中断与中断处理机制

操作系统 lab3 实验报告

梁景铭　　王一诺　　强博
南开大学计算机学院

2025 年 11 月 3 日

摘要

本报告详细阐述了 ucore 操作系统实验三（Lab 3）中关于 RISC-V 架构的中断与异常处理机制。实验的核心是构建了从硬件陷入（trap）到 C 语言处理函数的完整流程。这包括：深入学习 S 模式下的中断委托、关键 CSR 寄存器（如 `stvec`, `sepc`, `scause`）及特权指令；编写汇编入口（`__alltraps`），通过 `SAVE_ALL` / `RESTORE_ALL` 宏实现完整的上下文（`TrapFrame`）保存与恢复；设计 C 语言分发函数（`trap_dispatch`）以根据 `scause` 路由中断和异常。

在练习中，报告实现了对 S 模式时钟中断（`IRQ_S_TIMER`）的处理，通过调用 SBI 设置定时器，完成了周期性打印与定时关机功能；并实现了对非法指令和断点异常的处理，通过修改 `sepc` 跳过异常指令。报告还深入分析了上下文切换中 `sscratch` 寄存器的使用、保存 `scause` 等只读 CSR 的目的，以及处理 RISC-V 压缩指令（c.ebreak）时遇到的 `sepc` 偏移问题。

目录

1 实验内容与目标

1.1 实验内容概述

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。

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

2 ucore 实验文档学习

2.1 riscv64 中断介绍

2.1.1 中断概念

中断机制　中断机制，就是不管 CPU 现在手里在干啥活，收到“中断”的时候，都先放下来去处理其他事情，处理完其他事情可能再回来干手头的活。

例如，CPU 要向磁盘发一个读取数据的请求，由于磁盘速度相对 CPU 较慢，在“发出请求”到“收到磁盘数据”之间会经过很多时间周期，如果 CPU 干等着磁盘干活就相当于 CPU 在磨洋工。因此我们可以让 CPU 发出读数据的请求后立刻开始干另一件事情。但是，等一段时间之后，磁盘的数据取到了，而 CPU 在干其他的事情，我们怎么办才能让 CPU 知道之前发出的磁盘请求已经完成了呢？我们可以让磁盘给 CPU 一个“中断”，让 CPU 放下手里的事情来接受磁盘的数据。

再比如，为了保证 CPU 正在执行的程序不会永远运行下去，我们需要定时检查一下它是否已经运行“超时”。想象有一个程序由于 bug 进入了死循环，如果 CPU 一直运行这个程序，那么其他的所有程序都会因为等待 CPU 资源而无法运行，造成严重的资源浪费。但是检查是否超时，需要 CPU 执行一段代码，也就是让 CPU 暂停当前执行的程序。我们不能假设当前执行的程序会主动地定时让出 CPU，那么就需要 CPU 定时“打断”当前程序的执行，去进行一些处理，这通过时钟中断来实现。

从这些描述我们可以看出，中断机制需要软件硬件一起来支持。硬件进行中断和异常的发现，然后交给软件来进行处理。回忆一下组成原理课程中学到的各个控制寄存器以及他们的用途，这些寄存器构成了重要的硬件/软件接口。由此，我们也可以得到在一般 OS 中进行中断处理支持的方法：

1. 编写相应的中断处理代码
2. 在启动中正确设置控制寄存器

3. CPU 捕获异常
4. 控制转交给相应中断处理代码进行处理
5. 返回正在运行的程序

2.1.2 Trap 分类

在 RISC-V 架构中，`Trap` 是一个总称，指代任何导致控制权从当前执行流转移到陷阱处理程序的事件。Trap 分为两类：

- **中断**：异步事件，由外部设备触发，与当前执行的指令无关。例如：定时器中断、磁盘 I/O 完成中断。
- **异常**：同步事件，由当前执行的指令触发。例如：访问无效内存地址、执行非法指令、缺页异常，以及执行 `ecall` 和 `ebreak` 指令。

由于中断处理需要进行较高权限的操作，中断处理程序一般处于内核态，或者说，处于” 比被打断的程序更高的特权级”。注意，在 RISCV 里，中断和异常统称为 trap。

须知 在传统操作系统中，我们通常将 Trap 狭义地定义为主动触发的异常，如系统调用（通过 `ecall` 指令）和断点（通过 `ebreak` 指令）。而在 RISC-V 架构中，Trap 被用作所有控制转移事件（包括中断和异常）的总称。此外，传统分类中的 **内部中断（软件中断）** 大致对应 RISC-V 中的异常，而 **外部中断（硬件中断）** 对应 RISC-V 中的中断。请注意这一术语差异。

2.1.3 riscv64 权限模式

我们在 lab1 中简单的提及过特权级，现在我们具体介绍 RISC-V 的三个特权级。在现代处理器中，系统通常会区分 **用户态（User Mode）** 和 **特权态（Supervisor/Kernel Mode）**。用户态运行应用程序，权限有限，不能直接访问硬件和关键寄存器。特权态运行操作系统内核，可以控制硬件、管理内存和调度任务。这种分层的目的，是保证系统的安全与稳定：用户程序即使出现错误，也不会直接破坏底层系统。中断与异常机制负责完成这两种模式之间的切换。例如，当用户态程序发起系统调用，或出现异常/中断时，处理器会切换到特权态，交由内核代码进行处理，处理完成后再返回用户态继续执行。

RISC-V 的三个特权级：

- **M 模式（Machine Mode）**：是 RISC-V 中最高的特权级。在 M 模式下，hart（硬件线程）对内存、I/O 和底层功能有完全的控制权。默认情况下，所有的中断和异常都会首先进入 M 模式处理。M 模式是所有标准 RISC-V 处理器必须实现的特权级，通常用于处理器启动、底层固件（如 OpenSBI）的执行。
- **S 模式（Supervisor Mode）**：是支持现代类 Unix 操作系统的特权级。操作系统内核运行在 S 模式，它支持基于页面的虚拟内存机制，使得多任务和内存隔离成为可能。在 Unix 系统中，大多数系统调用和异常都在 S 模式下处理。

- **U 模式（User Mode）**：是最低的特权级。用户程序运行在 U 模式，只能执行普通指令，不能直接访问硬件或执行特权操作。当用户程序需要操作系统服务时，必须通过系统调用（`ecall` 指令）陷入到 S 模式。

2.1.4 中断的路由：从默认路径到委托机制

在 RISC-V 架构中，中断与异常是构建一个健壮、可响应系统的基石。理解其处理机制，关键在于把握一个核心事实：中断或异常可以发生在处理器执行时的任何特权级。这引出了我们首要解答的问题：一个发生在特定特权级的事件，最终会由哪个特权级的代码来处理？答案并非一成不变，而是由 RISC-V 灵活的中断委托机制与系统软件的设计共同决定的。

默认情况下，RISC-V 遵循最保守的安全设计：所有中断与异常都会首先陷入最高特权级——M 模式。M 模式作为硬件的直接管理者，拥有最终控制权，此设计确保了系统的底层安全。然而，对于运行现代操作系统的场景，让所有中断（例如来自用户态的系统调用或时钟中断）都经由 M 模式处理，会带来不必要的性能开销和灵活性限制。操作系统内核期望能直接管理属于自己的事务。

为解决此问题，RISC-V 引入了中断委托机制。M 模式可以通过设置两个关键寄存器，将特定的中断与异常处理权” 下放” 给 S 模式：

- `mideleg`：负责委托中断。例如，将软件中断、定时器中断或外部中断委托给 S 模式。
- `medeleg`：负责委托异常。例如，将用户态环境调用、非法指令、页错误等异常委托给 S 模式。

在 ucore 启动时，OpenSBI 固件会进行初始化，将绝大部分 S 模式与 U 模式相关的中断和异常委托出去。这意味着，在委托发生后，中断的处理路由发生了根本性变化。现在，我们可以分场景审视一个中断的完整流程：

- **U 模式触发中断**：这是最常见的情况。当用户程序执行 `ecall` 指令发起系统调用，或执行指令时发生页错误，该异常会被 `medeleg` 委托。硬件将直接陷入到 S 模式，而不再经过 M 模式。这是操作系统为用户提供服务的主要通道。
- **S 模式触发中断**：内核自身在运行时也可能遇到两种性质不同的陷阱。第一种是被动陷入，例如设备 I/O 完成产生的外部中断，或内核代码触发的缺页异常（内核本身一般不引发缺页异常，但是在尝试访问用户空间传入的数据的时候，也可能触发）。如果该中断类型已被委托，则处理流程为 S 模式陷入到 S 模式——这被称为“自陷”。第二种是主动陷入，当内核需要访问硬件特权功能（如设置定时器）时，它会主动执行 `ecall` 指令。此时，由于 `ecall` 在 S 模式下执行，特权级将从 S 模式提升至 M 模式，由 M 模式的固件（如 OpenSBI）提供服务，处理完毕后再通过 `mret` 指令返回 S 模式。
- **M 模式触发中断**：某些与最底层硬件管理紧密相关的中断，例如 M 模式的定时器中断或某些安全性事件，通常不会被委托。它们始终在 M 模式内处理，由固件负责，这与操作系统的常规运行无关。

2.1.5 S 模式的中断处理框架

一旦中断被路由到 S 模式，处理器便需要知道如何开始执行处理代码。实际上，RISCV 架构有个 CSR 叫做 `stvec`，即所谓的中断向量表基址。中断向量表的作用就是把不同种类的中断映射到对应的中断处理程序。如果只有一个中断处理程序，那么可以让 `stvec` 直接指向那个中断处理程序的地址。

扩展 `stvec` 会把最低位的两个二进制位用来编码一个"模式", 如果是 00 就说明更高的 SXLEN-2 个二进制位存储的是唯一的中断处理程序的地址 (SXLEN 是 `stval` 寄存器的位数)，如果是 01 说明更高的 SXLEN-2 个二进制位存储的是中断向量表基址，通过不同的异常原因来索引中断向量表。但是怎样用 62 个二进制位编码一个 64 位的地址？RISCV 架构要求这个地址是四字节对齐的，总是在较高的 62 位后补两个 0。

在旧版本的 RISCV privileged ISA 标准中（1.9.1 及以前），RISCV 不支持中断向量表，用最后两位数编码一个模式是 1.10 版本加入的。可以思考一下这个改动如何保证了后向兼容。历史版本的 ISA 手册

1.9.1 版本的 RISCV privileged architecture 手册： 当我们触发中断，进入 S 模式进行处理前，硬件会自动扮演一名忠实的"现场记录员"，为处理程序准备好一份详尽的上下文报告。这份报告由三个关键寄存器构成：

- `sepc`：它自动保存了被中断指令的虚拟地址。这份记录回答了"事发时程序执行到了哪里?"的问题，是未来恢复执行的关键。
- `scause`：它记录了一个编码，精确指出了中断或异常的具体原因。例如，是用户态的系统调用，还是指令页错误？或者是外部设备中断？处理程序通过查阅此寄存器来辨别事件性质。
- `stval`：它提供了与异常相关的附加信息，是重要的"现场证据"。当发生缺页异常时，`stval` 会存放导致失败的访存地址；当遇到非法指令时，它可能会记录该指令本身的内容。

凭借这份由硬件自动生成的报告，处理程序便能清晰地了解现场情况，从而做出正确的响应。

2.1.6 使能与屏蔽

在 S 模式处理中断时，一个关键的设计抉择是中断的使能状态。有些时候，禁止 CPU 产生中断很有用，操作系统内核通常希望在执行中断处理程序的过程中，不会被新的中断所打断，就像你在做重要的事情，如操作系统 lab 的时候，并不想被打断。这保证了内核数据结构的修改等关键操作具有原子性。

所以，`sstatus` 寄存器里面有一个二进制位 `SIE`，数值为 0 的时候，如果当程序在 S 态运行，将禁用全部中断。（对于在 U 态运行的程序，`SIE` 这个二进制位的数值没有任何意义），`sstatus` 还有一个二进制位 `UIE` 可以在置零的时候禁止用户态程序产生中断。

实际上，`sstatus` 寄存器中还包含多个重要的控制位，比如 `SPIE` 会记录进入 S 模式之前的 `SIE` 值。当通过 `trap` 进入 S 模式时，硬件会将 `SPIE` 设置为 `SIE` 的旧值，然后将 `SIE` 清零。在

从 S 模式返回时，硬件会将 SIE 恢复为 SPIE 的值，然后将 SPIE 设置为 1。再比如 SPP 会记录进入 S 模式之前的特权级。0 表示来自 U 模式，1 表示来自 S 模式。从 S 模式返回时会根据 SPP 的值决定返回到哪个特权级。

除了 sstatus 寄存器中的 SIE 控制位，还存在一个 sie 寄存器，即使 sstatus.SIE 为 1，也可以通过 sie 寄存器屏蔽特定类型的中断。例如，可以只允许时钟中断，而屏蔽软件中断。

2.1.7 特权指令

在整个中断的生命周期中，特权指令扮演着流程控制器的角色。RISCV 支持以下和中断相关的特权指令：

- ecall：这是主动发起 trap 的指令。它的行为高度依赖于执行它的当前特权级。在 U 模式下执行 ecall，会触发一个 ecall-from-u-mode-exception，这是应用程序请求操作系统服务的标准方式，路径为 U → S。在 S 模式下执行 ecall，则会触发 ecall-from-s-mode-exception，这通常用于内核向 M 模式固件（如 OpenSBI）请求服务，路径为 S → M。它本质上是提升特权级的工具。

- sret：这是从 S 模式返回的指令。它通常用于在完成 U 模式触发的异常处理后，从 S 模式返回 U 模式。它会同时恢复 sepc 和 sstatus.SIE 的状态，将 sepc 赋给 pc，并将 SIE 恢复为 SPIE 的值。

- mret：与 sret 类似，但用于从 M 模式陷阱中返回。当 M 模式处理完中断，或 S 模式通过 ecall 请求的服务完成后，使用 mret 可以从 M 模式返回 S 模式。

- ebreak：这条指令用于触发一个断点异常，通常用于调试。它也会导致控制流跳转到 stvec 指定的中断处理程序，但其目的并非服务请求，而是调试目的。

2.1.8 从 U 模式陷入 S 模式的完整流程

我们最常接触到的便是 U 模式陷入到 S 模式。当一个用户程序运行在 U 模式时，它通常在执行过程中有可能因为某些原因陷入 S 模式。运行在 U 模式的程序如果需要向操作系统请求服务，它会通过执行 ecall 指令发起系统调用，主动将控制权交给内核。同时，定时器也会在指定时间触发时钟中断，这会导致操作系统对当前进程进行时间片轮转调度，决定是否切换进程。另外如果程序发生了异常，比如访问非法内存地址、发生缺页等，系统也会转入内核进行处理。这些事件都触发了相关处理流程，将程序的运行状态从 U 模式转入 S 模式。

首先，保存中断发生时的 pc 值，即程序计数器的值，这个值会被保存在 sepc 寄存器中。对于异常来说，这通常是触发异常的指令地址，而对于中断来说，则是被打断的指令地址。然后，记录中断或异常的类型，并将其写入 scause 寄存器。这里的 scause 会告诉系统是中断还是异常，且会给出具体的中断类型。

接下来，保存相关的辅助信息。如果异常与缺页或访问错误相关，将相关的地址或数据保存到 stval 寄存器，以便中断处理程序在后续处理中使用。紧接着，保存并修改中断使能状态。将当前的中断使能状态 sstatus.SIE 保存到 sstatus.SPIE 中，并且会将 sstatus.SIE 清零，从而禁用 S 模式下的中断。这是为了保证在处理中断时不会被其他中断打断。

然后，保存当前的特权级信息。将当前特权级（即 U 模式，值为 0）保存到 `sstatus.SPP` 中，并将当前特权级切换到 S 模式。此时，系统已经进入 S 模式，准备跳转到中断处理程序。将 `pc` 设置为 `stvec` 寄存器中的值，并跳转到中断处理程序的入口。

进入中断处理程序后，系统会执行一系列处理步骤，下面我们就去看看在做些什么。

2.2 中断入口点

2.2.1 掉进兔子洞 (中断入口点)

在前面我们已经了解了 RISC-V 中断机制的基本原理：当中断发生时，会保存 `sepc`、设置 `scause`、跳转到 `stvec` 指定的地址等。我们也知道了从 U 模式到 S 模式的完整切换流程。现在我们来解决如何实现中断入口点，以及完成完整的上下文切换。

首先，我们需要初始化 `stvec` 寄存器。我们采用 Direct 模式，也就是 `stvec` 直接指向唯一的中断处理程序入口点，所有类型的中断和异常都会跳转到这里。

中断的处理需要放下当前的事情但之后还能回来接着之前往下做，对于 CPU 来说，实际上只需要把原先的寄存器保存下来，做完其他事情把寄存器恢复回来就可以了。这些寄存器也被叫做 CPU 的 context(上下文，情境)。我们要用汇编实现上下文切换 (context switch) 机制，这包含两步：

1. 保存 CPU 的寄存器（上下文）到内存中（栈上）
2. 从内存中（栈上）恢复 CPU 的寄存器

为了方便我们组织上下文的数据（几十个寄存器），我们定义一个结构体。`sscratch` 寄存器在处理用户态程序的中断时才起作用。在目前其实用处不大。

须知：RISCV 汇编的通用寄存器别名和含义 *The RISC-V Instruction Set Manual Volume I: Unprivileged ISA Document Version 20191213 Chapter 25 RISC-V Assembly Programmer's Handbook*

kern/trap/trap.h

```
#ifndef __KERN_TRAP_TRAP_H__
#define __KERN_TRAP_TRAP_H__

#include <defs.h>

struct pushregs {
    uintptr_t zero;  // 硬编码的零
    uintptr_t ra;    // 返回地址
    uintptr_t sp;    // 栈指针
    uintptr_t gp;    // 全局指针
    uintptr_t tp;    // 线程指针
    uintptr_t t0;    // 临时寄存器
    uintptr_t t1;    // 临时寄存器
    uintptr_t t2;    // 临时寄存器
    uintptr_t s0;    // 保存寄存器/帧指针
```

表 1: RISC-V 寄存器 ABI 别名与含义

寄存器	ABI 别名	描述	保存者
x0	zero	硬编码的零	—
x1	ra	返回地址	调用者
x2	sp	栈指针	被调用者
x3	gp	全局指针	—
x4	tp	线程指针	—
x5	t0	临时寄存器/备用链接寄存器	调用者
x6–7	t1–2	临时寄存器	调用者
x8	s0/fp	保存寄存器/帧指针	被调用者
x9	s1	保存寄存器	被调用者
x10–11	a0–1	函数参数/返回值	调用者
x12–17	a2–7	函数参数	调用者
x18–27	s2–11	保存寄存器	被调用者
x28–31	t3–6	临时寄存器	调用者

```
    uintptr_t s1;    // 保存寄存器
    uintptr_t a0;    // 函数参数/返回值
    uintptr_t a1;    // 函数参数/返回值
    uintptr_t a2;    // 函数参数
    uintptr_t a3;    // 函数参数
    uintptr_t a4;    // 函数参数
    uintptr_t a5;    // 函数参数
    uintptr_t a6;    // 函数参数
    uintptr_t a7;    // 函数参数
    uintptr_t s2;    // 保存寄存器
    uintptr_t s3;    // 保存寄存器
    uintptr_t s4;    // 保存寄存器
    uintptr_t s5;    // 保存寄存器
    uintptr_t s6;    // 保存寄存器
    uintptr_t s7;    // 保存寄存器
    uintptr_t s8;    // 保存寄存器
    uintptr_t s9;    // 保存寄存器
    uintptr_t s10;   // 保存寄存器
    uintptr_t s11;   // 保存寄存器
    uintptr_t t3;    // 临时寄存器
    uintptr_t t4;    // 临时寄存器
    uintptr_t t5;    // 临时寄存器
    uintptr_t t6;    // 临时寄存器
};

```

```
41 struct trapframe {
42     struct pushregs gpr;
43     uintptr_t status; // sstatus 寄存器
44     uintptr_t epc; // sepc 寄存器
45     uintptr_t badvaddr; // sbadvaddr 寄存器
46     uintptr_t cause; // scause 寄存器
47 };
48
49 void trap(struct trapframe *tf);
50
51 #endif /* !__KERN_TRAP_TRAP_H__ */
```

C 语言里面的结构体，是若干个变量在内存里直线排列。也就是说，一个 trapframe 结构体占据 36 个 uintptr_t 的空间（在 64 位 RISCV 架构里我们定义 uintptr_t 为 64 位无符号整数），里面依次排列通用寄存器 x0 到 x31, 然后依次排列 4 个和中断相关的 CSR, 我们希望中断处理程序能够利用这几个 CSR 的数值。

首先我们定义一个汇编宏 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）。

kern/trap/trapentry.S (SAVE_ALL)

```
1
2
3  #include <riscv.h>
4
5      .macro SAVE_ALL #定义汇编宏
6
7      csrw sscratch, sp #保存原先的栈顶指针到sscratch
8
9      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
10     #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。
11     #除了32个通用寄存器, 我们还要保存4个和中断有关的CSR
12
13     #依次保存32个通用寄存器。但栈顶指针需要特殊处理。
14     #因为我们想在trapFrame里保存分配36个REGBYTES之前的sp
15     #也就是保存之前写到sscratch里的sp的值
16     STORE x0, 0*REGBYTES(sp)
17     STORE x1, 1*REGBYTES(sp)
18     STORE x3, 3*REGBYTES(sp)
19     STORE x4, 4*REGBYTES(sp)
20     STORE x5, 5*REGBYTES(sp)
21     STORE x6, 6*REGBYTES(sp)
22     STORE x7, 7*REGBYTES(sp)
23     STORE x8, 8*REGBYTES(sp)
```

```
24      STORE x9, 9*REGBYTES(sp)
25      STORE x10, 10*REGBYTES(sp)
26      STORE x11, 11*REGBYTES(sp)
27      STORE x12, 12*REGBYTES(sp)
28      STORE x13, 13*REGBYTES(sp)
29      STORE x14, 14*REGBYTES(sp)
30      STORE x15, 15*REGBYTES(sp)
31      STORE x16, 16*REGBYTES(sp)
32      STORE x17, 17*REGBYTES(sp)
33      STORE x18, 18*REGBYTES(sp)
34      STORE x19, 19*REGBYTES(sp)
35      STORE x20, 20*REGBYTES(sp)
36      STORE x21, 21*REGBYTES(sp)
37      STORE x22, 22*REGBYTES(sp)
38      STORE x23, 23*REGBYTES(sp)
39      STORE x24, 24*REGBYTES(sp)
40      STORE x25, 25*REGBYTES(sp)
41      STORE x26, 26*REGBYTES(sp)
42      STORE x27, 27*REGBYTES(sp)
43      STORE x28, 28*REGBYTES(sp)
44      STORE x29, 29*REGBYTES(sp)
45      STORE x30, 30*REGBYTES(sp)
46      STORE x31, 31*REGBYTES(sp)
47      # 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
48      csrrw s0, sscratch, x0
49      csrr s1, sstatus
50      csrr s2, sepc
51      csrr s3, sbadaddr
52      csrr s4, scause
53
54      STORE s0, 2*REGBYTES(sp)
55      STORE s1, 32*REGBYTES(sp)
56      STORE s2, 33*REGBYTES(sp)
57      STORE s3, 34*REGBYTES(sp)
58      STORE s4, 35*REGBYTES(sp)
59      .endm #汇编宏定义结束
60  \end{verbatim}
61
62  然后是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。
63
64  \paragraph{kern/trap/trapentry.S (RESTORE\_ALL)}
65  \begin{verbatim}
66  .macro RESTORE_ALL
```

```
67
68  LOAD s1, 32*REGBYTES(sp)
69  LOAD s2, 33*REGBYTES(sp)
70
71  # 注意之前保存的几个CSR并不都需要恢复
72  csrw sstatus, s1
73  csrw sepc, s2
74
75  # 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
76  LOAD x1, 1*REGBYTES(sp)
77  LOAD x3, 3*REGBYTES(sp)
78  LOAD x4, 4*REGBYTES(sp)
79  LOAD x5, 5*REGBYTES(sp)
80  LOAD x6, 6*REGBYTES(sp)
81  LOAD x7, 7*REGBYTES(sp)
82  LOAD x8, 8*REGBYTES(sp)
83  LOAD x9, 9*REGBYTES(sp)
84  LOAD x10, 10*REGBYTES(sp)
85  LOAD x11, 11*REGBYTES(sp)
86  LOAD x12, 12*REGBYTES(sp)
87  LOAD x13, 13*REGBYTES(sp)
88  LOAD x14, 14*REGBYTES(sp)
89  LOAD x15, 15*REGBYTES(sp)
90  LOAD x16, 16*REGBYTES(sp)
91  LOAD x17, 17*REGBYTES(sp)
92  LOAD x18, 18*REGBYTES(sp)
93  LOAD x19, 19*REGBYTES(sp)
94  LOAD x20, 20*REGBYTES(sp)
95  LOAD x21, 21*REGBYTES(sp)
96  LOAD x22, 22*REGBYTES(sp)
97  LOAD x23, 23*REGBYTES(sp)
98  LOAD x24, 24*REGBYTES(sp)
99  LOAD x25, 25*REGBYTES(sp)
100 LOAD x26, 26*REGBYTES(sp)
101 LOAD x27, 27*REGBYTES(sp)
102 LOAD x28, 28*REGBYTES(sp)
103 LOAD x29, 29*REGBYTES(sp)
104 LOAD x30, 30*REGBYTES(sp)
105 LOAD x31, 31*REGBYTES(sp)
106 # 最后恢复sp
107 LOAD x2, 2*REGBYTES(sp)
108 .endm
```

现在我们编写真正的中断入口点

kern/trap/trapentry.S (__alltraps)

```
1
2
3     .globl __alltraps
4
5  .align(2) #中断入口点 __alltraps必须四字节对齐
6  __alltraps:
7     SAVE_ALL #保存上下文
8
9     move  a0, sp #传递参数。
10    #按照RISCV calling convention, a0寄存器传递参数给接下来调用的函数trap。
11    #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
12    jal trap
13    #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
14
15    .globl __trapret
16 __trapret:
17    RESTORE_ALL
18    # return from supervisor call
19    sret
```

我们可以看到，trapentry.S 作用是一个” 包装器”：它负责保存和恢复上下文，并把上下文包装成结构体，传递给真正的中断处理函数 trap 那里去。

在上面的代码中，我们看到最后一条指令是 sret。这是一条特权指令，用于从 S 模式返回，完成从内核态到用户态的切换。

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。

接下来，我们将详细介绍 trap 函数的实现，看看如何处理各种中断和异常。

2.3 中断处理程序

中断处理需要初始化，所以我们在 init.c 里调用一些初始化的函数。

kern/init/init.c

```
1 #include <trap.h>
```

```
2
3  int kern_init(void) {
4      extern char edata[], end[];
5      memset(edata, 0, end - edata);
6      cons_init();  // init the console
7      const char *message = "(THU.CST) os is loading ...\0";
8      //cprintf("%s\n\n", message);
9      cputs(message);
10     print_kerninfo();
11     // grade_backtrace();
12     idt_init();  // init interrupt descriptor table
13     pmm_init();  // init physical memory management
14     idt_init();  // init interrupt descriptor table
15     clock_init();   // init clock interrupt
16     intr_enable();  // enable irq interrupt
17     // LAB3: CAHLLENGE 1 If you try to do it, uncomment lab3_switch_test()
18     // user/kernel mode switch test
19     // lab3_switch_test();
20
21     /* do nothing */
22     while (1)
23         ;
24 }
```

kern/trap/trap.c

```
1  void idt_init(void) {
2      extern void __alltraps(void);
3      //约定: 若中断前处于S态, sscratch为0
4      //若中断前处于U态, sscratch存储内核栈地址
5      //那么之后就可以通过sscratch的数值判断是内核态产生的中断还是用户态产生
           的中断
6      //我们现在是内核态所以给sscratch置零
7      write_csr(sscratch, 0);
8      //我们保证__alltraps的地址是四字节对齐的, 将__alltraps这个符号的地址直
           接写到stvec寄存器
9      write_csr(stvec, &__alltraps);
10 }
```

kern/driver/intr.c

```
1  #include <intr.h>
2  #include <riscv.h>
3
4  /* intr_enable - enable irq interrupt, 设置sstatus的Supervisor中断使能位 */
5  void intr_enable(void) { set_csr(sstatus, SSTATUS_SIE); }
6
```

```
/* intr_disable - disable irq interrupt */
void intr_disable(void) { clear_csr(sstatus, SSTATUS_SIE); }
```

trap.c 的中断处理函数 trap, 实际上把中断处理, 异常处理的工作分发给了 interrupt_handler(), exception_handler(), 这些函数再根据中断或异常的不同类型来处理。

kern/trap/trap.c (trap_dispatch)

```
/* trap_dispatch - dispatch based on what type of trap occurred */
static inline void trap_dispatch(struct trapframe *tf) {
    //scause的最高位是1，说明trap是由中断引起的
    if ((intptr_t)tf->cause < 0) {
        // interrupts
        interrupt_handler(tf);
    } else {
        // exceptions
        exception_handler(tf);
    }
}

/* *
 * trap - handles or dispatches an exception/interrupt. if and when trap()
 * returns,
 * the code in kern/trap/trapentry.S restores the old CPU state saved in
    the
 * trapframe and then uses the iret instruction to return from the
    exception.
 * */
void trap(struct trapframe *tf) { trap_dispatch(tf); }
```

我们可以看到, interrupt_handler() 和 exception_handler() 的实现还比较简单, 只是简单地根据 scause 的数值更仔细地分了下类, 做了一些输出就直接返回了。switch 里的各种 case, 如 IRQ_U_SOFT,CAUSE_USER_ECALL, 是 riscv ISA 标准里规定的。我们在 riscv.h 里定义了这些常量。我们接下来主要关注时钟中断的处理。

在这里我们对时钟中断进行了一个简单的处理, 即每次触发时钟中断的时候, 我们会给一个计数器加一, 并且设定好下一次时钟中断。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。通过在模拟器中观察输出我们即刻看到是否正确触发了时钟中断, 从而验证我们实现的异常处理机制。

kern/trap/trap.c (interrupt_handler)

```
void interrupt_handler(struct trapframe *tf) {
    intptr_t cause = (tf->cause << 1) >> 1; //抹掉scause最高位代表“这是中
       断不是异常”的1
    switch (cause) {
        case IRQ_U_SOFT:
            cprintf("User software interrupt\n");
```

```
            break;
        case IRQ_S_SOFT:
            cprintf("Supervisor software interrupt\n");
            break;
        case IRQ_H_SOFT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_SOFT:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_TIMER:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_TIMER:
            //时钟中断
            /* LAB3 EXERCISE2   YOUR CODE :  */
            /*(1)设置下次时钟中断
             *(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
            */
            break;
        case IRQ_H_TIMER:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_TIMER:
            cprintf("Machine software interrupt\n");
            break;
        case IRQ_U_EXT:
            cprintf("User software interrupt\n");
            break;
        case IRQ_S_EXT:
            cprintf("Supervisor external interrupt\n");
            break;
        case IRQ_H_EXT:
            cprintf("Hypervisor software interrupt\n");
            break;
        case IRQ_M_EXT:
            cprintf("Machine software interrupt\n");
            break;
        default:
            print_trapframe(tf);
            break;
```

```
    }
}
```

kern/trap/trap.c (exception_handler)

```
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   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
             */
            break;
        case CAUSE_BREAKPOINT:
            //非法指令异常处理
            /* 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
             */
            break;
        case CAUSE_MISALIGNED_LOAD:
            break;
        case CAUSE_FAULT_LOAD:
            break;
        case CAUSE_MISALIGNED_STORE:
            break;
        case CAUSE_FAULT_STORE:
            break;
        case CAUSE_USER_ECALL:
            break;
        case CAUSE_SUPERVISOR_ECALL:
            break;
        case CAUSE_HYPERVISOR_ECALL:
            break;
        case CACODE_MACHINE_ECALL:
            break;
        default:
            print_trapframe(tf);
            break;
```

```
42      }
43  }
```

下面是 RISCV 标准里 `scause` 的部分，可以看到有个 `scause` 的数值与中断/异常原因的对应表格。

下一节我们将仔细讨论如何设置好时钟模块。

2.4 滴答滴答（时钟中断）

时钟中断需要 CPU 硬件的支持。CPU 以” 时钟周期” 为工作的基本时间单位，对逻辑门的时序电路进行同步。我们的 “时钟中断” 实际上就是” 每隔若干个时钟周期执行一次的程序 “。

” 若干个时钟周期 “是多少个? 太短了肯定不行。如果时钟中断处理程序需要 100 个时钟周期执行，而你每 50 个时钟周期就触发一个时钟中断，那么间隔时间连一个完整的时钟中断程序都跑不完。如果你 200 个时钟周期就触发一个时钟中断，那么 CPU 的时间将有一半消耗在时钟中断，开销太大。一般而言，可以设置时钟中断间隔设置为 CPU 频率的 1%，也就是每秒钟触发 100 次时钟中断，避免开销过大。

我们用到的 RISCV 对时钟中断的硬件支持包括:

- OpenSBI 提供的 `sbi_set_timer()` 接口，可以传入一个时刻，让它在那个时刻触发一次时钟中断
- `rdtime` 伪指令，读取一个叫做 `time` 的 CSR 的数值，表示 CPU 启动之后经过的真实时间。在不同硬件平台，时钟频率可能不同。在 QEMU 上，这个时钟的频率是 10MHz, 每过 1s, `rdtime` 返回的结果增大 10000000

趣闻 在 RISCV32 和 RISCV64 架构中，`time` 寄存器都是 64 位的。`rdcycle` 伪指令可以读取经过的时钟周期数目，对应一个寄存器 `cycle`

注意，我们需要 “每隔若干时间就发生一次时钟中断”，但是 OpenSBI 提供的接口一次只能设置一个时钟中断事件。我们采用的方式是: 一开始只设置一个时钟中断，之后每次发生时钟中断的时候，设置下一次的时钟中断。

在 `clock.c` 里面初始化时钟并封装一些接口

libs/sbi.c

```
1  //当time寄存器(rdtime的返回值)为stime_value的时候触发一个时钟中断
2  void sbi_set_timer(unsigned long long stime_value) {
3      sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
4  }
```

kern/driver/clock.c

```
1  #include <clock.h>
2  #include <defs.h>
3  #include <sbi.h>
4  #include <stdio.h>
5  #include <riscv.h>
```

```
6
7  //volatile告诉编译器这个变量可能在其他地方被瞎改一通，所以编译器不要对这个
       变量瞎优化
8  volatile size_t ticks;
9
10 //对64位和32位架构，读取time的方法是不同的
11 //32位架构下，需要把64位的time寄存器读到两个32位整数里，然后拼起来形成一个
       64位整数
12 //64位架构简单的一句rdtime就可以了
13 //__riscv_xlen是gcc定义的一个宏，可以用来区分是32位还是64位。
14 static inline uint64_t get_time(void) { //返回当前时间
15 #if __riscv_xlen == 64
16     uint64_t n;
17     __asm__ __volatile__("rdtime %0" : "=r"(n));
18     return n;
19 #else
20     uint32_t lo, hi, tmp;
21     __asm__ __volatile__(
22         "1:\n"
23         "rdtimeh %0\n"
24         "rdtime %1\n"
25         "rdtimeh %2\n"
26         "bne %0, %2, 1b"
27         : "=&r"(hi), "=&r"(lo), "=&r"(tmp));
28     return ((uint64_t)hi << 32) | lo;
29 #endif
30 }
31
32 // Hardcode timebase
33 static uint64_t timebase = 100000;
34
35 void clock_init(void) {
36     // sie这个CSR可以单独使能/禁用某个来源的中断。默认时钟中断是关闭的
37     // 所以我们要在初始化的时候，使能时钟中断
38     set_csr(sie, MIP_STIP); // enable timer interrupt in sie
39     //设置第一个时钟中断事件
40     clock_set_next_event();
41     // 初始化一个计数器
42     ticks = 0;
43
44     cprintf("++ setup timer interrupts\n");
45 }
46
47 //设置时钟中断：timer的数值变为当前时间 + timebase 后，触发一次时钟中断
48 //对于QEMU，timer增加1，过去了10^-7 s， 也就是100ns
```

```
49 void clock_set_next_event(void) { sbi_set_timer(get_time() + timebase); }
```

回来看 trap.c 里面时钟中断处理的代码, 还是很简单的：每秒 100 次时钟中断，触发每次时钟中断后设置 10ms 后触发下一次时钟中断，每触发 100 次时钟中断（1 秒钟）输出一行信息到控制台。

kern/trap/trap.c

```
1  #include<clock.h>
2  #define TICK_NUM 100
3
4  static void print_ticks() {
5      cprintf("%d ticks\n", TICK_NUM);
6  #ifdef DEBUG_GRADE
7      cprintf("End of Test.\n");
8      panic("EOT: kernel seems ok.");
9  #endif
10 }
11
12 void interrupt_handler(struct trapframe *tf) {
13     intptr_t cause = (tf->cause << 1) >> 1;
14     switch (cause) {
15             /* blabla 其他case*/
16         case IRQ_S_TIMER:
17             clock_set_next_event();//发生这次时钟中断的时候，我们要设置下一
                   次时钟中断
18             if (++ticks % TICK_NUM == 0) {
19                 print_ticks();
20             }
21             break;
22         /* blabla 其他case*/
23     }
24 }
```

现在执行 make qemu, 应该能看到打印一行行的 100 ticks

2.5 断亦有断 (中断的关闭和开启)

中断机制固然重要，然而在操作系统运行过程中，依然有一些操作是不能接受被中断打断的，例如修改内存管理相关的数据结构等。为了确保这些操作不受干扰，操作系统提供了原子操作的机制。

以我们刚刚完成的物理内存管理为例，我们添加了以下内容来保证操作的原子性：

- kern/sync/sync.h：为确保内存管理修改相关数据时不被中断打断，提供两个功能，一个是保存 sstatus 寄存器中的中断使能位 (SIE) 信息并屏蔽中断的功能，另一个是根据保存的中断使能位信息来使能中断的功能。

- libs/atomic.h：定义了对一个二进制位进行读写的原子操作，确保相关操作不被中断打断。包括 set_bit() 设置某个二进制位的值为 1, change_bit() 给某个二进制位取反，test_bit() 返回某个二进制位的值。

kern/sync/sync.h

```
#ifndef __KERN_SYNC_SYNC_H__
#define __KERN_SYNC_SYNC_H__

#include <defs.h>
#include <intr.h>
#include <riscv.h>

static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void __intr_restore(bool flag) {
    if (flag) {
        intr_enable();
    }
}

//思考：这里宏定义的 do{}while(0)起什么作用?
#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)

#define local_intr_restore(x) __intr_restore(x);

#endif /* !__KERN_SYNC_SYNC_H__ */
```

在分配和释放内存的过程中，我们会先关闭中断，然后进行对应操作，最后重新开启中断。

kern/mm/pmm.c

```
struct Page *alloc_pages(size_t n) {
    struct Page *page = NULL;
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        page = pmm_manager->alloc_pages(n);
    }
```

```
    local_intr_restore(intr_flag);
    return page;
}

// free_pages - call pmm->free_pages to free a continuous n*PAGESIZE memory
void free_pages(struct Page *base, size_t n) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        pmm_manager->free_pages(base, n);
    }
    local_intr_restore(intr_flag);
}
```

3 实验练习解答

3.1 练习 1：完善中断处理

3.1.1 问题重述

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。

3.1.2 设计实现过程

以下是时钟中断处理程序的代码实现：

Listing 1: 时钟中断处理程序

```
case IRQ_S_TIMER:
    clock_set_next_event();
    ticks++;
    if(ticks % TICK_NUM == 0){
        cprintf("100ticks\n");
        num++;

        if(num == 10){
            cprintf("shutting down...\n");
            sbi_shutdown();
        }
    }
```

```
13      break;
```

1. **设置下次时钟中断**：调用 `clock_set_next_event()` 函数设置下一次时钟中断
2. **计数器递增**：全局变量 `ticks` 加一，记录时钟中断发生次数
3. **周期性输出**：每 100 次时钟中断输出一次”`100ticks`”
4. **系统关机**：当输出 10 次”`100ticks`” 后（即 1000 次时钟中断），调用关机函数

3.2 扩展练习 Challenge1：描述与理解中断流程

3.2.1 问题重述

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。

3.2.2 问题解答

当用户态或内核态的指令触发 trap 时，处理器首先由硬件自动把被中断指令的地址写入 sepc，把异常原因写入 scause，把可能的故障地址写入 sbadaddr，并更新 sstatus 里的 SPP/SPIE 等位；随后依据 stvec 指向的入口转入内核的陷入向量。idt_init 事先把 stvec 指向 __alltraps，同时把 sscratch 清零，用来标记当前正处于内核环境。

Listing 2: trap.c:49-55

```
1 void idt_init(void) {
2     extern void __alltraps(void);
3     /* Set sup0 scratch register to 0, indicating to exception vector
4        that we are presently executing in the kernel */
5     write_csr(sscratch, 0);
6     /* Set the exception vector address */
7     write_csr(stvec, &__alltraps);
8 }
```

于是控制权流向 trapentry.S 中的 __alltraps，该入口首先执行 SAVE_ALL，把原来的 sp 暂存进 sscratch 并在栈上为完整的 trapframe 腾出 36×REGBYTES 的空间，然后依次保存 32 个通用寄存器以及 sstatus/sepc/sbadaddr/scause 等 CSR；完成后通过 move a0, sp 把新栈顶（也就是 trapframe 首地址）作为实参，调用 C 语言实现的 trap，由它根据 scause 决定交给 interrupt_handler 还是 exception_handler 做进一步处理。trap 处理完成后返回到 __trapret，再由 RESTORE_ALL 按逆序恢复 CSR 和寄存器，最后执行 sret，借助已经写回的 sepc 和 sstatus 把控制权和上下文还原到被中断的程序。

move a0, sp 的目的是遵循 RISC-V 的调用约定：a0 是第一个整型参数寄存器，把 sp 当前指向的 trapframe 传给即将调用的 C 函数 trap，这样 trap 可以把这块内存直接解释为 struct trapframe *，读取或修改其中保存的所有寄存器和 CSR 信息。

Listing 3: trapentry.S:103-107

```
  __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
```

如果不做这一步，trap 将无法获得触发 trap 时的完整上下文，自然也无法根据异常类型决定继续运行的位置、是否调整 sepc 等。

SAVE_ALL 中每个寄存器保存在栈中的偏移其实是由 C 侧的 struct trapframe/struct pushregs 定义决定的：结构体成员在内存里线性排列，前 32 个成员恰好对应 x0–x31，之后紧接着是 status/epc/badvaddr/cause 四个字段。

Listing 4: trap.h:6-47

```
struct pushregs {
    uintptr_t zero;  // Hard-wired zero
    uintptr_t ra;    // Return address
    uintptr_t sp;    // Stack pointer
    uintptr_t gp;    // Global pointer
    uintptr_t tp;    // Thread pointer
    uintptr_t t0;    // Temporary
    uintptr_t t1;    // Temporary
    uintptr_t t2;    // Temporary
    uintptr_t s0;    // Saved register/frame pointer
    uintptr_t s1;    // Saved register
    uintptr_t a0;    // Function argument/return value
    uintptr_t a1;    // Function argument/return value
    uintptr_t a2;    // Function argument
    uintptr_t a3;    // Function argument
    uintptr_t a4;    // Function argument
    uintptr_t a5;    // Function argument
    uintptr_t a6;    // Function argument
    uintptr_t a7;    // Function argument
    uintptr_t s2;    // Saved register
    uintptr_t s3;    // Saved register
    uintptr_t s4;    // Saved register
    uintptr_t s5;    // Saved register
    uintptr_t s6;    // Saved register
```

```
25      uintptr_t s7;    // Saved register
26      uintptr_t s8;    // Saved register
27      uintptr_t s9;    // Saved register
28      uintptr_t s10;   // Saved register
29      uintptr_t s11;   // Saved register
30      uintptr_t t3;    // Temporary
31      uintptr_t t4;    // Temporary
32      uintptr_t t5;    // Temporary
33      uintptr_t t6;    // Temporary
34  };
35
36  struct trapframe {
37      struct pushregs gpr;
38      uintptr_t status;
39      uintptr_t epc;
40      uintptr_t badvaddr;
41      uintptr_t cause;
42  };
```

宏把每个寄存器写到与这套布局一致的偏移（例如旧 sp 被放在 2*REGBYTES(sp)，对应结构中的第三个字段），于是 trap 收到的指针无需再做搬运就能当作标准的 trapframe 来访问；同样地，RESTORE_ALL 也按相同偏移读回，保证保存/恢复一一对应。

在 __alltraps 里选择“保存全部寄存器”是出于通用性和安全性考虑：一方面，trap 及其调用的 C 代码会使用编译器能自由分配的寄存器，我们无法预知它会破坏哪些寄存器；另一方面，中断可能发生在用户态或内核态的任意指令后，只有保存了完整的上下文才能做到无痕返回，支持嵌套中断、调度和调试。虽然某些特定的中断按照 RISC-V ABI 只需保护少量 callee-saved 寄存器，但为了用统一的入口覆盖所有 trap 并避免人为遗漏，我们在 __alltraps 里一律抢先把 32 个通用寄存器和关键 CSR 都备份下来，再交给 C 层做差异化处理。

3.3 扩展练习 Challenge2：理解上下文切换机制

3.3.1 问题重述

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？

3.3.2 问题解答

首先理解一下 sscratch 是个什么东西：

sscratch 是 RISC-V 特权架构在 S-mode 下提供的一只通用可读写 CSR，全称“Supervisor Scratch Register”。它的设计就像一个“临时寄存器”，专供陷入向量或者内核保存/传递少量上下文信息使用：当 hart 发生 trap 并跳转到 S 态入口时，硬件不会自动备份任何通用寄存器，所以内核常在第一时间把当前栈指针或其它关键信息写进 sscratch，等到把栈调整好、构造好 trapframe

后再把这份备份读出来。在当前 ucore 的实现里，idt_init 还会把 sscratch 预置为 0，用作“当前是否已经在内核上下文”的标记，避免递归陷入时把内核态 trap 误判成用户态。因为 sscratch 完全由软件自由支配，没有硬件定义的语义，因此它非常适合做这种小范围的上下文传递：用完就重置，下一次 trap 可以再复用。

csrw sscratch, sp 在陷入入口一开始把当前栈指针写入 S 态的 sscratch CSR，先备份触发 trap 时的原始栈顶；随后的 csrrw s0, sscratch, x0 再把这份备份读回到 s0，同时把 sscratch 置零。这样做有两层目的：其一，在 SAVE_ALL 里我们马上要把 sp 下拉 36×REGBYTES，为 trapframe 腾空间，如果不先把“原始 sp”塞进 sscratch 就无法在 trapframe 中完整记录被打断现场；其二，把 sscratch 归零相当于给递归陷入打了标记——下一次 trap 再来时只要发现 sscratch==0 就知道当前已经在内核栈上，可以按内核路径处理，避免把内核态 trap 当成用户态 trap 去重新切栈。

SAVE_ALL 里把 sbadaddr/stval、scause 等 CSR 也写进栈，是为了让 C 语言层的 trap(struct trapframe *tf) 能够读取这些只读状态寄存器，判断异常类型、出错地址乃至做调试打印；可以看到 trap.h 中 struct trapframe 的末尾就按顺序定义了 status/epc/badvaddr/cause 字段。RESTORE_ALL 没有把它们写回，一是因为这些 CSR 本身是“事件描述器”，没有恢复现场的语义，下一次 trap 来时硬件会重新覆盖；二是很多 CSR（例如 scause/stval）在 S 态下是只读的，硬写反而会触发非法操作；真正需要恢复的只有会参与返回流程的 sstatus 和 sepc，因此 RESTORE_ALL 只写回这两个寄存器，其余的 trap 信息纯粹用来让处理代码观测与决策——这也就是“保存但不恢复”的意义所在。

3.4 扩展练习 Challenge3：完善异常中断

3.4.1 问题重述

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”。

3.4.2 设计实现过程

非法指令异常和断点异常的处理机制非常的简单，直接在 kern/trap/trap.c 的异常处理函数中，针对 CAUSE_ILLEGAL_INSTRUCTION 和 CAUSE_BREAKPOINT 分别补齐处理分支即可。

非法指令分支里先用 cprintf 打印 tf->epc 指向的出错地址（Illegal instruction caught at 0x%08x），再输出异常类型描述（Exception type: Illegal instruction），最后将 tf->epc += 4，跳过导致异常的 32 位指令，避免返回后再次陷入。

Listing 5: 非法指令分支

```
1 void exception_handler(struct trapframe *tf) {
2     switch (tf->cause) {
3         ...
4         case CAUSE_ILLEGAL_INSTRUCTION:
5              // 非法指令异常处理
6             /*(1)输出指令异常类型（ Illegal instruction）
```

```
7                *(2)输出异常指令地址
8                *(3)更新 tf->epc寄存器
9               */
10              cprintf("Illegal instruction caught at 0x%08x\n", tf->epc);
11              cprintf("Exception type: Illegal instruction\n");
12              // 跳过导致异常的指令，避免返回后再次陷入
13              tf->epc += 4;
14              break;
15          ...
16      }
17  }
```

断点分支同样先报出 ebreak 的触发地址（ebreak caught at 0x%08x）和类型（Exception type: breakpoint），再将 tf->epc += 2 跳过断点指令，符合 RISC-V 压缩指令集里 ebreak 的长度 (此处一开始未注意导致出错，后面的“实验所遇问题”中将会详细解释这一问题)。

Listing 6: 断点分支

```
1  void exception_handler(struct trapframe *tf) {
2      switch (tf->cause) {
3          ...
4          case CAUSE_BREAKPOINT:
5              //断点异常处理
6              /*(1)输出指令异常类型 ( breakpoint)
7               *(2)输出异常指令地址
8               *(3)更新 tf->epc寄存器
9              */
10             cprintf("ebreak caught at 0x%08x\n", tf->epc);
11             cprintf("Exception type: breakpoint\n");
12             tf->epc += 2;
13             break;
14         ...
15     }
16 }
```

由于 tf->epc 是 __alltraps 在保存 trapframe 时抓取的原始 EPC，这两处修改会在 RESTORE_ALL 写回 sepc 后生效，sret 就能从新的地址继续执行。

为了能直观的检测新增的这两个处理机制能否正常运行，在 kern/init/init.c 中在内核初始化序列中手工构造了两个触发点：通过.word 0x00000000 注入一条全零指令作为非法指令；紧接着执行 ebreak 作为断点指令。这些内联汇编直接触发异常，使得 __alltraps 汇编入口创建 trapframe 后，trap() 能够把控制权转给 exception_handler()。

Listing 7: init.c 新增触发点

```
1  int kern_init(void) {
2      ...
3      cprintf("非法指令异常测试...\n");
```

```
    __asm__ volatile(".word 0x00000000");

    cprintf("断点异常测试...\n");
    __asm__ volatile("ebreak");
    ...
}
```

运行 make qemu 检验一下：

Listing 8: make 终端信息

```
...
check_alloc_page() succeeded!
satp virtual address: 0xffffffffc0206000
satp physical address: 0x0000000080206000
非法指令异常测试...
Illegal instruction caught at 0xc02000a0
Exception type: Illegal instruction
断点异常测试...
ebreak caught at 0xc02000b0
Exception type: breakpoint
++ setup timer interrupts
100ticks
100ticks
100ticks
100ticks
100ticks
100ticks
100ticks
100ticks
100ticks
100ticks
shutting down...
```

可见实现成功。此外，观察三个异常和中断测试的打印顺序和异常信息也证实了 save/restore 流程运转正常。也验证了 trapframe 中 CSR 信息的意义：tf->epc 能用于打印回溯，同时可直接修改用于控制返回地址。

4 实验与理论对应

4.1 实验中出现相关知识点

本次实验将中断理论与 S 模式编程实践相结合：

- **中断/异常区分**：通过 `scause` 寄存器在软件中区分异步中断和同步异常。
- **权限模式**：在 S 模式下编程，处理来自 U 模式的陷入（Trap）。

- **中断处理流程**：
  - **硬件**：自动保存 sepc、scause 并跳转到 stvec。
  - **软件**：通过 __alltraps 汇编入口保存和恢复完整上下文（SAVE_ALL / RESTORE_ALL）。
- **中断分发**：在 trap_dispatch 中根据 scause 的原因码（Cause Code）调用不同的 C 语言处理函数。
- **时钟中断**：通过 SBI 调用设置 S 模式时钟中断（IRQ_S_TIMER），实现了理论上的定时器。
- **中断使能/屏蔽**：通过控制 sstatus.SIE（全局开关）和 sie.STIE（特定源）来管理中断。
- **临界区保护**：使用 local_intr_save/restore 宏（开关中断）来保护 alloc_pages 等临界区代码，实现原子操作。

4.2 实验中未涉及相关知识点

本次实验为了聚焦核心机制，简化了部分高级中断特性：

- **中断向量表模式**：实验仅用了 stvec 的 Direct 模式，未用 Vectored 模式。
- **中断嵌套**：实验中中断处理全程关中断，未实现高优先级中断抢占低优先级中断。
- **外部设备中断**：未处理来自磁盘、网卡等的外部中断（IRQ_S_EXT），这通常需要配置 **PLIC**（平台级中断控制器）。
- **完整的异常处理**：未实现 **缺页异常**的处理，这是虚拟内存的核心。
- **软件中断**：未实际使用 IRQ_S_SOFT（常用于核间中断或延迟任务）。

5 实验所遇问题

5.1 问题描述

在第一次实现断点异常处理机制时，按 tf->epc += 4 处理，make 后终端不断刷出 trapframe 信息，陷入死循环。

5.2 问题解决

断点异常被捕获以后，在 trap.c 里把 tf->epc 固定加了 4。但在 kern/init/init.c 写的内联 ebreak 被汇编成了 16 位的压缩指令 c.ebreak（指令长度是 2 字节）。因此返回时 sepc 被改成了 0xc02000b4，正好落在下一条指令的中间。

CPU 从这个半截位置取指，会把后面的字节当成完全不同的指令去执行，结果立即触发 “Load page fault” （cause = 0x0000000d，badvaddr = 0x70），进入异常处理的 default 分支打印 trapframe。因为 tf->epc 仍然指向那条坏指令，返回后又重新陷入，如此循环，就看到终端不断刷出 trapframe。

要解决这个问题，需要根据实际指令长度来前进 tf->epc，对于 c.ebreak ，改为加 2 即可；或者直接用.word 0x00100073 强制生成 32 位的 ebreak。