操作系统 Lab3——中断与中断处理

2313226 肖俊涛 2312282 张津硕 2311983 余辰民

密码与网络空间安全学院

OS启动与中断处理结构图

```
上电 → OpenSBI → （entry.S） → kern_init()(init.c)
      → cons_init()(console.c)                          # 早期输出
      → pmm_init()(pmm.c)                               # 物理内存管理
      → trap_init()(trap.c+trapentry.S)                 # 设置 stvec，中断门
      → clock_init()(clock.c)                           # 设置下一次时钟中断
      → intr_enable()(intr.c)                           # 开全局中断
      → 进入监控台/主循环(kmonitor.c)

(发生中断/异常)
  stvec → trapentry.S(__alltraps保存上下文)
        → trap(tf)(trap.c)
          → trap_dispatch(tf)
            ├─ 计时器中断：clock_set_next_event()+ticks++
            ├─ 外部/软件中断：按需分发
            ├─ 异常(非法指令/系统调用)：调整tf->epc等
            └─ 未处理：打印trapframe并panic
        → __trapret(恢复上下文) → sret 返回
```

实验任务	关注对象	目的
练习 1	时钟中断	理解中断分发与时钟机制

实验任务	关注对象	目的
Challenge 1	异常流程	理解从陷入到返回的全过程
Challenge 2	栈与寄存器切换	理解上下文保存与恢复的机制
Challenge 3	异常处理	区分中断与异常、学会打印与恢复

实验文件解析

根目录

- **Makefile**
  核心构建脚本，调用 `tools/function.mk`；提供 `make`，`make qemu`，`make gdb`，`make grade` 等目标。把 `kern/` 下各模块与 `tools/kernel.ld` 链接在一起

kern/debug（内核调试与监控）

- **assert.h / panic.c**：断言与内核致命错误处理；`panic()` 会打印信息并停机。
- **kdebug.c / kdebug.h / stab.h**：符号解析/回溯支持（结合链接的符号表/调试表），用于在异常时打印更友好的调用栈。
- **kmonitor.c / kmonitor.h**：内核监控台（简单命令行），常用命令如 `help`，`backtrace`。中断相关调试时，可在异常后进入监控台查看状态。

> **与中断的关系**：当发生未处理 trap 时，打印寄存器与回溯信息；写好 `print_trapframe()` 后，配合 kmonitor 能快速定位问题。

kern/driver（硬件驱动）

- **console.c / console.h**
  基于 SBI 的控制台 I/O（`sbi_console_putchar/getchar`），为 `cprintf`、监控台读取键入提供底层支持。
- **clock.c / clock.h**
  计时器（时钟中断）相关：
  - `clock_init()`：设置第一次触发时间（通常：当前时间 + 1/HZ），并初始化 `ticks`、`HZ`。
  - `clock_set_next_event()`：在每次时钟中断后，**重新**预约下一次触发（通过 `sbi_set_timer`）。
  - 常见行为：`ticks++`，可每若干 tick 打印提示，验证中断节拍是否稳定。
- **intr.c / intr.h**
  开关中断与查询的封装（RISC-V 的 `sstatus` / `sie` 等 CSR 操作）：
  - `intr_enable()` / `intr_disable()`：开/关 **全局**中断（设置/清除 `SSTATUS_SIE`）。
  - `read_csr`，`write_csr` 等宏通常在 `libs/riscv.h` 中，`intr.c` 进行更方便的封装。
  - 可能包含 `push_off()/pop_off()` 一类的嵌套关中断计数。

> **与中断的关系**：这三者共同保证**能打印**、**能定时触发**、**能安全开关**中断。

kern/init（启动入口）

- **entry.S**
  RISC-V 启动阶段汇编入口：
  - 设置内核栈指针（`sp`），清零 `.bss`；
  - 必要时设置 `tp`（hart id）等；
  - 跳转到 `kern_init()`（C 语言）
- **init.c**
  `kern_init()` 是**内核 C 级入口**：
  典型步骤：`cons_init()` → 打印横幅 → `pmm_init()` → `trap_init()`（设置 `stvec` 指向 `trapentry.S` 的入口，注册 C 级处理函数）→ `clock_init()` → `intr_enable()` → 进入监控台或主循环。

> **与中断的关系**：`trap_init()` 是本实验关注的重点之一；配合 `clock_init()` + `intr_enable()` 才能真正让时钟中断跑起来。

kern/libs（内核内部的标准库替代）

- **stdio.c**：提供 `cprintf/vcprintf` 等，基于 `libs/printfmt.c` 格式化输出。

kern/mm（内存与地址空间）

- **memlayout.h**：内核虚拟地址布局、物理内存上限等常量定义。
- **mmu.h**：页表项（PTE）位定义、`SATP` 宏等。
- **pmm.c / pmm.h**：物理页分配器初始化与接口（如 `pmm_init()`、`alloc_pages/free_pages`）。

> **与中断的关系**：确保中断保存/恢复现场时栈与数据结构地址有效。

kern/trap（中断/异常核心）

- **trapentry.S**（**关键**）
  RISC-V trap 向量入口与**上下文保存/恢复**：
  - `__alltraps`：保存通用寄存器（x1~x31）、`sepc`、`sstatus`、`stval`、`scause` 到一个 `struct trapframe`；切换到内核栈（若需要）；然后 `call trap`（C 函数）。
  - `__trapret`：从 `trapframe` 恢复寄存器/CSR，最后 `sret` 返回到 `sepc`。
  - 导出给 C 侧的入口符号（供 `trap_init()` 设置 `stvec`）。

  > **一定要保证**：寄存器保存/恢复顺序与 `trap.h` 中的 `struct trapframe` 字段**完全一致。**

- **trap.h**
  定义 `struct trapframe`：包含所有通用寄存器和关键 CSR 影子（`sepc/sstatus/stval/scause` 等）。
- **trap.c**（**关键**）
  - `trap_init()`：设置 `stvec = (uintptr_t)__alltraps`（直达模式），必要时初始化 `sie` 指定位。
  - `trap(struct trapframe *tf)`：C 级总入口，通常做一些通用检查后调用 `trap_dispatch(tf)`。
  - `print_trapframe(tf)` / `print_regs(regs)`：打印现场，便于调试。

- `trap_in_kernel(tf)`：判断陷入时是否处于内核态（决定是否允许某些恢复策略/是否 panic）。

> **与中断的关系**：这是"从硬件陷入→内核 C 逻辑→返回"的**桥梁**，实验的核心改动点在这里和 `trapentry.S`、`clock.c`。

顶层 libs（为内核提供对应库）

- **riscv.h**：RISC-V CSR 操作与位定义（`read_csr/write_csr/set_csr/clear_csr`，`SSTATUS_SIE`、`SIE_STIE` 等）。
- **sbi.c / sbi.h**：SBI 调用封装（console putchar/getchar、set_timer 等）。
- **printfmt.c / stdio.h / stdarg.h / string.c / string.h / readline.c / defs.h / error.h**：格式化、字符串与通用类型/错误码。

> **与中断的关系**：`sbi_set_timer` 触发下一次时钟；控制台输出在中断打印时必需。

tools（构建与运行工具）

- **kernel.ld**：链接脚本，定义内核各段（.text/.data/.bss）地址与对齐、内核起始符号等；确保启动与栈、陷入保存区域布局正确。
- **function.mk**：Makefile 复用函数；
- **grade.sh**：自动评分脚本，按测试点跑 qemu 并比对输出；
- **gdbinit**：预置 GDB 脚本（断点、符号加载等）。

实验文件总结

一级	二级	主要内容与功能	与中断处理的关系
kern	**debug**	提供内核调试工具，包括断言（`assert.h`）、监控台（`kmonitor.c`）、堆栈回溯（`kdebug.c`）、异常打印（`panic.c`）	发生未处理中断或异常时，用于打印寄存器状态、调用栈和调试信息
	driver	设备与硬件抽象层，包括时钟（`clock.c`）、控制台（`console.c`）、中断控制（`intr.c`）	提供时钟中断触发（`clock_set_next_event`）、中断开关（`intr_enable/disable`）和输出接口
	init	启动入口（`entry.S`）、系统初始化（`init.c`）	初始化中断系统：`trap_init()` 设置 stvec，`clock_init()` 初始化定时器，`intr_enable()` 打开中断
	libs	内核自用的标准输入输出函数（`stdio.c`）	支持中断与异常打印（`cprintf`），便于输出调试信息
	mm	内存管理模块，定义物理内存布局（`memlayout.h`）、页表结构（`mmu.h`）、页分配（`pmm.c`）	为中断栈空间、内核数据结构提供内存支撑（间接相关）

一级	二级	主要内容与功能	与中断处理的关系
	trap	中断与异常核心模块：入口汇编（`trapentry.S`）、C级分发（`trap.c`）、上下文定义（`trap.h`）	实现中断保存、恢复、分发与异常打印，是本实验的**核心模块**
libs		提供通用库函数：RISC-V 寄存器操作（`riscv.h`）、SBI 调用（`sbi.c`）、字符串/格式化函数	`sbi_set_timer` 触发时钟中断，`riscv.h` 提供 CSR 操作宏，用于读写 `sstatus`、`scause` 等寄存器
tools		构建与调试工具：链接脚本（`kernel.ld`）、自动评分脚本（`grade.sh`）、GDB 调试配置（`gdbinit`）	`kernel.ld` 决定内核代码与栈的布局，对中断栈和陷入地址定位至关重要
根目录	**Makefile**	定义编译、链接、运行命令	组织编译中断相关汇编与C文件，生成可运行内核

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. 核心代码实现

根据实验要求，我们在 `kern/trap/trap.c` 的 `interrupt_handler` 函数中，针对 `IRQ_S_TIMER` 分支添加了以下处理逻辑：

```
// trap.c
#include <sbi.h>

#define TICK_NUM 100
static volatile int num = 0;  /* 记录已经打印了多少次 "100 ticks" */
// 'ticks' 变量在 clock.c 中定义并由 clock_init() 初始化为 0

// ...
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1; // 抹掉scause最高位
    switch (cause) {
        // ... 其他中断 case ...
        case IRQ_S_TIMER:
            /*
             * (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，输出 "100ticks"，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_shutdown() 关机
             */
            // 1. 立即重新设置下一次时钟中断，是实现周期性中断的关键
```

```
            clock_set_next_event();
            // 2. 增加全局时钟滴答计数
            ticks += 1;
            // 3. 判断是否达到了 TICK_NUM (100)
            if (ticks % TICK_NUM == 0) {
                num += 1;         // 增加打印行数计数
                print_ticks(); // 调用子程序打印 "100 ticks"
                // 4. 判断是否已打印10行
                if (num == 10) {
                    sbi_shutdown(); // 调用SBI服务关机
                }
            }
            break;
        // ... 其他中断 case ...
    }
}
```

2. 定时器中断处理流程分析

2.1. 阶段一：中断与时钟初始化 (系统启动)

在内核启动过程中（`kern/init/init.c` 中的 `kern_init` 函数），系统会执行一系列初始化来为中断处理做准备：

1. **设置中断向量表 (`idt_init`)**:
   - 在 `kern/trap/trap.c` 的 `idt_init` 函数中，内核通过 `write_csr(stvec, &__alltraps)` 指令设置了 `stvec` (Supervisor Trap Vector Base Address Register) 寄存器。
   - `stvec` 指向了 `kern/trap/trapentry.S` 中定义的汇编入口点 `__alltraps`。
   - **作用**：这使CPU无论发生何种 S 模式的中断或异常，都应该跳转到 `__alltraps` 标签处开始执行。
2. **初始化时钟 (`clock_init`)**:
   - 在 `kern/driver/clock.c` 的 `clock_init` 函数中，内核首先通过 `set_csr(sie, MIP_STIP)` 来设置使能 `sie` (Supervisor Interrupt Enable) 寄存器中的 STIP 位，即允许 S 模式的定时器中断。
   - 接着，它调用 `clock_set_next_event()`。
   - `clock_set_next_event()` 内部通过 `sbi_set_timer(get_time() + timebase)` 来设置**第一次**时钟中断。它读取当前时间，加上一个时间间隔（`timebase`），然后通过 `ecall` 请求 M 模式的 OpenSBI 固件在未来的那个时间点触发一次中断。
   - 同时，全局变量 `ticks` 被初始化为 0。
3. **使能全局中断 (`intr_enable`)**:
   - 最后，`kern_init` 调用 `intr_enable()` （定义于 `kern/driver/intr.c`）。
   - 此函数通过 `set_csr(sstatus, SSTATUS_SIE)` 设置 `sstatus` (Supervisor Status Register) 寄存器中的 `SIE` (Supervisor Interrupt Enable) 位。
   - **作用**：这是 S 模式下中断的总开关。只有 `sstatus.SIE` 和 `sie.STIP` *同时*为 1 时，S 模式时钟中断才会被 CPU 真正处理。

2.2. 阶段二：中断触发 (硬件)

当 CPU 内部的 `time` 寄存器值达到了 `sbi_set_timer` 设定的目标值时，时钟中断被触发。此时，CPU 硬件会自动执行以下原子操作：

1. **保存现场**：
   - 将当前 PC 的值保存到 `sepc` (Supervisor Exception Program Counter) 寄存器，以便后续返回。
   - 在 `scause` (Supervisor Cause Register) 寄存器中设置中断原因。对于 S 模式时钟中断，`scause` 的最高位（Interrupt 位）为 1，Cause 值为 5 (`IRQ_S_TIMER`)。
2. **更新状态**：
   - 将 `sstatus` 寄存器中当前的 `SIE` 位（此时为 1）的值复制到 `SPIE` (Supervisor Previous Interrupt Enable) 位。
   - 将 `sstatus` 寄存器中的 `SIE` 位清零（即**自动关闭中断**），防止在处理中断入口时被新的中断打断。
   - 记录当前特权级（S 模式或 U 模式）到 `sstatus` 的 `SPP` (Supervisor Previous Privilege) 位。
3. **跳转**：
   - 将 PC 设置为 `stvec` 寄存器中保存的地址，即 `__alltraps`。

2.3. 阶段三：汇编入口与上下文保存 (`trapentry.S`)

执行流跳转到 `kern/trap/trapentry.S` 中的 `__alltraps`：

1. `SAVE_ALL` **宏**：
   - 首先在栈上分配一个 `trapframe` 结构体的空间（`addi sp, sp, -36 * REGBYTES`）。
   - 将全部 32 个通用寄存器（x0-x31）保存到栈上的 `trapframe` 相应位置。
   - 通过 `csrr` 指令读取 `sstatus`, `sepc`, `sbadaddr`, `scause` 这几个 CSRs，并将它们也保存到 `trapframe` 中。
2. **调用 C 处理函数**：
   - 执行 `move a0, sp`，将当前栈指针（即 `trapframe` 结构体的地址）放入 `a0` 寄存器。
   - 根据 RISC-V 调用约定， `a0` 是 C 函数的第一个参数。
   - 执行 `jal trap`，跳转到 `kern/trap/trap.c` 中定义的 `trap` C 函数，并将 `trapframe` 的指针作为参数传递。

2.4. 阶段四：C 语言分发与处理 (`trap.c`)

1. `trap(struct trapframe \*tf)`：
   - `trap` 函数接收到 `trapframe` 指针 `tf`。
   - 它调用 `trap_dispatch(tf)` 进行分发。
2. `trap_dispatch(tf)`：
   - 此函数检查 `tf->cause` 的最高位。如果为 1（即 `(intptr_t)tf->cause < 0`），说明是中断（Interrupt）；否则是异常（Exception）。
   - 对于时钟中断， `tf->cause` 为负，因此调用 `interrupt_handler(tf)`。
3. `interrupt_handler(tf)`：
   - 函数首先从 `tf->cause` 中提取中断号（`cause = (tf->cause << 1) >> 1`）。
   - `switch (cause)` 语句跳转到 `case IRQ_S_TIMER:`。
   - **执行我们的代码**：

1. `clock_set_next_event()`：**立即**调用 SBI 服务设置**下一次**时钟中断。这是保证时钟中断能够周期性重复的关键。
2. `ticks += 1`：全局 `ticks` 计数器加 1。
3. `if (ticks % TICK_NUM == 0)`：检查是否满 100 次。
4. 如果是，`num` 加 1 并调用 `print_ticks()` 打印 "100 ticks"。
5. `if (num == 10)`：检查是否已打印 10 行。
6. 如果是，调用 `sbi_shutdown()`，系统将停止运行。

- `break` 语句执行，`interrupt_handler` 函数返回。

2.5. 阶段五：上下文恢复与返回 (`trapentry.S`)

1. **C 函数返回**：
   - `interrupt_handler` -> `trap_dispatch` -> `trap` 逐级返回。
   - 执行流回到 `trapentry.S` 中 `jal trap` 指令的下一条，即 `__trapret` 标签处。
2. `RESTORE_ALL` **宏**：
   - 从栈上的 `trapframe` 中，将 `sstatus` 和 `sepc` 的值加载回对应的 CSRs。
   - 将 `trapframe` 中保存的 32 个通用寄存器（x1-x31，x0 不需要）恢复到 CPU 寄存器中。
   - 最后，恢复 `sp` 寄存器，使栈顶指回中断发生之前的位置，等于"弹出了" `trapframe`。
3. `sret` **指令**：
   - 执行 `sret` (Supervisor Return) 特权指令。
   - 这是 `trap` 的逆过程。硬件自动执行：
     1. **恢复中断**：将 `sstatus.SPIE` 的值（在阶段二中保存的 1）复制回 `sstatus.SIE`，从而**重新使能中断。**
     2. **恢复特权级**：根据 `sstatus.SPP` 恢复到中断前的特权级（S 模式或 U 模式）。
     3. **返回**：将 `sepc` 寄存器中的地址（在阶段二中保存的被中断指令的地址）加载到 PC。

执行流又回到了被中断的地方，继续执行下一条指令。

3. 总结

1. **初始化**：内核设置 `stvec` 指向 `__alltraps`，并通过 `sbi_set_timer` 准备第一次中断。
2. **触发**：硬件自动保存 `sepc` / `scause`，关闭中断（`SIE=0`），并跳转到 `__alltraps`。
3. **处理**：汇编代码 `SAVE_ALL` 保存完整上下文（`trapframe`），调用 C 函数 `trap` -> `trap_dispatch` -> `interrupt_handler`。
4. **核心逻辑**：在 `case IRQ_S_TIMER` 中，我们**必须调用** `clock_set_next_event()` **重新准备下一次中断**来实现周期性，然后执行计数和打印逻辑。
5. **返回**：C 函数返回后，汇编代码 `RESTORE_ALL` 恢复上下文，`sret` 指令恢复 `pc` 和中断使能状态（`SIE=1`）。

运行结果如下所示：

```
Kernel executable memory footprint: 30KB
memory management: default_pmm_manager
physcial memory map:
  memory: 0x0000000008000000, [0x0000000080000000, 0x0000000087ffffff].
check_alloc_page() succeeded!
satp virtual address: 0xfffffffffc0206000
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
oslab@Edison:~/labcode/lab3$
```

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。

Q1: 描述ucore中处理中断异常的流程（从异常的产生开始）

ucore 处理中断异常的流程高度依赖于 RISC-V 架构，可概括为以下五个阶段，这在报告的练习一中已有详细分析：

1. **中断/异常触发（硬件）**：当事件发生（如时钟中断、`ecall` 指令、非法指令），CPU 硬件自动执行：
   - 保存返回地址到 `sepc`。
   - 在 `scause` 中记录原因。
   - 在 `stval`（即 `sbadaddr`）中记录相关值（如出错地址）。
   - 保存当前中断使能状态（`sstatus.SIE` -> `sstatus.SPIE`）并**关闭 S 模式中断**（`sstatus.SIE = 0`）。
   - 保存当前特权级（U/S 模式）到 `sstatus.SPP`。
   - 将 PC 跳转到 `stvec` 寄存器指向的地址，即 `__alltraps`。
2. **汇编入口与上下文保存（`trapentry.S: __alltraps`）**：
   - 执行 `SAVE_ALL` 宏，在内核栈上分配 `trapframe` 空间。
   - 将 *所有* 通用寄存器（x0-x31）和关键 CSRs（`sstatus`, `sepc`, `scause`, `stval`）保存到栈上的 `trapframe` 结构中。
3. **C 语言分发处理（`trap.c`）**：
   - 通过 `move a0, sp` 将 `trapframe` 的指针作为参数。
   - `jal trap` 调用 C 函数 `trap()`。
   - `trap()` 调用 `trap_dispatch()`，根据 `tf->cause` 的最高位判断是中断还是异常。
   - 分别调用 `interrupt_handler()` 或 `exception_handler()`。
   - `switch` 语句根据 `tf->cause` 的具体值执行相应的处理逻辑（例如我们实现的 `case IRQ_S_TIMER`）。
4. **C 语言处理返回**：

- C 代码处理完毕后，从 `interrupt_handler`/`exception_handler` 逐级返回到 `trap` 函数，最后返回到 `trapentry.S`。

5. **上下文恢复与中断返回（`trapentry.S: __trapret`）**：
   - 执行 `RESTORE_ALL` 宏，从栈上的 `trapframe` 中恢复 *除* `scause` *和* `stval` *之外* 的所有寄存器，特别是 `sepc` 和 `sstatus` 会被恢复到 CSRs 中。
   - 执行 `sret` 特权指令。硬件自动执行：
     - 恢复中断使能状态（`sstatus.SIE = sstatus.SPIE`）。
     - 恢复到先前的特权级（根据 `sstatus.SPP`）。
     - 将 PC 设置为 `sepc` 寄存器中的值，返回到被中断的指令继续执行。

Q2: `mov a0, sp` 的目的是什么?

目的是传递参数。 根据 RISC-V 的 C 语言调用约定（Calling Convention），`a0` 寄存器用于传递函数的**第一个参数**。 在 `__alltraps` 汇编代码中：

1. `SAVE_ALL` 宏执行完毕后，`sp`（栈指针）正指向刚刚在栈上创建并填充好的 `trapframe` 结构体的起始地址。
2. `move a0, sp` 这条指令就是将 `sp`（即 `trapframe` 的地址）复制到 `a0` 寄存器中。
3. 紧接着的 `jal trap` 指令会跳转到 C 函数 `void trap(struct trapframe *tf)`。 因此，`mov a0, sp` 的作用就是将 `trapframe` 的指针作为第一个参数传递给 `trap` 函数，使 C 代码能够通过 `tf` 指针访问和操作（例如读取 `tf->cause`）被保存的上下文。

Q3: `SAVE_ALL` 中寄存器保存在栈中的位置是什么确定的?

由 `kern/trap/trap.h` 中定义的 `struct trapframe` 结构体布局确定的。 `SAVE_ALL` 宏是 `trapentry.S` 中的汇编代码，它必须严格按照 `trap.h` 中 C 语言 `struct trapframe` 和 `struct pushregs` 的定义顺序，将寄存器**手动**存放到栈上的正确偏移位置。 例如，`struct pushregs` 中 `ra` (x1) 是第二个成员（偏移量为 1 * REGBYTES），`sp` (x2) 是第三个成员（偏移量为 2 * REGBYTES）。 `SAVE_ALL` 中的 `STORE x1, 1*REGBYTES(sp)` 和 `STORE s0, 2*REGBYTES(sp)`（`s0` 此时存有原始 `sp`）就精确地对应了这个 C 结构体布局。 这种 C 结构体和汇编代码之间的**硬编码约定**，确保了 C 代码（如 `trap` 函数）可以通过 `tf->gpr.ra` 或 `tf->cause` 等方式正确地访问到汇编代码保存的寄存器值。

Q4: 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，在这个设计中必须保存所有寄存器。 理由如下：

1. **统一入口的通用性**：`__alltraps` 是 ucore 中 *所有* S 模式中断和异常的唯一入口点。在刚进入 `__alltraps` 时，汇编代码无法预知即将调用的 C 语言处理函数（`trap` 及其后续调用）会使用或修改哪些寄存器。
2. **保证上下文完整恢复**：中断/异常可能发生在任何地方（内核代码或用户代码）。为了确保在中断处理完成后，`sret` 能够完美地恢复到被中断前的状态，*必须* 假设 C 语言处理函数可能会破坏（clobber）任何一个通用寄存器。
3. **遵循 C 调用约定**：根据 RISC-V 调用约定，`trap` 函数作为被调用者（callee），可以自由使用“调用者保存”（caller-saved）寄存器（如 `t0-t6`, `a0-a7`）。如果 `__alltraps`（作为调用者 caller）不保存它们，它们的值就会丢失。虽然 `trap` 函数 *应该* 保存和恢复“被调用者保存”（callee-saved）寄存器（如 `s0-s11`），但最简单、最安全的设计是 `SAVE_ALL` 统一保存所有寄

存器。 **总结**：为了保证 C 语言中断处理函数可以自由执行，并且确保被中断的上下文在 `sret` 后能被精确还原，最简单、最健壮的策略就是在进入 C 函数前保存所有通用寄存器。

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？

Q1: `csrw sscratch, sp`; `csrrw s0, sscratch, x0` 实现了什么操作，目的是什么?

这两句指令配合 `STORE s0, 2*REGBYTES(sp)`，共同实现了一个核心目的：**将在进入 `SAVE_ALL` 之前的 *原始* 栈指针 `sp` 保存到 `trapframe` 结构体的 `sp` 字段中**，并重置 `sscratch` 为 0。

分解操作如下：

1. `csrw sscratch, sp`
   - **操作**：`write CSR`。将当前 `sp` 寄存器（即进入中断时的内核栈顶）的值写入 `sscratch` (Supervisor Scratch Register) 寄存器。
   - **目的**：`sscratch` 此时充当了一个**临时变量**，暂存了 *原始的* `sp` 值。
2. `addi sp, sp, -36 * REGBYTES`
   - **操作**：在栈上为 `trapframe` 分配空间，`sp` 指针（栈顶）向低地址移动。此时 `sp` 寄存器的值已经 *改变*。
3. `csrrw s0, sscratch, x0`
   - **操作**：`read and write CSR` (原子操作)。
     1. **Read**：读取 `sscratch` 寄存器的值（即第 1 步保存的 *原始 sp*）并将其存入 `s0` 寄存器。
     2. **Write**：同时将 `x0` 寄存器（硬编码为 0）的值写入 `sscratch` 寄存器。
   - **目的**：
     1. 将 *原始 sp* 从 `sscratch` 转移到 `s0`，以便后续存入 `trapframe`。
     2. **恢复内核 `sscratch` 为 0 的约定**。如 `idt_init` 中注释所述，ucore 约定在 S 模式（内核态）下 `sscratch` 应为 0。这条指令高效地完成了这一重置。
4. `STORE s0, 2*REGBYTES(sp)`
   - **操作**：将 `s0` 寄存器（现在存有 *原始 sp*）的值，存入当前 `sp` 偏移 `2*REGBYTES` 的位置。
   - **目的**：根据 `struct pushregs` 的定义，这个位置（偏移量 2）正是 `sp` 字段。此操作最终完成了 *原始 sp* 的保存。

总结：我们不能在第 2 步之后直接 `STORE sp, 2*REGBYTES(sp)`，因为那存入的是 *错误的*、已经偏移过的 `sp`。因此，必须使用 `sscratch` 作为临时中转，来保存和传递 *原始的* `sp` 值。

Q2: `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?

意义在于**向 C 语言处理函数传递信息**，而不是为了"恢复"它们。

1. **CSR 的分类**：
   - **功能性 CSR (如 `sstatus`, `sepc`)**：这些是控制 CPU 状态和流程的寄存器。`sret` 指令 *需要* 依赖它们的值才能正确返回（恢复特权级、恢复中断使能、跳转到正确地址）。因此，它们**必**

须在 `RESTORE_ALL` 中被还原。
    - **信息性 CSR (如 `scause`, `stval/sbadaddr`)**：这些是硬件在触发中断时 *写入* 的只读（从软件角度看）信息。它们告诉操作系统 "发生了什么"（`scause` - 原因）和 "相关数据是什么"（`stval` - 如出错的地址）。
2. **Store 的意义**： `SAVE_ALL` 保存 `scause` 和 `stval` 的**唯一目的**，是把它们作为 `trapframe` 结构体的一部分（`tf->cause` 和 `tf->badvaddr`），**传递给 C 语言中断处理函数**（`trap_dispatch`）。 C 代码 *必须* 读取 `tf->cause` 来 `switch` 到正确的分支，也可能需要读取 `tf->badvaddr` 来处理缺页等异常。
3. **为什么不 Restore**： `scause` 和 `stval` 记录的是 *刚刚发生的* 这次中断的信息。当中断处理完成，准备 `sret` 返回时，这些信息就已经"过时"了。
    - **毫无意义**："恢复" 一个旧的 `scause` 值到 `scause` 寄存器中没有任何意义，被中断的代码根本不关心上一次中断的原因。
    - **有害**：如果强行在 `sret` 之前写 `scause`，反而可能会干扰下一次中断的诊断。
    - **硬件行为**： `sret` 指令 *只* 关心 `sstatus` 和 `sepc`，完全不使用 `scause` 或 `stval`。

总结：`scause` 和 `stval` 被 `STORE` 是为了**给 `trap` 函数看（Read）**； `sstatus` 和 `sepc` 被 `STORE` 是为了给 `trap` 函数看（Read），并且为了在 `RESTORE_ALL` 时**被写回（Write）**，以便 `sret` 指令能正确工作。

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”。

异常处理相关代码补充

首先，我们要清楚`tf->epc` 是结构体 `trapframe`（中断帧）中的一个字段，对应于 CPU 的 `sepc` 寄存器。其意义就是保存异常发生时的指令地址。当发生异常或中断时，RISC-V 硬件会自动把当前正在执行的指令地址保存到 `sepc`。

接下来，由于我们是异常指令会触发异常，所以当我们异常处理结束之后，我们需要将seqc寄存器指向下一条指令，避免一直在异常处理中。由此我们可以来补充异常处理代码。

trap.c

```
case CAUSE_ILLEGAL_INSTRUCTION:
             // 非法指令异常处理
             /* LAB3 CHALLENGE3   YOUR CODE : 张津硕 */
            /*(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 type: Illegal instruction\n");
            cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
            tf->epc += 4; // 跳过触发异常的那条指令
            break;
        case CAUSE_BREAKPOINT:
            //断点异常处理
            /* LAB3 CHALLLENGE3   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 type: breakpoint\n");

            cprintf("ebreak caught at 0x%08x\n", tf->epc);
            tf->epc += 4; // 同样跳过断点指令
            break;
```

添加相关异常指令

首先我们要知道如何添加相关的异常指令，可以通过内联汇编加入，也可以直接修改汇编。为了我们快速测试我们的异常处理，我们选择使用**内联汇编的方式（也就是直接在init.c中写入）**

`ebreak`：标准的 **断点指令**，执行时会产生 **Breakpoint Exception**（`scause = CAUSE_BREAKPOINT`）。用于调试/测试，行为确定。

`mret`：是 **特权返回指令**（从 Machine mode 返回），**在不适当的权限下执行会产生 Illegal Instruction**，所以常被用来触发 **Illegal Instruction** 来做测试（注意是特权指令，实际效果依执行上下文和特权级而异）。

综上，我们使用内联汇编的方式，以这两条指令作为我们添加的异常指令。

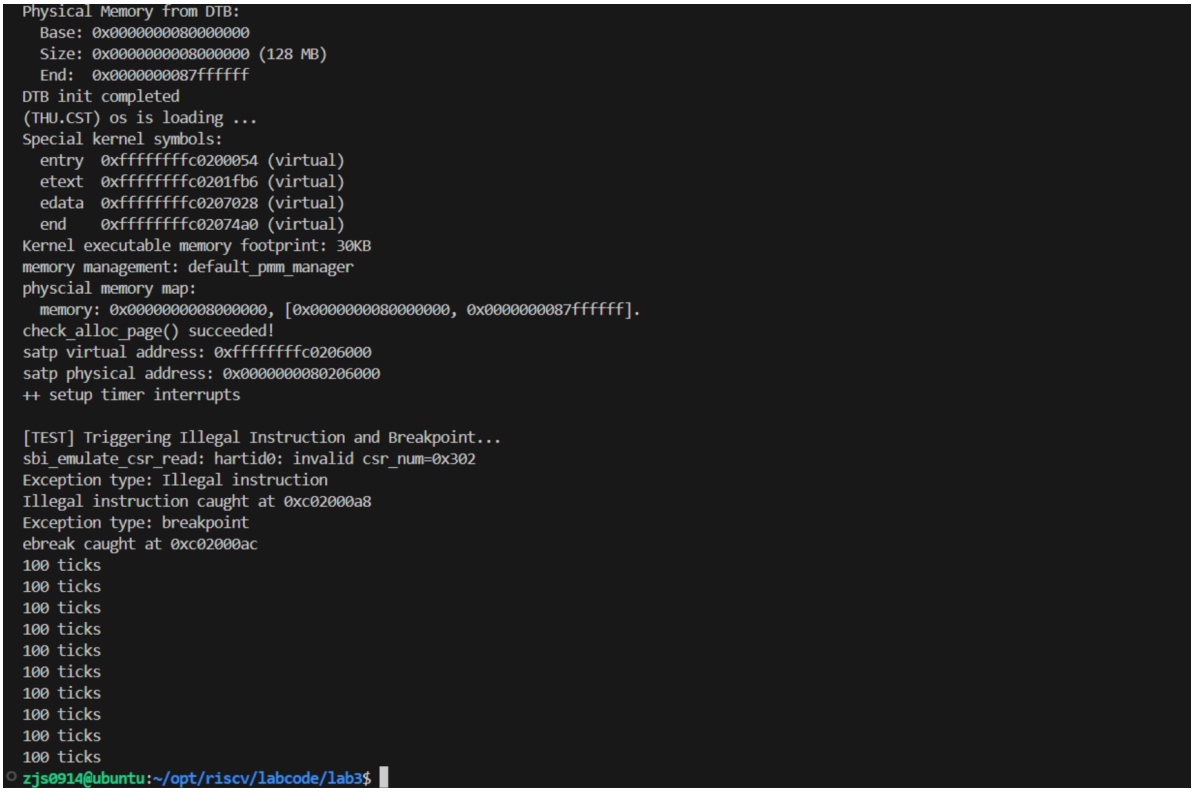
`asm`：标志内联汇编的关键字

`volatile`：告诉编译器"**这条指令有副作用，不能删、不能移走**"。否则在优化时编译器可能会把它折叠/消除，导致测试无效。

```
int kern_init(void) {
    extern char edata[], end[];
    // 先清零 BSS，再读取并保存 DTB 的内存信息，避免被清零覆盖（为了解释变化 正式上传时我觉得应该删去这句话）
    memset(edata, 0, end - edata);
    dtb_init();
    cons_init();  // init the console
    const char *message = "(THU.CST) os is loading ...\0";
    //cprintf("%s\n\n", message);
    cputs(message);

    print_kerninfo();

    // grade_backtrace();
    idt_init();  // init interrupt descriptor table

    pmm_init();  // init physical memory management

    idt_init();  // init interrupt descriptor table

    clock_init();   // init clock interrupt
    intr_enable();  // enable irq interrupt
// ======== Challenge 3: 测试异常处理 ========
    cprintf("\n[TEST] Triggering Illegal Instruction and Breakpoint...\n");

    // 触发非法指令异常（mret 在内核态不允许执行）
    asm volatile("mret");
```

```
    // 触发断点异常（ebreak）
    asm volatile(
     "ebreak\n\t"
     "nop\n\t"
     );
     /* do nothing */
     while (1)
         ;
}
```

我们的异常指令会在 C 程序的“顺序流”中执行，也就是在 `intr_enable()` （开启终端响应能力）之后执行异常指令。



我们完成了异常处理的逻辑。

一些问题

在实验过程中，我们发现，当我们使用`ebreak`指令之后，进入到了异常处理的程序后，我们明明添加了`tf->epc += 4`;也就是进入到下一条指令，但是结果却是，我们下一条指令是不知名的异常，进入了default的那个分支，后来我在`ebreak`指令之后添加了一个空指令，解决了这个问题，为什么会出现这个问题，查询资料未能找到一些结果。

后续询问助教已解决！

知识点整理

实验中的重要知识点与OS原理对应

1. 异常向量与特权级切换（`stvec`/`sepc`/`scause`）

- 内核在 S 态设置 `stvec` 指向 `__alltraps`（direct 模式），一旦U/S态发生异常或中断，硬件跳到该入口，并把**返回地址**放到 `sepc`、把**原因**放到 `scause`。
- 这奠定了“统一入口→再分发”的处理范式，是一切 trap 处理的门槛。
- `stvec` 必须在开中断前就位；direct 模式低 2 位为 0；`sepc` 指向**触发指令**，后续是否 `epc += 4` 决定是否“越过”该指令。

2. 上下文保存/恢复与栈切换（`trapframe`/`sscratch`/`sret`）

- `__alltraps` 保存 x1~x31 与关键 CSR（`sstatus/sepc/scause/stval`）到 `trapframe`，必要时用 `sscratch` 交换/暂存用户栈指针，切到内核栈；C 层处理后在 `__trapret` 中按**同顺序**恢复并 `sret`。
- 保证被打断的执行流“可恢复”。“保存—恢复”的**对称性**决定系统稳不稳。
- 保存/恢复顺序必须与 `struct trapframe` 完全一致；`sstatus.SPIE/SIE` 的保存与恢复决定返回时中断位状态；RISC-V ABI 下寄存器约定不可随意破坏。

3. 中断分发与时钟中断（ticks/HZ 与 SBI 定时器）

- `trap_dispatch` 解析 `scause`，识别 **S 模式定时器中断**，在处理函数里 `ticks++` 并调用 `clock_set_next_event()`（底层 `sbi_set_timer()`）预约下一拍。
- ： 时钟中断提供“时间片”与心跳，是调度、超时与统计的基石。
- 需同时打开 `sie.STIE`（源使能）与 `sstatus.SIE`（全局使能）； `HZ` 与 timebase 统一单位，避免节拍漂移

4. 异常处理与可恢复策略（`Illegal`/`Breakpoint`/`ecall`）

- 对典型异常打印 `scause/stval/sepc`，并对可恢复场景执行 `tf->epc += 4` 跳过触发指令（如断点、演示用非法指令）， `ecall` 还要遵循 ABI 取参与返回。
- 区分“致命”与“可恢复”的异常，决定是 `panic()` 还是继续运行。
- `stval` 在地址类异常里是故障地址； `ecall` 需要规范约定（参数寄存器与返回值寄存器）。

OS原理中重要但实验未涉及的知识点

1. 进程/线程与调度算法（从 trap 到可抢占调度）

- PCB/TCB、上下文切换、就绪队列、时间片轮转/优先级/MLFQ、可抢占/不可抢占、实时（EDF/RM）。
- 本实验没有真正切换进程/线程，也无调度策略与就绪队列。

2. 高级内存管理与缺页异常（VM/TLB/置换）

- 多级页表、TLB、页面权限与保护、缺页异常、页面置换（LRU/Clock）、写时复制（COW）。
- 我们只保存了 `stval`/`sepc`，未实现缺页处理与页表填充，也没有换页与回收策略。

3. 并发同步与异常处理

- 多个执行流同时访问共享资源时，通过关中断或锁等机制保持一致性。常见的同步手段有关中断（最原始的方法），缺点是会影响系统响应速度，现代系统只在极短的关键区使用。现代操作系统用锁（Lock）机制，保证同一时间只有一个线程进入关键区。
- 本实验只演示了“关/开中断”，未涉及真正的锁、延迟执行与多核并行的并发控制。