浙江大学实验报告

课程名称：操作系统
实验项目名称：RV64 内核引导与时钟中断处理
学生姓名：展翼飞 学号：3190102196
电子邮件地址：1007921963@qq.com
实验日期： 2024年 10 月 11 日

一、实验内容

1. 准备工程

- 进入os24fall文件夹并使用git命令拉去最新实验工程文件，并检查

```shell
cd os24fall    #进入工程文件夹
git pull       #拉取更新
cd src/lab1    #进入相应文件夹检查
```

2. RV64内核引导

2.1 完善 Makefile 脚本

```
# 查找源文件并生成对象文件列表
C_SRC       = $(sort $(wildcard *.c))
OBJ         = $(patsubst %.c,%.o,$(C_SRC))

# 默认目标
all: printk.o

# 编译 C 文件为 .o 对象文件
%.o: %.c
    $(GCC) $(CFLAG) $(INCLUDE) -c $< -o $@

# 清理编译生成的文件
clean:
    $(shell rm -f $(OBJ) printk.o 2>/dev/null)
    @echo "Cleaned lib folder."
```

在根目录的 Makefile 中存在 all 任务为目标任务，该任务依次进入 lib 等子文件夹运行相应目录下 Makfile 文件进行工程编译。在 lib 子文件夹中，仅需使用根目录 Makefile export 的编译变量对 printk c 文件进行编译打包。

2.2 编写 head. S

为即将运行的第一个 C 函数设置程序栈（栈的大小可以设置为 4 KB），并将该栈放置在 `.bss.stack` 段

```
    .extern start_kernel                # 声明外部符号 start_kernel
    .section .text.entry                # 定义 .text.entry 段，代码将会加载到这里
    .globl _start                       # 声明全局符号 _start，表示程序的入口
_start:
    # 设置堆栈指针
    la sp, boot_stack_top               # 加载 boot_stack_top 地址到 sp 中，设置堆栈顶

    # 跳转到操作系统内核的入口函数
    jal start_kernel                   # 无条件跳转到 start_kernel，不保存返回地址


    .section .bss.stack                 # 定义 .bss.stack 段，分配堆栈空间
    .globl boot_stack                   # 声明全局符号 boot_stack
boot_stack:
    .space 4096                         # 分配 4KB 的堆栈空间

    .globl boot_stack_top               # 声明全局符号 boot_stack_top
boot_stack_top:                         # boot_stack_top 的值会是 boot_stack + 堆栈大小

```

在内核引导过程中，OpenSBI 会将 Linux 内核映像加载到内存中。这包括 `head.s`，这是内核启动的汇编代码，负责完成初始设置。它为即将运行的第一个 C 函数设置程序栈，并将该栈放置在 `.bss.stack` 段。接下来我们只需要通过跳转指令，跳转至 main.c 中的 `start_kernel` 函数

2.3 补充sbi.c

```
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"       // 将 eid 放入寄存器 a7
        "mv a6, %[fid]\n"       // 将 fid 放入寄存器 a6
        "mv a0, %[arg0]\n"      // 将 arg0 放入寄存器 a0
        "mv a1, %[arg1]\n"      // 将 arg1 放入寄存器 a1
        "mv a2, %[arg2]\n"      // 将 arg2 放入寄存器 a2
        "mv a3, %[arg3]\n"      // 将 arg3 放入寄存器 a3
        "mv a4, %[arg4]\n"      // 将 arg4 放入寄存器 a4
        "mv a5, %[arg5]\n"      // 将 arg5 放入寄存器 a5
        "ecall\n"               // 触发系统调用
        "mv %[error], a0\n"     // 将返回的 error code 放入 ret.error
        "mv %[value], a1\n"     // 将返回的 value 放入 ret.value
        : [error] "=r" (ret.error), [value] "=r" (ret.value) // 输出操作数
        : [eid] "r" (eid), [fid] "r" (fid),
          [arg0] "r" (arg0), [arg1] "r" (arg1),
          [arg2] "r" (arg2), [arg3] "r" (arg3),
          [arg4] "r" (arg4), [arg5] "r" (arg5) // 输入操作数
        : "a0", "a1", "a2", "a3", "a4", "a5", "a6", "a7" // 被修改的寄存器
    );

    return ret; // 返回结果
}
```

```c
// 设置定时器
struct sbiret sbi_set_timer(uint64_t stime_value) {
    return sbi_ecall(0x54494D45, 0, stime_value, 0, 0, 0, 0, 0);
}

// 向终端写入数据
struct sbiret sbi_debug_console_write(uint8_t byte) {
    return sbi_ecall(0x4442434E, 0, byte, 0, 0, 0, 0, 0);
}

// 从终端读取数据
struct sbiret sbi_debug_console_read(void) {
    return sbi_ecall(0x4442434E, 1, 0, 0, 0, 0, 0, 0);
}

//向终端写入单个字符
struct sbiret sbi_debug_console_write_byte(uint8_t byte) {
    return sbi_ecall(0x4442434E, 2, byte, 0, 0, 0, 0, 0);
}

//重置系统（关机或重启）
struct sbiret sbi_system_reset(uint32_t reset_type, uint32_t reset_reason) {
    return sbi_ecall(0x53525354, 0, reset_type, reset_reason, 0, 0, 0, 0);
}
```

使用内联汇编与命名占位符格式编写 sbi_ecall，完成以下内容：

1. 将 `eid`（Extension ID）放入寄存器 `a7` 中，`fid`（Function ID）放入寄存器 `a6` 中，将 `arg[0-5]` 放入寄存器 `a[0-5]` 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。
   完成 sbi_ecall 后，其余 sbi 调用均可以通过 sbi_ecall 实现

完成 makefile 与sbi_ecall 编写后，尝试在根目录进行 make，出现以下错误信息：

```
riscv64-linux-gnu-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../..
riscv64-linux-gnu-ld: ../../lib/printk.o: can't link double-float modules
riscv64-linux-gnu-ld: failed to merge target specific data of file ../../l
riscv64-linux-gnu-ld: ../../lib/printk.o: in function `print_dec_int':
printk.c:(.text+0x2f6): undefined reference to `__stack_chk_guard'
...
make[1]: *** [Makefile:3: all] Error 1
make[1]: Leaving directory '/home/frey/os24fall-stu/src/lab1/arch/riscv'
make: *** [Makefile:19: all] Error 2
```

第一个错误表明链接过程中，尝试将使用双精度浮点数的模块与使用软浮点的模块链接在一起。这通常表明编译选

项不一致。需要检查 `Makefile` 中是否有关于浮点 ABI 的设置，并确保所有模块都使用相同的设置。
第二个错误表明在 `printk.c` 中调用了一些函数，但链接器找不到这些函数的定义。通常，`__stack_chk_guard` 是与栈保护机制相关的符号，可能与编译选项或库的链接有关。解决这个问题可以尝试：确保启用了栈保护机制，并且链接了适当的库。

经检查，lib 中的 makefile 编译命令选项 `{CFLAG}` 被写成了 `{CFLAGS}`，修改后编译完成

```
nm ../../vmlinux >  ../../System.map
make[1]: Leaving directory '/home/frey/os24fall-stu/src/lab1/arch/riscv'

Build Finished OK
```

2.4 修改defs

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

使用 `#csr` 的目的是为了将宏参数 `csr` 转换为字符串，使得内联汇编能够使用合适的 CSR 名称，确保生成正确的汇编指令。
如当我们调用 `csr_read(mstatus)` 时（此时宏的参数为直接的 csr 寄存器名字字符串没有加双引号），宏会展开为：

```c
({
  uint64_t __v;
  asm volatile("csrr %0, " "mstatus" : "=r"(__v));
  __v;
})
```

重新编译并 make run，qemu 中可以正确得到字符输出并退出：

```
Boot HART PMP Granularity : 4
Boot HART PMP Address Bits: 54
Boot HART MHPM Count      : 16
Boot HART MIDELEG         : 0x0000000000001666
Boot HART MEDELEG         : 0x00000000f0b509
2024 ZJU Operating System
frey@DESKTOP-5BJAETF:~/os24fall-stu/src/lab1$
```

3. RV64时钟中断处理

3.0 准备工作

修改 `vmlinux.lds` 以及 `head.S`

```
20      .text : ALIGN(0x1000) {
21          _stext = .;
22
23          *(.text.init)
24          *(.text.entry)
25          *(.text .text.*)
```

```
1       .extern start_kernel                # 声明外部符号 start_kernel
2       .section .text.init                 # 改为 .text.init 段，代码将会加载到这里
3       .globl _start                       # 声明全局符号 _start，表示程序的入口
4   _start:
```

修改 head. S 中 text 段段名，并在 lds 文件中添加相应的段排布

3.1 开启trap处理

```
# Initial stack 设置堆栈指针
la sp, boot_stack_top               # 加载 boot_stack_top 地址到 sp 中，设置堆栈顶

# set stvec = _traps 设置中断处理入口
la t0, _traps                   # 将中断处理函数 _traps 的地址加载到临时寄存器 t0
csrw stvec, t0                  # 将 _traps 的地址写入 stvec 寄存器

# set sie[STIE] = 1 启用始终中断
li t0, 0x20                     # 0x20 是 STIE 位的掩码（即 sie 寄存器的第5位）
csrs sie, t0                    # 将 sie 寄存器的 STIE 位置为 1，启用定时器中断

# set first time interrupt 设置第一次时钟中断
rdtime a0                       # 读取当前时间，存入 a0
li a1, 1000000                  # 设置下次中断时间间隔为 1000000 个时钟周期
add a0, a0, a1                  # 计算下一次时钟中断触发的时间
mv a1, a0                       # 将计算出的时间传递给 a1（sbi_set_timer 的参数）
jal ra, sbi_set_timer           # 调用 sbi_set_timer 设置下一次时钟中断

# set sstatus[SIE] = 1 启用s态中断
li t0, 0x2                      # 0x2 是 SIE 位的掩码（即 sstatus 寄存器的第1位）
csrs sstatus, t0                # 将 sstatus 寄存器的 SIE 位置为 1，启用 S 态中断响应


# Jump to start_kernel 跳转到操作系统内核的入口函数
jal start_kernel                    # 无条件跳转到 start_kernel，不保存返回地址
```

补充代码，在 `head.s` 中对 CSR 进行初始化，并调用 `sbi_set_timer` 调用时钟中断，开启 trap 处理

- 在 `start_kernel` 处打下短点，查看相关CSR 寄存器设置情况：

```
[1] break at 0x0000000080200000 in head.S:6 for _start hit 1 time
[2] break at 0x0000000080200040 in entry.S:7 for _traps
[3] break at 0x0000000080200230 in sbi.c:70 for sbi_set_timer hit 1 time
>>> p/x $sstatus
$1 = 0x8000000200006002
>>> p/x $sie
$2 = 0x20
>>> p/x $stvec
$3 = 0x80200040
>>>
```

可见 `sstatus` 第一位置为 1（即 `sstatus[sie]` = 1），`sie[STIE]` 即 S 态时钟中断，以及 `stvec` 也成功设置，

3.2 实现上下文切换

其中 `sepc` 的值不能通过 lw 和 sw 直接保存在栈上，故而保存时先存 `t0` 再通过 `t0` 间接 sw，恢复寄存器则先通过 `t0` 恢复 `sepc`

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

    # 2. call trap_handler
    csrr a0, scause # check which are the argument registers
    csrr a1, sepc
    jal x1, trap_handler

```

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

addi sp, sp, 256

# 4. Return from trap (restore PC and status)
sret
```

需要注意的是此处从 `_traps` 返回应该使用 `sret` 指令，它会读取 `mepc` 并置 `pc`，返回中断发生前状态，如果使用 `mret` 则会发生错误。

3.3 实现trap处理函数

查询手册中中断 ID 相关内容，得到结果如下：

Interrupt / Exception mcause[XLEN-1]	Exception Code mcause[XLEN-2:0]	Description
1	1	Supervisor software interrupt
1	3	Machine software interrupt
1	5	Supervisor timer interrupt
1	7	Machine timer interrupt
1	9	Supervisor external interrupt
1	11	Machine external interrupt
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store address misaligned
0	7	Store access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	15	Store page fault

图 10.3：RISC-V 异常和中断的原因。中断时 mcause 的最高有效位置 1，同步异常时置 0，且低有效位标识了中断或异常的具体原因。只有在实现了监管者模式时才能处理监管者模式中断和页面错误异常（参见第 10.5 节）。（来自[Waterman and Asanovic 2017]中的表 3.6。）

即 `S Mode` 中时钟中断 id 号为 5，编写如下的 `trap` 处理函数：

```c
#include "stdint.h"
#include "printk.h"
#include "sbi.h"
#include "clock.h"

#define SCAUSE_INTERRUPT_MASK 0x8000000000000000 // 中断标志位
#define SUPERVISOR_TIMER_INTERRUPT_ID 5 // 时钟中断 ID

void trap_handler(uint64_t scause, uint64_t sepc) {
    // 检查中断类型
    if (scause & SCAUSE_INTERRUPT_MASK) {
        // 这是一个中断
        uint64_t interrupt_id = scause & 0x7FFFFFFF; // 获取中断 ID

        // 检查是否是时钟中断
        if (interrupt_id == SUPERVISOR_TIMER_INTERRUPT_ID) {
            // 打印调试信息
            printk("[S] Supervisor Mode Timer Interrupt\n");
            // 处理时钟中断
            clock_set_next_event();
        } else {
            // 其他中断可以直接忽略，打印以供调试
            printk("Unhandled interrupt: interrupt_id = %lu\n", interrupt_id);
        }
    } else {
        // 这是一个异常
        printk("Unhandled exception: scause = %lu, sepc = %lu\n", scause, sepc);
    }
}
```

3.4 实现时钟中断处理函数

实现 `clock.h` 头文件与 `clock.c`，方便其他库函数调用与顺利编译如下：

```c
#ifndef __CLOCK_H__
#define __CLOCK_H__

#include "stdint.h"

uint64_t get_cycles();
void clock_set_next_event();

#endif
```

```c
#include "sbi.h"
#include "stdint.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;

// 获取当前的时钟周期
uint64_t get_cycles() {
    uint64_t cycles;
    __asm__ volatile (
        "rdtime %0"  // 使用 rdtime 指令读取当前时间寄存器的值
        : "=r" (cycles)  // 将读取到的值存储到 cycles 变量中
    );
    return cycles;  // 返回当前时钟周期
}

// 设置下一次时钟中断事件
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
    struct sbiret ret = sbi_set_timer(next);

    // 检查 ret.error 来判断是否设置成功
    if (ret.error != 0) {
        // 错误处理代码，可以打印错误信息
        printk("Error occor in clock_set_next_event\n");
    }
}
```

3.5 修改 test 函数

修改 `test` 函数如下，与 `_trap` 的中断处理函数相呼应，每过一秒钟时间输出 `Kernel is running`，与每秒调用一次的时钟中断匹配

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

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

3.6 编译及测试

经检查，`Makefile` 文件中的依赖与目标文件模式匹配能够支持新增的 `.c .s` 文件正常编译链接，进行 `make run` 结果如下：

```
2024 ZJU Operating System
[S] Supervisor Mode Timer Interrupt
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
[S] Supervisor Mode Timer Interrupt
kernel is running!
```

二、思考题

1. 请总结一下 RISC-V 的 calling convention，并解释 Caller / Callee Saved Register 有什么区别？
   RISC-V 中寄存器关于函数调用的，`ra` 保存函数返回的地址，`sp` 指向栈顶管理函数调用过程中的局部变量和上下文保存。`a0 - a7` 一般作为函数调用参数和返回值 `a0 a1` 常用于返回值，`t0-t6` 为临时寄存器，他们为 `Caller saved register`。而保存寄存器 `s0-s11` 为帧指针局部变量等函数上下文，为 `callee saved register`
   常见的调用过程为：保存 `Caller saved register`，传递参数到 `a0-a7`，使用跳转指令调用函数，保存 `Callee saved register`，函数执行，恢复 `Callee saved register`，传递返回值到 `a0-a1`，使用跳转指令返回，读取返回值，恢复 `Caller saved register`
   这两种寄存器主要区别在于功能不同（前者作为函数参数和临时寄存器，后者存储函数上下文信息）与保存时机不同（前者调用者保存，后者被调用者保存）

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

```
0000000080200000 A BASE_ADDR
0000000080203000 D TIMECLOCK
0000000080203008 d _GLOBAL_OFFSET_TABLE_
0000000080205000 B _ebss
0000000080203008 D _edata
0000000080205000 B _ekernel
0000000080202131 R _erodata
00000000802015cc T _etext
0000000080204000 B _sbss
0000000080203000 D _sdata
0000000080200000 T _skernel
0000000080202000 R _srodata
0000000080200000 T _start
0000000080200000 T _stext
000000008020003c T _traps
0000000080204000 B boot_stack
0000000080205000 B boot_stack_top
0000000080200184 T clock_set_next_event
000000008020015c T get_cycles
0000000080200700 T isspace
0000000080202120 r lowerxdigits.0
0000000080200a54 t print_dec_int
000000008020154c T printk
00000000802006b8 T putc
00000000802009cc t puts_wo_nl
00000000802003e0 T sbi_debug_console_read
0000000080200350 T sbi_debug_console_write
0000000080200468 T sbi_debug_console_write_byte
00000000802001f0 T sbi_ecall
00000000802002c4 T sbi_set_timer
00000000802004f8 T sbi_system_reset
0000000080200624 T start_kernel
0000000080200760 T strtol
0000000080200668 T test
0000000080200594 T trap_handler
0000000080202108 r upperxdigits.1
0000000080200d5c 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。
   在 `main.c` 中使用 `csr_read` 宏读取 `sstatus` 得到结果如下：

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

    uint64_t result;
    result = csr_read(sstatus);
    printk("csr_read result : %llx\n", result);

    test();
    return 0;
}
```

```
2024 ZJU Operating System
csr_read result : 8000000200006002
```

查询手册：

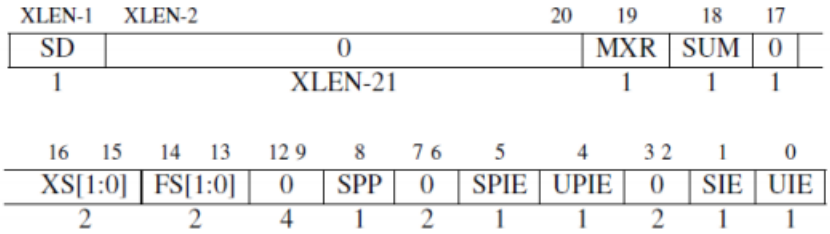

图 10.9：sstatus CSR。sstatus 是 mstatus（图 10.4）的一个子集，因此它们的布局类似。SIE 和 SPIE 中分别保存了当前的和异常发生之前的中断使能，类似于 mstatus 中的 MIE 和 MPIE。RV32 的 XLEN 为 32，RV64 为 40。（来自[Waterman and Asanovic 2017]中的图 4.2；有关其他域的说明请参见该文档的第 4.1 节。）

`csr_read` 结果说明 ：
`UIE` = 0, user模式中断使能关闭
`SIE` = 1, supervisor 模式中断使能开启
`UPIE = SPIE = 0`，陷入中断或异常前，两个模式的中断使能关闭
`SPP = 0` 陷入异常时 S 模式前一个模式为U

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。
   `sscratch` 用于 s 模式的上下文保存和恢复，在 `main.c` 中使用 `csr_write` 宏写入 `sscratch` 得到结果如下：

```
13      uint64_t writeIn = 0x12345678, result = 0x0;
14      csr_write(sscratch, writeIn);
15      result = csr_read(sscratch);
16      printk("csr_read result : %llx\n", result);
2024 ZJU Operating System
csr_read result : 12345678
```

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. 安装交叉编译工具链

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

   2. 进入 linux 内核文件夹，设置编译配置为 `arm64` 默认配置:

```
frey@DESKTOP-5BJAETF:~/linux-6.11-rc7$ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig
*** Default configuration is based on 'defconfig'
#
# configuration written to .config
#
```

   3. 指定需要生成的中间文件：`make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- ./arch/arm64/kernel/sys.i`

```
  LD      arch/arm64/kernel/vdso/vdso.so.dbg
  VDSOSYM include/generated/vdso-offsets.h
  OBJCOPY arch/arm64/kernel/vdso/vdso.so
  CPP     arch/arm64/kernel/sys.i
frey@DESKTOP-5BJAETF:~/linux-6.11-rc7$ cd arch/arm64/kernel
frey@DESKTOP-5BJAETF:~/linux-6.11-rc7/arch/arm64/kernel$ ls
Makefile                cpu_errata.c      entry-ftrace.S  io.c                paravirt.c        reloc_test_core.c  smp_spin_table.c
Makefile.syscalls       cpu_ops.c         entry.S         irq.c               patching.c        reloc_test_syms.S  stacktrace.c
acpi.c                  cpufeature.c      fpsimd.c        jump_label.c        pci.c             relocate_kernel.S  suspend.c
acpi_numa.c             cpuinfo.c         ftrace.c        kaslr.c             perf_callchain.c  return_address.c   sys.c
acpi_parking_protocol.c crash_dump.c      head.S          kexec_image.c       perf_regs.c       sdei.c             sys.i
alternative.c           debug-monitors.c  hibernate-asm.S kgdb.c              pi                setup.c            sys32.c
armv8_deprecated.c      efi-header.S      hibernate.c     kuser32.S           pointer_auth.c    signal.c           sys_compat.c
asm-offsets.c           efi-rt-wrapper.S  hw_breakpoint.c machine_kexec.c      probes            signal32.c         syscall.c
asm-offsets.s           efi.c             hyp-stub.S      machine_kexec_file.c process.c        sigreturn32.S      time.c
cacheinfo.c             elfcore.c         idle.c          module-plts.c       proton-pack.c     sleep.S            topology.c
compat_alignment.c      entry-common.c    image-vars.h    module.c            psci.c            smccc-call.S       trace-events-emulation.h
cpu-reset.S             entry-fpsimd.S    image.h         mte.c               ptrace.c          smp.c              traps.c
```

6. 寻找 Linux v6.0 中 ARM32 RV32 RV64 ×86_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。
  相应架构的系统调用表在 `./arch/*` 路径下，不需要编译对应内核即可查看，但每一种架构路径有所不同
- ARM 32: `arch/arm/tools/syscall.tbl`

syscall.tbl

arch > arm > tools > 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 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
```

- RV 32/64: `arch/riscv/kernel/syscall_table.c`:

arch > riscv > kernel > C syscall_table.c

```c
// SPDX-License-Identifier: GPL-2.0-only
/*
 * Copyright (C) 2009 Arnd Bergmann <arnd@arndb.de>
 * Copyright (C) 2012 Regents of the University of California
 */

#include <linux/linkage.h>
#include <linux/syscalls.h>
#include <asm-generic/syscalls.h>
#include <asm/syscall.h>

#define __SYSCALL_WITH_COMPAT(nr, native, compat) __SYSCALL(nr, native)

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

```

- x 86_64: `arch/x86/entry/syscalls/syscall_64.tbl`

arch > x86 > entry > syscalls > 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
```

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 文件：（Executable and Linkable Format）文件是一种可执行文件格式，它平台无关，可动态链接并可以包含调试信息。ELF文件将其内容分为多个部分，主要包括：
     - **程序头表（Program Header Table）**：定义了程序的执行结构，如加载到内存的地址、大小等。
     - **节头表（Section Header Table）**：包含文件各个节的描述信息，包括代码段、数据段、符号表等
     - **节（Sections）**：`.text`：代码段 `.data`：数据段等
     - **程序表头(Program Header Table)**：文件在运行时需要加载的段
     - **符号表 (Symbol Table)**：包含了程序中所有符号的信息，如函数名、变量名
     - **重定位表 (Relocation Table)**: 包含需要在加载或链接时修改的地址信息

1. 创建一个简单的 c 文件，并编译它，得到 elf 文件：

C hello.c

```c
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}

```

```
frey@DESKTOP-5BJAETF:~$ gcc -o hello hello.c
frey@DESKTOP-5BJAETF:~$ ls
a.out  hello    linux-6.11-rc7         os24fall-stu  test.c
c      hello.c  linux-6.11-rc7.tar.gz  qemu
frey@DESKTOP-5BJAETF:~$ file hello
hello: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
reter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=5c613775d2a4a3
 for GNU/Linux 3.2.0, not stripped
```

2. 使用 readelf 查看 elf 文件:
   - `readelf` 是一个专门用于显示ELF文件信息的工具。它可以显示ELF文件的各种头信息、节信息、符号表等。
   - 查看 elf 头部信息：

```
frey@DESKTOP-5BJAETF:~$ readelf -h 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:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          13968 (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
```

- 查看节表头：

```
frey@DESKTOP-5BJAETF:~$ readelf -S hello
There are 31 section headers, starting at offset 0x3690:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000318  00000318
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.gnu.pr[...] NOTE             0000000000000338  00000338
       0000000000000030  0000000000000000   A       0     0     8
  [ 3] .note.gnu.bu[...] NOTE             0000000000000368  00000368
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .note.ABI-tag     NOTE             000000000000038c  0000038c
       0000000000000020  0000000000000000   A       0     0     4
  [ 5] .gnu.hash         GNU_HASH         00000000000003b0  000003b0
       0000000000000024  0000000000000000   A       6     0     8
  [ 6] .dynsym           DYNSYM           00000000000003d8  000003d8
       00000000000000a8  0000000000000018   A       7     1     8
  [ 7] .dynstr           STRTAB           0000000000000480  00000480
       000000000000008d  0000000000000000   A       0     0     1
  [ 8] .gnu.version      VERSYM           000000000000050e  0000050e
       000000000000000e  0000000000000002   A       6     0     2
  [ 9] .gnu.version_r    VERNEED          0000000000000520  00000520
       0000000000000030  0000000000000000   A       7     1     8
  [10] .rela.dyn         RELA             0000000000000550  00000550
       00000000000000c0  0000000000000018   A       6     0     8
```

- 查看程序头表：

```
frey@DESKTOP-5BJAETF:~$ readelf -l hello

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
```

- 查看符号表：

```
frey@DESKTOP-5BJAETF:~$ readelf -s hello

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 hello.c
```

- 查看重定位表：

```
frey@DESKTOP-5BJAETF:~$ readelf -r hello

Relocation section '.rela.dyn' at offset 0x550 contains 8 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000003db8  000000000008 R_X86_64_RELATIVE                    1140
000000003dc0  000000000008 R_X86_64_RELATIVE                    1100
000000004008  000000000008 R_X86_64_RELATIVE                    4008
000000003fd8  000100000006 R_X86_64_GLOB_DAT 0000000000000000 __libc_start_main@GLIBC_2.34
 + 0
000000003fe0  000200000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_deregisterTM[...] + 0
000000003fe8  000400000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000003ff0  000500000006 R_X86_64_GLOB_DAT 0000000000000000 _ITM_registerTMCl[...] + 0
000000003ff8  000600000006 R_X86_64_GLOB_DAT 0000000000000000 __cxa_finalize@GLIBC_2.2.5 +
 0

Relocation section '.rela.plt' at offset 0x610 contains 1 entry:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000003fd0  000300000007 R_X86_64_JUMP_SLO 0000000000000000 puts@GLIBC_2.2.5 + 0
```

3. 使用 objdump 查看 elf 文件：
   - `objdump` 是一个功能更为广泛的二进制文件分析工具。除了显示ELF文件的信息外，它还可以反汇编代码段，显示调试信息等。

- 显示文件头信息（节表头）：

```
frey@DESKTOP-5BJAETF:~$ objdump -h hello

hello:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000030  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000368  0000000000000368  00000368  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000038c  000000000000038c  0000038c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003b0  00000000000003b0  000003b0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynsym       000000a8  00000000000003d8  00000000000003d8  000003d8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .dynstr       0000008d  0000000000000480  0000000000000480  00000480  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version  0000000e  000000000000050e  000000000000050e  0000050e  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .gnu.version_r 00000030  0000000000000520  0000000000000520  00000520  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.dyn     000000c0  0000000000000550  0000000000000550  00000550  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .rela.plt     00000018  0000000000000610  0000000000000610  00000610  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

- 查看所有节的内容：

```
frey@DESKTOP-5BJAETF:~$ objdump -s hello

hello:     file format elf64-x86-64

Contents of section .interp:
 0318 2f6c6962 36342f6c 642d6c69 6e75782d  /lib64/ld-linux-
 0328 7838362d 36342e73 6f2e3200           x86-64.so.2.
Contents of section .note.gnu.property:
 0338 04000000 20000000 05000000 474e5500  .... .......GNU.
 0348 020000c0 04000000 03000000 00000000  ................
 0358 028000c0 04000000 01000000 00000000  ................
Contents of section .note.gnu.build-id:
 0368 04000000 14000000 03000000 474e5500  ............GNU.
 0378 5c613775 d2a4a3eb 772fb7c4 eaaf368c  \a7u....w/....6.
 0388 86a13de6                             ..=.
Contents of section .note.ABI-tag:
 038c 04000000 10000000 01000000 474e5500  ............GNU.
 039c 00000000 03000000 02000000 00000000  ................
Contents of section .gnu.hash:
 03b0 02000000 06000000 01000000 06000000  ................
 03c0 00008100 00000000 06000000 00000000  ................
 03d0 d165ce6d                             .e.m
Contents of section .dynsym:
 03d8 00000000 00000000 00000000 00000000  ................
 03e8 00000000 00000000 06000000 12000000  ................
 03f8 00000000 00000000 00000000 00000000  ................
 0408 48000000 20000000 00000000 00000000  H... ...........
 0418 00000000 00000000 01000000 12000000  ................
 0428 00000000 00000000 00000000 00000000  ................
```

- 反汇编代码段：

```
frey@DESKTOP-5BJAETF:~$ objdump -d hello

hello:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax        # 3fe8 <__gmon_st
rt__@Base>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   call   *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      ret

Disassembly of section .plt:

0000000000001020 <.plt>:
    1020:       ff 35 9a 2f 00 00       push   0x2f9a(%rip)        # 3fc0 <_GLOBAL_OFFSET_
TABLE_+0x8>
    1026:       ff 25 9c 2f 00 00       jmp    *0x2f9c(%rip)        # 3fc8 <_GLOBAL_OFFSET
_TABLE_+0x10>
    102c:       0f 1f 40 00             nopl   0x0(%rax)
    1030:       f3 0f 1e fa             endbr64
```

8. 在我们使用 make run 时，OpenSBI 会产生如下输出：
   - 通过查看 RISC-V Privileged Spec 中的 `medeleg` 和 `mideleg` 部分，解释上面 `MIDELEG` 和 `MEDELEG` 值的含义。
   - Boot HART MIDELEG : 0×0000000000000222
   - Boot HART MEDELEG : 0×000000000000b109
     这两个寄存器统称 machine trap delegation registers，mideleg（machind interrupt delegation register）用于保存哪些中断要委托给 S 模式进行处理。medeleg（machine exception delegation register）用于保存哪些异常要委托给 S 模式进行处理。
     上图中 `MIDELEG` 位 1,5,9 置 1代表委托 `Supervisor Software Interrupt`、`Supervisor Timer Interrupt`、`Supervisor External Interrupt` 给 S 模式，其余给 M 模式处理。
     而 `MEDELEG` 表示位 0,3,9,12 置 1 代表委托了 `Instruction Address Misaligned`、`Breakpoint`、`Environment Call from S-mode`、`Instruction Page Fault` 给 S 模式，其余给 M 模式处理。

三、讨论心得

1. 使用 makefile 时需要注意编译命令的可选项，比如包含路径；在本实验中子文件夹 lib 中的 makefile 编译 printk 需要在编译命令加入包含路径· `${INCLUDE}`。
2. 对于每一个子文件的完成，可以先注释掉对于后面未完成函数的调用，先测试本文件基本功能是否正常，保证每一个子文件 debug 完成，减少最后整体工程调试工作量与出错可能性。