OS lab3

练习1：完善中断处理 （需要编程）

> 请编程完善trap.c中的中断处理函数trap，在对时钟中断进行处理的部分填写kern/trap/trap.c函数中处理时钟中断的部分，使操作系统每遇到100次时钟中断后，调用print_ticks子程序，向屏幕上打印一行文字”100 ticks”，在打印完10行后调用sbi.h中的shut_down()函数关机。
>
> 要求完成问题1提出的相关函数实现，提交改进后的源代码包（可以编译执行），并在实验报告中简要说明实现过程和定时器中断中断处理的流程。实现要求的部分代码后，运行整个系统，大约每1秒会输出一次”100 ticks”，输出10行。

要求代码实现

这里我们先根据项目注释的提示，补全要编写的所有代码：

代码块

```
#include <sbi.h>
......
#define TICK_NUM 100 //宏定义 100次 tick

// 用于输出`100ticks`的函数
static void print_ticks() {
    cprintf("%d ticks\n", TICK_NUM);
#ifdef DEBUG_GRADE
    cprintf("End of Test.\n");
    panic("EOT: kernel seems ok.");
#endif
}
......
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1;
    static int num = 0;  // 静态变量计数打印次数
    switch (cause) {
        ......
        case IRQ_S_TIMER:
            //(1)设置下次时钟中断
            clock_set_next_event();
            //(2)计数器（ticks）加一
            ticks++;
            //(3)当计数器加到100的时候，输出`100ticks`，同时打印次数（num）加一
            if (ticks % TICK_NUM == 0) {
                print_ticks();
```

```
27                  num++;
28                  //(4)判断打印次数，当打印次数为10时，调用<sbi.h>中的关机函数关机
29                  if (num == 10) {
30                      sbi_shutdown();
31                  }
32              }
33              break;
34          ......
35      }
36  }
```

interrupt_handler 捕获到 IRQ_S_TIMER 后处理时钟中断的实现过程如下：

1. interrupt_handler 首先立刻再执行 clock_set_next_event() 设置下次时钟中断，并令计数器（clock.c中的ticks）加一，随后陷入“中断——捕获——处理——中断”的循环；
2. 循环直到计数器加到100时，输出一句`100ticks`，同时打印次数（num）加一；继续循环；
3. 循环直到打印次数为10时，调用<sbi.h>中的关机函数 sbi_shutdown() 关机；循环结束，整个系统结束；

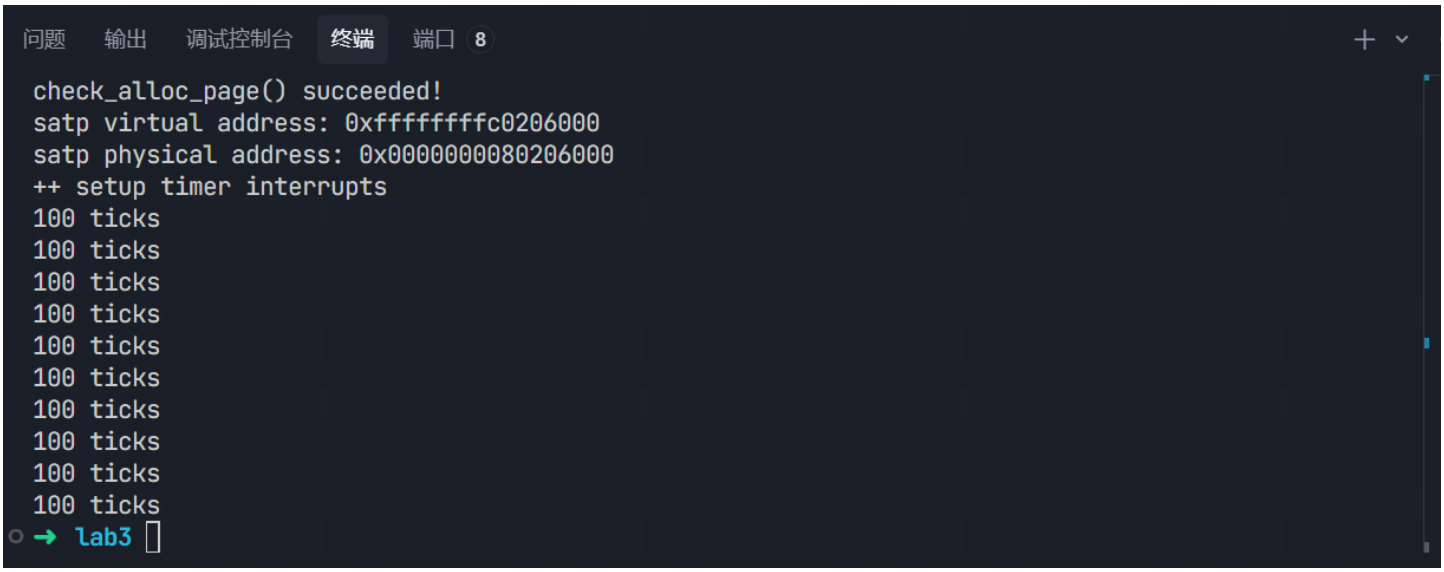
定时器中断处理流程

interrupt_handler() 捕获第一次 IRQ_S_TIMER 之前的流程分析如下：

1. 在 `kern_init ()` 中，程序首先执行 `idt_init()` 初始化中断描述符表IDT，让 CPU 以后一旦发生中断或异常，就跳转执行 `__alltraps`；（后续我们再对 `idt_init()` 详细分析）
2. 随后程序执行 `clock_init()`，该函数完成以下操作：
   - 在 sie 中设置 STIP 位，允许 S 态接收“定时器中断”；
   - 调用 clock_set_next_event() 安排第一次定时器事件；
   - 将全局计数 ticks 置零，并输出调试信息。
3. clock_set_next_event()只会做一件事：`sbi_set_timer(get_cycles() + timebase)`，即调用sbi_set_timer() 把下一次触发时间设为“当前时间 + 周期”；get_cycles读当前时钟，timebase 是中断间隔；到达该时间点会产生 IRQ_S_TIMER 时钟中断！
4. 自此，第一次 IRQ_S_TIMER 就设置完毕，它会立刻被 interrupt_handler() 捕获，从而让执行流陷入时钟中断处理的”100 ticks”打印循环中。

代码运行结果

运行整个系统后，大约每1秒会输出一次”100 ticks”，共输出10行，如下图所示，证明我们所编写的代码正确。



扩展练习 Challenge1：描述与理解中断流程

> 回答：描述ucore中处理中断异常的流程（从异常的产生开始），其中mov a0，sp的目的是什么？SAVE_ALL中寄寄存器保存在栈中的位置是什么确定的？对于任何中断，__alltraps 中都需要保存所有寄存器吗？请说明理由。

一、ucore中处理中断/异常的流程（从中断/异常的产生开始）

中断/异常发生时，**硬件**先在特权寄存器中留下现场：`sepc` 指向发生中断/异常的指令地址（中断则为吓下一条指令地址，异常则为当前指令地址，ecall特判），`scause` 标明类型并设置最高位表示“中断位”，`sstatus` 里 `SPP` 位记录之前的特权级，`sbadaddr` 保存与异常相关的地址。

> 注意：上述操作是硬件做的，这意味着在代码中无法体现

处理入口由 `stvec` 指定。在 `idt_init()` 中，内核将 `stvec` 设置为 `__alltraps`，并把 `sscratch` 置 0：

代码块

```
// kern/trap/trap.c
write_csr(sscratch, 0);
write_csr(stvec, &__alltraps);
```

宫老师上课讲到 `write_csr(sscratch, 0)` 表示中断后由S态接管，我进一步研究了该指令，总结来讲，这条指令就是**把 `sscratch` 清零，表示当前在内核态，不需要切换栈，用于区分用户态与内核态陷入。**

硬件跳转到 `__alltraps` 后，首先执行 `SAVE_ALL` 宏：

代码块

```
# kern/trap/trapentry.S
    .macro SAVE_ALL
```

```
3
4       csrw sscratch, sp
5
6       addi sp, sp, -36 * REGBYTES
7       # save x registers
8       STORE x0, 0*REGBYTES(sp)
9       STORE x0, 0*REGBYTES(sp)
10      STORE x1, 1*REGBYTES(sp)
11      # x2特殊处理
12      STORE x3, 3*REGBYTES(sp)
13      # ...
14
15      # get sr, epc, badvaddr, cause
16      # Set sscratch register to 0, so that if a recursive exception
17      # occurs, the exception vector knows it came from the kernel
18      csrrw s0, sscratch, x0
19      csrr s1, sstatus
20      csrr s2, sepc
21      csrr s3, sbadaddr
22      csrr s4, scause
23
24      STORE s0, 2*REGBYTES(sp)
25      # ...
26      STORE s4, 35*REGBYTES(sp)
27      .endm
```

- 先用 `csrw sscratch, sp` 暂存原始 `sp`，再将 `sp` 下移开辟 `trapframe` 空间（`addi sp, sp, -36 * REGBYTES`）。
- 按既定布局把通用寄存器（`x0..x31`，其中 `x2=sp` 特殊处理）写入当前栈帧；随后读取 `sstatus/sepc/sbadaddr/scause` 并也写入栈帧。

`move a0, sp` 将当前栈帧地址（即 `struct trapframe` 的起始地址）放到 `a0`，作为 C 函数的第一个参数传入，然后 `jal trap` 调用 `trap(struct trapframe *tf)`：

代码块

```
1   # kern/trap/trapentry.S
2   __alltraps:
3       SAVE_ALL
4
5       move  a0, sp
6       jal trap
```

在 C 层的 `trap()` 中，根据 `tf->cause` 的符号位判断中断还是异常：

代码块

```
// kern/trap/trap.c
static inline void trap_dispatch(struct trapframe *tf) {
    // 把 tf->cause 当作有符号数，检查最高位（硬件设为 1 表示“中断”）来区分中断与异常
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);  // 中断
    } else {
        // exceptions
        exception_handler(tf);  // 异常
    }
}
```

- 最高位为 1 的 `scause` 会被当作负数（`(intptr_t)tf->cause < 0`），走 `interrupt_handler()`；
- 否则走 `exception_handler()`。

不同的中断/异常对应不同的处理，在此不再赘述。执行完对应的处理后，`trap()` 返回，`__trapret` 执行 `RESTORE_ALL`：

代码块

```
    .macro RESTORE_ALL

    LOAD s1, 32*REGBYTES(sp)
    LOAD s2, 33*REGBYTES(sp)

    csrw sstatus, s1
    csrw sepc, s2

    # restore x registers
    LOAD x1, 1*REGBYTES(sp)
    # ...

    # restore sp last
    LOAD x2, 2*REGBYTES(sp)
    .endm
```

先恢复 `sstatus/sepc`，再按约定顺序恢复所有通用寄存器，最后用保存的原始 `sp` 复位栈指针。最后 `sret` 根据恢复后的 `sstatus` 与 `sepc`，回到被打断的指令之后或异常处理后的下一条指令，整个中断/异常处理流程就此结束。

需要注意的是，对于中断来说，这一切的一切能够顺利执行，都是因为在此之前已经执行了三个关键的置位操作。

以时钟中断为例：`clock_init()` 已经启用了 `sie` 的计时器中断位 `MIP_STIP`，`intr_enable()` 也开启了全局中断使能位 `SSTATUS_SIE`：

代码块

```
// kern/driver/clock.c
    set_csr(sie, MIP_STIP);  // SSTATUS_SIE

// kern/driver/intr.c
    set_csr(sstatus, SSTATUS_SIE);
```

除此之外，还有最后一位：当预约时间到来，硬件把 `sip` 的 `STIP` 置为挂起，指示时钟中断已经发生且尚未被处理。最后，我们形象总结三个寄存器的协作关系：

- sstatus.SIE = 1 → "我愿意接收中断"（全局的）
- sie.STIP = 1 → "我愿意接收定时器中断"（更细粒度的）
- sip.STIP = 1 → "定时器中断来了"

只有这样，才能顺利开始时钟中断的处理。

而对于异常来说，不需要这些复杂的操作，因为异常是"必须处理的错误"/用于调试，必须被捕获。

二、mov a0，sp的目的

在 RISC-V 调用约定里，`a0` 是 C 函数的第一个参数寄存器。此处 `sp` 指向我们刚刚在 `SAVE_ALL` 中构造出来的 `struct trapframe`，所以 `move a0, sp` 的意思就是 **把 trapframe 的指针传给 `trap(struct trapframe *tf)`**，这样 C 层可以通过 `tf` 访问并解析被保存的通用寄存器、`sstatus`、`sepc`、`badvaddr`、`cause` 等，从而做中断/异常分发与具体处理。

需要注意的是，要保证 `sp` 在 `jal trap` 前后保持一致（入口处有注释），便于 `__trapret` 用同一个 `sp` 定位同一份现场进行恢复。

三、SAVE_ALL中寄存器保存在栈中的位置是什么确定的

寄存器在栈中的布局必须严格与 `struct trapframe` 的内存结构对齐：

代码块

```
// kern/trap/trap.h
struct trapframe {
    struct pushregs gpr;
```

```
    uintptr_t status;
    uintptr_t epc;
    uintptr_t badvaddr;
    uintptr_t cause;
};
```

- 前面是 `struct pushregs gpr`，按“zero, ra, sp, gp, tp, t0..t2, s0..s1, a0..a7, s2..s11, t3..t6”这样的字段顺序排布；
- 后面依次是 `status`、`epc`、`badvaddr`、`cause`。

`SAVE_ALL` 通过固定偏移 `N * REGBYTES` 实现这个排布，即 `xi` 存在 `i * REGBYTES`，`status`、`epc`、`badvaddr`、`cause` 分别位于 `32..35`。

需要注意的是，`x2(sp)` 比较特殊，先不直接写入，而是用 `sscratch` 暂存，将其他通用寄存器全部保存后，经 `csrrw s0, sscratch, x0` 读回到 `s0`，再写入 `2 * REGBYTES`：

代码块

```
    csrw sscratch, sp
    addi sp, sp, -36 * REGBYTES
    # save x registers
    STORE x0, 0*REGBYTES(sp)
    # ...
    csrrw s0, sscratch, x0
    # ...
    STORE s0, 2*REGBYTES(sp)
```

这样做有两大原因：

- 首先，如果直接保存x2，保存的是新sp，而不是我们想要的旧sp。又考虑到语法问题（CSR寄存器不能直接作为普通指令的操作数），所以必须使用现有代码“倒来倒去”的方式保存旧sp。
- 其次，想实现保存旧sp，其实没必要 `csrrw s0, sscratch, x0`，直接 `csrr s0, sscratch` 即可。那么x0的作用是什么？如注释所言，为了进行**递归异常检测**。每次保存上下文前先检测 `sscratch` 的值是否为0，若为0则发生了内核态->内核态的递归异常，进而采取一些特殊措施。

总的来说，这种映射让 `trap.c` 在 C 层用 `struct trapframe` 的自然字段顺序就能读写对应值，避免了手工拆装寄存器的复杂，同时也让 `RESTORE_ALL` 可以按同样的偏移恢复。

> 补充：
>
> - 虽然 `x0` 永远为 0，从“功能”上讲保存它并非必需，但为了让 `struct pushregs` 的字段与偏移保持完全一致，还是按统一规则保存了。
> - `REGBYTES` 在 `libs/riscv.h` 中按位宽定义为 4 或 8。

四、__alltraps 是否需要对任何中断都保存所有寄存器

从安全与简单性的角度，保存所有寄存器是合理的，也是本次实验的设计选择。进一步分析，中断/异常可能打断任意指令序列，后续调用的 C 函数可能根据编译器分配使用任意寄存器。为了保证返回后原上下文不受破坏，最省心的办法就是把通用寄存器和关键特权寄存器一次性保存完整。

当然，**为节省开销**，我们可以：

- 根据调用约定只保存“会被破坏”的寄存器：中断发生时，我们相当于“调用者”，即将调用中断处理函数。因此，我们有必要保存Caller-saved的那些寄存器的上下文，它们可能被“被调用者”破坏。
- 针对特定中断采用更轻量的入口：比如发生系统调用时，需要保存上下文可能比时钟中断要多一些。

但是，这样做的缺点也可想而知：

- 不同编译器/优化级别下的寄存器分配存在差异，仅保存标准的Caller-saved寄存器可能不够。
- 需要为不同中断类型维护多套保存/恢复代码，增加出错概率。

综合考虑这些因素，如果不对任何情况都保存所有寄存器，是否真的能“节省开销”（无论是时间还是空间），都是有待商榷的。因此，本次实验的设计选择还是比较合理的，省心省力，效率还不低。

扩展练习 Challenge2：理解上下文切换机制

> 回答：在trapentry.S中汇编代码 csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么？save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢？

一、在trapentry.S中汇编代码 csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么？

代码块

```
1   csrw sscratch, sp
2   csrrw s0, sscratch, x0
```

这是 RISC-V 架构的控制与状态寄存器（CSR）访问指令，其中 `csrw` 是把一个通用寄存器的值写入 CSR，`csrrw` 是交换 CSR 与寄存器的值（Read and Write）。

1. `csrw sscratch, sp`

含义是把当前栈指针寄存器 sp 的值写入 CSR sscratch，也就是 `sscratch = sp;`

> 操作系统的约定是**如果 trap 发生时是 S 模式，则 sscratch 设为 0；如果 trap 发生时是 U 模式，则 sscratch 会存储该进程对应的内核栈地址（top）**。
>
> 所以在进入 trap 时，csrw sscratch, sp 就相当于告诉系统：“我现在要暂时接管 CPU，把原先的栈指针先存到 sscratch 里。”

一开始我以为如果是U特权的话这里是用来恢复状态的，但是我们这里模仿的是S，所以不需要。但其实不完全是这样的！还有一个微妙的补充点：即使我们现在不切栈，它依然能保护原始的 `sp` 栈指针，防止它被错误覆盖。**这里除了能“恢复 U 态栈”，更是能“安全保管原始 sp”，也是为了统一处理逻辑。**

2. `csrrw s0, sscratch, x0`

含义是把 CSR `sscratch` 的值读出到寄存器 `s0`，同时把 `x0`（恒为 0）写入 `sscratch`。把刚才存在 `sscratch` 里的值取出来放到 `s0`，这样 `s0` 暂时保存了中断发生时的旧栈指针；除此之外还把 0 写入 `sscratch`，这样如果再次发生异常（递归异常）时，异常就能知道它来自内核，避免错误地切换栈。

代码块

```
1       # Set sscratch register to 0, so that if a recursive exception
2       # occurs, the exception vector knows it came from the kernel
```

二、save all里面保存了stval scause这些csr，而在restore all里面却不还原它们？那这样store的意义何在呢？

trapentry.S 的大致逻辑如下：

代码块

```
1   __alltraps:
2       SAVE_ALL       # 保存现场
3       call trap      # 调用 C 层函数 trap(tf)
4       RESTORE_ALL    # 恢复现场
5       sret
```

- `SAVE_ALL` 是把所有寄存器（x0–x31）+ CSR 信息（`sepc`, `sstatus`, `scause`, `stval` 等）保存到 `trapframe`；
- `RESTORE_ALL` 是在返回前恢复通用寄存器（x0–x31）和部分 CSR（如 `sepc`, `sstatus`），让 `sret` 能正确回到原执行现场。

其中没有restore这些 CSR 描述了异常/中断的原因如下，因为不是程序状态所以不需要恢复

寄存器	含义
scause	Trap 的原因（中断/异常类型）
stval	Trap 相关的地址或值 （例如访问出错的地址、非法指令内容）
sepc	触发 Trap 时的程序计数器（PC）

当 trap 发生时：

- 硬件自动把异常原因写入 `scause`；
- 若是地址或指令错误，会把相关数值写入 `stval`；
- 再跳到 `stvec`（即 `__alltraps`）。

这些值只在异常处理过程中有意义，内核通过读取 `trapframe` 中的 `scause` / `stval` 来判断发生了什么类型的异常或中断。当中断处理结束后，**下一步程序要返回的地方是由 `sepc` 决定的，而不是 `scause` / `stval`。所以这些寄存器只要被读出来供 C 层使用即可，不需要恢复。**

在 trap.c 中：

代码块

```
1  void trap(struct trapframe *tf) {
2      trap_dispatch(tf);
3  }
```

trapframe 结构体里有：

代码块

```
1  struct trapframe {
2      ...uintptr_t stval;uintptr_t scause;uintptr_t sepc;
3  };
```

中断分发逻辑依赖这些值，例如 `if (tf->cause == IRQ_S_TIMER) ...`，**所以汇编保存它们的意义是传递中断类型和数据信息给trap。**

扩展练习 Challenge3：完善异常中断

> 编程完善在触发一条非法指令异常 mret和，在 kern/trap/trap.c的异常处理函数中捕获，并对其进行处理，简单输出异常类型和异常指令触发地址，即“Illegal instruction caught at 0x(地址)”，“ebreak caught at 0x（地址）”与“Exception type:Illegal instruction"，“Exception type: breakpoint”。（

Exception 处理代码

对于**非法指令异常和断点指令异常，**都采取**“越过”**处理：

1. 输出指令异常类型；

2. epc 寄存器的值即为异常发生的指令地址 ，获取后格式化输出，%x表示按十六进制格式输出，08表示输出宽度至少为 8 个字符，不足补零；
3. 读取该异常指令地址处的16位值并依据RISC-V编码约定判断指令长度：
   - 若低两位不为 0x3 ，说明是C扩展的16位压缩指令，则将 tf->epc 移动 2 字节到下一条指令；
   - 否则为标准32位指令，将 tf->epc 移动 4 字节到下一条指令；

   这样返回异常时会跳过触发异常的那条指令，避免再次陷入同一异常循环。

代码块

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        ......
        case CAUSE_ILLEGAL_INSTRUCTION:
             // 非法指令异常处理
            // (1)输出指令异常类型
            cprintf("Exception type:Illegal instruction\n");
            // (2)输出异常指令地址
            cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
            // (3)更新 tf->epc寄存器
            {
                // 从异常指令地址 tf->epc 所指的内存中，读取16位值，并存入变量。
                uint16_t insn16 = *(uint16_t *)(tf->epc);
                if ((insn16 & 0x3) != 0x3) {
                    tf->epc += 2;  // 压缩指令长度16位
                } else {
                    tf->epc += 4;  // 标准指令长度32位
                }
            }
            break;
        case CAUSE_BREAKPOINT:
            //断点异常处理
            // (1)输出指令异常类型
            cprintf("Exception type:breakpoint\n");
            // (2)输出异常指令地址
            cprintf("ebreak caught at 0x%08x\n", tf->epc);
            // (3)更新 tf->epc寄存器
            {
                // 从异常指令地址 tf->epc 所指的内存中，读取16位值，并存入变量。
                uint16_t insn16 = *(uint16_t *)(tf->epc);
                if ((insn16 & 0x3) != 0x3) {
                    tf->epc += 2;  // 压缩指令长度16位
                } else {
                    tf->epc += 4;  // 标准指令长度32位
                }
            }
```

```
            break;
        ......
    }
}
```

Exception 测试代码

我们使用内联汇编 `asm volatile (...)` 的方式，来测试编写的Exception处理是否正确。其中 `volatile` 表示禁止优化或删除这条汇编。

对于非法指令异常，我们让汇编器**直接在机器码中放入一个 32 位的 `0xFFFFFFFF` 和 16位的 `0x0000`**，这是两个常数值，其作为指令编码来说肯定是不合法的；

对于断点指令异常，我们就直接插入 **`ebreak` 标准断点指令**进行测试；

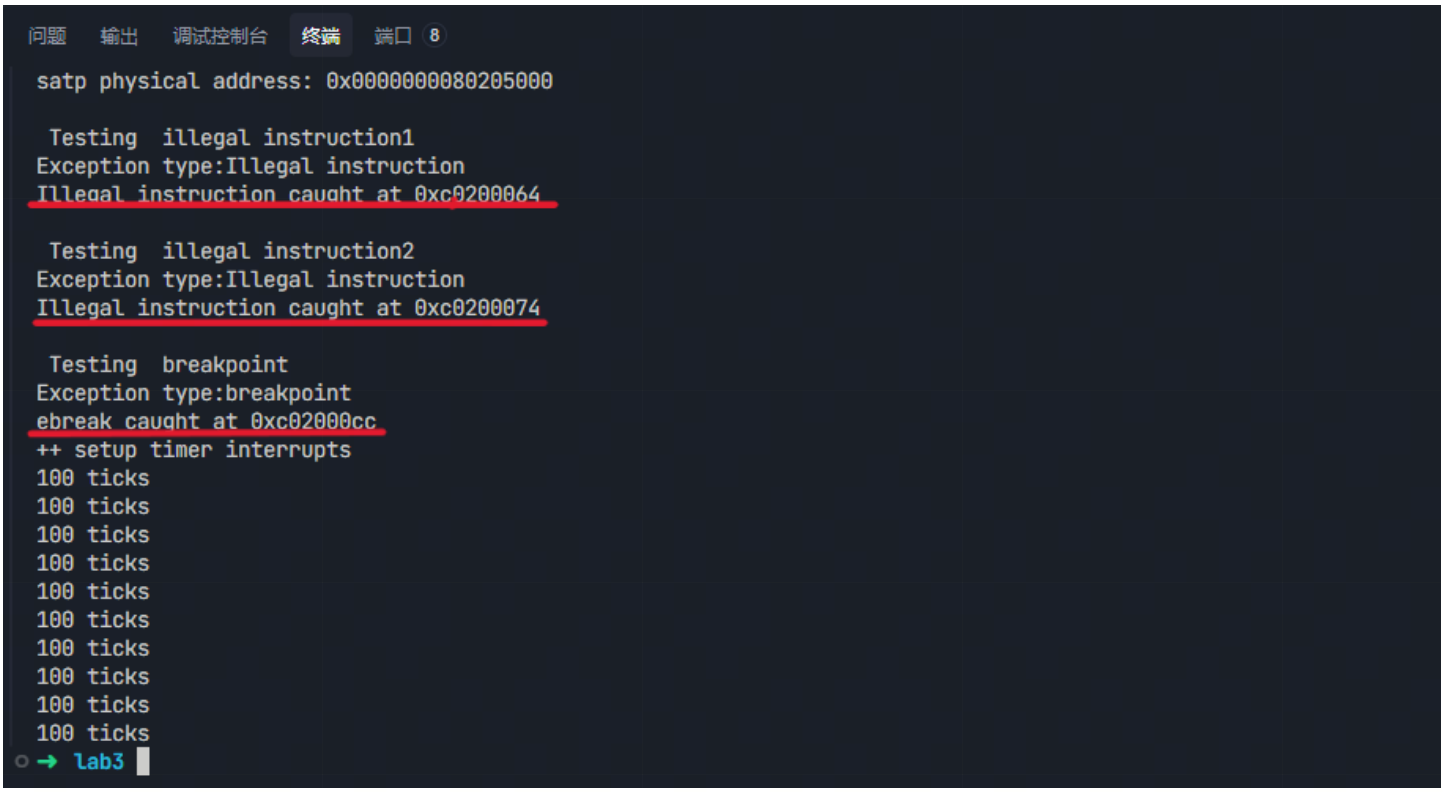
相关代码如下：

代码块

```
void test_illegal(void) {
    cprintf("\n Testing  illegal instruction1 \n");
    asm volatile (".word 0xffffffff");     // 非法 32 位编码，低两位 == 0x3

    cprintf("\n Testing  illegal instruction2 \n");
    asm volatile (".short 0x0000");        // 非法 16 位半字，低两位 != 0x3
}

void test_breakpoint(void) {
    cprintf("\n Testing  breakpoint \n");
    asm volatile ("ebreak");               // 标准断点
}

int kern_init(void) {
    ......
    // 异常测试
    test_illegal();
    test_breakpoint();
    ......
}
```

Exception 处理测试

执行 `make qemu` ，结果如下：可以看到所有异常全部被捕获且越过执行，说明我们编写的代码正确！



重要知识点总结

一、实验中的核心知识点及其与OS原理的对应关系

> - 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

本实验的重要知识点主要包括中断处理机制、时钟中断的捕获与响应、trap函数的实现流程、__alltraps的现场保存与恢复等。这些内容在理论课中分别对应于中断、异常与系统调用的响应机制，以及内核态与用户态的权限切换。

在OS课程中，**中断**是一种**异步事件**，由外部设备（如时钟）触发，用于打断当前执行流并转向预设处理函数，以实现多任务调度和时间片管理；而在本实验中，interrupt_handler函数捕获IRQ_S_TIMER时钟中断，通过clock_set_next_event()设置下次中断并递增ticks计数器，对应理论课中的中断响应与持续处理机制，通过代码模拟了理论课中的硬件-软件协作流程；**异常**是**同步事件**，由指令执行错误引发，理论课中强调杀死或重执行进程以维护系统稳定性，实验中的trap_dispatch处理CAUSE_USER_ECALL等异常事件与之对应，但实验未涉及复杂异常恢复，仅聚焦简单捕获；**系统调用**是用户主动请求内核服务的机制，理论课中通过中断向量表和权限提升（ring3到ring0）实现，实验中的__alltraps宏（SAVE_ALL）保存寄存器现场并切换栈，对应理论课中的中断/异常/系统调用统一入口，二者差异在于理论课强调权限环和中断向量表的抽象设计，而实验通过sscratch寄存器区分用户/内核态，实现了RISC-V特有的现场保护；**此外**，实验中的print_ticks和sbi_shutdown对应理论课中中断后的用户可见输出与系统退出，但实验简化了多进程环境，未涉及理论课中的进程隔离（如页表权限位）。

二、操作系统原理中重要但实验未涉及的知识点

- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

1. **进程间通信（IPC）机制**

如共享内存、管道、消息队列等，用于进程间数据传递。理论课中强调安全隔离（如借助页表实现进程地址空间独立）和权限管理，以防止跨进程访问错误，但实验仅处理中断，未涉及多进程协作。

2. **进程隔离与保护环的完整应用**

理论课中通过硬件权限等级（ring0到ring3）和段/页表权限位实现内核与用户空间隔离，以及进程间内存访问限制（如虚拟地址空间分布），实验虽提及权限切换，但未实现多进程页表管理和内存异常处理。

3. **中断、异常与系统调用的比较分析**

理论课中详细区分了三者的源头（中断异步外部、异常同步内部、系统调用主动请求）、响应方式和处理机制（如中断透明持续），实验中统一用trap处理，未深入区分三者的区别。