武汉大学计算机学院
本科生课程设计报告

实验四：中断处理与时钟管理

专 业 名 称　：计算机科学与技术

课 程 名 称　：操作系统实践 A

指 导 教 师　：李祖超　副教授

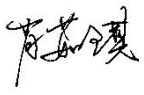
学 生 学 号　：2023302111416

学 生 姓 名　：肖茹琪

二〇二五年十月

郑 重 声 明

本人呈交的设计报告，是在指导老师的指导下，独立进行实验工作所取得的成果，所有数据、图片资料真实可靠。尽我所知，除文中已经注明引用的内容外，本设计报告不包含他人享有著作权的内容。对本设计报告做出贡献的其他个人和集体，均已在文中以明确的方式标明。本设计报告的知识产权归属于培养单位。

本人签名：  　　日期：2025.10.20

摘　要

本次实验旨在深入理解操作系统核心机制之一：中断与异常处理。基于RISC-V 架构的特权级规范，设计并实现了一个功能完备的内核中断处理框架。实验首先解决了从机器模式（M-mode）到监督模式（S-mode）的中断委托问题，通过配置 medeleg 和 mideleg 寄存器，使内核能够在 S 模式下直接响应特定的异常和中断。其次，通过编写汇编代码 kernelvec.S，实现了精细的上下文保存与恢复机制，确保了内核在处理中断前后寄存器状态的一致性。在此基础上，构建了统一的中断分发器 trap.c，能够准确区分软件中断、时钟中断、外部中断及各类同步异常（如系统调用、缺页异常等）。

针对时钟管理，实验采用了 RISC-V SBI 标准的时钟接口，利用 M 模式下的定时器硬件产生中断，并通过软件注入的方式通知 S 模式内核，从而建立起操作系统的时间基准。最后，通过一系列边界测试（如非法指令、断点异常、非法内存访问），验证了异常处理机制的健壮性。本次实验不仅打通了硬件事件到软件处理的链路，也为后续实现进程调度和抢占式多任务系统奠定了坚实的基础。

关键词：RISC-V；中断委托；上下文切换；时钟中断；异常处理

目录

1 实验目的和意义 5

1.1 实验目的 5
1.2 实验意义 5

2 实验准备 6

2.1 任务 1：理解 RISC-V 中断架构 6
2.2 任务 2：分析 xv6 的中断处理流程 7

3 实验原理与设计 8

3.1 总体架构设计 8
3.2 中断分发框架设计 9
3.3 上下文管理设计 10
3.4 异常处理策略设计 10

4 实验步骤与实现 10

4.1 机器模式初始化与中断委托 10
4.2 监督模式中断框架搭建 11
4.3 上下文切换的汇编实现 12
4.4 时钟中断系统的全链路实现 12
4.5 异常处理与系统保护 13

5 实验测试与结果 13

5.1 时钟中断功能验证 13
5.2 基础异常处理验证 13
5.3 致命错误保护验证 13

6 遇到的问题及解决 13

6.1 问题 1 13
6.2 问题 2 13
6.3 问题 3 13

7 思考题 13

7.1 中断设计 13
7.2 性能考虑 17
7.3 可靠性 18
7.4 扩展性 19
7.5 实时性 19

8 实验总结 20

1 实验目的和意义

1.1 实验目的

本实验旨在深入理解现代操作系统中中断与异常处理的核心机制。通过分析 xv6-riscv 的相关源码，并基于 RISC-V 架构的特权级规范，独立设计并实现一个简化的内核中断处理框架。具体目标包括：

1. 掌握RISC-V 中断架构与CSR 寄存器操作：深入理解 stvec、scause、sepc、sstatus 等关键控制状态寄存器的作用与配置方法。

2. 理解特权级切换与中断委托机制：掌握 RISC-V 架构中 M 模式与 S 模式的协作关系，学会如何通过 medeleg 和 mideleg 将底层硬件中断委托给操作系统内核处理。

3. 实现上下文保存与恢复：通过编写汇编代码，掌握在中断发生瞬间如何保存通用寄存器现场到内核栈或 Trapframe 中，以及在中断处理完成后如何无损恢复现场。

4. 构建通用的中断/异常分发框架：编写 C 语言中断处理程序，实现对时钟中断、软件中断、非法指令异常、内存访问错误等不同类型事件的解析与分发。

5. 实现操作系统时钟管理：利用硬件定时器实现系统 Tick 计数，理解时钟中断在操作系统时间管理和任务调度触发中的核心作用。

1.2 实验意义

中断机制被誉为操作系统的“灵魂”，本实验对于理解操作系统如何作为硬件与应用程序之间的协调者具有决定性意义。

通过本次实验，能够从根本上建立起“事件驱动”的操作系统观念，将编程思维从传统的被动指令执行转变为对异步硬件事件的主动响应。其中，时钟中断管理的实现标志着操作系统具备了从当前运行任务手中“夺回”CPU 控制权的能力，这不仅是系统时间维护的基础，更是未来实现时间片轮转调度和抢占式多任务处理的先决条件。同时，异常处理机制的构建极大提升了系统的健壮性与安全性，使内核能够有效隔离并处理用户程序可能产生的非法指令或错误内存访问，

防止系统崩溃。此外，本实验要求在直接操作硬件状态的汇编代码与处理内核逻辑的 C 语言代码之间频繁切换，这种深度的软硬件协同设计经历，有助于深刻理解计算机体系结构与操作系统软件之间的紧密耦合关系，为后续更复杂的内核开发打下坚实基础。

2 实验准备

在开始代码实现之前，必须深入理解 RISC-V 架构的中断模型以及 xv6 操作系统处理中断的控制流。本章主要针对实验前的理论学习（任务 1）和源码分析（任务 2）进行阐述。

2.1 任务 1：理解 RISC-V 中断架构

RISC-V 架构定义了三种特权模式：机器模式（M-mode）、监督模式（S-mode）和用户模式（U-mode）。操作系统内核运行在 S 模式，而底层的硬件控制和时钟中断产生通常发生在 M 模式。为了使内核能够处理中断，RISC-V 提供了一套复杂的委托机制。

2.1.1 中断与异常的区别

在 RISC-V 中，“Trap” 是所有特权级切换事件的统称，主要分为两类：

- **中断（Interrupt）**：异步发生，由外部硬件设备触发（如时钟中断、UART 输入）。其中断原因码最高位为 1。
- **异常（Exception）**：同步发生，由当前执行的指令触发（如非法指令、断点 ebreak、系统调用 ecall、缺页异常）。其原因码最高位为 0。

2.1.2 中断委托机制

默认情况下，所有的 Trap 都会陷入到 M-mode 处理。为了提高操作系统 S-mode 的效率，避免频繁陷入 M 模式，RISC-V 提供了委托寄存器：

- medeleg (Machine Exception Delegation)：用于设置哪些异常直接交给 S 模式处理。例如，系统调用和断点异常通常需要委托给 OS。
- mideleg (Machine Interrupt Delegation)：用于设置哪些中断直接交给 S 模式处理。 在本次实验中，我们关注将软件中断、时钟中断和外部中断委托给 S 模式，使得内核可以直接响应这些事件。

2．1．3 关键控制状态寄存器

在 S 模式下处理中断，主要依赖以下寄存器：

- stvec (Supervisor Trap Vector)：存储中断处理程序的入口地址。实验中需将其设置为内核中断向量表的地址。
- sie / sip (Enable / Pending)：分别控制中断的使能和查询当前挂起的中断。
- scause (Supervisor Cause)：记录导致 Trap 的具体原因。
- sepc (Supervisor Exception PC)：记录触发 Trap 的指令地址，用于中断返回。
- sstatus (Supervisor Status)：控制全局中断使能以及记录进入 Trap 前的特权级。

2.2 任务 2：分析 xv6 的中断处理流程

基于 xv6 的源码结构，一个完整的中断处理流程涉及机器模式的初始化、汇编级的上下文保存以及 C 语言级的逻辑分发。

2．2．1 机器模式初始化与时钟注入

```
// 时钟中断委托给S模式
w_mideleg(r_mideleg() | (1L << 5));
// 设置机器模式陷阱向量
w_mtvec((uint64)timervec);
```

图 1 start.c 中的机器模式设置

系统启动时首先处于 M 模式。在 start.c 中，主要完成以下工作：

1. 设置特权级:将 mstatus 的 MPP 位设置为 Supervisor,确保执行 mret 后切换到 S 模式。

2. 配置委托：通过写入 medeleg 和 mideleg，将绝大多数中断和异常的处理权移交给 S 模式。

3. 时钟中断的特殊处理：RISC-V 的时钟硬件（CLINT）只在 M 模式产生中断。因此，xv6 采用了一种“软件注入”的策略：

- M 模式设置 mtvec 指向 timervec（机器模式中断向量）。
- 当硬件时钟中断发生时，CPU 陷入 M 模式执行 timervec。

- timervec 在重置定时器后，通过写入 sip 寄存器触发一个 S 模式软件中断。
- S 模式内核接收到软件中断，从而间接响应该时钟事件。

2.2.2 上下文保存与恢复机制

当 S 模式发生中断时，硬件会跳转到 stvec 指向的地址。由于中断可能随时发生，会破坏寄存器状态，因此必须立即保存现场。 通过研读代码可知，kernelvec 的处理逻辑遵循以下步骤：

1. 分配栈空间：将栈指针 sp 下移，开辟空间存储 trapframe。
2. 保存寄存器：将所有通用寄存器（ra, gp, tp, t0-t6, a0-a7, s0-s11）保存到栈中。
3. 保存特殊状态：读取并保存 sepc 和 sstatus。
4. 调用 C 处理函数:将当前栈指针作为参数(mv a0, sp)，调用 kerneltrap。
5. 恢复现场：从栈中恢复所有寄存器，最后执行 sret 指令，硬件原子性地恢复 PC 和特权级。

2.2.3 中断分发逻辑

kerneltrap 是 S 模式中断处理的核心入口，其负责根据 scause 进行分发：

- 设备中断：检查 scause 最高位是否为 1。如果是，进一步判断是软件中断（对应时钟事件）还是外部中断（对应 UART/磁盘）。
- 异常处理：如果 scause 最高位为 0，则判定为异常。代码需根据具体编号（如 2 为非法指令，13 为缺页）调用相应的处理逻辑。在实验中，这部分通过 handle_exception 函数实现，确保了非法操作不会导致内核崩溃。

3 实验原理与设计

3.1 总体架构设计

本实验旨在构建一个基于 RISC-V 特权级架构的中断处理子系统。系统设计遵循“机制与策略分离”的原则，将底层的硬件响应（M 模式）与上层的业务处理（S 模式）解耦。

3．1．1　特权级协作模型

RISC-V 架构中，硬件中断默认由 M 模式捕获。为了使操作系统内核能直接管理系统资源，设计采用了中断委托模型：

- 异常委托：将系统调用、断点、页错误等同步异常委托给 S 模式，使内核能直接处理用户态或内核态的异常。
- 中断委托：将软件中断、外部中断委托给 S 模式。
- 时钟中断特殊处理：由于 RISC-V 硬件定时器中断无法直接委托，设计采用“软件注入”策略，即 M 模式捕获硬件时钟中断后，通过软件向 S 模式发送一个 Pending 信号，模拟出一个 S 模式的时钟中断。

3．1．2　中断处理流程设计

中断处理流程设计为三层结构：

- 汇编入口层：负责最底层的上下文保存与恢复，屏蔽硬件差异。
- 分发层：识别中断源（是时钟、外设还是异常），查询中断向量表。
- 处理层：执行具体的中断服务例程（ISR）。

3.2 中断分发框架设计

为了支持未来扩展更多的外设（如磁盘、键盘），中断分发系统设计为“注册表驱动”模式，而非硬编码的 switch-case 结构。

3．2．1　中断向量表数据结构

定义函数指针类型 interrupt_handler_t，并设计一个全局数组作为软件中断向量表。该表以中断号（IRQ Number）为索引，映射到具体的处理函数。

```
// 定义中断处理函数原型
typedef void (*interrupt_handler_t)(void);

// 定义支持16个中断源的软件向量表
// 索引即为 scause 中的中断原因码
static interrupt_handler_t interrupt_handlers[16];
```

图 2 中断向量表数据结构定义

3．2．2　接口设计

设计对外接口 register_interrupt，允许其他内核模块在初始化时动态注

册自己的中断处理函数。这种设计使得 trap.c（分发器）无需知道具体的设备细节，实现了高内聚低耦合。

3.3 上下文管理设计

下文切换是操作系统最核心的机制之一。设计重点在于确定保存位置和保存内容。

- 保存位置：鉴于当前实验环境为单核且未实现进程切换，设计直接利用内核栈来保存上下文。当发生中断时，栈指针 sp 下移，在栈上开辟一块 Trapframe 区域。
- 保存内容：必须保存所有的通用寄存器，因为中断处理程序可能会修改任意寄存器。此外，必须保存 sepc 和 sstatus，防止嵌套中断覆盖这些关键状态。

3.4 异常处理策略设计

针对同步异常，设计了两种不同的处理策略：

- 可恢复异常：如断点和系统调用。设计策略是处理完请求后，修改 sepc 跳过当前指令，让程序继续运行。
- 致命异常：如非法指令和页错误。由于当前未实现进程隔离和信号机制，设计策略是“快速失败”，即打印详细的错误现场（包括出错地址和原因）后，调用 panic 停止系统，防止错误扩散导致数据损坏。

4 实验步骤与实现

4.1 机器模式初始化与中断委托

系统启动初期处于 M-mode，首要任务是配置系统环境并切换到 S-mode。这一步主要在 kernel/start.c 中完成。

1. 配置特权级切换： 通过修改 mstatus 寄存器的 MPP 字段，将其设置为 Supervisor 模式(01)。同时将 mepc 设置为 main 函数地址。这样在执行 mret 指令后，CPU 将跳转到 main 函数并切换至 S 模式。

2. 配置中断与异常委托： 为了使操作系统内核能够直接响应硬件事件，必须设置委托寄存器：

- 异常委托：向 medeleg 寄存器写入 0xffff，将非法指令、断点、页错误等所有同步异常委托给 S 模式处理。
- 中断委托：向 mideleg 寄存器写入位掩码，分别使能软件中断、时钟中断和外部中断的委托。

3. M 模式时钟初始化： 调用 timer_init_hart() 初始化每个核心的定时器数据区，并设置 mtvec 寄存器指向机器模式中断向量 timervec，为后续的时钟注入做准备。

```
// kernel/start.c
w_medeleg(0xffff);
w_mideleg((1 << IRQ_S_SOFT) | (1 << IRQ_S_TIMER) | (1 << IRQ_S_EXT));
```

图 3 kernel/start.c 中的委托设置

4.2 监督模式中断框架搭建

进入 main 函数后，内核处于 S 模式，需要在 kernel/trap.c 的 trap_init 函数中完成中断系统的最终激活。

1. 设置 S 模式中断向量： 将 stvec 寄存器设置为 kernelvec 函数的地址。由于我们尚未实现针对不同中断源的硬件向量化跳转，因此采用 Direct 模式（最低两位为 0），所有 S 模式的 Trap 都会统一跳转到 kernelvec。

2. 开启中断使能：

- 设置 sie 寄存器，分别开启软件中断、时钟中断和外部中断的接收位。
- 设置 sstatus 的 SIE 位，开启全局中断开关。

```
// ==================== 初始化中断系统 ====================
void trap_init(void)
{
    printf("Initializing trap system...\n");

    // 清空中断处理函数表
    for(int i = 0; i < 16; i++) {
        interrupt_handlers[i] = 0;
        interrupt_counts[i] = 0;
    }

    // 设置S模式中断向量基址
    // MODE=0 (Direct): 所有中断都跳转到同一个地址
    // stvec必须4字节对齐(最低2位为00表示Direct模式)
    w_stvec((uint64)kernelvec);
    printf("Set stvec to %p\n", (void*)kernelvec);

    // 启用S模式中断
    // SIE: Supervisor Interrupt Enable
    // - SEIE: 外部中断使能
    // - STIE: 时钟中断使能
    // - SSIE: 软件中断使能
    w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);

    // 全局中断使能：设置sstatus.SIE位
    w_sstatus(r_sstatus() | SSTATUS_SIE);

    printf("Trap system initialized\n");
}
```

图 4 kernel/trap.c 的 trap_init 函数

4.3 上下文切换的汇编实现

当 Trap 发生时，硬件不会自动保存通用寄存器。为了保护被中断程序的现场，在 kernel/kernelvec.S 中实现了精细的汇编逻辑。

1. 保存现场：

- 栈分配:将栈指针 sp 向下移动 264 字节,为 struct trapframe 预留空间。
- 寄存器保存：依次将 ra, sp, gp, tp, t0-t6, a0-a7, s0-s11 等所有通用寄存器保存到栈中。
- 特殊状态保存：最关键的一步是读取 sepc 和 sstatus 并保存到栈中。这是为了防止在处理嵌套中断或异常时，这些 CSR 的值被覆盖导致无法正确返回。

2. 调用处理函数： 完成保存后，将当前的栈指针 sp 移动到 a0 寄存器，作为参数传递给 C 语言函数 kerneltrap(struct trapframe *tf)。

3. 恢复现场 ： 从 kerneltrap 返回后，执行逆操作。首先从栈中恢复 sstatus 和 sepc，然后恢复所有通用寄存器，最后执行 sret 指令。sret 会原子性地将 sepc 的值写入 PC，并根据 sstatus 恢复权限级，使程序从中断点继续执行。

4.4 时钟中断系统的全链路实现

RISC-V 的时钟中断处理比较特殊，因为硬件定时器只触发 M 模式中断。本实验实现了一套“M 模式触发 -> S 模式响应”的跨特权级机制。

1. 阶段一：机器模式触发 (kernel/timervec.S) 当硬件计时器达到 mtimecmp 设定值时，CPU 陷入 M 模式执行 timervec。该汇编程序执行两个核心操作：

- 重置定时器：读取 mtime，加上预设的时间间隔，写入 mtimecmp，从而设置下一次硬件中断时间。
- 注入软件中断：通过指令 csrw sip, 2 向 sip 寄存器的 SSIP 位写入 1。这会在 S 模式下人为触发一个“软件中断”。

2. 阶段二：监督模式响应 (kernel/trap.c) S 模式的 devintr 函数捕获到 IRQ_S_SOFT（软件中断）。

- 首先清除 sip 中的软件中断标志，防止重复触发。
- 然后调用 timer_interrupt()，更新系统的 ticks 计数器，并打印系统运行时间。
- 通过这种方式，S 模式内核成功感知并处理了物理时钟的滴答事件。

4.5 监异常处理与系统保护

在 kernel/trap.c 的 handle_exception 函数中，我实现了针对不同异常类型的分级处理策略，确保系统的健壮性。

1. 可恢复异常处理： 对于断点异常和系统调用，系统打印提示信息后，手动将 tf->sepc 增加指令长度（2 字节或 4 字节），跳过当前指令继续执行。

2. 致命错误保护： 对于非法指令和页错误，由于当前内核尚未实现进程查杀功能，继续执行可能导致更严重的内存破坏。因此，我的实现逻辑是打印详细的错误现场（包括 scause 原因码、sepc 错误指令地址、stval 错误内存地址），然后调用 panic() 强制停机。这种“快速失败”机制在开发阶段能有效帮助定位内核 Bug。

```
// kernel/trap.c
case CAUSE_LOAD_PAGE_FAULT:
case CAUSE_STORE_PAGE_FAULT:
    printf("Exception: Page Fault at %p\n", r_stval());
    printf("Instruction address: %p\n", tf->sepc);
    panic("page fault"); // 保护系统，防止错误扩散
    break;
```

图 5 handle_exception 中的 panic 保护逻辑

5 实验测试与结果

本章展示实验的运行结果，通过三个主要场景的测试截图，验证中断处理框架、时钟驱动机制以及异常分发系统的正确性。测试环境为 QEMU 模拟器，内核运行在 RISC-V 64 位 S 模式下。

5.1 时钟中断功能验证

首先验证系统的时间管理功能。系统启动后，M 模式下的定时器开始工作，并定期向 S 模式注入中断。

- 测试方法：在 main.c 的主循环中，系统进入等待状态。每当 S 模式接收

到中断时，timer_interrupt 函数会被调用，更新全局 ticks 计数，并每隔一秒打印 "System uptime"。

- 测试结果：如图 5-1 所示，控制台连续输出了“ [Timer] System uptime: 1 seconds、2 seconds ”等信息。
- **结果分析：**这证明了中断全链路的畅通：硬件时钟触发 -> M 模式 timervec 响应 -> 软件注入 sip -> S 模式 devintr 捕获 -> timer_interrupt 处理。系统成功建立了稳定的时间基准。

5.2 基础异常处理验证（可恢复异常）

接下来验证内核对同步异常的响应能力。首先测试那些不应导致系统崩溃的“良性”异常，如调试断点和系统调用。

- 测试方法:在 exception_test.c 中,通过内联汇编 asm volatile("ebreak") 人为触发断点异常，以及通过 ecall 指令触发系统调用异常。
- 测试结果：如图 5-2 所示，当执行 ebreak 时，内核捕获到了 Cause 3 ，打印提示信息后，程序没有崩溃，而是成功跳过指令继续执行，输出了 "Breakpoint handled successfully"。随后系统调用测试也成功返回。
- **结果分析：**这验证了 kernelvec.S 的上下文保存与恢复机制是正确的。内核能够准确识别异常类型，并在处理完成后无损地恢复寄存器状态（特别是 sepc 的调整），使程序流得以继续。

5.3 致命错误保护验证（页错误）

最后验证系统的自我保护机制。当内核代码尝试访问非法内存地址时，异常处理程序应当阻止该操作并报告错误，防止数据损坏。

- 测试方法：在测试代码中开启 test_load_page_fault()，该函数试图读取一个未映射的虚拟地址（如 0xdeadbeef 或空指针），从而触发加载页错误（Load Page Fault）。
- 测试结果:如图 5-3 所示,内核立即捕获到了异常,输出了 Exception: Load Page Fault，并打印了详细的错误现场（Fault address 和 Instruction address）。随后，系统执行了 panic 操作，终止了运行。
- **结果分析：**这符合预期的安全设计。内核成功拦截了非法内存访问，scause 寄存器正确记录了异常原因（Cause 13）。虽然当前未实现进程查杀，但“报

错并停机”的机制证明了内核具备了处理致命错误的能力，实现了对系统完整性的保护。

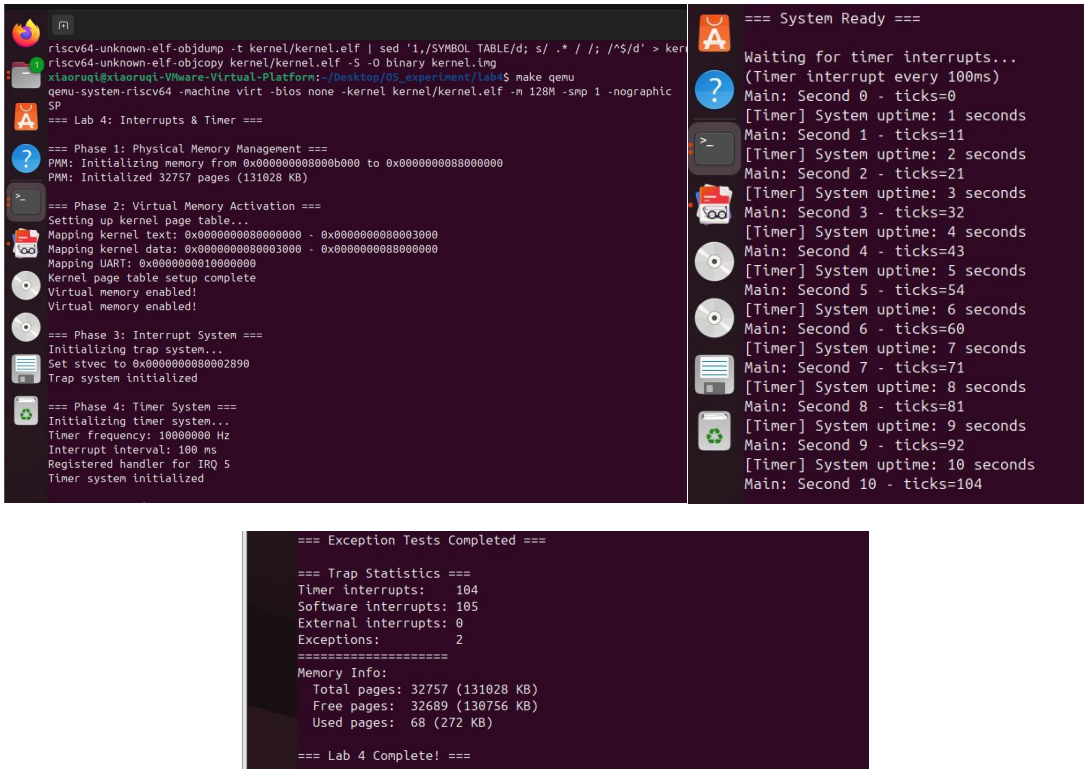


图 6 测试代码输出结果

6 遇到的问题及解决

6.1 问题 1：S 模式无法接收时钟中断

- **问题描述：**在完成 M 模式定时器初始化和委托设置后，系统能够进入机器模式的 timervec 中断向量，但在执行 mret 返回 S 模式后，trap.c 中的 devintr 函数从未被调用，导致 S 模式无法感知时钟滴答。
- **原因分析：**查阅 RISC-V 特权级规范发现，mtime 和 mtimecmp 是 M-mode 专有的 CSR，硬件产生的时钟中断默认只在 M 模式触发。虽然可以通过 mideleg 将时钟中断委托给 S 模式，但在某些硬件实现或 QEMU 环境中，直接委托可能存在兼容性问题。更重要的是，标准做法通常利用 M 模式作为“固件”层，统一管理时钟硬件。
- **解决方案：**采用了“软件中断注入”的策略。在 M 模式的中断处理程序 timervec.S 中，不仅重置了 mtimecmp，还增加了以下指令：

```
# 触发 S 模式软件中断
li a1, 2
csrw sip, a1
```

图 7 timervec.S 添加代码

6.2 问题 2：异常测试陷入死循环

- **问题描述：**在测试断点异常时，执行 asm volatile("ebreak") 后，控制台不断重复打印 "Exception: Breakpoint"，系统无法继续执行后续代码，陷入了无限的中断循环。
- **原因分析：**当异常发生时，硬件会将当前指令的地址保存到 sepc 寄存器。中断处理完成后，sret 指令会将 PC 恢复为 sepc 的值。对于异常而言，sepc 指向的是触发异常的那条指令（即 ebreak 本身）。因此，如果不修改 sepc，程序返回后会再次执行 ebreak，再次触发异常，从而形成死循环。
- **解决方案：**在 handle_exception 函数中，针对非致命异常，手动调整返回地址。由于 RISC-V 的压缩指令集中 ebreak 占用 2 字节，我在处理逻辑中增加了 tf->sepc += 2；对于系统调用 ecall（4 字节），则执行 tf->sepc += 4。修改后，程序能够正确跳过异常指令，继续向下执行。

6.3 问题 3：中断返回后系统状态异常

- **问题描述：**在进行高频率的时钟中断测试时，系统运行一段时间后会出现随机的寄存器数值错误或特权级模式错误（SPP 位异常），最终导致 Panic。
- **分析：**最初在编写 kernelvec.S 时，我只保存了 32 个通用寄存器。然而,在中断处理过程中,CPU 可能会跳转到 C 语言函数(如 kerneltrap)。如果发生嵌套中断，或者 C 函数内部修改了 CSR 寄存器，那么 sepc 和 sstatus 的值就会被覆盖。由于这两个寄存器没有被保存到栈上，sret 返回时使用了错误的状态值，导致系统崩溃。
- **解决：**修改 kernelvec.S 的上下文保存逻辑，在保存通用寄存器之后，立即读取 sepc 和 sstatus 并保存到内核栈的 trapframe 结构中：

```
# ========== 保存 sepc 和 sstatus ==========
csrr t0, sepc
sd t0, 248(sp)

csrr t1, sstatus
sd t1, 256(sp)
```

图 8 kernelvec.S 的上下文保存逻辑修改部分

7 思考题

7.1 中断设计

7.1.1 为什么时钟中断需要在 M 模式处理后再委托给 S 模式？

这主要是由 RISC-V 的硬件架构规范所决定的。RISC-V 的核心计时器资源物理上属于机器模式的硬件资源，仅能在最高特权级下进行配置，运行在监督模式的操作系统内核无法直接控制这些硬件触发器。此外，从系统架构的角度来看，为了保证特权级隔离与系统安全，底层的时钟硬件通常交由固件统一管理。因此，xv6 及大多数 RISC-V 系统采取了一种折中的“转发”机制：当时钟硬件中断触发时，首先陷入 M 模式的 timervec 进行处理，M 模式在完成硬件复位后，通过向 sip 寄存器写入数据人为注入一个软件中断，从而间接地通知 S 模式内核响应时钟事件。

7.1.2 如何设计一个支持中断优先级的系统？

1. 硬件支持：利用 PLIC 硬件控制器。PLIC 允许为每个中断源配置优先级和阈值，只有优先级高于当前阈值的中断才会被发送给 CPU。

2. 软件实现：当前实验的实现是“关中断处理”，不支持嵌套，即处理一个中断时 sstatus.SIE 为 0。要支持优先级，需要在中断处理函数中手动开启全局中断。在开启前，需读取 scause 确认当前中断优先级，并调整 CPU 的中断屏蔽掩码，仅允许更高优先级的中断触发。这需要精细的栈管理，确保被抢占的中断上下文不会被破坏。

7.2 性能考虑

7.2.1 中断处理的时间开销主要在哪里？如何优化？

主要开销集中在三个方面：

- 上下文切换带来的内存访问延迟，特别是保存和恢复 32 个通用寄存器及 CSR 状态时会产生大量的内存读写，且容易引发 Cache Miss。
- 中断发生瞬间导致的 CPU 流水线冲刷和分支预测失效。
- 特权级切换可能伴随的 TLB 刷新或页表切换开销。

针对这些瓶颈，常见的优化策略包括：利用 ABI 规范仅保存调用者保存寄存器以减少内存访问量，或者依赖硬件支持的“影子寄存器”组实现快速切换。此外，像 xv6 使用的 Trampoline 机制，将中断入口代码映射到所有地址空间的高端，也能有效减少进出内核时的页表切换开销。

7.2.1 高频率中断对系统性能有什么影响？

如果中断频率过高（例如达到微秒级），会对系统性能造成严重打击。最直接的后果是“中断风暴”，即 CPU 将绝大部分时间耗费在保存现场、恢复现场和流水线冲刷等“管理动作”上，而真正用于执行有效逻辑的时间极少，导致系统吞吐量断崖式下跌。在极端情况下，这会引发“活锁”现象，即系统看似在忙碌运行，但用户进程完全得不到调度，对外部请求无响应。此外，频繁进入内核态处理中断还会不断冲刷数据 Cache 和指令 Cache，导致用户态程序的执行效率显著降低。

7.3 可靠性

7.3.1 如何确保中断处理函数的安全性？

1. 独立内核栈：中断处理必须使用内核空间的独立栈，不能复用用户栈，防止用户程序通过破坏栈指针导致内核崩溃。

2. 原子性与锁：中断处理程序中访问共享数据结构时必须使用自旋锁，且在持锁期间必须关闭中断，防止死锁或数据竞争。

3. 状态保护：进入 C 处理函数前，必须先在汇编层将 sepc 和 sstatus 保存到栈中，防止嵌套中断覆盖这些关键状态导致无法返回。

7.3.2 中断处理中的错误应该如何处理？

1. 用户态错误（如 Page Fault, Illegal Instruction）：如果错误源自用户程序，内核应终止该进程，并回收资源，但不应导致 OS 崩溃。

2. 内核态错误：如果内核代码自身在中断处理中触发异常（如空指针引用），

由于无法安全恢复，通常采取 Panic 策略，打印调试信息后挂起系统，以便开发者调试。这也是本次实验采取的策略。

7.4 扩展性

7.4.1 如何支持更多类型的中断源？

采用“注册表”驱动的设计模式。正如实验中实现的 register_interrupt 接口，系统维护一个以中断号为索引的函数指针数组。当需要添加新设备支持时，只需分配一个新的中断号，并调用注册接口将对应的驱动程序 ISR 绑定到该中断号上，而无需修改核心的 trap 分发逻辑。对于外部硬件设备，还需要在 devintr 函数中增加对外部中断的分支处理，通过查询 PLIC 控制器获取具体的设备 ID 来进行二级分发。

7.4.2 如何实现中断的动态路由？

实现中断动态路由主要依赖于软硬件的灵活配置。一方面，可以通过系统调用允许特权进程在运行时修改中断向量表中的函数指针，从而动态改变特定中断的处理逻辑。另一方面，在多核系统中，可以利用高级中断控制器的亲和性设置，将特定高负载设备的中断动态路由到当前负载较轻的 CPU 核心上处理。这种机制能够实现系统的负载均衡，避免单个核心因处理过多中断而成为性能瓶颈。

7.5 实时性

7.5.1 当前实现的中断延迟特征如何？

当前实现的系统属于非实时系统，其中断延迟具有一定的不确定性。首先，由于实验采用“关中断处理”策略，在处理一个中断的整个过程中全局中断是关闭的。如果当前正在执行的中断处理函数逻辑过长，后续的高优先级中断就必须等待，这种阻塞导致了不可预测的延迟抖动。其次，时钟中断采用的“M 模式转发”机制比直接的硬件响应多了一次特权级切换和软件注入的过程，这也在固有的硬件延迟基础上增加了额外的软件开销。

7.5.2 如何设计一个满足实时要求的中断系统？

1. 抢占式内核：允许高优先级任务或中断随时打断低优先级任务（甚至内核态代码）。

2. 中断嵌套：必须支持高优先级中断打断低优先级中断。

3. 确定性延时：严格限制关中断的最长时间，确保最坏情况下的中断响应时间是可预测且有界的。

4. 实时调度器：配合中断系统，使用优先级调度或截止时间调度算法，确保关键任务在 Deadline 前完成。

8 实验总结

通过本次实验，我从零开始构建了一个基于 RISC-V 架构的操作系统中断与异常处理子系统。这不仅是对课堂理论知识的实践验证，更是一次深入操作系统的探索之旅。

在实验初期，RISC-V 复杂的特权级切换机制和众多的 CSR 寄存器（如 stvec, scause, sepc）曾让我感到无从下手。但通过逐行研读 xv6 源码并亲手编写 kernelvec.S，我逐渐理清了硬件自动跳转与软件手动保存上下文之间的协作关系，深刻体会到了汇编代码在处理精细硬件状态时的不可替代性。

实验中最具挑战性也最有成就感的环节，是解决时钟中断的跨特权级驱动问题。面对 RISC-V 硬件定时器只能在 M 模式触发的限制，我没有止步于理论困境，而是通过查阅规范，利用“软件注入”技术成功在 M 模式向 S 模式发送信号，打通了特权级之间的通信壁垒。当控制台上第一次稳定打印出 "System Uptime" 时，我真切感受到了操作系统是如何将底层的硬件脉冲转化为上层的逻辑时间的。

此外，异常处理机制的实现让我对系统的健壮性有了直观认识。通过区分“可恢复异常”和“致命错误”，并分别实现指令跳过和 Panic 保护，我理解了操作系统如何在开放服务的同时自我防御。本次实验让冷冰冰的中断向量表变成了系统中活跃的事件流，标志着我的内核从此具备了感知外部世界和主动响应的能力，也为后续实现进程调度和抢占式多任务系统奠定了最关键的基石。

教师评语评分

评语：_____

_____

_____

_____

_____

_____

_____

_____

评分：_____

评阅人：

年　　月　　日

（备注：对该实验报告给予优点和不足的评价，并给出百分之评分。）