OSLab3报告

操作系统实验三

学号：2313501 姓名：杨楠欣 ;学号：2213230 姓名：向宇涵； 学号：2311366 姓名：邵莫涵

练习1：完善中断处理

1. trap.c 原始代码分析

1.1 print_ticks()

代码块

```
#define TICK_NUM 100
static void print_ticks() {
    cprintf("%d ticks\n", TICK_NUM);
#ifdef DEBUG_GRADE
    cprintf("End of Test.\n");
    panic("EOT: kernel seems ok.");
#endif
}
```

这个函数就是，到了一定次数的时钟中断，就打印 “xxx ticks”。后面在时钟中断里会去调用它输出结果。

如果编译时开了 `DEBUG_GRADE`，函数还会 `panic` 告诉自动判分器内核 OK，可以结束。

1.2 idt_init()

这个函数负责初始化中断、异常入口

代码块

```
void idt_init(void) {
    extern void __alltraps(void);
    write_csr(sscratch, 0);
    write_csr(stvec, &__alltraps);
}
```

`extern` 一句是声明，函数实际上位于 `trapentry.S` 文件中。 `trapentry.S` 是一个RISC-V 汇编源文件，所有的中断和异常都会跳转到 `__alltraps` 这个地址上面来执行。所以，

`__alltraps` 实际上就是所有traps（陷入）的入口。

CSR是控制与状态寄存器(Control and Status)，用来保存中断、异常的信息，管理特权级（M/S/U）之间的切换。只能通过专门的汇编指令访问，例如 `csrr`（读 CSR）和 `csrw`（写 CSR）。在第3行中，sscratch是专门给 **S 模式**的寄存器，保存一个 S 模式 trap 处理程序使用的临时值，没有固定作用，内核规定干什么。在程序里，是用来判断“当前 trap 是从用户态陷进来的，还是本来就在内核态”。这里把sscratch写为0，就是说明现在本来就在内核态。

第4行是核心，stvec是S 模式 trap 的入口地址寄存器，只要 S 模式发生了 trap（中断/异常），CPU 就跳到 `stvec` 里写的那个地址去执行。这里把刚刚声明的那个汇编入口的地址，塞进了 `stvec`，也就规定了trap的入口地址。

1.3 trap_in_kernel()

函数的作用是判断这次 trap（中断/异常）是不是发生在内核态S

代码块

```
bool trap_in_kernel(struct trapframe *tf) {
    return (tf->status & SSTATUS_SPP) != 0;
}
```

trapframe是陷入帧，保存了 CPU 在进入 trap 时的**全部寄存器状态**。当 trap 发生时，汇编入口（`__alltraps`）会：把所有通用寄存器压入栈；把一些特殊 CSR（`sstatus` 等）也保存；最后形成一个结构体—— `struct trapframe`。

tf->status对应RISC-V 的 `sstatus` 寄存器，保存trap发生时的 **sstatus** 寄存器的值。

`SPP` 是 `sstatus` 寄存器的一个比特位，当 trap 发生时，CPU 自动把 trap 前的特权级写进这一位（SPP是硬件写入，sscratch是OS软件写）。0是用户态，1是内核态。SSTATUS_SPP是SPP位的掩码，只有 SPP 这一位是 1，其他全是 0。

所以，返回值是在判断SPP位是否为0，不为0返回True就在内核态。否则就在用户态。

1.4 打印 trapframe / 寄存器

这两段就是纯打印，方便调试。

代码块

```
void print_trapframe(struct trapframe *tf) {
    cprintf("trapframe at %p\n", tf);
    print_regs(&tf->gpr);
    cprintf("  status   0x%08x\n", tf->status);
    cprintf("  epc      0x%08x\n", tf->epc);
    cprintf("  badvaddr 0x%08x\n", tf->badvaddr);
    cprintf("  cause    0x%08x\n", tf->cause);
}
```

`tf->epc` 是异常发生时的 PC（RISC-V 里是 `sepc` 存的，进 C 结构体以后叫 `epc`）

`tf->cause` 是异常/中断的原因号。最高1位区分是中断/异常，低63位表示中断或异常的具体编号

`tf->badvaddr` 是访问异常地址

代码块

```
1  void print_regs(struct pushregs *gpr) {
2      cprintf("  zero     0x%08x\n", gpr->zero);
3      ...
4      cprintf("  t6       0x%08x\n", gpr->t6);
5  }
```

`struct pushregs` 是“被汇编代码推到栈上的所有通用寄存器”。

在这里，RISC-V 的 32 个寄存器都打印出来了。

1.5 中断处理 `interrupt_handler(struct trapframe *tf)`

代码块

```
1  void interrupt_handler(struct trapframe *tf) {
2      intptr_t cause = (tf->cause << 1) >> 1;
3      switch (cause) {
4          case IRQ_U_SOFT: ...
5          case IRQ_S_SOFT: ...
6          ...
7          case IRQ_S_TIMER:
8              // 这里是要完成的实验
9              break;
10         ...
11         default:
12             print_trapframe(tf);
13             break;
14     }
15 }
```

第2行的作用是把最高位砍掉，只留下中断号本体。因为在 RISC-V 里，`scause` 的**最高位**表示“这是中断还是异常”（1是中断，0是异常）。剩下的低位才是“到底是第几个中断/第几个异常”。

下面的switch-case就是判断具体是哪一个中断，然后根据中断类型执行对应的操作。

`IRQ_S_TIMER` 这个就是超级模式的时钟中断，也是本次基础任务要实现的操作。

1.6 异常处理 `exception_handler(struct trapframe *tf)`

代码块

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        case CAUSE_MISALIGNED_FETCH:
            break;
        ...
        case CAUSE_ILLEGAL_INSTRUCTION:
            // 实验要补的
            break;
        case CAUSE_BREAKPOINT:
            // 这里也是
            break;
        ...
        default:
            print_trapframe(tf);
            break;
    }
}
```

跟上面中断指令是一样的逻辑。因为异常的最高位是0，所以不用特别的砍掉最高位。

在challenge中，需要写非法指令异常+断点异常。

1.7 区分中断/异常 trap_dispatch(struct trapframe *tf)

代码块

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

RISC-V 的 `scause` 当做**有符号数**看的话，中断那一类最高位是 1，所以会变成负数。所以< 0就是中断，去 interrupt_handler()函数；反之去异常处理函数。

1.8 顶层入口 `trap(struct trapframe *tf)`

代码块

```
void trap(struct trapframe *tf) {
    // dispatch based on what type of trap occurred
    trap_dispatch(tf);
```

```
4   }
```

这个函数就是C 语言层面的总入口。汇编的 `__alltraps` 把寄存器都保存好，做成一个 `struct trapframe`，把它的指针传进来。这里的 `trap()` 收到以后，就调用 `trap_dispatch()` 去分“中断/异常”。分完以后处理掉中断/异常，就返回。

2. 完成时钟中断计数

2.1 引入sbi.h头文件

引入sbi.h的目的是为了使用 `sbi_shutdown` 函数，因为在打印10行后需要关机操作。

2.2 实现 IRQ_S_TIMER 逻辑

代码块

```
1   case IRQ_S_TIMER: {
2       static size_t print_count = 0;
3
4       clock_set_next_event();
5       ticks++;
6       if (ticks % TICK_NUM == 0) {
7           print_ticks();
8           print_count++;
9           if (print_count == 10) {
10              sbi_shutdown();
11          }
12      }
13      break;
14  }
```

里面就是一个很简单的计时器逻辑，ticks每增加TICK_NUM次，就输出一次，然后print_count增1。当print_count＝＝10，说明已经打印10次，调用sbi_shutdown()关机。

`clock_set_next_event()` 用来**重新设置下一次时钟中断。**因为在RISCV中，每次时钟中断触发后，硬件的定时器标志会被清零，需要软件手动告诉它“下一次什么时候再中断”。所以不写这行的话就会只执行一次时钟中断。

3. 主路径解释

3.1 启动阶段

链接脚本 `kernel.ld` 告诉链接器：程序的入口函数是 `kern_entry`。当 CPU 启动内核时，SBI 会把控制权交给这个地址。

`kern_entry` 在 `entry.S` 文件中，是整个系统启动的第一条执行指令。在执行完接受参数、建立页表、设置栈指针之后，会跳转到 C 函数 `kern_init` 。

3.2 内核主函数 kern_init()

1. 首先就是初始化控制台，让 `cprintf()` 打印调试信息。
2. 之后用设置中断向量寄存器，调用 `idt_init()` ，把所有 trap（异常/中断）入口指向统一入口 `__alltraps` 。
3. 之后初始化物理内存管理， `pmm_init()` 建立页分配器
4. 再次 `idt_init()` ，确保安全。
5. 初始化时钟中断 `clock_init()` ，打开定时器中断源，调用 `clock_set_next_event()` 注册第一次时钟事件。
6. 开启总中断 `intr_enable()` 。设置 `sstatus.SIE = 1` ，允许 CPU 响应中断信号。
7. 进入主循环，主循环while (1)看似什么都不做，实际上是CPU 空转等待中断。中断一到，就被打断进入异常处理流程。

主流程实际上在： 等待中断 + 响应中断 + 返回原地。

扩展练习Challenge1：描述与理解中断流程

1. 处理终端异常的流程

1.1 异常产生

当CPU指令出现以下情况的时候，会触发异常或者中断：

- 指令错误（非法指令）
- 地址错误
- 时钟中断
- 外设中断

出现这些现象的时候，硬件就会开始工作。首先会自动保存当前指令的PC到sepc，接着需要保存产生异常的原因到scause，接着需要保存发生异常的地址到sbadaddr，做完这些最后就需要跳转到stvec，也就是异常处理的入口。在 uCore 的初始化中， `stvec` 被设置为 `__alltraps` ，也就是说**所有异常和中断都跳转到 `__alltraps` 汇编入口。**

```
代码块
1   write_csr(stvec, &__alltraps);  // trap.c → idt_init()
```

1.2 异常入口

代码块

```
__alltraps:
    SAVE_ALL
    move  a0, sp
    jal trap
```

`__alltraps` 在 `trapentry.S` 中，是中断/异常的统一入口点。这一段完成了几个关键工作，首先是 `SAVE_ALL`，来保存上下文，宏 `SAVE_ALL` 会为 trapframe 分配空间（`addi sp, sp, -36*REGBYTES`），并且把所有通用寄存器（x0~x31）保存到当前栈上；接着再保存 CSR（`sstatus`, `sepc`, `sbadaddr`, `scause`）到后面固定位置。最终栈上保存的结构与 `struct trapframe` 对齐，这样 C 语言中的 `trap(struct trapframe *tf)` 就能直接使用这些寄存器。

`move a0, sp` 会把当前的栈指针传给 `a0`，这里会保存好的 trapframe 地址传递给 `trap()`，使得 C 层的 trap 处理函数能访问中断现场。最后跳转到 `trap`，调用 C 函数 `trap(tf)` 进行分类处理。

1.3 C层的处理流程

`trap()` 会根据中断类型选择不同的处理逻辑，这里的 `trap_dispatch` 中，

若 `cause` 为负，表示中断（最高位为 1），否则为异常。

代码块

```
void trap(struct trapframe *tf) {
    trap_dispatch(tf);
}

if ((intptr_t)tf->cause < 0)
    interrupt_handler(tf);
else
    exception_handler(tf);
```

1.4 异常或者中断的具体处理函数

代码块

```
case IRQ_S_TIMER:
    clock_set_next_event();
    ticks++;
    if (ticks % 100 == 0) print_ticks();
```

在 `interrupt_handler()` 里有 `IRQ_S_TIMER`,这个会设置下一次时钟事件、递增全局时间、打印计数。在 `exception_handler()` 中根据 `tf->cause` 分类，可输出错误信息、修改 `tf->epc` 让程序跳过异常指令等。

1.5 返回

代码块

```
__trapret:
    RESTORE_ALL
    sret
```

C 函数 `trap()` 执行完毕后，返回到 `trapentry.S` 的下一条指令。 `RESTORE_ALL` 会恢复之前保存的寄存器，并且写回 `sstatus` 和 `sepc` ，最后执行 `sret` 返回到用户或内核态。在 RISC-V 调用约定里，函数第一个参数放在 `a0` ，所以这一条指令就是让 `trap()` 函数能直接拿到当前 trapframe的地址，从而访问被保存的寄存器和中断信息。

2. `mov a0, sp` 的目的

这条指令出现在 `__alltraps` 中，这条指令的作用是把当前的栈指针（此时已经保存好所有寄存器的 trapframe 起始地址）传递给 `a0` ，作为参数传入 `trap(struct trapframe *tf)` 。

3. `SAVE_ALL` 中寄存器在栈中的位置是怎么确定的

在 `SAVE_ALL` 宏中，保存顺序是：

代码块

```
STORE x0, 0*REGBYTES(sp)
STORE x1, 1*REGBYTES(sp)
...
STORE x31, 31*REGBYTES(sp)
```

之后再保存：

代码块

```
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)
```

这些对应到 `trapframe` 结构体中的成员排列：

代码块

```
struct trapframe {
    struct pushregs gpr; // 32 个通用寄存器
    uintptr_t status;
    uintptr_t epc;
    uintptr_t badvaddr;
    uintptr_t cause;
};
```

因此寄存器在栈中的位置是由 `trapframe` 结构体定义的顺序**以及汇编宏 SAVE_ALL 的保存顺序一一对应确定的。**

4. 对于任何中断，__alltraps 中都需要保存所有寄存器吗

不一定需要，但在 uCore 中选择“统一保存”是为了简化实现。因为在硬件级别中，有的中断不会破坏所有寄存器；其次在软件实现上，如果每种中断保存不同寄存器，恢复时就很麻烦，还得单独写不同的恢复逻辑；为了通用性，采用“统一入口 + 保存全部寄存器”方案（`SAVE_ALL`），这样 C 代码层不需要考虑寄存器差异，恢复时也可以 `RESTORE_ALL` 一次性还原。

因此，在 uCore 中，`__alltraps` 对所有中断都保存全部寄存器，是为了保证一致性和简化设计，而不是每种中断都真的需要保存所有寄存器。

扩展练习 Challenge2：理解上下文切换机制

1. 汇编代码 csrw sscratch, sp；csrrw s0, sscratch, x0实现了什么操作，目的是什么？

1.1 csrw sscratch, sp

代码块

```
.macro SAVE_ALL
    csrw sscratch, sp
    addi sp, sp, -36 * REGBYTES
    ...
```

这一条指令的意思是先把当前的 `sp`（陷入异常前正在用的栈指针）存进 CSR `sscratch` 中；之后把 `sp` 往下挪一大块，用来放 trapframe。也就是说，这一步是**先把“原来的栈指针”备份到一个地方**，防止一会儿改了 `sp` 就找不回来了。

1.2 csrrw s0, sscratch, x0

代码块

```
# Set sscratch register to 0, so that if a recursive exception
# occurs, the exception vector knows it came from the kernel
```

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

这条指令是把所有的寄存器都存完之后，**把刚才存进 sscratch 的“原 sp”读出来，放进 s0；同时把 sscratch 写成 0。**

所以这两条指令的目的就是，进 trap 的第一刻，先把陷入前的栈指针保存进sscratch；等到栈空间建好了，再把它从 sscratch 取出来，存到 trapframe 里；同时把 sscratch 清 0，让后续的嵌套异常能看出来这是内核里的 trap，不是用户态过来的。

2. restore all里面却不还原save中的csr，这里store的意义

SAVE ALL

```
1   csrr s3, sbadaddr
2   csrr s4, scause
3   STORE s3, 34*REGBYTES(sp)
4   STORE s4, 35*REGBYTES(sp)
```

STORE ALL

```
1   LOAD s1, 32*REGBYTES(sp)   # sstatus
2   LOAD s2, 33*REGBYTES(sp)   # sepc
3   csrw sstatus, s1
4   csrw sepc, s2
```

可以看到在恢复寄存器时，**没有**恢复 `sbadaddr` / `scause`，这是因为这两个寄存器保存的是异常有关的信息，是用来给内核看的，不需要恢复给用户看。 `sbadaddr` 保存了这次异常访问到的出错地址， `scause` 表示这次 trap 的**原因。**C 代码里要用这俩东西来判断是时钟中断、外部中断、缺页、非法指令，于是它必须先把它们存到 trapframe 里，也就是说，这里store的意义就是让C层能够看到它们，而不是最后还回去。

其次就是返回用户态/内核原点真正要恢复的是 `sstatus` 和 `sepc`，表示回去后是什么特权级以及要回到哪条指令，因此这两个恢复指令会重点写。再就是很多实现中 `sbadaddr` / `scause` 这两个CSR本来就不是要求被中断返回指令还原的寄存器，trap 发生时硬件把“发生了什么”写进这些CSR，；OS 看一眼、处理完、返回；下次再发生 trap，会再写新的值进去。所以它们是**一次性的“报告寄存器”**，而不是需要保持的“上下文寄存器”。

扩展练习 Challenge3：完善异常中断处理

1. 功能目标

完善操作系统对非法指令异常和断点异常的处理能力。当系统触发这两类异常时，在异常处理函数中捕获并输出异常类型、异常指令的触发地址，并通过修改指令指针（`epc`）确保CPU能正确跳过异常指令，避免无限触发。

2. 代码实现

2.1 编译器参数说明

为简化异常测试逻辑，通过编译器参数 `-march=rv64ima` 禁用 RISC-V 的 C 扩展（压缩指令集）。该参数指定目标架构为：64位基础架构（`rv64`）+ 整数指令集（`i`）+ 乘法除法扩展（`m`）+ 原子操作扩展（`a`），不含压缩指令扩展（`c`）。

禁用 C 扩展后，所有指令均为 32位（4字节） 且强制4字节对齐。这确保异常处理中通过 `tf->epc += 4` 即可稳定跳转到下一条完整的有效指令，无需额外添加 `nop` 填充，简化了测试函数设计。

2.2 测试函数编写（`kern/init/init.c`）

为验证操作系统对非法指令异常和断点异常的处理能力，在 `init.c` 中设计测试函数主动触发异常。

1. 非法指令异常测试函数

代码块

```
// 触发非法指令异常：插入一条RISC-V未定义的指令
void test_illegal_instruction(void) {
    cprintf("=== 触发非法指令异常 ===\n");
    __asm__ __volatile__(".word 0x0000000b");  // 插入无效操作码的指令
}
```

设计逻辑：

- 通过内联汇编 .word 0x0000000b 插入一条 RISC-V 未定义的操作码，CPU 执行时会识别为 “非法指令”，触发 CAUSE_ILLEGAL_INSTRUCTION 异常。

当异常处理通过 `tf->epc += 4` 跳过非法指令后，CPU 会跳转到后一条合法指令，继续推进 `test_breakpoint` 函数的调用流程。

2. 断点异常测试函数

代码块

```
// 触发断点异常：执行32位ebreak指令，无需nop填充
void test_breakpoint(void) {
    cprintf("=== 触发断点异常 ===\n");
    __asm__ __volatile__("ebreak");  // 32位断点指令
}
```

设计逻辑：

- ebreak 是 RISC-V 基础指令集（i）中的 32 位断点指令，地址为 4 字节对齐。
- 执行后触发 CAUSE_BREAKPOINT 异常，异常处理中 tf->epc += 4 跳转到下一个 4 字节地址。

> 注：在之前的实践中发现，若未禁用 C 扩展，ebreak 会被编译为 16 位压缩指令，此时若直接 tf->epc += 4 会跳转到指令外的非法地址，引发重复异常。需动态判断指令长度（加 2 或 4）或添加 nop 填充，流程会更复杂。禁用 C 扩展后，所有指令为 32 位且 4 字节对齐，tf->epc += 4 可直接稳定跳转，简化了异常处理逻辑。

2.3 测试函数的调用时机（`kern/init/init.c`）

测试函数需在中断/异常机制启用后调用，确保异常能被正确捕获：

代码块

```
int kern_init(void) {
    // 初始化控制台、物理内存、中断描述符表等
    // ...
    clock_init();    // 初始化时钟中断
    intr_enable();   // 启用中断/异常响应（关键：确保异常能进入处理逻辑）

    // 按顺序调用测试函数，先触发非法指令异常，再触发断点异常
    test_illegal_instruction();
    test_breakpoint();

    while (1);
}
```

`intr_enable()` 开启 CPU 对异常的响应能力（设置 `sstatus` 寄存器的 `SIE` 位），此时触发的异常才能被 `exception_handler` 捕获并处理，完成异常类型识别、地址输出和指令跳过等核心操作。

2.3 异常处理逻辑完善（`kern/trap/trap.c`）

在 `exception_handler` 中针对 `CAUSE_ILLEGAL_INSTRUCTION` 和 `CAUSE_BREAKPOINT` 两个异常类型，补充处理逻辑：

代码块

```
void exception_handler(struct trapframe *tf) {
    switch (tf->cause) {
        // 其他异常类型处理...

        case CAUSE_ILLEGAL_INSTRUCTION:
            // 非法指令异常处理
            cprintf("Exception type: Illegal instruction\n");
```

```
            cprintf("Illegal instruction caught at 0x%p\n", (void*)tf->epc);
            tf->epc += 4;  // 跳过当前4字节非法指令
            break;

        case CAUSE_BREAKPOINT:
            // 断点异常处理
            cprintf("Exception type: breakpoint\n");
            cprintf("breakpoint caught at 0x%p\n", (void*)tf->epc);
            tf->epc += 4;  // 跳过当前4字节断点指令
            break;

        // 其他异常类型处理...
    }
}
```

- **关键逻辑说明**：`tf->epc` 存储了触发异常的指令地址，通过 `tf->epc += 4` 可让CPU跳过当前异常指令（RISC-V指令为4字节对齐），避免异常无限触发。

3. 功能验证

3.1 编译运行与输出验证

执行 `make qemu` 启动系统后，控制台输出如下内容：

代码块

```
=== 触发非法指令异常 ===
Exception type: Illegal instruction
Illegal instruction caught at 0x0xffffffffc02000b8
=== 触发断点异常 ===
Exception type: breakpoint
breakpoint caught at 0x0xffffffffc02000c8
...（时钟中断正常打印）
```

3.2 地址正确性验证

通过查看内核镜像反编译生成的 kernel.asm 文件，可验证输出地址与异常指令的实际虚拟地址完全一致：

- 非法指令 `.word 0x0000000b` 对应地址 `0x0xffffffffc02000b8`。
- 断点指令 `ebreak` 对应地址 `0x0xffffffffc02000c8`。

```
ffffffffc02000b8:   0000000b            .word   0x0000000b
    cprintf("=== 触发断点异常 ===\n");
ffffffffc02000bc:   00003517            auipc   a0,0x3
ffffffffc02000c0:   80c50513            addi    a0,a0,-2036
ffffffffc02000c4:   06c000ef            jal ffffffffc0200130
    __asm__ volatile__(
ffffffffc02000c8:   00100073            ebreak
```