

# 实验5——RV64 时钟中断处理

姓名：汤尧

学号：3200106252

课程名称：计算机系统II

指导老师：申文博

## 一 实验目的

- 学习 RISC-V 的异常处理相关寄存器与指令，完成对异常处理的初始化。
- 理解 CPU 上下文切换机制，并正确实现上下文切换功能。
- 编写异常处理函数，完成对特定异常的处理。
- 调用 OpenSBI 提供的接口，完成对时钟中断事件的设置。

## 二 实验原理和内容

### 2.1 相关寄存器

除了32个通用寄存器之外，RISC-V 架构还有大量的 **控制状态寄存器** Control and Status Registers(CSRs)，下面将介绍几个和异常机制相关的重要寄存器。

Supervisor Mode 异常相关寄存器:

- sstatus ( Supervisor Status Register )中存在一个 SIE ( Supervisor Interrupt Enable ) 比特位，当该比特位设置为 1 时，会对所有的 S 态异常**响应**，否则将会禁用所有 S 态异常。
- sie ( Supervisor Interrupt Eable Register )。在 RISC-V 中，Interrupt 被划分为三类 Software Interrupt, Timer Interrupt, External Interrupt。在开启了 sstatus[SIE]之后，系统会根据 sie 中的相关比特位来决定是否对该 Interrupt **进行处理**。
- stvec ( Supervisor Trap Vector Base Address Register ) 即所谓的”中断向量表基址”。stvec 有两种模式：Direct 模式，适用于系统中只有一个中断处理程序，其指向中断处理入口函数（本次实验中我们所用的模式）。Vectored 模式，指向中断向量表，适用于系统中有多个中断处理程序（该模式可以参考 [RISC-V 内核源码](#)）。
- scause ( Supervisor Cause Register ), 会记录异常发生的原因，还会记录该异常是 Interrupt 还是 Exception。
- sepc ( Supervisor Exception Program Counter ), 会记录触发异常的那条指令的地址。

Machine Mode 异常相关寄存器:

- 类似于 Supervisor Mode，Machine Mode 也有相对应的寄存器，但由于本实验同学不需要操作这些寄存器，故不在此作介绍。

以上寄存器的详细介绍请同学们参考 [RISC-V Privileged Spec](#)

## 2.2 相关特权指令

- `ecall` (Environment Call), 当我们在 S 态执行这条指令时, 会触发一个 `ecall-from-s-mode-exception`, 从而进入 M 模式中的中断处理流程(如设置定时器等); 当我们在 U 态执行这条指令时, 会触发一个 `ecall-from-u-mode-exception`, 从而进入 S 模式中的中断处理流程(常用来进行系统调用)。
- `sret` 用于 S 态异常返回, 通过 `sepc` 来设置 `pc` 的值, 返回到之前程序继续运行。

以上指令的详细介绍请同学们参考 [RISC-V Privileged Spec](#)

## 2.3 上下文处理

由于在处理异常时, 有可能会改变系统的状态。所以在真正处理异常之前, 我们有必要对系统的当前状态进行保存, 在异常处理完成之后, 我们再将系统恢复至原先的状态, 就可以确保之前的程序继续正常运行。

这里的系统状态通常是指寄存器, 这些寄存器也叫做CPU的上下文(Context)。

## 2.4 异常处理程序

异常处理程序根据 `scause` 的值, 进入不同的处理逻辑, 在本次试验中我们需要关心的只有 `Supervisor Timer Interrupt`。

## 2.5 时钟中断

时钟中断需要 CPU 硬件的支持。CPU 以“时钟周期”为工作的基本时间单位, 对逻辑门的时序电路进行同步。而时钟中断实际上就是“每隔若干个时钟周期执行一次的程序”。下面介绍与时钟中断相关的寄存器以及如何产生时钟中断。

- `mtime` 与 `mtimecmp` (Machine Timer Register)。`mtime` 是一个实时计时器, 由硬件以恒定的频率自增。`mtimecmp` 中保存着下一次时钟中断发生的时间点, 当 `mtime` 的值大于或等于 `mtimecmp` 的值, 系统就会触发一次时钟中断。因此我们只需要更新 `mtimecmp` 中的值, 就可以设置下一次时钟中断的触发点。OpenSBI 已经为我们提供了更新 `mtimecmp` 的接口 `sbi_set_timer` (见 lab4 4.4节)。
- `mcounteren` (Counter-Enable Registers)。由于 `mtime` 是属于 M 态的寄存器, 我们在 S 态无法直接对其读写, 幸运的是 OpenSBI 在 M 态已经通过设置 `mcounteren` 寄存器的 `TM` 比特位, 让我们可以在 S 态中可以通过 `time` 这个只读寄存器读取到 `mtime` 的当前值, 相关汇编指令是 `rdtime`。

# 三 实验环境

- Ubuntu虚拟机
- Docker in Lab3

# 四 实验步骤与分析

## 4.1 准备工程

新建文件夹, 进行文件的修改。

## 4.2 开启异常处理

在运行 `start_kernel` 之前，我们要对上面提到的 CSR 进行初始化，初始化包括以下几个步骤：

1. 设置 `stvec`，将 `_traps` (`_trap` 在 4.3 中实现) 所表示的地址写入 `stvec`，这里我们采用 Direct 模式，而 `_traps` 则是中断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 4.5 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。

按照下方模版修改 `arch/riscv/kernel/head.S`，并补全 `_start` 中的逻辑。

`stvec`：保存s模式的trap向量基址。

`sie`：sip寄存器是一个xlen位的读/写寄存器，包含关于挂起中断的信息。

**csrci** `csr, zimm[4:0]`

`CSRs[csr] |= zimm`

立即数置位控制状态寄存器 (*Control and Status Register Set Immediate*). 伪指令 (Pseudoinstruction), RV32I and RV64I.

对于五位的零扩展的立即数中每一个为 1 的位，把控制状态寄存器 `csr` 的对应位清零，等同于 `csrrsi x0, csr, zimm`.

**csrr** `rd, csr`

`x[rd] = CSRs[csr]`

读控制状态寄存器 (*Control and Status Register Read*). 伪指令 (Pseudoinstruction), RV32I and RV64I.

把控制状态寄存器 `csr` 的值写入 `x[rd]`，等同于 `csrrs rd, csr, x0`.

调试中遇到问题：寄存器从0开始计位。

如，`sstatus`寄存器

XLEN-1	XLEN-2	19	18	17	16	15	14	13	12	9	8	7	6	5	4	3	2	1	0
SD	0	PUM	0	XS[1:0]	FS[1:0]	0	SPP	0	SPIE	UPIE	0	SIE	UIE						
1	XLEN-20	1	1	2	2	4	1	2	1	1	2	1	1						

Figure 4.1: Supervisor-mode status Register.

<https://blog.csdn.net/Pandacooker>

`sie`寄存器

XLEN-1	10	9	8	7	6	5	4	3	2	1	0
0	SEIP	UEIP	0	STIP	UTIP	0	SSIP	USIP			
XLEN-10	1	1	2	1	1	2	1	1			

Figure 4.3: Supervisor interrupt-pending register (`sip`).

XLEN-1	10	9	8	7	6	5	4	3	2	1	0
0	SEIE	UEIE	0	STIE	UTIE	0	SSIE	USIE			
XLEN-10	1	1	2	1	1	2	1	1			

Figure 4.4: Supervisor interrupt-enable register (`sie`).

<https://blog.csdn.net/Pandacooker>

其实需要修改的是第二位(2)和第六位(0x20)，因为是从0开始计数。

```
1  ## set stvec = _traps
2      la a0, _traps
3      csrw stvec, a0
4
5  ## set sie[SIE] = 1
6      #stip is located in fifth bit in sie
7      csrr a0, sie
8      ori a0, a0, 32
9      csrw sie, a0
10
11 ## set first time interrupt
12      rdttime a0
13      li a1, 100000000
14      add a0, a0, a1
15      li a7, 0
16      li a1, 0
17      li a2, 0
18      li a3, 0
19      li a4, 0
20      li a5, 0
21      li a6, 0
22      ecall
23
24 ## set sstatus[SIE] = 1
25      #sie is located in second bit in sstatus
26      csrr a0, sstatus
27      ori a0, a0, 2
28      csrw sstatus, a0
```

## 4.3 实现上下文切换

我们要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 中的值传入异常处理函数 trap\_handler（trap\_handler 在 4.4 中介绍），我们将会 trap\_handler 中实现对异常的处理。

sepc寄存器

当陷阱发生时，RISC-V将程序计数器的值保存在这里，因为随后pc的值将被stvec的值覆盖掉；  
sret指令拷贝sepc的值到pc中；  
内核可向spec中写入值来控制sret返回到哪里；

4. 在完成对异常的处理之后，我们从内存中（栈上）恢复CPU的寄存器（上下文）。
5. 从 trap 中返回。

```
1  .....#saving registers(omitted)
2      csrr a1, sepc
3      sd a1, -256(sp)
4      addi sp, sp, -256
5
```

```

6  ## 2. call trap_handler
7      li a0, 0x80000005
8      call trap_handler
9
10 ## 3. restore sepc and 32 registers (a(sp) should be restore last) from stack
11      ld t6, 0(sp)
12      csw sepc, t6
13      addi sp, sp, 8
14      .....#Loading regisers (omitted)

```

## 4.4 实现异常处理函数

1. 在 arch/riscv/kernel/ 目录下添加 trap.c 文件。
2. 在 trap.c 中实现异常处理函数 trap\_handler(), 其接收的两个参数分别是 scause 和 sepc 两个寄存器中的值。

3.

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。)

```

1  #include "types.h"
2  #include "printk.h"
3  #include "sbi.h"
4  void trap_handler(unsigned long scause, unsigned long sepc) {
5      // 通过 `scause` 判断trap类型
6      // 如果是interrupt 判断是否是timer interrupt
7      // 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
8      // `clock_set_next_event()` 见 4.5 节
9      // 其他interrupt / exception 可以直接忽略
10     if((int)scause<0){

```

```

11         if(scause%16==5) {
12             printk("[S] Supervisor Mode Timer Interrupt\nkernel is running\n");
13             clock_set_next_event();
14         }
15     }
16 }

```

## 4.5 实现时钟中断相关函数

1. 在 arch/riscv/kernel/ 目录下添加 clock.c 文件。
2. 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。
3. 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。

Function Name	Function ID	Extension ID
sbi_set_timer (设置时钟相关寄存器)	0	0x00
sbi_console_putchar (打印字符)	0	0x01
sbi_console_getchar (接收字符)	0	0x02
sbi_shutdown (关机)	0	0x08

```

1  #include "types.h"
2  #include "sbi.h"
3  // QEMU中时钟的频率是10MHz, 也就是1秒钟相当于10000000个时钟周期。
4  unsigned long TIMECLOCK = 10000000;
5
6  unsigned long get_cycles() {
7      // 使用 rdtime 编写内联汇编, 获取 time 寄存器中 (也就是mtime 寄存器 )的值并返回
8      unsigned long time;
9      __asm__ volatile (
10         "rdtime a0\n"
11         "mv %[time], a0\n"
12         :[time] "=r" (time)
13         :
14         : "memory"
15     );
16     return time;
17 }
18
19
20 void clock_set_next_event() {
21     // 下一次 时钟中断 的时间点
22     unsigned long next = get_cycles() + TIMECLOCK;
23
24     // 使用 sbi_ecall 来完成对下一次时钟中断的设置
25     sbi_ecall(00,0,0,0,0,0,0,0);
26 }

```

## 4.6 编译及测试

由于加入了一些新的.c文件,可能需要修改一些Makefile文件,请同学自己尝试修改,使项目可以编译并运行。

一个窗口make debug

```
root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/lab5

make[1]: Leaving directory '/have-fun-debugging/sys2lab-21fall/lab5/arch/riscv'

Build Finished OK
root@f7bf9d5e9abb:/have-fun-debugging/sys2lab-21fall/lab5# vsp
bash: vsp: command not found
root@f7bf9d5e9abb:/have-fun-debugging/sys2lab-21fall/lab5# make debug
make -C lib all
make[1]: Entering directory '/have-fun-debugging/sys2lab-21fall/lab5/lib'
make[1]: 'all' is up to date.
make[1]: Leaving directory '/have-fun-debugging/sys2lab-21fall/lab5/lib'
make -C init all
make[1]: Entering directory '/have-fun-debugging/sys2lab-21fall/lab5/init'
make[1]: 'all' is up to date.
make[1]: Leaving directory '/have-fun-debugging/sys2lab-21fall/lab5/init'
make -C arch/riscv all
make[1]: Entering directory '/have-fun-debugging/sys2lab-21fall/lab5/arch/riscv'
make -C kernel all
make[2]: Entering directory '/have-fun-debugging/sys2lab-21fall/lab5/arch/riscv/kernel'
make[2]: Nothing to be done for 'all'.
make[2]: Leaving directory '/have-fun-debugging/sys2lab-21fall/lab5/arch/riscv/kernel'
riscv64-unknown-elf-ld -T kernel/vmlinux.lds kernel/*.o ../../init/*.o ../../lib/*.o -o ../../vmlin
ux
riscv64-unknown-elf-objcopy -O binary ../../vmlinux ./boot/Image
nm ../../vmlinux > ../../System.map
make[1]: Leaving directory '/have-fun-debugging/sys2lab-21fall/lab5/arch/riscv'

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

```
System.map [Read-Only]
~/sys2lab-21fall/lab5

entry.S  System.map

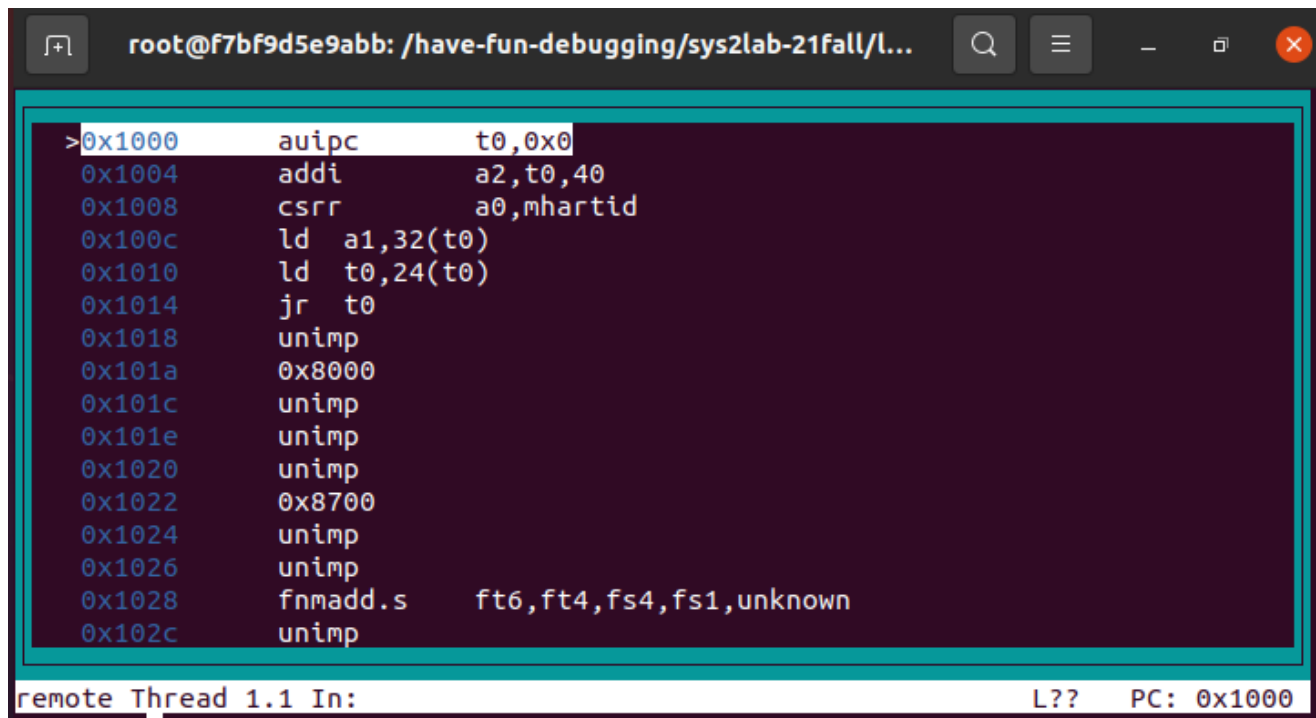
1 0000000008020000 A BASE_ADDR
2 0000000008020200 G TIMECLOCK
3 0000000008020400 B _ebss
4 0000000008020200 G _edata
5 0000000008020400 B _ekernel
6 00000000080201090 R _erodata
7 000000000802006d4 T _etext
8 0000000008020300 B _sbss
9 0000000008020200 G _sdata
10 0000000008020000 T _skernel
11 00000000080201000 R _srodata
12 0000000008020000 T _start
13 0000000008020000 T _stext
14 00000000080200060 T _traps
15 0000000008020300 B boot_stack
16 0000000008020400 B boot_stack_top
17 00000000080200190 T clock_set_next_event
18 00000000080200184 T get_cycles
19 00000000080200284 T printk
20 00000000080200260 T putc
21 000000000802001bc T sbi_ecall
22 0000000008020022c T start_kernel
23 0000000008020025c T test
24 000000000802001f4 T trap_handler
```



另一个窗口输入以下指令可远程连接。

```
1 riscv64-unknown-linux-gnu-gdb vmlinux
2 (gdb) target LAYOUT ASMremote :1234
```

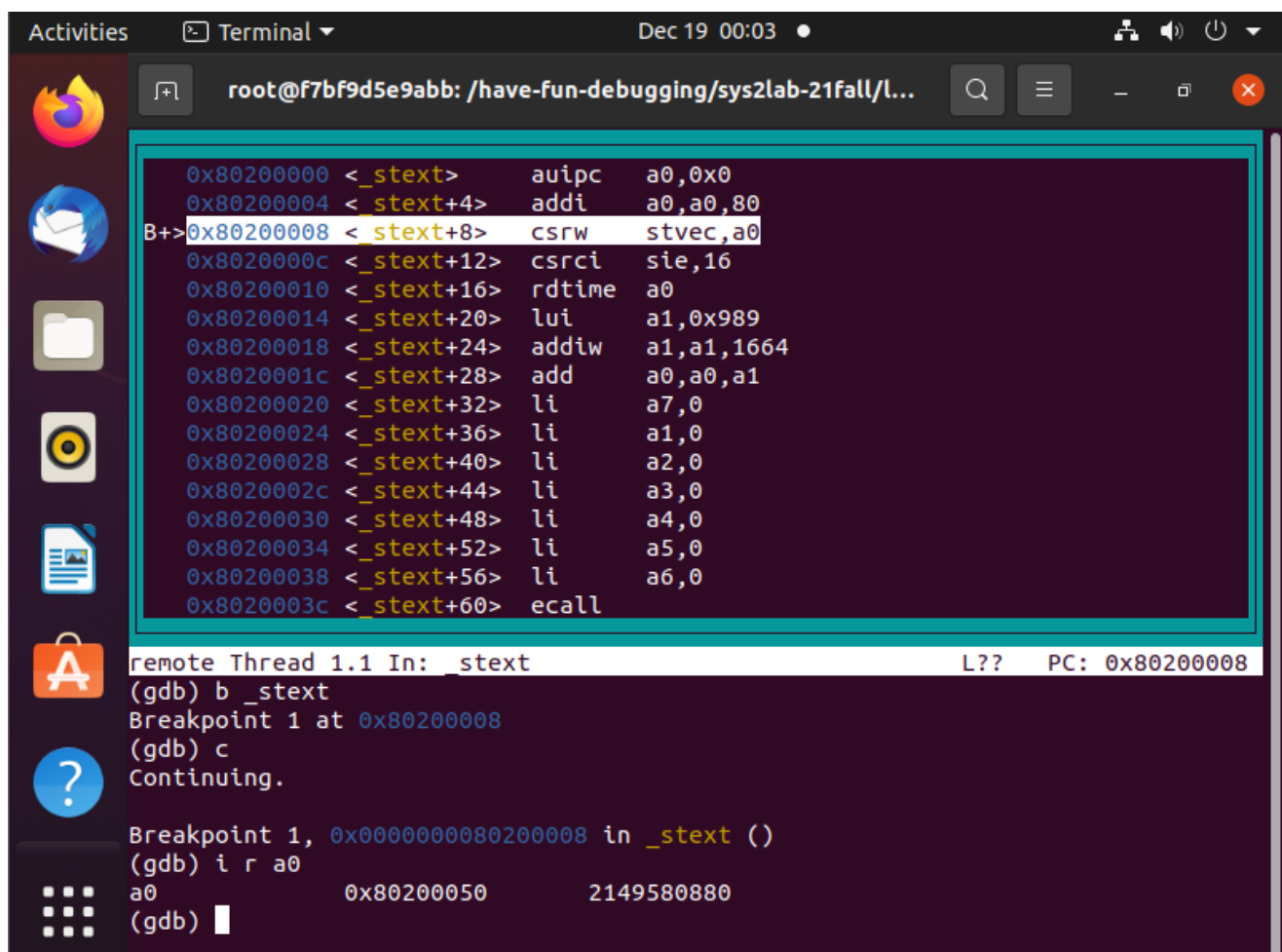
初始进入的函数地址。



```
>0x1000    auipc    t0,0x0
0x1004    addi     a2,t0,40
0x1008    csrr     a0,mhartid
0x100c    ld      a1,32(t0)
0x1010    ld      t0,24(t0)
0x1014    jr      t0
0x1018    unimp
0x101a    0x8000
0x101c    unimp
0x101e    unimp
0x1020    unimp
0x1022    0x8700
0x1024    unimp
0x1026    unimp
0x1028    fnmadd.s    ft6,ft4,fs4,fs1,unknown
0x102c    unimp

remote Thread 1.1 In: L?? PC: 0x1000
```

开始设置b \_stext, 进入到\_start。因为 \_start是全局的第一个程序, 写在 \_stext模块中, 故 \_stext和 \_start起始地址一样。



```
0x80200000 <_stext> auipc a0,0x0
0x80200004 <_stext+4> addi a0,a0,80
B+>0x80200008 <_stext+8> csrr stvec,a0
0x8020000c <_stext+12> csrrci sie,16
0x80200010 <_stext+16> rdtm a0
0x80200014 <_stext+20> lui a1,0x989
0x80200018 <_stext+24> addiw a1,a1,1664
0x8020001c <_stext+28> add a0,a0,a1
0x80200020 <_stext+32> li a7,0
0x80200024 <_stext+36> li a1,0
0x80200028 <_stext+40> li a2,0
0x8020002c <_stext+44> li a3,0
0x80200030 <_stext+48> li a4,0
0x80200034 <_stext+52> li a5,0
0x80200038 <_stext+56> li a6,0
0x8020003c <_stext+60> ecall

remote Thread 1.1 In: _stext L?? PC: 0x80200008
(gdb) b _stext
Breakpoint 1 at 0x80200008
(gdb) c
Continuing.

Breakpoint 1, 0x0000000080200008 in _stext ()
(gdb) i r a0
a0 0x80200050 2149580880
(gdb)
```



```
root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/l...
0x80200000 <_stext> auipc a0,0x0
0x80200004 <_stext+4> addi a0,a0,96
B+ 0x80200008 <_stext+8> csrw stvec,a0
0x8020000c <_stext+12> csrr a0,sie
0x80200010 <_stext+16> ori a0,a0,16
0x80200014 <_stext+20> csrw sie,a0
>0x80200018 <_stext+24> rdtime a0
0x8020001c <_stext+28> lui a1,0x989
0x80200020 <_stext+32> addiw a1,a1,1664
0x80200024 <_stext+36> add a0,a0,a1
0x80200028 <_stext+40> li a7,0
0x8020002c <_stext+44> li a1,0
0x80200030 <_stext+48> li a2,0
0x80200034 <_stext+52> li a3,0
0x80200038 <_stext+56> li a4,0
0x8020003c <_stext+60> li a5,0

remote Thread 1.1 In: _stext L?? PC: 0x80200018
0x0000000008020014 in _stext ()
(gdb) i r a0
a0 0x10 16
(gdb) si
0x0000000008020018 in _stext ()
(gdb) i r sie
sie 0x0 0
(gdb) i r a0
a0 0x10 16
(gdb)
```

上图中查看寄存器值发现赋值未成功，结果是赋值位数出错，从0开始第五位。改正后可看到。此时sie，sepc，sstatus都被成功赋值，进入printk函数。

```
0x0000000008020054 in _stext ()
(gdb) i r sstatus
sstatus 0x80000000000006002 -9223372036854751230
(gdb) si
0x0000000008020058 in _stext ()
```

在调试窗口中单步运行，发现在printk和sbi\_ecall函数中来回跳转，每次进入sbi\_ecall函数时，调用ecall指令。

S 模式不直接控制时钟中断和软件中断，而是使用ecall指令请求M模式设置定时器或代表它发送处理器间中断。所以在调用ecall指令后，系统便执行了

此时在调试窗口持续执行，在另一个窗口内可看到traps中不断输出字符。发现和样例有所不同，是因为字符串中没有换行符。



root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/l...



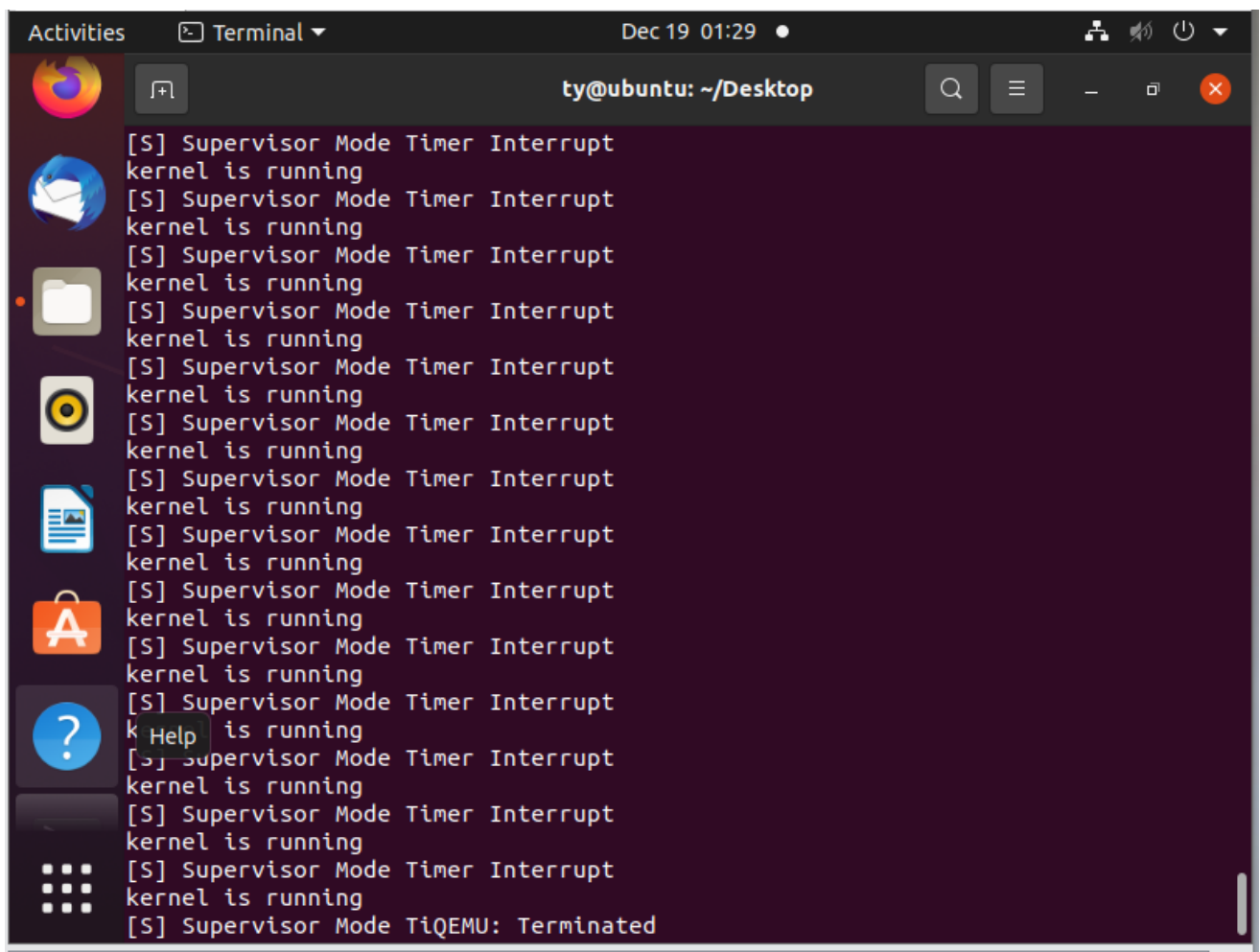
```
ervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor M
ode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer
Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interru
t[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supe
rvisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mo
de Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer
Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt
[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Super
visor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mod
e Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer I
nterrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[
S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Superv
isor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode
Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer In
terrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S
] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervi
sor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode
Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Int
errupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S]
Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervis
or Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode T
imer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Inte
rrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S]
Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Superviso
r Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Ti
mer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Inter
rupt[S] Supervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] S
upervisor Mode Timer Interrupt[S] Supervisor Mode Timer Interrupt[S] Supervisor
Mode Ti
```

```
root@f7bf9d5e9abb: /have-fun-debugging/sys2lab-21fall/l...
>0x800005ee ld t0,264(sp)
0x800005f0 csrw mstatus,t0
0x800005f4 ld t0,40(sp)
0x800005f6 ld sp,16(sp)
0x800005f8 mret
0x800005fc nop
0x80000600 add sp,a0,zero
0x80000604 ld ra,8(sp)
0x80000606 ld gp,24(sp)
0x80000608 ld tp,32(sp)
0x8000060a ld t1,48(sp)
0x8000060c ld t2,56(sp)
0x8000060e ld s0,64(sp)
0x80000610 ld s1,72(sp)
0x80000612 ld a0,80(sp)
0x80000614 ld a1,88(sp)

remote Thread 1.1 In: L?? PC: 0x800005ee
Terminal
Continuing.

Program received signal SIGINT, Interrupt.
0x000000000800005ee in ?? ()
(gdb) i r sepc
sepc 0x802002bc 2149581500
(gdb) c
Continuing.
```

更改换行符之后，可以看到：



与给出的样例相吻合。

## 思考题

1. 在我们使用make run时， OpenSBI 会产生如下输出：

```
1  OpenSBI v0.9
2
3  / _ \      / _ \      / _ \
4  | | | | _ _ | ( _ | | |
5  | | | | ' _ \ / _ \ ' _ \ < | |
6  | | | | |_) | _/ | |_) | | | _
7  \_/_/| ._/ \_/_/ | |_/ |_/ |
8
9  | |
10
11  .....
12
13  Boot HART MIDELEG      : 0x0000000000000222
14  Boot HART MEDELEG     : 0x000000000000b109
15
16  .....
```

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

hart 是硬件线程 (hardware thread)的缩略形式。

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。）

mideleg（Machine Interrupt Delegation，机器中断委托）CSR 控制将哪些中断（上图上部分）委托给 S 模式。与 mip 和 mie 一样，mideleg 中的每个位对应不同的异常。mideleg[5]对应于 S 模式的时钟中断，如果把它置位，S 模式的时钟中断将会移交 S 模式的异常处理程序，而不是 M 模式的异常处理程序。0x0000000000000222 的后十二位是 0010 0010 0010，第五位是 1，表明 S 模式的时钟中断会移交给 S 模式的异常处理程序，第 1，9 位是 1，表明相应中断都会移交给 S 模式。

M 模式还可以通过 medeleg CSR 将同步异常委托给 S 模式。该机制类似于刚才提到的中断委托，但 medeleg 中的位对应的不再是中断，而是图 10.3 中的同步异常编码（上图下部分）。置上 medeleg[15]便会把 store page fault（store 过程中出现的缺页）委托给 S 模式。0x000000000000b109 后 16 位是 1011 0001 0000 1001，第 0，3，8，12，13，15 位置 1，表明相应的异常会被移交给 S 模式来处理。