Lab1：RV64内核引导与时钟中断处理

张志心 `3210106357`

1 实验目的

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

2 实验环境

- 本次实验环境为 Mac 系统下 Ubuntu22.04 VM。（不同于 Lab0）。

  环境配置过程与 Lab0 类似。

3 实验步骤

3.1 RV64 内核引导

3.1.1 编写 head.S

为即将运行的第一个 C 函数设置程序栈，大小为 4kb，将栈放置在 `.bss.stack` 段。
然后通过跳转指令，跳转至 `main.c` 中的 `start_kernel`。

```
# head.S
.extern start_kernel

    .section .text.entry
    .globl _start
_start:
    # ------------------
    # - your code here -
    # ------------------
    la sp, boot_stack_top
    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:
```

3.1.2 完善 Makefile 脚本

补充 `lib/Makefile` 如下：

```
SRCS = $(shell find . -name *"*.S") $(shell find . -name "*.c")
OBJS = $(addsuffix .o, $(basename $(SRCS)))

all: $(OBJS)

%.o: %.S
    @echo CC $< $@
    @$(GCC) $(CFLAG) -c $< -o $@


%.o: %.c
    @echo CC $< $@
    @$(GCC) $(CFLAG) -c $< -o $@


clean:
    -@rm *.o 2>/dev/null
```

3.1.3 补充 sbi.c

1. 将 ext (Extension ID) 放入寄存器 a7 中，fid (Function ID) 放入寄存器 a6 中，将 arg0 ~ arg5 放入寄存器 a0 ~ a5 中。
2. 使用 `ecall` 指令。`ecall` 之后系统会进入 M 模式，之后 OpenSBI 会完成相关操作。
3. OpenSBI 的返回结果会存放在寄存器 a0 ， a1 中，其中 a0 为 error code， a1 为返回值， 我们用 sbiret 来接受这两个返回值。

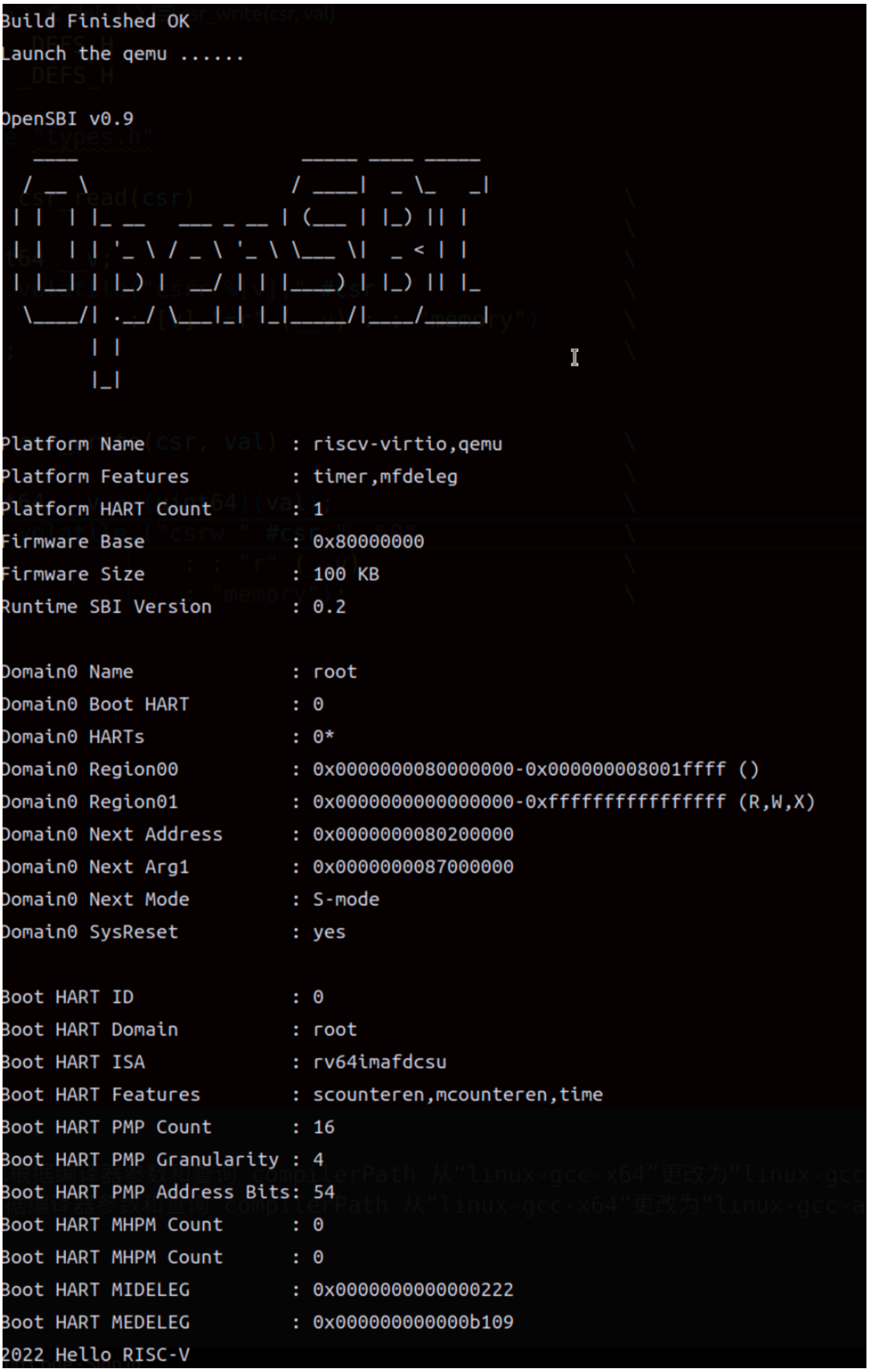
```
struct sbiret result;
    __asm__ volatile (
        "\tmv a7, %[ext]\n"
        "\tmv a6, %[fid]\n"
        "\tmv a0, %[arg0]\n"
        "\tmv a1, %[arg1]\n"
        "\tmv a2, %[arg2]\n"
        "\tmv a3, %[arg3]\n"
        "\tmv a4, %[arg4]\n"
        "\tmv a5, %[arg5]\n"
        "\tecall\n"
        "\tmv %[err], a0\n"
        "\tmv %[res], a1\n"
        : [err] "=r" (result.error), [res] "=r" (result.value)
        : [ext] "r" (ext), [fid] "r" (fid),
          [arg0] "r" (arg0), [arg1] "r" (arg1), [arg2] "r" (arg2), [arg3] "r"
(arg3), [arg4] "r" (arg4), [arg5] "r" (arg5)
    );
    return result;
```

3.1.4 修改 defs

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

3.1.5 qemu 运行 `make` 得到内核

在 `/lab1` 目录下进行 `make`


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

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

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

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

Boot HART ID              : 0
Boot HART Domain          : root
Boot HART ISA             : rv64imafdcsu
Boot HART Features        : scounteren,mcounteren,time
Boot HART PMP Count       : 16
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 0
Boot HART MHPM Count      : 0
Boot HART MIDELEG         : 0x0000000000000222
Boot HART MEDELEG         : 0x000000000000b109
2022 Hello RISC-V
```

3.2 RV64 时钟中断处理

- 准备工作

```
# vmlinux.lds
.text : ALIGN(0x1000){
    _stext = .;

    *(.text.init)      <- 加入了 .text.init
    *(.text.entry)     <- 之后我们实现 中断处理逻辑 会放置在 .text.entry
    *(.text .text.*)

    _etext = .;
}
# head.S
.extern start_kernel

    # .section .text.entry
    .section .text.init         <- 将 _start 放入.text.init section
    .globl _start
```

3.2.1 开启 trap 处理

1. 设置 `stvec`， 将 `_traps` 所表示的地址写入 `stvec`，这里我们采用 `Direct 模式`，而 `_traps` 则是 trap 处理入口函数的基地址。
2. 开启时钟中断，将 `sie[STIE]` 置 1。
3. 设置第一次时钟中断，参考 `clock_set_next_event()` (`clock_set_next_event()` 在 4.3.4 中介绍) 中的逻辑用汇编实现。
4. 开启 S 态下的中断响应， 将 `sstatus[SIE]` 置 1。

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

    # enable supervisor time interrupt
    csrr t0, sie
    ori t1, t0, 0x00000020
    csrw sie, t1

    # set first time interrupt
    rdtime t0
    li a0, 10000000
    add a0, t0, a0
    li a1, 0
    li a2, 0
    li a3, 0
    li a4, 0
    li a5, 0
    li a6, 0
    li a7, 0
    ecall
```

```
    # set sstatus[SIE] = 1
    csrr t0, sstatus
    ori t1, t0, 0x00000002
    csrw sstatus, t1
```

3.2.2 实现上下文切换

使用汇编实现上下文切换机制， 包含以下几个步骤：

1. 在 `arch/riscv/kernel/` 目录下添加 `entry.S` 文件。
2. 保存 CPU 的寄存器（上下文）到内存中（栈上）。
3. 将 `scause` 和 `sepc` 中的值传入 trap 处理函数 `trap_handler` （ `trap_handler` 在 4.3.3 中介绍），我们将会在 `trap_handler` 中实现对 trap 的处理。
4. 在完成对 trap 的处理之后， 我们从内存中（栈上）恢复CPU的寄存器（上下文）。
5. 从 trap 中返回。

```
.section .text.entry

#define context_size 256
    .align 2
    .globl _traps
_traps:

    # allocate context struct
    addi sp, sp, -context_size

    sd x1, 8(sp)
    sd x3, 24(sp)
    sd x4, 32(sp)
    sd x5, 40(sp)
    sd x6, 48(sp)
    sd x7, 56(sp)
    sd x8, 64(sp)
    sd x9, 72(sp)
    sd x10, 80(sp)
    sd x11, 88(sp)
    sd x12, 96(sp)
    sd x13, 104(sp)
    sd x14, 112(sp)
    sd x15, 120(sp)
    sd x16, 128(sp)
    sd x17, 136(sp)
    sd x18, 144(sp)
    sd x19, 152(sp)
    sd x20, 160(sp)
    sd x21, 168(sp)
    sd x22, 176(sp)
    sd x23, 184(sp)
    sd x24, 192(sp)
    sd x25, 200(sp)
    sd x26, 208(sp)
```

```
    sd x27, 216(sp)
    sd x28, 224(sp)
    sd x29, 232(sp)
    sd x30, 240(sp)
    sd x31, 248(sp)

    # save sepc
    csrr t0, sepc
    sd t0, 0(sp)

    csrr t0, scause
    mv a0, t0
    csrr t0, sepc
    mv a1, t0
    call trap_handler

    ld t0, 0(sp)
    csrw sepc, t0

    ld x1, 8(sp)
    ld x3, 24(sp)
    ld x4, 32(sp)
    ld x5, 40(sp)
    ld x6, 48(sp)
    ld x7, 56(sp)
    ld x8, 64(sp)
    ld x9, 72(sp)
    ld x10, 80(sp)
    ld x11, 88(sp)
    ld x12, 96(sp)
    ld x13, 104(sp)
    ld x14, 112(sp)
    ld x15, 120(sp)
    ld x16, 128(sp)
    ld x17, 136(sp)
    ld x18, 144(sp)
    ld x19, 152(sp)
    ld x20, 160(sp)
    ld x21, 168(sp)
    ld x22, 176(sp)
    ld x23, 184(sp)
    ld x24, 192(sp)
    ld x25, 200(sp)
    ld x26, 208(sp)
    ld x27, 216(sp)
    ld x28, 224(sp)
    ld x29, 232(sp)
    ld x30, 240(sp)
    ld x31, 248(sp)

    addi sp, sp, context_size
    sret
```

3.2.3 实现 trap 处理函数

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

```
void trap_handler(unsigned long scause, unsigned long sepc) {
    if ((long) scause < 0 && (scause & ((1l << 63) - 1)) == 5) {
        printk("%s", "Get STI!\n");
        clock_set_next_event();
    }
}
```

3.2.4 实现时钟中断相关函数

1. 在 `clock.c` 中实现 get_cycles ( )：使用 `rdtime` 汇编指令获得当前 `time` 寄存器中的值。
2. 在 `clock.c` 中实现 clock_set_next_event ( )：调用 `sbi_ecall`，设置下一个时钟中断事件。

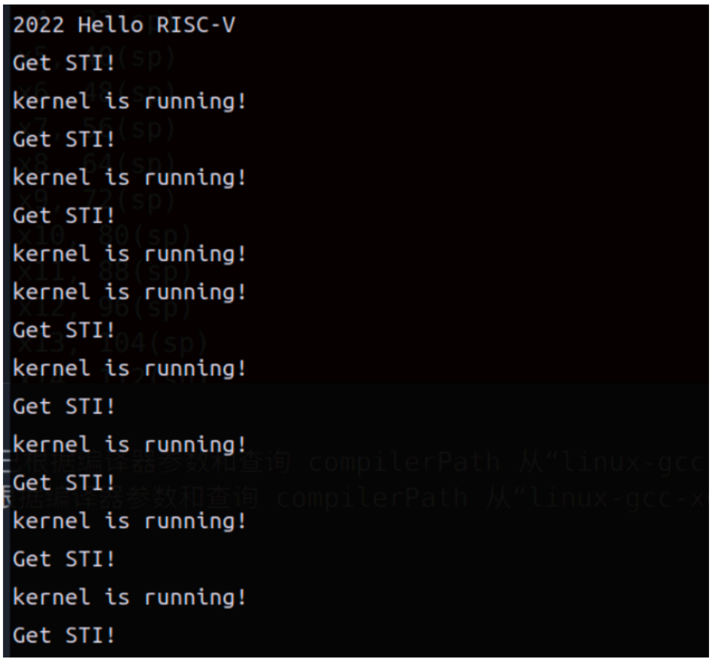
```
unsigned long get_cycles() {
    unsigned long m_time;
    __asm__ volatile (
        "rdtime t0\n"
        "mv %[m_time], t0\n"
        : [m_time] "=r" (m_time)
    );
    return m_time;
}

void clock_set_next_event() {
    unsigned long next = get_cycles() + TIMECLOCK;

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

3.2.5 编译及测试

```
// test.c
void test() {
    for(int i = 1; i <= 120000000; ++i) {
        if(i == 120000000) {
            printk("%s", "kernel is running!\n");
            i = 0;
        }
    }
}
```



4 思考题

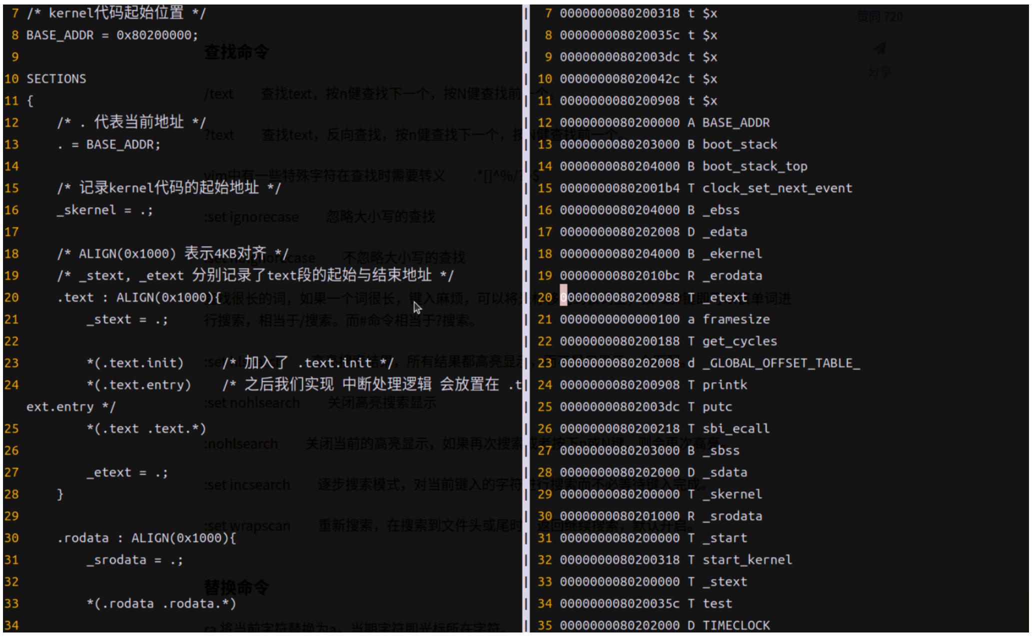
1. 请总结一下 **RISC-V** 的 **calling convention**，并解释 **Caller / Callee Saved Register** 有什么区别？

   （**1**）**calling convention** 即调用规约：

   - 将参数放到寄存器或栈上；
   - 按需将调用者保存寄存器的值压到栈上；
   - 使用 `jal` 或 `jalr` 指令，调用函数；
   - 被调用者按需保存被调用者保存寄存器；
   - 运行被调用函数代码；
   - 恢复被调用者保存寄存器；
   - 执行 `ret` 返回；
   - 恢复调用者保存寄存器。

   （**2**）假设 **A** 调用 **B**，调用者保存寄存器（**Caller**）是 **A** 在调用 **B** 之前，需要将其值压到栈上保存，并在 **B** 返回后恢复的寄存器，**B** 可以对其任意修改而不用恢复；被调用者保存寄存器（**Callee Saved Register**）是 **B** 在被调用之后需要第一时间压到栈上保存的寄存器，并在退出前恢复。

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



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

修改 `test.c`

```c
#include <defs.h>
#include <printk.h>

struct sstatus {
    uint64 wpri_0 : 1;
    uint64 sie : 1;
    uint64 wpri_1 : 3;
    uint64 spie : 1;
    uint64 ube : 1;
    uint64 wpri_2 : 1;
    uint64 spp : 1;
    uint64 vs : 2;
    uint64 wpri_3 : 2;
    uint64 fs : 2;
    uint64 xs_l : 1;
    uint64 xs_h : 1;
    uint64 wpri_4 : 1;
    uint64 sum : 1;
    uint64 mxr : 1;
    uint64 wpri_5 : 4;
    uint64 wpri_6 : 8;
    uint64 uxl : 2;
    uint64 wpri_7 : 6;
    uint64 wpri_8 : 8;
    uint64 wpri_9 : 8;
    uint64 wpri_10 : 7;
    uint64 sd : 1;
}__attribute__((packed));

void test() {
    uint64 sstatus_v = csr_read(sstatus);
```

```c
    struct sstatus *sstatus_0 = &sstatus_v;
    printk("sstatus: \n");
    printk("sie %lld\n", sstatus_0->sie);
    printk("spie %lld\n", sstatus_0->spie);
    printk("ube %lld\n", sstatus_0->ube);
    printk("spp %lld\n", sstatus_0->spp);
    printk("vs %lld\n", sstatus_0->vs);
    printk("fs %lld\n", sstatus_0->fs);
    printk("xs %lld\n", sstatus_0->xs_l | (sstatus_0->xs_h << 1));
    printk("sum %lld\n", sstatus_0->sum);
    printk("mxr %lld\n", sstatus_0->mxr);
    printk("uxl %lld\n", sstatus_0->uxl);
    printk("sd %lld\n", sstatus_0->sd);
    csr_write(sscratch, 0x57);
    printk("sscratch: 0x%x\n", csr_read(sscratch));
    for(int i = 1; i <= 120000000; ++i) {
        if(i == 120000000) {
            printk("kernel is running!\n");
            i = 0;
        }
    }
}
```

```
Boot HART MEDELEG          : 0x000000000000b109
2022 Hello RISC-V
sstatus:
sie 1
spie 0
ube 0
spp 0
vs 0
fs 3
xs 0
sum 0
mxr 0
uxl 0
sd 1
sscratch: 0x00000057
kernel is running!
Get STI!
```

如图 `csr_read` 和 `csr_write` 的实现正确。

【**sie = 1**】允许中断。

【**spie = 0**】进入**S mode**之前不允许中断。

【**ube = 0**】小端内存访问。

【**spp = 0**】之前的**mode**是**U mode**。

【**vs = 0**】禁止扩展**S mode**中虚拟内存功能。

【**sum = 0**】不允许用户访问内存。

【**mxr = 0**】不允许执行从用户模式内存读取的指令。

【**sd = 1**】允许**S mode**下的中断在**S mode**中处理。

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

见上一题。

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

在 `/linux-6.6-rc2` 目录下执行：

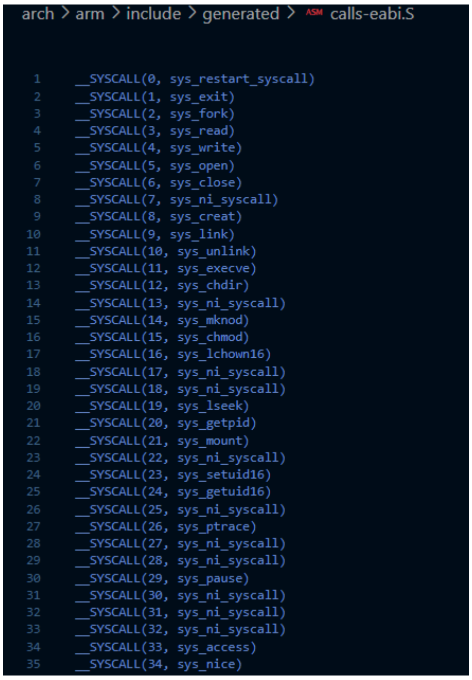
```
sudo apt install g++-aarch64-linux-gnu binutils-aarch64-linux-gnu
make ARCH=arm64 defconfig
make arch/arm64/kernel/sys.i ARCH=arm64 CROSS_COMPILE=aarch64-linux-
gnu-
```

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

- `ARM32`： `arch/arm/kernel/entry-common.S`

```
    syscall_table_start sys_call_table
#ifdef CONFIG_AEABI
#include <calls-eabi.S>
#else
#include <calls-oabi.S>
#endif
    syscall_table_end sys_call_table
```

`arch/arm/include/generated/calls-eabi.S`



arch > arm > include > generated > calls-eabi.S

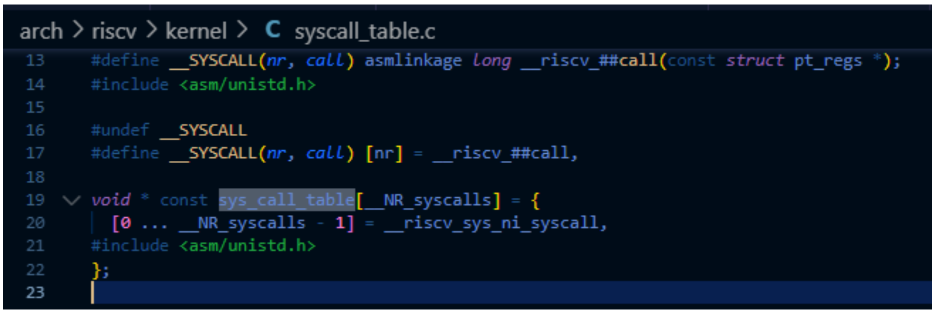
```
__SYSCALL(0, sys_restart_syscall)
__SYSCALL(1, sys_exit)
__SYSCALL(2, sys_fork)
__SYSCALL(3, sys_read)
__SYSCALL(4, sys_write)
__SYSCALL(5, sys_open)
__SYSCALL(6, sys_close)
__SYSCALL(7, sys_ni_syscall)
__SYSCALL(8, sys_creat)
__SYSCALL(9, sys_link)
__SYSCALL(10, sys_unlink)
__SYSCALL(11, sys_execve)
__SYSCALL(12, sys_chdir)
__SYSCALL(13, sys_ni_syscall)
__SYSCALL(14, sys_mknod)
__SYSCALL(15, sys_chmod)
__SYSCALL(16, sys_lchown16)
__SYSCALL(17, sys_ni_syscall)
__SYSCALL(18, sys_ni_syscall)
__SYSCALL(19, sys_lseek)
__SYSCALL(20, sys_getpid)
__SYSCALL(21, sys_mount)
__SYSCALL(22, sys_ni_syscall)
__SYSCALL(23, sys_setuid16)
__SYSCALL(24, sys_getuid16)
__SYSCALL(25, sys_ni_syscall)
__SYSCALL(26, sys_ptrace)
__SYSCALL(27, sys_ni_syscall)
__SYSCALL(28, sys_ni_syscall)
__SYSCALL(29, sys_pause)
__SYSCALL(30, sys_ni_syscall)
__SYSCALL(31, sys_ni_syscall)
__SYSCALL(32, sys_ni_syscall)
__SYSCALL(33, sys_access)
__SYSCALL(34, sys_nice)
```

由于 **AS** 没有预处理选项，所以无法进行宏展开。

- `RISC-V(32 bit)`

```
make ARCH=riscv 32-bit.config
make arch/riscv/kernel/syscall_table.i ARCH=riscv CROSS_COMPILE
=riscv64-linux-gnu-
```

`arch/riscv/kernel/syscall_table.c`



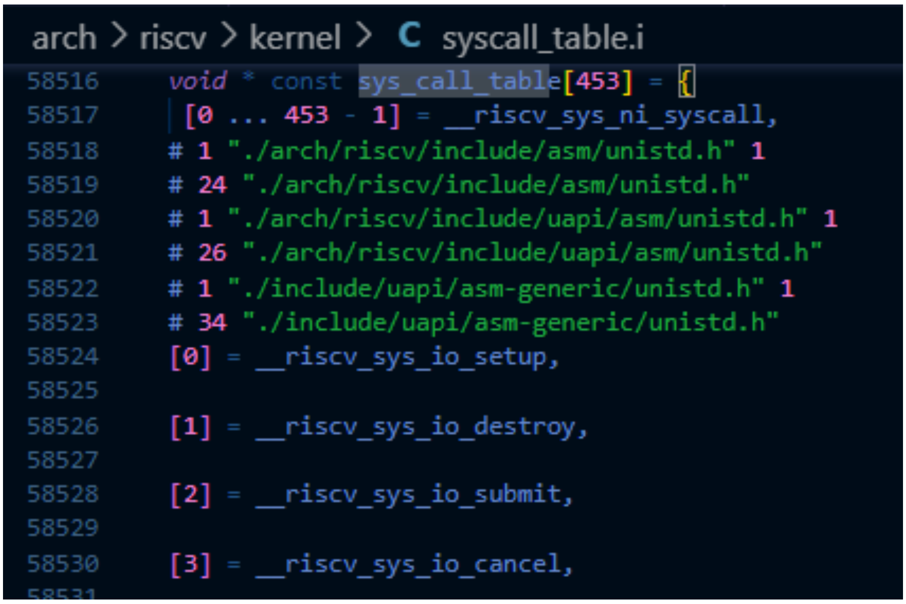
arch > riscv > kernel > syscall_table.c

```
#define __SYSCALL(nr, call) asmlinkage long __riscv_##call(const struct pt_regs *);
#include <asm/unistd.h>

#undef __SYSCALL
#define __SYSCALL(nr, call) [nr] = __riscv_##call,

void * const sys_call_table[__NR_syscalls] = {
    [0 ... __NR_syscalls - 1] = __riscv_sys_ni_syscall,
#include <asm/unistd.h>
};
```

`arch/riscv/kernel/syscall_table.i`



- `RISC-V(64 bit)`

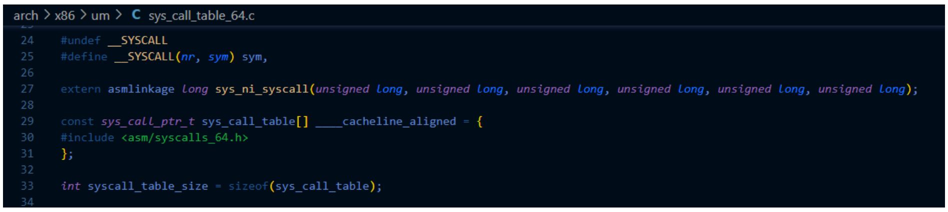
```
make ARCH=riscv 64-bit.config
make arch/riscv/kernel/syscall_table.i ARCH=riscv CROSS_COMPILE
=riscv64-linux-gnu-
```

`arch/riscv/kernel/syscall_table.i`

- `x86 (32 bit)`

```
make ARCH=x86 i386_defconfig
make arch/x86/um/sys_call_table_32.i CROSS_COMPILE= ARCH=x86
```

`arch/x86/um/sys_call_table_32.c`



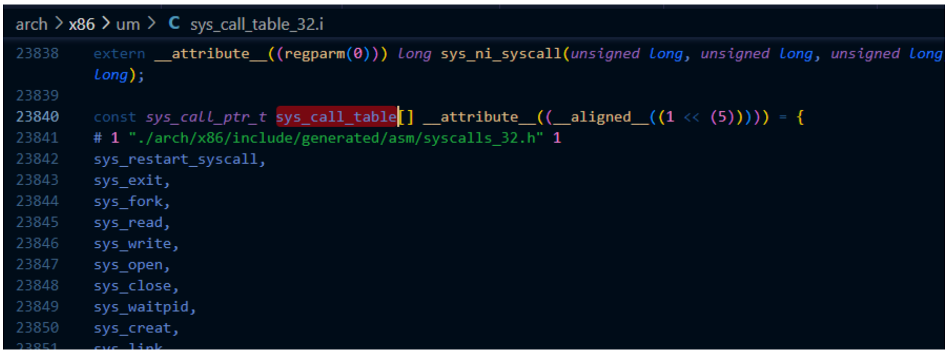
```
arch > x86 > um > C sys_call_table_64.c
#undef __SYSCALL
#define __SYSCALL(nr, sym) sym,

extern asmlinkage long sys_ni_syscall(unsigned long, unsigned long, unsigned long, unsigned long, unsigned long, unsigned long);

const sys_call_ptr_t sys_call_table[] ____cacheline_aligned = {
#include <asm/syscalls_64.h>
};

int syscall_table_size = sizeof(sys_call_table);
```

`arch/x86/um/sys_call_table_32.i`



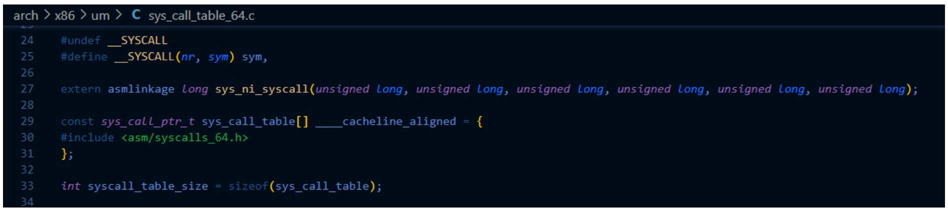
```
arch > x86 > um > C sys_call_table_32.i
extern __attribute__((regparm(0))) long sys_ni_syscall(unsigned long, unsigned long, unsigned long
long);

const sys_call_ptr_t sys_call_table[] __attribute__((__aligned__((1 << (5))))) = {
# 1 "./arch/x86/include/generated/asm/syscalls_32.h" 1
sys_restart_syscall,
sys_exit,
sys_fork,
sys_read,
sys_write,
sys_open,
sys_close,
sys_waitpid,
sys_creat,
```

- `x86_64`

```
make ARCH=x86 x86_64_defconfig
make arch/x86/um/sys_call_table_64.i CROSS_COMPILE= ARCH=x86
```

`arch/x86/um/sys_call_table_64.c`



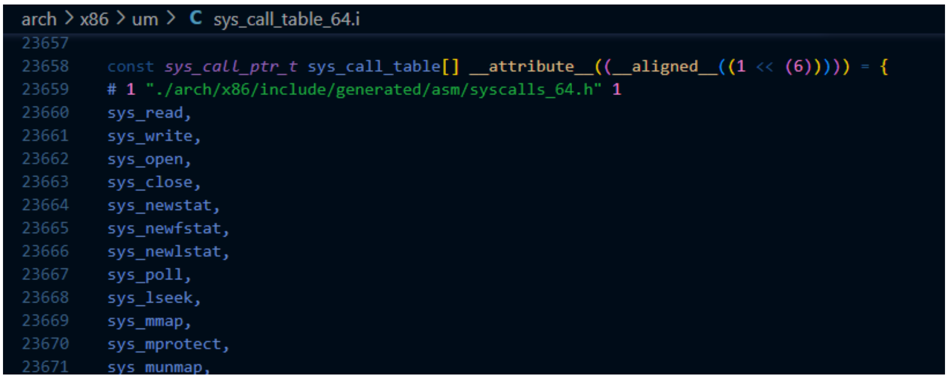
```
arch > x86 > um > C sys_call_table_64.c
#undef __SYSCALL
#define __SYSCALL(nr, sym) sym,

extern asmlinkage long sys_ni_syscall(unsigned long, unsigned long, unsigned long, unsigned long, unsigned long, unsigned long);

const sys_call_ptr_t sys_call_table[] ____cacheline_aligned = {
#include <asm/syscalls_64.h>
};

int syscall_table_size = sizeof(sys_call_table);
```

`arch/x86/um/sys_call_table_64.i`



```
arch > x86 > um > C sys_call_table_64.i
const sys_call_ptr_t sys_call_table[] __attribute__((__aligned__((1 << (6))))) = {
# 1 "./arch/x86/include/generated/asm/syscalls_64.h" 1
sys_read,
sys_write,
sys_open,
sys_close,
sys_newstat,
sys_newfstat,
sys_newlstat,
sys_poll,
sys_lseek,
sys_mmap,
sys_mprotect,
sys_munmap,
```

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

ELF 包含将序加载到内存中所必要的程序内存布局的数据结构（如程序头表、符号表、节头表）和各个段的具体数据。

如下为读取 **ELF** 头的截图：

```
litrehinn@litrehinn-soft-router:~/zzx/tmp$ readelf -h zzx
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:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13976 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
```

如下为读取 **ELF** 符号表的截图：

```
litrehinn@litrehinn-soft-router:~/zzx/tmp$ readelf -s zzx

Symbol table '.dynsym' contains 7 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _[...]@GLIBC_2.34 (2)
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
     3: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (3)
     4: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     5: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
     6: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND [...]@GLIBC_2.2.5 (3)

Symbol table '.symtab' contains 36 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS Scrt1.o
     2: 000000000000038c    32 OBJECT  LOCAL  DEFAULT    4 __abi_tag
     3: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
     4: 0000000000001090     0 FUNC    LOCAL  DEFAULT   16 deregister_tm_clones
     5: 00000000000010c0     0 FUNC    LOCAL  DEFAULT   16 register_tm_clones
     6: 0000000000001100     0 FUNC    LOCAL  DEFAULT   16 __do_global_dtors_aux
     7: 0000000000004010     1 OBJECT  LOCAL  DEFAULT   26 completed.0
     8: 0000000000003dc0     0 OBJECT  LOCAL  DEFAULT   22 __do_global_dtor[...]
     9: 0000000000001140     0 FUNC    LOCAL  DEFAULT   16 frame_dummy
    10: 0000000000003db8     0 OBJECT  LOCAL  DEFAULT   21 __frame_dummy_in[...]
    11: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS zzx.c
    12: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS crtstuff.c
    13: 00000000000020f0     0 OBJECT  LOCAL  DEFAULT   20 __FRAME_END__
    14: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS
    15: 0000000000003dc8     0 OBJECT  LOCAL  DEFAULT   23 _DYNAMIC
    16: 0000000000002010     0 NOTYPE  LOCAL  DEFAULT   19 __GNU_EH_FRAME_HDR
    17: 0000000000003fb8     0 OBJECT  LOCAL  DEFAULT   24 _GLOBAL_OFFSET_TABLE_
    18: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __libc_start_mai[...]
    19: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterT[...]
    20: 0000000000004000     0 NOTYPE  WEAK   DEFAULT   25 data_start
    21: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND puts@GLIBC_2.2.5
    22: 0000000000004010     0 NOTYPE  GLOBAL DEFAULT   25 _edata
    23: 0000000000001168     0 FUNC    GLOBAL HIDDEN    17 _fini
    24: 0000000000004000     0 NOTYPE  GLOBAL DEFAULT   25 __data_start
    25: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
    26: 0000000000004008     0 OBJECT  GLOBAL HIDDEN    25 __dso_handle
    27: 0000000000002000     4 OBJECT  GLOBAL DEFAULT   18 _IO_stdin_used
    28: 0000000000004018     0 NOTYPE  GLOBAL DEFAULT   26 _end
    29: 0000000000001060    38 FUNC    GLOBAL DEFAULT   16 _start
    30: 0000000000004010     0 NOTYPE  GLOBAL DEFAULT   26 __bss_start
    31: 0000000000001149    30 FUNC    GLOBAL DEFAULT   16 main
    32: 0000000000004010     0 OBJECT  GLOBAL HIDDEN    25 __TMC_END__
    33: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMC[...]
    34: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@G[...]
    35: 0000000000001000     0 FUNC    GLOBAL HIDDEN    12 _init
```

如下为读取 **ELF** 程序头表（各段的信息）的截图：

```
litrehinn@litrehinn-soft-router:~/zzx/tmp$ readelf -l zzx

Elf file type is DYN (Position-Independent Executable file)
Entry point 0x1060
There are 13 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000002d8 0x00000000000002d8  R      0x8
  INTERP         0x0000000000000318 0x0000000000000318 0x0000000000000318
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000628 0x0000000000000628  R      0x1000
  LOAD           0x0000000000001000 0x0000000000001000 0x0000000000001000
                 0x0000000000000175 0x0000000000000175  R E    0x1000
  LOAD           0x0000000000002000 0x0000000000002000 0x0000000000002000
                 0x00000000000000f4 0x00000000000000f4  R      0x1000
  LOAD           0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000258 0x0000000000000260  RW     0x1000
  DYNAMIC        0x0000000000002dc8 0x0000000000003dc8 0x0000000000003dc8
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  NOTE           0x0000000000000368 0x0000000000000368 0x0000000000000368
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_PROPERTY   0x0000000000000338 0x0000000000000338 0x0000000000000338
                 0x0000000000000030 0x0000000000000030  R      0x8
  GNU_EH_FRAME   0x0000000000002010 0x0000000000002010 0x0000000000002010
                 0x0000000000000034 0x0000000000000034  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000002db8 0x0000000000003db8 0x0000000000003db8
                 0x0000000000000248 0x0000000000000248  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
   03     .init .plt .plt.got .plt.sec .text .fini
   04     .rodata .eh_frame_hdr .eh_frame
   05     .init_array .fini_array .dynamic .got .data .bss
   06     .dynamic
   07     .note.gnu.property
   08     .note.gnu.build-id .note.ABI-tag
   09     .note.gnu.property
   10     .eh_frame_hdr
   11
   12     .init_array .fini_array .dynamic .got
```

如下为读取 **ELF** 各节的截图：

```
litrehinn@litrehinn-soft-router:~/zzx/tmp$ readelf -S zzx.o
There are 14 section headers, starting at offset 0x258:

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
       000000000000001e  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000198
       0000000000000030  0000000000000018   I      11     1     8
  [ 3] .data             PROGBITS         0000000000000000  0000005e
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  0000005e
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  0000005e
       000000000000000c  0000000000000000   A       0     0     1
  [ 6] .comment          PROGBITS         0000000000000000  0000006a
       000000000000002c  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000096
       0000000000000000  0000000000000000           0     0     1
  [ 8] .note.gnu.pr[...] NOTE             0000000000000000  00000098
       0000000000000020  0000000000000000   A       0     0     8
  [ 9] .eh_frame         PROGBITS         0000000000000000  000000b8
       0000000000000038  0000000000000000   A       0     0     8
  [10] .rela.eh_frame    RELA             0000000000000000  000001c8
       0000000000000018  0000000000000018   I      11     9     8
  [11] .symtab           SYMTAB           0000000000000000  000000f0
       0000000000000090  0000000000000018          12     4     8
  [12] .strtab           STRTAB           0000000000000000  00000180
       0000000000000011  0000000000000000           0     0     1
  [13] .shstrtab         STRTAB           0000000000000000  000001e0
       0000000000000074  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  D (mbind), l (large), p (processor specific)
```

对 `zzx.o` 进行反汇编得到的汇编代码：

```
zzx.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <main>:
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 8d 05 00 00 00 00    lea    0x0(%rip),%rax        # f <main+0xf>
   f: 48 89 c7                mov    %rax,%rdi
  12: e8 00 00 00 00          call   17 <main+0x17>
  17: b8 00 00 00 00          mov    $0x0,%eax
  1c: 5d                      pop    %rbp
  1d: c3                      ret
```

对 `zzx.o` 的各节进行解析得到的二进制 **dump**：

```
zzx.o:     file format elf64-x86-64

Contents of section .text:
 0000 f30f1efa 554889e5 488d0500 00000048  ....UH..H......H
 0010 89c7e800 000000b8 00000000 5dc3      ............].
Contents of section .rodata:
 0000 48656c6c 6f20776f 726c6400           Hello world.
Contents of section .comment:
 0000 00474343 3a202855 62756e74 75203131  .GCC: (Ubuntu 11
 0010 2e342e30 2d317562 756e7475 317e3232  .4.0-1ubuntu1~22
 0020 2e303429 2031312e 342e3000           .04) 11.4.0.
Contents of section .note.gnu.property:
 0000 04000000 10000000 05000000 474e5500  ............GNU.
 0010 020000c0 04000000 03000000 00000000  ................
Contents of section .eh_frame:
 0000 14000000 00000000 017a5200 01781001  .........zR..x..
 0010 1b0c0708 90010000 1c000000 1c000000  ................
 0020 00000000 1e000000 00450e10 8602430d  .........E....C.
 0030 06550c07 08000000                    .U......
```

执行 **ELF** 文件，并输出其内存布局：

```
litrehinn@litrehinn-soft-router:~/zzx/tmp$ ./zzx &
[3] 2930591
litrehinn@litrehinn-soft-router:~/zzx/tmp$ Hello world
cat /proc/2930591/maps
55c3360db000-55c3360dc000 r--p 00000000 103:03 30160894                  /home/litrehinn/zzx/tmp/zzx
55c3360dc000-55c3360dd000 r-xp 00001000 103:03 30160894                  /home/litrehinn/zzx/tmp/zzx
55c3360dd000-55c3360de000 r--p 00002000 103:03 30160894                  /home/litrehinn/zzx/tmp/zzx
55c3360de000-55c3360df000 r--p 00002000 103:03 30160894                  /home/litrehinn/zzx/tmp/zzx
55c3360df000-55c3360e0000 rw-p 00003000 103:03 30160894                  /home/litrehinn/zzx/tmp/zzx
55c3365a9000-55c3365ca000 rw-p 00000000 00:00 0                          [heap]
7f049292d000-7f0492930000 rw-p 00000000 00:00 0
7f0492930000-7f0492958000 r--p 00000000 103:03 6555420                   /usr/lib/x86_64-linux-gnu/libc.so.6
7f0492958000-7f0492aed000 r-xp 00028000 103:03 6555420                   /usr/lib/x86_64-linux-gnu/libc.so.6
7f0492aed000-7f0492b45000 r--p 001bd000 103:03 6555420                   /usr/lib/x86_64-linux-gnu/libc.so.6
7f0492b45000-7f0492b49000 r--p 00214000 103:03 6555420                   /usr/lib/x86_64-linux-gnu/libc.so.6
7f0492b49000-7f0492b4b000 rw-p 00218000 103:03 6555420                   /usr/lib/x86_64-linux-gnu/libc.so.6
7f0492b4b000-7f0492b58000 rw-p 00000000 00:00 0
7f0492b79000-7f0492b7b000 rw-p 00000000 00:00 0
7f0492b7b000-7f0492b7d000 r--p 00000000 103:03 6555385                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f0492b7d000-7f0492ba7000 r-xp 00002000 103:03 6555385                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f0492ba7000-7f0492bb2000 r--p 0002c000 103:03 6555385                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f0492bb3000-7f0492bb5000 r--p 00037000 103:03 6555385                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7f0492bb5000-7f0492bb7000 rw-p 00039000 103:03 6555385                   /usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
7ffd8d696000-7ffd8d6b7000 rw-p 00000000 00:00 0                          [stack]
7ffd8d7ba000-7ffd8d7be000 r--p 00000000 00:00 0                          [vvar]
7ffd8d7be000-7ffd8d7c0000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0                  [vsyscall]
```

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

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

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt

`MIDELEG` 值 **0222**:

- `MIDELEG[1] = 1` 表示把 **Supervisor software interrupt** 委托给 **S-mode**
- `MIDELEG[5] = 1` 表示把 **Supervisor timer interrupt** 委托给 **S-mode**
- `MIDELEG[9] = 1` 表示把 **Supervisor external interrupt** 委托给 **S-mode**