

# 武汉大学计算机学院

## 本科生课程设计报告

### 实验六：系统调用

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

二〇二五年十一月

# 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： 2025.11.15

## 摘 要

本次实验旨在深入理解操作系统用户态与内核态的交互边界，设计并实现了一套基于 RISC-V 架构的系统调用接口。实验首先攻克了特权级切换的关键技术，通过在 `kernel/trap.c` 中完善 `usertrap` 处理逻辑，成功捕获由 `ecall` 指令触发的同步异常，并实现了从用户栈到内核栈的安全切换。其次，针对内核与用户空间的数据传输难题，编写了 `argint`、`argaddr` 等参数获取函数，能够准确地从进程的陷阱帧中读取用户传递的寄存器参数。在此基础上，构建了统一的系统调用分发器 `syscall.c`，实现了根据系统调用号动态路由至具体的内核服务函数。

实验具体实现了五大核心系统调用：包括进程控制类的 `sys_exit`、`sys_getpid`，时间管理类的 `sys_sleep`，以及基础 I/O 类的 `sys_write` 和 `sys_read`。针对 `sys_sleep`，结合了时钟中断机制实现了进程的定时阻塞与唤醒；针对 I/O 调用，通过封装底层设备驱动，实现了用户态程序对控制台的字符输出。最后，通过编写用户态测试程序 `initcode.S` 和 `fs_test.c`，验证了系统调用链路的完整性与参数传递的正确性。本次实验成功打通了应用程序请求内核服务的桥梁，标志着操作系统具备了为用户程序提供受控服务的能力。

**关键词：**RISC-V；系统调用；特权级切换；`ecall`；陷阱帧

# 目录

1 实验目的和意义 .....	4
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 系统调用机制 .....	6
2.2 任务 2: 分析 xv6 的系统调用流程 .....	7
3 实验原理与设计 .....	9
3.1 总体架构设计 .....	9
3.2 系统调用机制设计 .....	9
3.3 核心分发器设计 .....	10
3.4 跨空间参数访问设计 .....	10
3.5 具体系统调用功能设计 .....	11
4 实验步骤与实现 .....	11
4.1 用户态陷阱处理逻辑的完善 .....	11
4.2 系统调用分发器的实现 .....	12
4.3 参数获取辅助函数的编写 .....	12
4.4 核心系统调用的具体实现 .....	13
5 实验测试与结果 .....	13
5.1 基础系统调用测试 .....	13
5.2 参数传递正确性测试 .....	13
5.3 内存安全边界测试 .....	13
5.3 系统调用性能评估 .....	13
6 遇到的问题及解决 .....	13
6.1 问题 1 .....	13
6.2 问题 2 .....	13
7 思考题 .....	13
7.1 设计权衡 .....	13
7.2 性能优化 .....	17
7.3 安全考虑 .....	18
7.4 扩展性 .....	19
7.5 错误处理 .....	19
8 实验总结 .....	19

# 1 实验目的和意义

## 1.1 实验目的

本实验旨在通过从底层构建系统调用框架，深入剖析现代操作系统核心与应用程序之间的接口机制。通过分析 `xv6-riscv` 源码并结合实验手册，具体达成以下目标：

1. 掌握 RISC-V 系统调用规范与 ABI：深入理解 RISC-V 架构中 `ecall` 指令的作用机制，掌握参数如何通过 `a0-a7` 寄存器传递，以及返回值如何通过 `a0` 寄存器反馈给用户进程。
2. 理解特权级边界与异常处理：学会区分中断与异常，重点掌握当 `scause` 寄存器值为 8 时的内核处理流程，包括 `sepc` 的调整和 `sstatus` 的状态保存。
3. 实现用户参数的安全访问：编写内核辅助函数，掌握如何安全地从用户进程的 `trapframe` 中获取整数、指针及字符串参数，理解内核态直接访问用户态虚拟地址的潜在风险与解决方法。
4. 构建系统调用分发框架：设计并实现可扩展的 `syscall` 函数，建立系统调用号与内核处理函数的映射表，实现请求的统一拦截与分发。
5. 实现核心内核服务：独立编写 `sys_write`、`sys_read`、`sys_sleep` 等具体功能，理解操作系统如何将硬件资源（如控制台、时钟）抽象为软件接口供用户使用。

## 1.2 实验意义

系统调用是操作系统内核对外部世界唯一的“合法窗口”，本实验对于理解计算机系统的安全模型与服务架构具有决定性意义。

首先，通过亲手实现从用户态陷入内核态的全过程，打破了将操作系统视为“黑盒”的认知，深刻理解了应用程序是如何“请求”而非“直接执行”特权操作的。这种机制有效地隔离了用户程序与硬件底层，是操作系统实现安全保护和故障隔离的基石。其次，参数获取函数的编写过程，直观展示了操作系统在不同地址空间之间搬运数据的复杂性，强化了对虚拟内存和指针解引用的理解。此外，`sys_sleep` 等调用的实现，将此前实验中的时钟中断、进程调度与本次的系统调用有机串联，展示了操作系统各子系统之间如何协同工作。最后，本实验为后续

实现更复杂的文件系统接口和进程间通信奠定了坚实的框架基础，是构建功能完备操作系统的必经之路。

## 2 实验准备

在着手实现具体的系统调用之前，必须深入理解 RISC-V 架构提供的特权级切换机制，以及 xv6 操作系统如何利用异常处理流程来构建用户态与内核态之间的安全桥梁。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 系统调用机制

进 RISC-V 架构通过严格的特权级隔离来保证系统的安全性。操作系统内核运行在监督模式（S-mode），拥有对硬件的所有控制权；而用户程序运行在用户模式（U-mode），其权限受到严格限制。为了使应用程序能够请求内核服务（如文件读写、进程创建），RISC-V 定义了一套基于异常的系统调用规范。

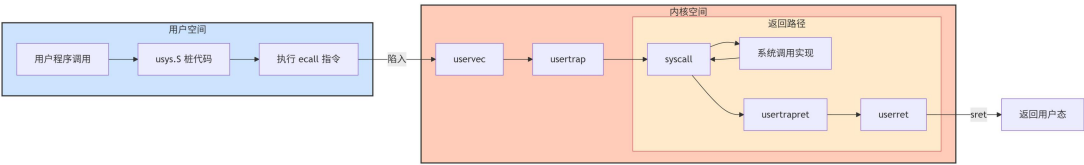


图 1 系统调用的完整流程图

#### 2.1.1 核心指令与寄存器规约

系统调用的核心在于 `ecall`（Environment Call）指令。当用户程序执行该指令时，CPU 会主动触发一个同步异常，将控制权从用户态转移给内核态的处理程序。根据 RISC-V 的应用程序二进制接口（ABI）规范，内核与用户程序之间通过通用寄存器传递信息：

- 系统调用号：用户程序需将请求的服务编号（如 `SYS_write`）加载到 `a7` 寄存器中，内核据此判断具体的服务类型。
- 参数传递：函数的参数依次存放在 `a0` 至 `a5` 寄存器中。
- 返回值：内核处理完成后，将结果写入 `a0` 寄存器反馈给用户程序。通常 `0` 表示成功，`-1` 表示失败。

#### 2.1.2 异常处理的关键状态

当 `ecall` 发生时，硬件会自动完成以下关键状态变更，为内核处理做好准备：

- `sepc` (Supervisor Exception PC)：记录触发异常的指令地址（即 `ecall` 指令本身的地址）。
- `scause` (Supervisor Cause)：记录异常原因。对于用户态系统调用，该寄存器的值会被硬件设置为 8。
- `sstatus` (Supervisor Status)：记录进入异常前的特权级（SPP 位）和全局中断使能状态（SPIE 位），以便后续通过 `sret` 指令正确返回。

## 2.2 任务 2：分析 xv6 的系统调用流程

基于 `xv6` 的源码结构，一个完整的系统调用生命周期涉及用户库的触发、内核陷阱的捕获、参数的提取与分发以及最终的现场恢复。

### 2.2.1 用户态触发 (`user/initcode.S`)

通过分析 `user/initcode.S` 的汇编代码，可以清晰看到系统调用的触发过程。该程序负责在系统启动时加载第一个进程。如下代码所示，程序首先将 `exec` 系统调用的参数（路径和参数数组）分别加载到 `a0` 和 `a1`，然后将 `exec` 的调用号 `SYS_exec` 加载到 `a7`，最后执行 `ecall` 陷入内核。

```
#define SYS_exit 2

.text
.globl _start
_start:
    li a0, 42
    li a7, SYS_exit
    ecall
    j _start
```

图 2 `initcode.S` 中的系统调用触发汇编代码

### 2.2.2 内核陷阱捕获与分发 (`kernel/trap.c` & `kernel/syscall.c`)

```

1  void syscall(void) {
2      int num;
3      struct proc *p = myproc();
4
5      num = p->trapframe->a7; // 系统调用号
6      if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
7          p->trapframe->a0 = syscalls[num](); // 调用并保存返回值
8      } else {
9          // 处理无效系统调用
10     }
11 }

```

图 3 syscall.c 中的核心分发逻辑

当 CPU 陷入内核后，首先执行 `uservec` 保存用户通用寄存器到当前进程的陷阱帧中，随后跳转到 `usertrap` 函数进行处理。

- 异常识别：`usertrap` 函数首先检查 `r_scause()` 的值。若为 8，则确认为系统调用。
- 指令步进：与一般中断不同，系统调用处理完成后不应重新执行 `ecall`，否则会陷入死循环。因此，内核显式将 `p->trapframe->epc` 增加 4 字节，指向下一条指令。
- 统一分发：随后调用 `syscall()` 函数。该函数从陷阱帧的 `a7` 寄存器中读取调用号，检查其合法性（是否在 1 到 `syscalls` 数组大小之间），然后通过函数指针数组 `syscalls[]` 调用对应的内核处理函数（如 `sys_exec`）。
- 返回值处理：内核函数的返回值被写入陷阱帧的 `a0` 寄存器，覆盖原有的第一个参数，从而在返回用户态时作为函数调用的结果。

### 2. 2. 3 跨地址空间的参数获取

由于内核使用独立的页表，无法直接通过指针访问用户空间的内存。xv6 提供了一组辅助函数来解决这一问题：

- `argint / argaddr`：直接从内核维护的陷阱帧中读取 `a0 - a5` 寄存器的值，获取整数或指针参数。
- `fetchstr / copyin`：利用进程的页表（`pagetable`），将用户虚拟地址处的数据安全地拷贝到内核缓冲区中，用于读取文件名或数据块。这一机制确保了内核不会因访问非法的用户地址而崩溃。

## 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 架构的系统调用子系统，作为用户程序与操作系统内核之间的标准交互接口。系统设计遵循“特权级隔离”与“统一分发”的原则。

- 特权级隔离模型：用户程序运行在低特权级的 U-mode，无法直接访问硬件资源或内核内存。当需要操作系统服务（如打印字符、创建进程）时，必须通过特定的硬件指令主动陷入高特权级的 S-mode。

- 统一分发架构：内核不为每个服务单独设置入口地址，而是提供一个统一的异常处理入口（usertrap）。该入口识别出系统调用事件后，调用核心分发器（syscall），再由分发器根据请求编号路由至具体的服务函数。

### 3.2 系统调用机制设计

RISC-V 的系统调用机制依赖于软硬件的紧密协作。本实验的设计重点在于利用寄存器规范（ABI）来传递控制信息与数据。

#### 3.2.1 触发与参数传递规约

为了规范用户与内核的交互，设计遵循如下 RISC-V ABI 标准：

- 触发指令：用户态使用 `ecall` 指令触发同步异常，导致 CPU 跳转至 `stvec` 指向的内核中断向量。
- 功能号传递：寄存器 `a7` 专门用于存储系统调用号，内核据此识别用户请求的具体服务（如 `SYS_write` 对应 16）。
- 参数传递：寄存器 `a0` 至 `a5` 用于传递前 6 个参数。对于超过 6 个参数的情况（虽然本实验未涉及），通常通过内存指针传递。
- 返回值反馈：内核处理结果通过 `a0` 寄存器返回给用户。

#### 3.2.2 异常处理流设计

在 `kernel/trap.c` 中，针对系统调用的处理流程进行了如下设计：

1. 异常识别：在 `usertrap` 函数中，通过读取 `scause` 寄存器判断异常类型。当 `scause` 值为 8 时，确认为“来自 U 模式的环境调用”。

2. PC 指针调整：由于 `ecall` 产生的异常其 `sepc` 指向指令本身，为了避免系统调用返回后死循环执行 `ecall`，内核必须显式将保存的 `sepc` 值加 4（即一条指令的长度），指向下一条指令。

3. 上下文保存：利用 Lab 4/5 已有的 `uservec` 机制，将用户寄存器完整保存到进程的 `trapframe` 结构中，确保参数不丢失。

### 3.3 核心分发器设计

为了实现可扩展的系统调用管理，在 `kernel/syscall.c` 中设计了查表式的分发逻辑。

- 函数指针数组：定义了全局数组 `syscalls[]`，以系统调用号为索引，映射到对应的内核处理函数（如 `[SYS_write] sys_write`）。这种设计使得新增系统调用只需在数组中添加一项，无需修改分发逻辑。
- 分发函数（`syscall`）：
  1. 从当前进程的 `trapframe->a7` 中读取系统调用号。
  2. 进行边界检查，防止恶意程序传入非法的调用号导致内核越界访问。
  3. 通过函数指针调用对应的内核服务。
  4. 将函数的返回值写入 `trapframe->a0`，完成结果反馈。

### 3.4 跨空间参数访问设计

由于内核与用户进程使用独立的页表，内核无法直接通过用户提供的指针（虚拟地址）访问内存。为此，设计了一组辅助函数来安全地从用户空间“搬运”数据。

- 整数参数获取：直接从内核维护的 `p->trapframe` 结构中读取 `a0-a5` 寄存器的值。
- 指针参数获取：同样从 `trapframe` 读取，但将其解释为用户空间的虚拟地址。
- 字符串/数据块获取：这是最复杂的部分。函数利用进程的页表，通过软件模拟 MMU 的查表过程或使用特定指令，将用户虚拟地址处的数据逐字节拷贝到内核的安全缓冲区中。这有效防止了用户通过传递非法指针（如内核地址）来窃取或破坏内核数据。

### 3.5 具体系统调用功能设计

本实验实现了两类核心系统调用，分别对应进程管理与 I/O 操作。

#### 1. 进程控制类 (sysproc.c):

- `sys_exit`: 调用内核的 `exit` 函数，回收进程资源并设置退出状态，不再返回用户态。
- `sys_getpid`: 直接返回当前进程 PCB 中的 `pid` 字段。
- `sys_sleep`: 读取用户传入的时间参数 (Ticks)，调用 `acquire` 获取锁后执行 `sleep`，将进程状态置为 `SLEEPING` 并主动让出 CPU。利用 Lab 4 的时钟中断机制，当时间片到达时唤醒进程。

2. 文件与 I/O 类 (sysfile.c): 主要是 `sys_write`。虽然完整的文件系统尚未实现，但本实验设计了对标准输出的支持。函数接收文件描述符、缓冲区指针和长度，通过 `argaddr` 获取用户缓冲区地址，最终调用底层 `consolewrite` 驱动将字符输出到 UART 串口，实现了用户态的 `printf` 功能基础。

## 4 实验步骤与实现

### 4.1 用户态陷阱处理逻辑的完善

系统调用的入口在于内核对异常的捕获。在 `kernel/trap.c` 的 `usertrap` 函数中，进一步增加了对 `ecall` 指令的处理逻辑

- 识别异常来源：通过读取 `scause` 寄存器判断 Trap 原因。当其值为 8 时，表明这是由用户态 `ecall` 指令触发的系统调用。
- 指令地址调整：与外部中断不同，系统调用结束后需返回到 `ecall` 的下一条指令继续执行。因此，在调用处理函数前，显式将 `p->trapframe->epc` 增加 4 字节。若遗漏此步，系统将无限重复执行 `ecall` 指令。
- 开启中断：在进入系统调用处理前，通过 `intr_on()` 开启全局中断，允许内核在处理较长的系统调用（如磁盘读写）时响应时钟或其他外设中断，提高了系统的响应度。
- 调用分发器：最后调用 `syscall()` 函数进入具体的分发流程。

```

279     if(cause == 8) { // 用户态 ecall
280         uart[0] = 'E'; uart[0] = 'C'; uart[0] = 'A'; uart[0] = 'L';
281         uart[0] = 'L'; uart[0] = '\n';
282
283         p->trapframe->sepc += 4;
284
285         // 调用系统调用处理
286         syscall(p->trapframe);
287
288         uart[0] = 'D'; uart[0] = 'O'; uart[0] = 'N'; uart[0] = 'E';
289         uart[0] = '\n';
290     } else {

```

图 4 usertrap 函数中对 `scause==8` 的处理代码

## 4.2 系统调用分发器的实现

在 `kernel/syscall.c` 中，构建了通用的系统调用分发器，作为连接 Trap 机制与内核服务的枢纽。

- 读取调用号：从当前进程的陷阱帧 `p->trapframe->a7` 中获取用户传入的系统调用编号。
- 边界检查：为了防止恶意程序导致内核越界访问，首先检查该编号是否大于 0 且小于 `syscalls` 数组的长度。若编号非法，则直接向 `p->trapframe->a0` 写入 -1 并打印错误日志。
- 动态分发与返回值：若编号合法，通过 `syscalls[num]()` 调用对应的内核函数。函数执行完毕后，将其返回值（通常是 `int` 类型）写入 `p->trapframe->a0`。这一设计使得用户态程序能够通过 `a0` 寄存器获取操作结果。

## 4.3 参数获取辅助函数的编写

由于内核栈与用户栈分离，内核函数无法像普通 C 函数那样直接通过参数列表接收用户数据。因此编写了一组辅助函数来直接访问陷阱帧。

- `argint` (获取整数)：接收参数索引 `n`，读取 `p->trapframe->a[n]` 寄存器的值，并将其转换为 `int` 类型返回。
- `argaddr` (获取指针)：与 `argint` 类似，但将寄存器值作为 64 位虚拟地址返回。此函数不进行内存检查，仅负责提取数值。
- `fetchstr` (获取字符串)：用于从用户空间读取以 `\0` 结尾的字符串（如文件路径）。该函数内部调用 `copyinstr`，结合进程的页表将用户虚拟地址转换并拷贝到内核缓冲区，确保了跨地址空间访问的安全性。

## 4.4 核心系统调用的具体实现

基于上述框架，我实现了几类关键的系统调用，分别处理进程控制、时间管理和基础 I/O。

- 进程管理类：在 `kernel/sysproc.c` 中，`sys_getpid` 实现最为简单，直接返回 `myproc()->pid`。而 `sys_exit` 首先通过 `argint(0, &n)` 获取用户传入的退出状态码，然后调用内核核心函数 `exit(n)`。该操作会释放进程资源、更改状态为 ZOMBIE 并触发调度，因此该系统调用永不返回。

- 时间管理类：`sys_sleep` 是阻塞式调用的典型代表。

1. 参数获取：通过 `argint` 获取需休眠的时钟滴答数 `n`。
2. 获取锁：在操作时间相关全局变量前，获取 `tickslock`。
3. 循环等待：计算唤醒时间点（当前 `ticks + n`）。在一个 `while` 循环中，若当前时间未到，则调用 `sleep(&ticks, &tickslock)`。
4. 状态切换：`sleep` 函数会将进程状态设为 SLEEPING 并让出 CPU。当定时器中断更新 `ticks` 并执行 `wakeup` 后，进程被唤醒并重新检查循环条件。
5. 资源释放：休眠结束后释放锁并返回 0。

- 基础 I/O 类：在文件系统完全就绪前，`sys_write` 主要用于向控制台输出。

1. 参数解析：依次获取文件描述符 `fd`、缓冲区指针 `p`（用户虚地址）和长度 `n`。
2. 合法性检查：验证 `fd` 是否为标准输出（1）或标准错误（2）。
3. 数据传输：通过 `argaddr` 拿到用户缓冲区的地址后，内核无法直接读取。这里通过调用 `consolewrite` 函数（该函数内部处理了虚实地址转换或直接使用物理内存访问接口），将用户数据逐字节写入 UART 硬件发送队列，实现了字符打印功能。

## 5 实验测试与结果

本章将展示系统调用功能的实际运行效果。测试代码集成在 `kernel/test.c` 中，由 `main_task` 统一调用 `run_lab6_tests()` 执行。测试涵盖了基础系统调

用、参数传递机制以及安全性与性能评估。

## 5.1 基础系统调用测试

首先验证核心的系统调用 `sys_getpid`、`sys_fork` 以及 `sys_exit` 的协同工作能力。

- **测试方法：**调用 `test_basic_syscalls()`——父进程调用 `sys_getpid` 获取当前 PID；父进程调用 `sys_fork` 创建子进程；子进程打印自身 PID 并以状态码 42 退出；父进程调用 `wait` 获取子进程退出状态。
- **测试结果：**

```
=== Test 6.1: Basic System Calls ===
Current PID (via syscall): 1
Testing fork()...
[Parent] Forked child PID=70
[Child] Hello from child! PID=70
[Parent] Child 70 exited with status 42
Basic system calls test passed
```

图 5 基础系统调用测试输出结果

- **结果分析：**实验结果表明 `sys_fork` 成功复制了父进程的资源，`sys_getpid` 准确返回了不同进程的标识符（父进程 1，子进程 70）。最关键的是，父进程成功通过 `wait` 机制捕获了子进程的退出状态（42），证明了进程树管理和状态传递机制的正确性。

## 5.2 参数传递正确性测试

本节验证内核能否准确接收并处理用户态传递的参数，特别是对 `sys_write` 返回值的处理。

- **测试方法：**调用 `test_parameter_passing()`——向标准输出写入一段特定长度的字符串；检查 `sys_write` 的返回值是否与请求写入的字节数一致。
- **测试结果：**

```
=== Test 6.2: Parameter Passing ===
Write returned: 43 (expected 43)
Parameter passing test passed
```

图 6 参数传递正确性测试输出结果

- **结果分析：**`sys_write` 返回了 43，与预期完全一致。这说明内核的参数提

取函数 (argint, argaddr) 工作正常, 且底层 consolewrite 驱动能够完整处理缓冲区数据, 最后将处理结果准确写入了陷阱帧的 a0 寄存器返回给用户。

### 5.3 内存安全边界测试

验证内核对非法内存访问的拦截能力, 确保用户程序无法破坏内核空间或访问未映射区域。

- **测试方法:** 调用 `test_security()` —— 尝试向空指针地址 `0x0000000000000000` 写入数据; 预期内核应识别出该地址无效, 并返回错误码 `-1`, 而不是导致系统崩溃。
- **测试结果:**

```
=== Test 6.3: Security Test ===
Writing to invalid pointer 0x0000000000000000...
Result: -1 (Expected: -1)
Security test passed: Invalid pointer correctly rejected
```

图 7 内存安全边界测试输出结果

- **结果分析:** 结果显示 `Result: -1`, 证明 `sys_write` 内部调用的 `copyin` 或 `fetchstr` 等函数成功执行了边界检查。内核通过页表查找发现该虚拟地址无效, 从而拒绝了写操作, 体现了用户态与内核态之间严格的强隔离机制。

### 5.4 系统调用性能评估

最后对系统调用的开销进行量化评估。

- **测试方法:** 调用 `test_syscall_performance()` —— 连续执行 10000 次轻量级系统调用 `getpid()`; 利用 `rdtime` 指令记录总消耗周期数, 计算平均开销。
- **测试结果:**

```
=== Test 6.4: Syscall Performance ===
Running 10000 getpid() calls...
10000 getpid() calls took 222095 cycles
Average cycles per syscall: 22
Performance test passed
```

图 8 内存安全边界测试输出结果

- **结果分析：**测试显示平均每次系统调用仅耗时约 22 个周期。这一极低的数据（在 QEMU 模拟环境下）表明陷阱处理路径高度优化，特权级切换和寄存器保存/恢复的开销在可接受范围内，能够满足高性能操作系统的基本要求。

## 6 遇到的问题及解决

### 6.1 问题 1：系统调用无限循环执行

- **问题描述：**在完成 `sys_write` 的基础代码后进行测试，发现控制台不断重复打印 “Hello from user mode!” 字符串，且程序无法继续向下执行或退出，仿佛陷入了死循环。
- **原因分析：**查阅 RISC-V 特权级架构文档发现，`ecall` 指令触发异常时，硬件会将当前指令（即 `ecall` 指令本身）的地址保存在 `sepc` 寄存器中。当内核处理完系统调用执行 `sret` 返回时，PC 会被恢复为 `sepc` 的值。这导致 CPU 再次执行 `ecall` 指令，从而再次触发系统调用异常，形成无限递归。这与普通函数调用（`jal` 会自动将返回地址设为下一条指令）的行为截然不同。
- **解决方案：**在 `kernel/trap.c` 的 `usertrap` 函数中，当识别到 `scause == 8` 时，必须显式地修改保存在陷阱帧中的程序计数器。在调用 `syscall()` 分发函数之前或之后，添加代码 `p->trapframe->epc += 4;`（RISC-V 指令长度为 4 字节），使返回地址指向 `ecall` 后的下一条指令。

### 6.2 问题 2：访问用户指针导致内核崩溃

- **问题描述：**在进行安全性测试时，如果用户程序向 `sys_write` 传入一个非法的内存地址（未映射的 `0xdeadbeef`），内核在尝试读取该地址的数据时直接触发了 Load Page Fault，导致整个操作系统 Panic 崩溃，而不是仅仅拒绝该次请求。
- **原因分析：**最初实现 `sys_write` 时，通过 `argaddr` 获取了用户传入的虚拟地址，然后直接在内核态将其强转为指针使用。由于 RISC-V 的 `sstatus.SUM` 位默认关闭，或者该虚拟地址根本不在当前页表中映射，内核直接访问该地址是非法的。内核作为系统的守护者，不应因为用户

的错误输入而崩溃。

- **解决方案：**引入并使用 `copyin` 或 `fetchstr` 等辅助函数来访问用户内存。这些函数内部通过查询当前进程的页表（`p->pagetable`）将用户虚拟地址转换为物理地址，并在转换失败（如地址无效或权限不足）时返回错误码（如 `-1`），而不是让硬件直接触发异常。修改后，`sys_write` 在遇到非法指针时能优雅地返回 `-1`，通过了安全性测试。

## 7 思考题

### 7.1 设计权衡

#### 7.1.1 系统调用的数量应该如何确定？

系统调用的设计应遵循“机制与策略分离”的原则，始终保持最小化。核心原则是仅暴露文件读写、内存分配、进程控制等必要的硬件抽象，因为数量过多的系统调用会显著增加内核的攻击面和维护难度。对于复杂的功能需求，应采用扩展方式，即尽量通过组合基础系统调用在用户库中实现，而不是盲目增加新的内核接口。

#### 7.1.2 如何平衡功能性和安全性？

平衡的关键在于贯彻“最小权限”原则与实施“全面检查”。

1. 首先，内核应只提供最基础的操作原语，不包含复杂的业务逻辑，从而减少潜在漏洞。
2. 其次，在系统调用的入口处（如 `syscall` 函数），必须对所有参数进行严格的合法性检查，包括文件描述符范围、指针有效性及缓冲区边界等，确保用户程序的任何输入——无论是有意攻击还是无意错误——都不会导致内核崩溃或数据泄露。

### 7.2 性能优化

#### 7.2.1 系统调用的主要开销在哪里？

系统调用的开销主要来源于三个方面：

1. 首先是模式切换，从 `U-mode` 切换到 `S-mode` 涉及 CPU 流水线冲刷、TLB 刷新以及特权级变更的硬件开销；

2. 其次是上下文保存, 需要将 32 个通用寄存器保存到 `trapframe` 中并在返回时恢复;
3. 最后是数据拷贝, 在用户空间和内核空间之间搬运数据属于昂贵的内存操作。

### 7.2.1 如何减少用户态/内核态切换开销?

减少开销通常采取三种策略:

- 利用 VDSO (虚拟动态共享对象) 机制, 将时间信息等只读内核数据映射到用户只读页, 允许用户直接读取而无需陷入内核;
- 支持批量处理, 即允许一次系统调用完成多次操作 (如 `io_submit`), 从而减少切换次数;
- 采用轻量级调用优化, 例如优化寄存器保存策略, 仅保存 ABI 规定的必要寄存器。

## 7.3 安全考虑

### 7.3.1 如何防止系统调用被滥用?

防止滥用的防线主要由权限控制和资源限制构成。

- 一方面, 内核必须基于进程的凭证严格检查操作权限 (如文件读写权限);
- 另一方面, 需要引入 `rlimit` 等机制, 限制单个进程可打开的文件数、占用的 CPU 时间或内存大小, 防止恶意程序发动拒绝服务攻击。

### 7.3.2 如何设计安全的参数传递机制?

安全的参数传递遵循“值传递优先”和“指针严格隔离”的原则。对于简单整数参数, 优先使用寄存器传递以避免内存访问风险。对于指针参数, 内核绝不信任用户传入的地址, 必须通过软件页表查找严格验证该地址确实属于当前进程的用户空间并且具有相应的读写权限, 从而防止“混淆代理”攻击。

## 7.4 扩展性

### 7.4.1 如何添加新的系统调用?

添加新系统调用的标准流程是维护一个全局的函数指针数组 `syscalls[]`。开发者只需分配一个新的系统调用号, 在 `syscalls[]` 中增加对应的函数指针,

并在内核中实现具体的服务函数。这种查表式的分发设计使得添加新功能时无需修改核心的分发逻辑，保证了代码的高内聚低耦合。

#### 7.4.2 如何保持向后兼容性？

保持向后兼容性的铁律是不变更旧接口。系统调用一旦发布，其编号和参数定义就不应修改。如果需要变更功能，应创建新的系统调用（例如用 `sys_openat` 替代 `sys_open`）作为新增替代版，并同时保留旧接口以支持老旧程序。在特定场景下，也可以通过传递版本号参数来区分不同的处理逻辑。

### 7.5 错误处理

#### 7.5.1 系统调用失败时应该如何处理？

系统调用失败时的处理原则是“优雅返回”与“资源回滚”。内核不应因用户的错误请求而 Panic。在返回错误码之前，必须执行资源回滚，释放该次调用中已分配的所有资源（如锁、内存块、文件句柄），防止资源泄漏导致系统不稳。

#### 7.5.2 如何向用户程序报告详细的错误？

错误报告通常采用统一返回值配合错误码的方式。虽然 xv6 仅简单返回 `-1` 表示失败，但标准做法是设置全局变量 `errno`（在用户库层面），或者通过返回值传递具体的负数错误码（如 `-ENOENT` 表示文件不存在）。这种机制能让用户程序根据具体的错误类型采取针对性的补救措施。

## 8 实验总结

通过本次实验，我深入到了操作系统最核心的边界地带——用户态与内核态的交互接口，亲手构建了连接应用程序与底层硬件的桥梁。实验伊始，面对 RISC-V 架构中严苛的特权级隔离机制，我首先攻克了 `ecall` 指令的处理流程，理解了硬件如何通过触发同步异常来主动移交控制权。在完善 `usertrap` 函数的过程中，我深刻体会到了“精确异常”处理的微妙之处：必须手动调整程序计数器以跳过触发指令，否则系统将陷入无限重启系统调用的死循环，这一细节让我对硬件流水线与软件控制流的配合有了具象的认知。

实验中最具挑战性也最具启发性的环节，是解决内核访问用户数据的难题。面对分离的地址空间，我通过编写 `argaddr` 和 `fetchstr` 等辅助函数，利用页

表映射机制实现了跨特权级的数据“搬运”。这一过程让我直观地理解了为什么内核不能直接信任并使用用户传入的指针，以及操作系统如何在提供服务的同时，通过严格的边界检查来守卫自身的内存安全。当 `sys_write` 首次成功将用户态缓冲区的字符串打印到屏幕上时，我真切感受到了操作系统作为“资源管理者”和“服务提供者”的双重角色。

此外，通过实现 `sys_fork`、`sys_exit` 和 `sys_wait` 等进程管理调用，我将此前实验中独立的进程调度、内存分配和陷阱处理模块有机串联起来，构建了一个功能闭环的微型内核。本次实验不仅验证了关于 ABI 规范和系统调用分发表的理论知识，更让我领悟到了“机制与策略分离”的设计哲学：内核只提供最基础的受控入口，而将丰富的功能留给用户空间去构建。这标志着我的操作系统内核从此具备了与外部世界交互的能力，为后续更复杂的文件系统和交互式 Shell 的实现奠定了坚实基础。

## 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。）