LAB3 实验报告

练习1：完善中断处理

需求概述

- 目标：在 S 模式时钟中断到达时，每累计 100 次中断输出一次 `100 ticks`，累计 10 行后调用 `sbi_shutdown` 关机。
- 相关文件：`kern/trap/trap.c`、`kern/driver/clock.c`、`kern/driver/intr.c`。

核心实现过程

```
// kern/trap/trap.c : interrupt_handler 中的 S-TIMER 分支
case IRQ_S_TIMER:
    // "All bits besides SSIP and USIP in the sip register are
read-only."
    // -- privileged spec1.9.1, 4.1.4, p59
    // 实际上，调用 sbi_set_timer 会清除 STIP，当然你也可以直接清。
    /* LAB3 EXERCISE1   YOUR CODE :  */
    /*(1)设置下次时钟中断- clock_set_next_event()
     *(2)计数器（ticks）加一
     *(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（num）加一
     *(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
     */
    clock_set_next_event();             // (1) 续订下一次时钟中断
    ticks_count++;                      // (2) 心跳+1
    if (ticks_count == TICK_NUM) {      // (3) 满100次
        print_ticks();                  //     打印 "100 ticks"
        ticks_count = 0;                //     清零以便下一轮计数
        print_count++;                  //     已打印次数+1
        if (print_count == 10) {        // (4) 打印到第10行
            sbi_shutdown();             //     请求关机
        }
    }
    break;
```

该分支先调用 `clock_set_next_event()` 续订下一次时钟中断（底层经 `sbi_set_timer(rdtime()+timebase)` 预约下一拍，并清除本次 `STIP`），把续订放在最前面可减少处理开销对下一次触发点的漂移。随后将 `ticks_count++` 记录一次“心跳”；当累计到 `TICK_NUM==100` 时调用 `print_ticks()` 输出一行 `"100 ticks"`，并把 `ticks_count` 清零形成周期；同时 `print_count++` 统计打印次数，累计到 **10 行**后调用 `sbi_shutdown()` 请求 OpenSBI 关机。陷入期间硬件已将 `sstatus.SIE` 清零、返回由 `sret` 按 `SPIE` 自动恢复，因此对两个计数器的更新是**原子**的。要保证该逻辑持续工作，初始化阶段需已设置 `stvec=&__alltraps`、开启 `sie.STIE` 与 `sstatus.SIE`，并在 `clock_init()` 中完成首次 `clock_set_next_event()` 的预约。

中断处理流程分析

1. `clock_init` 调用 `clock_set_next_event`，设置第一次触发时间；同时 `sie` 置位 `STIE`。
2. OpenSBI 到时硬件置位 `sip.STIP`，内核收到 S 模式定时器中断，将 `scause` 最高位置 1 并写入原因码。
3. CPU 跳转至 `stvec=__alltraps`，`SAVE_ALL` 宏在栈上构造 `struct trapframe`。
4. 汇编例程将 `sp` 作为参数传给 `trap`，由 `trap_dispatch` 跳转到 `interrupt_handler`。
5. `IRQ_S_TIMER` 分支执行计数与关机逻辑，随后返回到 `__trapret`，`RESTORE_ALL` 恢复通用寄存器和 `sepc`，`sret` 回到被中断位置。
6. 当打印次数达到 10 次时调用 `sbi_shutdown`，OpenSBI 控制器执行实际关机。

调试与验证

- 在 QEMU/Spike 下执行 `make run`，观察串口输出约每秒出现一行 `100 ticks`，累计 10 行后看见平台退出。若启用 `DEBUG_GRADE`，会在第一次输出后终止于 `panic`，可帮助判定流程是否到位。
- 使用 `cprintf` 在关键路径打印辅助信息（调试阶段），确认 `ticks_count` 重置与 `clock_set_next_event` 调用次数吻合。

```
satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
```

Challenge1：描述与理解中断流程

uCore 中断/异常处理完整链路

1. **异常产生**：硬件检测到事件（如时钟、非法指令），设置 `sepc/stval/scause` 等 CSR。
2. **入口跳转**：根据 `stvec` 模式进入 `__alltraps`。
3. **上下文保存**：`SAVE_ALL` 先暂存当前 `sp` 至 `sscratch`，再按 `struct trapframe` 的字段顺序压栈全部通用寄存器及关键 CSR（`kern/trap/trapentry.S:3-55`）。
4. **转入 C 语言**：`move a0, sp` 将 `trapframe` 指针作为实参，`jal trap` 执行调度。
5. **分发处理**：`trap_dispatch` 通过带符号检查区分中断/异常，并进入 `interrupt_handler` 或 `exception_handler`。
6. **返回路径**：处理完毕回到 `__trapret`，通过 `RESTORE_ALL` 恢复上下文，`sret` 返回原控制流。

问题回答

- `move a0, sp` **的目的**：按照 RISC-V ABI，`a0` 是第一个参数寄存器。这里将当前栈顶（即 `trapframe` 起始地址）传给 `trap`，以便 C 代码读写保存的现场信息。
- `SAVE_ALL` **中寄存器在栈上位置的确定方式**：顺序与 `struct trapframe/struct pushregs` 的字段声明严格对应，且使用统一的 `REGBYTES` 偏移，保证汇编保存顺序与 C 结构体布局匹配。
- **是否必须保存所有寄存器**：为了简化实现和支持内核后续的进程调度、上下文切换，`__alltraps` 统一保存全部通用寄存器与关键 CSR。尽管部分中断理论上只需保存调用约定中“易失”寄存器，统一处理可以避免在异常嵌套、调度切换时产生遗漏，是 uCore 简洁可靠的折衷。

Challenge2：理解上下文切换机制

汇编语句解析

- `csrw sscratch, sp`：将当前 `sp` 写入 `sscratch`。当陷入来自用户态时，硬件会把用户栈指针放在 `sp`，而内核预先在 `sscratch` 中放入自己的内核栈；这条指令交换两者，使得后续 `csrrw s0, sscratch, x0` 能取回“旧栈指针”并将 `sscratch` 清零，便于区分陷入是否来自内核态。
- `csrrw s0, sscratch, x0`：把 `sscratch` 的内容读入 `s0`，同时把 0 写回 `sscratch`。`s0` 立即被保存到 `trapframe` 中，从而保留了原始栈指针供异常恢复或调度使用。

CSR 保存的意义

- 把 `sstatus/sepc/stval/scause` 存进 `trapframe` 的意义在于：**把这次陷入的完整“现场信息”一次性快照下来**，供 C 侧的分发与处理（判断中断/异常类型、打印调试、修复返回地址等）使用，并在返回前能把**真正影响继续执行的状态**恢复回去。具体来说：`sstatus` 记录当下的 S 模式状态（含 SIE/SPIE/SPP 等），用于返回时按原样恢复中断开关与目标特权级；`sepc` 是“要从哪里继续”的指令地址，异常路径下还可能被 handler 改写（例如从 `ecall` 的下一条继续）；`stval`（badvaddr）提供异常的附加上下文，如页故障的出错虚拟地址或非法指令本体；`scause` 则给出陷入的类别与原因码（最高位标识中断/异常，低位是具体原因），驱动 `trap_dispatch/interrupt_handler/exception_handler` 的分流逻辑。
- 之所以 `RESTORE_ALL` **只写回** `sstatus` **与** `sepc`，**而不恢复** `stval/scause`，是因为后两者只是这次陷入的“现场报告”——它们不参与控制返回执行，也会在下一次陷入时被硬件重新填充；相反，能真正决定“怎么、从哪儿回去”的只有 `sepc`（返回 PC）和 `sstatus`（恢复中断开关与返回特权级），因此恢复它们即可，去改写 `stval/scause` 既无必要也无意义。

Challenge3：完善异常中断

目标概述

- 在异常发生时，**打印异常类型与触发地址**，并**安全前移** `sepc`（兼容 16/32 位指令），避免在同一条异常指令上反复陷入形成死循环。
- 提供一段**可重复触发的验证代码**（`ebreak` 与人为构造的非法指令），验证异常处理是否正确。

核心代码与作用说明

1) 非法指令异常和断点异常

```
case CAUSE_ILLEGAL_INSTRUCTION:
    // 非法指令异常处理（Challenge3）
    cprintf("Exception type:Illegal instruction\n");
    cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
    advance_epc(tf); // 智能跳过指令（兼容16位/32位）
    break;
case CAUSE_BREAKPOINT:
    // 断点异常处理（Challenge3）
    cprintf("Exception type: breakpoint\n");
    cprintf("ebreak caught at 0x%08x\n", tf->epc);
    advance_epc(tf); // 智能跳过指令（兼容16位/32位）
    break;
```

这段代码为 **CAUSE_ILLEGAL_INSTRUCTION** 和 **CAUSE_BREAKPOINT** 增加了**打印与前移**逻辑：先输出异常类型与触发地址（`tf->epc`），再调用 `advance_epc(tf)` 按指令低两位判断 **16 位压缩**或 **32 位**指令并**前移** `sepc`，从而避免回到原指令再次触发同一异常导致死循环。

2) 验证代码

```
// kern/init/init.c
int kern_init(void) {
    ...
    // 测试异常处理 - Challenge3
    cprintf("Testing exception handlers...\n");

    // 触发断点异常（Breakpoint）
    cprintf("Testing breakpoint exception:\n");
    asm volatile("ebreak");

    // 触发非法指令异常（Illegal instruction）
    cprintf("Testing illegal instruction exception:\n");
    asm volatile(".word 0xffffffff");  // 明确为 32 位非法指令编码

    cprintf("Exception tests completed!\n");

    while (1) ;
}
```

这段验证代码在**完成中断入口设置、内存管理初始化与时钟预约**后，依次触发 `ebreak`（断点异常）与一条**人为构造的 32 位非法指令**，用于验证 `exception_handler`：应当在控制台看到两条类型化输出，并且**不会反复陷入**（因为 `advance_epc` 已将 `sepc` 前移到下一条指令）。

流程与结果观察

1. 内核完成基础初始化后打印测试提示；
2. 执行 `ebreak` 触发断点异常，控制台输出 `Exception type: breakpoint` 与地址信息；
3. `advance_epc` 使 `sret` 返回到下一条指令；
4. 继续执行非法指令 `.word 0xffffffff`，产生 `CAUSE_ILLEGAL_INSTRUCTION`，输出异常类型与地址；
5. 若后续继续运行，可看到计时器中断输出，最终触发关机。