

Lab3

小组成员：朱欣宁、雷雨彦、宁宇嫣

练习1

编程完善trap.c中的中断处理函数trap，填写kern/trap/trap.c函数中处理时钟中断的相关部分，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。

所以我们按照题目提示和代码中的相关注释对kern/trap/trap.c中的部分代码进行修改，修改后代码如下：

```
// 定义全局变量
static int ticks_count = 0; // 时钟中断计数器
static int print_count = 0; // 打印次数计数器
case IRQ_S_TIMER:
    // (1) 设置下次时钟中断- clock_set_next_event()
    clock_set_next_event();
    // (2) 计数器 ( ticks_count ) 加一
    ticks_count++;
    // (3) 当计数器加到100的时候，我们会输出一个`100ticks`表示我们触发了100次时钟中断，
    同时打印次数 ( print_count ) 加一
    if (ticks_count % TICK_NUM == 0) {
        print_ticks();
        print_count++;
    }
    // (4) 判断打印次数，当次数为10时，调用<sbi.h>中的关机函数关机
    if (print_count == 10) {
        sbi_shutdown();
    }
}
break;
```

简单分析我们的代码/定时器中断处理流程

- 分析前面的代码。中断触发后，CPU会自动跳转到stvec指向的__alltraps，保存所有寄存器到trapframe结构体并完成中断分发。
- 我们所编写的这个模块是**时钟中断处理程序**，用于处理S模式的定时器中断，实现系统的定时功能和关机控制；
- 先定义了两个计数器ticks_count和num_count，分别用于记录发生的时钟中断次数和记录已经打印的次数；
- 设置下一次时钟中断，中断后ticks_count加1，再用if语句实现每100次中断执行一次打印（即num_count++）；
- 判断打印次数，调用sbi_shutdown()实现在10次时关机（注：因为要使用这个函数，所以我们要在trap.c这一文件的头文件中添上<sbi.h>）。

运行结果

我们用make qemu编译文件后得到下图，完成了打印和关机，说明我们的程序是正确的：

```
leiyuyan@leiyuyan-VMware-Virtual-Platform: ~/riscv/lab3
entry 0xffffffffc0200054 (virtual)
etext 0xffffffffc0201fa0 (virtual)
edata 0xffffffffc0207028 (virtual)
end 0xffffffffc02074a0 (virtual)
Kernel executable memory footprint: 30KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000008000000, 0x00000000087fffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000
satp physical address: 0x00000000080206000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
leiyuyan@leiyuyan-VMware-Virtual-Platform:~/riscv/lab3$ S
```

总结整个实现流程

这个实验主要完成了中断处理机制的搭建，具体包括：

- 中断的初始化。在idt_init()中设置中断向量表后，将stvec寄存器指向中断处理入口__alltraps，保存当前程序状态并调用trap()函数，执行时钟中断处理代码后恢复现场，继续原来程序；
- 中断的分发。通过 trap() → trap_dispatch() 将中断/异常分发给对应的处理程序；
- 时钟中断处理。在IRQ_S_TIMER分支中实现定时功能，每100次中断打印一次，10次后关机（即我们实现的功能）

扩展练习Challenge一

1. ucore 中断异常处理流程分析

整体处理流程

ucore 的中断异常处理可以分为四个主要阶段：

阶段一：异常产生与硬件自动处理

1. 保存当前 PC → sepc 寄存器
2. 记录异常原因 → scause 寄存器
3. 保存附加信息 → stval 寄存器
4. 保存中断使能状态：sstatus.SIE → sstatus.SPIE
5. 禁用中断：sstatus.SIE = 0
6. 保存当前特权级 → sstatus.SPP
7. 跳转到 stvec 指向地址 (__alltraps)

阶段二：汇编处理入口(trapentry.S)

```
__alltraps:
    SAVE_ALL                # 保存完整寄存器上下文
    move a0, sp             # 传递 trapframe 指针参数
    jal trap                # 调用 C 语言处理函数
```

阶段三：C 语言中断分发 (trap.c)

```
void trap(struct trapframe *tf) {
    trap_dispatch(tf);        // 根据异常类型分发处理
}

static inline void trap_dispatch(struct trapframe *tf) {
    if ((intptr_t)tf->cause < 0) {
        interrupt_handler(tf); // 处理硬件中断
    } else {
        exception_handler(tf); // 处理异常
    }
}
```

阶段四：上下文恢复与返回

```
__trapret:
    RESTORE_ALL              # 恢复保存的寄存器上下文
    sret                    # 返回原执行流
```

2. 关键问题分析

2.1 mov a0,sp指令的作用

目的：将当前的栈指针sp作为参数传递给C语言中断处理函数trap()

具体分析：

- 在SAVE_ALL宏执行后，栈上已经构建了一个完整的trapframe结构体

- `sp`指向这个结构体的起始地址
- 按照RISC-V调用约定，第一个参数通过`a0`寄存器传递
- 这样在`trap(struct trapframe *tf)`函数中，就能通过`tf`参数访问所有保存的寄存器状态

2.2 SAVE_ALL 中寄存器保存位置确定机制

保存位置完全由C语言的 `struct trapframe` 结构体定义决定。SAVE_ALL 宏只是在栈上按这个结构体的内存布局，把寄存器值精确存入对应的偏移量位置。

```
struct pushregs {
    uintptr_t zero; // 槽位 0: 对应 x0
    uintptr_t ra;    // 槽位 1: 对应 x1
    uintptr_t sp;    // 槽位 2: 对应 x2
    uintptr_t gp;    // 槽位 3: 对应 x3
    uintptr_t tp;    // 槽位 4: 对应 x4
    // ... 以此类推，直到 x31
};

struct trapframe {
    struct pushregs gpr; // 这占据了前 32 个槽位 (0-31)
    uintptr_t status;    // 槽位 32
    uintptr_t epc;       // 槽位 33
    uintptr_t badvaddr;  // 槽位 34
    uintptr_t cause;     // 槽位 35
};
```

通用寄存器 `xN` 被保存在 `struct pushregs` 中对应的成员位置。

- 例如：`x1 (ra)` 保存在 `gpr.ra`，对应槽位1，所以汇编是 `STORE x1, 1*REGBYTES(sp)`。
- 关键例外：`x2 (sp)` 保存在 `gpr.sp`，对应槽位2，所以汇编是 `STORE s0, 2*REGBYTES(sp)` (这里的 `s0` 暂存了原始`sp`的值)。

CSR 寄存器 被保存在 `struct trapframe` 的后续成员中。

- 例如：`sstatus` 是 `struct trapframe` 的第1个CSR成员，所以在槽位32，汇编是 `STORE s1, 32*REGBYTES(sp)` (这里的 `s1` 暂存了`sstatus`的值)。

特殊处理机制：

1. 栈指针特殊保存：

```
csrw sscratch, sp      # 初始保存原 sp 到 sscratch
csrrw s0, sscratch, x0 # 交换获取原 sp 值到 s0
STORE s0, 2*REGBYTES(sp) # 保存到 trapframe 中
```

1. CSR 寄存器保存：

```
csrr s1, sstatus      # 读取状态寄存器
csrr s2, sepc         # 读取异常 PC
csrr s3, sbadaddr     # 读取错误地址
csrr s4, scause       # 读取异常原因
# 然后通过临时寄存器保存到栈中
```

2.3 全寄存器保存的必要性分析

需要保存所有寄存器，理由如下：

必要性：

1. 通用性设计：中断处理程序不知道具体是哪种中断，必须假设所有寄存器都可能被使用
2. 上下文完整性：为了能够正确返回到被中断的代码，必须完整保存所有寄存器状态
3. 特权级切换：当中断来自用户态时，需要完整保存用户态上下文以便正确返回

特殊情况考虑：

1. 时钟中断：可能触发进程调度，需要保存完整上下文以便切换到其他进程
2. 系统调用：需要访问参数寄存器，修改返回值寄存器
3. 页错误：处理完成后需要重新执行指令，依赖所有寄存器状态

扩展练习Challenge二

1. csrw sscratch, sp; csrrw s0, sscratch, x0 操作分析

1.1 指令处理流程

```
csrwr sscratch, sp      # 将当前sp的值保存到sscratch寄存器
csrrw s0, sscratch, x0  # 将sscratch的值读取到s0，同时将x0(零)写入sscratch
```

1.2 操作目的与机制

主要目的：

- 保存原始栈指针：将进入中断前的栈指针安全保存
- 标记内核态中断：通过设置sscratch=0来标记当前中断来自内核态
- 递归中断检测：为可能发生的嵌套中断提供检测机制

详细执行过程：

- 步骤一：保存原始指针

```
csrwr sscratch, sp
```

- 执行前：sscratch = 未知值, sp = 用户栈或内核栈指针
- 执行后：sscratch = 原始sp值, sp = 保持不变
- 步骤二：交换并标记

```
csrrw s0, sscratch, x0
```

- 操作：s0 \leftarrow sscratch, sscratch \leftarrow x0(0)
- 执行后：s0 = 原始sp值, sscratch = 0

1.3 设计原理

上下文区分机制：

```
// sscratch 的值用于判断中断来源：
// sscratch == 0: 中断来自内核态 (S-mode)
// sscratch != 0: 中断来自用户态 (U-mode) · 此时sscratch存储内核栈指针
```

嵌套中断保护：

```
# 如果发生递归中断，硬件会检查sscratch：
# - 如果sscratch == 0，说明已经在处理内核中断
# - 可以采取特殊处理避免栈溢出
```

2. CSR寄存器保存与恢复的不对称性分析

2.1 保存的CSR寄存器

在SAVE_ALL中保存的CSR：

```
csrr s1, sstatus    # 保存状态寄存器
csrr s2, sepc       # 保存异常程序计数器
csrr s3, sbadaddr   # 保存错误地址(stval)
csrr s4, scause     # 保存异常原因
```

2.2 恢复的CSR寄存器

在RESTORE_ALL中只恢复了：

```
csrw sstatus, s1    # 恢复状态寄存器
csrw sepc, s2       # 恢复异常程序计数器
# 注意：sbadaddr(stval)和scause没有恢复
```

2.3 不对称设计的理论依据

寄存器性质分类

寄存器	性质	是否需要恢复	理由
sstatus	系统状态	必须恢复	控制中断使能、特权级等关键状态
sepc	返回地址	必须恢复	决定中断返回后继续执行的位置
scause	原因记录	不需要恢复	只读寄存器，记录历史信息
sbadaddr/stval	附加信息	不需要恢复	只读寄存器，提供诊断信息

2.4 保存的意义

虽然不恢复，但保存这些CSR仍有重要价值：保存CSR（控制状态寄存器）的意义主要体现在以下几个方面：

- 保存sstatus和sepc是为了系统状态的完整恢复：**sstatus寄存器包含了中断使能位、特权级状态等关键系统配置，必须在中断处理后恢复原状以确保系统继续正常运行。sepc保存了中断发生时的程序计数器值，是程序能够正确返回到中断点继续执行的关键。
- 保存scause和stval（sbadaddr）是为了中断诊断和处理：**虽然这些寄存器不需要恢复，但它们的保存为中断处理程序提供了重要的上下文信息。scause记录了中断的具体原因，使处理程序能够区分时钟中断、系统调用、页错误等不同类型的中断。stval则提供了与异常相关的附加信息，如在缺页异常中保存触发异常的虚拟地址，在非法指令异常中保存出错的指令内容。
- 完整的CSR保存保证了系统调试和错误诊断的能力：**当系统出现异常时，保存的所有CSR信息可以用于事后分析，帮助开发者理解系统状态和异常发生的原因。这种设计也为嵌套中断处理提供了支持，在复杂的中断场景下能够维护完整的上下文信息。
- 保持trapframe结构的一致性：**为系统扩展和维护提供了便利，即使某些CSR在当前不需要恢复，统一的保存机制确保了代码结构的清晰和未来功能扩展的灵活性。

扩展练习Challenge三：完善异常中断

异常中断概述

什么是异常中断？

异常中断是指CPU在执行指令过程中遇到的非正常情况，如非法指令、断点、除零等。当异常发生时，CPU会暂停当前程序执行，跳转到预设的异常处理程序，处理完成后可能返回原程序继续执行。

异常中断的解决方法

- 识别异常类型：**通过异常原因码确定具体异常
- 保存现场：**保存寄存器状态等执行上下文
- 处理异常：**执行相应的异常处理逻辑
- 恢复执行：**更新程序计数器，恢复或终止程序执行

非法指令异常和断点异常

非法指令异常 (Illegal Instruction)

- 原因码：`CAUSE_ILLEGAL_INSTRUCTION` (2)
- 触发条件：CPU遇到无法识别或执行的指令编码
- 常见情况：未定义的操作码、权限不足的指令

断点异常 (Breakpoint)

- 原因码：`CAUSE_BREAKPOINT` (3)
- 触发条件：执行`ebreak`指令
- 主要用途：调试器设置断点、程序调试

代码修改位置

文件位置：`kern/trap/trap.c`

修改的代码

```
// 调整 epc 到下一条指令
static void advance_epc(struct trapframe *tf) {
    // 1. 获取指令本身
    // (uint16_t *) 将这个地址强制转换为一个指向 16 位无符号整数的指针。
    uint16_t instruction = *(uint16_t *)(tf->epc);

    // 2. 检查最低两位
    if ((instruction & 0x3) == 0x3) {
        // 如果最低两位是 '11'，说明这是一条 32 位标准指令。
        // 我们需要跳过 4 个字节才能到下一条指令。
        tf->epc += 4;
    } else {
        // 如果最低两位不是 '11'，说明这是一条 16 位压缩指令。
        // 我们只需要跳过 2 个字节。
        tf->epc += 2;
    }
}

case CAUSE_ILLEGAL_INSTRUCTION:
    // 非法指令异常处理
    /* LAB3 CHALLENGE3 2313686 : */
    /*(1)输出指令异常类型 ( Illegal instruction)
    *(2)输出异常指令地址
    *(3)更新 tf->epc寄存器
    */
    cprintf("Exception type: Illegal instruction\n");
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
    advance_epc(tf);
    break;

case CAUSE_BREAKPOINT:
    //断点异常处理
    /* LAB3 CHALLLENGE3 2313686 : */
    /*(1)输出指令异常类型 ( breakpoint)
```

```
*(2)输出异常指令地址
*(3)更新 tf->epc寄存器
*/
cprintf("Exception type: breakpoint\n");
cprintf("Breakpoint caught at 0x%08x\n", tf->epc);
advance_epc(tf);
break;
```

设计原理说明

为什么要这样设计？

1. 智能指令长度检测

- RISC-V架构支持16位压缩指令和32位标准指令混合编码
- 传统固定偏移方法（如总是+4）无法正确处理压缩指令
- 通过检查指令最低两位智能判断指令长度：
 - 0x3 (11₂)：32位标准指令 → 偏移4字节
 - 其他值：16位压缩指令 → 偏移2字节

2. 通用性提升

- `advance_epc()`函数封装了指令前进逻辑
- 可同时处理非法指令和断点异常
- 支持各种指令格式，提高代码复用性

3. 调试信息完善

- 输出异常类型和发生地址
- 便于定位问题和调试程序

测试示例

测试代码

```
cprintf("Testing exception handling...\n");

// 测试断点异常
cprintf("Trigger breakpoint...\n");
asm volatile("ebreak");
cprintf("Breakpoint handled.\n");

// 测试非法指令异常
cprintf("Trigger undefined instr...\n");
asm volatile(".word 0xffffffff");
cprintf("Undefined instr handled.\n");
```

测试结果展示

```
Size: 0x0000000008000000 (128 MB)
End: 0x00000000087ffffff
DTB init completed
(THU.CST) os is loading ...
Special kernel symbols:
entry 0xffffffffc0200054 (virtual)
etext 0xffffffffc0201ea4 (virtual)
edata 0xffffffffc0206028 (virtual)
end 0xffffffffc02064a0 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physical memory map:
memory: 0x0000000008000000, [0x0000000008000000, 0x00000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x00000000080205000
++ setup timer interrupts
Testing exception handling...
Trigger breakpoint...
Exception type: breakpoint
Breakpoint caught at 0xc02000b4
Breakpoint handled.
Trigger undefined instr...
Exception type: Illegal instruction
Illegal instruction caught at 0xc02000ce
Undefined instr handled.
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

结果分析

1. 断点异常测试：

- 成功识别 `ebreak` 指令触发的断点异常
- 正确输出异常类型和地址
- 正常恢复程序执行

2. 非法指令测试：

- 正确识别未定义指令 `0xffffffff`
- 准确报告非法指令异常
- 程序继续执行后续代码

总结

本次异常中断处理的完善实现了：

1. 正确的异常识别：能够准确区分非法指令和断点异常

- 这种设计不仅解决了基本的异常处理需求，还通过指令长度自适应机制提升了代码的健壮性和可移植性，为后续的系统开发和调试工作奠定了坚实基础。

[illegible]

1.中断与异常的基本概念

- OS原理：中断是异步事件，由外部设备触发；异常是同步事件，由当前执行指令触发。二者统称为“陷阱（Trap）”，是操作系统实现并发、保护与响应机制的基础。
- 实验内容：在 trap.c 中通过 scause 寄存器区分中断与异常，并在 trap_dispatch() 中分别调用 interrupt_handler() 和 exception_handler()。
- 关系与差异：实验严格遵循RISC-V架构对Trap的分类，与原理一致；差异在于RISC-V将系统调用（ecall）也归类为异常，而传统OS中常将其视为“软中断”。

2.中断处理流程与上下文切换

- OS原理：中断处理需保存当前执行上下文（寄存器状态），跳转至中断处理程序，处理完毕后恢复上下文，继续原流程。
- 实验内容：在 `trapentry.S` 中通过 `SAVE_ALL` 和 `RESTORE_ALL` 宏实现上下文的保存与恢复，使用 `trapframe` 结构体组织寄存器与CSR。
- 关系与差异：实验通过汇编代码直接实现上下文切换，与原理一致；差异在于实验中使用 `sscratch` 辅助判断中断来源（U/S模式），这是RISC-V特有的机制。

3.中断向量表与入口点

- OS原理：中断向量表将不同类型的中断映射到不同的处理程序入口。

- 实验内容：使用 `stvec` 寄存器设置为 `__alltraps` 地址，采用直接模式 (Direct Mode)，所有中断均跳转至同一入口。
- 关系与差异：实验未使用向量模式，而是统一入口再分发，简化实现；这与某些体系结构 (如x86) 使用多入口向量表不同。

4.特权级与中断路由

- OS原理：现代CPU通过特权级实现权限隔离，中断可在不同特权级间触发与处理。
- 实验内容：RISC-V 的 M/S/U 三级特权级，通过 `mideleg/medeleg` 将中断/异常委托至 S 模式处理。
- 关系与差异：实验通过 OpenSBI 完成初始委托，使大部分中断由 S 模式处理，符合微内核与分层保护的设计思想。

5. 时钟中断与调度机制

- OS原理：时钟中断是抢占式调度的基础，通过定期中断实现时间片轮转。
- 实验内容：在 `clock.c` 中初始化时钟，设置周期性中断，并在 `trap.c` 中处理 `IRQ_S_TIMER`，累加 `ticks` 并触发下一次中断。
- 关系与差异：实验实现了时钟中断的触发与响应，但未实现完整的进程调度，仅通过计数模拟时间片。

6.中断使能与屏蔽

- OS原理：内核在执行关键代码段时需屏蔽中断，保证原子性。
- 实验内容：通过 `sstatus.SIE` 控制中断使能，并在 `sync.h` 中提供 `local_intr_save/restore` 接口，用于内存分配等关键操作。
- 关系与差异：实验通过CSR直接控制中断状态，与原理一致；但未讨论中断优先级或嵌套中断处理。

实验中未覆盖的操作系统原理知识点

中断优先级与嵌套中断：

- 原理中常见的中断控制器 (如APIC) 支持优先级与嵌套处理，实验未涉及。

完整的中断描述符表 (IDT)：

- 实验中使用单一入口点，未展示多入口向量表的实现。

中断与进程调度的深度融合：

- 实验仅模拟时钟中断，未实现基于时间片的进程切换、调度队列等。

设备中断与驱动模型：

- 实验仅涉及时钟中断，未涉及其他外设 (如磁盘、网络) 的中断处理与驱动框架。

异常处理的恢复策略：

- 如缺页异常的处理与页面调入、指令重试等机制未在实验中实现。

系统调用接口与参数传递：

- 实验未实现完整的系统调用机制，如参数从用户态传递至内核态。