Operating System Lab 3

禹相祐（2312900）查科言（2312189）董丰瑞（2311973）

一、练习1：完善中断处理

> 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。

实现代码如下：

```
clock_set_next_event();
extern volatile size_t ticks;
ticks++;
if (ticks % TICK_NUM == 0) {
    print_ticks();
    tick_print_times++;
    if (tick_print_times >= 10) {
        sbi_shutdown();
    }
}
```

实现过程：

- **设置下一次时钟中断**：通过`clock_set_next_event()`函数，基于当前时间（`get_time()`）加上固定时间间隔（`timebase`，`QEMU`中为 10ms），确保时钟中断周期性触发（每 10ms 一次）。
- **计数与判断**：
  - 用全局变量`ticks`累计时钟中断次数，每次中断触发时`ticks++`。
  - 当`ticks`是`TICK_NUM`（100）的倍数时（即累计 1 秒），调用`print_ticks()`输出 “100 ticks”。

- 用 `tick_print_times` 记录打印次数，累计到 10 次（即 10 秒）时，调用 `sbi_shutdown()` 触发系统关机。

定时器中断流程处理

- **中断触发**：当 CPU 的 `time` 寄存器达到 `clock_set_next_event()` 设置的时间时，硬件自动触发 `IRQ_S_TIMER` 中断。
- **上下文保存**：CPU 跳转到 `__alltraps` 入口，通过 `SAVE_ALL` 宏将寄存器和中断相关 CSR（`sstatus`、`sepc` 等）保存到栈，形成 `trapframe`
- **中断分发与处理**
  - `trap()` 函数通过 `trap_dispatch()` 识别为时钟中断，进入 `interrupt_handler()` 的 `IRQ_S_TIMER` 分支。执行我们的代码逻辑，设置下次中断→更新计数器→判断是否打印或关机。
- **恢复与返回**：通过 `RESTORE_ALL` 宏恢复寄存器和 `CSR`，执行 `sret` 返回被中断的程序，等待下一次时钟中断。

整个流程确保系统每 1 秒输出一次提示，10 秒后自动关机，验证了时钟中断的周期性和正确性。

二、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），其中 `mov a0，sp` 的目的是什么?
> 2. SAVE_ALL中寄寄存器保存在栈中的位置是什么确定的?
> 3. 对于任何中断，`__alltraps` 中都需要保存所有寄存器吗？请说明理由。

（1）异常处理的流程

ucore 中断异常处理完整流程

1. **异常 / 中断触发**：
   - 中断（异步）：如时钟到点、外设完成 I/O，硬件自动触发。
   - 异常（同步）：如非法指令、缺页，执行当前指令时触发。
2. **硬件自动处理**：
   - 保存被中断指令地址到 `sepc`，记录原因到 `scause`，附加信息到 `stval`。
   - 保存当前特权级到 `sstatus.SPP`，保存中断使能状态到 `sstatus.SPIE`，并禁用 S 模式中断（`sstatus.SIE=0`）。
   - 根据 `stvec` 寄存器的值，跳转到中断入口 `__alltraps`（汇编代码）
3. **汇编层上下文保存（`trapentry.S:__alltraps`）**
   - 调用 `SAVE_ALL` 宏，将 32 个通用寄存器和 4 个关键 `CSR`（`sstatus`、`sepc`、`sbadvaddr`、`scause`）保存到栈上，形成 `struct trapframe`。
   - 执行 `move a0, sp`，将栈顶（即 `trapframe` 的地址）作为参数传给 C 函数 `trap()`。
   - 调用 `jal trap`，跳转到 C 语言中断处理函数。
4. **C 语言层分发与处理**
   - `trap()` 调用 `trap_dispatch()`，根据 `tf->cause` 判断是中断还是异常。
   - 中断交给 `interrupt_handler()`，按类型处理（如时钟中断计数、外设中断响应）；异常交给 `exception_handler()`，按类型处理（如非法指令提示、缺页处理）。
5. **恢复与返回**

- 处理完成后，返回到 `__trapret`，调用 `RESTORE_ALL` 宏，从栈上恢复通用寄存器和关键 CSR（`sstatus`、`sepc`）。
- 执行 `sret` 指令，恢复 `pc` 为 `sepc` 的值，按 `sstatus.SPP` 切回原特权级，恢复中断使能（`sstatus.SIE = sstatus.SPIE`）。

(2) `move a0 ,sp` 的作用

我们已经知道，`RICS-V` 的函数调用规定，`a0` 寄存器用于传递第一个函数参数。

`SAVE_ALL` 将上下文保存到栈，`sp` 此时指向栈上的 `struct trapframe`，将 `sp` 传给 `trap()`，本质是让 C 函数拿到 “上下文快照” 即 `trapframe` 的地址，后续可通过 `tf` 指针读取中断原因（`tf->cause`）、被中断地址（`tf->epc`）等关键信息，完成针对性处理。

(3) `SAVE_ALL` 中寄存器在栈中的位置由什么确定

- 由 `struct trapframe` 和 `struct pushregs` 的结构体定义**固定确定。**
- 栈上的 `trapframe` 与结构体成员一一对应：前 32 个位置（0×8~31×8）对应 `struct pushregs` 的 x0~x31，后 4 个位置（32×8~35×8）对应 `sstatus`、`sepc`、`sbadvaddr`、`scause`。
- 汇编代码 `STORE x0, 0*REGBYTES(sp)`、`STORE s1, 32*REGBYTES(sp)` 等操作，正是严格按照结构体成员的内存偏移来保存，确保后续恢复时能正确映射到寄存器。

(4) 任何中断都需要在 `__alltraps` 中保存所有寄存器吗？理由

- **是，必须保存所有寄存器。**
- 理由 1：中断具有 “随机性”，无法预知被中断程序正在使用哪些寄存器。若只保存部分寄存器，恢复后未保存的寄存器值会被破坏，导致程序执行错误，如计算结果错误、函数返回地址丢失。
- 理由 2：`trapframe` 是 **“完整上下文快照”**，后续若需要进程切换（如时钟中断触发调度），需基于完整的寄存器状态切换到新进程，缺少任何一个寄存器都会导致切换失败。

三、Challenge 2：理解上下文切换机制

> 1. 在 `trapentry.S` 中汇编代码 `srw sscratch, sp`；`csrrw s0sscratch, x0` 实现了什么操作，目的是什么?
> 2. 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?

(1) `csrw sscratch, sp`；`csrrw s0, sscratch, x0` 的操作与目的

操作解析：

- `csrw sscratch, sp`：将当前的栈指针（`sp`）写入 `sscratch` 寄存器（RISC-V 专门用于临时存储栈指针的 `CSR`）。
- `csrrw s0, sscratch, x0`：原子交换 `sscratch` 和 `s0` 的值 —— 最终 `s0` 保存 “`SAVE_ALL` 执行前的旧 `sp`”，`sscratch` 被清零。

目的：

- **保存 “未开辟栈空间前的旧 `sp`**
  - `SAVE_ALL` 第一步会执行 `addi sp, sp, -36*REGBYTES`（栈向下开辟空间存 `trapframe`），此时 `sp` 会被修改为 “新栈顶”。而我们需要在 `trapframe` 中保存的是 “修改前的旧 `sp`”（被中断程序的栈指针），

- 因此先通过`csrw sscratch, sp`暂存旧`sp`，再通过交换指令将其存入`s0`，后续通过`STORE s0, 2*REGBYTES(sp)`写入`trapframe`的`x2（sp）`位置，确保上下文完整。

- **区分中断来源（内核态 / 用户态）**
  - 系统初始化时`sscratch`被设为 0（`idt_init()`中`write_csr(sscratch, 0)`）。
  - 若中断来自内核态，`sscratch`初始为 0，交换后`s0`为旧`sp`，`sscratch`为 0；
  - 若中断来自用户态，`sscratch`提前存入内核栈地址，交换后可切换到内核栈处理，避免污染用户栈。

（2）`SAVE_ALL`保存`stval`、`scause`等 CSR，`RESTORE_ALL`却不还原的原因

- **保存的意义**
  - `stval`（异常相关地址）、`scause`（中断 / 异常原因）是 中断处理的关键信息，保存到`trapframe`是为了让 C 函数`trap()`能读取这些信息，完成针对性处理，如：`scause`判断是时钟中断还是非法指令，`stval`定位缺页地址

- **不还原的原因**
  - 这些 CSR 是 “一次性事件信息”，仅对当前中断 / 异常有效，无需恢复给被中断程序：
    1. `scause`：记录的是 “本次中断原因”，被中断程序原本的执行中不存在该 “原因”，恢复会导致错误。
    2. `stval`：记录的是 “本次异常的附加信息”（如非法地址），被中断程序后续执行不需要该信息，恢复无意义。
    3. 仅需还原 “影响程序继续执行的状态”：`sstatus`（中断使能）、`sepc`（被中断地址）、通用寄存器（运算状态），这些是程序恢复执行的必要条件；而`stval`、`scause`属于 “一次性条件”，处理完后即可丢弃。

四、Challenge3：完善异常中断

> 编程完善在触发一条非法指令异常和断点异常，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`。

（1）非法指令异常处理

```
case CAUSE_ILLEGAL_INSTRUCTION:
             // 非法指令异常处理
             /* LAB3 CHALLENGE3   YOUR CODE :  */
            /*(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("Illegal instruction caught at 0x%08x\n", tf->epc);
            cprintf("Exception type:Illegal instruction\n");
            {
                uint16_t first_half = *(uint16_t *)(tf->epc);
                tf->epc += ((first_half & 0x3) == 0x3) ? 4 : 2;
            }
            break;
```

我们的目的是定位非法指令位置，避免程序陷入死循环。

- 实现思路：
  - **输出异常类型与地址**：通过`cprintf`打印 **“Illegal instruction”** 类型，并读取`tf->epc`（硬件自动保存的异常指令地址），输出具体位置，方便定位错误。
  - **动态调整`tf->epc`**：
    - RISC-V 支持 16 位压缩指令（`RVC`）和 32 位标准指令，如果指令最后 2 位是`0x3`（二进制`11`），说明是 32 位标准指令，要跳 4 字节才能跳过它，如果不是`0x3`，就是 16 位压缩指令，跳 2 字节就行。
    - 通过读取指令低 16 位（`first_half`），判断低 2 位是否为`0x3`：若为`0x3`，说明是 32 位指令（`epc+4`）；否则是 16 位压缩指令（`epc+2`）。
    - `tf->epc += ...`：最终让`epc`指向“异常指令的下一条有效指令”，这样程序返回后就不会再执行异常指令，避免死循环。

（2）断点异常处理

```
case CAUSE_BREAKPOINT:
            //断点异常处理
            /* LAB3 CHALLENGE3   YOUR CODE :  */
            /*(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("ebreak caught at 0x%08x\n", tf->epc);
            cprintf("Exception type: breakpoint\n");
            {
                uint16_t first_half = *(uint16_t *)(tf->epc);
                tf->epc += ((first_half & 0x3) == 0x3) ? 4 : 2;
            }
            break;
```

我们的目的是响应`ebreak`断点指令，让程序能继续执行后续逻辑。

- **实现思路**：
  - **输出异常类型与地址**：打印“breakpoint”类型，通过`tf->epc`输出`ebreak`指令的地址，明确断点位置。
  - **动态调整`tf->epc`**：与非法指令处理逻辑一致，根据指令长度（16/32 位）调整`epc`，跳过`ebreak`指令，确保程序返回后从下一条有效指令继续执行，而非重复进入断点。

我们的代码支持 RISC-V 的 16 位压缩指令，避免因指令长度固定偏移（如仅`+4`）导致的执行混乱。按照要求，我们实现了通过打印异常类型和地址，快速定位错误指令，降低调试难度。并且，代码可以跳过异常指令，防止程序陷入“异常 - 处理 - 再异常”的死循环，保障系统基本运行。

以下是我们的终端运行结果，包括练习一和challenge3

```
++ setup timer interrupts
[CH3] trigger ebreak (breakpoint) test...
ebreak caught at 0xc02000a8
Exception type: breakpoint
[CH3] returned after breakpoint handler.
[CH3] trigger illegal instruction (mret in S-mode) test...
sbi_emulate_csr_read: hartid0: invalid csr_num=0x302
Illegal instruction caught at 0xc02000c2
Exception type:Illegal instruction
[CH3] returned after illegal-instruction handler.
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

五、实验知识点与 OS 原理对应关系分析

本次实验较为简单，我们做完实验后，分析了实验知识点与原理的对应，并且发现OS原理中有一些重要内容实验中没有涉及。

（1）知识点对应

1. **中断上下文保存 / 恢复**：对应 OS 原理的 “中断上下文切换”，实验用 `SAVE_ALL` / `RESTORE_ALL` 保存寄存器和 `CSR`，是原理中 “硬件状态保存” 的简化版，未涉及进程 PCB 等管理数据。
2. **时钟中断处理**：对应 “时钟中断与系统计时”，实验通过 `clock_set_next_event()` 实现周期性中断和计数，是原理 “进程调度时间基准” 的基础，未涉及进程切换逻辑。
3. **中断分发**：对应 “中断向量表与分发机制”，实验通过 `trap_dispatch()` 按 `cause` 分发给处理函数，模拟原理核心思想，未用硬件向量表，效率较低。
4. **中断开关控制**：对应 “中断屏蔽与原子操作”，实验用 `local_intr_save()` / `local_intr_restore()` 确保操作原子性，是原理机制的直接实现，未涉及硬件原子指令。
5. **异常处理**：对应 “异常处理与程序恢复”，实验调整 `epc` 跳过异常指令，是原理 “程序恢复” 的简化版，未涉及缺页修复等复杂逻辑。

（2）实验中缺失知识点

这一部分是中断，我们发现有一些知识点实验中没有涉及到：

1. **中断优先级与嵌套**：实际中不同中断（如磁盘 I/O 中断、时钟中断）有优先级，高优先级中断可打断低优先级中断处理；实验未设置中断优先级，也未处理中断嵌套场景。
2. **中断共享**：原理中多个外设可共享同一中断号，通过中断处理函数识别具体设备；实验未涉及多外设，无中断共享场景及处理逻辑。
3. **中断下半部（Bottom Half）机制**：原理中为避免中断处理耗时过长，将非紧急逻辑（如数据处理）放到下半部（如 Linux 的软中断、任务队列）执行；实验中断处理逻辑简单，未拆分上下半部，所有操作均在中断上下文完成。