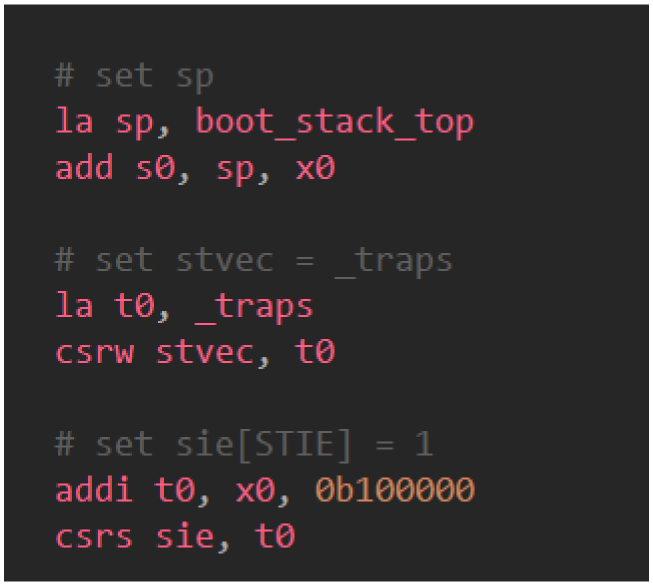
lab2

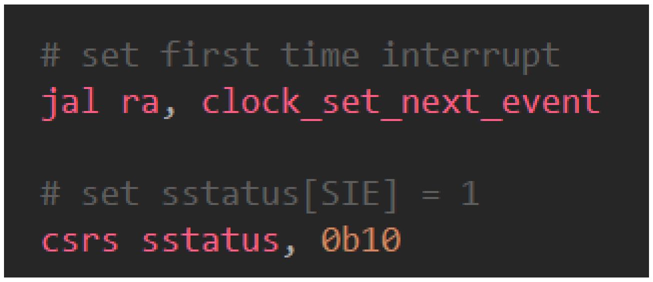
4.2 开启trap处理

简单来说，就是处理 `head.S` 中的 `_start` 逻辑：

1. 首先是 `set stvec` ，这只需用 `la` 获得 `_traps` ，再用 `csrw` 写入 `stvec` 即可。
2. 之后是设置 `sie` 的 `STIE` 位，这一位在第5位，所以我们利用 `csrs` 命令将第5位置1。注意不能用 `csrsi` 命令，因为其只支持5位imm的0扩展，也就是只能管到第4位。
3. `set first time interrupt` ，只要调用 `clock_set_next_event()` 即可。注意，一定要提前设置好 `sp` 的值，不然会调用错误。这也是我踩的坑。
4. 设置 `sstatus` 的 `SIE` 位为1，也是用 `csrsi` 即可。

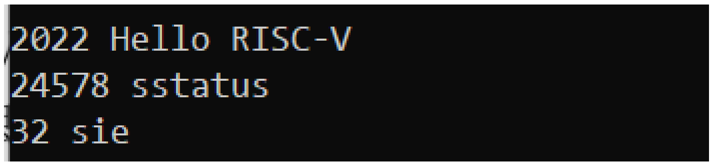


```
# set sp
la sp, boot_stack_top
add s0, sp, x0

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

# set sie[STIE] = 1
addi t0, x0, 0b100000
csrs sie, t0
```



```
# set first time interrupt
jal ra, clock_set_next_event

# set sstatus[SIE] = 1
csrs sstatus, 0b10
```

- 最好在完成 `set first time interrupt` 之前检验 `sie` 和 `sstatus` 是否设置正确。

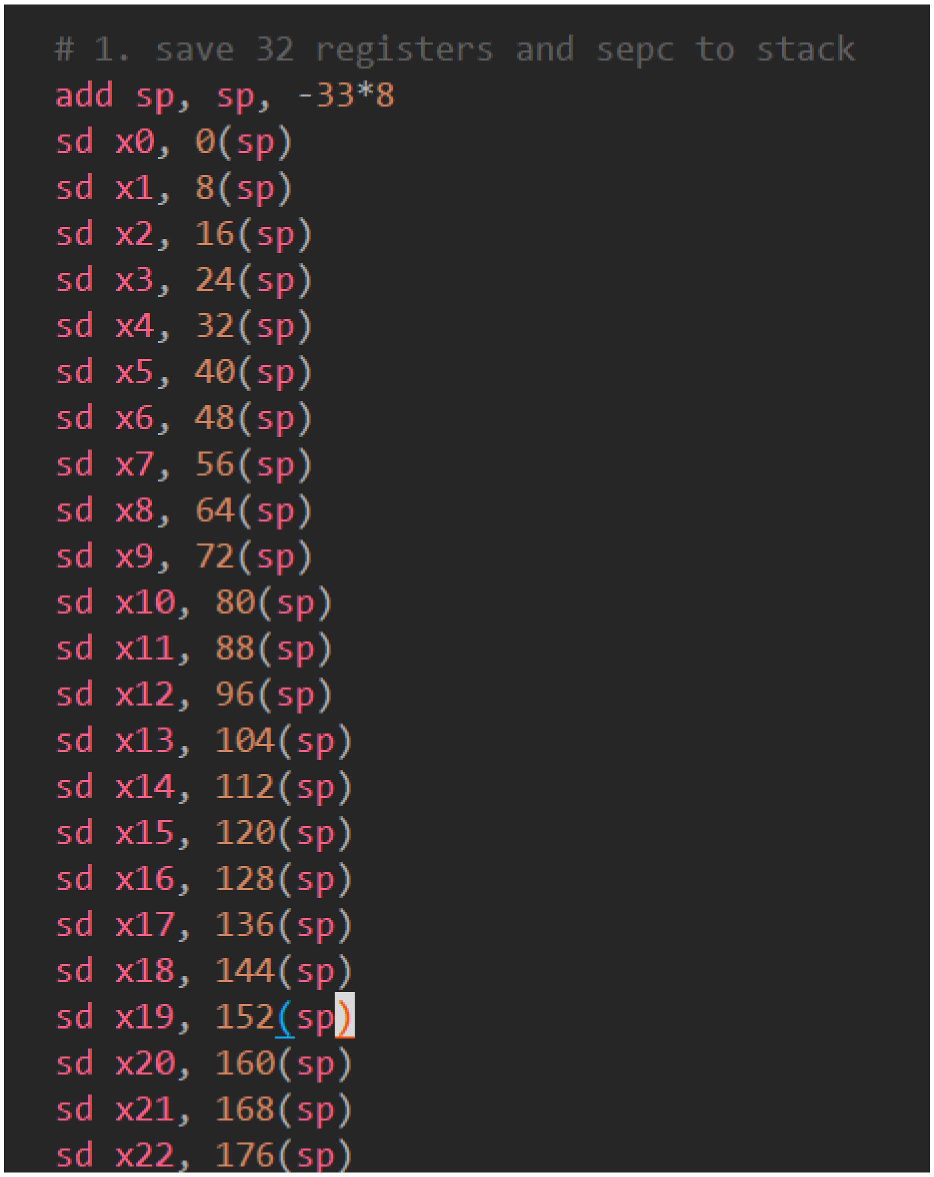


```
2022 Hello RISC-V
24578 sstatus
32 sie
```

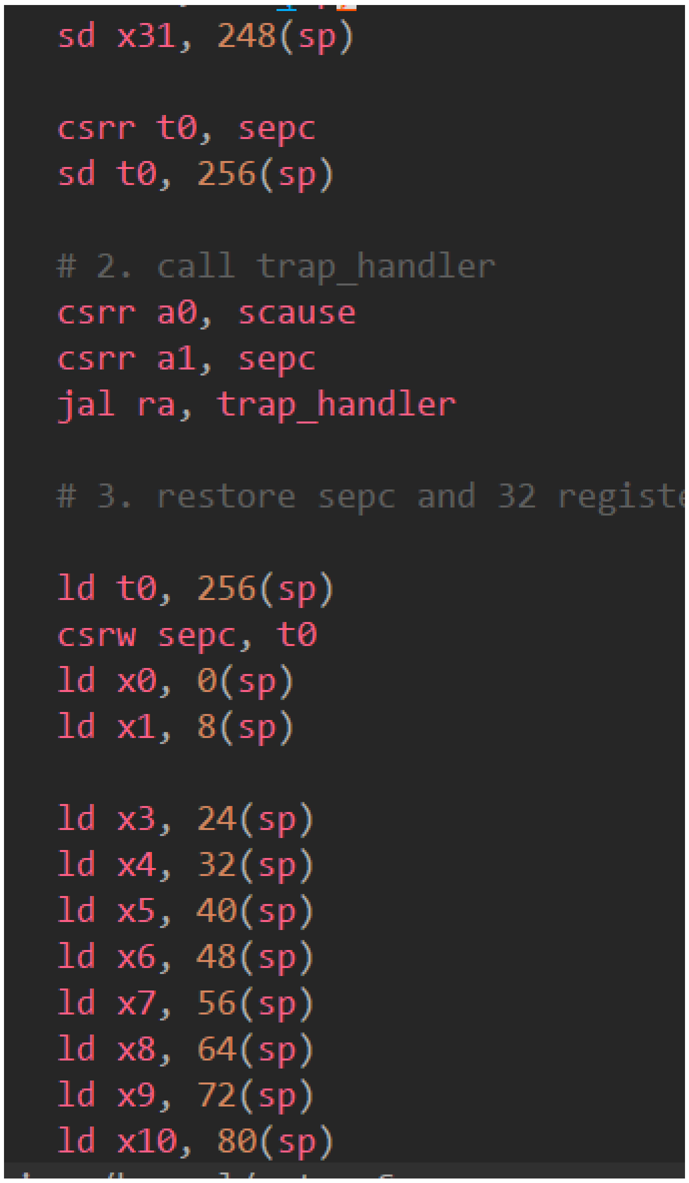
4.3 实现上下文切换

实现 `_trap` 的上下文。

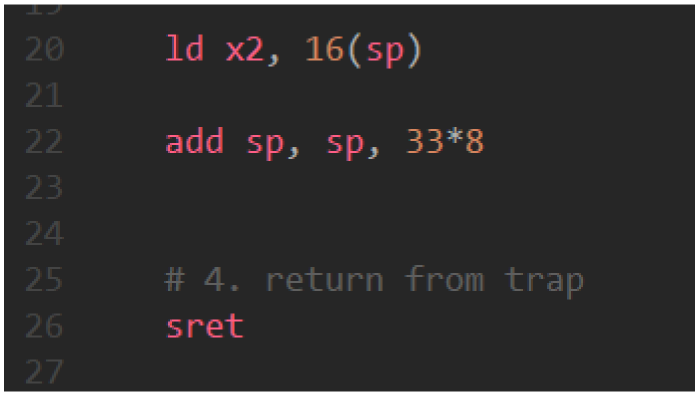
1. 先实现上下文切换逻辑，保证 `save` 和 `restore` 的正确实现。因此我们先存储并取出32个 `reg` 和 `sepc` 的值。 `sepc` 用 `csrr` 读出，注意等 `t0` 存好后再使用，同样等 `sepc` 取出后再取 `t0` .最后取 `sp` 。



```
# 1. save 32 registers and sepc to stack
add sp, sp, -33*8
sd x0, 0(sp)
sd x1, 8(sp)
sd x2, 16(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)
```

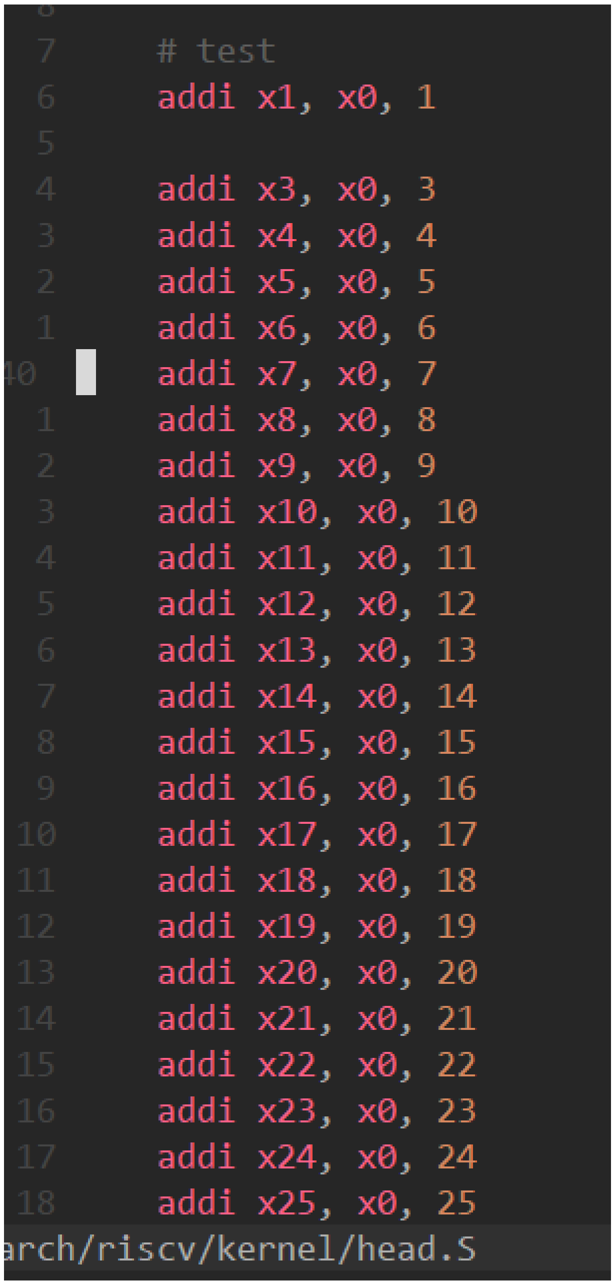


```
sd x31, 248(sp)

csrr t0, sepc
sd t0, 256(sp)

# 2. call trap_handler
csrr a0, scause
csrr a1, sepc
jal ra, trap_handler

# 3. restore sepc and 32 registe

ld t0, 256(sp)
csrw sepc, t0
ld x0, 0(sp)
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)
```

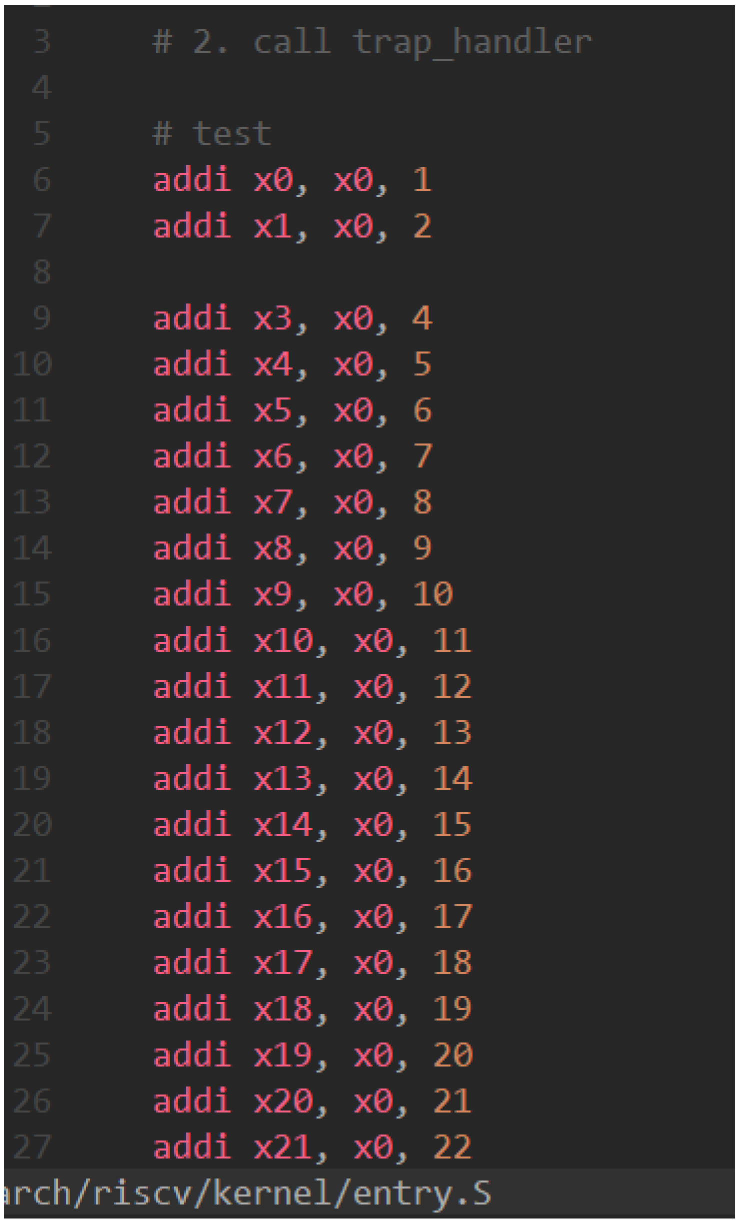


```
    ld x2, 16(sp)

    add sp, sp, 33*8


    # 4. return from trap
    sret

```

2. 之后，我写了测试代码简单改变了 `x0-x31` 的值用来验证，看结果说明正确写好了。

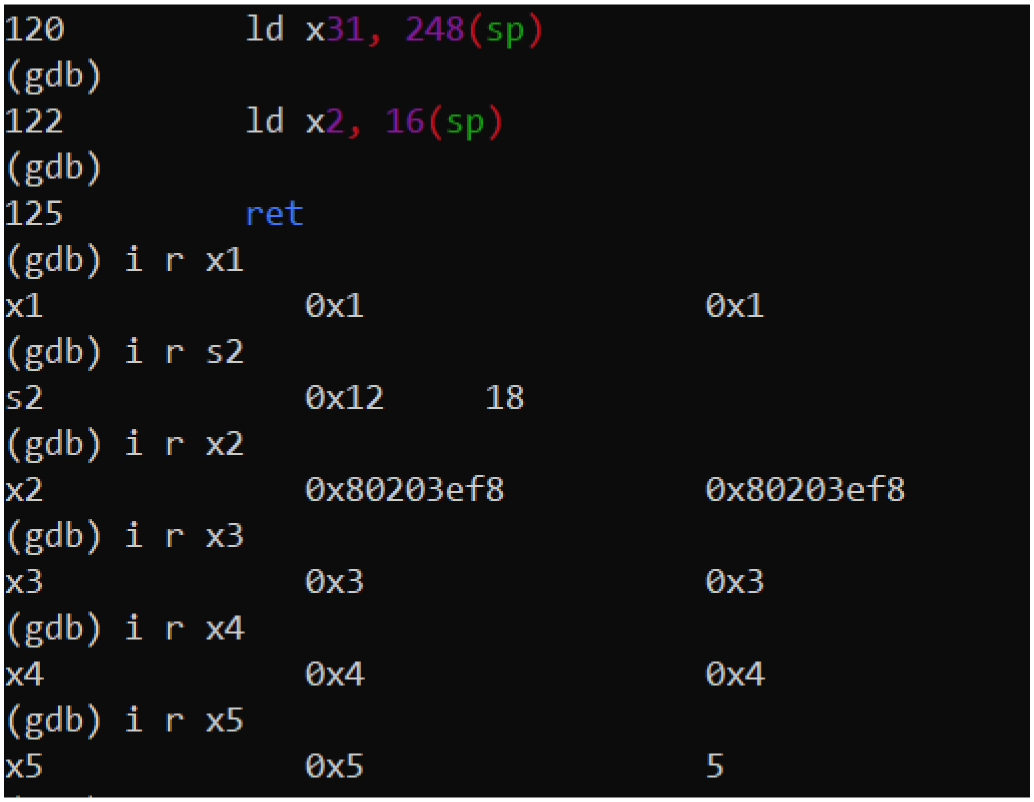


```
    # test
    addi x1, x0, 1

    addi x3, x0, 3
    addi x4, x0, 4
    addi x5, x0, 5
    addi x6, x0, 6
    addi x7, x0, 7
    addi x8, x0, 8
    addi x9, x0, 9
    addi x10, x0, 10
    addi x11, x0, 11
    addi x12, x0, 12
    addi x13, x0, 13
    addi x14, x0, 14
    addi x15, x0, 15
    addi x16, x0, 16
    addi x17, x0, 17
    addi x18, x0, 18
    addi x19, x0, 19
    addi x20, x0, 20
    addi x21, x0, 21
    addi x22, x0, 22
    addi x23, x0, 23
    addi x24, x0, 24
    addi x25, x0, 25
arch/riscv/kernel/head.S
```



```
    # 2. call trap_handler

    # test
    addi x0, x0, 1
    addi x1, x0, 2

    addi x3, x0, 4
    addi x4, x0, 5
    addi x5, x0, 6
    addi x6, x0, 7
    addi x7, x0, 8
    addi x8, x0, 9
    addi x9, x0, 10
    addi x10, x0, 11
    addi x11, x0, 12
    addi x12, x0, 13
    addi x13, x0, 14
    addi x14, x0, 15
    addi x15, x0, 16
    addi x16, x0, 17
    addi x17, x0, 18
    addi x18, x0, 19
    addi x19, x0, 20
    addi x20, x0, 21
    addi x21, x0, 22
```

rch/riscv/kernel/entry.S



```
120             ld x31, 248(sp)
(gdb)
122             ld x2, 16(sp)
(gdb)
125             ret
(gdb) i r x1
x1             0x1                 0x1
(gdb) i r s2
s2             0x12     18
(gdb) i r x2
x2             0x80203ef8          0x80203ef8
(gdb) i r x3
x3             0x3                 0x3
(gdb) i r x4
x4             0x4                 0x4
(gdb) i r x5
x5             0x5                 5
```

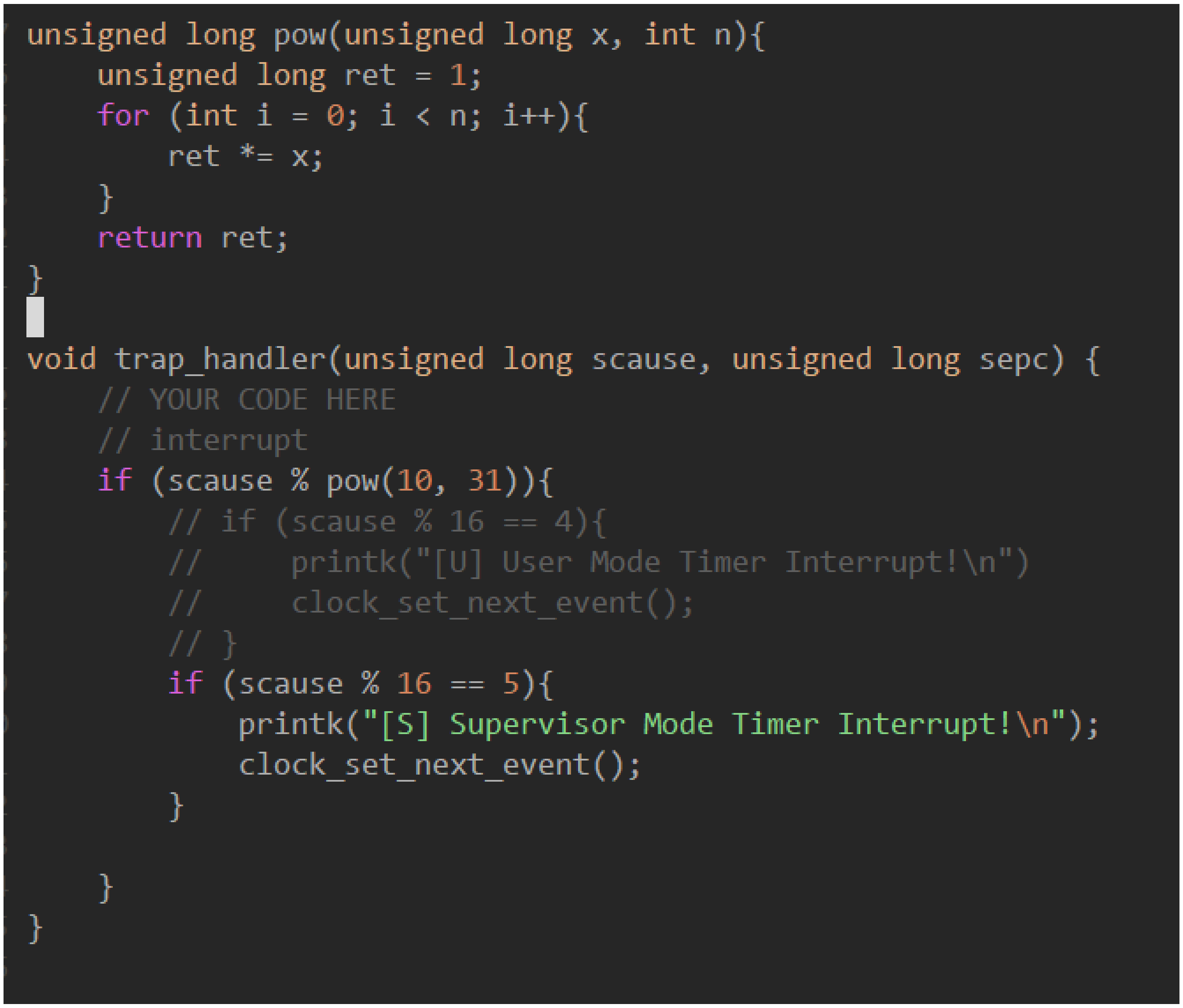
3. 验证上下文切换正确后，再实现 `call trap_handler` 。实际上就是把 `scause` 和 `sepc` 分别读入 `a0` 和 `a1` ，再 `jal ra, trap_handler` .
4. 还有一点，当你load出sp的时候，一定要记得加上之间的33*8，否则栈顶的位置就不对了，这是我之后发现的坑。
5. 注意，最后的return应该是 `sret` ，因为这是在内核层面上的。

4.4 实现 trap 处理函数

查看 `scause` 相关信息，可以知道我们要处理的是 `interrupt` 置1，`exception code` 置5的情况，于是我们先实现 `pow` 函数，再分别判断 `interrupt` 与 `exception code` 。

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2–3	*Reserved for future standard use*
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6–7	*Reserved for future standard use*
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10–11	*Reserved for future standard use*
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

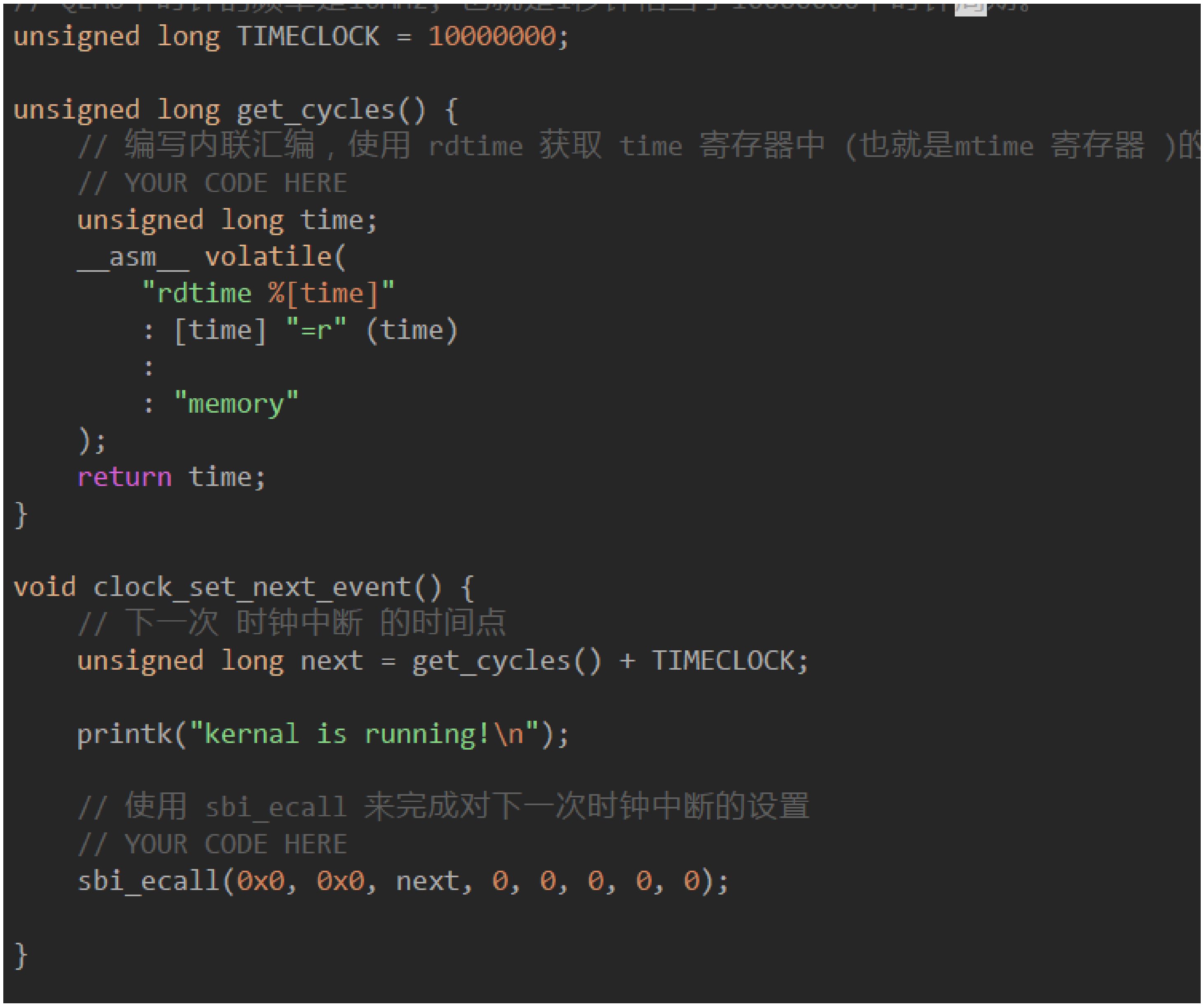
Table 4.2: Supervisor cause register (`scause`) values after trap. Synchronous exception priorities are given by Table 3.7.



```
unsigned long pow(unsigned long x, int n){
    unsigned long ret = 1;
    for (int i = 0; i < n; i++){
        ret *= x;
    }
    return ret;
}

void trap_handler(unsigned long scause, unsigned long sepc) {
    // YOUR CODE HERE
    // interrupt
    if (scause % pow(10, 31)){
        // if (scause % 16 == 4){
        //     printk("[U] User Mode Timer Interrupt!\n")
        //     clock_set_next_event();
        // }
        if (scause % 16 == 5){
            printk("[S] Supervisor Mode Timer Interrupt!\n");
            clock_set_next_event();
        }

    }
}
```

4.5 实现时钟中断相关函数

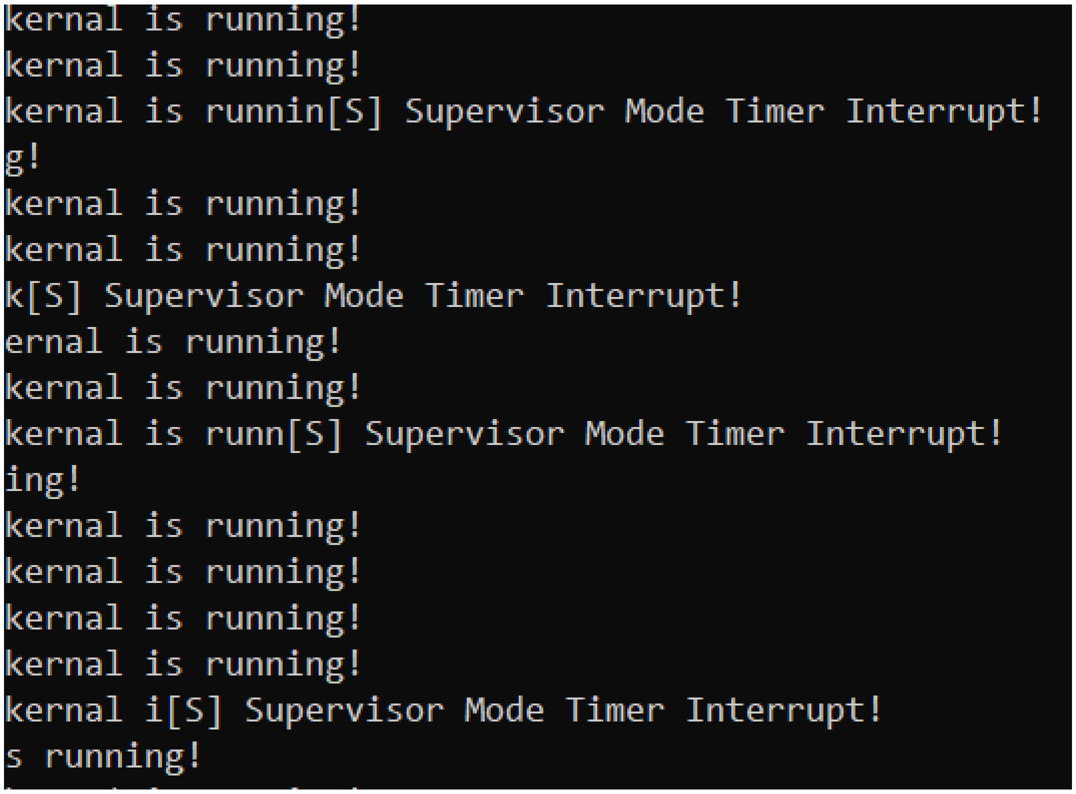
`get_cycles()` 用 `rdtime` 拿到时间， `clock_set_next_event()` 调用 `sbi_ecall()` 设置时钟中断。都很简单，不加赘述。



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

unsigned long 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
    // YOUR CODE HERE
    unsigned long time;
    __asm__ volatile(
        "rdtime %[time]"
        : [time] "=r" (time)
        :
        : "memory"
    );
    return time;
}

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

    printk("kernal is running!\n");

    // 使用 sbi_ecall 来完成对下一次时钟中断的设置
    // YOUR CODE HERE
    sbi_ecall(0x0, 0x0, next, 0, 0, 0, 0, 0);

}
```

4.6 编译测试

可以看到，结果符合要求。注意，根据助教群的说法，这个 `kernel is running` 是在test循环中的，于是我改了CLK为100，结果如下图所示。

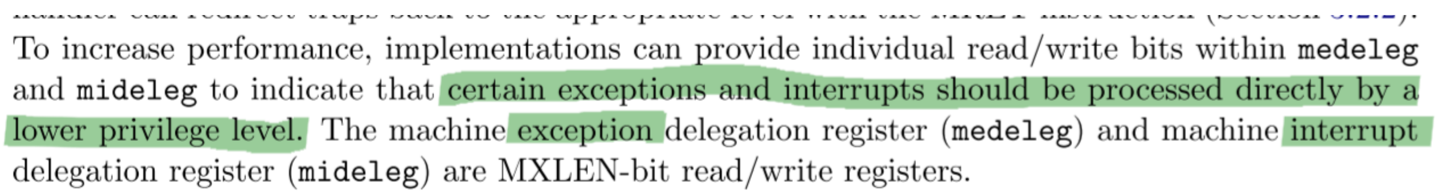


```
kernal is running!
kernal is running!
kernal is runnin[S] Supervisor Mode Timer Interrupt!
g!
kernal is running!
kernal is running!
k[S] Supervisor Mode Timer Interrupt!
ernal is running!
kernal is running!
kernal is runn[S] Supervisor Mode Timer Interrupt!
ing!
kernal is running!
kernal is running!
kernal is running!
kernal is running!
kernal i[S] Supervisor Mode Timer Interrupt!
s running!
```

思考题

1. 查看 RISC-V Privileged Spec 中的 medeleg 和 mideleg 解释上面 MIDELEG 值的含义。

看解释，是指哪些 `execption` 或者 `interrupt` 被委托给 `S-Mode` 执行。
`interrupt` 低16位是0000 0010 0010 0010，也就是第1,5,9位，也就是 `Supervisor` 的 `software` / `timer` / `external interrupt` 交给 `S-Mode` 处理;
`exception` 低16位是1011 0001 0000 1001，也就是第0,3,8,12,13,15位，也就是对应的交给 `S-Mode` 处理;



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.

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	*Reserved for future standard use*
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	*Reserved for future standard use*
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	*Reserved for future standard use*
1	11	Machine external interrupt
1	12–15	*Reserved for future standard use*
1	≥16	*Reserved for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved for future standard use*
0	15	Store/AMO page fault
0	16–23	*Reserved for future standard use*
0	24–31	*Reserved for custom use*
0	32–47	*Reserved for future standard use*
0	48–63	*Reserved for custom use*
0	≥64	*Reserved for future standard use*

Table 3.6: Machine cause register (`mcause`) values after trap.