Lab3 实验报告：中断与中断处理流程

2313310 熊诚义 2311887 陈语童 2310364 柳昕彤

练习1：完善中断处理（时钟中断）

1.1 实验要求

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 `sbi_shutdown()` 函数关机。

1.2 实现过程

1.2.1 添加必要的头文件

首先需要在 `trap.c` 文件开头添加 `<sbi.h>` 头文件，以便使用 `sbi_shutdown()` 函数：

```
#include <sbi.h>
```

1.2.2 实现时钟中断处理代码

在 `interrupt_handler()` 函数的 `IRQ_S_TIMER` 分支中添加以下代码：

```
case IRQ_S_TIMER:
    // 设置下次时钟中断
    clock_set_next_event();
    // 计数器加一
    ticks++;
    // 每100次时钟中断打印一次
    if (ticks % TICK_NUM == 0) {
        print_ticks();
        // 记录打印次数
        static int num = 0;
        num++;
        // 打印10次后关机
        if (num == 10) {
            sbi_shutdown();
        }
    }
    break;
```

1.2.3 代码说明

1. **设置下次时钟中断**：`clock_set_next_event()`
   - 调用此函数设置下一次时钟中断的触发时间
   - 必须在处理当前中断时立即设置，否则会丢失后续的时钟中断
2. **计数器加一**：`ticks++`
   - `ticks` 是全局变量，定义在 `clock.c` 中
   - 每次时钟中断发生时递增，记录中断总次数

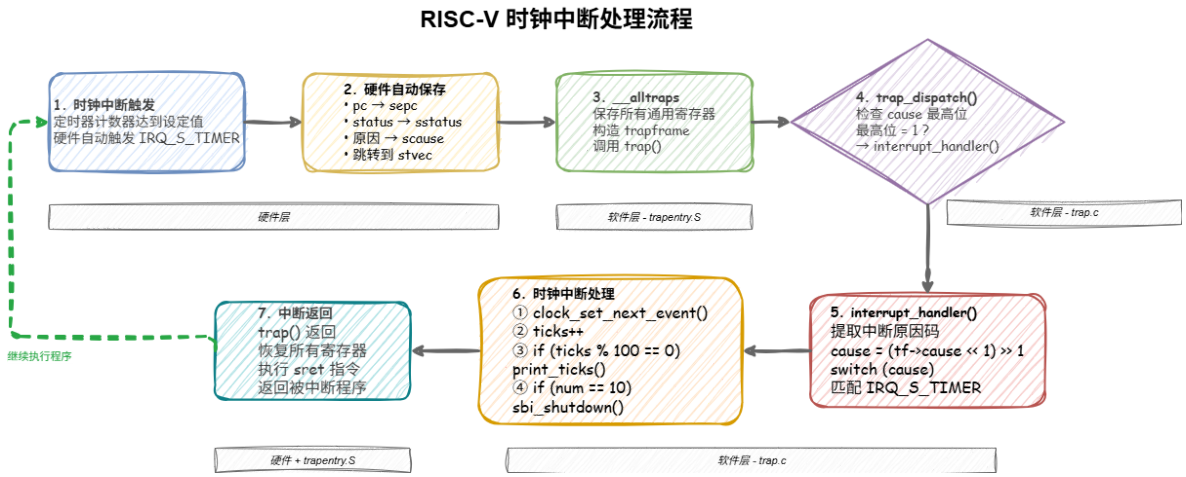
3. **每100次打印一次**：`if (ticks % TICK_NUM == 0)`
   - `TICK_NUM` 定义为 100
   - 当 `ticks` 能被 100 整除时，调用 `print_ticks()` 打印信息
4. **记录打印次数**：`static int num = 0`
   - 使用静态变量 `num` 记录已打印的次数
   - 静态变量在函数调用之间保持其值
5. **打印10次后关机**：`if (num == 10)`
   - 当打印次数达到 10 次时，调用 `sbi_shutdown()` 关闭系统

1.3 定时器中断处理流程

1.3.1 中断初始化流程

```
kern_init() [kern/init/init.c]
    ↓
idt_init() [kern/trap/trap.c]
    ↓
write_csr(sscratch, 0)           // 设置 sscratch 寄存器为 0
    ↓
write_csr(stvec, &__alltraps)    // 设置中断向量表入口地址
    ↓
clock_init() [kern/driver/clock.c]
    ↓
set_csr(sie, MIP_STIP)           // 使能时钟中断
    ↓
clock_set_next_event()           // 设置第一次时钟中断
    ↓
ticks = 0                        // 初始化计数器
    ↓
intr_enable()                    // 全局使能中断 (设置 sstatus.SIE)
```

1.3.2 中断处理流程



1.3.3 关键寄存器说明

寄存器	说明
`stvec`	S 模式中断向量表基址寄存器，指向 `__alltraps`
`sepc`	S 模式异常程序计数器，保存被中断指令的地址
`scause`	S 模式异常原因寄存器，记录中断/异常类型
`sstatus`	S 模式状态寄存器，包含 SIE (中断使能位) 等
`sscratch`	S 模式临时寄存器，用于保存内核栈指针
`sie`	S 模式中断使能寄存器，控制各类中断的使能

1.4 运行结果

编译运行后的输出：

```
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0x80200000
  etext  0x80201a34
  edata  0x80206020
  end    0x80206028
Kernel executable memory footprint: 25KB
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

运行特点：

- 系统正常启动并初始化
- 约每 1 秒输出一次 "100 ticks"
- 连续输出 10 行后自动调用 `sbi_shutdown()` 关机

1.5 实验总结

通过本练习，我们掌握了：

1. **中断处理的完整流程**：从中断触发到中断返回的每一个步骤
2. **CSR 寄存器的使用**：理解了 RISC-V 特权架构中的关键控制和状态寄存器
3. **时钟中断的实现机制**：如何通过 SBI 接口设置定时器并处理时钟中断
4. **中断嵌套的预防**：在处理当前中断时需要立即设置下次中断时间

Challenge 1: 描述与理解中断流程

1. 描述ucore中处理中断异常的流程（从异常的产生开始）

(1) 当前 `PC` 保存至 `sepc` 寄存器；中断/异常类型写入 `scause` 寄存器；如果异常与缺页或访问错误相关，将相关的地址或数据（辅助信息）保存到 `stval` 寄存器，以便中断处理程序在后续处理中使用。
(2) 将当前的中断使能状态 `sstatus.SIE` 保存到 `sstatus.SPIE` 中，`sstatus.SIE` 清零，禁用 S 模式中断，保证在处理中断时不会被其他中断打断。将当前特权级（即 U 模式，值为 0）保存到 `sstatus.SPP` 中；如果当前在 U 权限级，先切换到 S 权限级。
(3) `PC` 设置为 `stvec` 寄存器的值，跳转到中断处理程序的入口，即 `trapentry.S` 的 `__alltraps` 标签。
(4) 跳转后，执行保存所有通用寄存器、CSR 等的宏（`SAVE_ALL`），当前上下文，即寄存器状态存在栈上。
(5) 把当前栈指针 `sp` 作为一个**结构体参数**，调用/跳转到内核C函数 `trap()`（中断处理函数，通过 `jal trap` 跳转），这样 `trap()` 函数可以访问被保存的寄存器状态、CSR 状态。跳转后，就切换到了 `trap()` 的上下文，进入 `trap()` 的执行流。
(6) `trap.c` 判断是中断/异常，调用 `trap_dispatch()` 和 `interrupt_handler()` 按类型分发，分类处理。中断处理包括设置下一次时钟中断、更新内核全局时间计数器 `ticks`，周期性打印与关机。异常处理包括打印提示异常、如果可以修复的话进行修复。
(7) 处理完毕后返回，执行 `__trapret` 段，调用 `RESTORE_ALL` 宏从栈中恢复寄存器、CSR状态，如果是 S 权限级的中断/异常，通过 `sret` 指令回到用户态。

2. `mov a0，sp` 的目的是什么?

字面意义上，这句指令将栈指针 `sp` 的值拷贝到寄存器 `a0` 中。而往下看一行，发现指令 `jal trap`，那么 `a0` 寄存器在这里的角色是作为 `trap()` 的传入参数的。而向上溯源，`sp` 此时指向的是**“trapframe” 结构体**的起始地址，即刚刚在栈上保存的中断现场的一系列信息（寄存器/CSR状态）。因此，从原理上解释，这句指令的目的是：**将中断时保存的寄存器/CSR状态作为结构体传参给C函数 `trap()`**，使得 `trap.c`（`trap()` 所在）能够读取状态信息进一步处理中断/异常。

3. `SAVE_ALL` 中寄寄存器保存在栈中的位置是什么确定的?

`SAVE_ALL` 宏的定义在 `trapentry.S` 顶端：

```
    .macro SAVE_ALL

    csrw sscratch, sp

    addi sp, sp, -36 * REGBYTES
    # save x registers
    STORE x0, 0*REGBYTES(sp)
    STORE x1, 1*REGBYTES(sp)
    ... ... ... ...
    STORE x31, 31*REGBYTES(sp)

    # get sr, epc, badvaddr, cause
    # Set sscratch register to 0, so that if a recursive exception
    # occurs, the exception vector knows it came from the kernel
    csrrw s0, sscratch, x0
    csrr s1, sstatus
    csrr s2, sepc
    csrr s3, sbadaddr
    csrr s4, scause

    STORE s0, 2*REGBYTES(sp)
    ... ... ... ...
    STORE s4, 35*REGBYTES(sp)
    .endm
```

`SAVE_ALL` 相当于在对 CPU 当前所有通用寄存器与关键 CSR 寄存器进行编号，形成一个 “编号 *REGBYTES(sp)” 的偏移格式（注：REGBYTES = 8，从原sp开始向低地址偏移/增长），从而能够指导它们以如何的顺序/地址将值存到栈上。

`SAVE_ALL` 的模式本身是人为定义的，但是需要与 **trapframe 结构** 严格对应：

```
struct trapframe {
    struct pushregs gpr;      // 通用寄存器上下文
    uintptr_t status;         // sstatus 寄存器的值
    uintptr_t epc;            // sepc (异常发生时的 PC)
    uintptr_t badvaddr;       // stval (出错地址)
    uintptr_t cause;          // scause (trap 原因/类型)
};
```

4. 对于任何中断，__alltraps 中都需要保存所有寄存器吗?

是。`__alltraps` 中通常都保存**所有**通用寄存器和关键 CSR，无论是哪种中断或异常。目的主要是为了保证中断/异常处理的**通用性和安全性**。

代码原理性的支撑上，`SAVE_ALL` 的模式是规定死的，都需要保存；在进入 `trap.c` 进行处理之前也没有任何逻辑用于判断中断类型，也自然没有条件分支来决定哪些寄存器要保存，哪些不要。

Challenge 2: 理解上下文切换机制

1. `csrw sscratch, sp` 与 `csrrw s0, sscratch, x0` 实现了什么?

字面意义上完成的工作：把当前栈指针 `sp` 的值写入 `sscratch` CSR；读出 `sscratch` 的值到 `s0` 寄存器，写入 `x0` 寄存器值到 `sscratch` 中（`x0` 恒为0，所以就是清零了 `sscratch`）。

两句指令综合来看实现了将当前栈指针和其他所有寄存器、CSR一起压入栈，保存为中断时的现场状态成为 trapframe 结构体的一部分。

将 `sp` 压入栈一并处理的目的，一方面其他压栈操作时逻辑上 `sp` 的值在不断变化，通过上面这样的过程，可以通过 `sscratch` 的暂时转存准确无误地将原 `sp` 保存下来；另一方面，保存正确的 `sp`，能够在当出现递归异常/中断（即处理中断/异常的过程中又出现了中断/异常）的情况下，能够保证新的中断信息不覆盖上一层信息，能够通过保存的 `sp` 正确恢复上一层中断状态。

2. 保存了 `stval` `scause` 这些 CSR，却不还原它们？这样 store 的意义何在?

`stval` 保存访问错误或缺页的地址，`scause` 保存异常/中断类型，它们只在 `trap.c` 中进行中断处理的逻辑中有参考价值，并不代表原先程序的运行状态。相反地，`sepc` 和 `sstatus` 分别包含原 `PC`、中断使能信息和特权级，它们代表程序中断时的状态快照，或者能够影响全局运行状态。那这两个 CSR 在 `RESTORE_ALL` 中就出现了。

所以提问中涉及的两个 CSR，它们只是单纯提供中断/异常处理的参考信息，写回不会对程序接下去运行有任何帮助，那么就没有必要写回。但它们的 store 本身确实为 `trap.c` 的处理提供必要信息。

Challenge3：完善异常中断

1 实验要求

编程完善在触发一条非法指令异常和断点异常时，在 `kern/trap/trap.c` 的异常处理函数中捕获，并对其进行处理，输出异常类型和异常指令触发地址。

2 RISC-V 异常分类

我们查阅了资料，根据 RISC-V 特权级规范，异常类型通过 `scause` 寄存器标识：

异常代码	异常名称	说明
0	Instruction address misaligned	指令地址不对齐
1	Instruction access fault	指令访问错误
2	Illegal instruction	非法指令
3	Breakpoint	断点
4	Load address misaligned	加载地址不对齐
5	Load access fault	加载访问错误
6	Store/AMO address misaligned	存储地址不对齐
7	Store/AMO access fault	存储访问错误
8	Environment call from U-mode	用户态系统调用
9	Environment call from S-mode	内核态系统调用
12	Instruction page fault	指令页面错误
13	Load page fault	加载页面错误
15	Store/AMO page fault	存储页面错误

如果`scause` 最高位为 1，说明是中断（Interrupt），反之最高位为 0代表异常（Exception）。

2.1 非法指令异常（CAUSE_ILLEGAL_INSTRUCTION = 0x2）

非法指令异常的触发条件有三种：执行未定义的指令、在当前特权级不允许执行的指令（如在 S 模式执行 M 模式指令）以及访问当前特权级无权访问的 CSR 寄存器。比如`mret`在M 模式返回指令，在 S 模式下是非法的。

2.2 断点异常（CAUSE_BREAKPOINT = 0x3）

断点异常的触发条件是执行 `ebreak` 指令，主要发生在我们调试器设置断点或者用户程序主动请求进入调试模式，操作系统也可以利用断点实现特殊功能。

3 代码实现

3.1 异常处理代码

我们在 `kern/trap/trap.c` 的 `exception_handler()` 函数中添加：

```
case CAUSE_ILLEGAL_INSTRUCTION:
    // (1) 输出异常类型
    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epc寄存器，跳过非法指令（RISC-V指令长度为4字节）
    tf->epc += 4;
    break;

case CAUSE_BREAKPOINT:
    // (1) 输出异常类型
    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epc寄存器，跳过断点指令（RISC-V指令长度为4字节）
    tf->epc += 4;
    break;
```

为什么要更新 epc 寄存器（tf->epc += 4）？因为epc保存的是发生异常的指令地址，如果我们不修改epc，中断返回后会**重新执行相同的指令**，但对于非法指令和断点，我们希望跳过这条指令，继续执行后面的代码。RISC-V 标准指令长度为 4 字节，所以 `epc += 4` 指向下一条指令

执行流程：

```
异常发生 -> epc = 异常指令地址
    ↓
处理异常
    ↓
epc += 4 (跳过异常指令)
    ↓
sret 返回 -> PC = epc
    ↓
继续执行下一条指令
```

3.2 测试代码

在 `kern/init/init.c` 的 `kern_init()` 函数中添加测试代码：

```
// 测试断点异常 (ebreak)
cprintf("Testing breakpoint exception...\n");
asm volatile("ebreak");
cprintf("After ebreak\n");

// 测试非法指令异常，使用mret指令
cprintf("Testing illegal instruction exception...\n");
asm volatile(".word 0x30200073");  // mret指令的机器码
cprintf("After illegal instruction\n");
```

`mret` 是 Machine-mode Return 指令,只能在 M 模式执行,在 S 模式执行会触发非法指令异常

触发非法指令异常，我们有三种写法：

1.使用内联汇编插入非法指令

```
asm volatile("mret");  // M模式返回指令，在S模式非法
```

2.直接写入指令机器码

```
asm volatile(".word 0x30200073");  // mret的机器码
```

3.执行未定义的指令

```
asm volatile(".word 0x00000000");  // 未定义指令
```

4 异常触发时机

我们需要注意的是，异常只有在**被触发并且中断被使能**时才会被处理。

```
idt_init();      // 设置中断向量表
intr_enable();  // 使能中断

// 此时异常才能被正确处理
asm volatile("ebreak");
```

`idt_init()` 设置 `stvec` 寄存器，指向 `__alltraps`，`intr_enable()` 设置 `sstatus.SIE` 位，使能中断。接着我们才执行异常指令，之后CPU 硬件自动跳转到 `stvec` 指向的处理程序。

```
执行异常指令 (ebreak / mret)
    ↓
硬件检测到异常
    ↓
设置 scause = 异常类型 (2 或 3)
设置 sepc = 异常指令地址
设置 sstatus.SPP = 当前特权级
设置 sstatus.SPIE = 当前中断使能
清除 sstatus.SIE (禁用中断)
    ↓
跳转到 stvec (即 __alltraps)
    ↓
SAVE_ALL: 保存所有寄存器到 trapframe
    ↓
调用 trap(trapframe) -> trap_dispatch() -> exception_handler()
    ↓
根据 tf->cause 选择处理分支
    ↓
打印异常信息
修改 tf->epc += 4 (跳过异常指令)
    ↓
RESTORE_ALL: 恢复所有寄存器（包括修改后的 epc）
    ↓
sret: 返回到 tf->epc 指向的地址
    ↓
继续执行异常指令的下一条指令
```

5测试错误：压缩指令导致的死循环

在实际测试中，我们发现了一个严重的问题：**程序陷入死循环**，不断重复输出：

```
Testing breakpoint exception...
Exception type: breakpoint
ebreak caught at 0xc02000a8
Exception type: Illegal instruction
Illegal instruction caught at 0xc02000ac
Testing breakpoint exception...
Exception type: breakpoint
ebreak caught at 0xc02000a8
Exception type: Illegal instruction
Illegal instruction caught at 0xc02000ac
...（无限循环）
```

`ebreak` 地址：`0xc02000a8`，非法指令地址：`0xc02000ac`，两个地址相差 **4 字节**。但是 `"After ebreak"` 和 `"After illegal instruction"` 这两行**从未输出**，说明程序跳回到了 `"Testing breakpoint exception..."` 的位置。

5.1 问题分析

我们在最初的实现中，**简单地假设所有指令都是4字节**：

```
case CAUSE_BREAKPOINT:
    cprintf("Exception type: breakpoint\n");
    cprintf("ebreak caught at 0x%08x\n", tf->epc);
    tf->epc += 4;  // 假设所有指令都是4字节
    break;
```

查阅资料后发现RISC-V 架构支持**变长指令编码**：

指令类型	长度	编码特征
压缩指令（C 扩展）	16位（2字节）	最低2位 ≠ `11`
标准指令	32位（4字节）	最低2位 = `11`

常见的压缩指令有 `c.ebreak` - 压缩的断点指令、`c.nop` - 压缩的空操作、`c.addi` - 压缩的立即数加法等。

当我们启用 `-march=rv64gc` 或 `-march=rv32gc` 时，编译器会尽可能使用压缩指令来减小代码体积，所以 `ebreak` 很可能被编译为 `c.ebreak`（2字节）

5.2 错误执行流程分析

假设 我们预设的4字节 `ebreak` 被编译为 2字节的 `c.ebreak`：

```
地址布局：
0xc02000a8: [c.ebreak]      # 2字节压缩指令
0xc02000aa: [下一条指令]     # 正确的下一条指令
0xc02000ac: [某条指令]       # 实际上是另一条指令
错误执行流程：
1. 执行 c.ebreak @ 0xc02000a8
2. 异常处理：epc = 0xc02000a8
3. 错误更新：epc += 4 = 0xc02000ac
4. sret 返回到 0xc02000ac
问题：
- 正确的下一条指令在 0xc02000aa
```

```
- 但我们跳到了 0xc02000ac
- 跳过了正确的下一条指令（0xc02000aa）
- 0xc02000ac 可能是某条指令的中间部分或另一条指令
- 导致程序执行到错误的位置，最终跳回循环开始
```

程序重新执行测试代码，形成死循环。

5.3 解决方案

我们根据 RISC-V 规范，通过指令的**最低2位**判断指令类型：

```c
// 读取指令的前16位（2字节）
unsigned int instruction = *(unsigned short *)tf->epc;

// 判断最低2位
if ((instruction & 0x3) != 0x3) {
    // 最低2位不是 11 (0b11)，是压缩指令
    tf->epc += 2;
} else {
    // 最低2位是 11，是标准指令
    tf->epc += 4;
}
```

编码判断依据：

最低2位	二进制	指令类型	长度
`00`	`0b00`	压缩指令	2字节
`01`	`0b01`	压缩指令	2字节
`10`	`0b10`	压缩指令	2字节
`11`	`0b11`	标准指令	4字节

完整的异常处理代码：

```c
case CAUSE_ILLEGAL_INSTRUCTION:
    // 非法指令异常处理
    cprintf("Exception type: Illegal instruction\n");
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);

    // 正确判断指令长度
    unsigned int instruction = *(unsigned short *)tf->epc;
    if ((instruction & 0x3) != 0x3) {
        tf->epc += 2;  // 压缩指令，长度2字节
    } else {
        tf->epc += 4;  // 标准指令，长度4字节
    }
    break;

case CAUSE_BREAKPOINT:
    // 断点异常处理
    cprintf("Exception type: breakpoint\n");
    cprintf("ebreak caught at 0x%08x\n", tf->epc);

```

```
    // 正确判断指令长度
    // ebreak 可能是压缩指令 c.ebreak(2字节) 或标准 ebreak(4字节)
    unsigned int inst = *(unsigned short *)tf->epc;
    if ((inst & 0x3) != 0x3) {
        tf->epc += 2;  // 压缩指令，长度2字节
    } else {
        tf->epc += 4;  // 标准指令，长度4字节
    }
    break;
```

6 实验结果

修复压缩指令问题后，编译并运行，输出结果如图：

```
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
  entry  0xffffffffc0200090 (virtual)
  etext  0xffffffffc0202008 (virtual)
  edata  0xffffffffc0207028 (virtual)
  end    0xffffffffc0207498 (virtual)
Kernel executable memory footprint: 30KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
++ setup timer interrupts
Testing breakpoint exception...
Exception type: breakpoint
ebreak caught at 0xc0200064
After ebreak: breakpoint exception handled successfully!

Testing illegal instruction exception...
Exception type: Illegal instruction
Illegal instruction caught at 0xc020007e
Exception type: Illegal instruction
Illegal instruction caught at 0xc0200080
After illegal instruction: exception handled successfully!
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

```
Testing breakpoint exception...
Exception type: breakpoint
ebreak caught at 0xc02000a8
After ebreak: breakpoint exception handled successfully!

Testing illegal instruction exception...
Exception type: Illegal instruction
Illegal instruction caught at 0xc02000b0
After illegal instruction: exception handled successfully!
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

```
100 ticks
100 ticks
100 ticks
[系统关机]
```

异常信息正确输出，异常处理后继续执行时钟中断测试，输出10次 "100 ticks" 后系统关机。

实验知识点总结

1. 中断与异常处理机制

OS原理知识点：中断和异常是操作系统实现并发控制、设备管理和错误处理的核心机制。中断由外部设备异步触发，异常由CPU执行指令同步产生。操作系统需要建立中断向量表、保存上下文、分类处理、安全返回。

实验对应：本实验通过实现时钟中断和异常处理，完整展现了RISC-V架构下的中断处理流程。从`stvec`寄存器设置中断入口`__alltraps`，到`SAVE_ALL`保存上下文形成trapframe结构体，再到`trap_dispatch`分类处理，最后`RESTORE_ALL`恢复上下文并通过`sret`返回。实验特别强调了`sepc`、`scause`、`sstatus`等CSR寄存器在中断处理中的关键作用。

关系与差异：原理上中断处理是通用的，但具体实现依赖硬件架构。RISC-V与x86在寄存器组织、特权级设计上有显著差异。实验中需要深入理解RISC-V特有的CSR寄存器语义，如`sscratch`用于内核栈指针暂存，这在原理教学中较少涉及。

2. 上下文切换与状态保存

OS原理知识点：上下文切换是操作系统实现多任务的基础，需要在进程切换时完整保存和恢复CPU状态，包括通用寄存器、程序计数器、状态寄存器等。

实验对应：`SAVE_ALL`宏和`RESTORE_ALL`宏实现了完整的上下文保存与恢复。实验通过trapframe结构体将寄存器状态组织成统一格式，实现了从汇编到C函数的参数传递。特别值得注意的是对`sp`寄存器的特殊处理：通过`sscratch`暂存原栈指针，确保递归异常的正确处理。

深入理解：实验揭示了上下文保存不仅是简单的寄存器压栈，还需要考虑架构特性。如RISC-V中某些CSR（`stval`、`scause`）只需读取不需恢复，而`sepc`、`sstatus`必须精确恢复。这种区分反映了状态信息的生命周期差异。

3. 定时器与时钟中断

OS原理知识点：时钟中断是操作系统实现时间片轮转、定时任务、性能监控等功能的基础，提供了系统的时间感知能力。

实验对应：通过SBI调用设置定时器，处理`IRQ_S_TIMER`中断，维护全局`ticks`计数器。实验展示了时钟中断的完整生命周期：设置下次中断时间、更新计数器、周期性任务触发、系统关机控制。

实践意义：实验不仅实现了原理中的时钟中断概念，还揭示了实际系统中的时间管理细节。如必须在处理当前中断时立即设置下次中断，否则会丢失时钟事件；全局`ticks`需要原子性更新考虑。

4. 异常分类与处理策略

OS原理知识点：操作系统需要对不同类型的异常采取不同的处理策略，有的需要终止进程，有的可以修复后继续执行，有的需要内核介入。

实验对应：实验实现了非法指令和断点异常的处理，展示了异常恢复的典型模式：诊断信息输出、程序状态修复（`epc`调整）、继续执行。特别重要的是对指令长度的正确判断，避免因压缩指令导致的执行流错误。

架构特性：实验深入揭示了RISC-V的指令编码特性，通过指令最低2位区分压缩指令与标准指令。这种硬件细节在OS原理中通常抽象化，但在实际实现中至关重要。

实验未覆盖的知识点

1. 中断优先级与嵌套中断

OS原理中强调中断优先级管理和嵌套中断处理，高优先级中断可以打断低优先级中断的处理。实验中采用了简单的全程关中断策略，没有实现中断优先级和嵌套。实际系统中需要更精细的中断控制，如RISC-V的`sie`寄存器可以按类型单独控制中断使能。

2. 中断负载均衡与多核处理

现代多核系统中，中断可以定向到特定CPU核心，实现负载均衡。实验仅在单核环境下实现中断处理，没有涉及多核间的中断分发、核间中断（IPI）等复杂机制。

3. 中断下半部处理机制

原理中常讨论的中断上下半部分离机制（如软中断、tasklet、workqueue等）在实验中没有体现。实验中的所有处理都在中断上下文中完成，没有将耗时操作延迟到下半部。

4. 虚拟化相关中断处理

随着虚拟化技术的普及，中断处理需要区分物理中断和虚拟中断，支持中断重映射等虚拟化特性。实验仅涉及物理模式下的中断处理，没有触及虚拟化扩展。

5. 实时系统中的中断响应优化

实时操作系统对中断响应时间有严格要求，需要优化中断延迟、实现中断抢占等机制。实验环境没有考虑实时性要求，中断处理流程相对简单。