南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

操作系统实验报告

中断与中断处理流程

年级：2023 级

姓名：康志杰 王春晖 彭浩然

2025 年 11 月 1 日

目录

一、 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：

- RISC-V 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

二、 实验原理

(一) RISC-V 中断机制概述

在 RISC-V 架构中，中断（Interrupt）和异常（Exception）统称为 Trap，是导致控制权从当前执行流转移到陷阱处理程序的事件。中断是异步的，由外部设备触发（如时钟中断）；异常是同步的，由当前指令触发（如非法指令、断点）。

RISC-V 支持三个特权模式：M 模式（最高，Machine Mode，用于底层固件如 OpenSBI）、S 模式（Supervisor Mode，操作系统内核）和 U 模式（User Mode，用户程序）。中断默认路由到 M 模式，但通过 `mideleg` 和 `medeleg` 寄存器委托到 S 模式，提高效率。

关键 CSR 寄存器：

- `stvec`：中断向量基址，指向统一入口（如 `__alltraps`）。
- `sepc`：保存被中断指令的 PC。
- `scause`：中断/异常原因编码（最高位 1 表示中断）。
- `stval`：异常附加信息（如无效地址）。
- `sstatus`：状态寄存器，包含 SIE（中断使能）、SPIE（先前 SIE）、SPP（先前特权级）。

中断处理流程：

1. 硬件自动保存上下文：sepc ← PC, scause ← 原因, stval ← 附加信息；SIE ← 0（禁用中断），SPIE ← 原 SIE, SPP ← 原模式；切换到 S 模式。
2. PC ← stvec（Direct 模式下统一入口）。
3. 汇编入口（`trapentry.S`）：`SAVE_ALL` 保存寄存器到栈（TrapFrame 结构体，包括 32 个 GPR 和 CSR），调用 C 处理函数 `trap()`。
4. 分发：根据 scause 判断中断/异常，调用 `interrupt_handler` 或 `exception_handler`。
5. 处理具体事件（如时钟中断累加 ticks）。
6. 返回：`RESTORE_ALL` 恢复寄存器；`sret`：PC ← sepc, SIE ← SPIE, SPP ← 0，返回原模式。

上下文切换使用 sscratch 区分内核/用户栈。时钟中断通过 OpenSBI 的 sbi_set_timer 和 rdtime 实现，每 10ms 触发一次。

原子操作：使用 local_intr_save/restore 临时禁用中断，确保关键段（如内存分配）不被打断。

(二) 关键代码结构

- kern/trap/trapentry.S：汇编宏 SAVE_ALL 和 RESTORE_ALL 实现上下文保存/恢复。
- kern/trap/trap.c：TrapFrame 结构体，trap_dispatch 分发，处理时钟/异常。
- kern/driver/clock.c：初始化时钟，设置定时器事件。

三、 实验过程

(一) 实验重要知识点与对应 OS 原理知识点关联

1. RISCV 特权级（M/S/U 模式）、中断委托（mideleg/medeleg）对应 OS 原理中的特权级划分（内核态/用户态）、特权隔离机制：二者核心都是通过权限分层保障系统安全，实验中 RISC-V 的三级特权级是硬件层面的具体实现，OS 原理的内核态/用户态是更通用的软件逻辑抽象；中断委托是硬件提供的“权限下放”机制，对应原理中“内核直接处理用户态请求”的设计，让 OS 无需经过 M 模式即可响应中断，提升执行效率。

2. 中断/异常统一为 Trap、scause/sepc/stval 寄存器作用对应 OS 原理中的中断与异常概念及分类、中断上下文保存: 原理中中断是异步外部事件、异常是同步指令错误, 实验中 RISC-V 将二者统一为 Trap，是硬件层面的简化设计；scause/sepc/stval 是硬件自动保存的上下文关键信息，对应原理中“中断发生时保存现场”的核心步骤，实验更侧重硬件寄存器的具体操作，原理更侧重通用逻辑流程。

3. 上下文切换（SAVE_ALL/RESTORE_ALL 宏、trapframe 结构体）对应 OS 原理中的进程上下文切换、中断现场保护与恢复：二者目的一致，都是确保中断处理后能恢复原执行流；实验中通过汇编宏和结构体固化寄存器保存/恢复的顺序和位置，是原理“现场保护”的硬件适配实现，原理更具通用性，实验需贴合 RISC-V 寄存器布局和指令集特性。

4. stvec 寄存器配置、中断入口点（__alltraps）对应 OS 原理中的中断向量表、中断处理程序入口设计：实验中 stvec 的 Direct 模式是中断向量表的简化实现（单一入口），对应原理中“中断触发后跳转到指定处理程序”的核心逻辑；差异在于原理支持多中断类型对应多入口，实验为简化设计采用单一入口 +scause 判断，更适配小型内核场景。

5. 时钟中断处理、中断使能/屏蔽（sstatus.SIE、sie 寄存器）对应 OS 原理中的时钟中断与进程调度、中断屏蔽与原子操作：二者核心都是利用时钟中断实现 CPU 资源调度，通过屏蔽中断保证关键操作的原子性；实验中需结合 RISC-V 专用寄存器（sstatus/sie）实现中断的使能与屏蔽，原理是通用逻辑框架，实验需适配具体硬件寄存器的操作方式。

6. 非法指令/断点异常捕获（scause 类型判断、stval 地址获取）对应 OS 原理中的异常处理机制、错误检测与响应：原理强调异常发生后需先识别类型再针对性处理（如终止程序、调试），实验中通过解析 scause 编码（非法指令 =2、断点 =3）和 stval 获取相关地址，是原理“异常分类处理”的硬件级落地，实验更侧重具体编码解析和寄存器数据读取。

(二) OS 原理中重要但实验未对应的知识点

1. 外部设备中断（如磁盘 I/O 中断）处理：原理中 I/O 中断是设备与 CPU 协作的核心，实验仅聚焦时钟中断、非法指令/断点异常，未涉及真实外设中断的请求、响应与 I/O 完成处理，简化了外设交互场景。

2. 中断优先级与嵌套中断：原理中中断优先级决定处理顺序，支持高优先级中断打断低优先级处理，实验中未配置中断优先级，且默认关闭 S 模式中断避免嵌套，未体现优先级调度逻辑。

3. 系统调用（ecall）的完整处理流程：原理中系统调用是用户态请求内核服务的核心方式，实验未实现 ecall 指令的捕获与服务响应，仅处理了被动触发的时钟中断和异常，缺失用户态到内核态的主动调用场景。

4. 虚拟内存相关异常（如缺页异常）处理：原理中缺页异常是虚拟内存管理的关键，实验未涉及虚拟内存机制，因此未实现该类异常的处理，仅覆盖了简单的指令级异常。

5. 中断控制器（如 PLIC）的配置与使用：原理中中断控制器负责管理多个外设中断请求，实验未涉及多外设中断路由，简化了中断触发的硬件链路。

(三) 练习 1

这部分内容实现中断处理函数中的时钟中断。代码如下所示，首先得先定义一个全局变量 PRINT_COUNT 用于记录打印次数，并初始化为 0。触发时钟中断时需要立即调用函数 clock_set_next_event 设置下次时钟中断。该函数内部调用 sbi_set_timer 函数，参数为下次触发时钟中断的时间 get_cycles() + timebase。

设置好下次时钟中断后，记录时钟中断次数的变量 ticks 需要自增。接着判断时钟中断次数是否等于 TICK_NUM(100 次)，如果等于则调用函数 print_ticks 输出字符串"100ticks"。同时将 PRINT_COUNT 自增 1，代表打印次数加一。最后再判断 PRINT_COUNT 是否等于 10，如果等于则调用函数 sbi_shutdown, 该函数内部调用 sbi_call 函数实现系统关机。

```
        case IRQ_S_TIMER:
            clock_set_next_event();
            ticks++;
            if(ticks==TICK_NUM){
                print_ticks(TICK_NUM);
                ticks=0;
                PRINT_COUNT++;
            }
            if(PRINT_COUNT==10){
                 sbi_shutdown();

            }
            break;
```

接着在终端输入命令 make qemu，观察输出结果如图1所示，观察到字符串"++ setup timer interrupts", 该字符串是在 clock.c 文件中的 clock_init 函数实现的，说明已经完成了时钟初始化和设置好第一次时钟中断事件。接着观察到每隔 100 个时钟中断会输出字符串"100ticks"，而且打印到 10 次后，会退出 qemu。这一输出结果与预期结果一致。

```
Special kernel symbols:
  entry  0xffffffffc0200054 (virtual)
  etext  0xffffffffc0201e78 (virtual)
  edata  0xffffffffc0206028 (virtual)
  end    0xffffffffc02064a0 (virtual)
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
++ setup timer interrupts
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
100 ticks
kzhj@kzhj-VMware-Virtual-Platform:~/公共/labcode/lab3$
```

图 1: make qemu 的输出结果

最后，在 best_fit_pmm.c 文件中补全上次实验代码并在 pmm.c 中将内存管理单元置换为 best_fit_pmm_manager。在终端输入命令 make grade 进行测试，测试结果如图2所示，所有测试点均已通过，得到了满分。

```
kzhj@kzhj-VMware-Virtual-Platform:~/公共/labcode/lab3$ make grade
>>>>>>>>>> here_make>>>>>>>>>>>
gmake[1]: 进入目录“/home/kzhj/公共/labcode/lab3” + cc kern/init/entry.S + cc ker
n/init/init.c + cc kern/libs/stdio.c + cc kern/debug/kdebug.c + cc kern/debug/km
onitor.c + cc kern/debug/panic.c + cc kern/driver/clock.c + cc kern/driver/conso
le.c + cc kern/driver/dtb.c + cc kern/driver/intr.c + cc kern/trap/trap.c + cc k
ern/trap/trapentry.S + cc kern/mm/best_fit_pmm.c + cc kern/mm/default_pmm.c + cc
 kern/mm/pmm.c + cc libs/printfmt.c + cc libs/readline.c + cc libs/sbi.c + cc li
bs/string.c + ld bin/kernel /home/kzhj/公共/riscv/riscv-toolchain/bin/riscv64-un
known-elf-objcopy bin/kernel --strip-all -O binary bin/ucore.img gmake[1]: 离开
目录“/home/kzhj/公共/labcode/lab3”
>>>>>>>>>> here_make>>>>>>>>>>>
<<<<<<<<<<<<<<< here_run_qemu <<<<<<<<<<<<<<<<<<
try to run qemu
qemu pid=8489
<<<<<<<<<<<<<<< here_run_check <<<<<<<<<<<<<<<<<<
  -check physical_memory_map_information:    OK
  -check_best_fit:                           OK
  -check ticks:                              OK
Total Score: 30/30
```

图 2: make grade 的测试结果

(四) Challenge1

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（基于 RISC-V 架构）处理中断异常（Trap）的流程结合了硬件自动响应和软件上下文管理。以下从异常产生开始，详细描述完整流程。流程分为硬件自动阶段、软件保存/处理阶段和返回阶段。

1. Trap 产生（硬件自动阶段）

- **触发事件**: 异常 (Exception, 同步事件, 如非法指令 CAUSE_ILLEGAL_INSTRUCTION 或断点 CAUSE_BREAKPOINT) 或中断 (Interrupt, 异步事件, 如时钟 IRQ_S_TIMER) 发生。
- **硬件响应**:
  - 保存当前 PC（程序计数器）到 `sepc` CSR（被中断指令地址）。
  - 记录异常/中断原因到 `scause` CSR（最高位 1 表示中断，0 表示异常）。
  - 保存附加信息到 `stval` 或 `sbadaddr`（如异常地址）。
  - 修改 `sstatus`：SIE ← 0（禁用中断，避免嵌套）；SPIE ← 原 SIE；SPP ← 当前特权模式（0 为 U 模式）。
  - 切换特权模式：从 U/S 模式进入 S 模式（监督模式，内核态）。
  - PC ← `stvec`（中断向量基址，ucore 设置为 ___alltraps 地址，Direct 模式统一入口）。

  此时，CPU 已跳转到 Trap 入口，但寄存器（GPR）未保存——这是软件的责任。

2. 软件保存与分发阶段（___alltraps 入口）

- **跳转到 ___alltraps**（trapentry.S）：执行 `SAVE_ALL` 宏。
  - 扩展栈: `addi sp，sp，-36 * REGBYTES`（分配 TrapFrame 空间：32 GPR + 4 CSR）。
  - 保存 GPR：逐个 `STORE xN，offset(sp)`（x0-x31，x2/sp 通过 sscratch 间接保存原值）。
  - 保存 CSR: `csrrw s0，sscratch，x0`（清零 sscratch 标记内核态）; 读 sstatus/sepc/sbadaddr/scause 到临时寄存器，再 STORE 到栈（偏移 32-35）。
- **参数传递**：`move a0，sp`（a0 = TrapFrame 地址）。
- **调用处理**：`jal trap`（跳转到 C 函数 trap(tf)，trap.c 中调用 trap_dispatch(tf)）。
  - trap_dispatch：根据 `tf->cause`（scause 值）分发：
    - 若 < 0（中断）：interrupt_handler(tf)，switch 根据低位（如 IRQ_S_TIMER）处理（e.g., 时钟中断累加 ticks）。
    - 否则（异常）：exception_handler(tf)，switch 根据值处理（e.g., 断点输出信息，tf->epc += 4 跳过指令）。

    处理可能修改 tf（e.g., epc/status）。

3. 返回阶段（___trapret）

- trap() 返回后，执行 ___trapret：`RESTORE_ALL` 宏。
  - 先恢复 CSR：`LOAD s1, 32(sp)`（sstatus）；`LOAD s2, 33(sp)`（sepc）；`csrw sstatus, s1；csrw sepc, s2`。
  - 恢复 GPR：逐个 `LOAD xN, offset(sp)`（x1/x3-x31，最后 x2/sp）。
- **返回执行**：`sret` 指令（硬件执行）：
  - PC ← sepc（从恢复/调整地址继续）。
  - sstatus.SIE ← SPIE（恢复中断使能）。
  - sstatus.SPIE ← 1；SPP ← 0（为下次准备）。
  - 切换模式：基于 SPP（0 → U 模式）。

  恢复原执行流。

整个流程确保原子性和上下文完整，支持从用户态（U）到内核态（S）的切换。ucore Lab3 验证通过时钟/断点处理。

4. mov a0, sp 的目的是什么?

`move a0, sp`（等价于 `addi a0, sp, 0`）位于 ___alltraps 中 SAVE_ALL 后、jal trap 前。其目的是：将 TrapFrame 结构体地址（当前 sp 指向栈顶的 TrapFrame）作为第一个参数（a0 寄存器）传递给 C 函数 trap()。

根据 RISC-V ABI（Application Binary Interface）调用约定，函数第一个参数通过 a0 传递。这允许 trap(tf) 访问保存的上下文（tf->cause 分发、tf->epc 修改等）。不传递参数，C 函数无法读取硬件报告（scause 等），导致处理失败。

5. SAVE_ALL 中寄存器保存在栈中的位置是什么确定的?

寄存器保存在栈中的位置由**栈指针 sp 的当前值**（基址）确定。具体：

- `addi sp, sp, -36 * REGBYTES`：sp 调整为 TrapFrame 起始地址（低地址）。
- `STORE xN, offset * REGBYTES(sp)`：每个寄存器存到固定偏移（e.g., x0: 0(sp), x1: 1(sp), ... x31: 31(sp)；CSR: 32-35(sp)）。
- 偏移由 TrapFrame 结构体布局决定（kern/trap/trap.h：pushregs gpr[32] + status/epc/badvaddr/cause）。

位置固定（相对 sp），确保 TrapFrame 布局一致，便于 C 访问（tf = (struct trapframe *)sp）。x2/sp 特殊：通过 sscratch 保存原值，到 2(sp)。

6. 对于任何中断，___alltraps 中都需要保存所有寄存器吗? 请说明理由。

对于任何中断（或 Trap），___alltraps 中都需要保存所有寄存器（32 个 GPR + 关键 CSR）。理由如下：

- **统一入口设计**：ucore 使用 Direct 模式（stvec 指向单一 ___alltraps），所有 Trap（中断/异常）共享入口。无法预知类型，只能保存完整上下文，以支持任意返回点。

- **上下文完整性**：中断异步打断执行，寄存器可能含用户/内核数据。保存所有确保恢复时 CPU 状态精确（如 ra 返回地址、临时寄存器），防止数据丢失或崩溃。
- **支持切换**：允许 OS 修改 TrapFrame（e.g., 进程切换设新 epc），或递归 Trap（sscratch 区分栈）。不全保存，可能破坏调用约定或调试。
- **开销权衡**：虽有 40 条指令开销，但 RISC-V 高效；Vectored 模式可优化（不同类型不同入口），但 ucore 简化用 Direct，提高代码复用。

如果仅保存部分，需类型特定逻辑，复杂化实现。

（五） challenge2

1. csrw sscratch, sp 与 csrrw s0, sscratch, x0 的作用与目的

在发生异常或中断时, CPU 需要保存当前的栈指针以切换到内核栈执行。指令 csrw sscratch, sp 的作用是将当前的 sp 值写入控制状态寄存器 sscratch，相当于临时备份当前用户栈指针，防止在陷入内核后丢失。接着执行 csrrw s0, sscratch, x0，这条指令会将 sscratch 的旧值（也就是刚保存的 sp）读入通用寄存器 s0，同时把 sscratch 置为 0。这样一来，s0 保存了陷入前的用户栈指针，而 sscratch 的清零则起到标志作用，表明当前已经处于内核态。如果此时再次发生嵌套陷入，系统可以通过检查 sscratch 是否为 0 来判断陷入来源，从而避免错误地再次切换栈或破坏原有的上下文。总体来说，这两条指令的目的在于：安全地保存陷入前的栈指针，并通过清零 sscratch 来区分内核态与用户态陷入，为后续处理提供判断依据。

2. save all 保存了 stval/scause，但 restore all 不恢复的原因及意义

在陷入现场保存阶段（save all），系统会将包括 stval（异常相关地址）和 scause（异常原因）在内的若干寄存器保存到 trapframe 中。这么做的意义在于，stval 记录了异常时访问出错的地址（如页错误地址），scause 记录了异常的类型和编码信息，这些信息对于 C 语言层面的陷入处理函数（如 trap(struct trapframe *tf)）非常重要。上层函数需要依靠这些字段判断异常的种类、打印调试信息、执行异常恢复（如修改 sepc 跳过指令）或终止进程等操作。因此，保存它们是为了分析异常原因和支持软件层面的逻辑处理。

然而，在返回阶段（restore all）并不会恢复 stval 与 scause。这是因为它们是一次性只读诊断寄存器，只反映本次陷入的状态，不影响程序返回时的执行流。真正决定返回位置和特权级的寄存器是 sepc（保存返回地址）和 sstatus（保存状态位，如 SPP、SPIE）。而 stval 与 scause 在下一次陷入时会被硬件重新写入，若人为恢复旧值，不仅没有意义，甚至可能破坏硬件状态。因此，它们的保存仅用于“供读取分析”，并非为了恢复执行。

3. 陷入现场的最小必要恢复集

在异常返回阶段，系统只需要恢复那些与执行状态直接相关的寄存器即可。必须恢复的包括 sstatus 和 sepc，它们决定了返回的指令地址及特权级，同时还要恢复所有通用寄存器，以保证程序的正常继续执行。恢复完成后，通过执行 sret 指令回到陷入前的执行流。相对地，像 stval、scause、以及 sscratch 这样的寄存器无需恢复——stval 和 scause 是本次陷入的诊断信息，不影响程序状态，而 sscratch 在陷入时已经被清零，仅用于标识陷入来源，不参与返回过程。这样的设计既确保了系统返回的最小化路径，又保证了状态恢复的正确性与高效性。

(六) challenge3

1. 指令异常

指令异常是指 CPU 解码指令时，发现当前指令的机器码不属于 RISC-V 指令集中的任何一条合法指令就会触发非法指令异常并做出相应的非法指令异常处理。本次实验中需要触发非法指令异常，在 kern/trap/trap.c 的异常处理函数中捕获并对其进行处理。异常处理代码如下所示。

```
        case CAUSE_ILLEGAL_INSTRUCTION:
            cprintf("Exception type:Illegal instruction\n");
            cprintf("Illegal instruction caught at %p\n",tf->epc);
            tf->epc+=4;
             break;
```

捕获到非法指令异常后，首先会输出异常类型，在这里是“Illegal instruction”。tf 结构体的 epc 中存储着触发非法指令异常的指令地址。然后调用 cprintf 函数输出异常指令地址。最后更新 epc 属性的值。

实现好非法指令异常处理代码后，接着编写如下函数触发该异常并做出异常处理。在 init.c 文件中实现函数 test_illegal_instruction。___asm___ 是 C 语言中用于嵌入汇编代码的关键字，该函数通过该关键字以内联汇编的方式向代码中插入指令”.word 0x00000000\n”。而且 volatile 关键字确保编译器不会优化该指令并 1= 按原样执行。可以观察到 0x00000000 并不是一条有效的指令。所以在 kern_init 函数中调用该函数会在初始化内核过程中引发非法指令异常。

```
    void test_illegal_instruction(void) {
    // 插入一个未定义的指令
    __asm__ volatile (".word 0x00000000\n");
     }
```

接着在命令行输入 make qemu 后得到如图3所示，可以观察到输出触发的异常类型为 Illegal instruction。而且异常指令地址为 0xffffffffc02000a4。

```
Kernel executable memory footprint: 26KB
memory management: best_fit_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
calling test_breakpoint
Exception type:Illegal instruction
Illegal instruction caught at 0xffffffffc02000a4
```

图 3: 非法指令异常处理结果

在编译后生成的 kernel.asm 文件中可以观察到如下汇编代码片段。证明非法指令插入到汇编代码中，这里的 unimp 表示对未定义指令的标注。当 CPU 执行到此处时，未定义指令会触发非法指令异常。此时 CPU 暂停执行后面的指令，优先处理该异常。首先会跳转到宏 ___alltraps 汇编代码，先通过宏 SAVE_ALL 将所有通用寄存器和特权寄存器保存到栈中，接着调用 trap.c 中的 trap 函数，该函数接收 trapframe 的指针，然后通过 cause 判断异常类型，在该情况下调用

异常处理函数 exception_handler，通过 cause 匹配 CAUSE_ILLEGAL_INSTRUCTION，输出结果如图3所示。

```
    test_illegal_instruction();
ffffffffc02000a4:       0000                    unimp
ffffffffc02000a6:       0000                    unimp
    /* do nothing */
    while (1)
ffffffffc02000a8:       a001                    j       ffffffffc02000a8 <
    kern_init+0x54>
```

然后执行代码 tf->epc+=4,此时 epc=0xffffffffc02000a8。处理完异常后程序跳转到宏 ___trapret，通过宏 RESTORE_ALL 回复原来寄存器状态。最后调用命令 sret 从内核态退回到用户态。此时 CPU 的 pc 被 epc 赋值为 0xffffffffc02000a8。观察代码中可以看到，直接跳转到 while 循环中正常执行其他代码。说明非法指令异常代码实现正确，可以正确处理非法指令异常并且继续执行其他代码。由于非法指令长度为 4 字节，所以在 CAUSE_ILLEGAL_INSTRUCTION 中需要将 epc 加 4。

如果连续的非法指令异常，测试代码如下。0x00000000 和 0xFFFFFFFF 均为非法指令。

```
void test_more_illegal_instruction(void) {
    // 插入一个未定义的指令
    __asm__ volatile (
        ".word 0x00000000\n"
        ".word 0xFFFFFFFF\n"
        ".word 0x00000000\n"

    );
}
```

编译后的代码如下，当 CPU 运行到 0xffffffffc02000a4 会触发第一次异常。处理完异常后 pc=epc+4= 0xffffffffc02000a8. 该地址即为第二条非法指令，接着会处理第二次异常，处理完之后 pc=0x0xffffffffc02000ac。该地址也是非法指令，会触发第三次异常。处理完第三次异常后 pc 跳转到 0xffffffffc02000b0，即跳转到 while 循环执行其他正常代码。执行结果如图4所示，与预期结果一致。

```
    test_more_illegal_instruction();
ffffffffc02000a4:       0000                    unimp
ffffffffc02000a6:       0000                    unimp
ffffffffc02000a8:       ffff                    0xffff
ffffffffc02000aa:       ffff                    0xffff
ffffffffc02000ac:       0000                    unimp
ffffffffc02000ae:       0000                    unimp
    /* do nothing */
    while (1)
ffffffffc02000b0:       a001                    j       ffffffffc02000b0 <
    kern_init+0x5c>
```

```
calling test_breakpoint
Exception type:Illegal instruction
Illegal instruction caught at 0xffffffffc02000a4
Exception type:Illegal instruction
Illegal instruction caught at 0xffffffffc02000a8
Exception type:Illegal instruction
Illegal instruction caught at 0xffffffffc02000ac
```

图 4: 多条非法指令异常处理结果

2. 断点异常

断点异常是指在代码中调用 ebreak 命令而触发的异常。本次实验中需要触发断点指令异常，在 kern/trap/trap.c 的异常处理函数中捕获并处理异常。异常处理代码如下所示。首先输出异常类型为 breakpoint，然后输出触发断点异常的指令地址。由于 ebreak 通过其压缩形式 c.ebreak 触发的断点异常，而 c.ebreak 指令大小为 2 字节，所以 epc 只需要加 2 即可跳过当前的断点指令。

```
1     case CAUSE_BREAKPOINT:
2             cprintf("Exception␣type:␣breakpoint\n");
3             cprintf("ebreak␣caught␣at␣%p\n",tf->epc);
4             tf->epc += 2;
5              break;
```

为了测试断点异常处理代码的正确性，编写函数进行测试。将断点指令 ebreak 插入内核初始化代码中。当 cpu 执行到该指令地址时会触发断点异常。

```
1     case CAUSE_BREAKPOINT:
2             cprintf("Exception␣type:␣breakpoint\n");
3             cprintf("ebreak␣caught␣at␣%p\n",tf->epc);
4             tf->epc += 2;
5              break;
```

测试结果如图5所示，观察到输出异常类型为断点异常，触发异常的指令地址为 0xffffffffc02000a4。

```
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0205000
satp physical address: 0x0000000080205000
calling test_breakpoint
Exception type: breakpoint
ebreak caught at 0xffffffffc02000a4
```

图 5: 断点指令异常处理结果

现分析 kernel.asm 文件中的对应代码，代码片段如下所示。观察到 ebreak 指令的地址为 0xffffffffc02000a4，与输出结果一致，当 pc+2 跳过断点指令 ebreak 后，就不会触发断点异常。跳转到 while 循环中正常执行其他指令。证明断点指令异常处理代码实现正确。

```
        cprintf("calling test_breakpoint\n");
ffffffffc0200098:       00002517                auipc   a0,0x2
ffffffffc020009c:       e3850513                addi    a0,a0,-456 # ffffffffc0201ed0 <etext>
ffffffffc02000a0:       042000ef                jal     ra,ffffffffc02000e2 <cprintf>
    __asm__ volatile ("ebreak\n");
ffffffffc02000a4:       9002                    ebreak
    test_breakpoint(); // 调用断点测试函数
    /* do nothing */
    while (1)
ffffffffc02000a6:       a001                    j       ffffffffc02000a6 <kern_init+0x52>
```

最后实现函数 test_more_breakpoint，向代码中插入多个 ebreak 断点，执行命令 make qemu 后得到结果如图6所示，可以观察到代码正确处理了所有断点异常。

```
void test_more_breakpoint(void) {
    // ebreak指令会触发断点异常
    __asm__ volatile (
    "ebreak\n"
    "ebreak\n"
    "ebreak\n"
    );
}
```

```
calling test_more_breakpoint
Exception type: breakpoint
ebreak caught at 0xffffffffc02000a4
Exception type: breakpoint
ebreak caught at 0xffffffffc02000a6
Exception type: breakpoint
ebreak caught at 0xffffffffc02000a8
```

图 6: 多条断点指令异常处理结果

四、 实验总结

通过本实验，我们深入理解了 RISC-V 中断机制的核心原理，包括 Trap 分类、特权模式切换、CSR 寄存器的作用以及上下文保存/恢复的汇编实现。实验中完善了时钟中断处理（每 100 ticks 打印并计数 10 次后关机），验证了中断流程的正确性；扩展练习中描述了完整中断流程、分析了上下文切换细节（如 sscratch 和 CSR 保存意义），并实现了非法指令与断点异常的捕获和处理（输出类型/地址，更新 epc 跳过）。

实验验证了 ucore 的中断框架鲁棒性，运行中观察到稳定输出”100 ticks”，异常处理正确跳过指令。未来可扩展到多核中断或用户态支持。