实验6：系统调用

实验目标

通过分析xv6的系统调用机制，深入理解用户态与内核态的交互方式，实现完整的系统调用框架和常用系统调用功能。

核心学习资料

系统调用理论基础

- **操作系统概念 第2章**：操作系统结构
- **RISC-V特权级规范 第12.1节**：Supervisor Trap Handling
- **xv6手册 第2.5节和第4章**：系统调用和陷阱

xv6系统调用源码分析

- `kernel/syscall.c` - 系统调用分发机制
  - 重点函数：`syscall()`，`argint()`，`argstr()`，`argaddr()`
  - 学习要点：参数传递、返回值处理、错误检查
- `kernel/sysproc.c` - 进程相关系统调用实现
- `kernel/sysfile.c` - 文件相关系统调用实现
- `user/usys.pl` - 用户态系统调用桩代码生成
- `kernel/trampoline.S` - 用户态/内核态切换

RISC-V系统调用约定

- **调用约定**：ecall指令、寄存器使用、参数传递
- **特权级切换**：用户模式到监督模式的转换过程

任务列表

任务1：理解系统调用的实现原理

学习重点：

1. 分析系统调用的完整流程：

```
用户程序调用 → usys.S桩代码 → ecall指令 →
uservec → usertrap → syscall → 系统调用实现 →
返回用户态
```

- 每个环节的作用是什么？
- 参数是如何传递的？

- 返回值如何返回？

2. 研究RISC-V的ecall机制：
   - ecall指令的作用
   - scause寄存器中系统调用的编码
   - sepc寄存器的作用和更新
3. 理解特权级切换：
   - 用户栈到内核栈的转换
   - 寄存器状态的保存和恢复
   - 页表的切换时机

深入思考：

- 为什么需要陷阱帧(trapframe)？
- 系统调用和中断处理有什么相同和不同？

任务2：分析xv6的系统调用分发机制

代码阅读指导：

1. 研读 `syscall.c` 中的核心分发逻辑：

```
void syscall(void) {
  int num;
  struct proc *p = myproc();

  num = p->trapframe->a7;  // 系统调用号
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    p->trapframe->a0 = syscalls[num]();  // 调用并保存返回值
  } else {
    // 处理无效系统调用
  }
}
```

- 系统调用号是如何传递的？
- 返回值存储在哪里？
- 错误处理机制是什么？

2. 分析参数提取函数：

```
int argint(int n, int *ip);      // 获取整数参数
int argaddr(int n, uint64 *ip);  // 获取地址参数
int argstr(int n, char *buf, int max);  // 获取字符串参数
```

- 参数是从哪里提取的？
- 如何处理不同类型的参数？
- 边界检查是如何实现的？

3. 理解用户内存访问：

- `copyout()` 和 `copyin()` 的作用
- 为什么不能直接访问用户内存？
- 如何防止用户传递恶意指针？

任务3：设计你的系统调用框架

设计要求：

1. 定义系统调用表结构
2. 设计参数传递机制
3. 实现错误处理策略

核心组件设计：

```
// 系统调用描述符
struct syscall_desc {
    int (*func)(void);          // 实现函数
    char *name;                 // 系统调用名称
    int arg_count;              // 参数个数
    // 可选：参数类型描述
};

// 系统调用表
extern struct syscall_desc syscall_table[];

// 系统调用分发器
void syscall_dispatch(void);

// 参数提取辅助函数
int get_syscall_arg(int n, long *arg);
int get_user_string(const char __user *str, char *buf, int max);
int get_user_buffer(const void __user *ptr, void *buf, int size);

// 你需要考虑的问题：
// 1. 如何验证用户提供的指针？
// 2. 如何处理系统调用失败？
// 3. 如何支持可变参数的系统调用？
// 4. 如何实现系统调用的权限检查？
```

任务4：实现基础系统调用

必需实现的系统调用：

1. **进程控制类：**

```
   int sys_fork(void);        // 创建子进程
   int sys_exit(void);        // 终止进程
   int sys_wait(void);        // 等待子进程
   int sys_kill(void);        // 发送信号
   int sys_getpid(void);      // 获取进程ID
```

2. 文件操作类：

```
    int sys_open(void);        // 打开文件
    int sys_close(void);       // 关闭文件
    int sys_read(void);        // 读文件
    int sys_write(void);       // 写文件
```

3. 内存管理类：

```
    void* sys_sbrk(void);      // 调整堆大小
    // 可选：mmap, munmap等
```

实现策略：

```
// 以sys_write为例
int sys_write(void) {
    int fd;
    char *buf;
    int count;

    // 1. 提取参数
    if (argint(0, &fd) < 0 ||
        argaddr(1, (uint64*)&buf) < 0 ||
        argint(2, &count) < 0) {
        return -1;
    }

    // 2. 参数有效性检查
    if (fd < 0 || fd >= NOFILE || count < 0) {
        return -1;
    }

    // 3. 调用内核函数实现
    return filewrite(myproc()->ofile[fd], buf, count);
}
```

任务5：实现用户态系统调用接口

参考xv6的usys.pl，理解：

1. 桩代码生成机制：

```
    # 每个系统调用的桩代码格式
    .global write
    write:
     li a7, SYS_write     # 系统调用号加载到a7
     ecall                # 陷入内核
     ret                  # 返回
```

2. 用户库函数设计：

```
    // 用户库中的系统调用声明
```

```
int fork(void);
int exit(int) __attribute__((noreturn));
int wait(int*);
int pipe(int*);
int write(int, const void*, int);
int read(int, void*, int);
// ...
```

实现考虑：

- 如何处理系统调用的错误返回？
- 是否需要errno机制？
- 如何提供用户友好的接口？

任务6：系统调用安全性

安全检查要点：

1. **指针验证：**

```
// 检查用户指针是否有效
int check_user_ptr(const void *ptr, int size) {
    // 1. 指针是否在用户地址空间？
    // 2. 内存区域是否有相应权限？
    // 3. 是否会越界访问？
}
```

2. **缓冲区保护：**
   - 防止缓冲区溢出
   - 检查字符串是否正确终止
   - 限制数据传输大小
3. **权限检查：**
   - 文件访问权限
   - 进程操作权限
   - 资源使用限制
4. **竞态条件防护：**
   - TOCTTOU攻击防护
   - 原子操作保证
   - 锁的正确使用

测试与调试策略

基础功能测试

```
void test_basic_syscalls(void) {
    printf("Testing basic system calls...\n");
```

```
    // 测试getpid
    int pid = getpid();
    printf("Current PID: %d\n", pid);

    // 测试fork
    int child_pid = fork();
    if (child_pid == 0) {
        // 子进程
        printf("Child process: PID=%d\n", getpid());
        exit(42);
    }
    else if (child_pid > 0) {
        // 父进程
        int status;
        wait(&status);
        printf("Child exited with status: %d\n", status);
    }
    else {
        printf("Fork failed!\n");
    }
}
```

参数传递测试

```
void test_parameter_passing(void) {
    // 测试不同类型参数的传递
    char buffer[] = "Hello, World!";
    int fd = open("/dev/console", O_RDWR);

    if (fd >= 0) {
        int bytes_written = write(fd, buffer, strlen(buffer));
        printf("Wrote %d bytes\n", bytes_written);
        close(fd);
    }

    // 测试边界情况
    write(-1, buffer, 10);      // 无效文件描述符
    write(fd, NULL, 10);        // 空指针
    write(fd, buffer, -1);      // 负数长度
}
```

安全性测试

```
void test_security(void) {
    // 测试无效指针访问
    char *invalid_ptr = (char*)0x1000000;  // 可能无效的地址
    int result = write(1, invalid_ptr, 10);
```

```
    printf("Invalid pointer write result: %d\n", result);

    // 测试缓冲区边界
    char small_buf[4];
    result = read(0, small_buf, 1000);  // 尝试读取超过缓冲区大小

    // 测试权限检查
    // ...
}
```

性能测试

```
void test_syscall_performance(void) {
    uint64 start_time = get_time();

    // 大量系统调用测试
    for (int i = 0; i < 10000; i++) {
        getpid();  // 简单的系统调用
    }

    uint64 end_time = get_time();
    printf("10000 getpid() calls took %lu cycles\n",
           end_time - start_time);
}
```

调试建议

系统调用跟踪

```
// 在syscall.c中添加调试信息
void syscall(void) {
    int num;
    struct proc *p = myproc();

    num = p->trapframe->a7;

    // 调试输出
    if (debug_syscalls) {
        printf("PID %d: syscall %d (%s)\n",
               p->pid, num, syscall_names[num]);
    }

    // 原有逻辑...
}
```

参数检查调试

- 在参数提取函数中添加验证日志

- 跟踪用户内存访问
- 记录异常的参数值

性能分析

- 测量系统调用延迟
- 分析频繁调用的系统调用
- 识别性能瓶颈

思考题

1. **设计权衡**：
   - 系统调用的数量应该如何确定？
   - 如何平衡功能性和安全性？
2. **性能优化**：
   - 系统调用的主要开销在哪里？
   - 如何减少用户态/内核态切换开销？
3. **安全考虑**：
   - 如何防止系统调用被滥用？
   - 如何设计安全的参数传递机制？
4. **扩展性**：
   - 如何添加新的系统调用？
   - 如何保持向后兼容性？
5. **错误处理**：
   - 系统调用失败时应该如何处理？
   - 如何向用户程序报告详细的错误信息？

实验7：文件系统

实验目标

通过深入分析xv6的简化文件系统，理解现代文件系统的核心概念和实现原理，独立实现一个功能完整的日志文件系统。

核心学习资料

文件系统理论基础

- **操作系统概念 第13-14章**：文件系统接口和实现
- **xv6手册 第10章**：文件系统