

lab3:中断于中断处理流程

小组成员：

吴禹騫-2311272

谢小珂-2310422

杜泽琦-2313508

lab3:中断于中断处理流程

实验目的

实验内容

练习

练习1：完善中断处理（需要编程）

trap.c

实现过程及定时器中断处理流程：

扩展练习 Challenge1：描述与理解中断流程

ucore 中断/异常处理完整流程

move a0, sp 的目的是什么？

SAVE_ALL 中寄存器保存在栈中的位置是如何确定的？

对于任何中断，__alltraps 中都需要保存所有寄存器吗？请说明理由

扩展练习 Challenge2：理解上下文切换机制

trapentry.S代码

csrwr sscratch, sp; csrwr s0, sscratch, x0实现了什么操作，目的是什么？

指令含义（逐条解释）

实际效果（放回上下文理解）

为什么这样设计？

总结

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？

为什么保存 CSR？

为什么 restore 不恢复 `scause / stval`？

更深层的理解

既然不恢复 CSR，为什么还 STORE 到 trapframe？

总结

扩展练习Challenge3：完善异常中断

1. 补全代码

2. 测试

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。

本章你将学到：

- 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

在一般OS中进行中断处理支持的方法：

- 编写相应的中断处理代码
- 在启动中正确设置控制寄存器
- CPU捕获异常
- 控制转交给相应中断处理代码进行处理
- 返回正在运行的程序

练习

对实验报告的要求：

- 基于markdown格式来完成，以文本方式为主
- 填写各个基本练习中要求完成的报告内容
- 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）
- 列出你认为OS原理中很重要，但在实验中没有对应上的知识点
- 从oslab网站上取得实验代码后，进入目录labcodes/lab3，完成实验要求的各个练习。在实验报告中回答所有练习中提出的问题。在目录labcodes/lab3下存放实验报告，推荐用**markdown**格式。每个小组建一个gitee或者github仓库，对于lab3中编程任务，完成编写之后，再通过git push命令把代码和报告上传到仓库。最后请一定提前或按时提交到git网站。

注意有“LAB3”的注释，代码中所有需要完成的地方（challenge除外）都有“LAB3”和“YOUR CODE”的注释，请在提交时特别注意保持注释，并将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

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秒会输出一行“100 ticks”，输出10行。

trap.c

在文件顶部 声明一个打印计数器：

```
cstatic uint32_t print_cnt = 0;
```

```
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 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
    */
    /* LAB3 EXERCISE1 完整实现 */
    clock_set_next_event();           // (1) 设置下一次时钟中断
    ticks++;                          // (2) 计数器 +1

    if (ticks % TICK_NUM == 0) {      // (3) 每 100 次打印一次
        print_ticks();
        print_cnt++;
        if (print_cnt >= 10) {        // (4) 打印 10 行后关机
            cprintf("System shutdown after 10 prints.\n");
            sbi_shutdown();
        }
    }
    break;

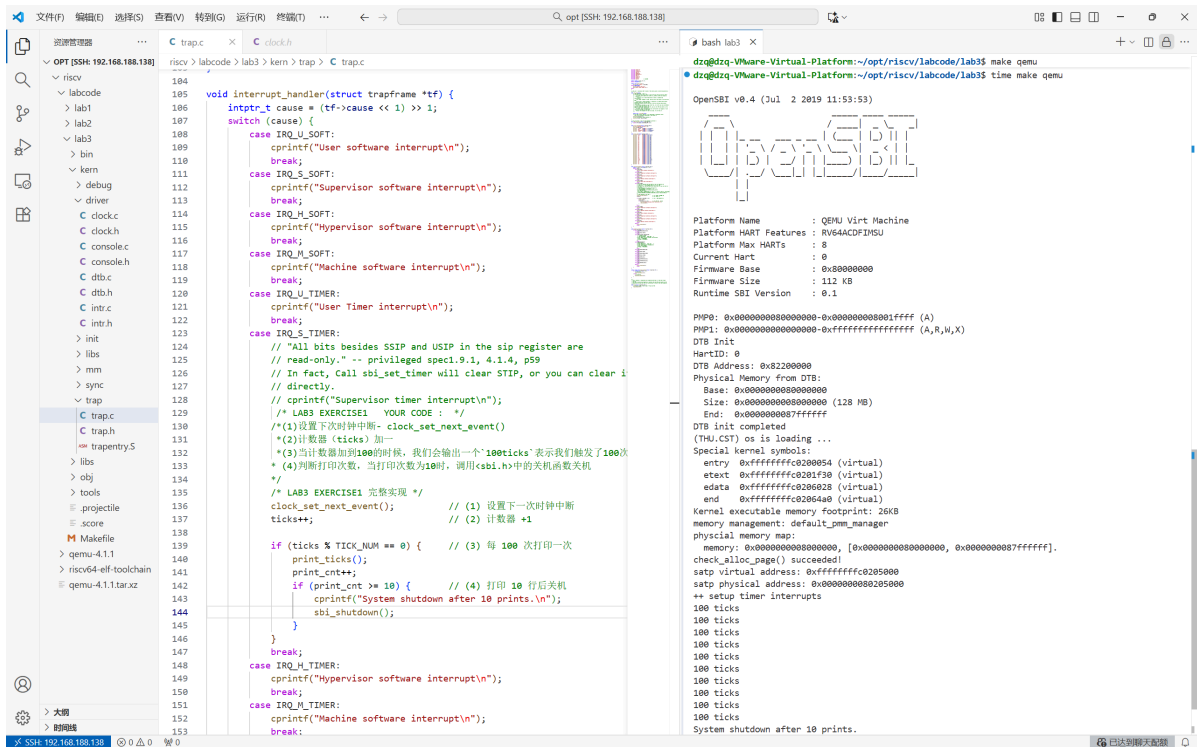
```

```

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("Exception: Illegal instruction\n");
    cprintf(" Bad instruction address: 0x%08x\n", tf->epc);
    tf->epc += 4; //跳过非法指令
    break;

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("Exception: Breakpoint\n");
    cprintf(" Bad instruction address: 0x%08x\n", tf->epc);
    tf->epc += 4; //跳过断点指令
    break;

```



time make qemu执行时间

```
real    0m10.223s
user    0m10.223s
sys     0m0.042s
```

实现过程及定时器中断处理流程：

1. 初始化：

`clock_init()` → 使能时钟中断 → 调用 `clock_set_next_event()` 设置 10ms 后第一次中断。

2. 触发：

10ms 到期 → OpenSBI 触发 **Supervisor Timer Interrupt**。

3. 入口：

CPU 跳到 `trapentry.S` 的 `__alltraps` → 保存寄存器 → 调用 `trap(tf)`。

4. 处理：

- `trap()` → `interrupt_handler()` → 匹配 `IRQ_S_TIMER`：
 - `clock_set_next_event()`：设置下一次 10ms 中断
 - `++ticks`：计数
 - 每 `ticks % 100 == 0`：打印 "100 ticks"（每秒一次）

5. 返回：

恢复寄存器 → `sret` → 继续原程序。

循环往复：每 10ms 中断一次，每 100 次（1 秒）打印一行。

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。

ucore 中断/异常处理完整流程

中断/异常产生 硬件(如时钟)或软件触发 → CPU 设置 scause(中断或异常的具体原因)、sepc(被中断指令的虚拟地址)、stval(与异常相关的附加信息) → 跳转至 stvec 指向的 __alltraps。

__alltraps 执行 SAVE_ALL

- csrw sscratch, sp: 保存原 sp 到 sscratch
- addi sp, sp, -36*8: 为 trapframe 分配 36*8=288 字节栈空间
- STORE x0~x31 (跳过 x2): 保存 31 个通用寄存器 (x2/sp 单独处理)
- 读取 CSR: sstatus, sepc, sbadaddr, scause → 存入栈中对应偏移
- 栈顶 sp 指向一个完整的 struct trapframe 结构体, 保存了中断发生前的全部 CPU 上下文

接下来执行 move a0, sp, 将 trapframe 的起始地址放入 a0 寄存器, 然后通过 jal trap 调用 trap.c 中的 C 函数 trap()。trap() 接收该指针后, 调用 trap_dispatch() 判断 tf->cause 的最高位: 若为 1 (负数), 表示中断, 进入 interrupt_handler(); 否则为异常, 进入 exception_handler(), 进行具体处理(如时钟中断计数、非法指令跳过等)。

处理完成后, trap() 返回到 trapentry.S 的 __trapret 标签处。执行 RESTORE_ALL 宏, 按相反顺序从 trapframe 中恢复 sstatus、sepc 和所有通用寄存器 (sp 最后恢复), 确保 CPU 状态完全还原。最后执行 sret 特权指令: 根据 sstatus.SPP 的值 (处理中已设为 0) 切换回用户态 (U-mode), 并跳转到 sepc 指向的指令继续执行用户程序。至此, 中断/异常处理流程结束, 用户程序在中断前被打断的位置 (或跳过异常指令后) 继续运行。(实验指导书: 在执行 sret 之前, 需要完成一些准备工作。首先, 从 trapframe 中恢复用户程序的寄存器值 (这由 RESTORE_ALL 宏完成), 使得用户程序能够继续运行。接着, 根据中断或者异常的类型重新设置 sepc, 确保程序能够从正确的地址继续执行。对于系统调用, 这通常是 ecall 指令的下一条指令地址 (即 sepc + 4); 对于中断, 这是被中断打断的指令地址 (即 sepc); 对于进程切换, 这是新进程的起始地址。然后, 将 sstatus.SPP 设置为 0, 表示要返回到 U 模式。

当准备工作完成后, 会执行 sret 指令, 根据 sstatus.SPP 的值 (此时为 0) 切换回 U 模式。随后, 恢复中断使能状态, 将 sstatus.SIE 恢复为 sstatus.SPIE 的值。由于在 U 模式下总是使能中断, 因此中断会重新开启。接着, 更新 sstatus, 将 sstatus.SPIE 设置为 1, sstatus.SPP 设置为 0, 为下一次中断做准备。最后, 将 sepc 的值赋给 pc, 并跳转回用户程序 (sepc 指向的地址) 继续执行。此时, 系统已经安全地从 S 模式返回到 U 模式, 用户程序继续执行。)

move a0, sp 的目的是什么?

将当前栈顶 (即 trapframe 的起始地址) 作为参数传给 trap() 函数

- SAVE_ALL 完成后, sp 指向栈中刚构造好的 struct trapframe
- move a0, sp 把这个地址放入 a0
- RISC-V 调用约定: **a0 是函数第一个参数**
- 因此 trap(struct trapframe *tf) 能正确接收上下文

asm

```
move a0, sp    # a0 = &trapframe
jal trap       # 调用 C 函数 trap()
```

SAVE_ALL 中寄存器保存在栈中的位置是如何确定的？

由 struct trapframe 的内存布局 + 固定偏移决定

```
struct trapframe {
    uint64_t gpr[32];    // x0~x31 → 偏移 0~31*8
    uint64_t status;     // sstatus → 偏移 32*8
    uint64_t epc;        // sepc → 偏移 33*8
    uint64_t badvaddr;   // stval → 偏移 34*8
    uint64_t cause;      // scause → 偏移 35*8
};
```

- SAVE_ALL使用类似：

```
STORE x10, 10*REGBYTES(sp)    # x10 存入 sp + 80
```

- 所有偏移 **编译时固定**，由结构体定义和 REGBYTES=8 决定
- RESTORE_ALL 按相同偏移恢复

对于任何中断，__alltraps 中都需要保存所有寄存器吗？请说明理由

是的，__alltraps 中必须保存全部 32 个通用寄存器，理由如下：

首先，中断具有完全的透明性，它可能在用户程序执行的任意时刻发生，此时任意一个通用寄存器都可能正在使用。如果只保存部分寄存器，就可能破坏用户程序的运行状态，导致逻辑错误。其次，trap() 是一个 C 函数，按照 RISC-V 调用约定，它会使用 a0_{a7}、t0_{t6} 等调用者保存寄存器进行计算。如果不先保存这些寄存器的原始值，C 函数执行后将覆盖用户态上下文，恢复时将导致程序崩溃。第三，所有中断和异常（包括时钟中断、系统调用、非法指令等）都共用同一个入口 alltraps，系统无法提前预知本次中断发生时哪些寄存器是“安全的”，因此只能采取最保守策略，全部保存。第四，从系统设计角度看，ucore 未来需要支持进程调度和上下文切换，完整保存所有通用寄存器是实现多任务切换的必要前提。最后，根据 RISC-V 特权架构规范，sret 指令仅负责恢复 sepc 和 sstatus，并不自动恢复通用寄存器，因此保存和恢复通用寄存器的任务必须完全由软件（即 alltraps 和 __trapret）负责。综上，保存全部寄存器是保证中断处理正确性、透明性、通用性和系统可扩展性的必要措施。

扩展练习 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？

trapentry.S代码

```
#include <riscv.h>

    .macro SAVE_ALL    #汇编宏 SAVE_ALL，用来保存所有寄存器到栈顶（实际上把一个trapFrame
                        结构体放到了栈顶）

        csrw sscratch, sp#保存原先的栈顶指针到sscratch
```

```
addi sp, sp, -36 * REGBYTES
```

#REGBYTES是riscv.h定义的常量，表示一个寄存器占据几个字节

#让栈顶指针向低地址空间延伸 36个寄存器的空间，可以放下一个trapFrame结构体。

#除了32个通用寄存器，我们还要保存4个和中断有关的CSR

#依次保存32个通用寄存器。但栈顶指针需要特殊处理。

#因为我们想在trapFrame里保存分配36个REGBYTES之前的sp

#也就是保存之前写到sscratch里的sp的值

```
STORE x0, 0*REGBYTES(sp)
STORE x1, 1*REGBYTES(sp)
STORE x3, 3*REGBYTES(sp)
STORE x4, 4*REGBYTES(sp)
STORE x5, 5*REGBYTES(sp)
STORE x6, 6*REGBYTES(sp)
STORE x7, 7*REGBYTES(sp)
STORE x8, 8*REGBYTES(sp)
STORE x9, 9*REGBYTES(sp)
STORE x10, 10*REGBYTES(sp)
STORE x11, 11*REGBYTES(sp)
STORE x12, 12*REGBYTES(sp)
STORE x13, 13*REGBYTES(sp)
STORE x14, 14*REGBYTES(sp)
STORE x15, 15*REGBYTES(sp)
STORE x16, 16*REGBYTES(sp)
STORE x17, 17*REGBYTES(sp)
STORE x18, 18*REGBYTES(sp)
STORE x19, 19*REGBYTES(sp)
STORE x20, 20*REGBYTES(sp)
STORE x21, 21*REGBYTES(sp)
STORE x22, 22*REGBYTES(sp)
STORE x23, 23*REGBYTES(sp)
STORE x24, 24*REGBYTES(sp)
STORE x25, 25*REGBYTES(sp)
STORE x26, 26*REGBYTES(sp)
STORE x27, 27*REGBYTES(sp)
STORE x28, 28*REGBYTES(sp)
STORE x29, 29*REGBYTES(sp)
STORE x30, 30*REGBYTES(sp)
STORE x31, 31*REGBYTES(sp)
```

RISCv不能直接从CSR写到内存，需要csrr把CSR读取到通用寄存器，再从通用寄存器STORE到内存

```
csrrw s0, sscratch, x0
csrr s1, sstatus
csrr s2, sepc
csrr s3, sbadaddr
csrr s4, scause
```

```
STORE s0, 2*REGBYTES(sp)
STORE s1, 32*REGBYTES(sp)
STORE s2, 33*REGBYTES(sp)
STORE s3, 34*REGBYTES(sp)
STORE s4, 35*REGBYTES(sp)
.endm #汇编宏定义结束
```

```
.macro RESTORE_ALL
```

#恢复上下文的汇编宏，恢复的顺序和当时保存的顺序反过来，先加载两个CSR，再加载通用寄存器

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



```

    LOAD s2, 33*REGBYTES(sp)
# 注意之前保存的几个CSR并不都需要恢复
    csrw sstatus, s1
    csrw sepc, s2

# 恢复sp之外的通用寄存器，这时候还需要根据sp来确定其他寄存器数值保存的位置
    LOAD x1, 1*REGBYTES(sp)
    LOAD x3, 3*REGBYTES(sp)
    LOAD x4, 4*REGBYTES(sp)
    LOAD x5, 5*REGBYTES(sp)
    LOAD x6, 6*REGBYTES(sp)
    LOAD x7, 7*REGBYTES(sp)
    LOAD x8, 8*REGBYTES(sp)
    LOAD x9, 9*REGBYTES(sp)
    LOAD x10, 10*REGBYTES(sp)
    LOAD x11, 11*REGBYTES(sp)
    LOAD x12, 12*REGBYTES(sp)
    LOAD x13, 13*REGBYTES(sp)
    LOAD x14, 14*REGBYTES(sp)
    LOAD x15, 15*REGBYTES(sp)
    LOAD x16, 16*REGBYTES(sp)
    LOAD x17, 17*REGBYTES(sp)
    LOAD x18, 18*REGBYTES(sp)
    LOAD x19, 19*REGBYTES(sp)
    LOAD x20, 20*REGBYTES(sp)
    LOAD x21, 21*REGBYTES(sp)
    LOAD x22, 22*REGBYTES(sp)
    LOAD x23, 23*REGBYTES(sp)
    LOAD x24, 24*REGBYTES(sp)
    LOAD x25, 25*REGBYTES(sp)
    LOAD x26, 26*REGBYTES(sp)
    LOAD x27, 27*REGBYTES(sp)
    LOAD x28, 28*REGBYTES(sp)
    LOAD x29, 29*REGBYTES(sp)
    LOAD x30, 30*REGBYTES(sp)
    LOAD x31, 31*REGBYTES(sp)
# 最后恢复sp
    LOAD x2, 2*REGBYTES(sp)
    .endm
#真正的中断入口点
    .globl __alltraps
    .align(2) #中断入口点 __alltraps必须四字节对齐
__alltraps:
    SAVE_ALL#保存上下文

    move a0, sp#传递参数
    #按照RISCV calling convention, a0寄存器传递参数给接下来调用的函数trap。
    #trap是trap.c里面的一个C语言函数，也就是我们的中断处理程序
    jal trap
    #trap函数指向完之后，会回到这里向下继续执行__trapret里面的内容，RESTORE_ALL,sret

    .globl __trapret
__trapret:
    RESTORE_ALL
    # return from supervisor call
    sret

```


trapentry.S 的执行流程：

1. 作为唯一的陷入入口 (trap vector)

在内核初始化里，stvec 被设置为 __alltraps，所以无论是中断还是异常，一发生就会跳到 __alltraps 执行，这使它成为 S 模式下的统一入口。

2. 建立 trapframe：完整保存上下文

入口首先执行 SAVE_ALL 宏：

- 把当前 sp 暂存进 sscratch，为区分内核/用户来源与后续恢复做准备；
- 在栈上为一个 trapframe 预留空间（包含 32 个通用寄存器 + 若干与 trap 相关的 CSR 槽位）；
- 逐个 STORE 所有通用寄存器到栈中；
- 用 csrr 读出 sstatus / sepc / stval(badvaddr) / scause 并写入 trapframe 对应位置。
经过该步骤，当时机状态被“定格”为一个 C 语言可读取的结构体 (struct trapframe)，为后续 C 层判断类型/原因/返回点提供依据。

3. 把 trapframe 交给 C 处理逻辑

move a0, sp 把当前栈顶（正对着 trapframe）作为第 1 参数传给 C 函数 trap()，随后 jal trap 进入 C 层；C 层会依据 tf->cause/epc/status/stval 分发到中断或异常处理，并可按需修改 tf->epc（如跳过触发异常的指令），或设置下一次时钟中断等。这里的关键思想：汇编只做“保存与封装”，决策留给 C。

4. 从 trap() 返回后执行“精确恢复”并返回原流

__trapret 调用 RESTORE_ALL：

- 仅恢复 sstatus 与 sepc（影响返回特权级与返回地址），以及所有通用寄存器；
- 最后 sret，按 sepc 返回到被打断/出错的现场（或 C 层改好的新地址），并按 sstatus 恢复中断使能与特权级。
注意：不会恢复 scause / stval——它们是“诊断信息”，给 C 用来判断与打印，恢复无意义。

5. 与内核其他模块的协作关系

- 初始化阶段由 idt_init() 将 sscratch=0 并把 stvec 指向 __alltraps，完成“接管陷入”的最后一环；
- 时钟/外设等引发的 trap 统一走 trapentry.S → trap() 的路径，C 层里再调用具体的时钟设置、计数与关机逻辑等。

总结：

trapentry.S 是 S 模式 trap 的“薄包装层”——对硬件现场做“原子快照” (SAVE_ALL)，把它打包成 trapframe 交给 C 处理，返回时只恢复“会影响继续执行的状态” (ssstatus / sepc + GPR)，最后 sret 精确回到正确的指令地址继续跑。

csrw sscratch, sp; csrrw s0, sscratch, x0实现了什么操作，目的是什么？

回答：这两条汇编指令在中断入口 trapentry.S 中的作用，就是为了正确保存“进入中断前的原始 sp（栈指针）”到 trapframe 结构里。

指令含义（逐条解释）

1. `csrw sscratch, sp`
 - `csrw` = *write CSR register*（向 CSR 寄存器写值）
 - 作用：把当前 CPU 正在使用的 `sp` 写入 `sscratch` 寄存器

简单理解：

“先把当前的 `sp` 暂时藏到 `sscratch` 里。”

在中断发生时，CPU 可能来自 **用户态**，也可能来自 **内核态**。用户态的中断会切换到 **内核栈**，但我们仍然需要记住中断发生前的用户态 `sp`，以便恢复。因此，`sscratch` 就起到 **临时存放原始 `sp` 的保险柜** 的作用。

1. `csrrw s0, sscratch, x0`
 - `csrrw` = *atomic read & write CSR register*（读-改-写 CSR）
 - 作用：
 - 将 `sscratch` 里的值读出到 `s0`
 - 同时将 `x0`（恒为0）写入 `sscratch`

简单理解：

“从 `sscratch` 取出先前的 `sp` 保存到 `s0`，并把 `sscratch` 清零。”

实际效果（放回上下文理解）

结合这两条指令：

步骤	操作	目的
<code>csrw sscratch, sp</code>	保存原始 <code>sp</code> 到 <code>sscratch</code>	不丢失用户态原始栈
<code>csrrw s0, sscratch, x0</code>	<code>s0</code> ← 原来的 <code>sp</code> ; <code>sscratch</code> ← 0	从 <code>sscratch</code> 取回 <code>sp</code> 放到 <code>trapframe</code> , 并清空标识

`s0` 随后被按照 `trapframe` 的位置存入内存（见 `STORE s0, 2*REGBYTES(sp)`）
`sscratch = 0` 表示当前处理中断发生在内核态（`trap.c` 中判断是否 `trap in kernel`）

为什么这样设计？

因为中断进入时，寄存器和栈会发生变化。如果是 **用户态** → **内核态**：

- 内核要切换到 *自己的内核栈* 来处理中断
- 如果不提前保存用户的 `sp`，就**无法恢复回用户态**

所以它必须这样：

进入中断 → 保存原始 `sp` → 切换内核栈 → 处理 → 恢复 `sp` 返回用户态

总结

`csrw sscratch, sp` 和 `csrrw s0, sscratch, x0` 的组合，就是为了 **安全保存 trap 前的栈指针**，并在形成 **trapframe** 时写入正确位置，确保中断处理后能够恢复执行。

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？

回答：SAVE_ALL 把 `stval / scause / status / epc` 这些 CSR 保存进 `trapframe` 是 **为了让 C 语言的 trap() 能读到它们用于中断 / 异常处理**，而不是为了在 `RESTORE_ALL` 时恢复它们。

也就是说：这些 CSR 被保存下来供软件“使用”，不是用来“恢复”。

为什么保存 CSR？

看 `trap.h`，`trapframe` 的结构体包含：

```
struct trapframe {
    struct pushregs gpr;
    uintptr_t status; // sstatus
    uintptr_t epc;    // sepc
    uintptr_t badvaddr; // stval
    uintptr_t cause;  // scause
};
```

这些内容 **会被 trap() / exception_handler() / interrupt_handler() 用来判断 trap 类型、出错地址、恢复点等逻辑**。

例如：

- `scause`：判断是中断还是异常
- `sepc`：决定 trap 返回到哪里
- `stval`：非法访问的虚拟地址
- `sstatus`：记录 trap 前 CPU 状态

保存的目的是：**把 trap 当时的机器状态传给 C 语言层**。

而不是为了“恢复”这些 CSR 到 trap 前。

证明：trap.c 中读取 `tf->cause`、`tf->epc`、`tf->status` 用于分发处理中断或异常：

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

为什么 restore 不恢复 `scause` / `stval`?

因为：

在 trap 返回 (`sret`) 时，只需要恢复：

- 通用寄存器
- `sstatus`
- `sepc`

这三个决定了：

1. CPU 状态是否允许中断 (`sstatus`)
2. 从哪里继续执行 (`sepc`)
3. 用户程序继续执行前的通用寄存器内容

但 `scause`、`stval` 是只读诊断寄存器，用于报告 trap 原因，它们不影响 CPU 下一步执行。

- trap 结束后，CSR 的内容已经“过去了”，不会影响继续执行。
- 恢复它们没有意义，也不允许直接恢复（有的 CSR 只读）。

特别是 `scause` 与 `stval`：

它们的存在目的 = 报告错误，不是供恢复。

更深层的理解

中断处理的流程是：

```
SAVE_ALL → 进入 trap() (C 部分利用 trapframe) → RESTORE_ALL → sret 返回
```

`trap()` 在处理中断或异常时，会自己决定是否修改 `epc` / `status`：

例如异常处理中会：

```
tf->epc += 4;    // 跳过 ecall 或非法指令
```

而最终：

```
csrw sepc, s2      // 只恢复 sepc
csrw sstatus, s1   // 只恢复 sstatus
sret               // 返回原先执行的位置
```

因为 `sepc` + `sstatus` 就足够使 CPU 恢复执行路径和 CPU 状态。

既然不恢复 CSR，为什么还 STORE 到 trapframe?

为了实现：

1. `trap()` 可以知道是什么 trap
2. 异常处理时可以用 `stval` 输出错误地址
3. 系统调用需要知道异常发生在哪条指令

4. 在用户态 trap 时更容易调试/打印信息

不保存就无法让 C 代码知道 trap 的来源。

总结

`SAVE_ALL` 保存所有寄存器（包括 CSR）是为了把 trap 当时完整 CPU 状态封装成 trapframe, 让 C 层 trap() 读取和处理;

`RESTORE_ALL` 只恢复 通用寄存器 + `sepc` + `sstatus`, 因为 trap 返回只依赖执行地址和状态;

`scause` / `stval` 是只读诊断寄存器, 不需要恢复。

扩展练习Challenge3: 完善异常中断

1. 补全代码

根据提示, 编写异常中断处理代码:

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_MISALIGNED_FETCH:
            break;
        case CAUSE_FAULT_FETCH:
            break;
        case CAUSE_ILLEGAL_INSTRUCTION:
            // 非法指令异常处理
            /* LAB3 CHALLENGE3 2311272 : */
            /*(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
            */
            printf("Exception type: Illegal instruction\n");
            printf("Illegal instruction caught at 0x%08x\n", tf->epc);
            tf->epc += 4; // 更新 epc 寄存器, 跳过当前非法指令
            break;
        case CAUSE_BREAKPOINT:
            //断点异常处理
            /* LAB3 CHALLENGE3 2311272 : */
            /*(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
            */
            printf("Exception type: breakpoint\n");
            printf("ebreak caught at 0x%08x\n", tf->epc);
            tf->epc += 4; // 更新 epc 寄存器, 跳过 ebreak 指令
            break;
        ...
    }
}
```

2. 测试

编写测试函数 `trap_test()`:

```
void trap_test(void) {
    cprintf("\n===== Test Begin =====\n");

    // 测试非法指令
    cprintf("Testing illegal instruction:\n");
    asm volatile(".word 0x00000000"); // 非法指令

    cprintf("Testing breakpoint:\n");
    asm volatile("ebreak"); // 断点指令

    // 添加一个明确的跳转或 NOP，确保指令流不会意外地再次触发异常
    asm volatile("nop");
    cprintf("===== Tests End =====\n");
}
```

在 `kern_init` 中调用测试函数，结果如下：

```
===== Test Begin =====
Testing illegal instruction:
Exception type: Illegal instruction
Illegal instruction caught at 0xc0200b5c
Testing breakpoint:
Exception type: breakpoint
ebreak caught at 0xc0200b6c
===== Tests End =====
```

使用 `riscv64-unknown-elf-objdump -d bin/kernel > kernel.asm` 指令对内核进行反汇编，在生成的 `kernel.asm` 文件中查找 `trap_test` 函数的标签：

```
ffffffffc0200b40 <trap_test>:
...
ffffffffc0200b58: d82ff0ef      jal    ffffffffc02000da <printf>
ffffffffc0200b5c: 00000000      .word  0x00000000
ffffffffc0200b60: 00002517      auipc  a0,0x2
ffffffffc0200b64: d3050513      addi   a0,a0,-720 # ffffffffc0202890
<etext+0x908>
ffffffffc0200b68: d72ff0ef      jal    ffffffffc02000da <printf>
ffffffffc0200b6c: 9002          ebreak
ffffffffc0200b6e: 0001          nop
ffffffffc0200b70: 60a2          ld     ra,8(sp)
...
```

可以看到我添加的非法指令地址为 `0xc0200b5c`，断点指令地址为 `0xc0200b6c`，测试结果正确无误。

注：在 RISC-V 架构中，`sbadaddr` CSR（控制和状态寄存器）已经被重命名为 `stval`。需要在 `trapentry.S` 中将 `csrr s3, sbadaddr` 改为 `csrr s3, stval`