Lab3 中断与中断处理流程

小组成员：吴昊然 2312321　袁宇菡 2312554　葛佳琦 2313202

实验目的

实验 3 主要讲解的是中断处理机制。操作系统是计算机系统的监管者，必须能对计算机系统状态的突发变化做出反应，这些系统状态可能是程序执行出现异常，或者是突发的外设请求。当计算机系统遇到突发情况时，不得不停止当前的正常工作，应急响应一下，这是需要操作系统来接管，并跳转到对应处理函数进行处理，处理结束后再回到原来的地方继续执行指令。这个过程就是中断处理过程。

本章将学到：

- riscv 的中断相关知识
- 中断前后如何进行上下文环境的保存与恢复
- 处理最简单的断点中断和时钟中断

实验内容

实验 3 主要讲解的是中断处理机制。通过本章的学习，我们了解了 riscv 的中断处理机制、相关寄存器与指令。我们知道在中断前后需要恢复上下文环境，用一个名为中断帧（TrapFrame）的结构体存储了要保存的各寄存器，并用了很大篇幅解释如何通过精巧的汇编代码实现上下文环境保存与恢复机制。最终，我们通过处理断点和时钟中断验证了我们正确实现了中断机制。

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。

1. 首先添加必要的头文件用于关机函数：#include <sbi.h>

2. 再定义全局变量用于打印次数计数器：

```
#define TICK_NUM 100

static size_t num = 0;
```

3. 接着进行时钟中断处理流程：设置下一次中断，调用 clock_set_next_event()确保时钟中断持续发生；计数器递增，使用 clock.h 中声明的全局变量 ticks 记录中断次数；条件输出，每 100 次中断调用 print_ticks()输出信息；关机控制，输出 10 次后调用 sbi_shutdown()关机。

```
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
    // In fact, Call sbi_set_timer will clear STIP, or you can clear it
    // directly.
    // cprintf("Supervisor timer interrupt\n");
     /* LAB3 EXERCISE1   2312321，2313202，2312554 :  */
    /*(1)设置下次时钟中断- clock_set_next_event()*/
```

```
    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();
        num++;
    }
 /*(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*/
    if (num == 10) {
        sbi_shutdown();
    }
    break;
```

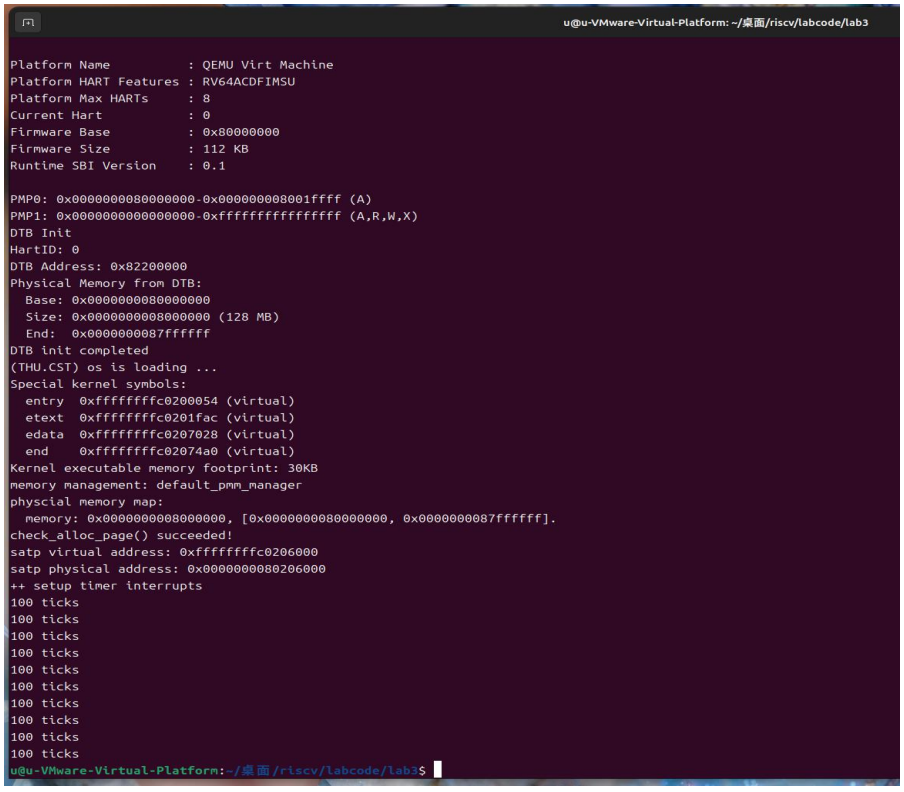
4. 再进行异常处理机制：指令跳过，通过 tf->epc += 4 跳过异常指令；信息输出，提供详细的异常类型和地址信息；流程恢复，确保异常处理后程序能继续执行。

```
case CAUSE_ILLEGAL_INSTRUCTION:
    // 非法指令异常处理
    /* LAB3 CHALLENGE3 2312321，2313202，2312554 */
    /*(1)输出指令异常类型（ Illegal instruction）*/
    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*/
    tf->epc += 4; // 跳过当前非法指令
    break;
```

```c
case CAUSE_BREAKPOINT:
    // 断点异常处理
    /* LAB3 CHALLENGE3 2312321, 2313202, 2312554 */
    /*(1)输出指令异常类型（ breakpoint）*/
    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*/
    tf->epc += 4; // 跳过 ebreak 指令
    break;
```

5. 时钟中断进行特殊处理：中断屏蔽，在处理过程中，通过 sstatus 寄存器控制中断使能；时间基准，使用 timebase 设置中断间隔，确保每秒约 100 次中断；原子操作，关键操作期间禁用中断，保证数据一致性。

经过 make qemu，可以看到每 1 秒会输出一次”100 ticks”，输出 10 行后自动关机：



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。

首先明确，在 RISC-V 中，中断（Interrupt）和异常（Exception）统称为 Trap。它的核心是——当 CPU 执行过程中遇到事件（如外设信号、非法指令等），需要：

1. 保存当前执行状态（上下文 Context）
2. 跳转到统一的 Trap 入口（trapentry.S）
3. 分发到合适的处理函数（trap.c）
4. 执行完毕后恢复上下文，返回原程序

- **uCore 中处理中断异常的流程**

（1）触发 trap

当出现中断（如时钟）或异常（如非法指令）时，硬件自动：

- 保存当前指令地址到 sepc
- 将中断/异常原因写入 scause
- 将错误信息写入 sbadaddr
- 跳转至 stvec 所设地址：__alltraps

stvec 是在 idt_init()中设置的：

```
void idt_init(void) {
       extern void __alltraps(void);
    /* Set sup0 scratch register to 0, indicating to exception vector
       that we are presently executing in the kernel */
    write_csr(sscratch, 0);
    /* Set the exception vector address */
    write_csr(stvec, &__alltraps);
}
```

（2）进入 trapentry.S 的 __alltraps

```
__alltraps:
    SAVE_ALL      # 保存当前上下文
    move  a0, sp      # 把 TrapFrame 地址传给 trap()
    jal trap     # 调用 C 语言的 trap 函数
    # sp should be the same as before "jal trap"
```

- SAVE_ALL：为 TrapFrame 结构分配空间（36×8 字节）；
- 把所有寄存器和 CSR (sstatus, sepc, sbadaddr, scause) 保存到栈；
- 将 sp（此时指向 TrapFrame）传递给 C 函数 trap。

Tips： CSR 是 Control and Status Register 的缩写，中文叫控制与状态寄存器。

（3）C 层处理：trap() 和 trap_dispatch()

trap() 接收 TrapFrame，调用 trap_dispatch(tf)，把 TrapFrame 交给 trap_dispatch()去判断到底是什么类型的 Trap（中断 or 异常）。

```
static inline void trap_dispatch(struct trapframe *tf) {
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
```

- RISC-V 中最高位区分中断/异常：

  scause 最高位 = 1 → (tf->cause < 0) → 中断

  scause 最高位 = 0 → (tf->cause >= 0) → 异常

Tips：在 C 语言里，整型（如 intptr_t）通常使用补码表示。所以，如果我们把 scause（无符号）转成 intptr_t（有符号），那么：

当 scause 最高位 = 1 → 转成有符号数后就变成负数；

当 scause 最高位 = 0 → 转成有符号数后仍是正数。

（4）中断处理：interrupt_handler()

时钟中断处理部分：

```
clock_set_next_event();  /*(1)设置下次时钟中断- clock_set_next_event()*/
            ticks++; /*(2)计数器（ticks）加一*/


      /*(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();
                num++;
            }
```

- 设置下一次时钟事件；
- 计数累加，每 100 次打印一次；
- 打印 10 次后调用 sbi_shutdown() 关机。

（5）异常处理：exception_handler()

以非法指令为例：

```
case CAUSE_ILLEGAL_INSTRUCTION:
            cprintf("Exception type: Illegal instruction\n");
            cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
            /*(3)更新 tf->epc 寄存器*/
            tf->epc += 4; // 跳过当前非法指令
            break;
```

- 输出异常类型和地址；
- tf->epc += 4 让系统跳过出错指令，防止死循环。

（6）trap 返回 ： __trapret → sret

trap() 返回后执行：

```
__trapret:
    RESTORE_ALL
    # return from supervisor call
```

- RESTORE_ALL 从栈（TrapFrame）中依次取出寄存器的原始值，恢复被打断前的 CPU 执行状态。
- sret 从内核态返回到中断前的指令。

sret 是 RISC-V 特权指令 Supervisor Return

执行后，CPU 会：

① 从 CSR sepc 取出中断前的程序计数器；

② 恢复 CSR sstatus 的中断位；

③ 跳回中断发生前的那条指令继续执行。

- **mov a0，sp 的目的**

在 __alltraps 中执行

```
move a0, sp
jal trap
```

- a0 是 RISC-V 调用约定中的第一个函数参数寄存器；
- 此时 sp 指向保存上下文的 TrapFrame；
- 因此该指令把 TrapFrame 地址传给 trap()。

总结起来就是：mov a0, sp 的目的是把保存的上下文结构体地址传入 C 函数 trap()，让 trap()可以访问寄存器状态。

- **SAVE_ALL 中寄存器保存在栈中的位置如何确定？**

对照 trap.h：

```
struct trapframe {
    struct pushregs gpr;
    uintptr_t status;
    uintptr_t epc;
    uintptr_t badvaddr;
```

汇编 trapentry.S 中

```
STORE x1, 1*REGBYTES(sp)
...
STORE s1, 32*REGBYTES(sp)
STORE s2, 33*REGBYTES(sp)
STORE s3, 34*REGBYTES(sp)
```

❖ 保存顺序与 trapframe 结构体成员顺序严格对应。

❖ 这样在 C 层访问 tf->status、tf->epc 等时，能精确映射。

总结：在 SAVE_ALL 宏中，寄存器在栈中的保存位置是根据 C 语言中结构体 struct trapframe 的定义顺序来确定的。

Tips：

在该结构体中，首先定义了一个 struct pushregs，用于保存通用寄存器 x0～x31；随后依次定义了 status、epc、badvaddr 和 cause 四个成员，分别对应 RISC-V 的 CSR 寄存器 sstatus、sepc、sbadaddr 和 scause。

因此，汇编代码在保存寄存器时，按照以下顺序与偏移进行存储：

- x0～x31 对应偏移 0～31 × REGBYTES；
- sstatus 对应偏移 32 × REGBYTES；
- sepc 对应偏移 33 × REGBYTES；
- sbadaddr 对应偏移 34 × REGBYTES；
- scause 对应偏移 35 × REGBYTES。

这种顺序与结构体字段完全一致，确保了 C 语言中通过 tf->status、tf->epc、tf->badvaddr 和 tf->cause 等字段访问时，能够准确读取到汇编保存的寄存器值，从而保证中断上下文保存与恢复的正确性。

REGBYTES 是一个宏（macro）常量，在 RISC-V 的汇编代码中用来表示：每个寄存器占用的字节数，RISC-V 64 位架构下：

每个寄存器（x0～x31、sstatus 等）占 64 位即 8 字节，所以一个寄存器保存到栈中，就占用 8 个字节空间。

- **对于任何中断，__alltraps 中都需要保存所有寄存器吗**

  在理论上，中断发生后只需要保存会被修改的寄存器即可，以减少中断处理的开销。如果中断处理程序只使用了部分寄存器，那么只保存可能被修改的寄存器即可，这样可以减少保存和恢复的开销，提高中断响应效率。

  但在实际实现中，比如我们的 uCore 实验，__alltraps 被设计成所有中断与异常的统一入口，为了简化逻辑并避免遗漏，选择在 SAVE_ALL 宏中保存所有寄存器（x0～x31 及相关 CSR）。操作系统的 Trap 机制是通用框架，保存和恢复要完全对称，否则 Trap 返回时寄存器内容会错乱。

  这种设计保证了中断返回时的上下文完全一致，便于调试和扩展，提高了系统的安全性与通用性，能保证中断处理后系统状态完全恢复。

扩增练习 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？

- **指令 csrw sscratch, sp 和 csrrw s0, sscratch, x0**

1. csrw sscratch, sp

- csrw = Control and Status Register Write
- 把寄存器 sp（当前栈指针）写入控制寄存器 sscratch，即将当前栈指针保存到 sscratch，**暂存当前的栈指针（stack pointer）。**
- 这一步在进入中断之前执行。当 CPU 触发 Trap 时，它会自动切换到一个新的栈空间（通常是内核栈）。为了在中断返回时能恢复原先用户或内核栈，必须先把旧的 sp 保存起来。

所以 sscratch 在这里被当作一个“临时存放点”，存的是 Trap 触发时的旧栈地址。

2. csrrw s0, sscratch, x0

- csrrw = Control and Status Register Read and Write（读并交换）
- 把 sscratch 内容（旧 sp）读回 s0，把 x0（恒为 0）写回 sscratch

目的：把中断发生前保存的旧 sp 取回来，放入 s0 暂存；把 sscratch 清零，标记现在正在内核中（防止嵌套中断混乱）。

总结：这两条指令配合起来，就能保证：Trap 发生时，旧的 sp 被安全保存；

返回时，能恢复原来的栈环境。这两条指令负责在陷入与返回之间安全切换栈环境，是中断嵌套保护机制的关键。

- **为什么 SAVE_ALL 保存了 CSR（stval、scause），但 RESTORE_ALL 不恢复？**

在 SAVE_ALL 里：

```
csrr s1, sstatus
    csrr s2, sepc
    csrr s3, sbadaddr
    csrr s4, scause

    STORE s0, 2*REGBYTES(sp)
```

但在 RESTORE_ALL 里，只恢复了以下内容，而没有恢复 stval 和 scause。

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

❖ 原因在于：

stval 和 scause 是只读状态寄存器，由硬件维护，它们反映的是这次 Trap 的信息：

stval / sbadaddr：触发异常的地址；

scause：异常或中断的原因。

这些寄存器的内容是由硬件在 Trap 发生时自动写入的。软件不需要、也不能去恢复它们，因为在下一次中断时硬件会重新写入，恢复它们没有意义。它们是由硬件自动维护的状态寄存器，不需要人工恢复。

保存它们的目的仅是便于软件读取、分析异常类型和触发位置，而非恢复硬件状态。因此，保存这些 CSR 的意义在于为内核提供中断信息分析能力，而不是为了在返回时复原它们。

❖ 那为什么还要保存（STORE）它们呢？

因为虽然不恢复，但我们需要在 C 层分析这些值，在 C 函数 trap(struct trapframe *tf)中，会看到 tf->cause、tf->badvaddr 等字段，就是从保存下来的 CSR 值读取的。我们保存它们是为了让 C 层能够打印、调试、分析异常原因，而不是为了在返回时恢复。

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”。

要验证在 *trap.c* 中编写的异常处理逻辑是否正常工作，我们要在*/kern/init/init.c* 中添加触发非法指令和断点异常的测试代码。目标在内核初始化完成后，主动触发一条非法指令和一条断点异常，然后由 *trap.c* 的 *exception_handler()*捕获并输出。

所以在 *intr_enable()*即打开中断后插入测试代码，

```
//测试非法指令和断点异常
    cprintf("\nNow testing exception handling...\n");

    // 触发非法指令异常（非法编码）
    asm volatile(".word 0xFFFFFFFF");

    // 触发断点异常（ebreak 指令）
    asm volatile("ebreak");

    cprintf("If you see this message, exceptions were handled correctly.\n");
```

之后执行 *make qemu* 进行运行测试，得到运行日志：

```
Now testing exception handling...
Exception type: Illegal instruction
```

Illegal *instruction caught at 0xc02000a8*
Exception *type: breakpoint*
ebreak *caught at 0xc02000ac*
Exception *type: Illegal instruction*
Illegal *instruction caught at 0xc02000b0*

可以看到成功触发了非法指令异常，内核成功捕获异常并打印异常发生地址 *0xc02000a8*；成功触发 *ebreak* 指令，同样捕获到断点异常，地址紧接在前一条前面；之后又一次非法指令异常触发，可能是编译器在跳过 *ebreak* 后继续执行到某个填充位置。

以上的分析说明了：

1. 异常类型识别正确

- Illegal instruction → *CAUSE_ILLEGAL_INSTRUCTION*
- Breakpoint → *CAUSE_BREAKPOINT*

2. 捕获函数路径正确

__alltraps → trap() → trap_dispatch() → exception_handler()
说明 *stvec*、*sscratch* 等 *CSR* 配置正确。

3. *EPC* 打印正确

每次输出的 *0xc02000XX* 地址就是异常发生的 PC，说明 *tf->epc* 正常保存与恢复。

4. *epc* 跳过机制生效

程序能继续执行到后续异常而非卡死，证明 *tf->epc += 4;* 正常起效。