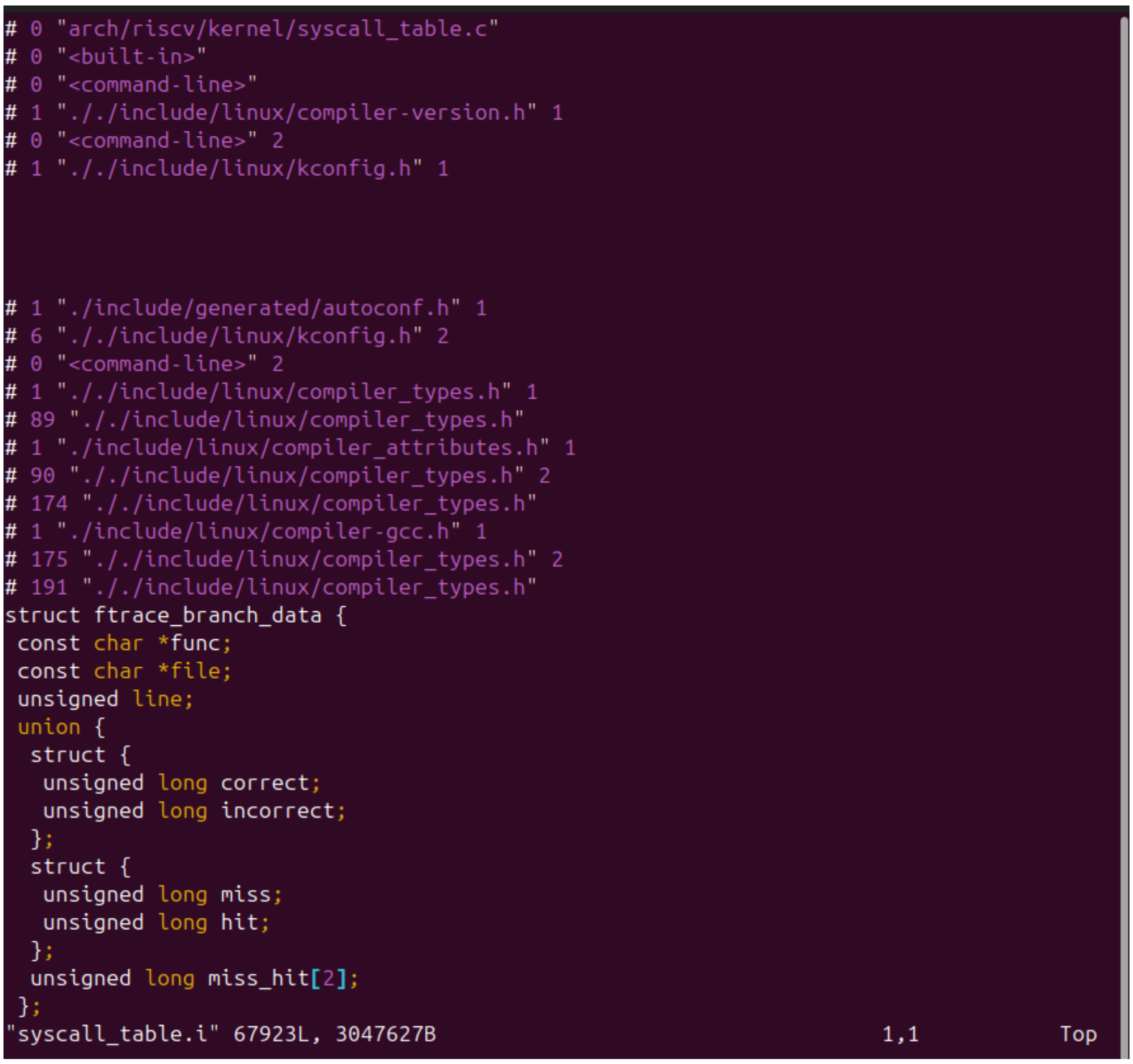


```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7$ make ARCH=riscv CROSS_COMPILE=riscv
64-linux-gnu- arch/riscv/kernel/syscall_table.i
  SYNC    include/config/auto.conf.cmd
  UPD     include/generated/compile.h
  CC      scripts/mod/empty.o
  MKELF   scripts/mod/elfconfig.h
  HOSTCC  scripts/mod/modpost.o
  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
  VDSOLD  arch/riscv/kernel/vdso/vdso.so.dbg
  VDSOSYM include/generated/vdso-offsets.h
  CPP     arch/riscv/kernel/syscall_table.i
```

可以发现出现syscall_table.i，可以通过vim来查看



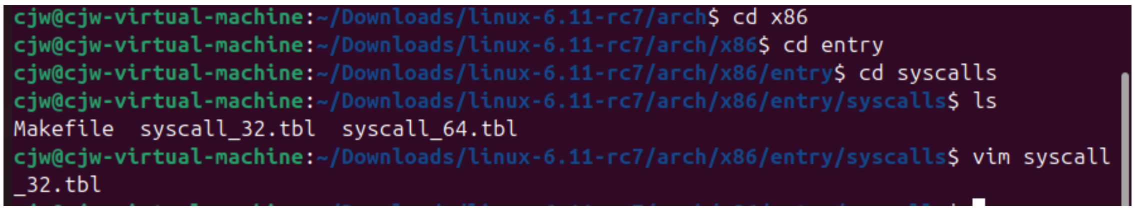
```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7$ cd arch/riscv/kernel
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/riscv/kernel$ ls
acpi.c                  hibernate.c           sbi-ipi.c
acpi_numa.c             image-vars.h          setup.c
alternative.c           irq.c                 signal.c
asm-offsets.c           jump_label.c          smpboot.c
asm-offsets.s           kernel_mode_fpu.c     smp.c
cacheinfo.c             kernel_mode_vector.c  soc.c
cfi.c                   kexec_relocate.S      stacktrace.c
compat_signal.c         kgdb.c                suspend.c
compat_syscall_table.c  machine_kexec.c       suspend_entry.S
compat_vdso             machine_kexec_file.c  syscall_table.c
copy-unaligned.h        Makefile              syscall_table.i
copy-unaligned.S        Makefile.syscalls     sys_hwprobe.c
cpu.c                   mcount-dyn.S          sys_riscv.c
cpufeature.c            mcount.S              tests
cpu-hotplug.c           module.c              time.c
cpu_ops.c               module-sections.c     traps.c
cpu_ops_sbi.c           paravirt.c            traps_misaligned.c
cpu_ops_spinwait.c      patch.c               unaligned_access_speed.c
crash_dump.c            perf_callchain.c      vdso
crash_save_regs.S       perf_regs.c           vdso.c
efi.c                   pi                    vector.c
efi-header.S            probes                vendor_extensions
elf_kexec.c             process.c             vendor_extensions.c
entry.S                 ptrace.c              vmcore_info.c
fpu.S                   reset.c               vmlinux.lds
ftrace.c                return_address.c      vmlinux.lds.S
head.h                  riscv_ksyms.c         vmlinux-xip.lds.S
head.S                  sbi.c
hibernate-asm.S         sbi_ecall.c
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/riscv/kernel$ vim syscall_table
.i
```



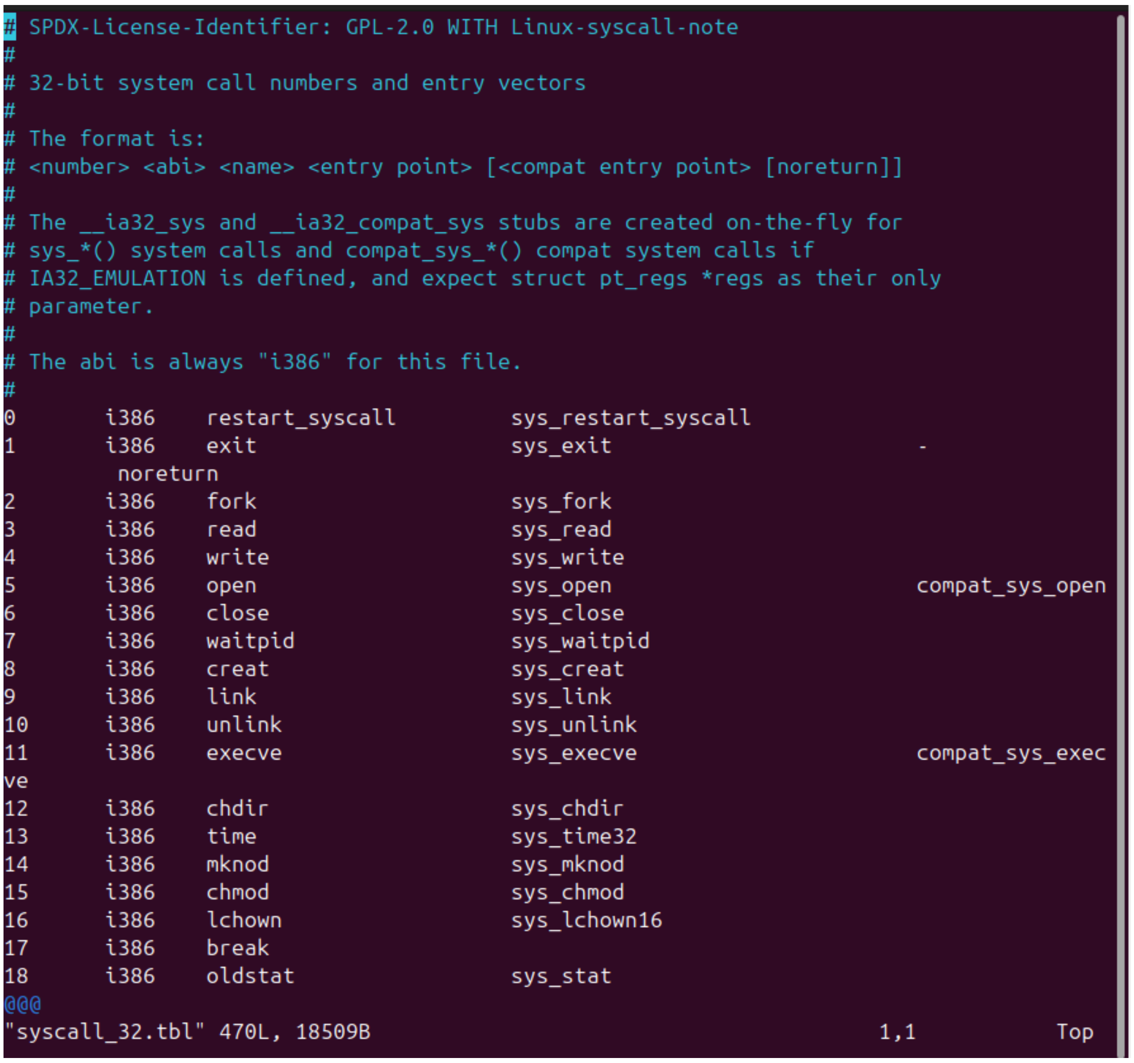
```
# 0 "arch/riscv/kernel/syscall_table.c"
# 0 "<built-in>"
# 0 "<command-line>"
# 1 "././include/linux/compiler-version.h" 1
# 0 "<command-line>" 2
# 1 "././include/linux/kconfig.h" 1




# 1 "./include/generated/autoconf.h" 1
# 6 "././include/linux/kconfig.h" 2
# 0 "<command-line>" 2
# 1 "././include/linux/compiler_types.h" 1
# 89 "././include/linux/compiler_types.h"
# 1 "./include/linux/compiler_attributes.h" 1
# 90 "././include/linux/compiler_types.h" 2
# 174 "././include/linux/compiler_types.h"
# 1 "./include/linux/compiler-gcc.h" 1
# 175 "././include/linux/compiler_types.h" 2
# 191 "././include/linux/compiler_types.h"
struct ftrace_branch_data {
 const char *func;
 const char *file;
 unsigned line;
 union {
  struct {
   unsigned long correct;
   unsigned long incorrect;
  };
  struct {
   unsigned long miss;
   unsigned long hit;
  };
  unsigned long miss_hit[2];
 };
"syscall_table.i" 67923L, 3047627B                          1,1           Top
```

3. RV64

直接通过vim查看

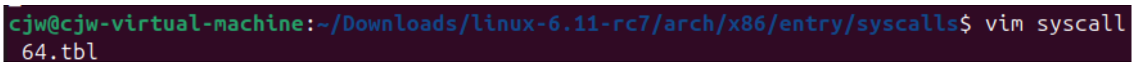


```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch$ cd x86
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/x86$ cd entry
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/x86/entry$ cd syscalls
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/x86/entry/syscalls$ ls
Makefile  syscall_32.tbl  syscall_64.tbl
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/x86/entry/syscalls$ vim syscall
_32.tbl
```

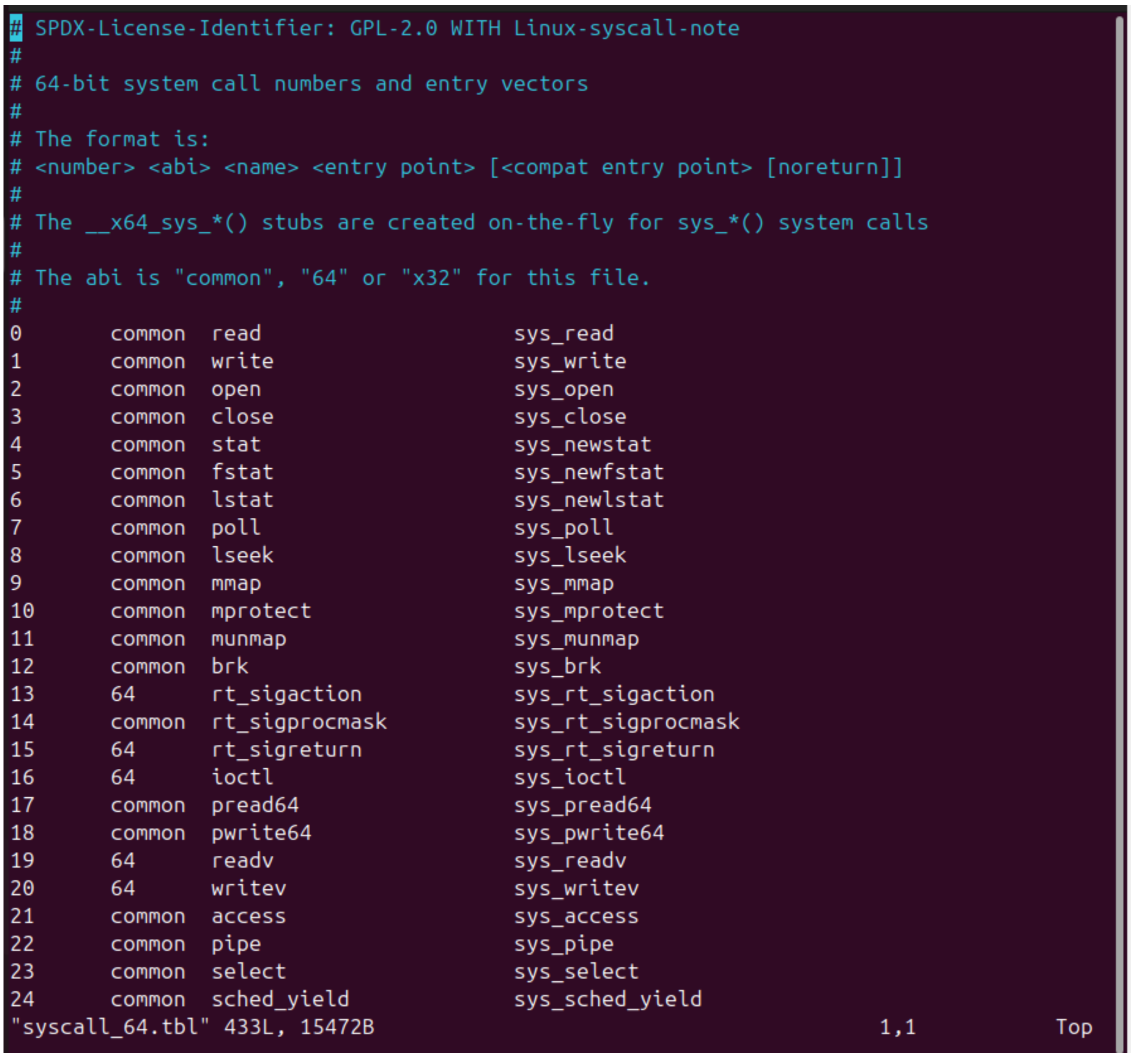


```
# SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
#
# 32-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> [<compat entry point> [noreturn]]
#
# The __ia32_sys and __ia32_compat_sys stubs are created on-the-fly for
# sys_*() system calls and compat_sys_*() compat system calls if
# IA32_EMULATION is defined, and expect struct pt_regs *regs as their only
# parameter.
#
# The abi is always "i386" for this file.
#
0       i386    restart_syscall         sys_restart_syscall
1       i386    exit                    sys_exit                        -
         noreturn
2       i386    fork                    sys_fork
3       i386    read                    sys_read
4       i386    write                   sys_write
5       i386    open                    sys_open                        compat_sys_open
6       i386    close                   sys_close
7       i386    waitpid                 sys_waitpid
8       i386    creat                   sys_creat
9       i386    link                    sys_link
10      i386    unlink                  sys_unlink
11      i386    execve                  sys_execve                      compat_sys_exec
ve
12      i386    chdir                   sys_chdir
13      i386    time                    sys_time32
14      i386    mknod                   sys_mknod
15      i386    chmod                   sys_chmod
16      i386    lchown                  sys_lchown16
17      i386    break
18      i386    oldstat                 sys_stat
@@@
"syscall_32.tbl" 470L, 18509B                                     1,1           Top
```

4. x86_64



```
cjw@cjw-virtual-machine:~/Downloads/linux-6.11-rc7/arch/x86/entry/syscalls$ vim syscall
_64.tbl
```



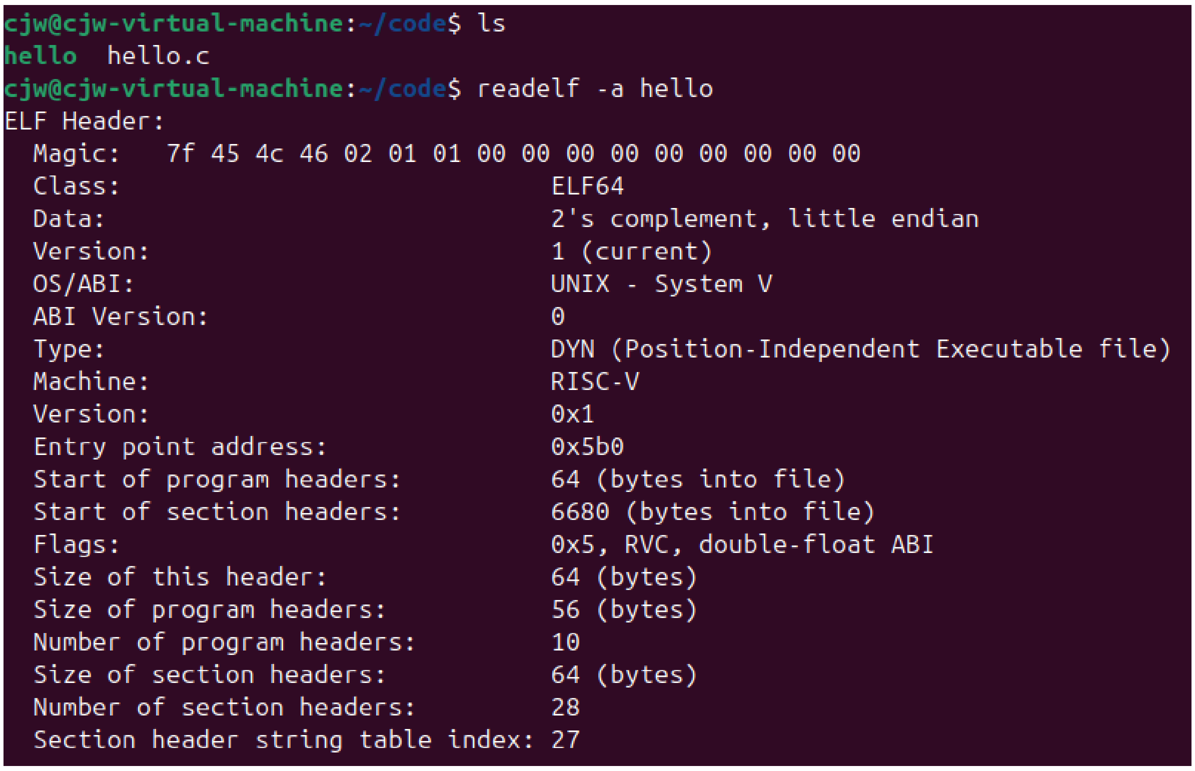
```
# SPDX-License-Identifier: GPL-2.0 WITH Linux-syscall-note
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point> [<compat entry point> [noreturn]]
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "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
24      common  sched_yield             sys_sched_yield
"syscall_64.tbl" 433L, 15472B                                     1,1           Top
```

同样直接通过vim查看

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。

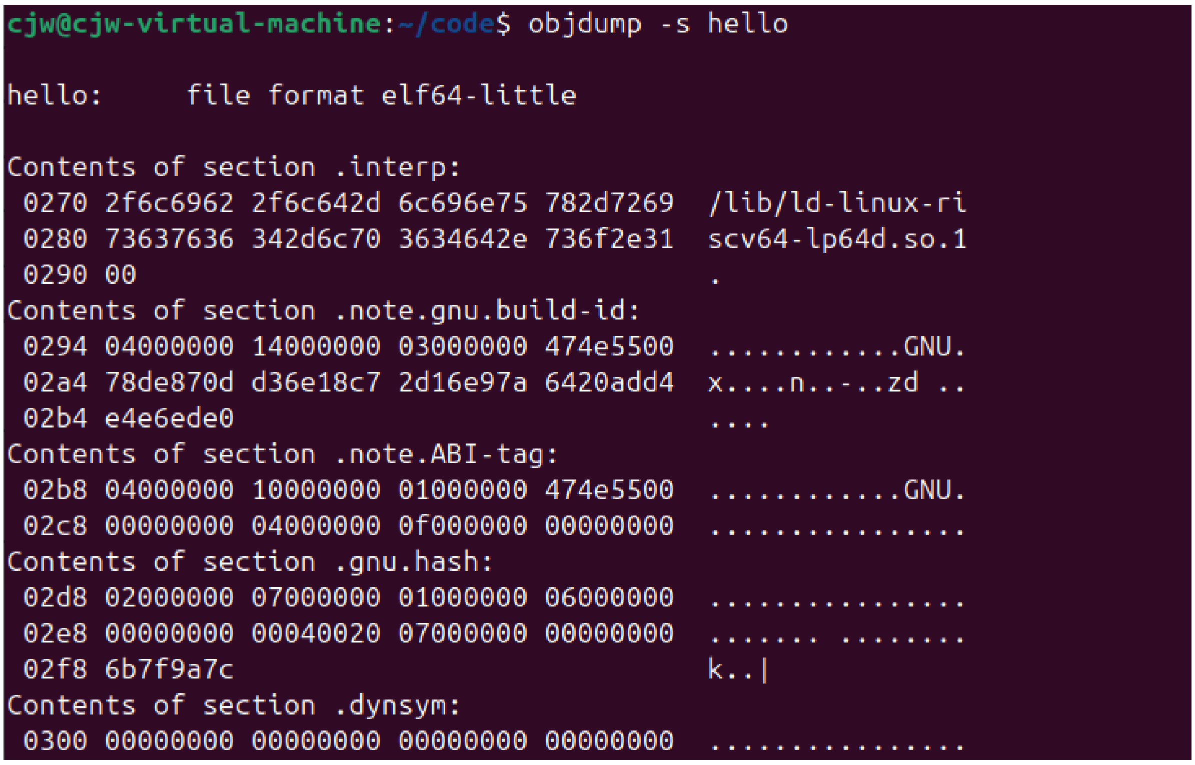
ELF文件是一种二进制标准文件格式，包含了程序的可执行代码，数据，符号表，重定位信息和其他程序执行相关的元数据，用于表示可执行文件，共享库，共享文件等。

使用了上次的hello.c文件，生成可执行文件后readelf：



```
cjw@cjw-virtual-machine:~/code$ ls
hello  hello.c
cjw@cjw-virtual-machine:~/code$ readelf -a hello
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:                              DYN (Position-Independent Executable file)
  Machine:                           RISC-V
  Version:                           0x1
  Entry point address:               0x5b0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          6680 (bytes into file)
  Flags:                             0x5, RVC, double-float ABI
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         28
  Section header string table index: 27
```

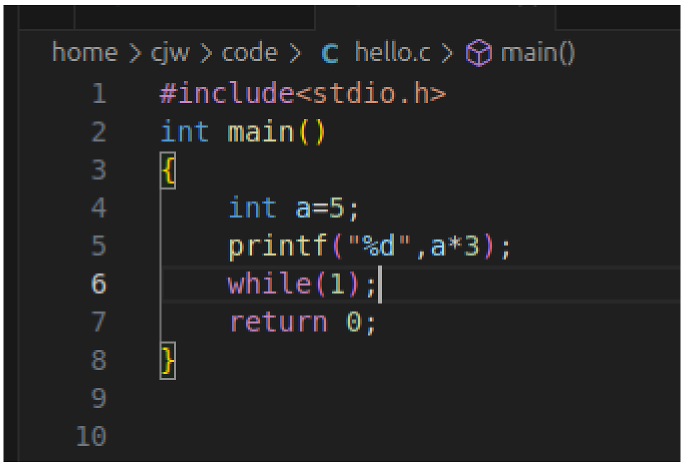
使用objdump：



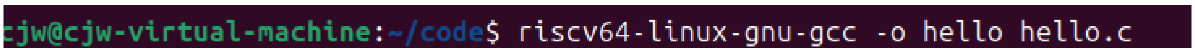
```
cjw@cjw-virtual-machine:~/code$ objdump -s hello

hello:     file format elf64-little

Contents of section .interp:
 0270 2f6c6962 2f6c642d 6c696e75 782d7269  /lib/ld-linux-ri
 0280 73637636 342d6c70 3634642e 736f2e31  scv64-lp64d.so.1
 0290 00                                   .
Contents of section .note.gnu.build-id:
 0294 04000000 14000000 03000000 474e5500  ............GNU.
 02a4 78de870d d36e18c7 2d16e97a 6420add4  x....n..-..zd ..
 02b4 e4e6ede0                             ....
Contents of section .note.ABI-tag:
 02b8 04000000 10000000 01000000 474e5500  ............GNU.
 02c8 00000000 04000000 0f000000 00000000  ................
Contents of section .gnu.hash:
 02d8 02000000 07000000 01000000 06000000  ................
 02e8 00000000 00040020 07000000 00000000  ....... ........
 02f8 6b7f9a7c                             k..|
Contents of section .dynsym:
 0300 00000000 00000000 00000000 00000000  ................
```

- **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。**

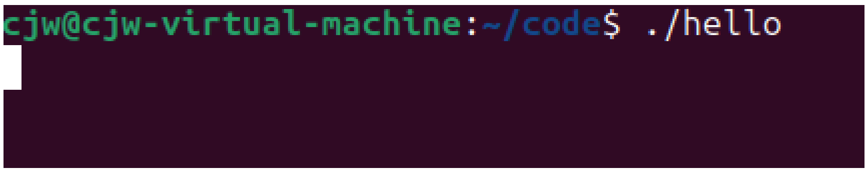
为了使进程一直存在，在代码末尾添加循环，使之不结束，另一个窗口查看PID，并获取maps文件内容查看memory layout。


home > cjw > code > hello.c > main()

```c
#include<stdio.h>
int main()
{
    int a=5;
    printf("%d",a*3);
    while(1);
    return 0;
}
```

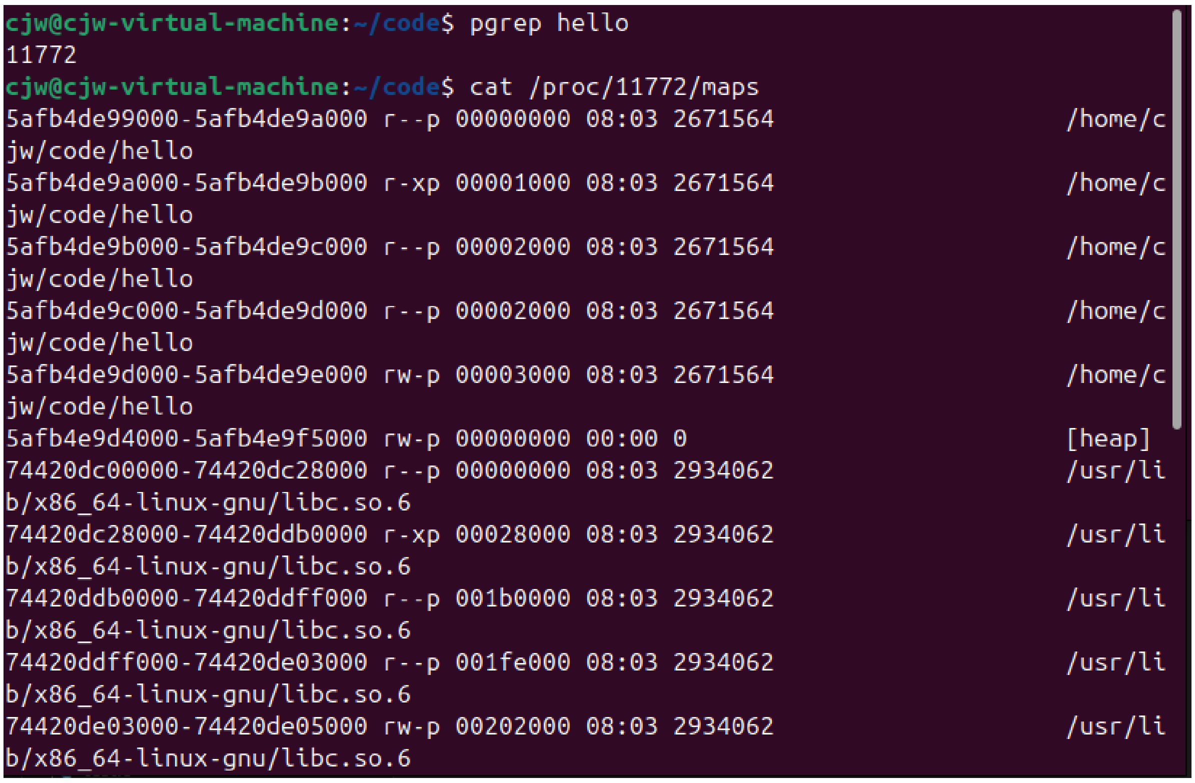


```
cjw@cjw-virtual-machine:~/code$ riscv64-linux-gnu-gcc -o hello hello.c
```

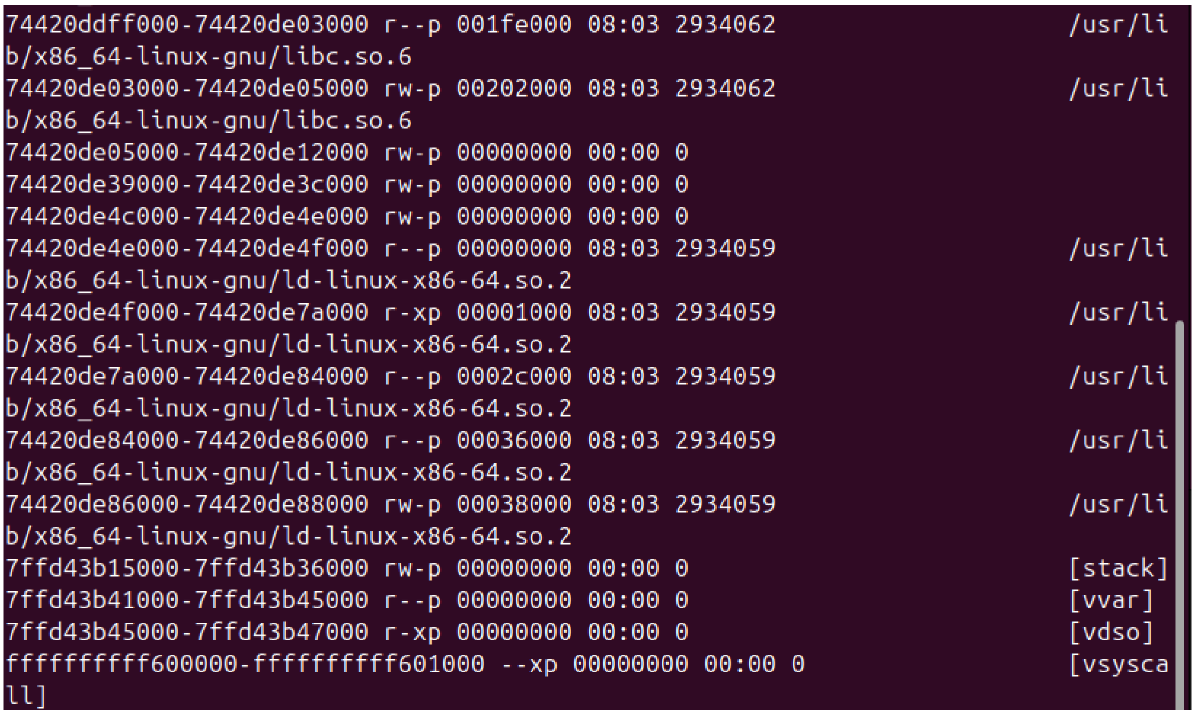


```
cjw@cjw-virtual-machine:~/code$ ./hello
```

这时另开一个终端



```
cjw@cjw-virtual-machine:~/code$ pgrep hello
11772
cjw@cjw-virtual-machine:~/code$ cat /proc/11772/maps
5afb4de99000-5afb4de9a000 r--p 00000000 08:03 2671564                    /home/c
jw/code/hello
5afb4de9a000-5afb4de9b000 r-xp 00001000 08:03 2671564                    /home/c
jw/code/hello
5afb4de9b000-5afb4de9c000 r--p 00002000 08:03 2671564                    /home/c
jw/code/hello
5afb4de9c000-5afb4de9d000 r--p 00002000 08:03 2671564                    /home/c
jw/code/hello
5afb4de9d000-5afb4de9e000 rw-p 00003000 08:03 2671564                    /home/c
jw/code/hello
5afb4e9d4000-5afb4e9f5000 rw-p 00000000 00:00 0                          [heap]
74420dc00000-74420dc28000 r--p 00000000 08:03 2934062                    /usr/li
b/x86_64-linux-gnu/libc.so.6
74420dc28000-74420ddb0000 r-xp 00028000 08:03 2934062                    /usr/li
b/x86_64-linux-gnu/libc.so.6
74420ddb0000-74420ddff000 r--p 001b0000 08:03 2934062                    /usr/li
b/x86_64-linux-gnu/libc.so.6
74420ddff000-74420de03000 r--p 001fe000 08:03 2934062                    /usr/li
b/x86_64-linux-gnu/libc.so.6
74420de03000-74420de05000 rw-p 00202000 08:03 2934062                    /usr/li
b/x86_64-linux-gnu/libc.so.6
```



```
74420ddff000-74420de03000 r--p 001fe000 08:03 2934062                    /usr/li
b/x86_64-linux-gnu/libc.so.6
74420de03000-74420de05000 rw-p 00202000 08:03 2934062                    /usr/li
b/x86_64-linux-gnu/libc.so.6
74420de05000-74420de12000 rw-p 00000000 00:00 0
74420de39000-74420de3c000 rw-p 00000000 00:00 0
74420de4c000-74420de4e000 rw-p 00000000 00:00 0
74420de4e000-74420de4f000 r--p 00000000 08:03 2934059                    /usr/li
b/x86_64-linux-gnu/ld-linux-x86-64.so.2
74420de4f000-74420de7a000 r-xp 00001000 08:03 2934059                    /usr/li
b/x86_64-linux-gnu/ld-linux-x86-64.so.2
74420de7a000-74420de84000 r--p 0002c000 08:03 2934059                    /usr/li
b/x86_64-linux-gnu/ld-linux-x86-64.so.2
74420de84000-74420de86000 r--p 00036000 08:03 2934059                    /usr/li
b/x86_64-linux-gnu/ld-linux-x86-64.so.2
74420de86000-74420de88000 rw-p 00038000 08:03 2934059                    /usr/li
b/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffd43b15000-7ffd43b36000 rw-p 00000000 00:00 0                          [stack]
7ffd43b41000-7ffd43b45000 r--p 00000000 00:00 0                          [vvar]
7ffd43b45000-7ffd43b47000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsysca
ll]
```

可以看到：

可执行文件hello被映射到地址范围5afb4de99000-5afb4de9a000，可读客执行，摆阔代码和只读数据

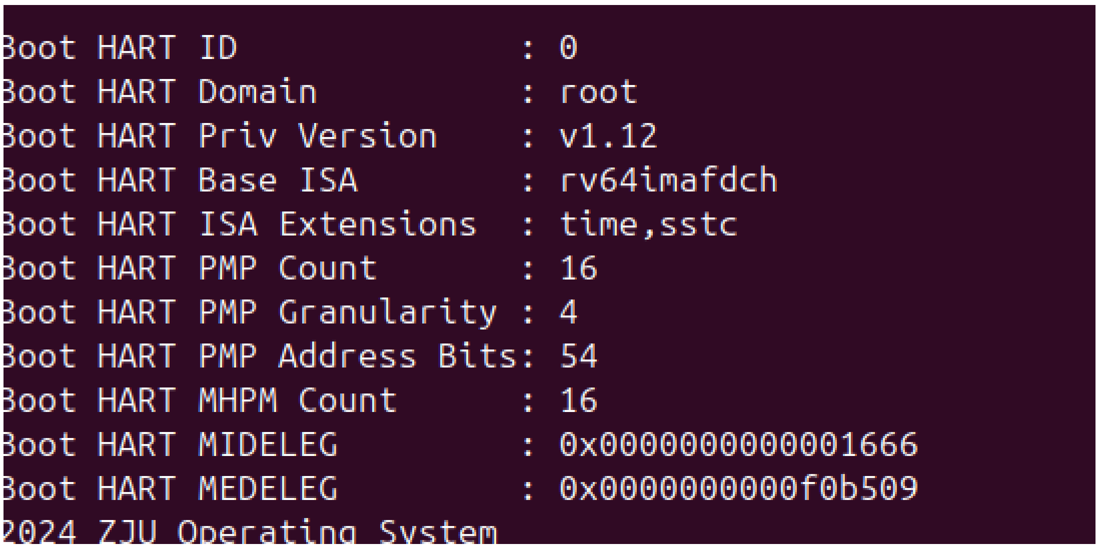
共享库libc.so.6被映射到范围74420dc0000-74420ddb0000，包括可读，可写和可执行段，这是c标准库。

还有一些其他映射，比如动态链接器ld-linux-x86-64.so.2，栈区stack，以及内核相关的映射vvar和vdso。

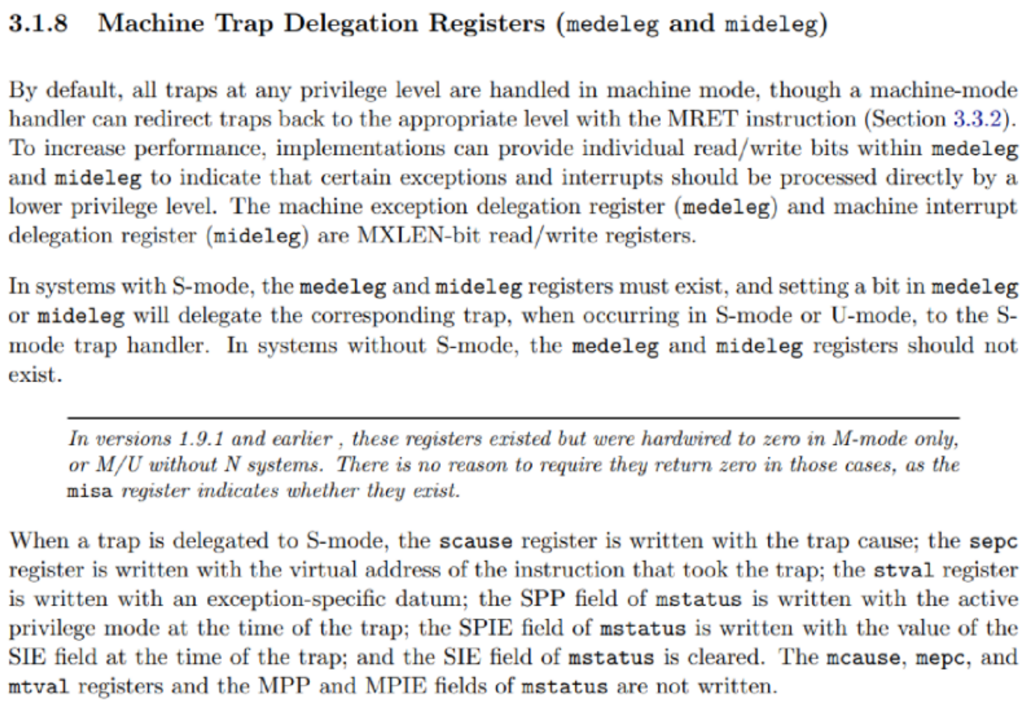
8.在我们使用 make run 时，OpenSBI 会产生如下输出：

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

      ......

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

      ......
```

- **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。**



```
Boot HART ID              : 0
Boot HART Domain          : root
Boot HART Priv Version    : v1.12
Boot HART Base ISA        : rv64imafdch
Boot HART ISA Extensions  : time,sstc
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x0000000000f0b509
2024 ZJU Operating System
```



3.1.8 Machine Trap Delegation Registers (`medeleg` and `mideleg`)

By default, all traps at any privilege level are handled in machine mode, though a machine-mode handler can redirect traps back to the appropriate level with the MRET instruction (Section 3.3.2). To increase performance, implementations can provide individual read/write bits within **`medeleg`** and **`mideleg`** to indicate that certain exceptions and interrupts should be processed directly by a lower privilege level. The machine exception delegation register (**`medeleg`**) and machine interrupt delegation register (**`mideleg`**) are MXLEN-bit read/write registers.

In systems with S-mode, the **`medeleg`** and **`mideleg`** registers must exist, and setting a bit in **`medeleg`** or **`mideleg`** will delegate the corresponding trap, when occurring in S-mode or U-mode, to the S-mode trap handler. In systems without S-mode, the **`medeleg`** and **`mideleg`** registers should not exist.

In versions 1.9.1 and earlier , these registers existed but were hardwired to zero in M-mode only, or M/U without N systems. There is no reason to require they return zero in those cases, as the **`misa`** *register indicates whether they exist.*

When a trap is delegated to S-mode, the **`scause`** register is written with the trap cause; the **`sepc`** register is written with the virtual address of the instruction that took the trap; the **`stval`** register is written with an exception-specific datum; the SPP field of **`mstatus`** is written with the active privilege mode at the time of the trap; the SPIE field of **`mstatus`** is written with the value of the SIE field at the time of the trap; and the SIE field of **`mstatus`** is cleared. The **`mcause`**, **`mepc`**, and **`mtval`** registers and the MPP and MPIE fields of **`mstatus`** are not written.

一般来说，异常和中断都在M mode下处理，即使是在S mode下的异常和中断也是在M mode处理。medeleg和mideleg两个寄存器允许系统将部分异常和中断委托给更低特权级别的处理器模式来处理，而不是强制性地在M mode下处理。通过这种机制，可以让 S mode 处理某些异常和中断，而不需要每次都返回 M mode，这样可以提高操作系统的效率。

本次MIDELEG值为0x0000000000001666(二进制为0001011001100110)，被委托的中断包括：监督软件中断（bit1设置为1），监督定时器中断（bit5设置为1），监督外部中断（bit9设置为1）。

本次MEDELEG值为0x0000000000f0b509(二进制为111100001011010100001001)，在该值中，多个异常被委托给了 S-mode，包括：U-mode 的系统调用（bit8设置为1），S-mode 的系统调用（bit9设置为1），以及其他一些异常。